DR.G.U.POPE
COLLEGE OF ENGINEERING

Name of the subject: Algoritham

subject code : CS3401

Ragulation :2021

Mr.danial raj AP/CSE ,
DR.G.U.Pope college of engineering,
sawyerpuram,

UNIT 1

Time and Space Complexity

Time complexity is a measure of how long an algorithm takes to run as a function of the size of the input. It is typically expressed using big O notation, which describes the upper bound on the growth of the time required by the algorithm. For example, an algorithm with a time complexity of O(n) takes longer to run as the input size (n) increases.

There are different types of time complexities:

O(1) or constant time: the algorithm takes the same amount of time to run regardless of the size of the input.

O(log n) or logarithmic time: the algorithm's running time increases logarithmically with the size of the input.

O(n) or linear time: the algorithm's running time increases linearly with the size of the input.

O(n log n) or linear logarithmic time: the algorithm's running time increases linearly with the size of the input and logarithmically with the size of the input.

O(n^2) or quadratic time: the algorithm's running time increases quadratically with the size of the input.

O(2^n) or exponential time: the algorithm's running time increases exponentially with the size of the input.

Space complexity, on the other hand, is a measure of how much memory an algorithm uses as a function of the size of the input. Like time complexity, it is typically expressed using big O notation. For example, an algorithm with a space complexity of O(n) uses more memory as the input size (n) increases. Space complexities are generally categorized as:

O(1) or constant space: the algorithm uses the same amount of memory regardless of the size of the input.

O(n) or linear space: the algorithm's memory usage increases linearly with the size of the input.

O(n^2) or quadratic space: the algorithm's memory usage increases quadratically with the size of the input.

O(2^n) or exponential space: the algorithm's memory usage increases exponentially with the

Big O notation (O(f(n))) provides an upper bound on the growth of a function. It describes the worst-case scenario for the time or space complexity of an algorithm. For example, an algorithm with a time complexity of O(n^2) means that the running time of the algorithm is at most n^2, where n is the size of the input.

Big Ω notation (Ω(f(n))) provides a lower bound on the growth of a function. It describes the best-case scenario for the time or space complexity of an algorithm. For example, an algorithm with a space complexity of Ω(n) means that the memory usage of the algorithm is at least n, where n is the size of the input.

Big Θ notation (Θ(f(n))) provides a tight bound on the growth of a function. It describes the average-case scenario for the time or space complexity of an algorithm. For example, an algorithm with a time complexity of Θ(n log n) means that the running time of the algorithm is both O(n log n) and Ω(n log n), where n is the size of the input.

It's important to note that the asymptotic notation only describes the behavior of the function for large values of n, and does not provide information about the exact behavior of the function for small values of n. Also, for some cases, the best, worst and average cases can be the same, in that case the notation will be simplified to O(f(n)) = Ω(f(n)) = Θ(f(n))

Additionally, these notations can be used to compare the efficiency of different algorithms, where a lower order of the function is considered more efficient. For example, an algorithm with a time complexity of O(n) is more efficient than an algorithm with a time complexity of O(n^2).

It's also worth mentioning that asymptotic notation is not only limited to time and space complexity but can be used to express the behavior of any function, not just algorithms.

There are three asymptotic notations that are used to represent the time complexity of an algorithm.

They are:

Input: Here our input is an integer array of size "n" and we have one integer "k" that we need to search for in that array.

Output: If the element "k" is found in the array, then we have return 1, otherwise we have

- *for-loop to iterate with each element in the array*

```
for (int i = 0; i < n; ++i)
```

```
  {
    • check if ith element is equal to "k" or not
    if (arr[i] == k)
      return 1; // return 1, if you find "k"
  }
  return 0; // return 0, if you didn't find "k"
}
```

If the input array is [1, 2, 3, 4, 5] and you want to find if "1" is present in the array or not, then the if-condition of the code will be executed 1 time and it will find that the element 1 is there in the array. So, the if-condition will take 1 second here.

If the input array is [1, 2, 3, 4, 5] and you want to find if "3" is present in the array or not, then the if-condition of the code will be executed 3 times and it will find that the element 3 is there in the array. So, the if-condition will take 3 seconds here.

If the input array is [1, 2, 3, 4, 5] and you want to find if "6" is present in the array or not, then the if-condition of the code will be executed 5 times and it will find that the element 6 is not there in the array and the algorithm will return 0 in this case. So, the if-condition will take 5 seconds here.

As we can see that for the same input array, we have different time for different values of "k". So, this can be divided into three cases:

Best case: This is the lower bound on running time of an algorithm. We must know the case that causes the minimum number of operations to be executed. In the above example, our array was [1, 2, 3, 4, 5] and we are finding if "1" is present in the array or not. So here, after only one comparison, we will get that ddelement is present in the array. So, this is the best case of our algorithm.

Average case: We calculate the running time for all possible inputs, sum all the calculated values and divide the sum by the total number of inputs. We must know (or predict) distribution of cases.

Worst case: This is the upper bound on running time of an algorithm. We must know the case that causes the maximum number of operations to be executed. In our example, the worst case can be if the given array is [1, 2, 3, 4, 5] and we try to find if element "6" is present in the array or not. Here, the if-condition of our loop will be executed 5 times and then the algorithm will give "0" as output.

So, we learned about the best, average, and worst case of an algorithm. Now, let's get back to the asymptotic notation where we saw that we use three asymptotic notation to represent the complexity of an algorithm i.e. Θ Notation (theta), Ω Notation, Big O Notation.

NOTE: In the asymptotic analysis, we generally deal with large input size.

Θ Notation (theta)

The Θ Notation is used to find the average bound of an algorithm i.e. it defines an upper bound and a lower bound, and your algorithm will lie in between these levels. So, if a function is g(n), then the theta representation is shown as Θ(g(n)) and the relation is shown as:

Θ(g(n)) = { f(n): there exist positive constants c1, c2 and n0

Ω Notation

The Ω notation denotes the lower bound of an algorithm i.e. the time taken by the algorithm can't be lower than this. In other words, this is the fastest time in which the algorithm will return a result.

Its the time taken by the algorithm when provided with its best-case input. So, if a function is g(n), then the omega representation is shown as Ω(g(n)) and the relation is shown as:

Ω(g(n)) = { f(n): there exist positive constants c and n0

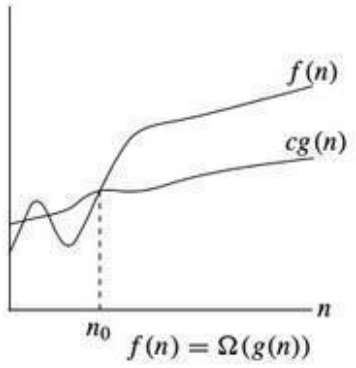
such that $0 \leq cg(n) \leq f(n)$ **for** all $n \geq n0$ }

The above expression can be read as omega of g(n) is defined as set of all the functions f(n) for which there exist some constants c and n0 such that c*g(n) is less than or equal to f(n), for all n greater than or equal to n0.

if f(n) = $2n^2 + 3n + 1$

and **g**(n) = n^2

then **for** c = 2 and n0 = 1, we can say that **f**(n) = Ω(n^2)



Big O Notation

The Big O notation defines the upper bound of any algorithm i.e. you algorithm can't take more time than this time. In other words, we can say that the big O notation denotes the maximum time taken by an algorithm or the worst-case time complexity of an algorithm. So, big O notation is the most used notation for the time complexity of an algorithm. So, if a function is g(n), then the big O representation of g(n) is shown as O(g(n)) and the relation is shown as:

O(g(n)) = { f(n): there exist positive constants c and n0

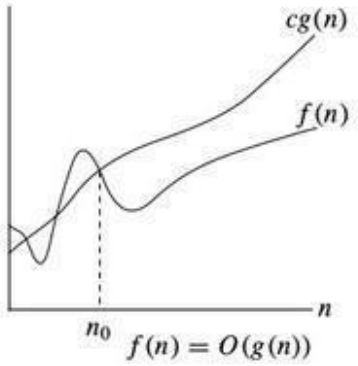
such that $0 \leq f(n) \leq cg(n)$ **for** all $n \geq n0$ }

The above expression can be read as Big O of g(n) is defined as a set of functions f(n) for which there exist some constants c and n0 such that f(n) is greater than or equal to 0 and f(n) is smaller than or equal to c*g(n) for all n greater than or equal to n0.

if f(n) = $2n^2 + 3n + 1$

and **g**(n) = n^2

then **for** c = 6 and n0 = 1, we can say that **f**(n) = O(n²



Big O notation example of Algorithms

Big O notation is the most used notation to express the time complexity of an algorithm. In this section of the blog, we will find the big O notation of various algorithms.

Example 1: Finding the sum of the first n numbers.

In this example, we have to find the sum of first n numbers. For example, if n = 4, then our output should be 1 + 2 + 3 + 4 = 10. If n = 5, then the ouput should be 1 + 2 + 3 + 4 + 5 = 15. Let's try various solutions to this code and try to compare all those codes.

O(1) solution

• *function taking input "n"* **int**

findSum(**int** n)

{

return n * (n+1) */ 2; // this will take some constant time c1*

}

In the above code, there is only one statement and we know that a statement takes constant time for its execution. The basic idea is that if the statement is taking constant time, then it will take the same amount of time for all the input size and we denote this as ***O(1)*** .

O(n) solution

In this solution, we will run a loop from 1 to n and we will add these values to a variable named "sum".

• *function taking input "n"* **int**

findSum(**int** n)

{

int sum = 0; *// ----------------> it takes some constant time "c1"*

for(**int** i = 1; i <= n; ++i) *// --> here the comparision and increment will take place n times(c2*n) and the creation of i takes place with some constant time*

sum = sum + i; *// ----------> this statement will be executed n times i.e. c3*n*

return sum; *// ----------------> it takes some constant time "c4"*

}

*/**

• Total time taken = time taken by all the statments to execute

• here in our example we have 3 constant time taking statements i.e. "sum = 0", "i = 0", and "return sum", so we can add all the constatnts and replacce with some **new** constant "c"

• apart from **this**, we have two statements running n-times i.e. "i < n(in real n+1)" and "sum = sum + i" i.e. c2*n + c3*n = c0*n

• Total time taken = c0*n + c

*/

The big O notation of the above code is O(c0*n) + O(c), where c and c0 are constants. So, the overall time complexity can be written as ***O(n)*** .

O(n²) solution

In this solution, we will increment the value of sum variable "i" times i.e. for i = 1, the sum variable will be incremented once i.e. sum = 1. For i = 2, the sum variable will be incremented twice. So, let's see the solution.

- *function taking input "n"* **int**

```
findSum(int n)
{
   int sum = 0; // ----------------------> constant time
   for(int i = 1; i <= n; ++i)
      for(int j = 1; j <= i; ++j)
         sum++; // -------------------> it will run [n * (n + 1) / 2]
   return sum; // ----------------------> constant time
}
```

/

* Total time taken = time taken by all the statments to execute

* the statement that is being executed most of the time is "sum++" i.e. n * (n + 1) */ 2*

- So, total complexity will be: c1*n² + c2*n + c3 [c1 is **for** the constant terms of n², c2 is **for** the constant terms of n, and c3 is **for** rest of the constant time]

/

The big O notation of the above algorithm is O(c1*n²) +O(c2*n) + O(c3). Since we take the higher order of growth in big O. So, our expression will be reduced to ***O(n²)*** .
So, until now, we saw 3 solutions for the same problem. Now, which algorithm will you prefer to use when you are finding the sum of first "n" numbers? If your answer is O(1) solution, then we have one bonus section for you at the end of this blog. We would prefer the O(1) solution because the time taken by the algorithm will be constant irrespective of the input size.

Recurrence Relation

A recurrence relation is a mathematical equation that describes the relation between the input size and the running time of a recursive algorithm. It expresses the running time of a problem in terms of the running time of smaller instances of the same problem.

A recurrence relation typically has the form T(n) = aT(n/b) + f(n) where:

T(n) is the running time of the algorithm on an input of size n a

is the number of recursive calls made by the algorithm

b is the size of the input passed to each recursive call

f(n) is the time required to perform any non-recursive operations

The recurrence relation can be used to determine the time complexity of the algorithm using techniques such as the Master Theorem or Substitution Method.

For example, let's consider the problem of computing the nth Fibonacci number. A simple recursive algorithm for solving this problem is as follows:

Fibonacci(n)

if n <= 1

return n

else

return Fibonacci(n-1) + Fibonacci(n-2)

The recurrence relation for this algorithm is T(n) = T(n-1) + T(n-2) + O(1), which describes the running time of the algorithm in terms of the running time of the two smaller instances of the problem with input sizes n-1 and n-2. Using the Master Theorem, it can be shown that the time complexity of this algorithm is O(2^n) which is very inefficient for large input sizes.

Searching

Searching is the process of fetching a specific element in a collection of elements. The collection can be an array or a linked list. If you find the element in the list, the process is considered successful, and it returns the location of that element.

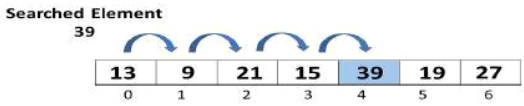
Two prominent search strategies are extensively used to find a specific item on a list. However, the algorithm chosen is determined by the list's organization.

- Linear Search
- Binary Search
- Interpolation search

Linear Search

Linear search, often known as sequential search, is the most basic search technique. In this type of search, we go through the entire list and try to fetch a match for a single element. If we find a match, then the address of the matching target element is returned. On the other hand, if the element is not found, then it returns a NULL value.

Following is a step-by-step approach employed to perform Linear Search Algorithm.



The procedures for implementing linear search are as follows:

Step 1: First, read the search element (Target element) in the array.

Step 2: In the second step compare the search element with the first element in the array.

Step 3: If both are matched, display "Target element is found" and terminate the Linear Search function.

Step 4: If both are not matched, compare the search element with the next element in the array. Step 5: In this step, repeat steps 3 and 4 until the search (Target) element is compared with the last element of the array.

Step 6 - If the last element in the list does not match, the Linear Search Function will be terminated, and the message "Element is not found" will be displayed.

Algorithm and Pseudocode of Linear Search Algorithm
Algorithm of the Linear Search Algorithm

Linear Search (Array Arr, Value a) // Arr is the name of the array, and a is the searched element.
Step 1: Set i to 0 // i is the index of an array which starts from 0

Step 2: ifi > n then go to step 7 // n is the number of elements in array

Step 3: if Arr[i] = a then go to step 6

Step 4: Set i to i + 1

Step 5: Goto step 2

Step 6: Print element a found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

Pseudocode of Linear Search Algorithm

Start

linear_search (Array , value)

For each element in the array

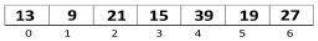
```
  If (searched element == value)
     Return's the searched element location
  end if
end for
```

end

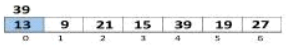
Example of Linear Search Algorithm

Consider an array of size 7 with elements 13, 9, 21, 15, 39, 19, and 27 that starts with 0 and ends with size minus one, 6.

Search element = 39

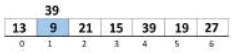


Step 1: The searched element 39 is compared to the first element of an array, which is 13.

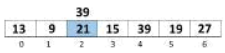


The match is not found, you now move on to the next element and try to implement a comparison.

Step 2: Now, search element 39 is compared to the second element of an array, 9.

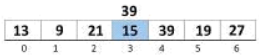


Step 3: Now, search element 39 is compared with the third element, which is 21.

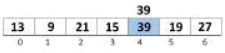


Again, both the elements are not matching, you move onto the next following element.

Step 4; Next, search element 39 is compared with the fourth element, which is 15.



Step 5: Next, search element 39 is compared with the fifth element 39.



A perfect match is found, display the element found at location 4.

The Complexity of Linear Search Algorithm

Three different complexities faced while performing Linear Search Algorithm, they are mentioned as follows.

- Best Case
- Worst Case
- Average Case

Best Case Complexity

The element being searched could be found in the first position. In this case, the search ends with a single successful comparison.

Thus, in the best-case scenario, the linear search algorithm performs O(1) operations.

Worst Case Complexity

The element being searched may be at the last position in the array or not at all. In the first case, the search succeeds in 'n' comparisons.

In the next case, the search fails after 'n' comparisons.

Thus, in the worst-case scenario, the linear search algorithm performs O(n) operations.

Average Case Complexity

When the element to be searched is in the middle of the array, the average case of the Linear Search Algorithm is O(n).

Space Complexity of Linear Search Algorithm

The linear search algorithm takes up no extra space; its space complexity is O(n) for an array of n elements.

Application of Linear Search Algorithm

The linear search algorithm has the following applications:

Linear search can be applied to both single-dimensional and multi-dimensional arrays.

Linear search is easy to implement and effective when the array contains only a few elements.

Linear Search is also efficient when the search is performed to fetch a single search in an unordered-List.

Code Implementation of Linear Search Algorithm

```
#include<stdio.h>

#include<stdlib.h>

#include<conio.h>

int main()

{

   int array[50],i,target,num;



   printf("How many elements do you want in the array");

   scanf("%d",&num);

   printf("Enter array elements:");

   for(i=0;i<num;++i)

      scanf("%d",&array[i]);

   printf("Enter element to search:");

   scanf("%d",&target);

   for(i=0;i<num;++i)

      if(array[i]==target)

         break;

   if(i<num)

      printf("Target element found at location %d",i);

   else

      printf("Target element not found in an array");

   return 0;
```

}

Binary Search

Binary search is the search technique that works efficiently on sorted lists. Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted. Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match

NOTE: Binary search can be implemented on sorted array elements. If the list elements are not arranged in a sorted manner, we have first to sort them.

Algorithm

- Binary_Search(a, lower_bound, upper_bound, val) // 'a' is the given array, 'lower_bound' is t he index of the first array element, 'upper_bound' is the index of the last array element, 'val' is the value to search
- Step 1: set beg = lower_bound, end = upper_bound, pos = - 1
- Step 2: repeat steps 3 and 4 while beg **<**=end
- Step 3: set mid = (beg + end)/2
- Step 4: if a[mid] = val
- set pos = mid
- print pos
- go to step 6
- else if a[mid] **>** val
- set end = mid - 1
- else
- set beg = mid + 1
- [end of if]
- [end of loop]

- Step 5: if pos = -1
- print "value is not present in the array"
- [end of if]
- Step 6: exit

Procedure binary_search

```
A ← sorted array
n ← size of array
x ← value to be searched
Set lowerBound = 1
Set upperBound = n
while x not found
  if upperBound < lowerBound
    EXIT: x does not exists.
    set midPoint = lowerBound + ( upperBound - lowerBound ) / 2 if
  A[midPoint] < x
    set lowerBound = midPoint + 1
  if A[midPoint] > x
    set upperBound = midPoint - 1
  if A[midPoint] = x
    EXIT: x found at location midPoint
end while
end procedure
```

Working of Binary search

To understand the working of the Binary search algorithm, let's take a sorted array. It will be easy to understand the working of Binary search with an example.

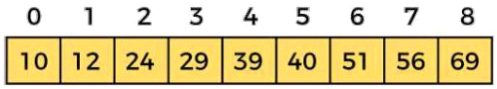
There are two methods to implement the binary search algorithm -

- Iterative method ○

Recursive method

The recursive method of binary search follows the divide and conquer approach.

Let the elements of array are -



Let the element to search is, **K = 56**

We have to use the below formula to calculate the **mid** of the array -

- mid = (beg + end)/2

So, in the given array -

beg = 0

end = 8

mid = (0 + 8)/2 = 4. So, 4 is the mid of the array.

Now, the element to search is found. So algorithm will return the index of the element matched.
Binary Search complexity

Now, let's see the time complexity of Binary search in the best case, average case, and worst case.

We will also see the space complexity of Binary search.

1. Time Complexity

Case	Time Complexity
Best Case	O(1)
Average Case	O(logn)
Worst Case	O(logn)

- **Best Case Complexity -** In Binary search, best case occurs when the element to search is found in first comparison, i.e., when the first middle element itself is the element to be

searched. The best-case time complexity of Binary search is **O(1).**

- **Average Case Complexity -** The average case time complexity of Binary search is **O(logn).**
- **Worst Case Complexity -** In Binary search, the worst case occurs, when we have to keep reducing the search space till it has only one element. The worst-case time complexity of Binary search is **O(logn).**

- Space Complexity

Space Complexity	O(1)

- The space complexity of binary search is O(1).

Implementation of Binary Search

Program: Write a program to implement Binary search in C language.

```
#include <stdio.h>
int binarySearch(int a[], int beg, int end, int val)
{
   int mid;
   if(end >= beg)
{    mid = (beg + end)/2;
/* if the item to be searched is present at middle */
   if(a[mid] == val)
   {
      return mid+1;
   }
      /* if the item to be searched is smaller than middle, then it can only be in left subarra
y */
   else if(a[mid] < val)
```

```
        {
            return binarySearch(a, mid+1, end, val);
        }
            /* if the item to be searched is greater than middle, then it can only be in right subarr
    ay */
        else
        {
            return binarySearch(a, beg, mid-1, val);
        }
     }
     return -1;
}
int main() {
   int a[] = {11, 14, 25, 30, 40, 41, 52, 57, 70}; // given array
   int val = 40; // value to be searched
   int n = sizeof(a) / sizeof(a[0]); // size of array
   int res = binarySearch(a, 0, n-1, val); // Store result
   printf("The elements of the array are - ");
   for (int i = 0; i < n; i++)
   printf("%d ", a[i]);
   printf("\nElement to be searched is - %d", val);
   if (res == -1)
   printf("\nElement is not present in the array");
   else
```

- printf("\nElement is present at %d position of array", res);
- **return** 0;
- }

Output

```
The elements of the array are - 11 14 25 30 40 41 52 57 70
Element to be searched is - 40
Element is present at 5 position of array
```

Interpolation Search

Interpolation search is an improved variant of binary search. This search algorithm works on the probing position of the required value. For this algorithm to work properly, the data collection should be in a sorted form and equally distributed.

Binary search has a huge advantage of time complexity over linear search. Linear search has worst-case complexity of O(n) whereas binary search has O(log n).There are cases where the location of target data may be known in advance. For example, in case of a telephone directory, if we want to search the telephone number of Morphius. Here, linear search and even binary search will seem slow as we can directly jump to memory space where the names start from 'M' are stored.

Position Probing in Interpolation Search

Interpolation search finds a particular item by computing the probe position. Initially, the probe position is the position of the middle most item of the collection.

If a match occurs, then the index of the item is returned. To split the list into two parts, we use the following method –

mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) * (X - A[Lo])

where —

A = list

Lo = Lowest index of the list

Hi = Highest index of the list

A[n] = Value stored at index n in the list

If the middle item is greater than the item, then the probe position is again calculated in the sub-array to the right of the middle item. Otherwise, the item is searched in the subarray to the left of the middle item. This process continues on the sub-array as well until the size of subarray reduces to zero.

Runtime complexity of interpolation search algorithm is **O(log (log n))** as compared to **O(log n)** of

BST in favorable situations.

Algorithm

As it is an improvisation of the existing BST algorithm, we are mentioning the steps to search the

'target' data value index, using position probing —

Step 1 — Start searching **data** from middle of the list.

Step 2 — If it is a match, return the index of the item, and exit.

Step 3 — If it is not a match, probe position.

Step 4 — Divide the list using probing formula and find the new midle.

Step 5 — If data is greater than middle, search in higher sub-list.

Step 6 — If data is smaller than middle, search in lower sub-list.

Step 7 — Repeat until match.

Pseudocode
A → Array list
N → Size of A
X → Target Value

Procedure Interpolation_Search()

Set Lo → 0
Set Mid → -1

```
  Set Hi → N-1

  While X does not match

    if Lo equals to Hi OR A[Lo] equals to A[Hi]
      EXIT: Failure, Target not found
    end if

    Set Mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) * (X - A[Lo])

    if A[Mid] = X
      EXIT: Success, Target found at Mid
    else
      if A[Mid] < X
        Set Lo to Mid+1
          else if A[Mid] > X
            Set Hi to Mid-1
      end if
    end if
  End While

End Procedure
```

Implementation of interpolation in C

```
#include<stdio.h>

#define MAX 10
```

• array of items on which linear search will be conducted. int list[MAX] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44 }; int find(int data) {

```
  int lo = 0;

  int hi = MAX - 1; int
  mid = -1;

  int comparisons = 1; int
  index = -1; while(lo <=
  hi) {

    printf("\nComparison %d \n" , comparisons ) ;
    printf("lo : %d, list[%d] = %d\n", lo, lo, list[lo]);
    printf("hi : %d, list[%d] = %d\n", hi, hi, list[hi]);

    comparisons++;
```

```
    // probe the mid point

    mid = lo + (((double)(hi - lo) / (list[hi] - list[lo])) * (data - list[lo]));
    printf("mid = %d\n",mid);

    • data found if(list[mid]
    == data) {

      index = mid;
      break;
    } else {

      if(list[mid] < data) {

        • if data is larger, data is in upper half lo
        = mid + 1;

      } else {

        • if data is smaller, data is in lower half hi
        = mid - 1;

      }

    }

  }

  printf("\nTotal comparisons made: %d", --comparisons);
  return index;

}

int main() {

  //find location of 33

  int location = find(33);

  • if element was found
  if(location != -1)

    printf("\nElement found at location: %d" ,(location+1));
  else
    printf("Element not found.");
  return 0;
```

}

If we compile and run the above program, it will produce the following result —

Output

Comparison 1

lo : 0, list[0] = 10

hi : 9, list[9] = 44

mid = 6

Total comparisons made: 1

Element found at location: 7

Time Complexity

Bestcase-O(1)

The best-case occurs when the target is found exactly as the first expected position computed using the formula. As we only perform one comparison, the time complexity is O(1).

Worst-case-O(n)

The worst case occurs when the given data set is exponentially distributed.

Averagecase-O(log(log(n)))

If the data set is sorted and uniformly distributed, then it takes O(log(log(n))) time as on an average (log(log(n))) comparisons are made.

Space Complexity

O(1) as no extra space is required.

Pattern Search

Pattern Searching algorithms are used to find a pattern or substring from another bigger string. There are different algorithms. The main goal to design these type of algorithms to reduce the time complexity. The traditional approach may take lots of time to complete the pattern searching task for a longer text.

Here we will see different algorithms to get a better performance of pattern matching.

In this Section We are going to cover.

Aho-Corasick Algorithm
Anagram Pattern Search
Bad Character Heuristic
Boyer Moore Algorithm

Efficient Construction of Finite Automata
kasai's Algorithm
Knuth-Morris-Pratt Algorithm
Manacher's Algorithm

Naive Pattern Searching
Rabin-Karp Algorithm Suffix
Array

Trie of all Suffixes Z
Algorithm

Naïve pattern searching is the simplest method among other pattern searching algorithms. It checks for all character of the main string to the pattern. This algorithm is helpful for smaller texts. It does not need any pre-processing phases. We can find substring by checking once for the string. It also does not occupy extra space to perform the operation.

The time complexity of Naïve Pattern Search method is O(m*n). The m is the size of pattern and n is the size of the main string.

Input and Output

Input:

Main String: "ABAAABCDBBABCDDEBCABC", pattern: "ABC"

Output:

Pattern found at position: 4

Pattern found at position: 10

Pattern found at position: 18

Algorithm

naive_algorithm(pattern, text)

Input – The text and the pattern

Output – locations, where the pattern is present in the text Start

```
  pat_len := pattern Size
  str_len := string size

  for i := 0 to (str_len - pat_len), do

    for j := 0 to pat_len, do

      if text[i+j] ≠ pattern[j], then

      break

  if j == patLen, then

  display the position i, as there pattern found

End
```

Implementation in C

```
#include <stdio.h>

#include <string.h>

int main (){

  char txt[] = "tutorialsPointisthebestplatformforprogrammers";

  char pat[] = "a";

  int M = strlen (pat);

  int N = strlen (txt);

  for (int i = 0; i <= N - M; i++){

    int j;

    for (j = 0; j < M; j++)

      if (txt[i + j] != pat[j])

    break;

    if (j == M)

                                        printf ("Pattern matches at index %d
```

", i);

  }

  return 0;

}

Output

Pattern matches at 6

Pattern matches at 25

Pattern matches at 39

Rabin-Karp matching pattern

Rabin-Karp is another pattern searching algorithm. It is the string matching algorithm that was proposed by Rabin and Karp to find the pattern in a more efficient way. Like the Naive Algorithm, it also checks the pattern by moving the window one by one, but without checking all characters for all cases, it finds the hash value. When the hash value is matched, then only it proceeds to check each character. In this way, there is only one comparison per text subsequence making it a more efficient algorithm for pattern searching.

Preprocessing time- O(m)

The time complexity of the Rabin-Karp Algorithm is **O(m+n)**, but for the worst case, it is **O(mn)**.
Algorithm

rabinkarp_algo(text, pattern, prime)

Input – The main text and the pattern. Another prime number of find hash location
Output – locations, where the pattern is found

Start

  pat_len := pattern Length

  str_len := string Length

  patHash := 0 and strHash := 0, h := 1

  maxChar := total number of characters in character set for
index i of all character in the pattern, do

  h := (h*maxChar) mod prime

```
for all character index i of pattern, do

  patHash := (maxChar*patHash + pattern[i]) mod prime
  strHash := (maxChar*strHash + text[i]) mod prime

for i := 0 to (str_len - pat_len), do

  if patHash = strHash, then

    for charIndex := 0 to pat_len -1, do

      if text[i+charIndex] ≠ pattern[charIndex], then

      break

if charIndex = pat_len, then

  print the location i as pattern found at i position.

if i < (str_len - pat_len), then

  strHash := (maxChar*(strHash – text[i]*h)+text[i+patLen]) mod prime, then

  if strHash < 0, then

  strHash := strHash + prime

End
```

Implementation In C

```
#include<stdio.h>

#include<string.h>

int main (){

  char txt[80], pat[80];

  int q;

 printf("Enterthecontainerstring ");
 scanf ("%s", &txt);
 printf("Enterthepatterntobesearched ");
  scanf ("%s", &pat);
  int d = 256;
printf("Enteraprimenumber ");
  scanf ("%d", &q);
```

```
  int M = strlen (pat);
  int N = strlen (txt);
  int i, j;
  int p = 0;
  int t = 0;
  int h = 1;
  for (i = 0; i < M - 1; i++)
    h = (h * d) % q;
  for (i = 0; i < M; i++){
    p = (d * p + pat[i]) % q;
    t = (d * t + txt[i]) % q;
  }
  for (i = 0; i <= N - M; i++){
    if (p == t){
      for (j = 0; j < M; j++){
        if (txt[i + j] != pat[j])
        break;
      }
      if (j == M)
printf("Patternfoundatindex%d ", i);
    }
    if (i < N - M){
      t = (d * (t - txt[i] * h) + txt[i + M]) % q;
      if (t < 0)
```

```
        t = (t + q);
    }
  }
  return 0;
}
```

Output

Enter the container string

tutorialspointisthebestprogrammingwebsite

Enter the pattern to be searched

p

Enter a prime number

3

Pattern found at index 8

Pattern found at index 21

n this problem, we are given two strings a text and a pattern. Our task is to create a program for KMP algorithm for pattern search, it will find all the occurrences of pattern in text string. Here, we have to find all the occurrences of patterns in the text.

Let's take an example to understand the problem,
Input

text = "xyztrwqxyzfg" pattern = "xyz"

Output

Found at index 0

Found at index 7

Here, we will discuss the solution to the problem using KMP (*Knuth Morris Pratt*) pattern searching algorithm, it will use a preprocessing string of the pattern which will be used for matching in the text. And help's in processing or finding pattern matches in the case where matching characters are followed by the character of the string that does not match the pattern.

We will preprocess the pattern wand to create an array that contains the proper prefix and suffix from the pattern that will help in finding the mismatch patterns. Program for KMP Algorithm for Pattern Searching

- C Program for KMP Algorithm for Pattern Searching

Example

```
#include<iostream>

#include<string.h>

using namespace std;

void prefixSuffixArray(char* pat, int M, int* pps) {

  int length = 0;

  pps[0] = 0;

  int i = 1;

  while (i < M) {

    if (pat[i] == pat[length]) {

      length++;

      pps[i] = length;

      i++;

    } else {

      if (length != 0)

      length = pps[length - 1];

      else {

        pps[i] = 0;

        i++;

      }

    }

  }
```

```
}
void KMPAlgorithm(char* text, char* pattern) {
  int M = strlen(pattern);
  int N = strlen(text);
  int pps[M];
  prefixSuffixArray(pattern, M, pps);
  int i = 0;
  int j = 0;
  while (i < N) {
    if (pattern[j] == text[i]) {
      j++;
      i++;
    }
    if (j == M)
{
printf("Foundpatternatindex%d", i - j);
      j = pps[j - 1];
    }
    else if (i < N && pattern[j] != text[i]) {
      if (j != 0)
      j = pps[j - 1];
      else
      i = i + 1;
    }
  }
```

```
}

int main() {

  char text[] = "xyztrwqxyzfg";

      char pattern[] = "xyz";

     printf("Thepatternisfoundinthetextatthefollowingindex : ");

      KMPAlgorithm(text, pattern);

      return 0;

}
```

Output

The pattern is found in the text at the following index −

Found pattern at index 0

Found pattern at index 7

Sorting : Insertion sort

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.

The same approach is applied in insertion sort. The idea behind the insertion sort is that first take one element, iterate it through the sorted array. Although it is simple to use, it is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is **$O(n^2)$**, where n is the number of items. Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.

Algorithm

The simple steps of achieving the insertion sort are listed as follows -

Step 1 - If the element is the first element, assume that it is already sorted. Return 1.

Step2 - Pick the next element, and store it separately in a **key.**

Step3 - Now, compare the **key** with all elements in the sorted array.

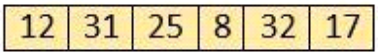
Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right. **Step 5 -** Insert the value.

Step 6 - Repeat until the array is sorted.

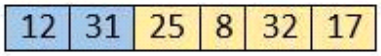
Working of Insertion sort Algorithm

Now, let's see the working of the insertion sort Algorithm.

To understand the working of the insertion sort algorithm, let's take an unsorted array. It will be easier to understand the insertion sort via an example. Let the elements of array are -

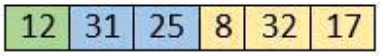


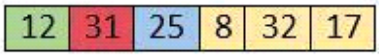
Initially, the first two elements are compared in insertion sort.



Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.

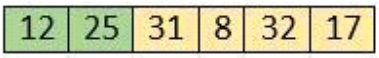
Now, move to the next two elements and compare them.



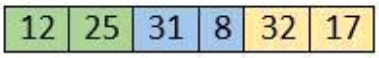


Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

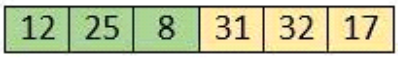
For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.



Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.

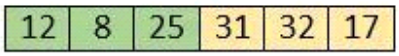


Both 31 and 8 are not sorted. So, swap them.

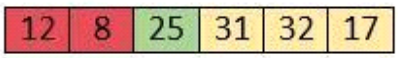


After swapping, elements 25 and 8 are unsorted.

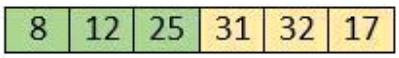
So, swap them.



Now, elements 12 and 8 are unsorted.



So, swap them too.



Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.

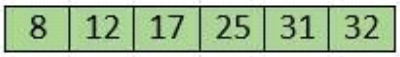
Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

Move to the next elements that are 32 and 17.

17 is smaller than 32. So, swap them.

Swapping makes 31 and 17 unsorted. So, swap them too.

Now, swapping makes 25 and 17 unsorted. So, perform swapping again.



Now, the array is completely sorted.

Insertion sort complexity

1. Time Complexity

Case	Time Complexity
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is **O(n)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of insertion sort is **$O(n^2)$**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is **$O(n^2)$**.

- Space Complexity

Space Complexity	O(1)
Stable	YES

- The space complexity of insertion sort is O(1). It is because, in insertion sort, an extra variable is required for swapping.

Implementation of insertion sort

Program: Write a program to implement insertion sort in C language.

- #include <stdio.h>
-
- **void** insert(**int** a[], **int** n) /* function to sort an aay with insertion sort */

```
{
    int i, j, temp;
    for (i = 1; i < n; i++) {
        temp = a[i];
        j = i - 1;
```

```
9.      while(j>=0 && temp <= a[j]) /* Move the elements greater than temp to one position a
10.
        head from their current position*/
11.     {
12.         a[j+1] = a[j];
13.         j = j-1;
14.     }
15.     a[j+1] = temp;
16.   }
17. }
18.
```

```
void printArr(int a[], int n) /* function to print the array */
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
}
25.
int main()
{
    int a[] = { 12, 31, 25, 8, 32, 17 };
    int n = sizeof(a) / sizeof(a[0]);
    printf("Before sorting array elements are - \n");
    printArr(a, n);
    insert(a, n);
```

- printf("\nAfter sorting array elements are - \n");
- printArr(a, n);

35.

- **return** 0;
- }

Output:

```
Before sorting array elements are -
12 31 25 8 32 17
After sorting array elements are -
8 12 17 25 31 32
```

Heap Sort

Heap Sort Algorithm

Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array. Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array.

Heap sort basically recursively performs two main operations -

- Build a heap H, using the elements of array.
- Repeatedly delete the root element of the heap formed in 1^{st} phase.

A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified.

Heapsort is a popular and efficient sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.

Algorithm

- HeapSort(arr)
- BuildMaxHeap(arr)
- for i = length(arr) to 2
- swap arr[1] with arr[i]
- heap_size[arr] = heap_size[arr] ? 1
- MaxHeapify(arr,1)
- End

BuildMaxHeap(arr)

- BuildMaxHeap(arr)
- heap_size(arr) = length(arr)
- for i = length(arr)/2 to 1
- MaxHeapify(arr,i)
- End

MaxHeapify(arr,i)

- MaxHeapify(arr,i)
- L = left(i)
- R = right(i)
- if L ? heap_size[arr] and arr[L] **>** arr[i]
- largest = L
- else
- largest = i
- if R ? heap_size[arr] and arr[R] **>** arr[largest]
- largest = R
- if largest != i

- swap arr[i] with arr[largest]
- MaxHeapify(arr,largest)
- End

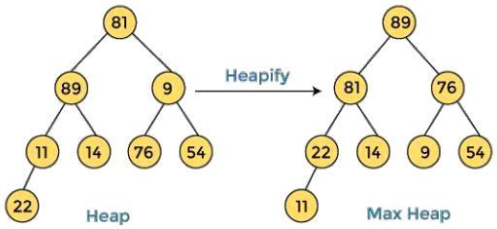
Working of Heap sort Algorithm

In heap sort, basically, there are two phases involved in the sorting of elements. By using the heap sort algorithm, they are as follows -

- The first step includes the creation of a heap by adjusting the elements of the array.
- After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.

81	89	9	11	14	76	54	22

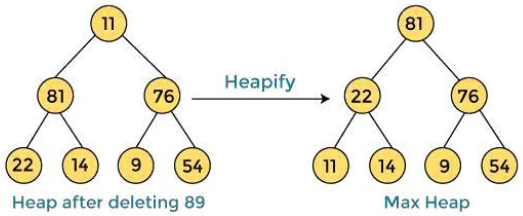
First, we have to construct a heap from the given array and convert it into max heap.



After converting the given heap into max heap, the array elements are -

89	81	76	22	14	9	54	11

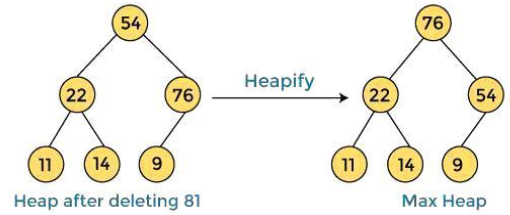
Next, we have to delete the root element **(89)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **89** with **11,** and converting the heap into max-heap, the elements of array are -

81	22	76	11	14	9	54	89

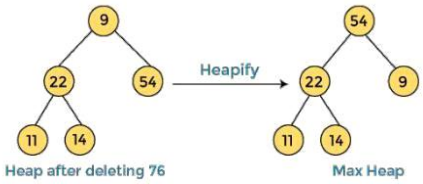
In the next step, again, we have to delete the root element **(81)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(54).** After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **81** with **54** and converting the heap into max-heap, the elements of array are -

76	22	54	11	14	9	81	89

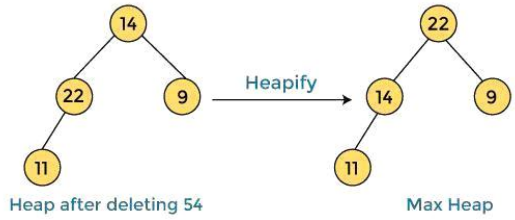
In the next step, we have to delete the root element **(76)** from the max heap again. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **76** with **9** and converting the heap into max-heap, the elements of array are -

54	22	9	11	14	76	81	89

In the next step, again we have to delete the root element **(54)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(14).** After deleting the root element, we again have to heapify it to convert it into max heap.



After swapping the array element **54** with **14** and converting the heap into max-heap, the elements of array are -

22	14	9	11	54	76	81	89

In the next step, again we have to delete the root element **(22)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.

After swapping the array element **22** with **11** and converting the heap into max-heap, the elements of array are -

14	11	9	22	54	76	81	89

In the next step, again we have to delete the root element **(14)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

After swapping the array element **14** with **9** and converting the heap into max-heap, the elements of array are -

11	9	14	22	54	76	81	89

In the next step, again we have to delete the root element **(11)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

After swapping the array element **11** with **9,** the elements of array are -

9	11	14	22	54	76	81	89

Now, heap has only one element left. After deleting it, heap will be empty.

After completion of sorting, the array elements are -

Time complexity of Heap sort in the best case, average case, and worst

case 1. Time Complexity

Case

Time Complexity

Best Case	O(n logn)
Average Case	O(n log n)
Worst Case	O(n log n)

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of heap sort is **O(n logn).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of heap sort is **O(n log n).**

- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of heap sort is **O(n log n).**

The time complexity of heap sort is **O(n logn)** in all three cases (best case, average case, and worst case). The height of a complete binary tree having n elements is **logn.**

2. Space Complexity

Space Complexity	O(1)
Stable	N0

- The space complexity of Heap sort is O(1).

Implementation of Heapsort

Program: Write a program to implement heap sort in C language.

```
#include <stdio.h>
/* function to heapify a subtree. Here 'i' is the
index of root node in array a[], and 'n' is the size of heap. */
void heapify(int a[], int n, int i)
{
   int largest = i; // Initialize largest as root
   int left = 2 * i + 1; // left child
   int right = 2 * i + 2; // right child
   // If left child is larger than root
   if (left < n && a[left] > a[largest])
      largest = left;
// If right child is larger than root
if (right < n && a[right] > a[largest])
   largest = right;
```

```
    // If root is not largest
    if (largest != i) {
      // swap a[i] with a[largest]
      int temp = a[i];
      a[i] = a[largest];
      a[largest] = temp;
     heapify(a, n, largest);
    }
  }
  /*Function to implement the heap sort*/
  void heapSort(int a[], int n)
  {
    for (int i = n / 2 - 1; i >= 0; i--)
      heapify(a, n, i);
    // One by one extract an element from heap
    for (int i = n - 1; i >= 0; i--) {
      /* Move current root element to end*/
      // swap a[0] with a[i]
      int temp = a[0];
      a[0] = a[i];
      a[i] = temp;
36.
      heapify(a, i, 0);
    }
```

```
}
/* function to print the array elements */
void printArr(int arr[], int n)
{
   for (int i = 0; i < n; ++i)
   {
      printf("%d", arr[i]);
      printf(" ");
   }
48.
}
int main()
{
   int a[] = {48, 10, 23, 43, 28, 26, 1};
   int n = sizeof(a) / sizeof(a[0]);
   printf("Before sorting array elements are - \n");
   printArr(a, n);
   heapSort(a, n);
   printf("\nAfter sorting array elements are - \n");
   printArr(a, n);
   return 0;
}
```

Output

```
Before sorting array elements are -
48 10 23 43 28 26 1
After sorting array elements are -
1 10 23 26 28 43 48
```

UNIT 2 - GRAPHS: basics, representation, traversals, and application

Basic concepts

Definition

A graph G(V, E) is a non-linear data structure that consists of node and edge pairs of objects connected by links.

There are 2 types of graphs:

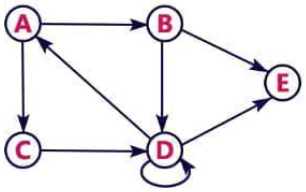
- Directed
- Undirected

Directed graph

A graph with only directed edges is said to be a **directed graph**.

Example

The following directed graph has 5 vertices and 8 edges. This graph G can be defined as G = (V, E), where V = {A,B,C,D,E} and E = {(A,B), (A,C) (B, E), (B,D), (D, A), (D, E),(C,D),(D,D)}.



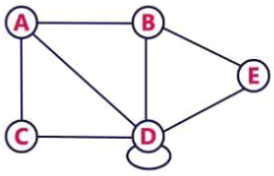
Directed Graph

Undirected graph

A graph with only undirected edges is said to be an **undirected graph**.

Example

The following is an undirected graph.



Undirected Graph

Representation of Graphs

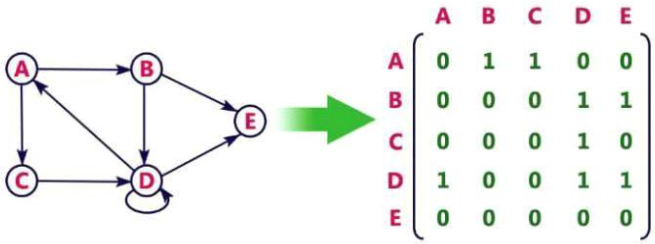
Graph data structure is represented using the following representations.

- Adjacency Matrix
- Adjacency List

Adjacency Matrix

In this representation, the graph can be represented using a matrix of size n x n, where n is the number of vertices. This matrix is filled with either 1's or 0's.

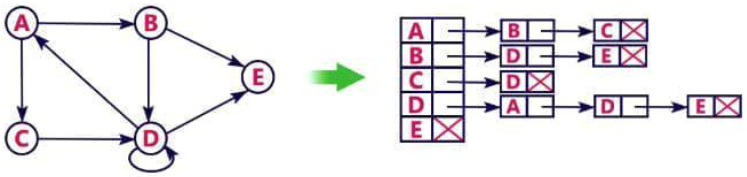
Here, 1 represents that there is an edge from row vertex to column vertex, and 0 represents that there is no edge from row vertex to column vertex.



Directed graph representation

Adjacency list

In this representation, every vertex of the graph contains a list of its adjacent vertices.
If the graph is not dense, i.e., the number of edges is less, then it is efficient to represent the graph through the adjacency list.



Adjacency List

Graph traversals

Graph traversal is a technique used to search for a vertex in a graph. It is also used to decide the order of vertices to be visited in the search process.
A graph traversal finds the edges to be used in the search process without creating loops. This means that, with graph traversal, we can visit all the vertices of the graph without getting into a looping path. There are two graph traversal techniques:

- DFS (Depth First Search)
- BFS (Breadth-First Search)

Applications of graphs

1. **Social network graphs :** To tweet or not to tweet. Graphs that represent who knows whom, who communicates with whom, who influences whom, or other relationships in social structures. An example is the twitter graph of who follows whom.

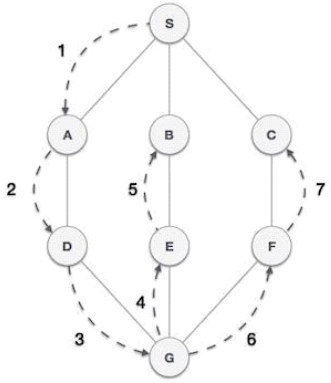
2. **Graphs in epidemiology:** Vertices represent individuals and directed edges to view the transfer of an infectious disease from one individual to another. Analyzing such graphs has become an important component in understanding and controlling the spread of diseases.
3. **Protein-protein interactions graphs:** Vertices represent proteins and edges represent interactions between them that carry out some biological function in the cell. These graphs can be used to, for

example, study molecular pathway—chains of molecular interactions in a cellular process.

4. **Network packet traffic graphs:** Vertices are IP (Internet protocol) addresses and edges are the packets that flow between them. Such graphs are used for analyzing network security, studying the spread of worms, and tracking criminal or non-criminal activity.
5. **Neural networks:** Vertices represent neurons and edges are the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 1011 neurons and close to 1015 synapses.

DFS – Depth First Search

Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

Rule 1 — Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

Rule 2 – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

Rule 3 – Repeat Rule 1 and Rule 2 until the stack is empty.

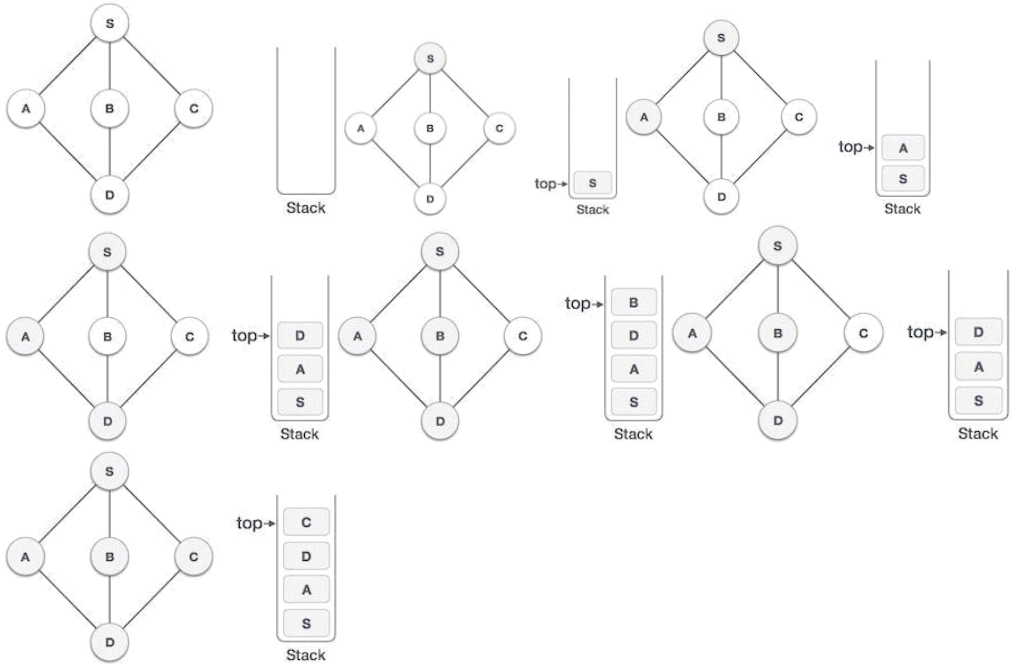
Step

Traversal

Description

1		
		Initialize the stack.
2		Mark **S** as visited and put it onto the
		stack. Explore any unvisited adjacent
		node from **S**. We have three nodes and
		we can pick any of them. For this
		example, we shall take the node in an
		alphabetical order.
3		Mark **A** as visited and put it onto the
		stack. Explore any unvisited adjacent
		node from A. Both **S** and **D** are
		adjacent to **A** but we are concerned
		for unvisited nodes only.
4		Visit **D** and mark it as visited and put
		onto the stack. Here, we
		have **B** and **C** nodes, which are
		adjacent to **D** and both are unvisited.
		However, we shall again choose in an
		alphabetical order.
5		We choose **B**, mark it as visited and
		put onto the stack. Here **B** does not
		have any unvisited adjacent node. So,
		we pop **B** from the stack.

6		
		We check the stack top for return to
		the previous node and check if it has
		any unvisited nodes. Here, we
		find **D** to be on the top of the stack.
7		
		Only unvisited adjacent node is
		from **D** is **C** now. So we visit **C**, mark it
		as visited and put it onto the stack.



Pseudocode of DFS

DFS(G, u)

u.visited = true

for each v			G.Adj[u]
if v.visited == false			

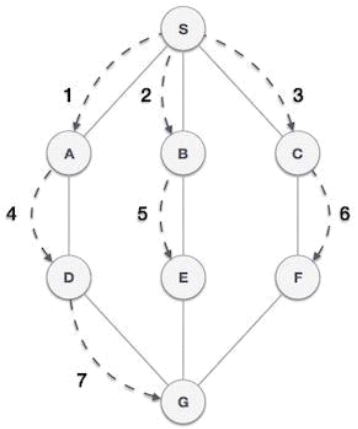
```
DFS(G,v)
init() {
  For each u ∈ G
  DFS(G, u)
}
```

Application of DFS Algorithm

- For finding the path
- To test if the graph is bipartite
- For finding the strongly connected components of a graph
- For detecting cycles in a graph

Breadth First Search

Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

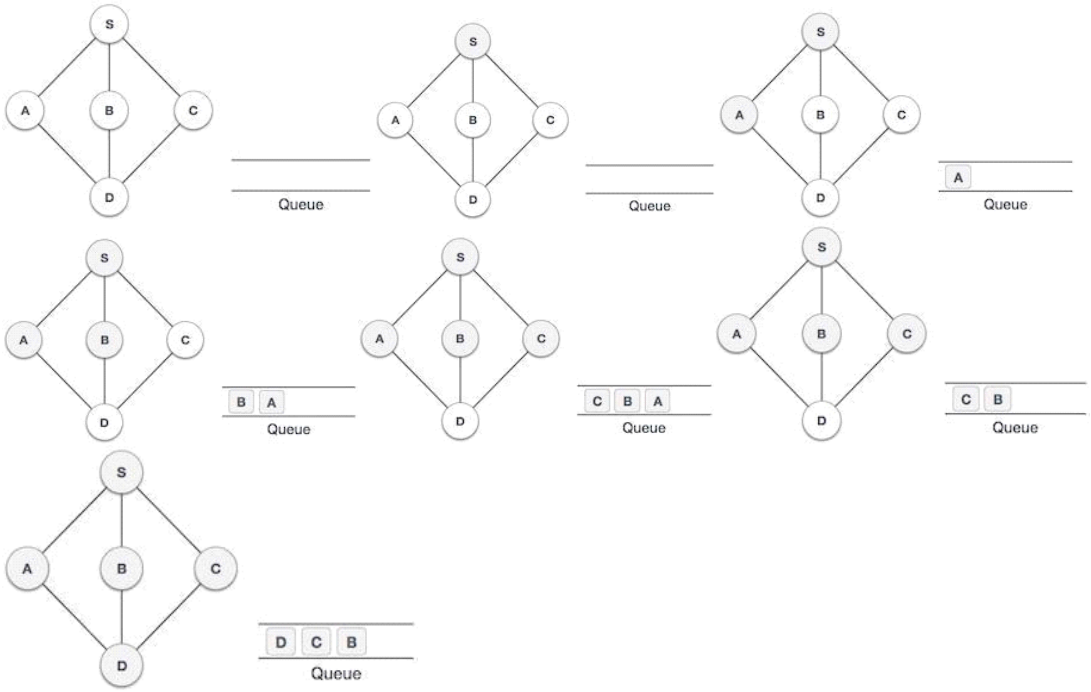
Rule 1 — Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.

Rule 2 — If no adjacent vertex is found, remove the first vertex from the queue.

Rule 3 — Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal		Description
1			
			Initialize the queue.
2			
			We start from visiting S (starting
			node), and mark it as visited.
3			We then see an unvisited adjacent
			node from S. In this example, we have
			three nodes but alphabetically we
			choose A, mark it as visited and
			enqueue it.
4			
			Next, the unvisited adjacent node
			from S is B. We mark it as visited and
			enqueue it.
5			
			Next, the unvisited adjacent node
			from S is C. We mark it as visited and
			enqueue it.
6			
			Now, S is left with no unvisited
			adjacent nodes. So, we dequeue and
			find A.

7			**From A we have D as**
			unvisited adjacent node. We
			mark it as visited and
			enqueue it.



BFS pseudocode

create a queue Q

mark v as visited and put v into Q

while Q is non-empty

  remove the head u of Q

  mark and enqueue all (unvisited) neighbours of u

BFS Algorithm Complexity

The time complexity of the BFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is O(V).

BFS Algorithm Applications

- To build index by search index
- For GPS navigation
- Path finding algorithms
- In Ford-Fulkerson algorithm to find maximum flow in a network
- Cycle detection in an undirected graph
- In minimum spanning tree

Connected graph , Strongly connected and Bi-Connectivity

Connected Graph Component

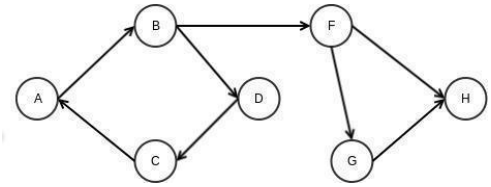
A connected component or simply component of an undirected graph is a subgraph in which each pair of nodes is connected with each other via a path.

```
Component_Count = 0;
for each vertex k ∈ V do
    Visited[k] = False;
end
for each vertex k ∈ V do
    if Visited[k] == False then
        DFS(V,k);
        Component_Count = Component_Count + 1;
    end
end
Print Component_Count;
Procedure DFS(V,k)
Visited[k] = True;
for each vertex p ∈ V.Adj[k] do
    if Visited[p] == False then
        DFS(V,p);
    end
end
```

Strongly Connected Graph

The **Kosaraju algorithm** is a DFS based algorithm used to find Strongly Connected

Components**(SCC)** in a graph. It is based on the idea that if one is able to reach a vertex v starting from vertex *u*, then one should be able to reach vertex *u* starting from vertex *v* and if such is the case, one can say that vertices *u* and *v* are **strongly connected** - they are in a strongly connected sub-graph.

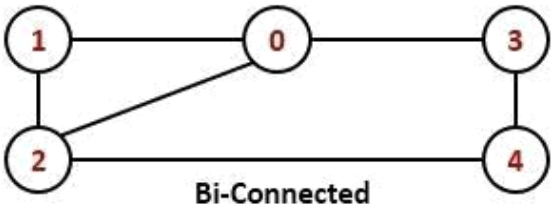



```
stack STACK

void DFS(int source) {

   visited[s]=true

   for all neighbours X of source that are not visited:

      DFS(X)

   STACK.push(source)

}

CLEAR ADJACENCY_LIST

for all edges e:

   first = one end point of e

   second = other end point of e

   ADJACENCY_LIST[second].push(first)

while STACK is not empty:

   source = STACK.top()
```

```
STACK.pop()
if source is visited :
    continue
else :
    DFS(source)
```

Bi Connectivity Graph

An undirected graph is said to be a biconnected graph, if there are two vertex-disjoint paths between any two vertices are present. In other words, we can say that there is a cycle between any two vertices.



We can say that a graph G is a bi-connected graph if it is connected, and there are no articulation points or cut vertex are present in the graph.

To solve this problem, we will use the DFS traversal. Using DFS, we will try to find if there is any articulation point is present or not. We also check whether all vertices are visited by the DFS or not, if not we can say that the graph is not connected.

Pseudocode for Bi connectivity

isArticulation(start, visited, disc, low, parent)

```
Begin
  time := 0 //the value of time will not be initialized for next function calls
  dfsChild := 0
```

```
  mark start as visited
  set disc[start] := time+1 and low[start] := time + 1
  time := time + 1
  for all vertex v in the graph G, do
    if there is an edge between (start, v), then
      if v is visited, then
        increase dfsChild
        parent[v] := start
        if isArticulation(v, visited, disc, low, parent) is true, then
          return ture
        low[start] := minimum of low[start] and low[v]
        if parent[start] is ϕ AND dfsChild > 1, then
          return true
        if parent[start] is ϕ AND low[v] >= disc[start], then
          return true
      else if v is not the parent of start, then
        low[start] := minimum of low[start] and disc[v]
  done
  return false
End
```

isBiconnected(graph)

```
Begin
  initially set all vertices are unvisited and parent of each vertices are ϕ if
  isArticulation(0, visited, disc, low, parent) = true, then
    return false
```

```
  for each node i of the graph, do
    if i is not visited, then
      return false
  done
  return true
End
```

Minimum Spanning Tree

A Spanning Tree is a tree which have V vertices and V-1 edges. All nodes in a spanning tree are reachable from each other.

A Minimum Spanning Tree(MST) or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree having a weight less than or equal to the weight of every other possible spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree. In short out of all spanning trees of a given graph, the spanning tree having minimum weight is MST.

Algorithms for finding Minimum Spanning Tree(MST):-

- Prim's Algorithm
- Kruskal's Algorithm

Prim's Algorithm

Prim's algorithm is a minimum spanning tree HYPERLINK "https://www.programiz.com/dsa/spanning-tree-and-minimum-spanning-tree" HYPERLINK "https://www.programiz.com/dsa/spanning-tree-and-minimum-spanning-tree" HYPERLINK "https://www.programiz.com/dsa/spanning-tree-and-minimum-spanning-tree" algorithm that takes a graph as input and finds the subset of the edges of that graph which

form a tree that includes every vertex

has the minimum sum of weights among all the trees that can be formed from the graph

How Prim's algorithm works

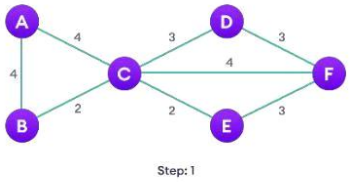
It falls under a class of algorithms called greedy algorithms HYPERLINK "https://www.programiz.com/dsa/greedy-algorithm" HYPERLINK "https://www.programiz.com/dsa/greedy-algorithm" HYPERLINK "https://www.programiz.com/dsa/greedy-algorithm" that find the local optimum in the hopes of finding a global optimum.

We start from one vertex and keep adding edges with the lowest weight until we reach our goal.

The steps for implementing Prim's algorithm are as follows:

- Initialize the minimum spanning tree with a vertex chosen at random.
- Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
- Keep repeating step 2 until we get a minimum spanning tree

Example of Prim's algorithm

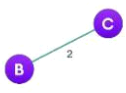


Step: 1

Start with a weighted graph

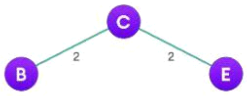
Step: 2

Choose a vertex



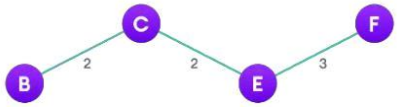
Step: 3

Choose the shortest edge from this vertex and add it



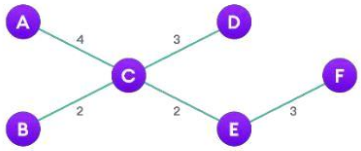
Step: 4

Choose the nearest vertex not yet in the solution



Step: 5

Choose the nearest edge not yet in the solution, if there are multiple choices, choose one at random



Step: 6

Prim's Algorithm pseudocode

The pseudocode for prim's algorithm shows how we create two sets of vertices U and V-U. U contains the list of vertices that have been visited and V-U the list of vertices that haven't. One by one, we move vertices from set V-U to set U by connecting the least weight edge.

```
T = ;
    {1};
while (U ≠ V)
    let (u, v) be the lowest cost edge such that u U and v V-U;
    T = T {(u, v)}
    U=U {v}
```

Prim's Algorithm Complexity

The time complexity of Prim's algorithm is O(E log V).

Kruskal Algorithm

Kruskal's algorithm is a minimum spanning tree HYPERLINK "https://www.programiz.com/dsa/spanning-tree-and-minimum-spanning-tree" HYPERLINK "https://www.programiz.com/dsa/spanning-tree-and-minimum-spanning-tree" HYPERLINK "https://www.programiz.com/dsa/spanning-tree-and-minimum-spanning-tree" algorithm that takes a graph as input and finds the subset of the edges of that graph which

form a tree that includes every vertex

has the minimum sum of weights among all the trees that can be formed from the graph

How Kruskal's algorithm works

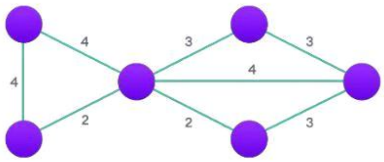
It falls under a class of algorithms called greedy algorithms HYPERLINK "http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Greedy/greedyIntro.htm" HYPERLINK "http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Greedy/greedyIntro.htm" HYPERLINK "http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Greedy/greedyIntro.htm" that find the local optimum in the hopes of finding a global optimum.

We start from the edges with the lowest weight and keep adding edges until we reach our goal.
The steps for implementing Kruskal's algorithm are as follows:

- Sort all the edges from low weight to high

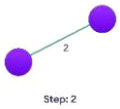
- Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
- Keep adding edges until we reach all vertices.

Example of Kruskal's algorithm



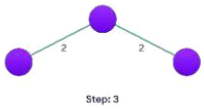
Step: 1

Start with a weighted graph



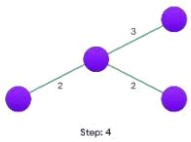
Step: 2

Choose the edge with the least weight, if there are more than 1, choose anyone



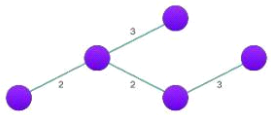
Step: 3

Choose the next shortest edge and add it



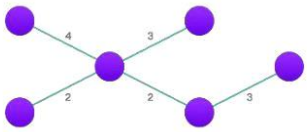
Step: 4

Choose the next shortest edge that doesn't create a cycle and add it



Step: 5

Choose the next shortest edge that doesn't create a cycle and add it



Step: 6

Repeat until you have a spanning tree

Kruskal Algorithm Pseudocode

KRUSKAL(G):

A =			G.V:
	∅	∈	

MAKE-SET(v)		

For each edge (u, v) G.E ordered by increasing order by weight(u, v):

if FIND-SET(u) ≠	
A = A {(u, v)}	
return A	

Shortest Path Algorithm

The shortest path problem is about **finding a path between vertices in a graph such that the total sum of the edges weights is minimum**.

Algorithm for Shortest Path

- Bellman Algorithm
- Dijkstra Algorithm
- Floyd Warshall Algorithm

Bellman Algorithm

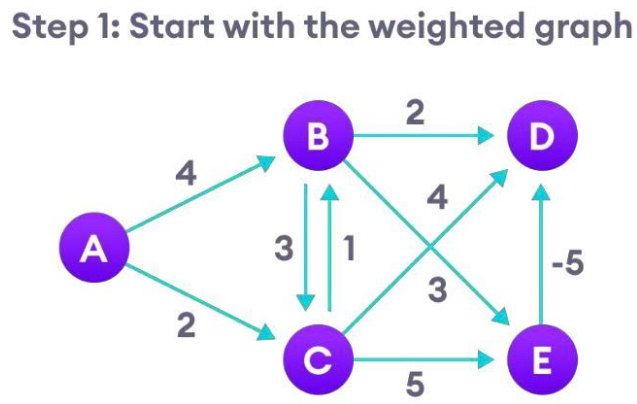
Bellman Ford algorithm helps us find the shortest path from a vertex to all other vertices of a weighted graph.

It is similar to Dijkstra's algorithm but it can work with graphs in which edges can have negative weights.

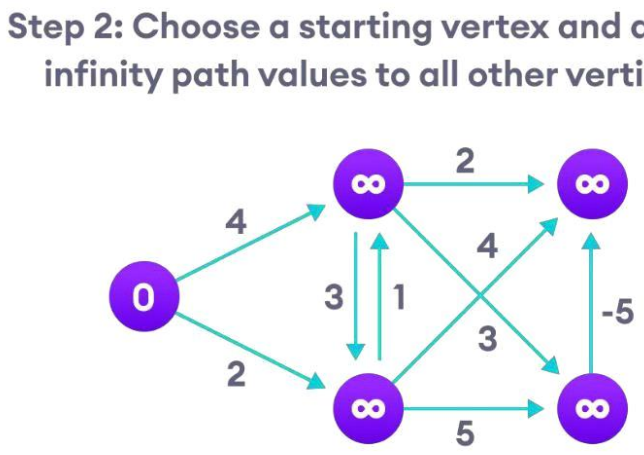
How Bellman Ford's algorithm works

Bellman Ford algorithm works by overestimating the length of the path from the starting vertex to all other vertices. Then it iteratively relaxes those estimates by finding new paths that are shorter than the previously overestimated paths.

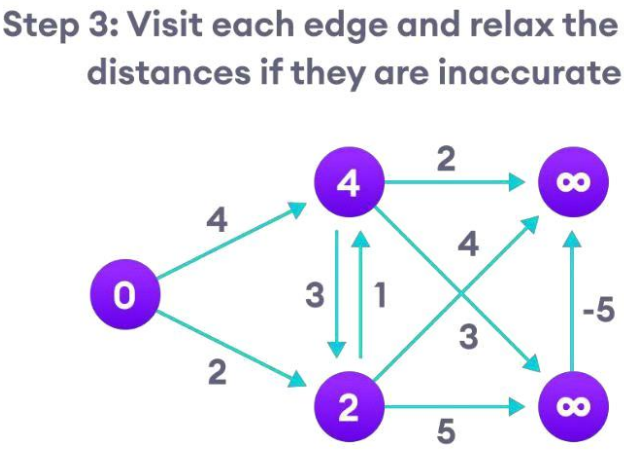
By doing this repeatedly for all vertices, we can guarantee that the result is optimized.



Step-1 for Bellman Ford's algorithm

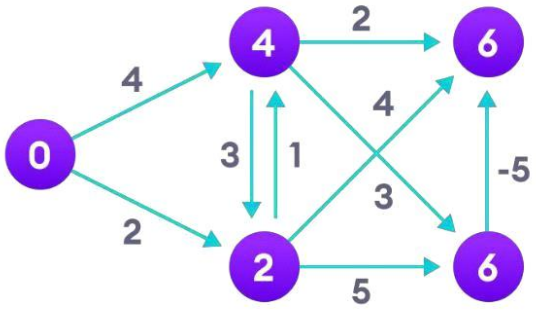


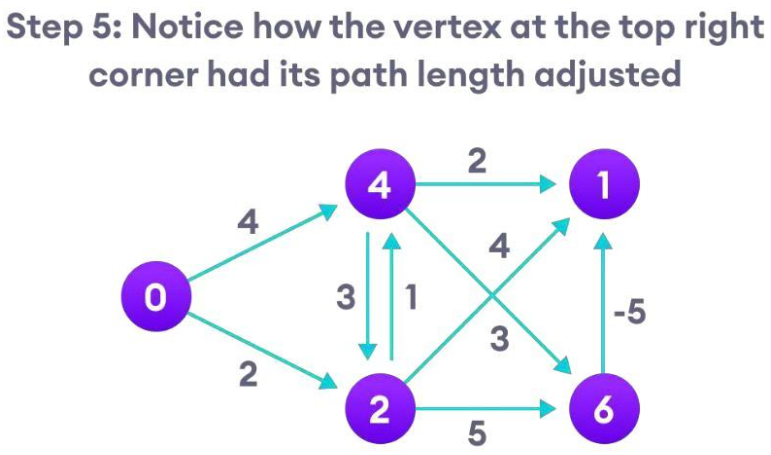
Step-2 for Bellman Ford's algorithm



Step-3 for Bellman Ford's algorithm

Step 4: We need to do this V times because in the worst case, a vertex's path length might need to be readjusted V times





Step-4 for Bellman Ford's algorithm

Step 6: After all the vertices have their path lengths, we check if a negative cycle is present

	B	C	D	E
0	∞	∞	∞	∞
0	4	2	∞	∞
0	3	2	6	6
0	3	2	1	6
0	3	2	1	6

Step-5 for Bellman Ford's algorithm

Step-6 for Bellman Ford's algorithm

Bellman Ford Pseudocode

We need to maintain the path distance of every vertex. We can store that in an array of size v, where v is the number of vertices.

We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.

Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.

```
function bellmanFord(G, S)
  for each vertex V in G
    distance[V] <- infinite
      previous[V] <- NULL
  distance[S] <- 0
  for each vertex V in G
    for each edge (U,V) in G
      tempDistance <- distance[U] + edge_weight(U, V)
      if tempDistance < distance[V]
        distance[V] <- tempDistance
        previous[V] <- U
  for each edge (U,V) in G
    If distance[U] + edge_weight(U, V) < distance[V}
      Error: Negative Cycle Exists
```

return distance[], previous[]	
Bellman Ford's Complexity	
Time Complexity	
Best Case Complexity	O(E)
Average Case Complexity	O(VE)
Worst Case Complexity	O(VE)

Dijkstra Algorithm

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.

It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

How Dijkstra's Algorithm works

Dijkstra's Algorithm works on the basis that any subpath B -> D of the shortest path A -> D between vertices A and D is also the shortest path between vertices B and D.

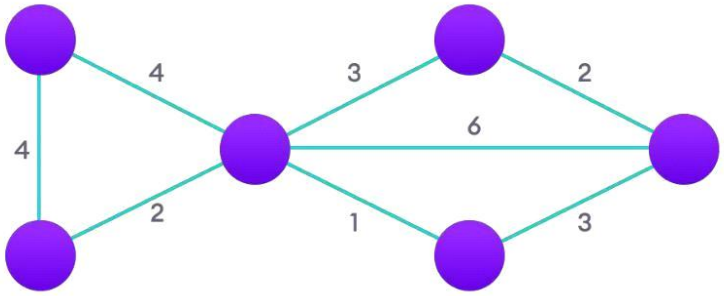
Each subpath is the shortest path

Djikstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbors to find the shortest subpath to those neighbors.

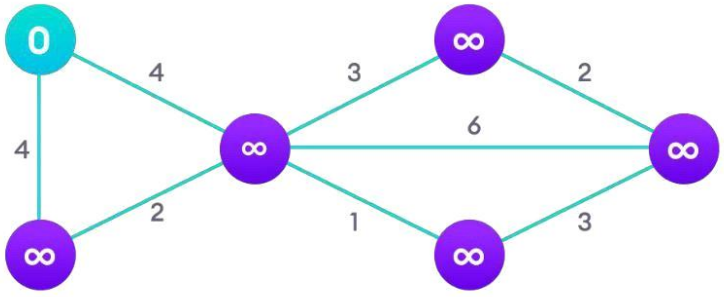
The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

Example of Dijkstra's algorithm

It is easier to start with an example and then think about the algorithm.

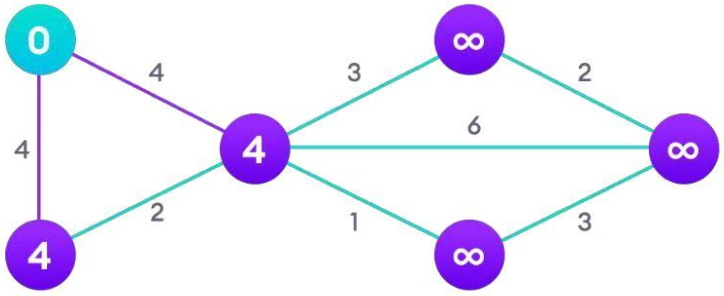


Step: 1



Step: 2

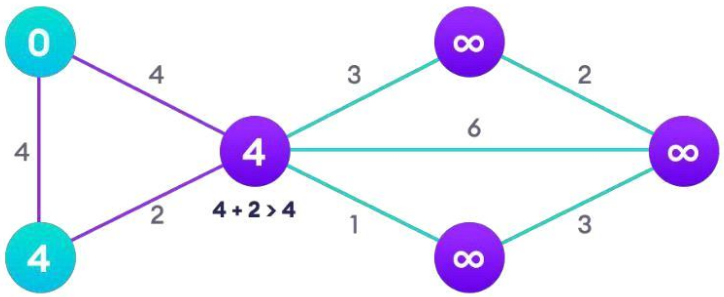
Start with a weighted graph



Step: 3

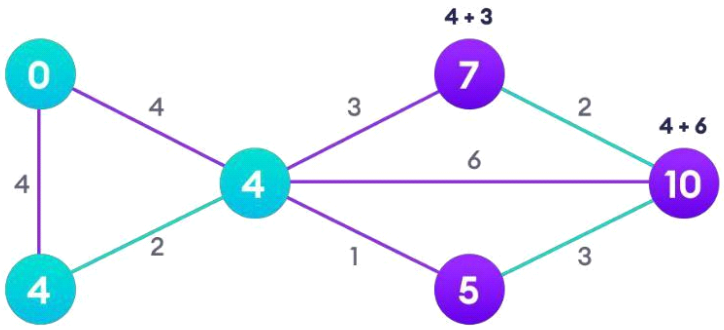
Choose a starting vertex and assign infinity path values to all other devices

Go to each vertex and update its path length



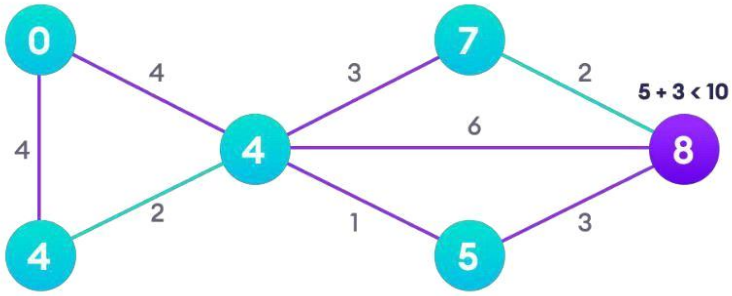
Step: 4

If the path length of the adjacent vertex is lesser than new path length, don't update it



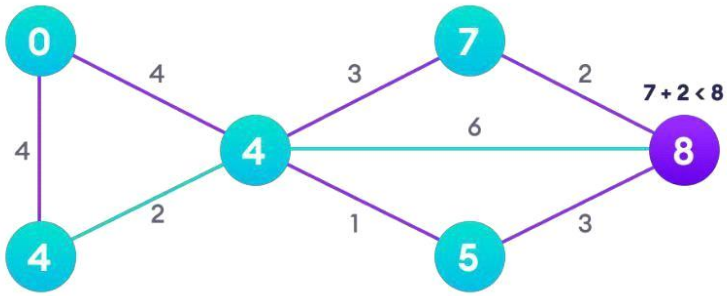
Step: 5

Avoid updating path lengths of already visited vertices



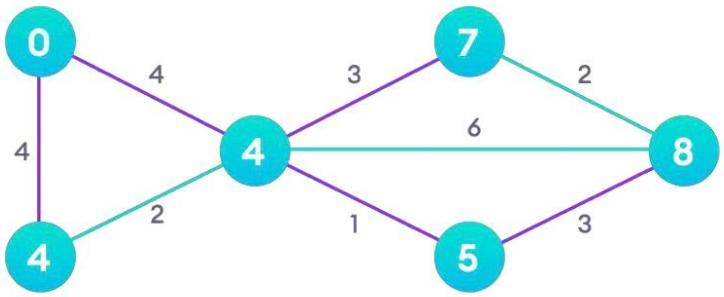
Step: 6

After each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7



Step: 7

Notice how the rightmost vertex has its path length updated twice



Step: 8

Repeat until all the vertices have been visited

Djikstra's algorithm pseudocode

We need to maintain the path distance of every vertex. We can store that in an array of size v, where v is the number of vertices.

We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.

Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.

A minimum priority queue can be used to efficiently receive the vertex with least path distance.

```
function dijkstra(G, S)
   for each vertex V in G
      distance[V] <- infinite
      previous[V] <- NULL
      If V != S, add V to Priority Queue Q
   distance[S] <- 0
   while Q IS NOT EMPTY
      U <- Extract MIN from Q
      for each unvisited neighbour V of U
         tempDistance <- distance[U] + edge_weight(U, V)
         if tempDistance < distance[V]
            distance[V] <- tempDistance
            previous[V] <- U
   return distance[], previous[]
```

Dijkstra's Algorithm Complexity

Time Complexity: O(E Log V)

where, E is the number of edges and V is the number of vertices.

Space Complexity: O(V)

Floyd Warshall Algorithm

Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).

A weighted graph is a graph in which each edge has a numerical value associated with it. Floyd-Warhshall algorithm is also called as Floyd's algorithm, Roy-Floyd algorithm, Roy-Warshall algorithm, or WFI algorithm.

This algorithm follows the dynamic programming HYPERLINK "https://www.programiz.com/dsa/dynamic-programming" HYPERLINK "https://www.programiz.com/dsa/dynamic-programming" HYPERLINK "https://www.programiz.com/dsa/dynamic-programming" approach to find the shortest paths.

How Floyd-Warshall Algorithm Works?

Let the given graph be:

Initial graph

Follow the steps below to find the shortest path between all the pairs of vertices.

- Create a matrix A^0 of dimension n*n where n is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph.

  Each cell A[i][j] is filled with the distance from the i^{th} vertex to the j^{th} vertex. If there is no path from i^{th} vertex to j^{th} vertex, the cell is left as infinity.

  Fill each cell with the distance between ith and jth vertex

- Now, create a matrix A^1 using matrix A^0. The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.
  Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex. A[i][j] is filled with (A[i][k] + A[k][j]) if (A[i][j] > A[i][k] + A[k][j]).

$A^0 =$

	1	2	3	4
1	0	3	∞	5
2	2	0	∞	4
3	∞	1	0	∞
4	∞	∞	2	0

That is, if the direct distance from the source to the destination is greater than the path h the vertex k, then the cell is filled with A[i][k] + A[k][j].

In this step, k is vertex 1. We calculate the distance from source vertex to destination vertex through this vertex k.Calcula te the distance from the source vertex to destination vertex through this vertex k

For example: For $A^1[2, 4]$, the direct distance from vertex 2 to 4 is 4 and the sum of the distance from vertex 2 to 4 through vertex (ie. from vertex 2 to 1 and from vertex 1 to 4) is 7. Since 4 < 7, $A^0[2, 4]$ is filled with 4.

- Similarly, A^2 is created using A^1. The elements in the second column and the second row are left as they are.

  In this step, k is the second vertex (i.e. vertex 2). The remaining steps are the same as in **step 2**.Calcula te the distance from the source vertex to destination vertex through this vertex 2

4. Similarly, A^3 and A^4 is also created.
   Calculat e the distance from the source vertex to destination vertex through this vertex C alculate the distance from the source vertex to destination vertex through this vertex 4

5. A^4 gives the shortest path between each pair of vertices.

Floyd-Warshall Algorithm

n = no of vertices

A = matrix of dimension n*n

```
for k = 1 to n
   for i = 1 to n
      for j = 1 to n
         A^k[i, j] = min (A^k-1[i, j], A^k-1[i, k] + A^k-1[k, j])
return A
```

Time Complexity

There are three loops. Each loop has constant complexities. So, the time complexity of the Floyd-Warshall algorithm is $O(n^3)$.

Network Flow

Flow Network is a directed graph that is used for modeling material Flow. There are two different vertices; one is a **source** which produces material at some steady rate, and another one is sink which consumes the content at the same constant speed. The flow of the material at any mark in the system is the rate at which the element moves.

Some real-life problems like the flow of liquids through pipes, the current through wires and delivery of goods can be modelled using flow networks.

Definition: A Flow Network is a directed graph G = (V, E) such that

1. For each edge (u, v) E, we associate a nonnegative weight capacity c (u, v) ≥ 0.If (u, v) ∉ E,
2. There are two distinguishing points, the source s, and the sink t;
3. For every vertex v ∈ V, there is a path from s to t containing v.

Let G = (V, E) be a flow network. Let s be the source of the network, and let t be the sink. A flow in G is a real-valued function f: V x V→R such that the following properties hold: Play Video

- **Capacity Constraint:** For all u, v V, we need f (u, v) ≤ c (u, v).
- **Skew Symmetry:** For all u, v ∈ V, we need f (u, v) = −f (v, u)
- **Flow Conservation:** For all u V-{s, t}, we need

$$\sum_{v \in V} f(u,v) = \sum_{u \in V} f(u,v) = 0$$

The quantity f (u, v), which can be positive or negative, is known as the net flow from vertex u to vertex v. In the **maximum-flow problem**, we are given a flow network G with source s and sink t, and

a flow of maximum value from s to t.

Ford-Fulkerson Algorithm

Initially, the flow of value is 0. Find some augmenting Path p and increase flow f on each edge of p by residual Capacity c_f (p). When no augmenting path exists, flow f is a maximum flow.

FORD-FULKERSON METHOD (G, s, t)

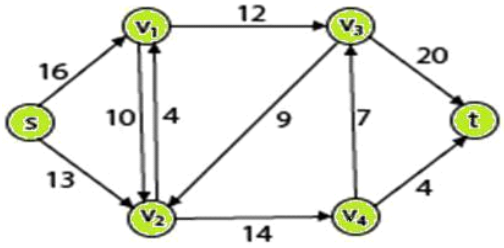
- Initialize flow f to 0
- while there exists an augmenting path p
- do argument flow f along p
- Return f

FORD-FULKERSON (G, s,∈ t)

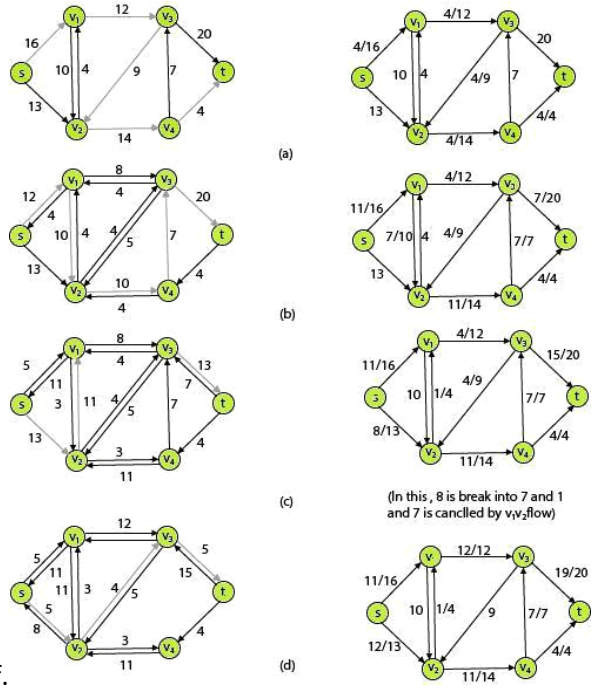
1. for each edge (u, v) E [G]

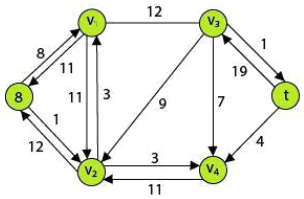
- do f [u, v] ← 0
- f [u, v] ← 0
- while there exists a path p from s to t in the residual network G_f.
- do c_f (p)←min?{ C_f (u,v):(u,v)is on p}
- for each edge (u, v) in p
- do f [u, v] ← f [u, v] + c_f (p)
- f [u, v] ←-f[u,v]

Example: Each Directed Edge is labeled with capacity. Use the Ford-Fulkerson algorithm to find the maximum flow.



Solution: The left side of each part shows the residual network G_f with a shaded augmenting path p,and the right side of each part shows the net flow f.



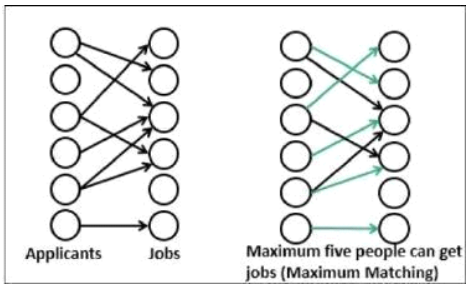


(e)

Now , it has no augmenting paths .So, the maximum flow shown in (d) is 23 is a maximum flow .

Maximum Bipartite Matching

The bipartite matching is a set of edges in a graph is chosen in such a way, that no two edges in that set will share an endpoint. The maximum matching is matching the maximum number of edges.



When the maximum match is found, we cannot add another edge. If one edge is added to the maximum matched graph, it is no longer a matching. For a bipartite graph, there can be more than one maximum matching is possible.

Algorithm

bipartiteMatch(u, visited, assign)

Input: Starting node, visited list to keep track, assign the list to assign node with another node.

Output — Returns true when a matching for vertex u is possible.

```
Begin
  for all vertex v, which are adjacent with u, do
    if v is not visited, then
      mark v as visited
      if v is not assigned, or bipartiteMatch(assign[v], visited, assign) is true, then
        assign[v] := u
        return true
  done
  return false
End
```

maxMatch(graph)

Input — The given graph.

Output — The maximum number of the match.

```
Begin
  initially no vertex is assigned
  count := 0
  for all applicant u in M, do
    make all node as unvisited
```

```
    if bipartiteMatch(u, visited, assign), then
      increase count by 1
  done
End
```

Unit 3

Divide and Conquer Algorithm

A divide and conquer algorithm is a strategy of solving a large problem by

- breaking the problem into smaller sub-problems
- solving the sub-problems, and
- combining them to get the desired output.

To use the divide and conquer algorithm, recursion is used.

How Divide and Conquer Algorithms Work?

Here are the steps involved:

- Divide: Divide the given problem into sub-problems using recursion.
- Conquer: Solve the smaller sub-problems recursively. If the subproblem is small enough, then solve it directly.
- Combine: Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.

Finding Maximum and Minimum

To find the maximum and minimum numbers in a given array *numbers[]* of size n, the following algorithm can be used. First we are representing the naive method and then we will present divide and conquer approach.

Naïve Method

Naïve method is a basic method to solve any problem. In this method, the maximum and minimum number can be found separately. To find the maximum and minimum numbers, the following straightforward algorithm can be used.

Algorithm: Max-Min-Element (numbers[])

max := numbers[1]

```
min := numbers[1]

for i = 2 to n do

  if numbers[i] > max then

    max := numbers[i]

  if numbers[i] < min then

    min := numbers[i]

return (max, min)
```

Analysis

The number of comparison in Naive method is 2n - 2.

The number of comparisons can be reduced using the divide and conquer approach.

Following is the technique.

Divide and Conquer Approach

In this approach, the array is divided into two halves. Then using recursive approach maximum and minimum numbers in each halves are found. Later, return the maximum of two maxima of each half and the minimum of two minima of each half.

In this given problem, the number of elements in an array is y−x+1 , where y is greater than or equal to x.

Max−Min(x,y) will return the maximum and minimum values of an array numbers[x...y].

Algorithm: Max - Min(x, y)

```
if y – x ≤ 1 then

  return (max(numbers[x], numbers[y]), min((numbers[x], numbers[y]))

else
  (max1, min1):= maxmin(x, ((x + y)/2) )
  (max2, min2):= maxmin(
```

Analysis

Let *T(n)* be the number of comparisons made by Max−Min(x,y), where the number of elements n=y−x+1.

If *T(n)* represents the numbers, then the recurrence relation can be represented as

$$T(n) = \begin{cases} T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + 2 & for\ n > 2 \\ 1 & for\ n = 2 \\ 0 & for\ n = 1 \end{cases}$$

Let us assume that *n* is in the form of power of 2. Hence, $n = 2^k$ where k is height of the recursion tree.

So,

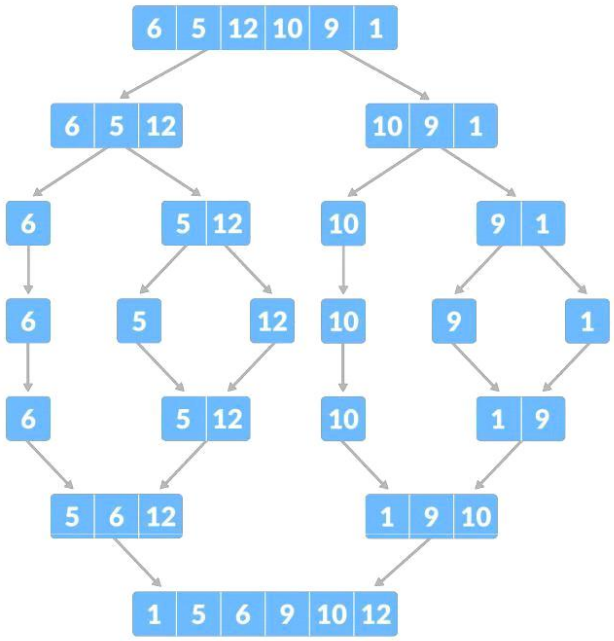
$$T(n) = 2.T(\frac{n}{2}) + 2 = 2.\left(2.T(\frac{n}{4}) + 2\right) + 2.... = \frac{3n}{2} - 2$$

Compared to Naïve method, in divide and conquer approach, the number of comparisons is less. However, using the asymptotic notation both of the approaches are represented by O(n).

Merge Sort

Merge Sort is one of the most popular sorting algorithms that is based on the principle of Divide and Conquer Algorithm.

Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution.



Merge Sort example

Divide and Conquer Strategy

Using the **Divide and Conquer** technique, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem.

Suppose we had to sort an array A. A subproblem would be to sort a sub-section of this array starting at index p and ending at index r, denoted as A[p..r].

Divide

If q is the half-way point between p and r, then we can split the subarray A[p..r] into two arrays A[p..q] and A[q+1, r].

Conquer

In the conquer step, we try to sort both the subarrays A[p..q] and A[q+1, r]. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

Combine

When the conquer step reaches the base step and we get two sorted

subarrays A[p..q] and A[q+1, r] for array A[p..r], we combine the results by creating a sorted array A[p..r] from two sorted subarrays A[p..q] and A[q+1, r].

MergeSort Algorithm

The MergeSort function repeatedly divides the array into two halves until we reach a stage where we try to perform MergeSort on a subarray of size 1 i.e. p == r.

After that, the merge function comes into play and combines the sorted arrays into larger arrays until the whole array is merged.

```
MergeSort(A, p, r):
   if p > r
      return
   q = (p+r)/2
   mergeSort(A, p, q)
   mergeSort(A, q+1, r)
   merge(A, p, q, r)
```

```
void merge(int arr[], int p, int q, int r)
{
```

- Create L ← A[p..q] and M ← A[q+1..r] int

```
   n1 = q - p + 1;
   int n2 = r - q; int
   L[n1], M[n2];
   for (int i = 0; i < n1; i++)
      L[i] = arr[p + i];
   for (int j = 0; j < n2; j++)
      M[j] = arr[q + 1 + j];
```

- Maintain current index of sub-arrays and main array int

```
   i, j, k;
   i = 0; j =
   0; k = p;
```

- Until we reach either end of either L or M, pick larger among
- elements L and M and place them in the correct position at A[p..r]

```
   while (i < n1 && j < n2)
         {
```

```
        if (L[i] <= M[j])
                {
                        arr[k] = L[i];

                        i++;
                 }
        else
                {
                         arr[k] = M[j];
                         j++;
                }
        k++;
 }
```

- When we run out of elements in either L or M,
- pick up the remaining elements and put in A[p..r]

```
        while (i < n1)

                {
                        arr[k] = L[i];
                        i++;

                        k++;
                 }

        while (j < n2)
                {
                         arr[k] = M[j];
                         j++;
                         k++;

                 }
}
```

Time Complexity

Best Case Complexity: O(n*log n)

Worst Case Complexity: O(n*log n)

Average Case Complexity: O(n*log n)

Dynamic Programming

Matrix Chain Multiplication

Dynamic programming is a method for solving optimization problems.

It is algorithm technique to solve a complex and overlapping sub-problems. Compute the solutions to the sub-problems once and store the solutions in a table, so that they can be reused (repeatedly) later.

Dynamic programming is more efficient then other algorithm methods like as Greedy method, Divide and Conquer method, Recursion method, etc….

The real time many of problems are not solve using simple and traditional approach methods. like as coin change problem , knapsack problem, Fibonacci sequence generating , complex matrix multiplication….To solve using Iterative formula, tedious method , repetition

again and again it become a more time consuming and foolish. some of the problem it should be necessary to divide a sub problems and compute its again and again to solve a

such kind of problems and give the optimal solution , effective solution the Dynamic programming is needed…

Basic Features of Dynamic programming :-

Get all the possible solution and pick up best and optimal solution.

Work on principal of optimality.

Define sub-parts and solve them using recursively. Less

space complexity But more Time complexity.

Dynamic programming saves us from having to recompute previously calculated sub-solutions.

Difficult to understanding.

We are covered a many of the real world problems.In our day to day life when we do making coin change, robotics world, aircraft, mathematical problems like Fibonacci sequence, simple matrix multiplication of more then two matrices and its multiplication possibility is many more so in that get the best and optimal solution. NOW we can look about one problem that is ***MATRIX CHAIN MULTIPLICATION PROBLEM.***

Suppose, We are given a sequence (chain) ***(A1, A2……An)*** of n matrices to be multiplied, and

we wish to compute the product ***(A1A2…..An)***.We can evaluate the above expression using

the standard algorithm for multiplying pairs of matrices as a subroutine once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together. Matrix multiplication is associative, and so all parenthesizations yield the same product. For example, if the chain of matrices is ***(A1, A2, A3, A4)*** then we can fully parenthesize the product (***A1A2A3A4)*** in five distinct ways:

1:-(A1(A2(A3A4))) ,

2:-(A1((A2A3)A4)),

3:- ((A1A2)(A3A4)),

4:-((A1(A2A3))A4),

5:-(((A1A2)A3)A4) .

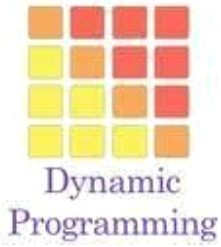
We can multiply two matrices A and B only if they are compatible. the number of columns of A must equal the number of rows of B. If A is a p x q matrix and B is a q x r matrix,the resulting matrix C is a p x r matrix. The time to compute C is dominated by the number of scalar multiplications is pqr. we shall express costs in terms of the number of scalar multiplications.For example, if we have three matrices (A1,A2,A3) and its cost is

(10x100),(100x5),(5x500) respectively. so we can calculate the cost of scalar multiplication is 10*100*5=5000 if ((A1A2)A3), 10*5*500=25000 if (A1(A2A3)), and so on cost calculation. ***Note that in the matrix-chain multiplication problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest cost.***that is here is minimum cost is 5000 for above example .So problem is we can perform a many time of cost multiplication and repeatedly the calculation is

performing. so this general method is very time consuming and tedious.So we can apply ***dynamic programming*** for solve this kind of problem.

when we used the Dynamic programming technique we shall follow some steps.

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution.
- Construct an optimal solution from computed information.



we have matrices of any of order. our goal is find optimal cost multiplication of matrices.when we solve the this kind of problem using DP step 2 we can get

m[i , j] = min { m[i , k] + m[i+k , j] + pi-1*pk*pj } if i < j…. where p is dimension of matrix , i ≤ k < j …..

The basic algorithm of matrix chain multiplication:-

- Matrix A[i] has dimension dims[i-1] x dims[i] for i = 1..n

MatrixChainMultiplication(int dims[])

{

- length[dims] = n + 1

n = dims.length - 1;

- m[i,j] = Minimum number of scalar multiplications(i.e., cost)
- needed to compute the matrix A[i]A[i+1]...A[j] = A[i..j]
- The cost is zero when multiplying one matrix

for (i = 1; i <= n; i++)

m[i, i] = 0;

for (len = 2; len <= n; len++){

// Subsequence lengths

for (i = 1; i <= n - len + 1; i++) {

j = i + len - 1;

m[i, j] = MAXINT;

for (k = i; k <= j - 1; k++) {

cost = m[i, k] + m[k+1, j] + dims[i-1]*dims[k]*dims[j];

if (cost < m[i, j]) {

m[i, j] = cost;

s[i, j] = k;

- Index of the subsequence split that achieved minimal cost

}

}

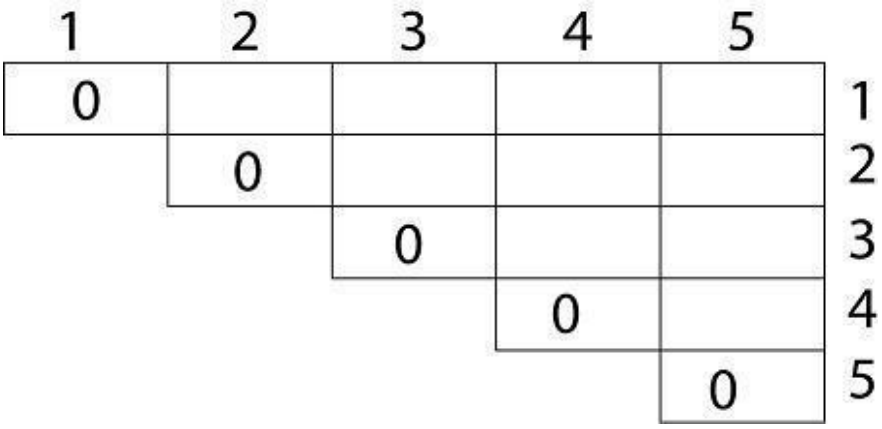
}

}

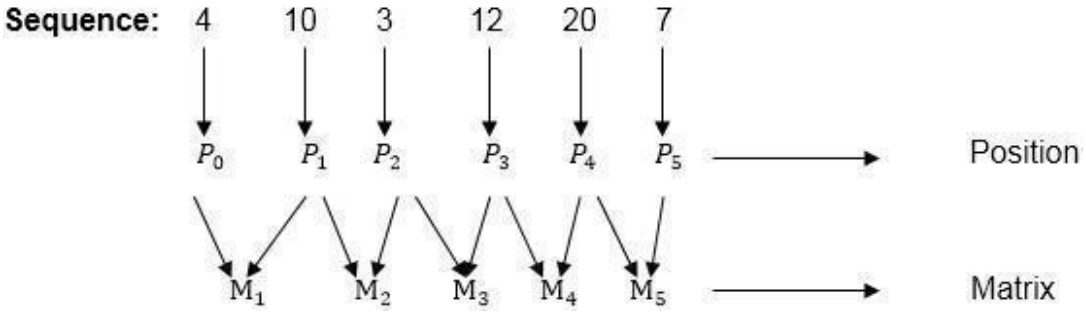
}

Example of Matrix Chain Multiplication

Example: We are given the sequence {4, 10, 3, 12, 20, and 7}. The matrices have size 4 x 10, 10 x 3, 3 x 12, 12 x 20, 20 x 7. We need to compute M [i,j], $0 \leq i, j \leq 5$. We know M [i, i] = 0 for all i.



Let us proceed with working away from the diagonal. We compute the optimal solution for the product of 2 matrices.



In Dynamic Programming, initialization of every method done by ‘0’.So we initialize it by ‘0’.It will sort out diagonally.

We have to sort out all the combination but the minimum output combination is taken into consideration.

Calculation of Product of 2 matrices:

- m (1,2) = m1 x m2
    - 4 x 10 x 10 x 3
    - 4 x 10 x 3 = 120
- m (2, 3) = m2 x m3
    - 10 x 3 x 3 x 12
    - 10 x 3 x 12 = 360
- m (3, 4) = m3 x m4
    - 3 x 12 x 12 x 20
    - 3 x 12 x 20 = 720
- m (4,5) = m4 x m5
    - 12 x 20 x 20 x 7
    - 12 x 20 x 7 = 1680

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

We initialize the diagonal element with equal i,j value with '0'.

After that second diagonal is sorted out and we get all the values corresponded to it Now the third diagonal will be solved out in the same way.

Now product of 3 matrices:

M[1,3]=M1M2M3

- There are two cases by which we can solve this multiplication: (M1 x M2) + M3, M1+ (M2x M3)
- After solving both cases we choose the case in which minimum output is there.

$$M[1,3] = \min \begin{cases} M[1,2] + M[3,3] + p_0 p_2 p_3 = 120 + 0 + 4.3.12 = 264 \\ M[1,1] + M[2,3] + p_0 p_1 p_3 = 0 + 360 + 4.10.12 = 840 \end{cases}$$

M [1, 3] =264

As Comparing both output **264** is minimum in both cases so we insert **264** in table and (M1 x M2) + M3 this combination is chosen for the output making.

M[2,4]=M2M3M4

- There are two cases by which we can solve this multiplication: (M2x M3)+M4, M2+(M3 x M4)

- After solving both cases we choose the case in which minimum output is there.

$$M[2,4] = \min \begin{cases} M[2,3] + M[4,4] + p_1p_3p_4 = 360 + 0 + 10.12.20 = 2760 \\ M[2,2] + M[3,4] + p_1p_2p_4 = 0 + 720 + 10.3.20 = 1320 \end{cases}$$

M [2, 4] = 1320

As Comparing both output **1320** is minimum in both cases so we insert **1320** in table and M2+(M3 x M4) this combination is chosen for the output making.

M[3,5]=M3 M4 M5

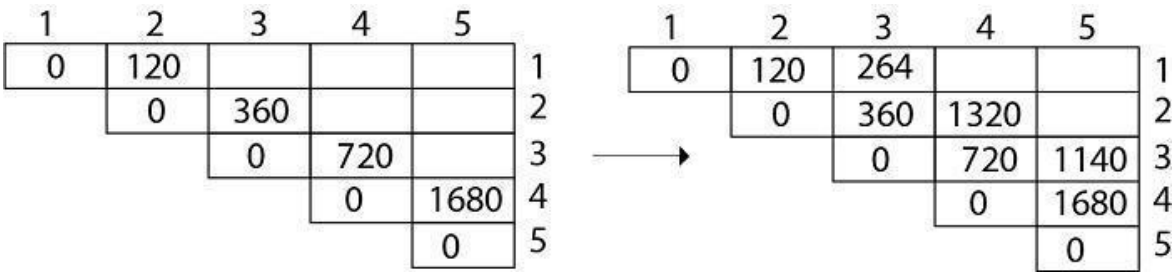
- There are two cases by which we can solve this multiplication: (M3 x M4) + M5, M3+ (M4xM5)

- After solving both cases we choose the case in which minimum output is there.

$$M[3,5] = \min \begin{cases} M[3,4] + M[5,5] + p_2p_4p_5 = 720 + 0 + 3.20.7 = 1140 \\ M[3,3] + M[4,5] + p_2p_3p_5 = 0 + 1680 + 3.12.7 = 1932 \end{cases}$$

M [3, 5] = 1140

As Comparing both output **1140** is minimum in both cases so we insert **1140** in table and (M3 x M4) + M5this combination is chosen for the output making.



Now Product of 4 matrices:

M[1,4]=M1 M2M3M4

There are three cases by which we can solve this multiplication:

- (M1 x M2 x M3) M4
- M1 x(M2 x M3 x M4)
- (M1 xM2) x (M3 x M4)

After solving these cases we choose the case in which minimum output is there

$$M[1,4] = \min \begin{cases} M[1,3] + M[4,4] + p_0p_3p_4 = 264 + 0 + 4.12.20 = 1224 \\ M[1,2] + M[3,4] + p_0p_2p_4 = 120 + 720 + 4.3.20 = 1080 \\ M[1,1] + M[2,4] + p_0p_1p_4 = 0 + 1320 + 4.10.20 = 2120 \end{cases}$$

M [1, 4] =1080

As comparing the output of different cases then '**1080**' is minimum output, so we insert 1080 in the table and (M1 xM2) x (M3 x M4) combination is taken out in output making,

M[2,5]=M2M3M4M5

There are three cases by which we can solve this multiplication:

- (M2 x M3 x M4)x M5
- M2 x(M3 x M4 x M5)
3. (M2 x M3)x (M4 x M5)

After solving these cases we choose the case in which minimum output is there

$$M[2,5] = \min \begin{cases} M[2,4] + M[5,5] + p_1 p_4 p_5 = 1320 + 0 + 10.20.7 = 2720 \\ M[2,3] + M[4,5] + p_1 p_3 p_5 = 360 + 1680 + 10.12.7 = 2880 \\ M[2,2] + M[3,5] + p_1 p_2 p_5 = 0 + 1140 + 10.3.7 = 1350 \end{cases}$$

M [2, 5] = 1350

As comparing the output of different cases then '**1350**' is minimum output, so we insert 1350 in the table and M2 x(M3 x M4xM5)combination is taken out in output making.

Now Product of 5 matrices:

M[1,5]=M1 M2M3M4M5

There are five cases by which we can solve this multiplication:

- (M1 x M2 xM3 x M4)x M5
- M1 x(M2 xM3 x M4 xM5)
- (M1 x M2 xM3)x M4 xM5
- M1 x M2x(M3 x M4 xM5)

After solving these cases we choose the case in which minimum output is there

$$M[1,5] = \min \begin{cases} M[1,4] + M[5,5] + p_0 p_4 p_5 = 1080 + 0 + 4.20.7 = 1544 \\ M[1,3] + M[4,5] + p_0 p_3 p_5 = 264 + 1680 + 4.12.7 = 2016 \\ M[1,2] + M[3,5] + p_0 p_2 p_5 = 120 + 1140 + 4.3.7 = 1344 \\ M[1,1] + M[2,5] + p_0 p_1 p_5 = 0 + 1350 + 4.10.7 = 1630 \end{cases}$$

M [1, 5] = 1344

As comparing the output of different cases then '**1344**' is minimum output, so we insert 1344 in the table and M1 x M2 x(M3 x M4 x M5)combination is taken out in output making.

Final Output is:

1	2	3	4	5	
0	120	264	1080		1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

→

1	2	3	4	5	
0	120	264	1080	1344	1
	0	360	1320	1350	2
		0	720	1140	3
			0	1680	4
				0	5

So we can get the optimal solution of matrices multiplication….

Multi Stage Graph

Multistage Graph problem is defined as follow:

- Multistage graph G = (V, E, W) is a weighted directed graph in which vertices are partitioned into k ≥∈2 disjoint sub∈ sets V = {V_1, V_2, …, V_k} such that if edge (u, v) is present in E then u V_i and v V_{i+1}, 1 ≤ i ≤ k. The goal of multistage graph problem is to find minimum cost path from source to destination vertex.

- The input to the algorithm is a k-stage graph, n vertices are indexed in increasing order of stages.

- The algorithm operates in the backward direction, i.e. it starts from the last vertex of the graph and proceeds in∈a backward direction∈ to find minimum cost path.

- Minimum cost of vertex j V_i from vertex r V_{i+1} is defined as,

Cost[j] = min{ c[j, r] + cost[r] }

where, c[j, r] is the weight of edge <j, r> and cost[r] is the cost of moving from end vertex to vertex r.

- Algorithm for the multistage graph is described below :

Algorithm for Multistage Graph

Algorithm MULTI_STAGE(G, k, n, p)

- Description: Solve multi-stage problem using dynamic programming

- Input:

- Number of stages in graph G = (V, E)

c[i, j]:Cost of edge (i, j)

// Output: p[1:k]:Minimum cost path

cost[n] ← 0

for j ← n – 1 to 1 **do**
  //Let r be a vertex such that (j, r) in E and c[j, r] + cost[r] is minimum

  cost[j] ← c[j, r] + cost[r]

  π[j] ← r

end

//Find minimum cost path

p[1] ← 1

p[k] ← n

for j ← 2 to k - 1 **do**

p[j] ← π[p[j - 1]]

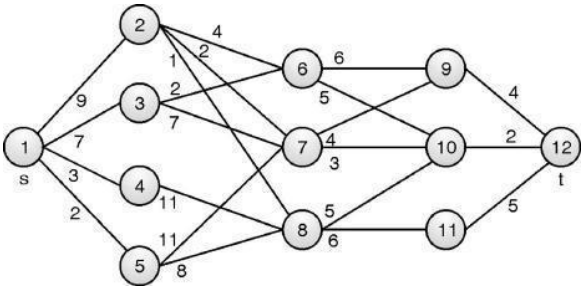
end

Complexity Analysis of Multistage Graph

If graph G has |E| edges, then cost computation time would be O(n + |E|). The complexity of tracing the minimum cost path would be O(k), k < n. Thus total time complexity of multistage graph using dynamic programming would be O(n + |E|).

Example

Example: Find minimum path cost between vertex s and t for following multistage graph using dynamic programming.



Solution:

Solution to multistage graph using dynamic programming is constructed as,

Cost[j] = min{c[j, r] + cost[r]}

Here, number of stages k = 5, number of vertices n = 12, source s = 1 and target t = 12

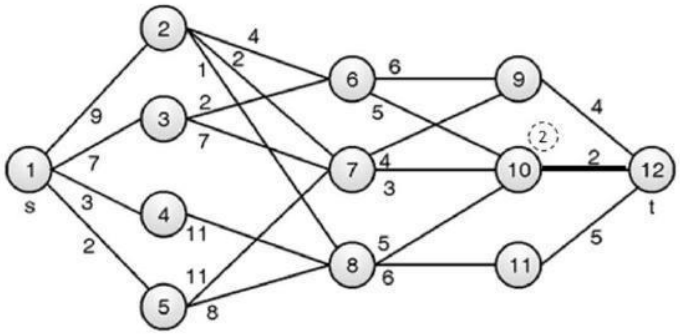
Initialization:

Cost[n] = 0 Cost[12] = 0.

p[1] = s	⇒	p[1] = 1
p[k] = t		p[5] = 12.

r = t = 12.

Stage 4:



Stage 3:

Vertex 6 is connected to vertices 9 and 10:

Cost[6] = min{ c[6, 10] + Cost[10], c[6, 9] + Cost[9] }

• min{5 + 2, 6 + 4} = min{7, 10} = 7

p[6] = 10

Vertex 7 is connected to vertices 9 and 10:

Cost[7] = min{ c[7, 10] + Cost[10], c[7, 9] + Cost[9] }

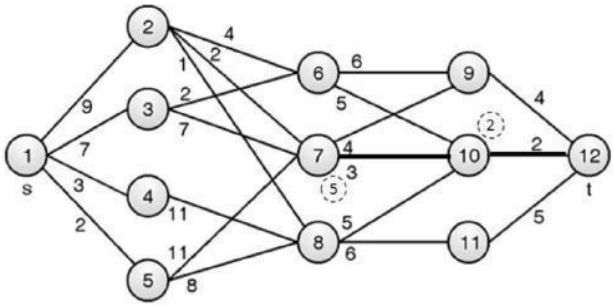
• min{3 + 2, 4 + 4} = min{5, 8} = 5

p[7] = 10

Vertex 8 is connected to vertex 10 and 11:

Cost[8] = min{ c[8, 11] + Cost[11], c[8, 10] + Cost[10] } =

min{6 + 5, 5 + 2} = min{11, 7} = 7 p[8] = 10



Stage 2:

Vertex 2 is connected to vertices6, 7 and 8:

Cost[2] = min{ c[2, 6] + Cost[6], c[2, 7] + Cost[7], c[2, 8] + Cost[8] } =
min{4 + 7, 2 + 5, 1 + 7} = min{11, 7, 8} = 7

p[2] = 7

Vertex 3 is connected to vertices 6and 7:

Cost[3] = min{ c[3, 6] + Cost[6], c[3, 7] + Cost[7] }

• min{2 + 7, 7 + 5} = min{9, 12} = 9

p[3] = 6

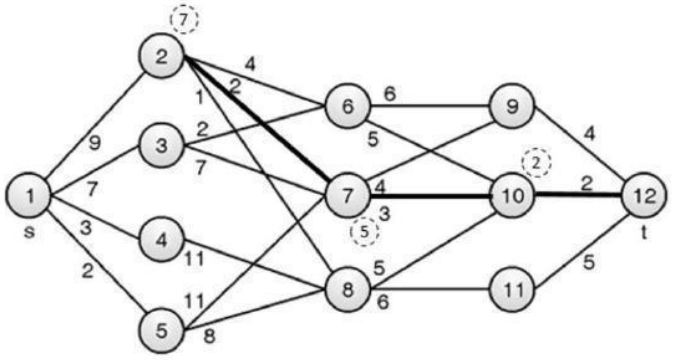
Vertex 4 is connected to vertex 8: Cost[4]

= c[4, 8] + Cost[8] = 11 + 7 = 18 p[4] = 8

Vertex 5 is connected to vertices 7 and 8: Cost[5] =

min{ c[5, 7] + Cost[7], c[5, 8] + Cost[8] }

• min{11 + 5, 8 + 7} = min{16, 15} = 15 p[5] = 8



Stage 1:

Vertex 1 is connected to vertices 2, 3, 4 and 5:

Cost[1] = min{ c[1, 2] + Cost[2], c[1, 3] + Cost[3], c[1, 4] + Cost[4], c[1, 5] + Cost[5]}

• min{ 9 + 7, 7 + 9, 3 + 18, 2 + 15 }

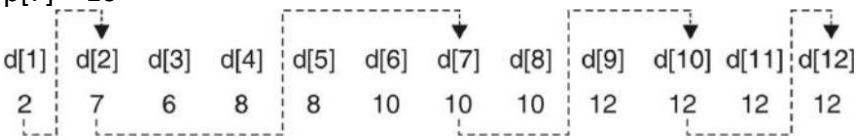
• min { 16, 16, 21, 17 } = 16 p[1] = 2

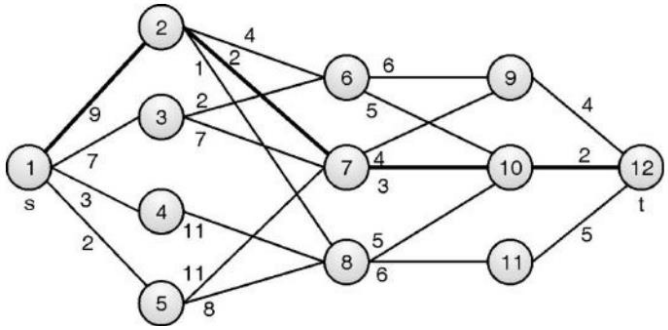
Trace the solution:

p[1] = 2

p[2] = 7

p[7] = 10





p[10] = 12

Minimum cost path is : 1 – 2 – 7 – 10 – 12

Cost of the path is : 9 + 2 + 3 + 2 = 16

Optimal Binary Search Tree

- Optimal Binary Search Tree extends the concept of Binary searc tree. Binary Search Tree (BST) is a *nonlinear* data structure which is used in many scientific applications for reducing the search time. In BST, left child is smaller than root and right child is greater than root. This arrangement simplifies the search procedure.

- Optimal Binary Search Tree (OBST) is very useful in dictionary search. The probability of searching is different for different words. OBST has great application in translation. If we translate the book from English to German, equivalent words are searched from English to German dictionary and replaced in translation. Words are searched same as in binary search tree order.

- Binary search tree simply arranges the words in lexicographical order. Words like *'the'*, *'is'*, *'there'* are very frequent words, whereas words

  like *'xylophone'*, *'anthropology'* etc. appears rarely.

- It is not a wise idea to keep less frequent words near root in binary search tree. Instead of storing words in binary search tree in lexicographical order, we shall arrange them according to their probabilities. This arrangement facilitates few searches for frequent words as they would be near the root. Such tree is called **Optimal Binary Search Tree**.

- Consider the sequence of *n*keys K = < k_1, k_2, k_3, …, k_n> of distinct probability in sorted order such that

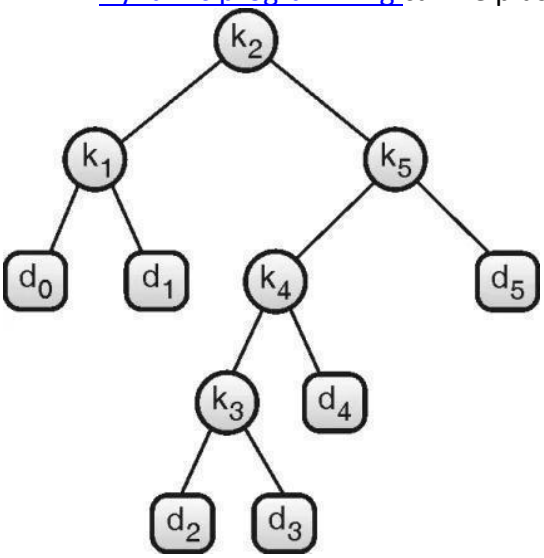
  $k_1 < k_2 < \ldots < k_n$. Words between each pair of key lead to unsuccessful search, so for n keys, binary search tree contains n + 1 dummy keys d_i, representing unsuccessful searches.

- Two different representation of BST with same five keys {k_1, k_2, k_3, k_4, k_5} probability is shown in following figure

- With n nodes, there exist (2n)!/((n + 1)! * n!) different binary search trees. An exhaustive search for optimal binary search tree leads to huge amount of time.

- The goal is to construct a tree which minimizes the total search cost. Such tree is called optimal binary search tree. OBST does not claim minimum height. It is also not necessary that parent of sub tree has higher priority than its child.

- Dynamic programming can help us to find such optima tree.



Binary search trees with 5 keys

Mathematical formulation

- We formulate the OBST with following observations
- Any sub tree in OBST contains keys in sorted order $k_i \ldots k_j$, where $1 \leq i \leq j \leq n$.
- Sub tree containing keys $k_i \ldots k_j$ has leaves with dummy keys $d_{i-1} \ldots . d_j$.
- Suppose k_r is the root of sub tree containing keys $k_i \ldots . . k_j$. So, left sub tree of root k_r contains keys

  $k_i \ldots . k_{r-1}$ and right sub tree contain keys k_{r+1} to k_j. Recursively, optimal sub trees are constructed from the left and right sub trees of k_r.
- Let e[i, j] represents the expected cost of searching OBST. With n keys, our aim is to find and minimize e[1, n].
- Base case occurs when j = i – 1, because we just have the dummy key d_{i-1} for this case. Expected search cost for this case would be $e[i, j] = e[i, i - 1] = q_{i-1}$.
- For the case $j \geq i$, we have to select any key k_r from $k_i \ldots k_j$ as a root of the tree.
- With k_r as a root key and sub tree $k_i \ldots k_j$, sum of probability is defined as

$$w(i, j) = \sum_{m=i}^{j} p_m + \sum_{m=i-1}^{j} q_m$$

(Actual key starts at index 1 and dummy key starts at index 0)

Thus, a recursive formula for forming the OBST is stated below :

$$e[i, j] = \begin{cases} q_{i-1} & \text{if } j = i - 1 \\ \min \{e[i, r-1] + e[r+1, j] + w(i, j) \\ i \le r \le j \} & \text{if } i \le j \end{cases}$$

e[i, j] gives the expected cost in the optimal binary search tree.

Algorithm for Optimal Binary Search Tree

The algorithm for optimal binary search tree is specified below :

Algorithm OBST(p, q, n)

- e[1…n+1, 0…n] : Optimal sub tree
- w[1…n+1, 0…n] : Sum of probability
- root[1…n, 1…n] : Used to construct OBST

for i ← 1 to n + 1 **do**

e[i, i – 1] ← qi – 1

w[i, i – 1] ← qi – 1

end

for m ← 1 to n **do**

for i ← 1 to n – m + 1 **do**

j ← i + m – 1

e[i, j] ← ∞

```
    w[i, j] ← w[i, j – 1] + pj + qj
    for r ← i to j do
      t ← e[i, r – 1] + e[r + 1, j] + w[i, j]
      if t < e[i, j] then
        e[i, j] ← t
        root[i, j] ← r
      end
    end
  end
end
return (e, root)
```

Complexity Analysis of Optimal Binary Search Tree

It is very simple to derive the complexity of this approach from the above algorithm. It uses three nested loops. Statements in the innermost loop run in Q(1) time. The running time of the algorithm is computed as

$$
\begin{aligned}
T(n) &= \sum_{m=1}^{n} \sum_{i=1}^{n-m+1} \sum_{j=i}^{n-l+1} \Theta(1) \\
&= \sum_{m=1}^{n} \sum_{i=1}^{n-m+1} n = \sum_{m=1}^{n} n^2 \\
&= \Theta(n^3)
\end{aligned}
$$

Thus, the OBST algorithm runs in cubic time

Example

Problem: Let p (1 : 3) = (0.5, 0.1, 0.05) q(0 : 3) = (0.15, 0.1, 0.05, 0.05) Compute and construct OBST for above values using Dynamic approach.

Solution:

Here, given that

i	0	1	2	3
p_i		0.5	0.1	0.05
q_i	0.15	0.1	0.05	0.05

Recursive formula to solve OBST problem is

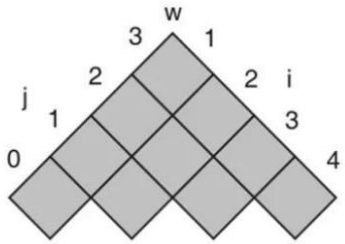
$$e[i, j] = \begin{cases} q_{i-1} & \text{if } j = i - 1 \\ \min \{e[i, r-1] + e[r+1, j] + w(i, j) & \\ i \le r \le j \} & \text{if } i \le j \end{cases}$$

Where,

$$w(i, j) = \sum_{m=i}^{j} p_m + \sum_{m=i-1}^{j} q_m$$

Initially,

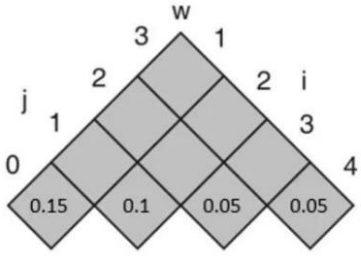
$$w[1, 0] = \sum_{m=1}^{0} p_m + \sum_{m=0}^{0} q_m = q_0 = 0.15$$



$$w[2, 1] = \sum_{m=2}^{1} p_m + \sum_{m=1}^{1} q_m = q_1 = 0.1$$

$$w[3, 2] = \sum_{m=3}^{2} p_m + \sum_{m=2}^{2} q_m = q_2 = 0.05$$

$$w[4, 3] = \sum_{m=4}^{3} p_m + \sum_{m=3}^{3} q_m = q_3 = 0.05$$



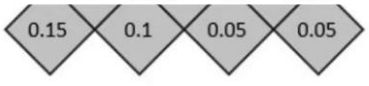
$$w[1,1] = \sum_{m=1}^{1} p_m + \sum_{m=0}^{1} q_m = p_1 + q_0 + q_1$$
$$= 0.5 + 0.25 = 0.75$$

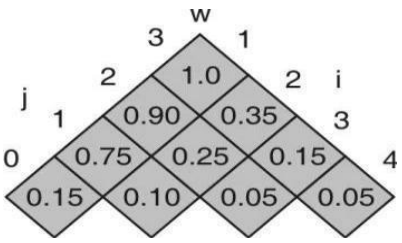
$$w[1,2] = \sum_{m=1}^{2} p_m + \sum_{m=0}^{2} q_m$$
$$= (p_1 + p_2) + (q_0 + q_1 + q_2)$$
$$= 0.6 + 0.3 = 0.90$$

$$w[2,2] = \sum_{m=2}^{2} p_m + \sum_{m=1}^{2} q_m = p_2 + q_1 + q_2$$
$$= 0.1 + 0.15 = 0.25$$

$$w[2,3] = \sum_{m=2}^{3} p_m + \sum_{m=1}^{3} q_m$$
$$= (p_2 + p_3) + (q_1 + q_2 + q_3)$$
$$= 0.15 + 0.2 = 0.35$$

$$w[3,3] = \sum_{m=3}^{3} p_m + \sum_{m=2}^{3} q_m = p_3 + q_2 + q_3$$
$$= 0.05 + 0.10 = 0.15$$

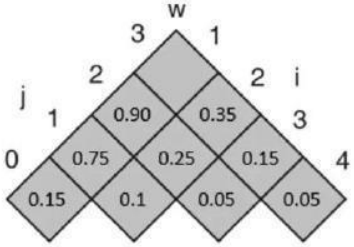




$$w[1,3] = \sum_{m=1}^{3} p_m + \sum_{m=0}^{3} q_m$$

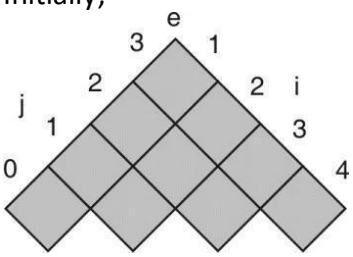
$$= (p_1 + p_2 + p_3) + (q_0 + q_1 + q_2 + q_3)$$

$$= 0.65 + 0.35 = 1$$

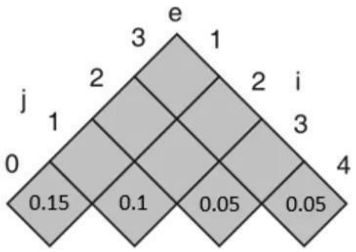


Now, we will compute e[i, j]

Initially,



e[1, 0]	= q_0	= 0.15	(	j = i – 1)
e[2, 1]	= q_1	= 0.1	(?	j = i – 1)
e[3, 2]	= q_2	= 0.05		j = i – 1)
e[4, 3]	= q_3	= 0.05	(∴	j = i – 1)



e[1, 1] = min { e[1, 0] + e[2, 1] + w(1, 1) }

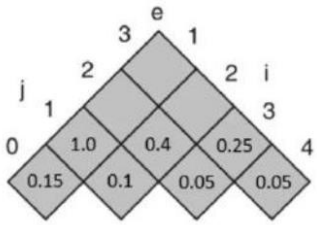
= min { 0.15 + 0.1 + 0.75 } = 1.0

e[2, 2] = min { e[2, 1] + e[3, 2] + w(2, 2) }

= min { 0.1 + 0.05 + 0.25 } = 0.4

e[3, 3] = min { e[3, 2] + e[4, 3] + w(3, 3) }

= min { 0.05 + 0.05 + 0.15 } = 0.25



$$e[1,2] = \min\begin{Bmatrix} e[1,0] + e[2,2] + w[1,2] \\ e[1,1] + e[3,2] + w[1,2] \end{Bmatrix}$$

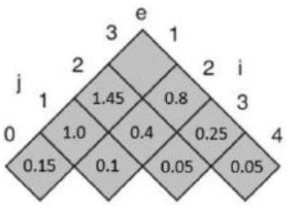
$$= \min\begin{Bmatrix} 0.15 + 0.4 + 0.90 \\ 1.0 + 0.05 + 0.90 \end{Bmatrix}$$

$$= \min\begin{Bmatrix} 1.45 \\ 1.95 \end{Bmatrix} = 1.45$$

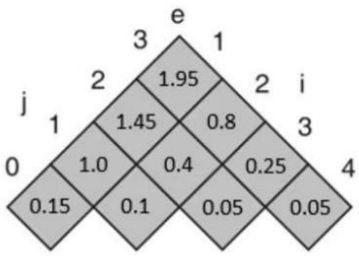
$$e[2,3] = \min\begin{Bmatrix} e[2,1] + e[3,3] + w(2,3) \\ e[2,2] + e[4,3] + w(2,3) \end{Bmatrix}$$

$$= \min\begin{Bmatrix} 0.1 + 0.25 + 0.35 \\ 0.4 + 0.05 + 0.35 \end{Bmatrix}$$

$$= \min\begin{Bmatrix} 0.90 \\ 0.80 \end{Bmatrix} = 0.8$$



$$
\begin{aligned}
e[1,3] &= \min\left\{\begin{array}{l} e[1,0]+e[2,3]+w(1,3) \\ e[1,1]+e[3,3]+w(1,3) \\ e[1,2]+e[4,3]+w(1,3) \end{array}\right\} \\
&= \min\left\{\begin{array}{l} 0.15+0.80+1.0 \\ 1.0+0.25+1.0 \\ 1.45+0.05+1.0 \end{array}\right\} \\
&= \min\left\{\begin{array}{l} 1.95 \\ 2.25 \\ 2.5 \end{array}\right\} = 1.95
\end{aligned}
$$



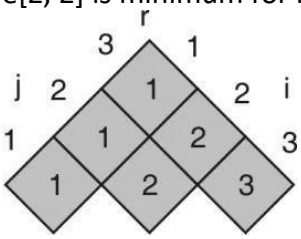
e[1, 3] is minimum for r = 1, so r[1, 3] = 1

e[2, 3] is minimum for r = 2, so r[2, 3] = 2

e[1, 2] is minimum for r = 1, so r[1, 2] = 1

e[3, 3] is minimum for r = 3, so r[3, 3] = 3

e[2, 2] is minimum for r = 2, so r[2, 2] = 2



e[1, 1] is minimum for r = 1, so r[1, 1] = 1

Let us now construct OBST for given data.

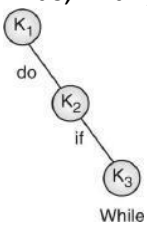
r[1, 3] = 1, so k_1 will be at the root.

$k_{2....3}$ are on right side of k_1

r[2, 3] = 2, So k_2 will be the root of this sub tree.

k_3 will be on the right of k_2.

Thus, finally, we get.



Greedy Technique

Activity Selection Problem

Activity Selection problem is a approach of selecting non-conflicting tasks based on start and end time and can be solved in O(N logN) time using a simple greedy approach. Modifications of this problem are complex and interesting which we will explore as well. Suprising, if we use a Dynamic Programming approach, the time complexity will be O(N^3) that is lower performance.

The problem statement for Activity Selection is that "Given a set of n activities with their start and finish times, we need to select maximum number of non-conflicting activities that can be performed by a single person, given that the person can handle only one activity at a time." The Activity Selection problem follows Greedy approach i.e. at every step, we can make a choice that looks best at the moment to get the optimal solution of the complete problem.

Our objective is to complete maximum number of activities. So, choosing the activity which is going to finish first will leave us maximum time to adjust the later activities. This is the intuition

that greedily choosing the activity with earliest finish time will give us an optimal solution. By induction on the number of choices made, making the greedy choice at every step produces an optimal solution, so we chose the activity which finishes first. If we sort elements based on their starting time, the activity with least starting time could take the maximum duration for completion, therefore we won't be able to maximise number of activities.

Algorithm

The algorithm of Activity Selection is as follows:

```
Activity-Selection(Activity, start, finish)
    Sort Activity by finish times stored in finish
    Selected = {Activity[1]}
    n = Activity.length
    j = 1
    for i = 2 to n:
       if start[i] ≥ finish[j]:
          Selected = Selected U {Activity[i]}
          j = i
 return Selected
```

Complexity

Time Complexity:

When activities are sorted by their finish time: **O(N)**

When activities are not sorted by their finish time, the time complexity is **O(N log N)** due to complexity of sorting

START	1	3	2	0	5	8	11
END	3	4	5	7	9	10	12

START[4] >=END[2], SELECTED

START	1	3	2	0	5	8	11
END	3	4	5	7	9	10	12

START[5]<END[4], REJECTED

START	1	3	2	0	5	8	11
END	3	4	5	7	9	10	12

START[6]>=END[4], SELECTED

	0	1	2	3	4	5	6
START	1	3	2	0	5	8	11
END	3	4	5	7	9	10	12

SELECTED

START	1	3	2	0	5	8	11
END	3	4	5	7	9	10	12

START[1]>=END[0], SELECTED

START	1	3	2	0	5	8	11
END	3	4	5	7	9	10	12

START[2]<END[1], REJECTED

START	1	3	2	0	5	8	11
END	3	4	5	7	9	10	12

START[3]<END[1], REJECTED

In this example, we take the start and finish time of activities as follows:

start = [1, 3, 2, 0, 5, 8, 11]

finish = [3, 4, 5, 7, 9, 10, 12]

Sorted by their finish time, the activity 0 gets selected. As the activity 1 has starting time which is equal to the finish time of activity 0, it gets selected. Activities 2 and 3 have smaller starting time than finish time of activity 1, so they get rejected. Based on similar comparisons, activities 4 and 6 also get selected, whereas activity 5 gets rejected. In this example, in all the activities 0, 1, 4 and 6 get selected, while others get rejected.

Optimal Merge Pattern

Merge a set of sorted files of different length into a single sorted file. We need to find an optimal solution, where the resultant file will be generated in minimum time.

If the number of sorted files are given, there are many ways to merge them into a single sorted file. This merge can be performed pair wise. Hence, this type of merging is called as 2-way merge patterns.

As, different pairings require different amounts of time, in this strategy we want to determine an optimal way of merging many files together. At each step, two shortest sequences are merged.

To merge a p-record file and a q-record file requires possibly p + q record moves, the obvious choice being, merge the two smallest files together at each step.

Two-way merge patterns can be represented by binary merge trees. Let us consider a set of **n** sorted files **{f_1, f_2, f_3, …, f_n}**. Initially, each element of this is considered as a single node binary tree. To find this optimal solution, the following algorithm is used.

Algorithm: TREE (n)

```
for i := 1 to n – 1 do
  declare new node
  node.leftchild := least (list)
  node.rightchild := least (list)
  node.weight) := ((node.leftchild).weight) + ((node.rightchild).weight)
  insert (list, node);
return least (list);
```

At the end of this algorithm, the weight of the root node represents the optimal cost.

Example

Let us consider the given files, f_1, f_2, f_3, f_4 and f_5 with 20, 30, 10, 5 and 30 number of elements respectively.

If merge operations are performed according to the provided sequence, then

M_1 = merge f_1 and f_2 => 20 + 30 = 50

M_2 = merge M_1 and f_3 => 50 + 10 = 60

M_3 = merge M_2 and f_4 => 60 + 5 = 65

M_4 = merge M_3 and f_5 => 65 + 30 = 95

Hence, the total number of operations is

50+60+65+95=270

Now, the question arises is there any better solution?

Sorting the numbers according to their size in an ascending order, we get the following sequence −

f_4, f_3, f_1, f_2, f_5

Hence, merge operations can be performed on this sequence

M_1 = merge f_4 and f_3 => 5 + 10 = 15

M_2 = merge M_1 and f_1 => 15 + 20 = 35

M_3 = merge M_2 and f_2 => 35 + 30 = 65

M_4 = merge M_3 and f_5 => 65 + 30 = 95

Therefore, the total number of operations is

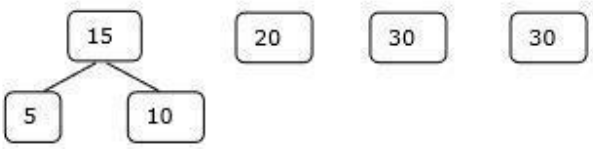
15+35+65+95=210

Obviously, this is better than the previous one.

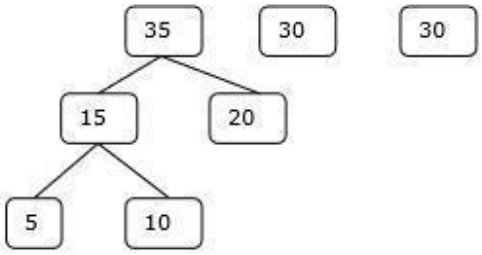
In this context, we are now going to solve the problem using this algorithm.

Initial Set

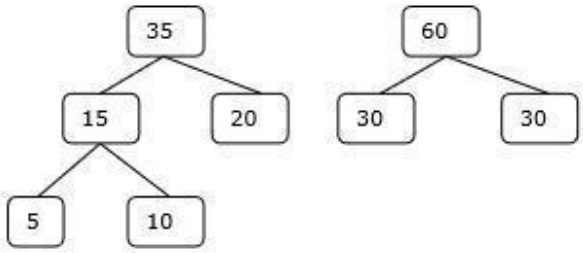
Step 1



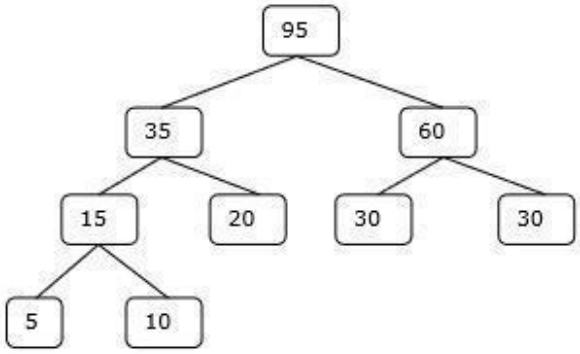
Step 2



Step 3



Step 4



Hence, the solution takes 15 + 35 + 60 + 95 = 205 number of comparisons.

Huffman Tree

Huffman coding provides codes to characters such that the length of the code depends on the relative frequency or weight of the corresponding character. Huffman codes are of variable-length, and without any prefix (that means no code is a prefix of any other). Any prefix-free

binary code can be displayed or visualized as a binary tree with the encoded characters stored at the leaves.

Huffman tree or Huffman coding tree defines as a full binary tree in which each leaf of the tree corresponds to a letter in the given alphabet.

The Huffman tree is treated as the binary tree associated with minimum external path weight that means, the one associated with the minimum sum of weighted path lengths for the given set of leaves. So the goal is to construct a tree with the minimum external path weight.

An example is given below-

Letter frequency table

Letter

z

k

m

c

u

d

l

e

Frequency

2

7

24

32

37

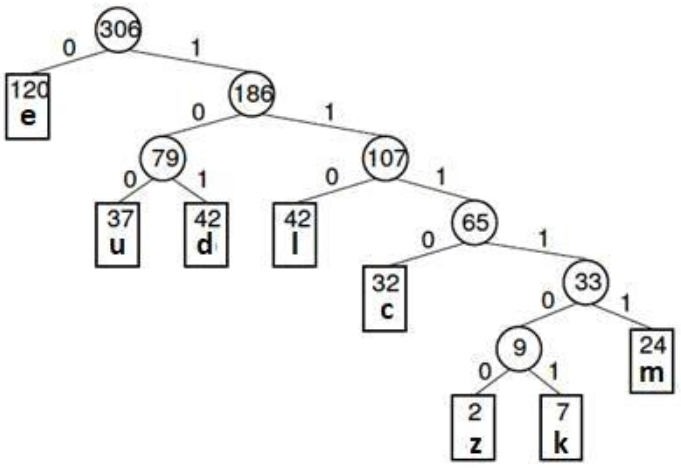
42

42

120

Huffman code

Letter	Freq	Code	Bits
e	120	0	1
d	42	101	3
l	42	110	3
u	37	100	3
c	32	1110	4
m	24	11111	5
k	7	111101	6
z	2	111100	6



The Huffman tree (for the above example) is given below -

```
Algorithm Huffman (c)
{

 n= |c|
 Q = c
 for i<-1 to n-1
 do
 {
     temp <- get node ()
    left (temp] Get_min (Q) right [temp] Get Min (Q)
    a = left [templ b = right [temp]
    F [temp]<- f[a] + [b]
    insert (Q, temp)
  }
```

```
return Get_min (0)

}
```

UNIT 4

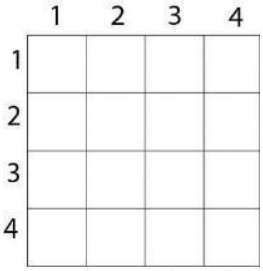
Backtracking

N queen Problem

N - Queens problem is to place n - queens in such a manner on an n x n chessboard that no queens attack each other by being in the same row, column or diagonal.

It can be seen that for n =1, the problem has a trivial solution, and no solution exists for n =2 and n =3. So first we will consider the 4 queens problem and then generate it to n - queens problem.

Given a 4 x 4 chessboard and number the rows and column of the chessboard 1 through 4.



4x4 chessboard

Since, we have to place 4 queens such as q_1 q_2 q_3 and q_4 on the chessboard, such that no two queens attack each other. In such a conditional each queen must be placed on a different row, i.e., we put queen "i" on row "i."

Now, we place queen q_1 in the very first acceptable position (1, 1). Next, we put queen q_2 so that both these queens do not attack each other. We find that if we place q_2 in column 1 and 2, then the dead end is encountered. Thus the first acceptable position for q_2 in column 3, i.e. (2, 3) but then no position is left for placing queen 'q_3' safely. So we backtrack one step and place the queen 'q_2' in (2, 4), the next best possible solution. Then we obtain the position for placing 'q_3' which is (3, 2). But later this position also leads to a dead end, and no place is found where 'q_4' can be placed safely. Then we have to backtrack till 'q_1' and place it to (1, 2) and then all other queens are placed safely by moving q_2 to (2, 4), q_3 to (3, 1) and q_4 to (4, 3). That is, we get the solution (2, 4, 1, 3). This is one possible solution for the 4-queens problem. For another possible solution, the whole method is repeated for all partial solutions. The other solutions for 4 - queens problems is (3, 1, 4, 2) i.e.

	1	2	3	4
1			q_1	
2	q_2			
3				q_3
4		q_4		

The implicit tree for 4 - queen problem for a solution (2, 4, 1, 3) is as follows:

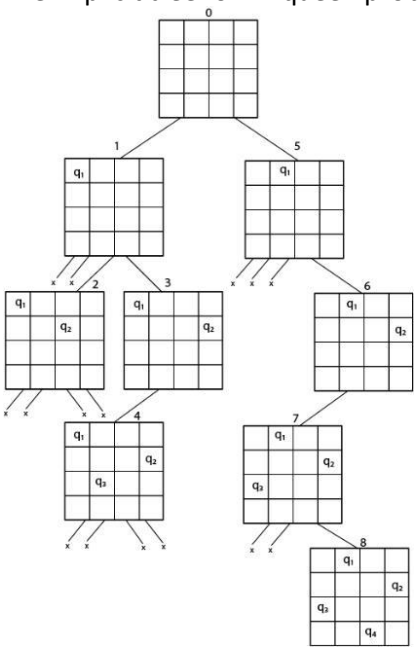
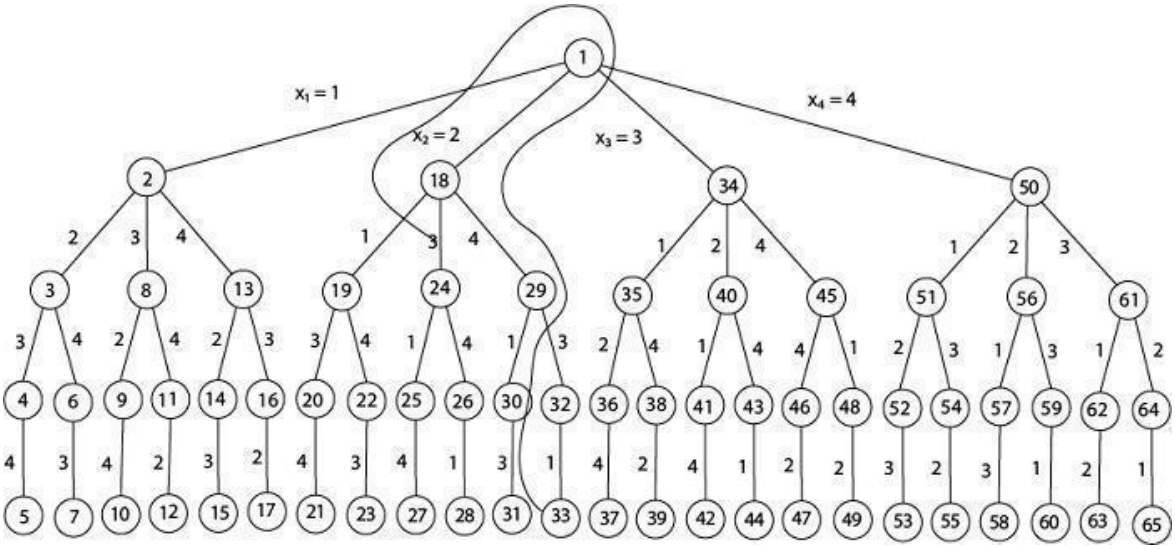


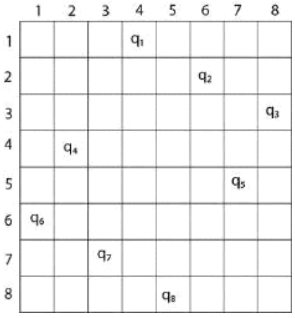
Fig shows the complete state space for 4 - queens problem. But we can use backtracking method to generate the necessary node and stop if the next node violates the rule, i.e., if two queens are attacking.



4 - Queens solution space with nodes numbered in DFS

It can be seen that all the solutions to the 4 queens problem can be represented as 4 - tuples (x_1, x_2, x_3, x_4) where x_i represents the column on which queen "q_i" is placed.

One possible solution for 8 queens problem is shown in fig:



- Thus, the solution **for** 8 -queen problem **for** (4, 6, 8, 2, 7, 1, 3, 5).
- If two queens are placed at position (i, j) and (k, l).
- Then they are on same diagonal only **if** (i - j) = k - l or i + j = k + l.

- The first equation implies that j - l = i - k.
- The second equation implies that j - l = k - i.
- Therefore, two queens lie on the duplicate diagonal **if** and only **if** |j-l|=|i-k|

Place (k, i) returns a Boolean value that is true if the kth queen can be placed in column i. It tests both whether i is distinct from all previous costs x_1, x_2,....x_{k-1} and whether there is no other queen on the same diagonal.

Using place, we give a precise solution to then n- queens problem.

1.	Place (k, i)	
2.	{	

- For j ← 1 to k - 1
- **do if** (x [j] = i)
- or (Abs x [j]) - i) = (Abs (j - k))
- then **return false**;
- **return true**;
- }

Place (k, i) return true if a queen can be placed in the kth row and ith column otherwise return is false.

x [] is a global array whose final k - 1 values have been set. Abs (r) returns the absolute value of r.

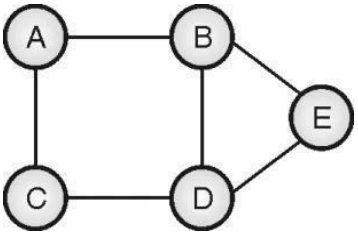
- N - Queens (k, n)
- {
- For i ← 1 to n
- **do if** Place (k, i) then
- {
- x [k] ← i;
- **if** (k ==n) then
- write (x [1....n));

- **else**
- N - Queens (k + 1, n);
- }
- }

Hamiltonian Circuit

*The **Hamiltonian cycle** is the cycle in the graph which visits all the vertices in graph exactly once and terminates at the starting node. It may not include all the edges*

- **The** Hamiltonian cycle problem is the problem of finding a Hamiltonian cycle in a graph if there exists any such cycle.
- The input to the problem is an undirected, connected graph. For the graph shown in Figure (a), a path A – B – E – D – C – A forms a Hamiltonian cycle. It visits all the vertices exactly once, but does not visit the edges <B, D>.



- The Hamiltonian cycle problem is also both, decision problem and an optimization problem. A decision problem is stated as, “Given a path, is it a Hamiltonian cycle of the graph?”.
- The optimization problem is stated as, “Given graph G, find the Hamiltonian cycle for the graph.”
- We can define the constraint for the Hamiltonian cycle problem as follows:
    - In any path, vertex i and (i + 1) must be adjacent.
    - 1st and (n – 1)th vertex must be adjacent (nth of cycle is the initial vertex itself).
    - Vertex i must not appear in the first (i – 1) vertices of any path.

- With the adjacency matrix representation of the graph, the adjacency of two vertices can be verified in constant time.

Algorithm

HAMILTONIAN (i)

- Description : Solve Hamiltonian cycle problem using backtracking.
- Input : Undirected, connected graph G = <V, E> and initial vertex i
- Output : Hamiltonian cycle

if

FEASIBLE(i)

then

if

(i == n - 1)

then

Print V[0… n – 1]

else

j ← 2

while

(j ≤ n)

do

V[i] ← j

HAMILTONIAN(i + 1)

j ← j + 1

e**nd**

end

end

function

FEASIBLE(i)

flag ← 1

for

j ← 1 to i – 1

do

if

Adjacent(Vi, Vj)

then

flag ← 0

end

end

if

Adjacent (Vi, Vi-1)

then

flag ← 1

else

flag ← 0

end

return

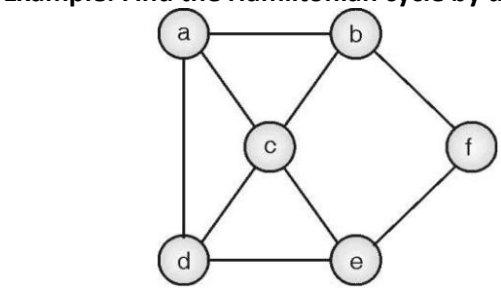
flag

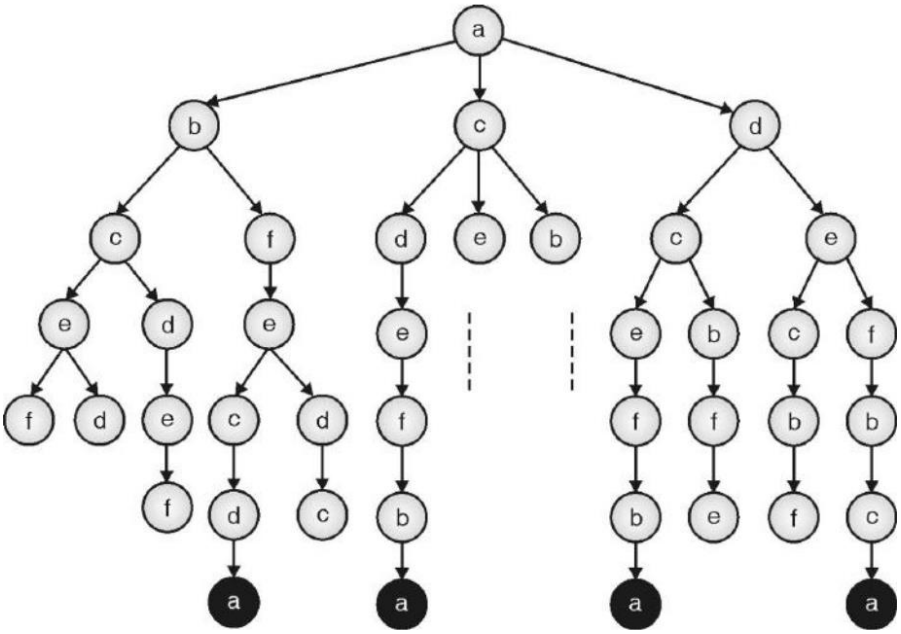
Complexity Analysis

Looking at the state space graph, in worst case, total number of nodes in tree would be, $T(n) = 1 + (n – 1) + (n – 1)^2 + (n – 1)^3 + … + (n – 1)^{n – 1}$ =frac(n−1)n−1n−2

$T(n) = O(n^n)$. Thus, the Hamiltonian cycle algorithm runs in exponential time.

Example: Find the Hamiltonian cycle by using the backtracking approach for a given graph.





The backtracking approach uses a state-space tree to check if there exists a Hamiltonian cycle in the graph.

Figure (g) shows the simulation of the Hamiltonian cycle algorithm. For simplicity, we have not explored all

possible paths, the concept is self-explanatory. It is not possible to include all the paths in the graph, so few

of the successful and unsuccessful paths are traced in the graph. Black nodes indicate the Hamiltonian cycle.

Subset Sum Problem

Sum of Subsets Problem: Given a set of positive integers, find the combination of numbers that sum to given value M.

Sum of subsets problem is analogous to the **knapsack problem**. The Knapsack Problem tries to fill the knapsack using a given set of items to maximize the profit. Items are selected in such a way that the total weight in the knapsack does not exceed the capacity of the knapsack. The inequality condition in the knapsack problem is replaced by equality in the sum of subsets problem.

Given the set of n positive integers, W = {w1, w2, …, wn}, and given a positive integer M, the sum of the subset problem can be formulated as follows (where wi and M correspond to item weights and knapsack capacity in the knapsack problem):

$$sum_{i=1}^{n} w_i x_i = M$$

Where,

$$x_i in 0, 1$$

Numbers are sorted in ascending order, such that $w_1 < w_2 < w_3 < .... < w_n$. The solution is often represented

using the solution vector X. If the ith item is included, set xi to 1 else set it to 0. In each iteration, one item is tested. If the inclusion of an item does not violet the constraint of the problem, add it. Otherwise, backtrack, remove the previously added item, and continue the same procedure for all remaining items. The solution is easily described by the state space tree. Each left edge denotes the inclusion of wi and the right edge denotes the exclusion of w_i. Any path from the root to the leaf forms a subset. A state-space tree for n = 3 is demonstrated in Fig. (a).

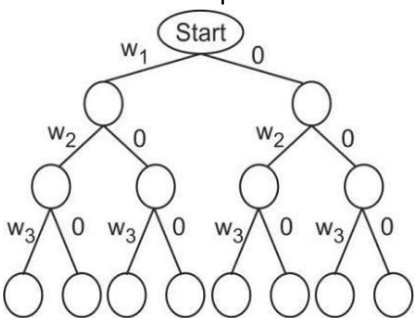


Fig. (a): State space tree for n = 3

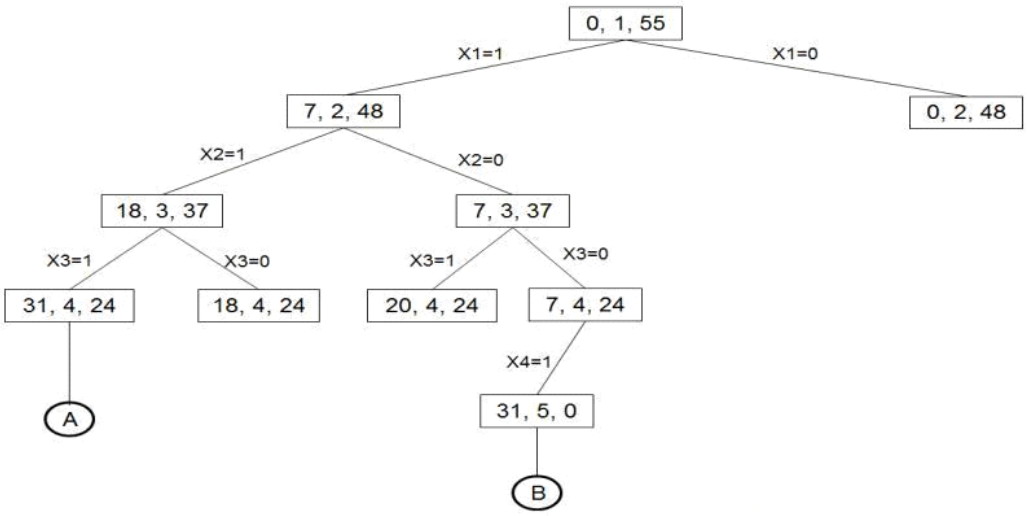
Algorithm for Sum of subsets

The algorithm for solving the sum of subsets problem using recursion is stated below:

```
Algorithm sumofsubsets(s,k,r)
{
  X[k]:=1;
  if (s+w[k]=m) then write (x[1:k]); // subset found
  else
    if (s+w[k]+w[k+1]<=m) then
       sumofsubsets(s+w[k],k+1,r-w[k]);

   //generate right child and evaluate Bk.

   if ((s+r-w[k]>=m) and (s+w[k+1]<=m)) then
    {
      X[k]:=0;
      sumofsubsets(s,k+1,r-w[k]);
    }
}
```

Examples

Ex) n=4, $(w_1, w_2, w_3, w_4)=(7,11,13,24)$ and m=31
Solution Vector=(x[1],x[2],x[3],x[4])



Portion of state space Tree

Solution A = {1,1,1,0}
Solution B = {1,0,0,1}

Graph Colouring

In this problem, an undirected graph is given. There is also provided m colors. The problem is to find if it is possible to assign nodes with m different colors, such that no two adjacent vertices of the graph are of the same colors. If the solution exists, then display which color is assigned on which vertex.

Starting from vertex 0, we will try to assign colors one by one to different nodes. But before assigning, we

have to check whether the color is safe or not. A color is not safe whether adjacent vertices are containing

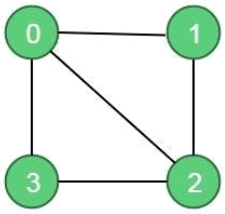
the same color.

Input and Output

Input:

The adjacency matrix of a graph G(V, E) and an integer m, which indicates the maximum number of colors that can be used.

0	1	1	1
1	0	1	0
1	1	0	1
1	0	1	0



Let the maximum color m = 3.

Output:

This algorithm will return which node will be assigned with which color. If the solution is not possible, it will return false.

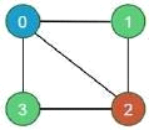
For this input the assigned colors are:

Node 0 -> color 1

Node 1 -> color 2

Node 2 -> color 3

Node 3 -> color 2



Algorithm

isValid(vertex, colorList, col)

Input — Vertex, colorList to check, and color, which is trying to assign.

Output — True if the color assigning is valid, otherwise false.

Begin

```
for all vertices v of the graph, do

  if there is an edge between v and i, and col = colorList[i],
    then return false

done
```

return true

End

graphColoring(colors, colorList, vertex)

Input — Most possible colors, the list for which vertices are colored with which color, and the starting vertex.

Output — True, when colors are assigned, otherwise false.

Begin

if all vertices are checked, then

return true

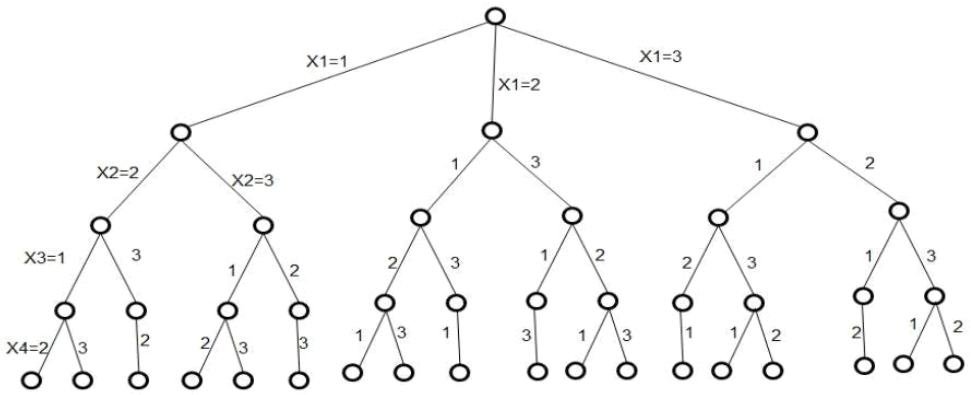
for all colors col from available colors, do

if isValid(vertex, color, col), then

add col to the colorList for vertex

if graphColoring(colors, colorList, vertex+1) = true,
then return true

remove color for vertex



A 4-node graph and all possible 3-colorings

done

return false

End

Branch and Bound

Solving 15 puzzle Problem (LCBB)

The problem cinsist of 15 numbered (0-15) tiles on a square box with 16 tiles(one tile is blank or empty). The objective of this problem is to change the arrangement of initial node to goal node by using series of legal moves.

The Initial and Goal node arrangement is shown by following figure.

1	2	4	15
2		5	12
7	6	11	14
8	9	10	13
	Initial Arrangement		

1		2	3	4
5		6	7	8
9		10	11	12
13		14	15	
	Final Arrangement			

In initial node four moves are possible. User can move any one of the tile like 2,or 3, or 5, or 6 to the empty tile. From this we have four possibilities to move from initial node.

The legal moves are for adjacent tile number is left, right, up, down, ones at a time.

Each and every move creates a new arrangement, and this arrangement is called state of puzzle problem. By using different states, a state space tree diagram is created, in which edges are labeled according to the direction in which the empty space moves.

The state space tree is very large because it can be 16! Different arrangements.

In state space tree, nodes are numbered as per the level. In each level we must calculate the value

or cost of each node by using given formula:

C(x)=f(x)+g(x),

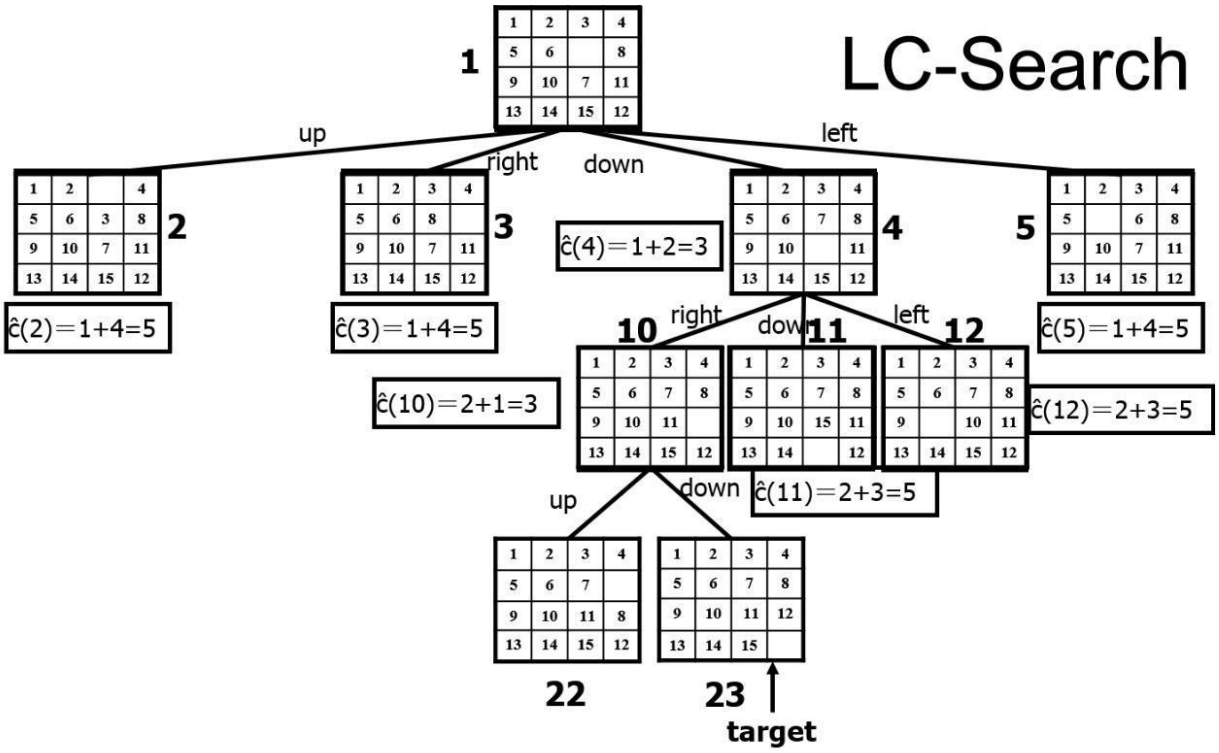
f(x) is length of path from root or initial node to node x,

g(x) is estimated length of path from x downward to the goal node. Number of non blank tile not in their correct position.

C(x)< Infinity.(initially set bound).

Each time node with smallest cost is selected for further expansion towards goal node. This node become the e-node.

State Space tree with node cost is shown in diagram.


LC-Search
1
up
right
down
left
2
3
4
5
ĉ(2)=1+4=5
ĉ(3)=1+4=5
ĉ(4)=1+2=3
ĉ(5)=1+4=5
10
11
12
right
down
left
ĉ(10)=2+1=3
ĉ(11)=2+3=5
ĉ(12)=2+3=5
up
down
22
23
target

Assignment Problem

Problem Statement

Let's first define a job assignment problem. In a standard version of a job assignment problem, there can be jobs and workers. To keep it simple, we're taking jobs and workers in our example:

	Job 1	Job 2	Job 3
A	9	3	4
B	7	8	4
C	10	5	2

We can assign any of the available jobs to any worker with the condition that if a job is assigned to a worker, the other workers can't take that particular job. We should also notice that each job has some cost associated with it, and it differs from one worker to another.

Here the main aim is to complete all the jobs by assigning one job to each worker in such a way that the sum of the cost of all the jobs should be minimized.

Branch and Bound Algorithm Pseudocode

Now let's discuss how to solve the job assignment problem using a branch and bound algorithm.

Let's see the pseudocode first:

Algorithm 1: Job Assignment Problem Using Branch And Bound

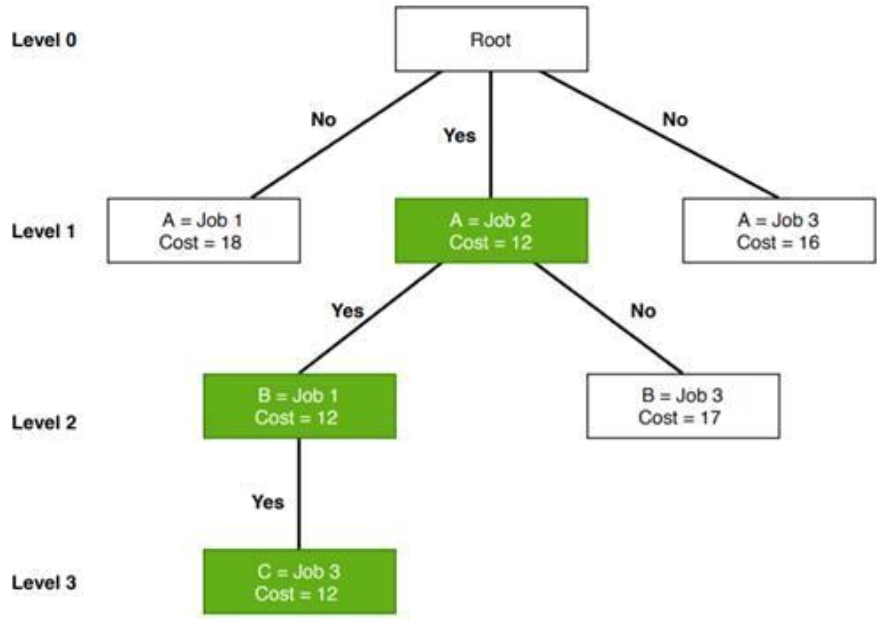
```
Data: Input cost matrix M[][]
Result: Assignment of jobs to each worker according to optimal
        cost
Function MinCost(M[][])
while True do
    E = LeastCost();
    if E is a leaf node then
        print();
        return;
    end
    for each child S of E do
        Add(S);
        S → parent = E;
    end
end
```

Here, is the input cost matrix that contains information like the number of available jobs, a list of available workers, and the associated cost for each job. The function MinCost() maintains a list of active nodes. The function Leastcost() calculates the minimum cost of the active node at each level of the tree. After finding the node with minimum cost, we remove the node from the list of active nodes and return it.

We're using the add() function in the pseudocode, which calculates the cost of a particular node and adds it to the list of active nodes.

In the search space tree, each node contains some information, such as cost, a total number of jobs, as well as a total number of workers.

Now let's run the algorithm on the sample example we've created:



Advantages

In a branch and bound algorithm, we don't explore all the nodes in the tree. That's why **the time complexity of the branch and bound algorithm is less when compared with other algorithms.**

If the problem is not large and if we can do the branching in a reasonable amount of time, **it finds an optimal solution for a given problem.**

The branch and bound algorithm find a minimal path to reach the optimal solution for a given problem. **It doesn't repeat nodes while exploring the tree. Disadvantages**

The branch and bound algorithm are time-consuming. Depending on the size of the given problem, the number of nodes in the tree can be too large in the worst case.

Knapsack Problem using branch and bound

Problem Statement

We are a given a set of *n* objects which have each have a value v_i and a weight w_i. The objective of the 0/1 Knapsack problem is to find a subset of objects such that the total value is maximized, and

> Initially, we've 3 jobs available. The worker A has the option to take any of the available jobs. So at level 1, we assigned all the available jobs to the worker A and calculated the cost. We can see that when we assigned jobs 2 to the worker A, it gives the lowest cost in level 1 of the search space tree. **So we assign the job 2 to worker A and continue the algorithm. "Yes" indicates that this is currently optimal cost.**
>
> After assigning the job 2 to worker A, we still have two open jobs. Let's consider worker B now. We're trying to assign either job 1 or 3 to worker B to obtain optimal cost.
>
> Either we can assign the job 1 or 3 to worker B. **Again we check the cost and assign job 1 to worker B as it is the lowest in level 2.**
>
> **Finally, we assign the job 3 to worker C, and the optimal cost is 12.**

the sum of weights of the objects does not exceed a given threshold *W*. An important condition here is that one can either take the entire object or leave it. It is not possible to take a fraction of the object.

Consider an example where *n* = 4, and the values are given by {10, 12, 12, 18}and the weights given by {2, 4, 6, 9}. The maximum weight is given by *W = 15*. Here, the solution to the problem will be including the first, third and the fourth objects.

Here, the procedure to solve the problem is as follows are:

> Calculate the cost function and the Upper bound for the two children of each node. Here, the $(i + 1)^{th}$ level indicates whether the i^{th} object is to be included or not.
> If the cost function for a given node is greater than the upper bound, then the node need not be explored further. Hence, we can kill this node. Otherwise, calculate the upper bound for this node. If this value is less than *U*, then replace the value of *U* with this value. Then, kill all unexplored nodes which have cost function greater than this value.
>
> The next node to be checked after reaching all nodes in a particular level will be the one with the least cost function value among the unexplored nodes.
>
> While including an object, one needs to check whether the adding the object crossed the threshold. If it does, one has reached the terminal point in that branch, and all the succeeding objects will not be included.

Time and Space Complexity

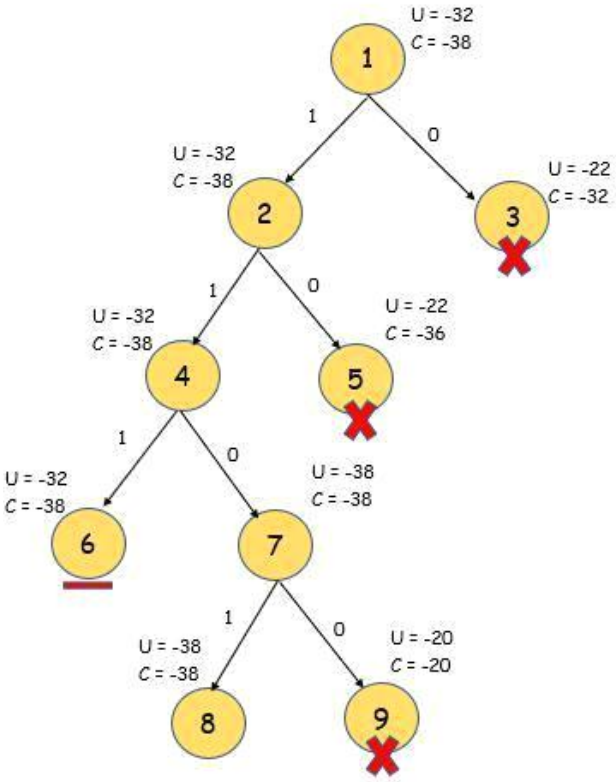
Even though this method is more efficient than the other solutions to this problem, its worst case time complexity is still given by $O(2^n)$, in cases where the entire tree has to be explored. However, in its best case, only one path through the tree will have to explored, and hence its best case time complexity is given by O(n). Since this method requires the creation of the state space tree, itsspace complexity will also be exponential.

Solving an Example

Consider the problem with n =4, V = {10, 10, 12, 18}, w = {2, 4, 6, 9} and W = 15. Here, we calculate the initital upper bound to be U = 10 + 10 + 12 = 32. Note that the 4th object cannot be included here,

since that would exceed W. For the cost, we add 3/9 th of the final value, and hence the cost function is 38. Remember to negate the values after calculation before comparison.

After calculating the cost at each node, kill nodes that do not need exploring. Hence, the final state space tree will be as follows (Here, the number of the node denotes the order in which the state space tree was explored):



Note here that node 3 and node 5 have been killed after updating U at node 7. Also, node 6 is not explored further, since adding any more weight exceeds the threshold. At the end, only nodes 6 and 8 remain. Since the value of U is less for node 8, we select this node. Hence the solution is {1, 1, 0, 1}, and we can see here that the total weight is exactly equal to the threshold value in this case.

Travelling salesman problem

- Travelling Salesman Problem (TSP) is an interesting problem. Problem is defined as "given n cities and distance between each pair of cities, find out the path which visits each city exactly once and come back to starting city, with the constraint of minimizing the travelling distance."

- TSP has many practical applications. It is used in network design, and transportation route design. The objective is to minimize the distance. We can start tour from any random city

  and visit other cities in any order. With n cities, n! different permutations are possible.

  Exploring all paths using brute force attacks may not be useful in real life applications.

LCBB using Static State Space Tree for Travelling Salseman Problem

- Branch and bound HYPERLINK "https://codecrucks.com/branch-and-bound-the-dummies-guide/" HYPERLINK "https://codecrucks.com/branch-and-bound-the-dummies-guide/" HYPERLINK "https://codecrucks.com/branch-and-bound-the-dummies-guide/" is an effective way to find better, if not best, solution in quick time by pruning some of the unnecessary branches of search tree.
- It works as follow:

Consider directed weighted graph G = (V, E, W), where node represents cities and weighted directed edges represents direction and distance between two cities.

- Initially, graph is represented by cost matrix C, where

  C_{ij} = cost of edge, if there is a direct path from city i to city j C_{ij} = ∞, if there is no direct path from city i to city j.

- Convert cost matrix to reduced matrix by subtracting minimum values from appropriate rows and columns, such that each row and column contains at least one zero entry.

- Find cost of reduced matrix. Cost is given by summation of subtracted amount from the cost matrix to convert it in to reduce matrix.
- Prepare state space tree for the reduce matrix

- Find least cost valued node A (i.e. E-node), by computing reduced cost node matrix with every remaining node.

- If <i, j> edge is to be included, then do following :

- Set all values in row i and all values in column j of A to ∞

- Set A[j, 1] = ∞

- Reduce A again, except rows and columns having all ∞ entries.

- Compute the cost of newly created reduced matrix as,

Cost = L + Cost(i, j) + r

Where, L is cost of original reduced cost matrix and r is A[i, j].

- If all nodes are not visited then go to step 4.

Reduction procedure is described below :

Raw Reduction:

Matrix M is called reduced matrix if each of its row and column has at least one zero entry or entire row or entire column has ∞ value. Let M represents the distance matrix of 5 cities. M can be reduced as follow:

$M_{RowRed} = \{M_{ij} - \min \{M_{ij} \mid 1 \leq j \leq n, \text{ and } M_{ij} < \infty \}\}$

Consider the following distance matrix:

M =

∞	20	30	10	11
15	∞	16	4	2
3	5	∞	2	4
19	6	18	∞	3
16	4	7	16	∞

Find the minimum element from each row and subtract it from each cell of matrix.

$$M = \begin{array}{|c|c|c|c|c|l|} \hline \infty & 20 & 30 & 10 & 11 & \rightarrow 10 \\ \hline 15 & \infty & 16 & 4 & 2 & \rightarrow 2 \\ \hline 3 & 5 & \infty & 2 & 4 & \rightarrow 2 \\ \hline 19 & 6 & 18 & \infty & 3 & \rightarrow 3 \\ \hline 16 & 4 & 7 & 16 & \infty & \rightarrow 4 \\ \hline \end{array}$$

Reduced matrix would be:

$$M_{RowRed} = \begin{array}{|c|c|c|c|c|} \hline \infty & 10 & 20 & 0 & 1 \\ \hline 13 & \infty & 14 & 2 & 0 \\ \hline 1 & 3 & \infty & 0 & 2 \\ \hline 16 & 3 & 15 & \infty & 0 \\ \hline 12 & 0 & 3 & 12 & \infty \\ \hline \end{array}$$

Row reduction cost is the summation of all the values subtracted from each rows:

Row reduction cost (M) = 10 + 2 + 2 + 3 + 4 = 21

Column reduction:

Matrix M_{RowRed} is row reduced but not the column reduced. Matrix is called column reduced if each of its column has at least one zero entry or all ∞ entries.

$M_{ColRed} = \{M_{ji} - \min \{M_{ji} \mid 1 \leq j \leq n, \text{ and } M_{ji} < \infty \}\}$

To reduced above matrix, we will find the minimum element from each column and subtract it from each cell of matrix.

$M_{RowRed} =$

∞	10	20	0	1
13	∞	14	2	0
1	3	∞	0	2
16	3	15	∞	0
12	0	3	12	∞
↓	↓	↓	↓	↓
1	0	3	0	0

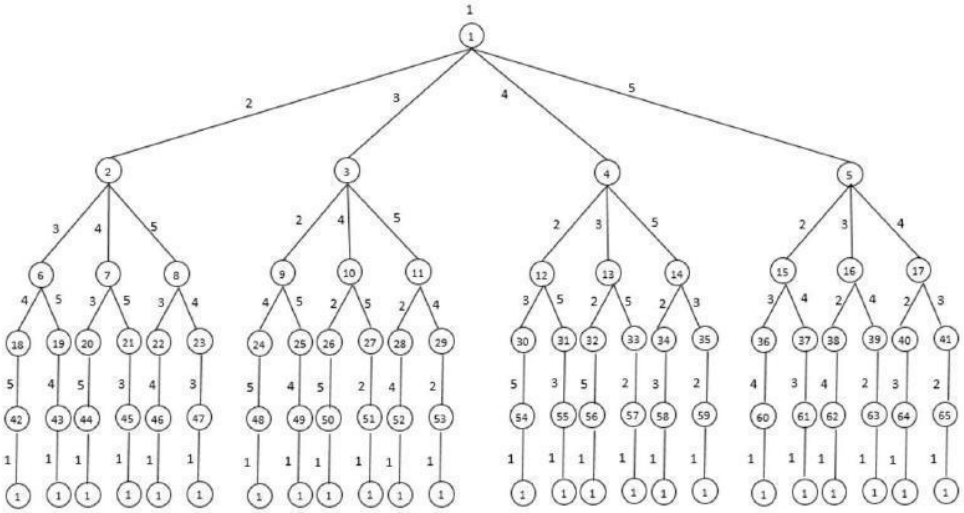
Column reduced matrix M_{ColRed} would be:

$M_{ColRed} =$

∞	10	17	0	1
12	∞	11	2	0
0	3	∞	0	2
15	3	12	∞	0
11	0	0	12	∞

Each row and column of M_{ColRed} has at least one zero entry, so this matrix is reduced matrix.
Column reduction cost (M) = 1 + 0 + 3 + 0 + 0 = 4

State space tree for 5 city problem is depicted in Fig. 6.6.1. Number within circle indicates the order in which the node is generated, and number of edge indicates the city being visited.



Example

Example: Find the solution of following travelling salesman problem using branch and bound method.

Solution:

- The procedure for dynamic reduction is as follow:
- Draw state space tree with optimal reduction cost at root node.
- Derive cost of path from node i to j by setting all entries in i^{th} row and j^{th} column as ∞.

Set M[j][i] = ∞

- Cost of corresponding node N for path i to j is summation of optimal cost + reduction cost + M[j][i]
- After exploring all nodes at level i, set node with minimum cost as E node and repeat the procedure until all nodes are visited.
- Given matrix is not reduced. In order to find reduced matrix of it, we will first find the row reduced matrix followed by column reduced matrix if needed. We can find row reduced matrix by subtracting minimum element of each row from each element of corresponding row. Procedure is described below:
- Reduce above cost matrix by subtracting minimum value from each row and column.

∞	20	30	10	11	→ 10	∞	10	20	0	1	
15	∞	16	4	2	→ 2	13	∞	14	2	0	
3	5	∞	2	4	→ 2	⇒ 1	3	∞	0	2	= M_1'
19	6	18	∞	3	→ 3	16	3	15	∞	0	
16	4	7	16	∞	→ 4	12	0	3	12	∞	
						↓	↓	↓	↓	↓	
						1	0	3	0	0	

M‘1

is not reduced matrix. Reduce it subtracting minimum value from corresponding column. Doing this we get,

∞	10	17	0	1
12	∞	11	2	0
0	3	∞	0	2
15	3	12	∞	0
11	0	0	12	∞

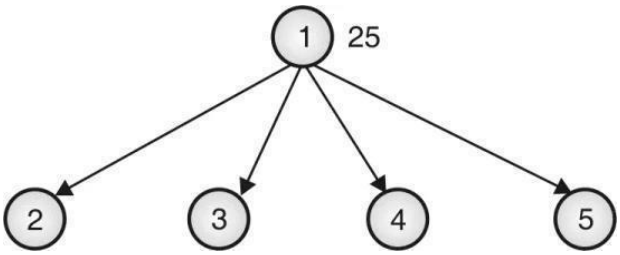
$= M_1$

Cost of M_1 = C(1)

- Row reduction cost + Column reduction cost
- (10+2+2+3+4)+(1+3)=25

This means all tours in graph has length at least 25. This is the optimal cost of the path.

State space tree



Let us find cost of edge from node 1 to 2, 3, 4, 5.

Select edge 1-2:

Set M_1 [1] [ ] = M_1 [ ] [2] = ∞

Set M_1 [2] [1] = ∞

Reduce the resultant matrix if required.

∞	∞	∞	∞	∞	→ x
∞	∞	11	2	0	→ 0
0	∞	∞	0	2	→ 0
15	∞	12	∞	0	→ 0
11	∞	0	12	∞	→ 0
↓	↓	↓	↓	↓	
0	x	0	0	0	

$= M_2$

M_2 is already reduced.

Cost of node 2 :

C(2) = C(1) + Reduction cost + M_1 [1] [2]

• 25+0+10=35

Select edge 1-3

Set M_1 [1][ ] = M_1 [ ] [3] = ∞ Set
M_1 [3][1] = ∞

Reduce the resultant matrix if required.

$M_1 \Rightarrow$

∞	∞	∞	∞	∞	→ x
12	∞	11	2	0	→ 0
0	∞	∞	0	2	→ 0
15	∞	12	∞	0	→ 0
11	∞	0	12	∞	→ 0
↓	↓	↓	↓	↓	
11	0	x	0	0	

$\Rightarrow$

∞	∞	∞	∞	∞
1	∞	∞	2	0
∞	3	∞	0	2
4	3	∞	∞	0
0	0	∞	12	∞

$= M_3$

Cost of node 3:

C(3) = C(1) + Reduction cost + M_1[1] [3]

=25+11+17=53

Select edge 1-4:

Set M_1 [1][ ] = M_1[ ][4] = ∞

Set M_1 [4][1] = ∞

Reduce resultant matrix if required.

	∞	∞	∞	∞	∞	→ x
	12	∞	11	∞	0	→ 0
$M_1 \Rightarrow$	0	3	∞	∞	2	→ 0
	∞	3	12	∞	0	→ 0
	11	0	0	∞	∞	→ 0
	↓	↓	↓	↓	↓	
	0	0	0	x	0	

$= M_4$

Matrix M_4 is already reduced.

Cost of node 4:

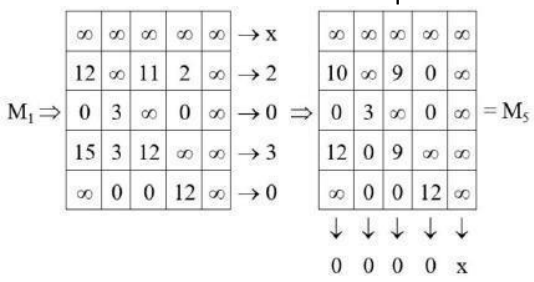
C(4) = C(1) + Reduction cost + M_1 [1] [4]

- 25+0+0=25

Select edge 1-5:

Set M_1 [1] [ ] = M_1 [ ] [5] = ∞
Set M_1 [5] [1] = ∞

Reduce the resultant matrix if required.

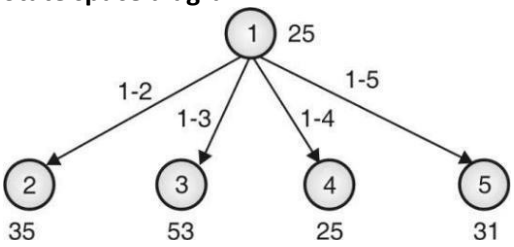


Cost of node 5:

C(5) = C(1) + reduction cost + M_1 [1] [5]

- 25+5+1=31

State space diagram:



Node 4 has minimum cost for path 1-4. We can go to vertex 2, 3 or 5. Let's explore all three nodes.

Select path 1-4-2 : (Add edge 4-2)

Set M_4	[1]	[]	= M_4[4][]=M_4[][2]=∞
Set M_4	[2]	[1]	= ∞

Reduce resultant matrix if required.

$M_4 \Rightarrow$

∞	∞	∞	∞	∞	→ x
∞	∞	11	∞	0	→ 0
0	∞	∞	∞	2	→ 0
∞	∞	∞	∞	∞	→ x
11	∞	0	∞	∞	→ 0
↓	↓	↓	↓	↓	
0	0	0	x	0	

$= M_6$

Matrix M_6 is already reduced.

Cost of node 6:

C(6) = C(4) + Reduction cost + M_4 [4] [2]

= 25+0+3	= 28	
Select edge 4-3	**(Path 1-4-3)**:	
Set M_4 [1] [ ]	=	M_4[4][]=M_4[][3]=∞
Set M [3][1]	=	∞

Reduce the resultant matrix if required.

$M_4 \Rightarrow$

∞	∞	∞	∞	∞	→ x
12	∞	∞	∞	0	→ 0
∞	3	∞	∞	2	→ 2
∞	∞	∞	∞	∞	→∞
11	0	∞	∞	∞	→ 0

$\Rightarrow$

∞	∞	∞	∞	∞
12	∞	∞	∞	0
∞	1	∞	∞	0
∞	∞	∞	∞	∞
11	0	∞	∞	∞
↓	↓	↓	↓	↓
11	0	x	x	0

$= M_7'$

M‘7

is not reduced. Reduce it by subtracting 11 from column 1.

$$\therefore M_7' \Rightarrow \begin{array}{|c|c|c|c|c|} \hline \infty & \infty & \infty & \infty & \infty \\ \hline 1 & \infty & \infty & \infty & 0 \\ \hline \infty & 1 & \infty & \infty & 2 \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline 0 & 0 & \infty & \infty & \infty \\ \hline \end{array} = M_7$$

Cost of node 7:

C(7) = C(4) + Reduction cost + M_4 [4] [3]

- 25+2+11+12=50

Select edge 4-5 (Path 1-4-5):

$$M_4 \Rightarrow \begin{array}{|c|c|c|c|c|l|} \hline \infty & \infty & \infty & \infty & \infty & \rightarrow x \\ \hline 12 & \infty & 11 & \infty & \infty & \rightarrow 11 \\ \hline 0 & 3 & \infty & \infty & \infty & \rightarrow 0 \\ \hline \infty & \infty & \infty & \infty & \infty & \rightarrow x \\ \hline \infty & 0 & 0 & \infty & \infty & \rightarrow 0 \\ \hline \end{array} \Rightarrow \begin{array}{|c|c|c|c|c|} \hline \infty & \infty & \infty & \infty & \infty \\ \hline 1 & \infty & 0 & \infty & \infty \\ \hline 0 & 3 & \infty & \infty & \infty \\ \hline \infty & \infty & \infty & \infty & \infty \\ \hline \infty & 0 & 0 & \infty & \infty \\ \hline \end{array} = M_8$$

$$\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow$$

$$0 \quad 0 \quad 0 \quad x \quad x$$

Matrix M_8 is reduced.

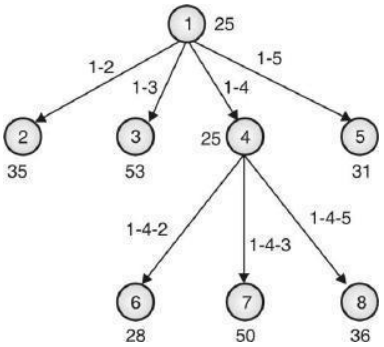
Cost of node 8:

C(8) = C(4) + Reduction cost + M_4 [4][5]

- 25+11+0=36

State space tree

Path 1-4-2 leads to minimum cost. Let's find the cost for two possible paths.



Add edge 2-3 (Path 1-4-2-3):

Set M_6 [1][ ] = M_6 [4][ ] = M_6 [2][ ]

= M_6 [][3] = ∞

Set M_6 [3][1] = ∞

Reduce resultant matrix if required.

$M_6 \Rightarrow$

∞	∞	∞	∞	∞	→ x
∞	∞	∞	∞	∞	→ x
0	∞	∞	∞	2	→ 0
∞	∞	∞	∞	∞	→ x
11	∞	∞	∞	∞	→ 11

$\Rightarrow$

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
0	∞	∞	∞	2
∞	∞	∞	∞	∞
0	∞	∞	∞	∞
↓	↓	↓	↓	↓
0	x	x	x	2

$= M_9'$

$\therefore M_9' \Rightarrow$

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
0	∞	∞	∞	0
∞	∞	∞	∞	∞
0	∞	∞	∞	∞

$= M_9$

Cost of node 9:

C(9) = C(6) + Reduction cost + M_6 [2][3]

- 28+11+2+11=52 **Add edge 2-5 (Path 1-4-2-5)**:

Set M_6 [1][ ] = M_6 [4][ ] = M_6 [2][ ] = M_6 [ ][5] = ∞ Set M_6 [5][1] = ∞

Reduce resultant matrix if required.

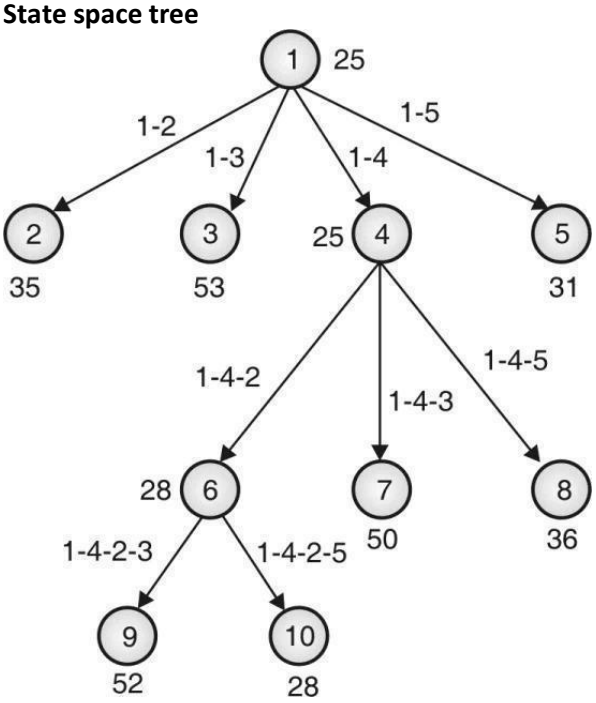
$$\therefore M_6 \Rightarrow \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix} = M_{10}$$

Cost of node 10:

C(10) = C(6) + Reduction cost + M_6 [2][5]

- 28+0+0=28

State space tree



Add edge 5-3 (Path 1-4-2-5-3):

$\therefore\ M_{10} \Rightarrow$

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞

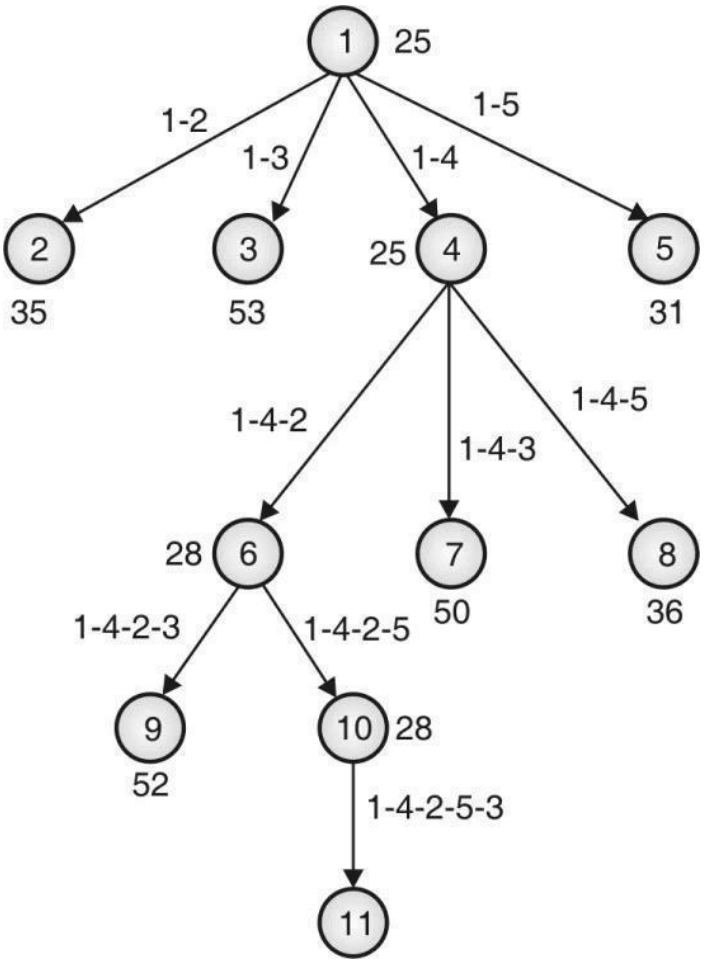
$= M_{11}$

Cost of node 11:

C(11) = C(10) + Reduction cost + M_{10} [5][3]

=28+0+0=28

State space tree:



So we can select any of the edge. Thus the final path includes the edges <3, 1>, <5, 3>, <1, 4>, <4, 2>, <2, 5>, that forms the path 1 – 4 – 2 – 5 – 3 – 1. This path has cost of 28.

UNIT 5

Tractable and Intractable Problems

Tractable problems refer to computational problems that can be solved efficiently using algorithms that can scale with the input size of the problem. In other words, the time required to solve a tractable problem increases at most polynomially with the input size.

On the other hand, intractable problems are computational problems for which no known algorithm can solve them efficiently in the worst-case scenario. This means that the time required to solve an intractable problem grows exponentially or even faster with the input size.

One example of a tractable problem is computing the sum of a list of n numbers. The time required to solve this problem scales linearly with the input size, as each number can be added to a running total in constant time. Another example is computing the shortest path between two nodes in a graph, which can be solved efficiently using algorithms like Dijkstra's algorithm or the A* algorithm.

In contrast, some well-known intractable problems include the traveling salesman problem, the knapsack problem, and the Boolean satisfiability problem. These problems are NP-hard, meaning that any problem in NP (the set of problems that can be solved in polynomial time using a non-deterministic Turing machine) can be reduced to them in polynomial time. While it is possible to find approximate solutions to these problems, there is no known algorithm that can solve them exactly in polynomial time.

In summary, tractable problems are those that can be solved efficiently with algorithms that scale well with the input size, while intractable problems are those that cannot be solved efficiently in the worst-case scenario.

Examples of Tractable problems

- Sorting: Given a list of n items, the task is to sort them in ascending or descending order. Algorithms like QuickSort and MergeSort can solve this problem in O(n log n) time complexity.

- Matrix multiplication: Given two matrices A and B, the task is to find their product C = AB. The best-known algorithm for matrix multiplication runs in O(n^2.37) time complexity, which is considered tractable for practical applications.

- Shortest path in a graph: Given a graph G and two nodes s and t, the task is to find the shortest path between s and t. Algorithms like Dijkstra's algorithm and the A* algorithm can solve this problem in O(m + n log n) time complexity, where m is the number of edges and n is the number of nodes in the graph.

- Linear programming: Given a system of linear constraints and a linear objective function, the task is to find the values of the variables that optimize the objective function subject to the constraints. Algorithms like the simplex method can solve this problem in polynomial time.

- Graph coloring: Given an undirected graph G, the task is to assign a color to each node such that no two adjacent nodes have the same color, using as few colors as possible. The greedy algorithm can solve this problem in O(n^2) time complexity, where n is the number of nodes in the graph.

These problems are considered tractable because algorithms exist that can solve them in polynomial time complexity, which means that the time required to solve them grows no faster than a polynomial function of the input size.

Examples of intractable problems

- Traveling salesman problem (TSP): Given a set of cities and the distances between them, the task is to find the shortest possible route that visits each city exactly once and returns to the starting city. The best-known algorithms for solving the TSP have an exponential worst-case time complexity, which makes it intractable for large instances of the problem.

- Knapsack problem: Given a set of items with weights and values, and a knapsack that can carry a maximum weight, the task is to find the most valuable subset of items that can be carried by the knapsack. The knapsack problem is also NP-hard and is intractable for large instances of the problem.

- Boolean satisfiability problem (SAT): Given a boolean formula in conjunctive normal form (CNF), the task is to determine if there exists an assignment of truth values to the variables that makes

the formula true. The SAT problem is one of the most well-known NP-complete problems, which means that any NP problem can be reduced to SAT in polynomial time.

- Subset sum problem: Given a set of integers and a target sum, the task is to find a subset of the integers that sums up to the target sum. Like the knapsack problem, the subset sum problem is also intractable for large instances of the problem.

- Graph isomorphism problem: Given two graphs G1 and G2, the task is to determine if there

- Linear search: Given a list of n items, the task is to find a specific item in the list. The time complexity of linear search is O(n), which is a polynomial function of the input size.

- Bubble sort: Given a list of n items, the task is to sort them in ascending or descending order. The time complexity of bubble sort is O(n^2), which is also a polynomial function of the input size.

- Shortest path in a graph: Given a graph G and two nodes s and t, the task is to find the shortest path between s and t. Algorithms like Dijkstra's algorithm and the A* algorithm can solve this problem in O(m + n log n) time complexity, which is a polynomial function of the input size.

- Maximum flow in a network: Given a network with a source node and a sink node, and capacities on the edges, the task is to find the maximum flow from the source to the sink. The Ford-Fulkerson algorithm can solve this problem in O(mf), where m is the number of edges in the network and f is the maximum flow, which is also a polynomial function of the input size.

- Linear programming: Given a system of linear constraints and a linear objective function, the task is to find the values of the variables that optimize the objective function subject to the constraints. Algorithms like the simplex method can solve this problem in polynomial time.

P (Polynomial) problems

P problems refer to problems where an algorithm would take a polynomial amount of time to solve, or where Big-O is a polynomial (i.e. O(1), O(n), O(n^2), etc). These are problems that would be considered ‘easy’ to solve, and thus do not generally have immense run times.

NP (Non-deterministic Polynomial) Problems

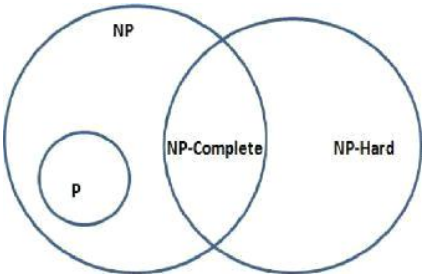
NP problems were a little harder for me to understand, but I think this is what they are. In terms of solving a NP problem, the run-time would not be polynomial. It would be something like O(n!) or something much larger.

NP-Hard Problems

A problem is classified as NP-Hard when an algorithm for solving it can be translated to solve *any* NP problem. Then we can say, this problem is *at least* as hard as any NP problem, but it could be much harder or more complex.

NP-Complete Problems

NP-Complete problems are problems that live in both the NP and NP-Hard classes. This means that NP-Complete problems can be verified in polynomial time and that any NP problem can be reduced to this problem in polynomial time.



Bin Packing problem

Bin Packing problem involves assigning n items of different weights and bins each of capacity c to a bin such that number of total used bins is minimized. It may be assumed that all items have weights smaller than bin capacity.

The following 4 algorithms depend on the order of their inputs. They pack the item given first and then move on to the next input or next item

1) Next Fit algorithm

The simplest approximate approach to the bin packing problem is the Next-Fit (NF)

algorithm which is explained later in this article. The first item is assigned to bin 1. Items

2,... ,n are then considered by increasing indices : each item is assigned to the current bin, if it fits; otherwise, it is assigned to a new bin, which becomes the current one.

Visual Representation

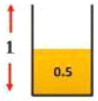
Let us consider the same example as used above and bins of size 1

Assuming the sizes of the items be {0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6}.

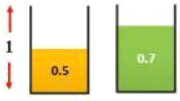
The minimum number of bins required would be Ceil ((Total Weight) / (Bin Capacity))= Celi(3.7/1) = 4 bins.

The Next fit solution (NF(I))for this instance I would be-

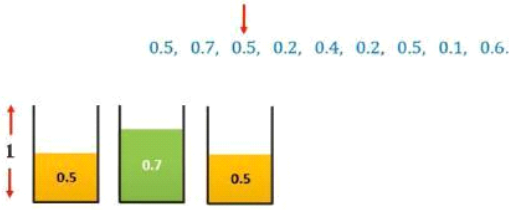
Considering 0.5 sized item first, we can place it in the first bin



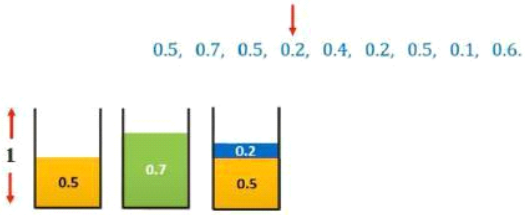
Moving on to the 0.7 sized item, we cannot place it in the first bin. Hence we place it in a new bin.



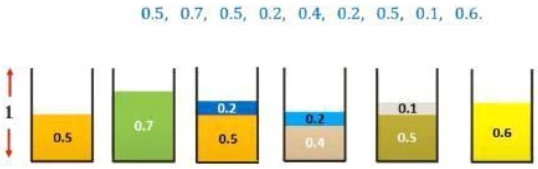
Moving on to the 0.5 sized item, we cannot place it in the current bin. Hence we place it in a new bin.



Moving on to the 0.2 sized item, we can place it in the current (third bin)



Similarly, placing all the other items following the Next-Fit algorithm we get-



Thus we need 6 bins as opposed to the 4 bins of the optimal solution. Thus we can see that this algorithm is not very efficient.

Analyzing the approximation ratio of Next-Fit algorithm

The time complexity of the algorithm is clearly O(n). It is easy to prove that, for any instance I of BPP,the solution value NF(I) provided by the algorithm satisfies the bound

NF(I)<2z(I)

where z(I) denotes the optimal solution value. Furthermore, there exist instances for which the ratio NF(I)/z(I) is arbitrarily close to 2, i.e. the worst-case approximation ratio of NF is r(NF) = 2.

Psuedocode

NEXT FIT (size[], n, c)

size[] is the array containg the sizes of the items, n is the number of items and c is the

capacity of the bin

{

Initialize result (Count of bins) and remaining capacity in current bin.

res = 0

bin_rem = c

Place items one by one

for (int i = 0; i < n; i++) {

- If this item can't fit in current bin if (size[i] > bin_rem) {

Use a new bin

```
            res++
            bin_rem = c - size[i]
        }
        else
            bin_rem -= size[i];
    }
    return res;
}
```

2) First Fit algorithm

A better algorithm, First-Fit (FF), considers the items according to increasing indices and assigns each item to the lowest indexed initialized bin into which it fits; only when the current item cannot fit into any initialized bin, is a new bin introduced

Visual Representation

Let us consider the same example as used above and bins of size 1

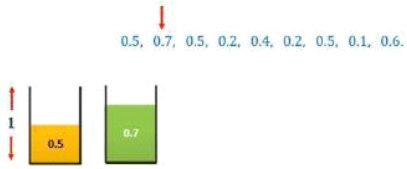
Assuming the sizes of the items be {0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6}.

The minimum number of bins required would be Ceil ((Total Weight) / (Bin Capacity))= Celi(3.7/1) = 4 bins.

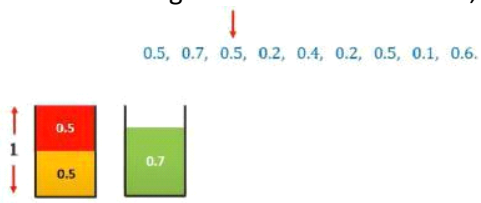
The First fit solution (FF(I))for this instance I would be-

Considering 0.5 sized item first, we can place it in the first bin

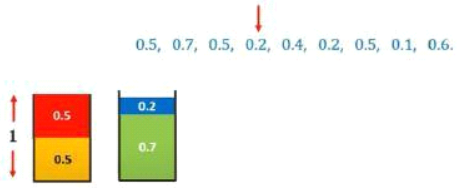
Moving on to the 0.7 sized item, we cannot place it in the first bin. Hence we place it in a new bin.



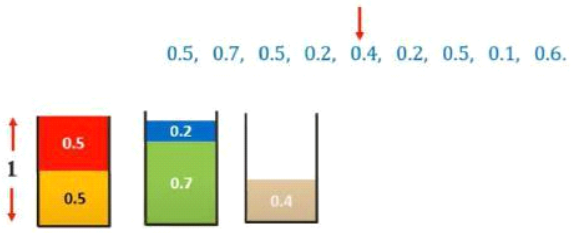
Moving on to the 0.5 sized item, we can place it in the first bin.



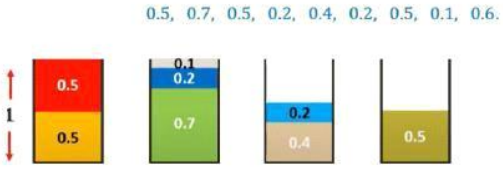
Moving on to the 0.2 sized item, we can place it in the first bin, we check with the second bin and we can place it there.



Moving on to the 0.4 sized item, we cannot place it in any existing bin. Hence we place it in a new bin.



Similarly, placing all the other items following the First-Fit algorithm we get-



Thus we need 5 bins as opposed to the 4 bins of the optimal solution but is much more efficient than Next-Fit algorithm.

Analyzing the approximation ratio of Next-Fit algorithm

If FF(I) is the First-fit implementation for I instance and z(I) is the most optimal solution, then:

$$FF(I) \leq \frac{17}{10} z(I) + 2$$

for all instances I of BPP, and that there exist instances I, with $z(I)$ arbitrarily large, for which

$$FF(I) > \frac{17}{10} z(I) - 8.$$

It can be seen that the First Fit never uses more than 1.7 * z(I) bins. So First-Fit is better than Next Fit in terms of upper bound on number of bins.

Psuedocode

```
FIRSTFIT(size[], n, c)

{

size[] is the array containg the sizes of the items, n is the number of items and c is the capacity of the bin

  /Initialize result (Count of bins)


  res = 0;

   Create an array to store remaining space in bins there can be at most n bins
  bin_rem[n];

  Plae items one by one

  for (int i = 0; i < n; i++) {

    Find the first bin that can accommodate weight[i]

    int j;

    for (j = 0; j < res; j++) {
```

```
            if (bin_rem[j] >= size[i]) {
                bin_rem[j] = bin_rem[j] - size[i];
                break;
            }
        }
        If no bin could accommodate size[i]
        if (j == res) {
            bin_rem[res] = c - size[i];
            res++;
        }
    }
    return res;
}
```

3) Best Fit Algorithm

The next algorithm, Best-Fit (BF), is obtained from FF by assigning the current item to the feasible bin (if any) having the smallest residual capacity (breaking ties in favor of the lowest indexed bin).

Simply put,the idea is to places the next item in the *tightest* spot. That is, put it in the bin so that the smallest empty space is left.

Visual Representation

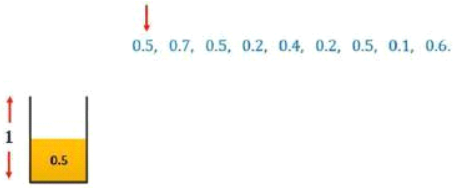
Let us consider the same example as used above and bins of size 1

Assuming the sizes of the items be {0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6}.

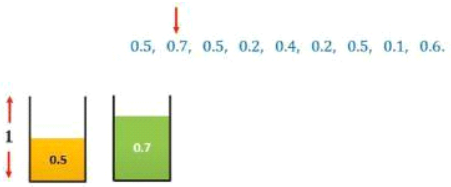
The minimum number of bins required would be Ceil ((Total Weight) / (Bin Capacity))= Celi(3.7/1) = 4 bins.

The First fit solution (FF(I))for this instance I would be-

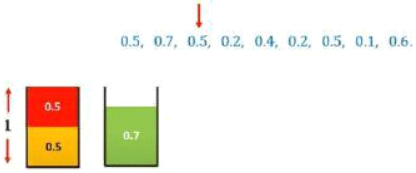
Considering 0.5 sized item first, we can place it in the first bin



Moving on to the 0.7 sized item, we cannot place it in the first bin. Hence we place it in a new bin.

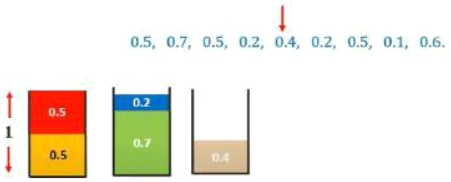


Moving on to the 0.5 sized item, we can place it in the first bin tightly.

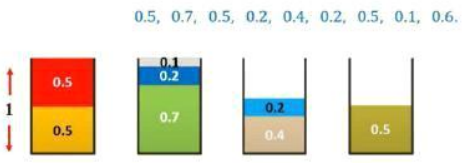


Moving on to the 0.2 sized item, we cannot place it in the first bin but we can place it in second bin tightly.

Moving on to the 0.4 sized item, we cannot place it in any existing bin. Hence we place it in a new bin.



Similarly, placing all the other items following the First-Fit algorithm we get-



Thus we need 5 bins as opposed to the 4 bins of the optimal solution but is much more efficient than Next-Fit algorithm.

Analyzing the approximation ratio of Best-Fit algorithm

It can be noted that Best-Fit (BF), is obtained from FF by assigning the current item to the feasible bin (if any) having the smallest residual capacity (breaking ties in favour of the lowest indexed bin). BF satisfies the same worst-case bounds as FF

Analysis Of upper-bound of Best-Fit algorithm

If z(I) is the optimal number of bins, then Best Fit never uses more than 2 * z(I)-2 bins. So Best Fit is same as Next Fit in terms of upper bound on number of bins.

Psuedocode

```
BESTFIT(size[],n, c)

{

size[] is the array containg the sizes of the items, n is the number of items and c is the capacity of the bin

   Initialize result (Count of bins)

   res = 0;

   Create an array to store remaining space in bins there can be at most n bins
   bin_rem[n];

   Place items one by one

   for (int i = 0; i < n; i++) {

      Find the best bin that can accommodate weight[i]
```

```
    int j;

    Initialize minimum space left and index of best bin

    int min = c + 1, bi = 0;

    for (j = 0; j < res; j++) {

      if (bin_rem[j] >= size[i] && bin_rem[j] - size[i] < min) { bi
        = j;

        min = bin_rem[j] - size[i];

      }

    }

    If no bin could accommodate weight[i],create a new bin if
    (min == c + 1) {

      bin_rem[res] = c - size[i];

      res++;

    }

    else

      Assign the item to best bin

      bin_rem[bi] -= size[i];

  }


  return res;

}
```

in the offline version, we have all items at our disposal since the start of the execution. The natural solution is to sort the array from largest to smallest, and then apply the algorithms discussed henceforth.

NOTE: In the online programs we have given the inputs upfront for simplicity but it can also work interactively

Let us look at the various offline algorithms

1) First Fit Decreasing

We first sort the array of items in decreasing size by weight and apply first-fit algorithm as discussed above

Algorithm

Read the inputs of items

Sort the array of items in decreasing order by their sizes

Apply First-Fit algorithm

Visual Representation

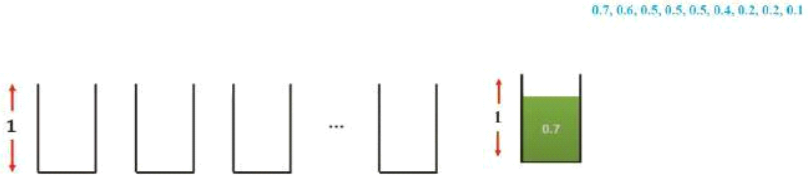
Let us consider the same example as used above and bins of size 1

Assuming the sizes of the items be {0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6}.

Sorting them we get {0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1}

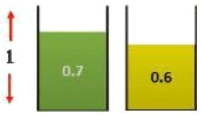
The First fit Decreasing solution would be-

We will start with 0.7 and place it in the first bin



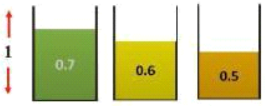
We then select 0.6 sized item. We cannot place it in bin 1. So, we place it in bin 2

0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1



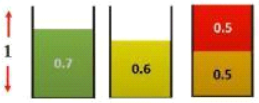
We then select 0.5 sized item. We cannot place it in any existing. So, we place it in bin 3

0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1



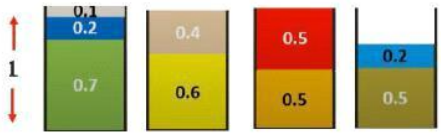
We then select 0.5 sized item. We can place it in bin 3

0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1



Doing the same for all items, we get.

0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1



Thus only 4 bins are required which is the same as the optimal solution.

2) Best Fit Decreasing

We first sort the array of items in decreasing size by weight and apply Best-fit algorithm as discussed above

Algorithm

Read the inputs of items

Sort the array of items in decreasing order by their sizes

Apply Next-Fit algorithm

Visual Representation

Let us consider the same example as used above and bins of size 1

Assuming the sizes of the items be {0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6}.

Sorting them we get {0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1}

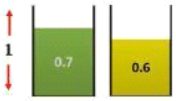
The Best fit Decreasing solution would be-

We will start with 0.7 and place it in the first bin

0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1

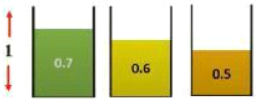
We then select 0.6 sized item. We cannot place it in bin 1. So, we place it in bin 2

0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1

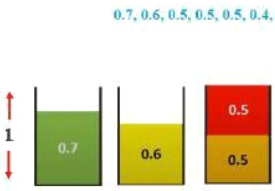


We then select 0.5 sized item. We cannot place it in any existing. So, we place it in bin 3

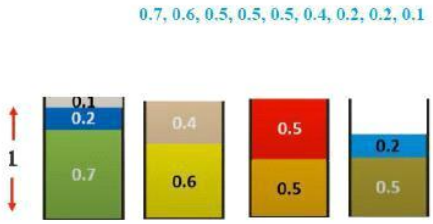
0.7, 0.6, 0.5, 0.5, 0.5, 0.4, 0.2, 0.2, 0.1



We then select 0.5 sized item. We can place it in bin 3



Doing the same for all items, we get.



Thus only 4 bins are required which is the same as the optimal solution.

Approximation Algorithms for the Traveling Salesman Problem

We solved the traveling salesman problem by exhaustive search in Section 3.4, mentioned its decision version as one of the most well-known *NP*-complete problems in Section 11.3, and saw how its instances can be solved by a branch-and-bound algorithm in Section 12.2. Here, we consider several approximation algorithms, a small sample of dozens of such algorithms suggested over the years for this famous problem.

But first let us answer the question of whether we should hope to find a polynomial-time approximation algorithm with a finite performance ratio on all instances of the traveling salesman problem. As the following theorem [Sah76] shows, the answer turns out to be no, unless ***P*** = ***N P*** .

THEOREM 1 If ***P*** != *NP*, there exists no ***c***-approximation algorithm for the traveling salesman problem, i.e., there exists no polynomial-time approximation algorithm for this problem so that for all instances

$$f(s_a) \leq cf(s^*)$$

for some constant c.

Nearest-neighbour algorithm

The following well-known greedy algorithm is based on the ***nearest-neighbor*** heuristic:

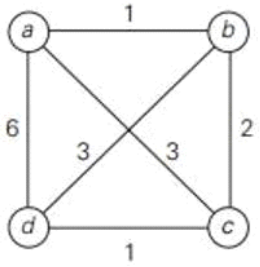
always go next to the nearest unvisited city.

Step 1 Choose an arbitrary city as the start.

Step 2 Repeat the following operation until all the cities have been visited: go to the unvisited city nearest the one visited last (ties can be broken arbitrarily).

Step 3 Return to the starting city.

EXAMPLE 1 For the instance represented by the graph in Figure 12.10, with ***a*** as the starting vertex, the nearest-neighbor algorithm yields the tour (Hamiltonian circuit) s_a: ***a*** − ***b*** − ***c*** − ***d*** − ***a*** of length 10.



The optimal$_*$ solution, as can be easily checked by exhaustive search, is the tour s : $a - b - d - c - a$ of length 8. Thus, the accuracy ratio of this approximation is

Unfortunately, except for its simplicity, not many good things can be said about the nearest-neighbor algorithm. In particular, nothing can be said in general about the accuracy of solutions obtained by this algorithm because it can force us to traverse a very long edge on the last leg of the tour. Indeed, if we change the weight of edge ***(a, d)*** from 6 to an arbitrary large number $w \geq 6$ in Example 1, the algorithm will still yield the tour $a - b - c - d - a$ of length 4 + ***w,*** and the optimal solution will still be $a - b - d - c - a$ of length 8. Hence,

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{4 + w}{8},$$

which can be made as large as we wish by choosing an appropriately large value of ***w***.

Hence, $R_A = \infty$ for this algorithm (as it should be according to Theorem 1).

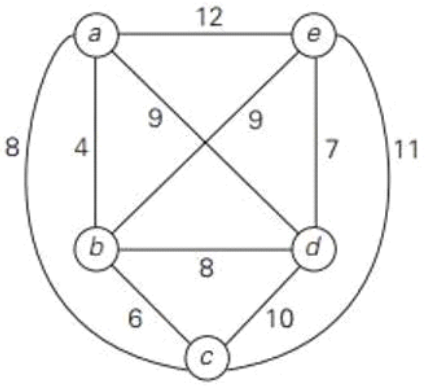
Twice-around-the-tree algorithm

Step 1 Construct a minimum spanning tree of the graph corresponding to a given instance of the traveling salesman problem.

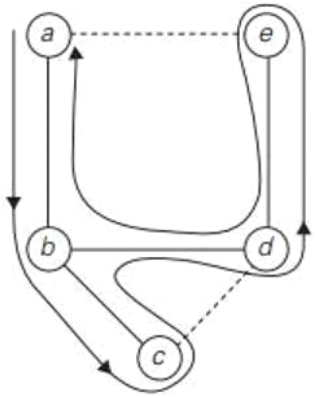
Step 2 Starting at an arbitrary vertex, perform a walk around the minimum spanning tree recording all the vertices passed by. (This can be done by a DFS traversal.)

Step 3 Scan the vertex list obtained in Step 2 and eliminate from it all repeated occurrences of the same vertex except the starting one at the end of the list. (This step is equivalent to making shortcuts in the walk.) The vertices remaining on the list will form a Hamiltonian circuit, which is the output of the algorithm.

EXAMPLE 2 Let us apply this algorithm to the graph in Figure 12.11a. The minimum spanning tree of this graph is made up of edges ***(a, b), (b, c)***, ***(b, d)***, and ***(d, e) .*** A twice-



(a)



(b)

around-the-tree walk that starts and ends at ***a*** is

a, b, c, b, d, e, d, b, a.

Eliminating the second ***b*** (a shortcut from ***c*** to ***d***), the second ***d***, and the third ***b*** (a shortcut from ***e*** to ***a***) yields the Hamiltonian circuit

a, b, c, d, e, a

of length 39.

The tour obtained in Example 2 is not optimal. Although that instance is small enough to find an optimal solution by either exhaustive search or branch-and-bound, we refrained from doing so to reiterate a general point. As a rule, we do not know what the length of an $*$ optimal tour actually is, and therefore we cannot compute the accuracy ratio $f(s_a)/f(s)$. For the twice-around-the-tree algorithm, we can at least estimate it above, provided the graph is Euclidean.

Fermat's Little Theorem:

If n is a prime number, then for every a, 1 < a < n-1,

$a^{n-1} \equiv 1 \pmod{n}$

OR

$a^{n-1} \% n = 1$

Example: Since 5 is prime, $2^4 \equiv 1 \pmod 5$ [or $2^4 \% 5 = 1$],

$3^4 \equiv 1 \pmod 5$ and $4^4 \equiv 1 \pmod 5$

Since 7 is prime, $2^6 \equiv 1 \pmod 7$,

$3^6 \equiv 1 \pmod 7$, $4^6 \equiv 1 \pmod 7$

$5^6 \equiv 1 \pmod 7$ and $6^6 \equiv 1 \pmod 7$

Algorithm

- Repeat following k times:
  - Pick a randomly in the range [2, n - 2]
  - If gcd(a, n) ≠ 1, then return false
  - If a^{n-1} ≢ 1 (mod n), then return false
- Return true [probably prime].

Unlike merge sort, we don't need to merge the two sorted arrays. Thus Quicksort requires lesser auxiliary space than Merge Sort, which is why it is often preferred to Merge Sort. Using a randomly generated pivot we can further improve the time complexity of QuickSort.

Algorithm for random pivoting

partition(arr[], lo, hi)

```
pivot = arr[hi]
i = lo    // place for swapping
```

```
    for j := lo to hi – 1 do
        if arr[j] <= pivot then
            swap arr[i] with arr[j]
            i = i + 1
    swap arr[i] with arr[hi]
    return i
partition_r(arr[], lo, hi)
    r = Random Number from lo to hi
    Swap arr[r] and arr[hi]
    return partition(arr, lo, hi)
quicksort(arr[], lo, hi)
    if lo < hi
        • = partition_r(arr, lo, hi)
        quicksort(arr, lo , p-1)
        quicksort(arr, p+1, hi)
```

Finding kth smallest element

Problem Description: Given an array A[] of n elements and a positive integer K, find the Kth smallest element in the array. It is given that all array elements are distinct.

For Example :

Input : A[] = {10, 3, 6, 9, 2, 4, 15, 23}, K = 4

Output: 6

Input : A[] = {5, -8, 10, 37, 101, 2, 9}, K = 6

Output: 37

Quick-Select : Approach similar to quick sort

This approach is similar to the quick sort algorithm where we use the partition on the input array recursively. But unlike quicksort, which processes both sides of the array recursively, this

algorithm works on only one side of the partition. We recur for either the left or right side according to the position of pivot.

Solution Steps

- Partition the array A[left .. right] into two subarrays A[left .. pos] and A[pos + 1 .. right] such that each element of A[left .. pos] is less than each element of A[pos + 1 .. right].
- Computes the number of elements in the subarray A[left .. pos] i.e. count = pos - left + 1
- if (count == K), then A[pos] is the Kth smallest element.
- Otherwise determines in which of the two subarrays A[left .. pos-1] and A[pos + 1 .. right] the Kth smallest element lies.

If (count > K) then the desired element lies on the left side of the partition

If (count < K), then the desired element lies on the right side of the partition. Since we already know i values that are smaller than the kth smallest element of A[left .. right], the desired element is the (K - count)th smallest element of A[pos + 1 .. right].

Base case is the scenario of single element array i.e left ==right. return A[left] or A[right].

Pseudo-Code

- *Original value for left = 0 and right = n-1* **int** **kthSmallest**(**int** A[], **int** left, **int** right, **int** K)

```
{

  if (left == right)
     return A[left]

  int pos = partition(A, left, right)
  count = pos - left + 1

  if ( count == K )
     return A[pos]
  else if ( count > K )

     return kthSmallest(A, left, pos-1, K)
  else

     return kthSmallest(A, pos+1, right, K-i)

}
```

```
int partition(int A[], int l, int r)
{
   int x = A[r]
   int i = l-1
   for ( j = l to r-1 )
   {
      if (A[j] <= x)
      {
         i = i + 1
         swap(A[i], A[j])
      }
   }
   swap(A[i+1], A[r])
   return i+1
}
```

Complexity Analysis

Time Complexity: The worst-case time complexity for this algorithm is $O(n^2)$, but it can be improved if we choose the pivot element randomly. If we randomly select the pivot, the expected time complexity would be linear, **O(n).**