Design and Analysis of Algorithm notes

Written by: Pratik Antoni Patekar

Table of contents:

Introduction to algorithms,
Sorting algorithms,
Time complexity,
Space complexity and
Asymptotic notations.

Introduction to Algorithms

What are algorithms?
An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. (Reference book definition).

Why do we need to study algorithms?
There are many reasons why we should study algorithms but here are few of them:

1. **Learn and build logic for solving a problem**. The algorithms can be used as building blocks while forming a logic and thus save time spent on unnecessary basic logic blocks.
2. Algorithm analysis gives information regarding their performance. Thus we can use specific algorithms in order to fine tune the performances or in other words **improve the performance**.
3. **Scalability**.
4. Performance can be considered analogous to currency. We can achieve only a limited amount of improvement in the performance of a code/ script.
5. On studying algorithms, we can understand whether a given problem is feasible or impossible to be achieved.

In this course, we will study the theoretical proofs of a few algorithms.

Course structure or grading:

Task	Weightage
1 project	30%
1 midterm exam	25%
1 final exam	40%
Attendance	5%
Total	**100%**

In this course we will be talking about performance and how it can be improved but in reality there are multiple things that are more important than performance. Following are a few which can be listed:

1. Correctness of the output
2. Modularity
3. Satisfying the constraints given by the client
4. Scalability
5. Time required by programmer to write the program
6. Robustness of code and so on.

Sorting algorithms: Sorting algorithms are used to sort a given list/ array of elements.Sorting is arranging the elements of the given array in ascending order (or in some cases descending order). Here elements can be anything i.e. numbers, strings, some data type such as alphanumeric strings. For a sorting algorithm the input and output can be defined as follows:
Input: $a_1, a_2, \ldots\ldots, a_n$
Output: $a'_1, a'_2, \ldots\ldots, a'_n$ where $a'_i \leq a'_{i+1}$

Properties of sorting:

1. **Stable and non stable:** An algorithm is said to be stable if the relative order of the items with equal key values do not change after sorting.
2. **Adaptive and non adaptive:** An algorithm is said to be adaptive if the time complexity depends on the input, i.e., if the input array is almost sorted then it will require less time for sorting.

Other aspects of sorting:

Time complexity: If the sorting algorithm is non adaptive then the time complexity of the sorting algorithm will not change. But if it is adaptive then the time complexity will change as per input. In this case we calculate the 3 different cases for time complexity: worst case (case in which the time complexity is maximum), best case (case in which the time complexity is minimum) and average case (average of worst and best case).

Number of data moves: Data move is nothing but operation involving copying/ swapping the data records. One data move is equal to 1 copy operation of complete data records. Note that data moves are not updates of variables independent of record size (eg. loop variable).

Space complexity: Space complexity depends on the extra memory used. For calculating the space complexity do not count the space required for holding the input data. Count only the extra space that is needed.

1. $\Theta(1)$: The space complexity is $\Theta(1)$ if the extra memory required is constant (in place methods).
2. $\Theta(N)$: The space complexity is $\Theta(N)$ if the extra space is proportional to the number of items. The space complexity is $\Theta(N)$ for pointers (eg. linked lists or indirect access), for a copy of data.

There are several sorting algorithms with different time complexities and space complexities and the applications in which they are suited depends on the time and space complexity of the algorithm. One such algorithm is insertion sort.

Insertion sort: In insertion sort, the input array is processed/ sorted from left to right. At every step it sorts one item more compared to the previous step.

Each row shows the array after one iteration of the outer loop (after step i).

original	5	3	7	8	5	0	4
1st	3	5	7	8	5	0	4
2nd	3	5	7	8	5	0	4
3rd	3	5	7	8	5	0	4
4th	3	5	5	7	8	0	4
5th	0	3	5	5	7	8	4
6th	0	3	4	5	5	7	8

We start with the second element in the input array and check whether it is smaller or greater than the element in the left. If the second element (in this case = 3) is smaller than the first element then we send it to the left (i.e. swap the elements). Now these two elements on the left are sorted as part of the input array.

In the next iteration, we check where the third element in the input array can be placed in this sorted left part of the array (gray shaded portion). This is repeated in every iteration. In each iteration we check and insert the new element from the right unsorted part of the input array to the left sorted part of the input array. By the time we reach the last element, the whole input array is sorted.

Pseudo code for insertion sort:
Assumption: A is the list of numbers to be sorted and n is the size of the array.

```
Insertion_sort(A, n):
        for i = 1 to n
                key = A[i]
                k = i - 1
                while k >= 0 and A[k] > key:
                        A[k+1] = A[k]
                        k = k - 1
                A[k+1] = key
```

There are different methods and algorithms to find the time complexity of an algorithm but we will be using **proof by loop invariant** method here. The proof by loop invariant includes 3 steps:

1. Initialization
2. Maintenance
3. Termination

We first need to find the loop invariant here. If we see the image in the insertion sort algorithm, we can see that the part of the array to the right of the key is always sorted. This is nothing but the loop invariant here.

While calculating the time complexity, we generally calculate the time complexity for the worst scenario. Here the worst case scenario is when the list of numbers is in the reverse order. Now assuming that the list of numbers is in reverse order we can say that,

(Initialization)

In first iteration, there is only one data move required - 1

In second iteration, there are 2 data moves required - 2

(Maintenance)

This is goes on till the last element

(Termination)

In nth iteration there are n data moves required - n

Thus in total, the number of data moves required are - (1 + 2 + 3 + ….. + n) = $\frac{n(n+1)}{2}$

Therefore the time complexity $T(n) = \frac{n(n+1)}{2} = O(n^2)$ ….(using Asymptotic analysis notation discussed at the end).

Insertion sort uses in-place memory i.e. it does not require extra memory therefore the space complexity is $O(n)$.

Note that here runtime depends on input order and input size.

Time complexity analysis:

As mentioned above the runtime depends on the input order. Depending on this there are three possible scenarios and thus 3 different time complexity analysis possible. They are best case, worst case and average case.

1. **Best case:** If the items in the array are already sorted then the while loop in the code will never be executed and thus the number of iterations will be N. Thus resulting in the best case TC to be $\Theta(n)$.
2. **Worst case:** If the items in the array are exactly in reverse order then the while loop will be executed every time to its maximum (N). Thus resulting in the worst case TC to be $\Theta(n^2)$.
3. **Average case:** It is the average of best and worst cases or in other words, it is the scenario when half of the elements are sorted and other half aren't. Thus the TC for average case is $(\Theta(n) + \Theta(n^2))/2 = \Theta((n + n^2)/2) = \Theta(n^2)$.

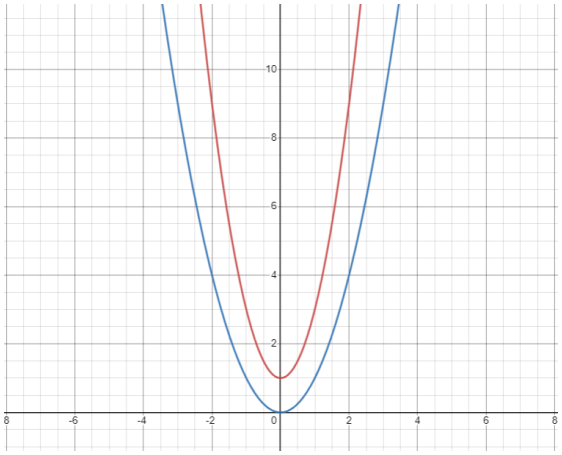
Advantages of insertion sort:

1. Simple implementation

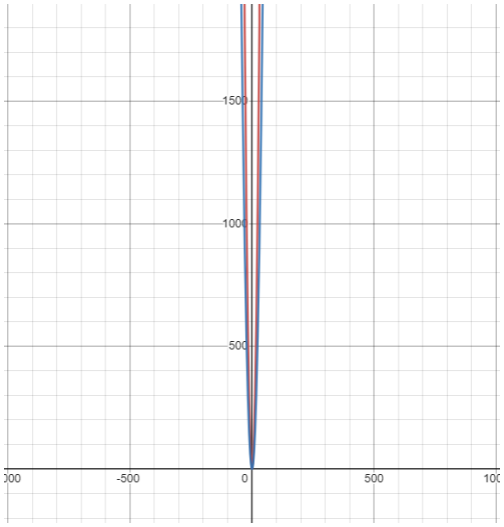
2. Adaptive in nature
3. Uses in-place memory and thus space complexity is linear.

Asymptotic analysis:

In all the analysis we do, we generally represent the TC and SC using asymptotic notations only as they preserve the nature in which the function grows. For example: $O(2n^2 + 14)$ follows the similar trend as $O(n^2)$.



(a) For small values of n



(b) For large values of n

Here we can see that for small values of n there is significant difference between both the graph functions but for large values of n there is not much difference between both the functions.

In asymptotic analysis notation we do the following 2 things:

1. Drop lower order terms
2. Ignore leading constants

For example: $O(\frac{n(n+1)}{2}) = O(n(n+1)) = O(n^2 + n) = O(n^2)$

Merge sort,
Recurrence relations (RRs) and
Methods to solve RR.

Merge sort:
Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sub list consists of a single element and merging those sub lists in a manner that results into a sorted list

Approach:

1. If n = 1 then done. ($O(1)$)
2. If n > 1 then recursively sort i.e. divide the unsorted list into 2 sub lists each time and recursively sort each sub list till the base case is reached. ($2T(n/2)$)
3. Once each sublist is sorted then merge the two sub lists such that the resulting merged list is sorted. ($O(n)$)

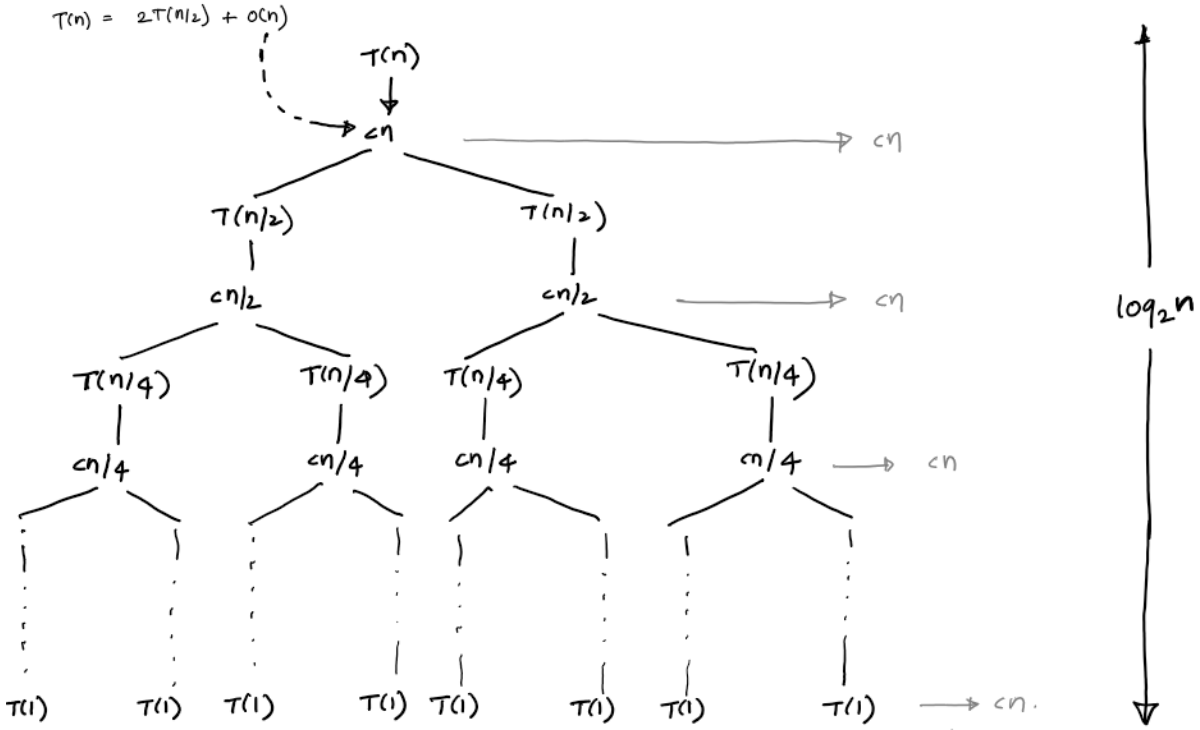
Running time $T(n) = 2T(n/2) + O(n)$
This recursive relation can be solved using recursion tree method (shown after Pseudocode)

Pseudocode for merge sort:

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

Recursion tree for $T(n) = 2T(n/2) + O(n)$:



Now at each level we can see that the sum is cn and the height of the recursion tree is $log_2 n$. Therefore we can say that the sum of complexity from each level is,

$$T(n) = 2T(n/2) + O(n) = \sum_{i=0}^{log_2 n} cn = cnlog_2 n = O(n.log_2 n)$$

$$\therefore T(n) = O(n.log_2 n)$$

Big-Oh notation "O":

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$

Big-Omega notation "Ω":

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}.$$

Little-oh notation "o":

$$o(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le f(n) < cg(n) \text{ for all } n \ge n_0\}.$$

Little-omega notation "ω":

$$\omega(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\} .$$

Solving recurrences:

There are 3 different methods that we can use for solving the recurrences. They are:

1. Substitution method: In the substitution method, we guess a bound and then use mathematical induction to prove our guess correct.
2. Recursion tree method: The recursion-tree method converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence.
3. Master method: The master method provides bounds for recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

Substitution method: (Guess and verify)

Let's solve the recurrence relation $T(n) = 4T(n/2) + n$ using the substitution method. For the substitution method we first need to guess the possible solution and then we can check if it is the correct solution or not.

Let's guess that the solution is $T(n) = O(n^3)$.

Guess: $T(n) = O(n^3) \rightsquigarrow \therefore T(n) \leq cn^3$

Assume: $T(m) = O(m^3) \leq cm^3 \quad (\forall m < n)$

Prove: $T(n) \leq cn^3$ given $T(m) \leq cm^3 \ (\forall m < n)$

As we know, $n/2 < n \longrightarrow$ put m as $n/2$

$\therefore T(n/2) = c(n/2)^3 = cn^3/8$

Now, $T(n) = 4T(n/2) + n$

$\leq \frac{4cn^3}{8} + n$

$\leq \frac{cn^3}{2} + n$

$\therefore T(n) \leq cn^3 \quad \left(\because \left[\left(cn^3/2\right) + n\right] < cn^3\right)$

$\therefore \boxed{T(n) = O(n^3)}$

Now let's check if $T(n) = O(n^2)$ is the correct solution or not.

Guess: $T(n) = O(n^2) \leq cn^2$ $\therefore$ Prove if $T(n) \leq cn^2$.

Assume: $T(m) = O(m^2) \leq cm^2$ $(\forall\, m < n)$

Put $m = n/2$ $(\because n/2 < n)$

$\therefore T(n/2) \leq c(n/2)^2$

$\leq cn^2/4$

$\therefore$ Prove if $T(n) \leq cn^2$ assuming $T(n/2) \leq cn^2/4$.

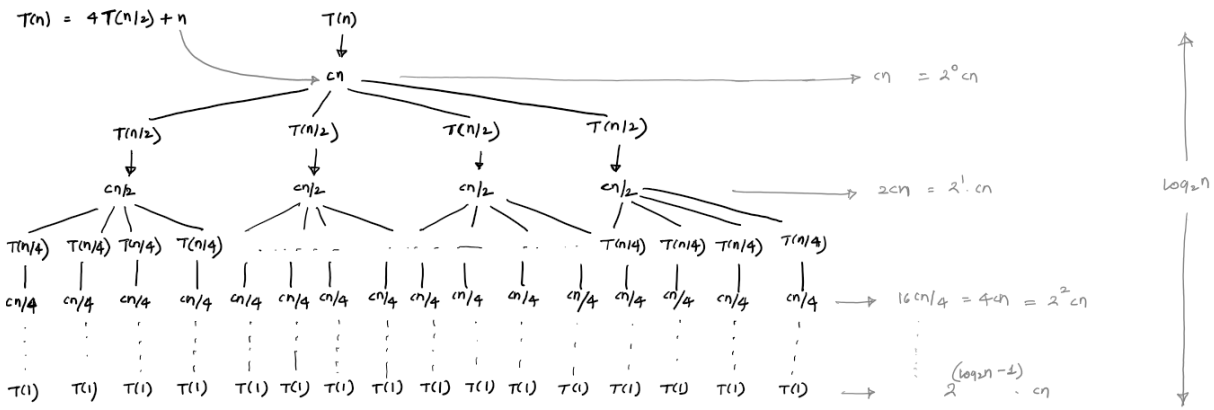
$\therefore T(n) = 4T(n/2) + n$

$\leq 4cn^2/4 + n$

$T(n) \leq cn^2 + n$ → and from this we can see that $\boxed{T(n) > cn^2}$

$\therefore$ The guess $T(n) = O(n^2)$ is WRONG!

Recursion tree:



$$\therefore T(n) = \sum_{i=0}^{\log_2 n - 1} 2^i \cdot cn = cn \sum_{i=0}^{\log_2 n - 1} 2^i = cn \frac{2^{\log_2 n - 1} - 1}{2 - 1} = cn \cdot [2^{\log_2 n - 1} - 1]$$

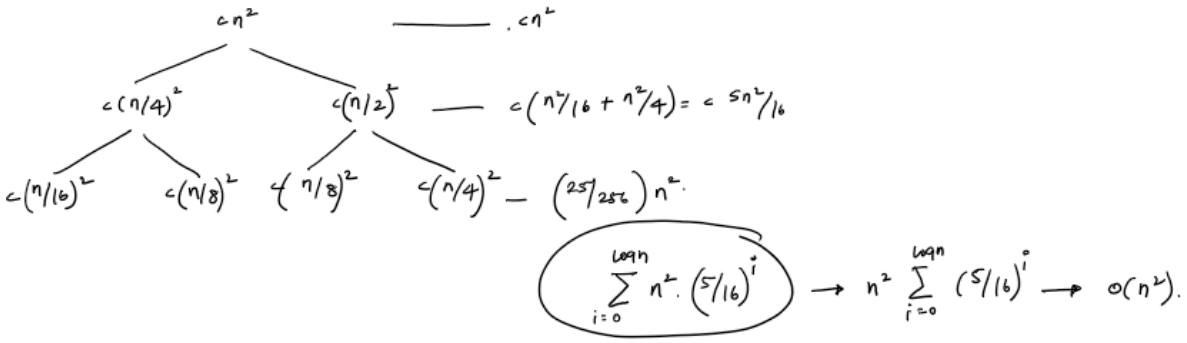
$$= cn \left[2^{\log_2 n - \log_2 2} - 1 \right]$$

$$= cn \left[2^{\log_2 n/2} - 1 \right]$$

$$= cn \left[n/2 - 1 \right] = cn(n-2)/2 = O(n)$$??

(Need to check with professor regarding above question)

$$T(n) = T(n/4) + T(n/2) + n^2$$



Master method:

As mentioned earlier the master method can be used to solve recurrence relations of the form $T(n) = aT(n/b) + f(n)$ where $f(n)$ can be any function of n.

Approach:

1. Calculate $n_{critical} = n^{log_b a}$
2. Now if,
   a. $f(n) < n_{critical}$ then the recurrence relation is given as, $T(n) = O(n_{critical})$
   b. $f(n) = n_{critical}$ then the recurrence relation is given as, $T(n) = O(n_{critical}. log_b(n))$
   c. $f(n) > n_{critical}$ then the recurrence relation is given as, $T(n) = O(f(n))$

Example: $T(n) = 4T(n/2)$

Here $a = 4,\ b = 2\ and\ thus\ n_{critical} = n^{log_2 4} = n^2$.

As $f(n) < n_{critical}$, we can say that $T(n) = O(n_{critical}) = O(n^2)$.

Binary search,
Exponential/ powering a number,
Fibonacci and
Matrix multiplication.

Binary search:

Binary search is the most famous search algorithm. It divides the search array/ list into half in every iteration by comparing the center element with the search element. For doing so, it requires the search array to be sorted.

Pseudocode:

```
Binary-search(A, low, high, x)
        if low > high then return -1
        mid = low + (high - low)/2
        if x == A[mid] then return mid
        if x < A[mid] then Binary-search(A, low, mid - 1, x)
        else return Binary-search(A, mid + 1, x)
```

Running time analysis:
We know that the binary search divides the given list of n elements into half in every new iteration. Thus, we can say that the recurrence relation for running time is,

$$T(n) = T(n/2) + O(1)$$

Using Master theorem,

$n_{critical} = n^{log_b a} = n^{log_2 1} = n^0 = 1$

$f(n) = n_{critical}$ therefore we can say that $T(n)$ is,

$$T(n) = n_{critical} \times log_2 n$$
$$T(n) = log_2 n$$

Exponential/ powering a number:

To calculate the exponent, in general, we need $O(n)$ operations to calculate a^n.

But this can be solved in $O(log(n))$ as well. We know that the power can be split into 2 parts.

$$\begin{aligned} a^n &= a^{n/2} . a^{n/2} && \dots if\ n\ is\ even \\ &= a^{(n-1)/2} . a^{(n-1)/2} . a && \dots if\ n\ is\ odd \\ &= 1 && \dots if\ n\ is\ 0 \end{aligned}$$

This above calculation will require only $O(log(n))$.

Fibonacci sequence:

The fibonacci sequence function is given as,

$$\begin{aligned} fib(n) &= 0 && \dots if\ n = 0 \\ &= 1 && \dots if\ n = 1 \\ &= fib(n-1) + fib(n-2) && \dots if\ n > 1 \end{aligned}$$

Approach 1: Recursive function.

```
fib(n):
        if n <= 1 return n
        else return fib(n-1)+fib(n-2)
```

Running time:

$T(n) = T(n-1) + T(n-2) = O(\Phi^n)\ where\ \Phi = \frac{1+\sqrt{5}}{2}$.

Approach 2: Iterative function.

```
fib(n):
        if n <= 1 then return n
        curr = 0
        prev2 = 0
        prev1 = 1
        for i = 2, …., n
                curr = prev1 + prev2
                prev2 = prev1
                prev1 = curr
        return curr
```

Running time:

$T(n) = \theta(n)$

Approach 3: Using Binet's formula

$fib(n) = \Phi^n/\sqrt{5}\ where\ \Phi = \frac{1+\sqrt{5}}{2}$.

Running time:

$T(n) = O(1) \times O(log(n))$

$(O(1)\ for\ calculation\ of\ \Phi\ and\ O(log(n))\ for\ calculation\ of\ exponent)$

$\therefore T(n) = O(log(n))$

Issues with approach 3 is that, there are rounding errors for large values of n.

Approach 4: Matrix exponential

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} fib(n+1) & fib(n) \\ fib(n) & fib(n+1) \end{bmatrix}$$

Running time: $T(n) = O(log(n))$.

Matrix multiplication:
In general, matrix multiplication can be defined as follows,

If A and B are two matrices of size $n \times n$ then the multiplication of A and B can be given as,

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \times B_{k,j}$$

where C is the resultant matrix.

Following approaches are discussed assuming that both the matrix A and B are of size (n x n).

Approach 1: Naive approach for matrix multiplication:

```
for i = 0 to n ~~> O(n^3)
    for j = 0 to n ~~> O(n^2)
        c[i][j] = 0
        for k = 0 to n ~~> O(n)
            c[i][j] = c[i][j] + (a[i][j] * b[i][j])
```

Here we can see that the running time required for naive approach is $O(n^3)$.

Approach 2: Divide and conquer:

1. Divide each matrix into 4 parts.

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

2. Calculate the matrix multiplication using the following formula.

$$r = (a \times e) + (b \times g)$$
$$s = (a \times f) + (b \times h)$$
$$t = (c \times e) + (d \times g)$$
$$u = (c \times f) + (d \times h)$$

3. Solve each subproblem recursively till the problem reduces to base case i.e. to a size of 2 x 2 matrix multiplication.

Running time analysis: Here we can see that the problem size reduces by have in each iteration i.e. for matrix sizes of 4 x 4, the problem subdivides the problem into matrix multiplication of 2 x 2. And each of these multiplications needs to be done 8 times recursively. In every recursive call we even add these $n^2/4$ matrices. Thus the recurrence relation of running time can be given as,

$$T(n) = 8T(n/2) + O(n^2)$$

Using master theorem, $n^{log_2 8} = n^3 > n^2$. Thus we can say that the running time $T(n) = O(n^3)$.

But this method has no time complexity improvement over the naive approach. Following is the method which running time better than the above methods. This approach is called Strassen's matrix multiplication.

Approach 3: Strassen's matrix multiplication.
This also uses the divide and conquer approach as well. This divides the given matrices into 4 equal parts as follows:

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

Each of the elements of the result matrix can be calculated as follows,

$p1 = a(f - h)$
$p2 = (a + b)h$
$p3 = (c + d)e$
$p4 = d(g - e)$
$p5 = (a + d)(e + h)$
$p6 = (b - d)(g + h)$
$p7 = (a - c)(e + f)$

$r = p5 + p4 - p2 + p6$
$s = p1 + p2$
$t = p3 + p4$
$u = p5 + p1 - p3 - p7$

And these multiplications are performed recursively. Note that here there are only 7 multiplications that need to be performed compared to 8 multiplications required in the previous approach. Thus the running time recurrence relation can be given as,

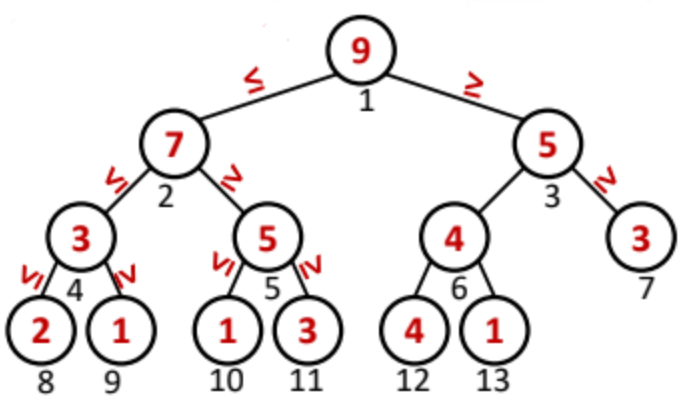
$$T(n) = 7T(n/2) + O(n^2)$$

Using master theorem, $n^{\log_2 7} = n^{2.81} > n^2$. And thus the running time can be given as $T(n) = O(n^{2.81}) = O(n^{\log_2 7})$.

Heaps,
Heapsort and
Priority queues using heaps.

Heaps: Heap is a data structure to store elements in the form of a tree where parent and child nodes are arranged in a certain manner depending on the type of the heap. There are two types of heaps i.e. max heap and min heap. A binary heap is a heap in which every node has at most 2 child nodes.
In the binary max heap, the left child and the right child are greater than or equal to the parent node.

value	9	7	5	3	5	4	3	2	1	1	3	4	1
index	1	2	3	4	5	6	7	8	9	10	11	12	13

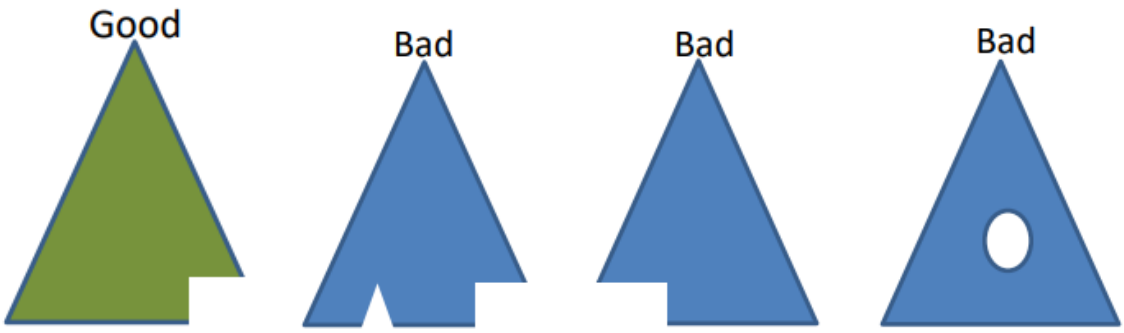


The heap is generally viewed as a tree but stored in the form of an array.

	Index calculation: parent <->node<->child			
Root at index	Node	Left Child	Right child	Parent
1 (used here)	**i**	**2i**	**2i+1**	$\lfloor i/2 \rfloor$
0	i	2i+1	2i+2	$\lfloor (i-1)/2 \rfloor$

Properties of heap:

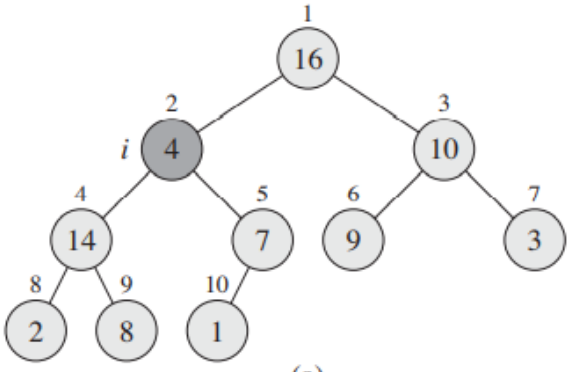
1. Order: The priority of each node is smaller than or equal to that of his parent.
2. Shape: The tree should be complete. There can not be any holes in the tree. All the levels should be complete with an exception for the last level. The last level can or cannot be complete. If it is incomplete then all nodes should be to the left.



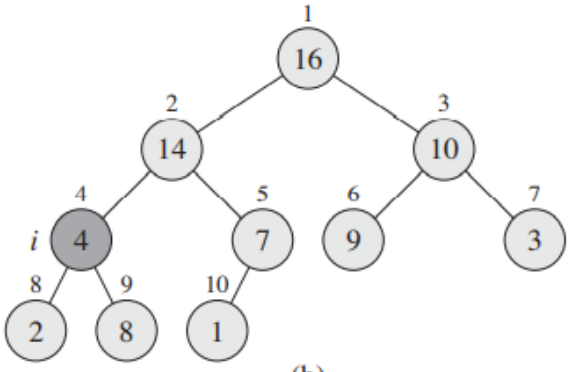
Maintaining the heap property:
The max-heap property that the value/ priority of each child node is less than or equal to it node can be maintained using the following pseudocode Max-Heapify.

```
MAX-HEAPIFY(A, i)
1   l = LEFT(i)
2   r = RIGHT(i)
3   if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5   else largest = i
6   if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8   if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

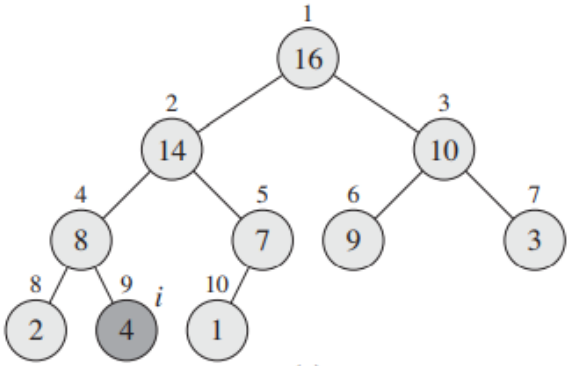
The LEFT(i) and RIGHT(i) basically return the index of the left child node and right child node of i. Then the values of left child, right child and the parent node itself are compared to find the largest. If the largest node is the parent node itself then the code exits but if the largest node is left or right child then parent value and the child value are swapped and a recursive call is made for the child node whose value was just swapped with parent node.
The running time of Max-Heapify(A,i) is equal to the height of the tree. If the tree is complete as mentioned above then the running time is $O(log(n))$.



(a)



(b)



(c)

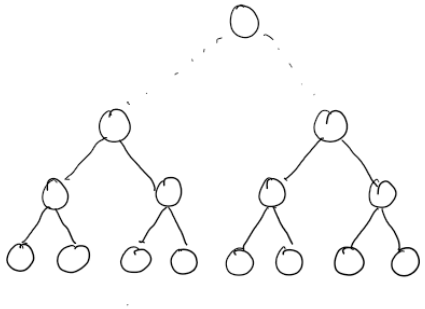
Building a max heap:
A max heap can be built by just calling the Max-Heapify(A,i) for all nodes in the heap from second last level to top. The pseudo code can be given as follows:

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

The for loop calls Max-Heapify(A,i) n/2 times. We know that the running time of Max-Heapify(A,i) is $O(log(n))$.
The running time of the Build-Max-Heap(A) can be given as,

$$T(n) = O((n/2)log(n)) = O(nlog(n))$$

But there is a tighter bound on the running time of build max heap function.



Level root (log n) → # of nodes = $n/2^{level+1} = n/2^{\log_2(n)}$ — # of swaps: log(n)

Level 2 → $= n/2^2 = n/2^{level+1}$ — 2

Level 1 → # of nodes $= n/2^2 = n/2^{level+1}$ — 1 = level

Level 0 → # of nodes $\leq n/2 = n/2^{level+1}$ — 0 = level

Thus the running time can be given as,

$$T(n) = \sum_{i=0}^{\log n} \frac{n}{2^{i+1}} \cdot i = \frac{n}{2} \sum_{i=0}^{\log n} i/2^i \leq \frac{n}{2} \sum_{i=0}^{\infty} i/2^i$$

The summation term can be solved as follows,

$$\sum_{i=0}^{\infty} i/2^i \rightsquigarrow S = 1/2 + 2/4 + 3/8 + 4/16 + \cdots$$

$$\therefore S/2 = 1/4 + 2/8 + 3/16 + 4/32 + \cdots$$

$$\therefore S - S/2 = S/2 = \underbrace{1/2 + 1/4 + 1/8 + 1/16 + \cdots}$$

geometric series
$a = 1/2$ and $r = 1/2$

$$\therefore S/2 = \frac{a}{1-r} = \frac{1/2}{1-1/2} = 1.$$

$$\therefore \boxed{S = 2}$$

Substituting the value of summation back in the equation of running time, we get,

$$\therefore T(n) \leq \frac{n}{2} \cdot 2$$

$$\therefore T(n) \leq n$$

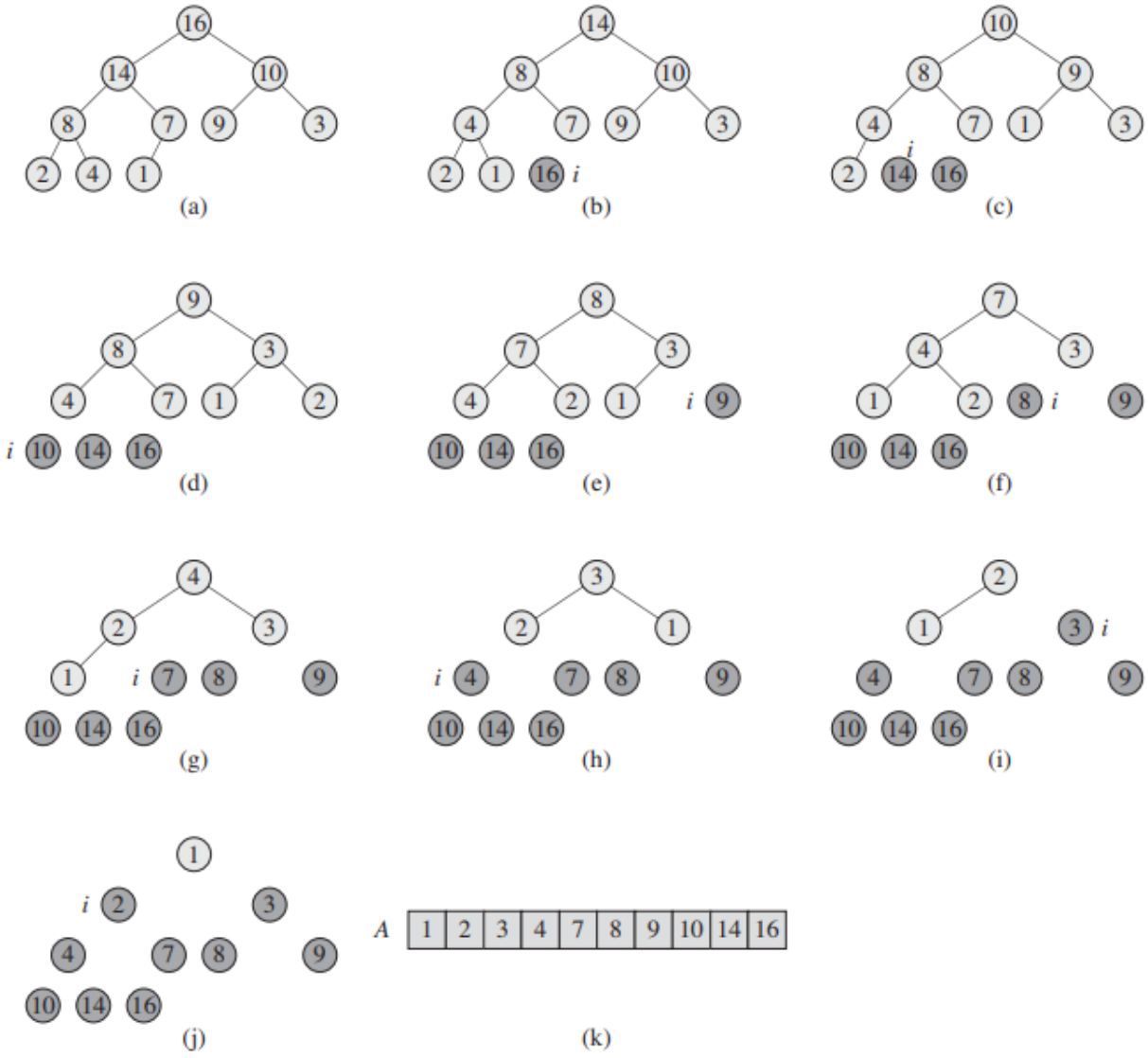
$$\therefore \boxed{T(n) = O(n)}$$

Thus we can say that the running time of build max heap is $O(n)$.

Heap sort:

Sorting using heaps can be easily performed by just swapping the root node with the last node, then reducing the size of the heap by 1 and then max-heapify the heap. Repeat this all the nodes are arranged in descending order.

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```



Priority queues - an application of heaps data structure:
A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key. A max-priority queue supports the following operations:

INSERT(S, x) inserts the element x into the set S, which is equivalent to the operation $S = S \cup \{x\}$.

MAXIMUM(S) returns the element of S with the largest key.

EXTRACT-MAX(S) removes and returns the element of S with the largest key.

INCREASE-KEY(S, x, k) increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

Alternatively, a min-priority queue supports the operations INSERT, MINIMUM, EXTRACT-MIN, and DECREASE-KEY.

Maximum operation can be performed easily using max-heap by just reading the root node value.

```
HEAP-MAXIMUM(A)
1  return A[1]
```

Extract maximum operation can be done by simply swapping the root node with the last node value in the array and then call Max-Heapify function on the root node.

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2       error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

The Increase key operation can be performed by just increasing the value of the node and then checking if the new key value is smaller than its parent. If it is smaller than parent then do nothing but if it is greater than its parent then swap them and check the same for that node till it reaches the root node.

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2       error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5       exchange A[i] with A[PARENT(i)]
6       i = PARENT(i)
```

The insert operation can be easily performed using the increase key function as follows.

```
MAX-HEAP-INSERT(A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = −∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)
```

Quicksort and Randomized Quicksort.

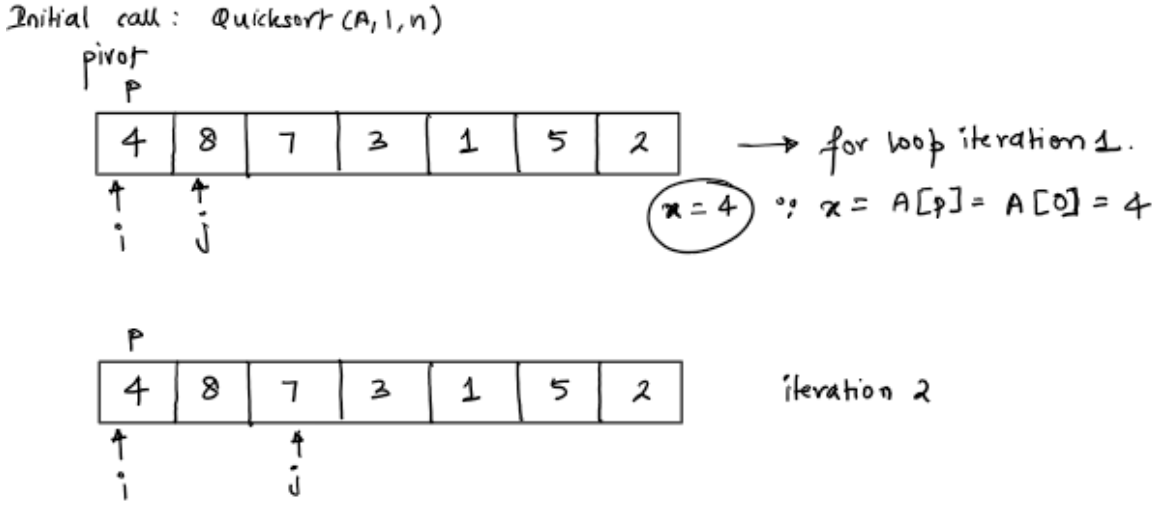
Quick sort: Quick sort is a divide and conquer sorting algorithm similar to merge sort. Quick sort involves the use of two main functions: Partition and Quick-Sort. Let's discuss then one by one.

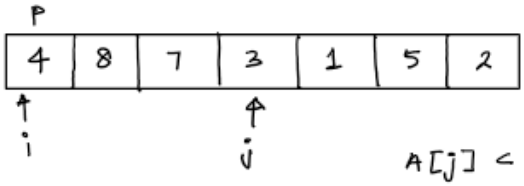
Partition function:

```
PARTITION (A, p, q) :
    x ← A[p]    // pivot point value
    i ← p       // set pointer i same as pivot
    for j ← (p+1) to q
        if A[j] ≤ x
            i = i + 1
            swap A[i] and A[j]
    swap A[p] and A[i]
    return i
```

This partition function basically divides the array A[p, ….., q] into two parts and returns the index i such that all values in A[p, p+1, …., i] are smaller than all values in A[i+1, i+2, …., q].
Also a point to note is that the value at i^{th} index in the array A after the partition function is executed will be in its correct position as compared to the final sorted array.

Example of partition function working:

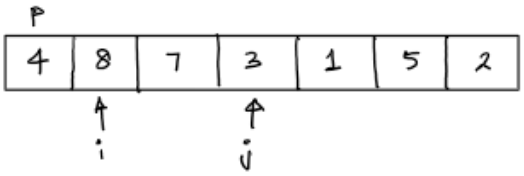


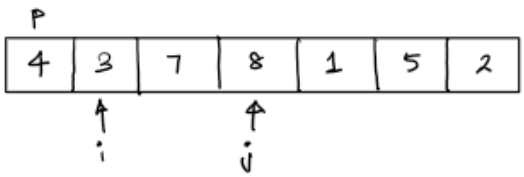


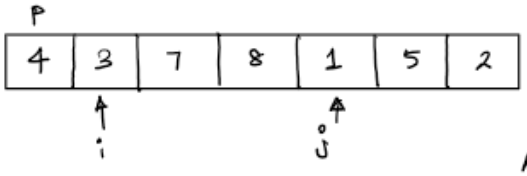
iteration 3

$A[j] < A[p]$ OR x.

$\therefore$ increment i & swap $A[i]$ & $A[j]$

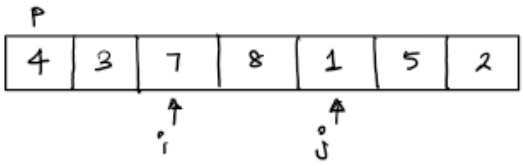


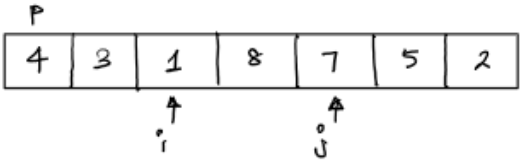


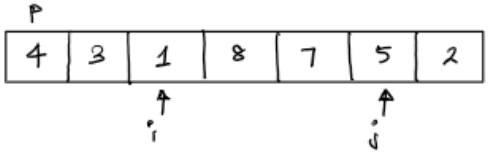


iteration 4

$A[j] < A[p]$ $\therefore$ inc i & swap $A[i]$ & $A[j]$

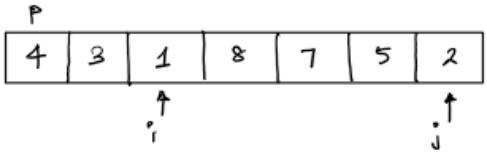






iteration 5

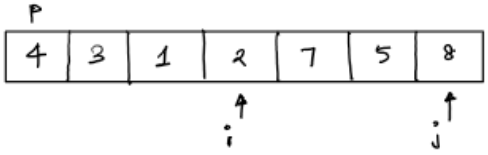
$A[j] > A[p]$ → $\therefore$ go to next iter^{n}

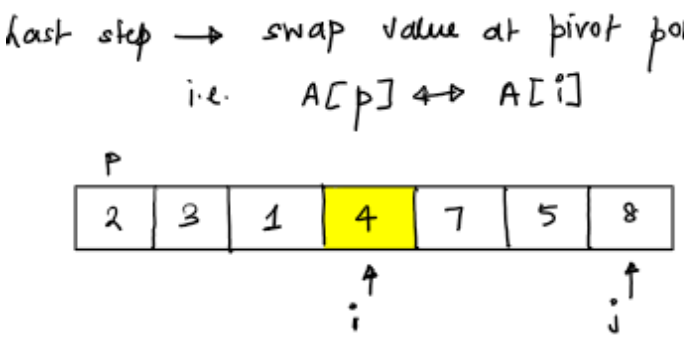


iteration 6.

$A[j] < A[p]$ $\therefore$ inc i & swap $A[i]$ & $A[j]$

$\therefore$ Resultant array :





Quicksort function:
The quick sort function uses the partition function and recursively calls itself to sort the given array. By now we understand that on calling partition function on an array A, the element $A[\lfloor n/2 \rfloor]$ is placed in the correct position.
Therefore we recursively call the quick sort function on the arrays to the left and right of this position. Let's see the quick sort function now.

```
QUICKSORT (A, p, r)
if p < r
    q ← PARTITION (A, p, r)
    QUICKSORT (A, p, q-1)
    QUICKSORT (A, q+1, p)
```

Running time analysis:
Worst case: The quick sort algorithm divides the problem into subproblems assuming the first point as pivot. But if this point is the smallest element in the array then it will result in two partitions where one array has n-1 elements and other has 0 elements.
Thus the recurrence relation can be given as,
$T(n) = T(n-1) + T(0) + O(n) = T(n-1) + O(n)$

And if the array is sorted (in increasing or reverse order) then this would be the case in every recursive call as well.
Thus the recursive calls go from n, $(n-1)$, $(n-2)$, …., 1 and the summation of this results in $\frac{n(n+1)}{2}$. Thus the running time in worst case can be given as, $T(n) = O(\frac{n(n+1)}{2}) = O(n^2)$.

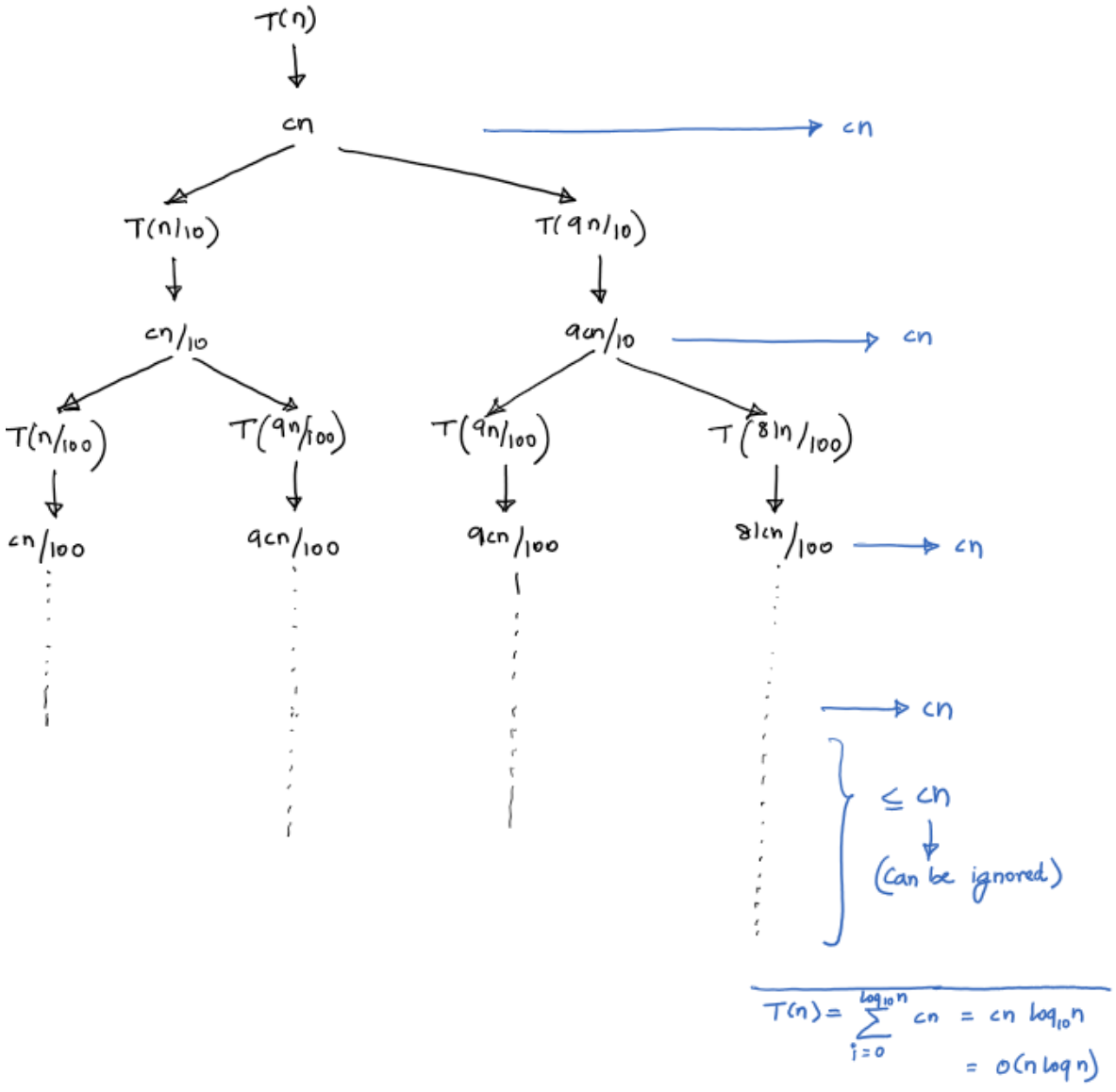
We can see that the performance of quick sort is worse than insertion sort in the worst case i.e. if the array is already sorted then insertion sort requires $O(n)$ whereas quick sort requires $O(n^2)$.

Best case: The best case is when the quick sort algorithm divides the problem into almost equal sub problems (like in the example of partition function where 4 was equal to the median of

the given array list). In such a case, the quick sort algorithm divides the array A into arrays of size $\lfloor n/2 \rfloor$ and $\lfloor n/2 \rfloor - 1$. The recurrence relation in the best case scenario can be given as,
$T(n) = 2T(n/2) + O(n)$
This can be easily solved using the master theorem.
$n^{log_2 2} = n$ which is equal to n. Thus the running time can be given $T(n) = O(nlog(n))$.

Case slightly better than worst case scenario i.e. 1:9 split:
In this scenario, the recurrence relation can be given as,
$T(n) = T(n/10) + T(9n/10) + O(n)$

Lets see the recursion tree for this recurrence relation.



We can see that the sum at each level of recursion tree is cn. This will be true for level of $log_{10}n$ as the height of the stem resulting from leftmost branch will be $log_{10}n$ and that of the rightmost will be $log_{10/9}n$. Thus the running time can be given as,

$$T(n) = \sum_{i=0}^{log_{10}n} cn = cn \sum_{i=0}^{log_{10}n} 1 = cn.log(n) = O(n.log(n)).$$

Thus we understand here that even for very unbalanced partitions the running time is $O(n.log(n))$. Let's understand this better using the average case.

Average case or the lucky-unlucky case:

As here the size of the partitions in recursive calls decide whether the scenario is best or worst we need to consider the average case scenario as the scenario where these cases are alternating. Let lucky case $L(n)$ be the case where the resulting partition is equal and unlucky case $U(n)$ be the case where the one partition contains $(n - 1)$ elements and the other contains 0.

$L(n) = 2U(n/2) + O(n)$

$U(n) = L(n - 1) + O(n)$

We can substitute the equation for $U(n)$ in $L(n)$.

$L(n) = 2(L(n/2 - 1) + O(n/2)) + O(n) = 2L(n/2 - 1) + O(n) \approx 2L(n/2) + O(n)$

Thus,

$L(n) \approx 2L(n/2) + O(n)$

And because of this, we can say that the running time of average case is $T(n) = O(n.log(n))$ which is the same as the best case scenario.

Randomized version of quick sort:

This version of quick sort is similar to the general version of quicksort. The only difference is that the pivot point that we select in the Partition function is randomly selected and then the same operations are performed.

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2      q = RANDOMIZED-PARTITION(A, p, r)
3      RANDOMIZED-QUICKSORT(A, p, q − 1)
4      RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Here the Random(p,r) function returns a random number between p and r (inclusive of p and r).

How can we improve randomized quicksort?

One way to improve the randomized quicksort procedure is to partition around a pivot that is chosen more carefully than by picking a random element from the subarray. One common approach is the **median-of-3 method**: choose the pivot as the median (middle element) of a set of 3 elements randomly selected from the subarray.

Order statistics.

Ith order statistic:
The ith order statistic of a set of n elements is the ith smallest element. For example, the minimum of a set of elements is the first order statistic (i = 1), and the maximum is the nth order statistic (i = n). A median, informally, is the “halfway point” of the set.

Minimum (0th order) and maximum (nth order):

```
MINIMUM(A)
1  min = A[1]
2  for i = 2 to A.length
3      if min > A[i]
4          min = A[i]
5  return min
```

This will require running time of $O(n-1)$ i.e. $O(n)$. And similarly the maximum can be found in $O(n)$ running time.
To find both the minimum and maximum of the array the running time required is $O(2n-2) = O(n)$ operations using a naive approach. But the number of comparisons can be reduced to $O(\lfloor 3n/2 \rfloor)$ or $O(n)$. Note that the running time is $O(n)$ in both the cases but the number of comparisons has been reduced.
This can be achieved by comparing numbers in pairs. In the naive approach, the number of comparisons required for finding min and max would be 4 i.e. 2 comparisons for min and 2 comparisons for max. This can be reduced to 3 comparisons,

1. Compare the number pair with each other.
2. Then compare the smaller one with min.
3. And then compare the larger one with the max.

Selection in expected linear time:

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2      return A[p]
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q - p + 1
5  if i == k            // the pivot value is the answer
6      return A[q]
7  elseif i < k
8      return RANDOMIZED-SELECT(A, p, q - 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i - k)
```

To get the ith order statistic, we can use the above function. The Random-Select function uses the Randomized-Partition function introduced in the quick sort discussion. The Randomized-Partition function will divide the given array sequence into 2 parts. The elements at index lower than q will have value lower than the value of element at the qth index. And similarly, the elements at index greater than q will have value greater than the value of element at the qth index. We can calculate the order statistic of the value at qth index by simple calculation of $k = q - p + 1$. Now if this k (order statistic of value at qth index) is equal to i then we found the

i^{th} order statistic. But if k is greater than i then we need to make a recursive call for array on the left and k is greater than i then we need to make a recursive call for array on the right.

Running time: We expect that the position of the random element in the sorted sequence will be uniformly distributed between 0 and N-1, where N is the length of the data set. This means there's at least a 1/2 chance that it's between N/4 and 3N/4. That means that at least half the time, we get a pivot that reduces the problem to no more than 3/4 of the previous problem size.

Each pivot selection and filtering of the elements by the pivot takes O(N) if there are N elements remaining. Based on the above, we expect the problem to be reduced to no more than 3/4 of its size after two of these O(N) passes. We therefore get a recurrence relationship of $T(n) = T(3n/4) + 2 * O(n) = T(3n/4) + O(n)$.
(Reference: Post on quora: https://www.quora.com/How-is-time-taken-by-a-randomized-quick-select-algorithm-in-O-n)

Thus using master theorem, $n^{log_{4/3}1} = n^0 = 1 < n$. Therefore the running time is $T(n) = O(n)$.

Selection in worst case linear time:

The SELECT algorithm determines the ith smallest of an input array of $n > 1$ distinct elements by executing the following steps. (If $n = 1$, then SELECT merely returns its only input value as the ith smallest.)

1. Divide the n elements of the input array into $\lfloor n/5 \rfloor$ groups of 5 elements each and at most one group made up of the remaining n mod 5 elements.
2. Find the median of each of the $\lceil n/5 \rceil$ groups by first insertion-sorting the elements of each group (of which there are at most 5) and then picking the median from the sorted list of group elements.
3. Use SELECT recursively to find the median x of the $\lceil n/5 \rceil$ medians found in step 2. (If there are an even number of medians, then by our convention, x is the lower median.)
4. Partition the input array around the median-of-medians x using the modified version of PARTITION. Let k be one more than the number of elements on the low side of the partition, so that x is the kth smallest element and there are $n-k$ elements on the high side of the partition.
5. If $i = k$, then return x. Otherwise, use SELECT recursively to find the ith smallest element on the low side if $i < k$, or the $(i-k)$th smallest element on the high side if $i > k$.

Stacks,
Queues,
Linked lists and
Hash tables.

Stacks: Stack is a data structure in which elements can be inserted and removed. The last inserted element is the first to be removed. It basically follows last-in, first-out policy.
Stack supports two general operations i.e. push and pop. Other than that we can stack-empty operation returns True if the stack is empty else False. They can be implemented as follows,

```
STACK-EMPTY(S)
1  if S.top == 0
2       return TRUE
3  else return FALSE
```

```
PUSH(S, x)
1  S.top = S.top + 1
2  S[S.top] = x
```

```
POP(S)
1  if STACK-EMPTY(S)
2       error "underflow"
3  else S.top = S.top − 1
4       return S[S.top + 1]
```

Each of these 3 stack operations can be performed in constant running time i.e. $O(n)$.

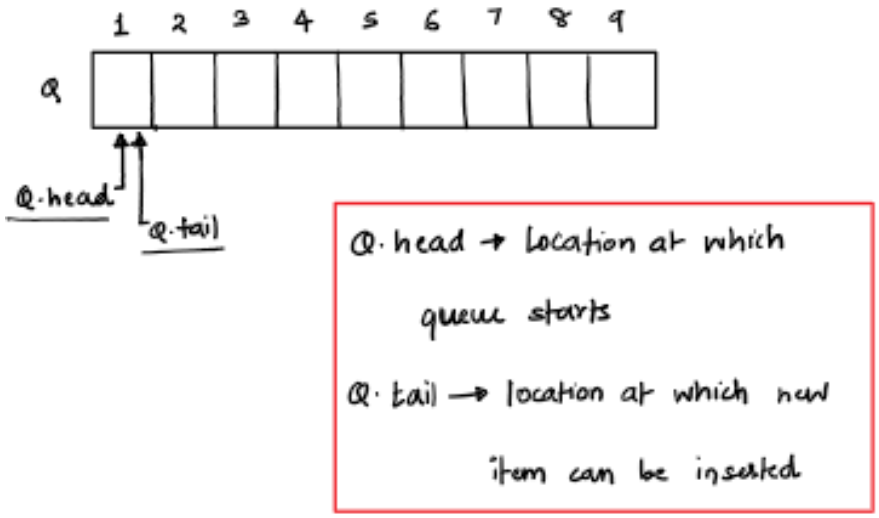
$S.top$ points to the top of the stack.

Queues: Queue is a data structure in which elements can be inserted and removed. The first inserted element is the first to be removed. It basically follows a first-in, first-out policy.
Queue supports two general operations i.e. Enqueue and dequeue. They can be implemented as follows,

```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3       Q.tail = 1
4  else Q.tail = Q.tail + 1
```

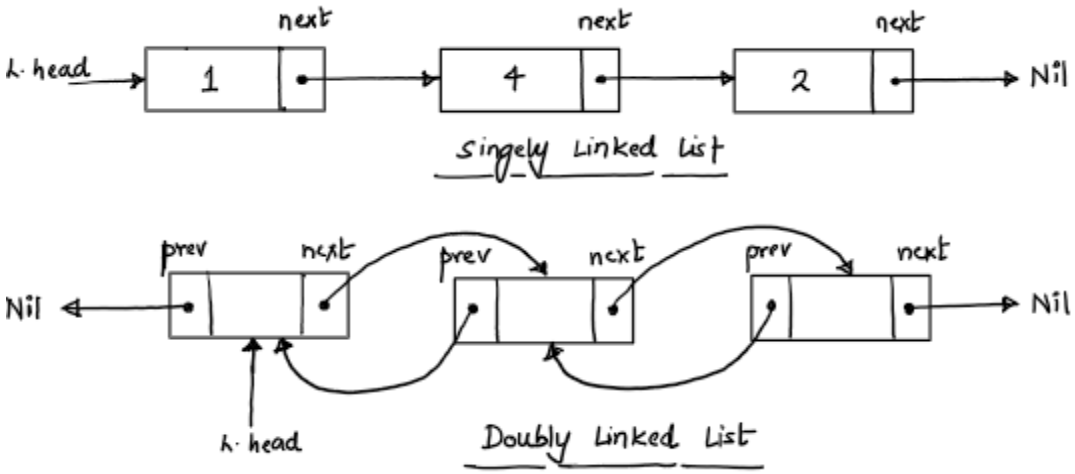
```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3       Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

The above functions allow adding n-1 elements to the queue. The queue can be imagined as follows,



Obviously the enqueue and dequeue functions will cause the $Q.head$ and $Q.tail$ to overlap if the number of elements enqueued are more than n-1. To avoid this, we need to implement some mechanism (not discussed here).

Linked lists: A linked list is a data structure in which the objects are arranged in a linear order. There are different types of linked lists depending upon the pointers that each node in the linked list contains. If every node consists of a pointer to the next node then it is called a singly linked list. And if every node consists of two pointers i.e. one pointing to the next node and one pointing to the previous node then it is called a doubly linked list.



The advantage of using a linked list is that the running time for insertion and deletion of a node is constant i.e. $O(1)$ as it just requires changing the pointers.
But the running time for searching an element in the linked list is $O(n)$ as it needs to iterate through every item in the list for the worst case.

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3      x = x.next
4  return x
```

```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3      L.head.prev = x
4  L.head = x
5  x.prev = NIL
```

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2      x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5      x.next.prev = x.prev
```

Hash tables: Refer these slides by Stefan Alexandra on Hash tables and hashing functions.
https://ranger.uta.edu/~alex/courses/3318/lectures/14_Hashing.pdf
Hash tables have running time complexity of $O(1)$ for searching. This is possible because of the hashing function. Hashing function is a function that generates the index at which the data should be stored depending on the value. For example, suppose we have an array of size 10 and we need to store values 1, 3, 15, 4, 17. A simple hash function can be a mod function. Here the hash function can be just mod 10 (since the range of index is 0 to 9 and mod 10 can generate only values of range 0 to 9).
Thus the value 1 will be stored at index = 1 mod 10 = 1,
Value 3 can be stored at index = 3 mod 10 = 3,
Value 15 can be stored at index = 15 mod 10 = 5 and similarly 4 at index 4 and 17 at index 7.

But this naive hash function is prone to collision.

Insert keys:
46 -> 6
15 -> 5
20 -> 0
37 -> 7
23 -> 3
25 -> 5 collision
35 ->
9 ->

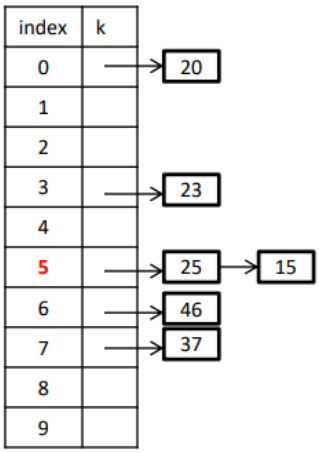
index	k
0	20
1	
2	
3	23
4	
5	15
6	46
7	37
8	
9	

This collision can be resolved using different techniques such as,

1. Separate chaining: Each table entry points to a list of all items whose keys were mapped to that index. Requires extra space for links in lists. Lists will be short.

- Let M = 10, h(k) = k%10

Insert keys:
46 -> 6
15 -> 5
20 -> 0
37 -> 7
23 -> 3
25 -> 5 collision
35
9 ->



2. Probing/ Open addressing
    a. Linear probing: h(k,i,M) = (h_1(k) + i)% M. Issues: Long chains.
    b. Quadratic probing: h(k,i,M) = (h_1(k) + c_1i + c_2i^2)% M. Issues: If two keys hash to the same value, they follow the same set of probes. But better than linear.
    c. Double hashing: h(k,i,M) = (h_1(k) + i* h_2(k)) % M.

Multiplication hash function:
If k is the key value and m is the size of the hash function H then the multiplication hash function H can be given as,

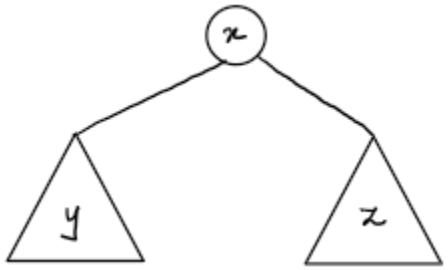
$$H(k) = \lfloor m((kA) mod 1) \rfloor$$

where $0 < A < 1$.
The part $(kA) mod 1$ results in some fraction between 0 and 1. The best results are obtained when value of A is equal to golden ratio.

Binary search trees.

Binary search tree: Binary search tree is a tree with following properties:

a. It has at most 2 child nodes.
b. Every node has a key (sometimes known as value), a parent (Nil for root node), left child and right child (Nil if node belongs to the last level).
c. Has an invariant property: $left\ side\ values \leq x.value \leq right\ side\ values$ where x is any node of a binary tree.



Inorder traversal: In this traversal, the left subtree values are first printed and then the root and then the right subtree values are printed. This is done recursively to traverse through all nodes of the tree.

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

By basic intuition we can say that the running time for this function is $O(n)$ because this would traverse through every node in the tree.

Searching in binary tree: Searching a given value in binary search tree is similar to binary search. We compare the given value with the root first and if it is equal to the root then the search item is found. If it is not equal to root then we check if it is smaller than root.value and compare it with the root of the left subtree via a recursive call. And if it is greater than the root.value then we compare it with the root of the right subtree via a recursive call.

```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2      return x
3  if k < x.key
4      return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

Iterative version:

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ x.key
2      if k < x.key
3          x = x.left
4      else x = x.right
5  return x
```

The running time of both the above search functions is $O(h)$ where h is the height of the binary search tree.

Minimum and maximum: $T(n) = O(h)$

```
TREE-MINIMUM(x)                TREE-MAXIMUM(x)
1  while x.left ≠ NIL          1  while x.right ≠ NIL
2      x = x.left              2      x = x.right
3  return x                    3  return x
```

Successor function: $T(n) = O(h)$

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2      return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5      x = y
6      y = y.p
7  return y
```

If not the last level node then go right and find the leftmost node. If the last level node then go left and find the first right node. (in upward direction)

Deletion of a node: $T(n) = O(h)$
If the node to be deleted is a leaf node i.e. has no child nodes then delete the node directly.
If it has only one child node then replace the node to be deleted with the child node.
If it has two child nodes then swap the node with its successor (which will be a leaf node or a node with one child) and then delete the node from there directly which can be done via a recursive call as well.

Tree-Delete-Node(x)
    If x.left = Nil and x.right = Nil then delete node x
    If x.left = Nil and x.right != Nil then replace node x with x.right
    If x.left != Nil and x.right = Nil then replace node x with x.left

If x.left != Nil and x.right != Nil then y = Successor(x) and swap node x.value with y.value and Tree-Delete-Node(y)

Dynamic Programming -
Rod cutting problem,
Matrix chain multiplication problem,
0/1 Knapsack problem,
Fibonacci,
Longest common subsequence and
Longest increasing subsequence.

Dynamic Programming: In the divide and conquer problems, we generally divide the given problem into subproblems and combine the results from these subproblems to solve the given/ bigger problem. But most of the time this causes us to do a lot more work than what we would do in a non-recursive/ non - divide and conquer/ iterative method.
This happens because most of the time these subproblems are overlapping and resulting in overall greater complexity. Dynamic programming solves this problem by maintaining these results of sub subproblems in a tabular form instead of computing those sub subproblems results multiple times. We generally use dynamic programming for optimization problems.

Dynamic programming generally follows a sequence of 4 steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of the optimal solution, typically in a bottom up fashion.
4. Construct an optimal solution from computed information.

Rod-cutting problem: Let's discuss the rod cutting problem and how it can be solved using dynamic programming.

Input: Different lengths of rods i and their corresponding prices p_i. Note that here we assume that the lengths of rod are integer values.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Output: The length at which the rod should be cut so that the total value of the pieces is maximized.

For example: If the given length of rod $i = 4$ then the rod can be cut into following configurations,
$length(1 + 1 + 1 + 1) \Rightarrow price(1 + 1 + 1 + 1) \Rightarrow price = 4$
$length(1 + 1 + 2) \Rightarrow price(1 + 1 + 5) \Rightarrow price = 7$
$length(1 + 3) \Rightarrow price(1 + 8) \Rightarrow price = 9$
$length(2 + 2) \Rightarrow price(5 + 5) \Rightarrow price = 10$
$length(4) \Rightarrow price(9) \Rightarrow price = 9.$

The maximum amount is returned 10 by cutting the rod of length 4 into 2 pieces of length 2 each.

(Note that there is even a possible configuration of cut where there is no cut required i.e. in above case the last one.)

Recursive (naive) approach:

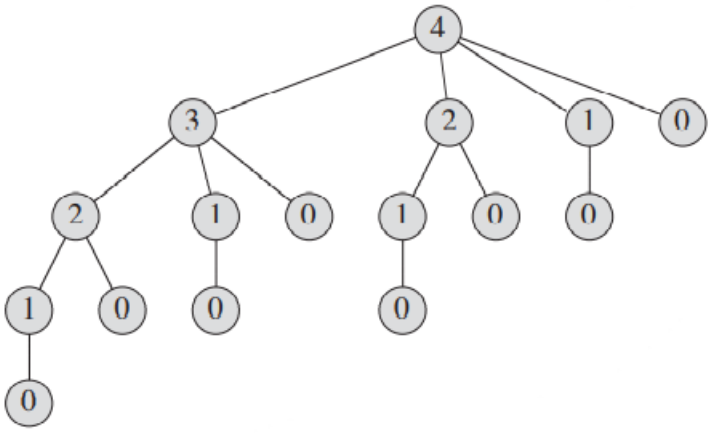
```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

Here p is the array containing prices and n is the length of the rod whose value has to be maximized. If the length of the rod is 0 then the rod can not be cut and moreover the price of rod with length 0 will be 0. The price of the rod cut at the location i can be given as,

$$price = p_i + p_{n-i}$$

In this approach, we cut the rod from left to right and thus the length of rod with price p_i is already the smallest possible length. The term on the right corresponds to the rod split that can be further split and price can be maximized. Thus for the term on the right we make a recursive call to the $Cut - Rod(p, n)$ function to get the maximum price possible further. The lines 3, 4 and 5 basically find the maximum possible price.

The recursion tree for the $n = 4$ can be shown as follows,



The number of nodes at each level can be given using the combination formula. The recurrence formula can be given as follows,

$T(n) = \sum_{k=0}^{n} C(n,k) = 2^n$ and thus $T(n) = 2^n$.

Thus we can see that the time complexity is exponential to n.

Memoization based approach:
In this approach, we first initialize an array r with all values set to -∞ and length $n + 1$ in the base case function/ initialization function. And in the recursive function, we first check if the maximum price is already calculated and stored at $r[i]$. And if not then we find the maximum price for i^{th} location and store it at $r[i]$.

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = -∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```

Bottom up approach:

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j − i])
7      r[j] = q
8  return r[n]
```

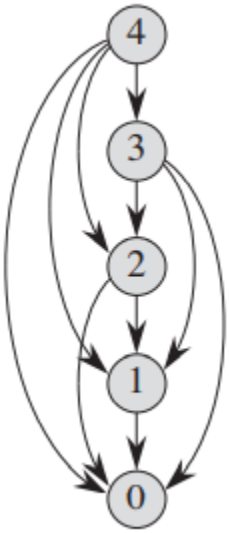
The bottom-up approach solves the smaller problems first and then the larger ones. The time complexity of Memoized approach and bottom-up approach is same i.e. $O(n^2)$ but the bottom-up approach out performs a memoized approach as it has a smaller constant factor.

Reconstructing a solution: The above methods give us the maximum price that can be achieved by cutting the rod but we didn't find the solution. The solution can be easily found by just modifying the above bottom-up approach function. We just need to change the $q = max(…)$ part with a comparison statement and if a cut position is found that maximizes then we store the cut position as well in the array s.

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
 1  let r[0..n] and s[0..n] be new arrays
 2  r[0] = 0
 3  for j = 1 to n
 4      q = -∞
 5      for i = 1 to j
 6          if q < p[i] + r[j - i]
 7              q = p[i] + r[j - i]
 8              s[j] = i
 9      r[j] = q
10  return r and s
```

Subproblem graphs:

Subproblem graph shows the set of subproblems that are involved and how they depend on each other. Below is an example of subproblem graph with $n = 4$ and



Matrix chain multiplication problem:

Suppose we are given x number of matrices that need to be multiplied with each other. What is the most optimal way in which the matrices can be multiplied so that the number of multiplications is minimum? This is nothing but the matrix multiplication problem.

For example: Matrix A_1 has size 2×3, matrix A_2 has size 3×4 and matrix A_3 has size 4×3. Here the matrices $A_1.A_2.A_3$ can be multiplied with each other but the order in which they are multiplied can differ resulting in different numbers of multiplications required.

Following are the 2 ways in which this matrix chain can be multiplied.

1. $A_1.(A_2.A_3)$
2. $(A_1.A_2).A_3$

The number of multiplications required for the first is as follows,

Multiplying A_2 and A_3 requires $3 \times 4 \times 3 = 36$ multiplications. And multiplying this $(A_2.A_3)$ with A_1 will require $2 \times 3 \times 3 = 18$ multiplications. Therefore a total of $36 + 18 = 54$ multiplications are required for the 1st way.

For 2nd way, multiplying A_1 and A_2 requires $2 \times 3 \times 4 = 24$ multiplications. And multiplying this $(A_1.A_2)$ with A_3 will require $2 \times 4 \times 3 = 24$ multiplications. Therefore a total of $24 + 24 = 48$ multiplications are required for the 2nd way.

Thus we can see that the number of multiplications required for the 2nd way is less than the 1st way. Our problem is to find the way which requires the least number of multiplications.

(Here for matrix chains of 3, there are 2 possible ways in which the multiplications can be done. Similarly for chain of 4 there are 5 possibilities. This number of possible ways can be given using a sequence of Catalan numbers. If n is the number of matrices in the matrix chain then the number of possible ways is $C(2(n-1),(n-1))/n$.)

This problem of finding the order of multiplication or basically where the parentheses should be placed can be easily defined as a recursive problem. But before that lets define the problem mathematically:

Recursive approach to find minimum no. of multiplications:

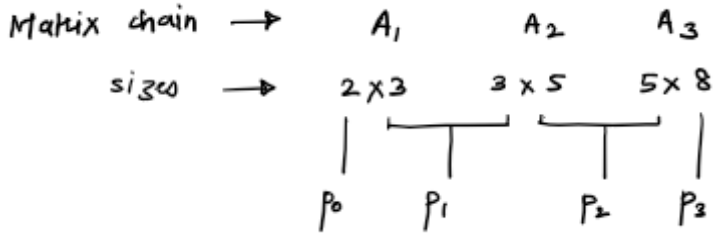
Input: Matrix chain is $A_i, A_{i+1},\ldots, A_j$. And the size of these matrices are given by in array p: $A_i \Rightarrow p_{i-1} \times p_i, A_{i+1} \Rightarrow p_i \times p_{i+1},\ldots\ldots, A_j \Rightarrow p_{j-1} \times p_j$. *(Note that, to multiply two matrices the number of rows of the first and columns of the second show the same.)*

Output: $m[i,j]$ is the minimum number of multiplications required to multiply matrix chain from A_i to A_j.

Formula to find the min. no. of multiplications:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j\ , \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j\ . \end{cases}$$

The above formula can be explained with a small example.
Example:



FORMULA : $m[i,j] = \min_{i \le k < j} \{ m[i,k] + m[k+1,j] + (p_{i-1} \times p_k \times p_j) \}$ — if $i \neq j$

$= 0$ → if $i = j$

# of multiplications to multiply matrices A_i to A_j.

Solution : $\min[1,3] = \min_{1 \le k < 3}$ $\begin{cases} k=1 \; m[1,1] + m[2,3] + (p_0 \times p_1 \times p_3), \\ k=2 \; m[1,2] + m[3,3] + (p_0 \times p_2 \times p_3) \end{cases}$

In this above equation, we know the values of $m[1,1]$ & $m[3,3]$ but, the values of $m[2,3]$ & $m[1,2]$ need to be found.

$m[2,3] = \min_{2 \le k < 3}$ $k=2 \{ m[2,2] + m[3,3] + (p_1 \times p_2 \times p_3) \}$

$= 0$

On finding $m[2,3]$, we calculate $m[1,2]$

$m[1,2] = \min_{1 \le k < 2}$ $k=1 \{ m[1,1] + m[2,2] + (p_0 \times p_1 \times p_2) \}$

0

These values are now substituted in main equation to get $m[1,3]$.

The values of $m[2,3]$ and $m[1,2]$ are calculated via recursive calls. As this was a small example i.e. matrix chain of only 3 there were only 2 calculations that were repeated i.e. $m[1,1]$ is calculated in main calculation $m[1,3]$ as well as in calculating a subproblem of $m[1,2]$.

This problem of calculating subproblems can be (obviously, duh!!) solved using dynamic programming and an optimal solution can be obtained.

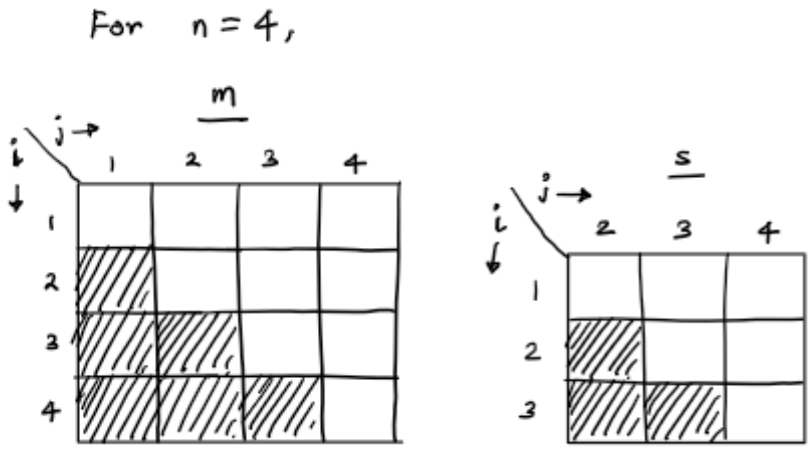
Matrix chain multiplication using dynamic programming:

Dynamic programming implementation can be easily done by just calculating the smaller values first (start from bottom) and store those values so that they can be used to solve larger problems without recursion.

For that we maintain two different arrays m and s. m stores the minimum cost/ number of multiplications and s stores the value of i where the chain is split for multiplication with minimum multiplications.

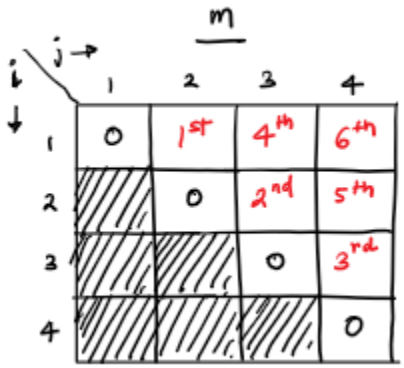
```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

In the above function p is an array that contains the sizes of all arrays. Array p contains values from p_{i-1} to p_j for matrices A_i to A_j. Thus the number of matrices can be given as length of array $p - 1$. Now instead of imagining the size of array m and s as given in the above function, for easier understanding, the size of m is $n \times n$ out of which we use only half and similarly for s, the size of s is $(n - 1) \times (n - 1)$.



The shaded portion of the arrays m and s are not used.

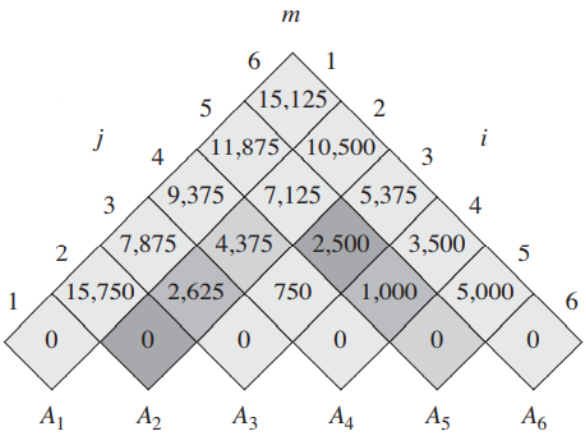
The diagonal values of array m are set to zero which forms the base case of the formula discussed in the recursive approach.
The first two for loops adjust the values of i and j such that the $m[i, j]$ is calculated in following order:

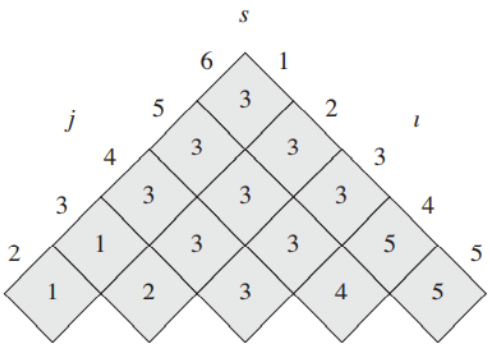


That is the values are calculated in diagonal fashion. On finding a minimum number of multiplication q, the value of k is stored in the array s at $s[i, j]$ so that the location of the split is recorded for back tracing the solution.

On executing the above function on matrix chain with $n = 6$ suppose we get the following.

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25





Initial: $A_1.A_2.A_3.A_4.A_5.A_6$. This chain is from $1\ to\ 6$. Thus we check $m[1, 6]$ and $s[1, 6]$. To get 15125 multiplications we need to split the matrix chain at A_3. Therefore, matrix chain now is,

$$(A_1.A_2.A_3).(A_4.A_5.A_6)$$

Now for $(A_1.A_2.A_3)$ we check $m[1,3]$ and $s[1,3] = 1$ and thus we split at A_1. Thus the matrix chain now is,

$$((A_1).(A_2.A_3)).(A_4.A_5.A_6)$$

Similarly for $(A_4.A_5.A_6)$, $s[4,6] = 5$ and thus we split at A_5. Thus the matrix chain now is,

$$((A_1).(A_2.A_3)).((A_4.A_5).(A_6))$$

Memoized matrix chain:

```
MEMOIZED-MATRIX-CHAIN(p)
1  n = p.length − 1
2  let m[1..n, 1..n] be a new table
3  for i = 1 to n
4      for j = i to n
5          m[i, j] = ∞
6  return LOOKUP-CHAIN(m, p, 1, n)

LOOKUP-CHAIN(m, p, i, j)
1  if m[i, j] < ∞
2      return m[i, j]
3  if i == j
4      m[i, j] = 0
5  else for k = i to j − 1
6          q = LOOKUP-CHAIN(m, p, i, k)
               + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
7          if q < m[i, j]
8              m[i, j] = q
9  return m[i, j]
```

0/1 Knapsack problem:
In this problem we are given a list of objects and their corresponding weights and prices/ profits. And we need to find the objects that can be fit in a bag that can carry a certain weight and also have a maximum profit.

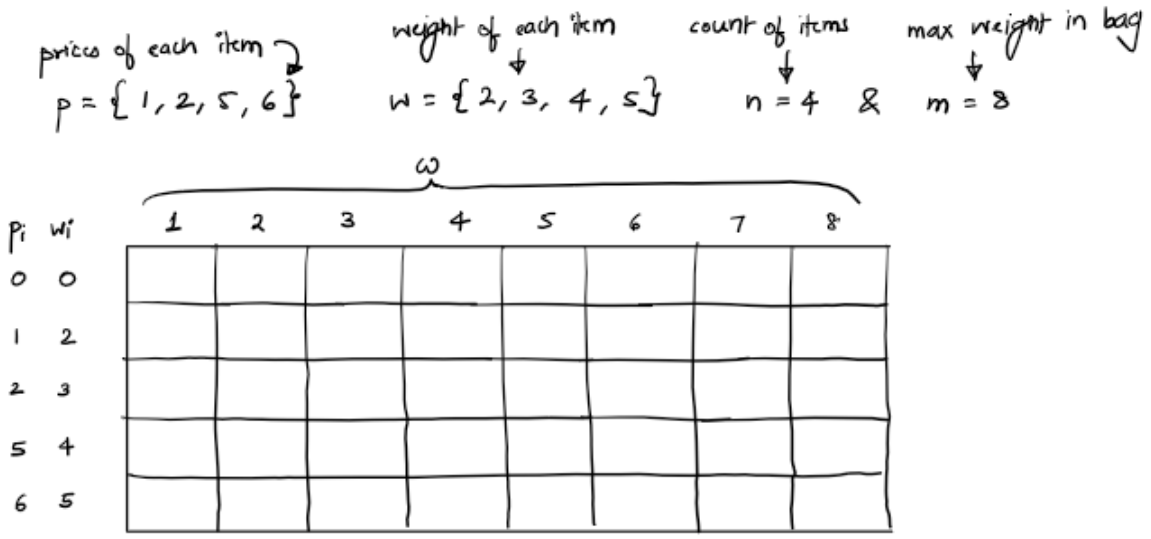
Input: Number of objects is n and array $p = [p_1, p_2, ..., p_n]$ consists of prices of all n objects. The array $w = [w_1, w_2, ..., w_n]$ consists of weights of all n objects. The maximum weight that the bag can carry is m.
Output: Maximum profit that can be achieved considering the maximum weight and the list of objects that need to be collected to get that maximum profit.

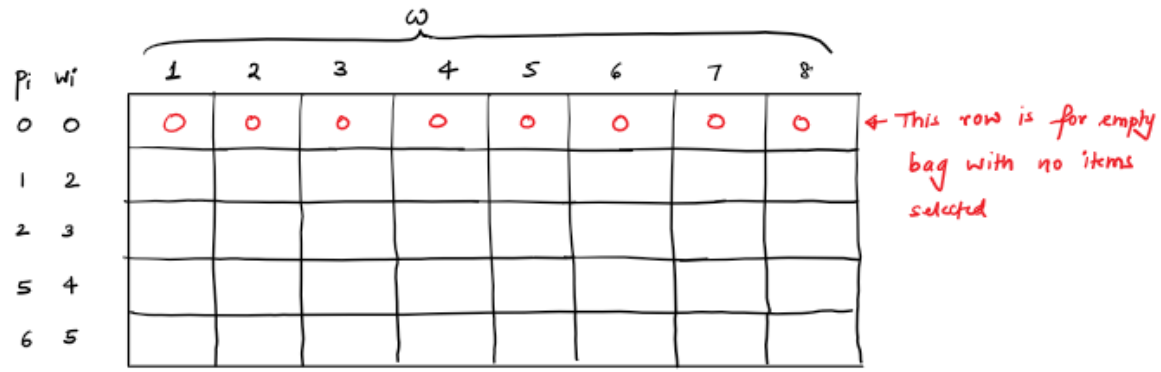
Let's assume an array $x = [x_1, x_2, ..., x_n]$ where each element is either 0 or 1 indicating whether the corresponding item is considered or not. For example, if $x = [1, 0, 0, 1]$ for problem size of 4 then we are considering the 1st and last objects only.

In short, as per the problem, we need to maximize $\sum_i p_i . x_i$ while $\sum_i w_i . x_i \leq m$.

Now as per naive approach, we need to iterate through each and every possible combination of x and find $\sum_i p_i . x_i$ with maximum value and $\sum_i w_i . x_i \leq m$. But doing so will require 2^n iterations and calculations i.e. exponential time complexity. This can be solved using dynamic programming as follows.

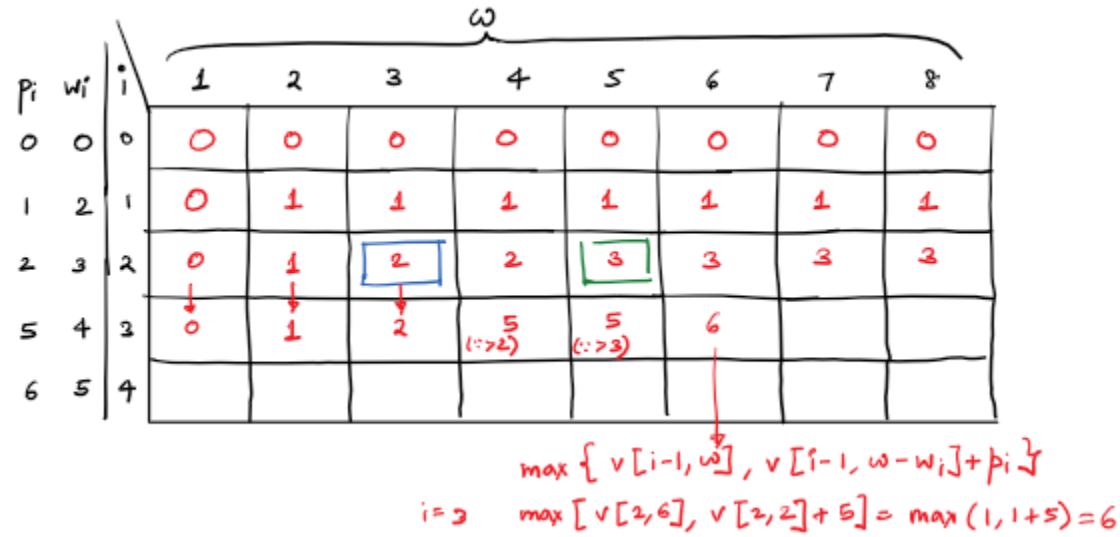
Dynamic programming approach/ tabulation method:
Let's solve a small knapsack problem.
$n = 4,\ p = [1, 2, 5, 6],\ w = [2, 3, 4, 5],\ m = 8.$





We use the following formula to find value at each location.
(Considering that the table name is V)

$$V[i, w] = \max \{ V[i-1, w],\ V[i-1, w - W_i] + P_i \}$$



$\max \{ V[i-1, w],\ V[i-1, w - W_i] + P_i \}$

$i = 3$ $\max [V[2,6],\ V[2,2] + 5] = \max (1, 1+5) = 6$

At this point, there are two possible objects that can be selected

i.e. → $(p_1, w_1) = (1, 2)$ OR
→ $(p_2, w_2) = (2, 3)$

As maximum price is achieved by object 2 : we select 2 & write p_2.

At this point, the maximum possible weight is 5 and the objects that we are considering have weights 2 & 3. Therefore we can select both objects and thus the profit/ price is $p_1 + p_2 = 1 + 2 = 3$

p_i	w_i	i \ w	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	1	1	1	1	1	1	1
2	3	2	0	1	2	2	3	3	3	3
5	4	3	0	1	2	5	5	6	7	7
6	5	4	0	1	2	5	6			

till weight reaches w_{i-1} copy values from above

→ $\max(5, v[3,0] + 6)$ = $\max(5, 6) = 6$ (∵ $v[3,0]$ does not exist $v[3,0] = 0$)

p_i	w_i	i \ w	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	1	1	1	1	1	1	1
2	3	2	0	1	2	2	3	3	3	3
5	4	3	0	1	2	5	5	6	7	7
6	5	4	0	1	2	5	6	6	7	8

$\max(7, v[3, 8-5] + 6)$
$\max(7, 2+6) = \max(7, 8) = 8$

Here we were basically checking if adding an item in the bag increases the profit or not. If it is increasing then we consider that weight and do the next calculation.

To back trace the items that should be selected we find the highest price in the calculated table. In the above table we can see that the maximum value is 8 and that exists only in row with $i = 4$. Thus we can say that the price was maximized after adding the 4th item.
$Selected\ items\ list: 4$

We now subtract the price of 4th item i.e. $8 - 6 = 2$. We now find these 2 in the table. We can find this item in row with $i = 3\ and\ 2$ but not in 1. Thus we can say that this price was added by item 2.
$Selected\ items\ list: 4, 2$

On subtracting 2 from remaining weight $2 - 2 = 0$. As the remaining weight is 0 we stop.

Thus the selected items are 2 and 4.

The time complexity of 0/1 Knapsack problem is $O(2^n)$ but the dynamic programming reduces it to $O(n * m)$ where n is the number of items and m is the capacity of the bag.

Fibonacci using dynamic programming:
We have defined Fibonacci sequence using recursive definition as follows:

$$\begin{aligned} Fib(n) &= 0 && if\ n = 0 \\ &= 1 && if\ n = 1 \\ &= Fib(n-1) + Fib(n-2) && if\ n > 1 \end{aligned}$$

Thus the running time complexity can be given as,
$T(n) = T(n-1) + T(n-2) + O(1)$

For finding the upper bound, the value of n is very larger and thus we can say that the $T(n-1) \approx T(n-2)$. Substituting $T(n-2)$ as $T(n-1)$ we get,

$$\begin{aligned} T(n) &= 2T(n-1) + O(1) \\ &= 2(2T(n-2) + O(1)) + O(1) \approx 4T(n-2) + O(1) \\ &= 4(2T(n-3) + O(1)) + O(1) \approx 8T(n-3) + O(1) \end{aligned}$$

And thus we can say,

$$T(n) = 2^k T(n-k) + O(1)$$

On substituting $n - k = 1$ i.e. $k = n - 1$ we get,

$$T(n) = 2^{n-1} T(1) + O(1)$$

Thus we can say that, $T(n) = 2^{n-1} + O(1) = O(2^n)$.

The same can be solved using dynamic programming as well with time complexity linear to n.

```
Fibonacci_init(n):
    memo = [ ]
    for i = 0 to n:
        if i == 0 then memo[i] = i
        if i == 1 then memo[i] = i
        else memo[i] = − ∞
    return Fibonacci_final(n, memo)

Fibonacci_final(n, memo):
    if memo[n] != − ∞:
        return memo[n]
    memo[n] = Fibonacci_final(n-1)+Fibonacci_final(n-2)
    return memo[n]
```

Here as every value calculated is stored and referred to when calculating the higher values, lesser computation is required.
Time complexity of this approach is: $T(n) = O(n)$

Longest common subsequence:
In this problem, we try to find the largest common subsequence length.
For example: $S_1 = bd$ and $S_2 = abcd$ then the longest common subsequence is bd.
A naive approach would be to select the larger string and generate all possible substrings. For every generated substring, check the common subsequence and note the maximum length of common subsequence found after comparing with every substring generated from 1st string.
Example:

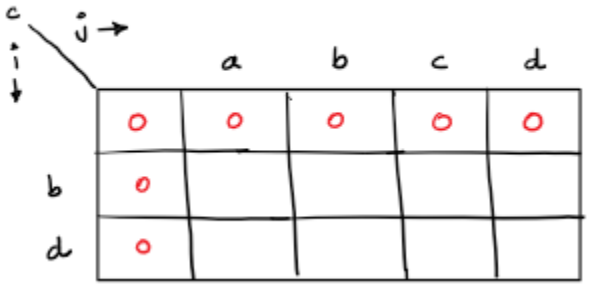
length	Substring generated	Common subsequence length with bd
0		0
1	a	0
	b	1
	c	0
	d	1
2	ab	1
	ac	0
	ad	1
	bc	1
	bd	2
	cd	1
3	abc	1
	abd	2
	acd	1
	bcd	2
4	$abcd$	2

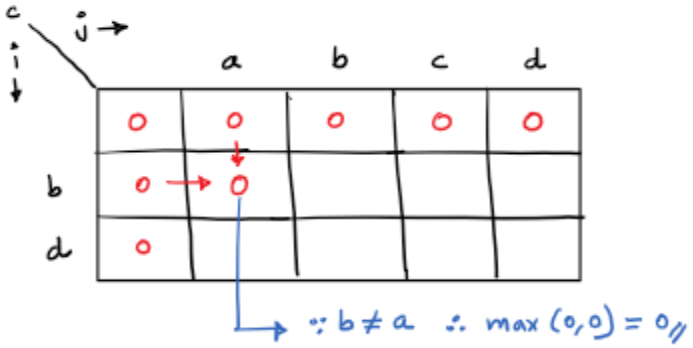
Therefore the maximum substring length found is 2. The problem with approach is that the time required to solve the problem grows exponentially. That is, the time complexity is $O(max(2^n, 2^m).min(n.m))$. Or for simplicity, assuming that $m > n$, $T(n) = O(2^m.n)$

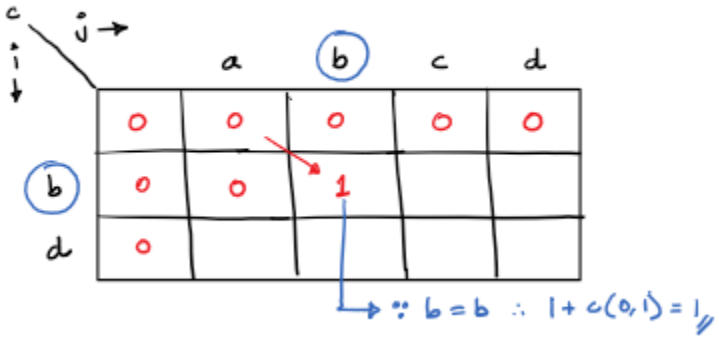
Solving using dynamic programming: This same problem can be solved by using dynamic programming with time complexity of $O(nm)$. Let's see how we can do that.

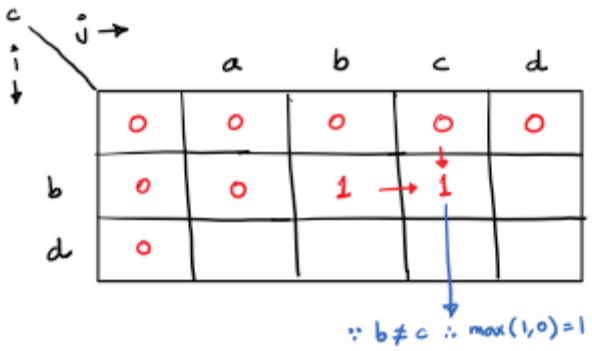
We create a 2d array/ table with longer string elements as columns and smaller string elements as rows.
The first row and first column are filled with zeros to form the base case.
Each row and column are calculated using the following formula (assuming that the table name is c and x, y are the two input strings).

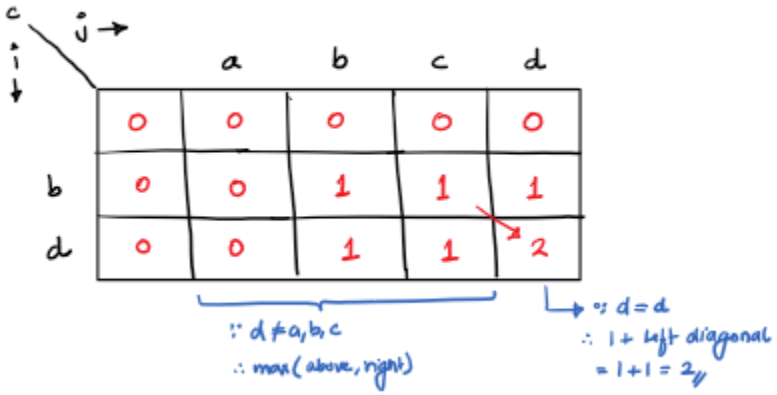
$$
\begin{aligned}
c[i,j] &= 0 && if\ i = 0\ or\ j = 0 \\
&= c[i-1, j-1] && if\ i,j > 0\ and\ x_i = y_j \\
&= max(c[i-1, j], c[i, j-1]) && if\ i,j > 0\ and\ x_i \neq y_j
\end{aligned}
$$











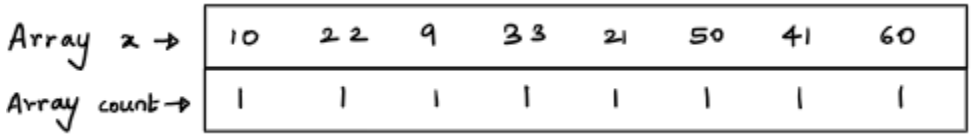
The time complexity of the above method can be easily guessed as $O(m.n)$.
Also the space complexity is $O(m.n)$ if you are storing it in a 2D array. But it can be reduced to $O(n)$ or $O(m)$ if we update the same row instead of having separate rows for b and d as in the above example.

Longest increasing subsequence:
An increasing subsequence is the subsequence of a given sequence which has all values in increasing order. For example: $s = 1, 4, 2, 6, 10, 2, 23$. One possible increasing subsequence is $1, 2, 6, 10, 23$. Another one is $1, 2, 10, 23$. We want to find the largest possible increasing subsequence.

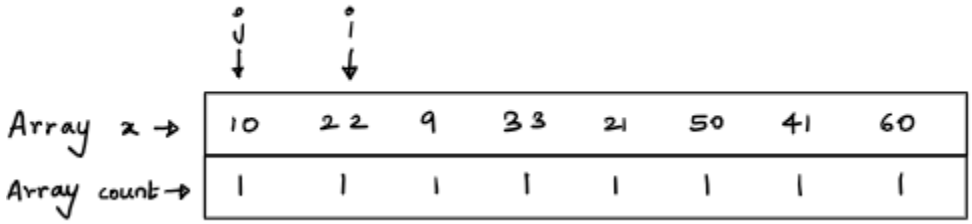
It is again possible here to generate all possible subsequences and then check if they are in increasing order but the time complexity will be exponential in that case. Let's directly jump to the dynamic programming approach which achieves the time complexity of $O(n^2)$.

Longest increasing subsequence using dynamic programming:
We create a 2D array with the elements of the given sequence as columns and one row that we calculate. This row will store the maximum number of increasing subsequences that terminate at that point of the sequence. It will be more clear with the help of an example.

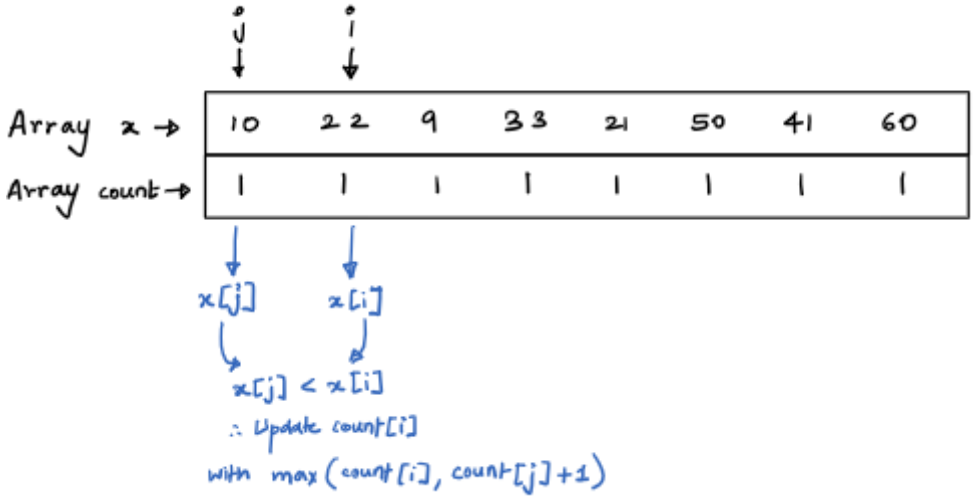
Example: Given array $x = 10, 22, 9, 33, 21, 50, 41, 60.$
We create the following arrays,



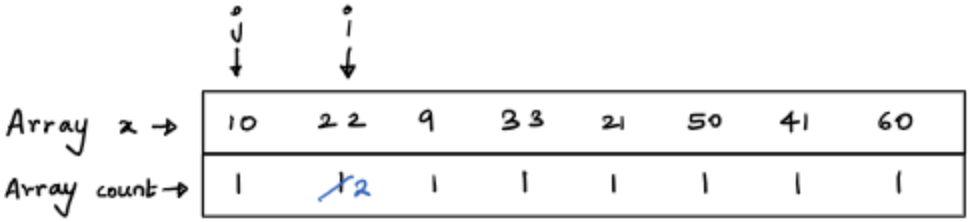
We first initialize the count array with all its values as 1.



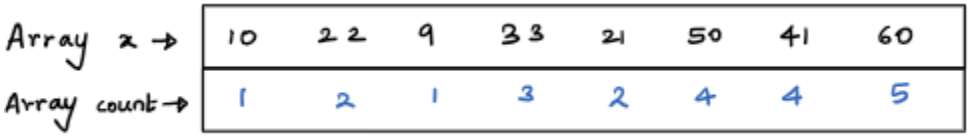
The pointers j and i are initialized as above. We increment the value of pointers j and i as follows:(Assuming that the array index starts from 0 and ends at (n-1))
Pointer i: from 0 to (n-1)
Pointer j (inside for loop for pointer i): from 0 to (i-1)



In every iteration of j, we compare the values in array x pointed by j and i. And if $x[j] < x[i]$ then we update the value in array $count$ pointed by pointer i with $max(count[i], count[j] + 1)$ as shown below.



On iterating through every possible values of i and j as given above we get the following counts.



The pseudocode can be given as follows,

```
Longest increasing subsequence(x):
    count = [ ]
    for i = 0 to (n-1)
        count[i] = 1
    for i = 0 to (n-1)
        for j = 0 to (i-1)
            if x[j] < x[i]
                count[i] = max(count[i], count[j] + 1)
```

Finding the largest increasing subsequence elements can be found from the $count$ and x arrays as follows,
Highest count is 5 where the value is 60. Therefore 60 has to be included in the increasing subsequence.
$Selected\ items:\ 60$
5-1=4. Now we find 4 on the left of 5. There are two 4s i.e. for values of 50 and 41. So we can consider any one from those.
$Selected\ items:\ 60, 41$ or $Selected\ items:\ 60, 50$
4-1=3. Count 3 is found at the value of 33. Add it to the selected items list.
$Selected\ items:\ 60, 41, 33$ or $Selected\ items:\ 60, 50, 33$
3-1=2. Count 2 is found at the value of 22. Adding it to the selected items list.
$Selected\ items:\ 60, 41, 33, 22$ or $Selected\ items:\ 60, 50, 33, 22$
2-1=1. Count 1 is found at the value of 10. Adding it to the selected items list.
$Selected\ items:\ 60, 41, 33, 22, 10$ or $Selected\ items:\ 60, 50, 33, 22, 10$

Now reverse the $Selected\ items$ list to get the longest increasing subsequence.
$Selected\ items:\ 10, 22, 33, 41, 60$ or $Selected\ items:\ 10, 22, 33, 50, 60.$

Time complexity analysis:
The pointer j iterates over values of 0 to i where i iterates over n different values.

Values of j	Count
0	1
0,1	2
0,1,2	3
0,1,2,3	4
….	….
0,1,2,3,....., (n - 1)	n
Sum =	$\sum_{i=1}^{n} i = n(n+1)/2 = O(n^2)$

Thus the time complexity can be given as $T(n) = O(n^2)$.

The space complexity is $O(n)$ as it uses only one extra array $count$ of length n.

Graphs,
Graph representation and
Graph traversal algorithms.

Graphs: Graphs are representations of structures, set relations, and states and state-transitions. Following are different examples of graphs:

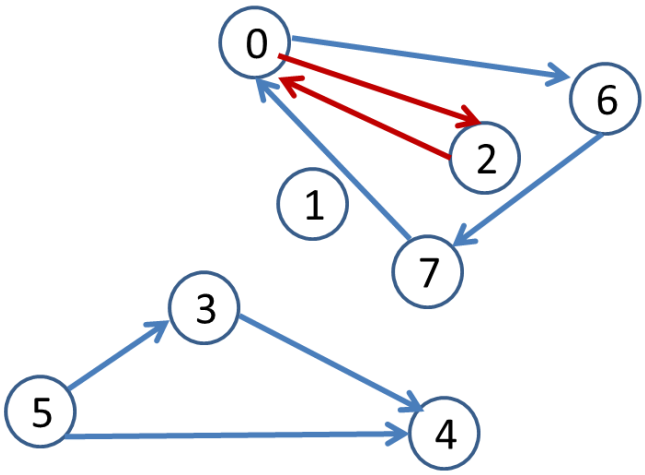
1. Direct representation of a real-world structures: Networks (roads, computers, social)
2. States and state transitions of a problem: Game-playing algorithms (e.g., Rubik's cube), Problem-solving algorithms (e.g., for automated proofs).
3. Social media networks
4. Web page links graph and Maps (vertices are road intersections and edges are road segments)

A graph is defined as $G = (V, E)$ where V is set of vertices (or nodes) and E is the set of edges. Edge is a pair of two vertices i.e. if v_1 and v_2 are two vertices then edge e joining those vertices can be given as $e = (v_1, v_2)$.

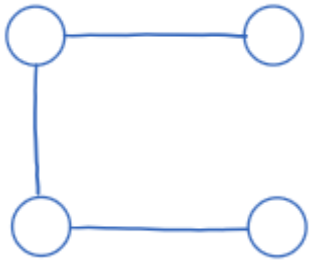
Types of graphs:

Undirected graphs: Undirected graphs have no direction for the edges. In undirected graphs an edge (A, B) means that we can go from node A to node B and from node B to node A as well.

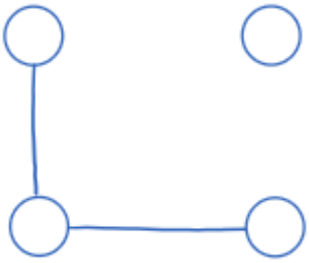
Directed graphs: Directed graphs have directions for edges. An edge (A, B) means that we can go from node A to node B but not the other way. To go both ways there must be an edge present between both nodes in both directions. Eg. edges between node 0 and 2 in below diagram.



Connected graphs: A graph is a connected graph if all its nodes have at least one edge.



(a) Connected graph



(b) Not connected graph

If there are n number of nodes in the graph then for a graph to be a connected graph the number of edges e required can be given as,

1. Minimum number of edges, $min(|e|) = n - 1$
2. Maximum number of edges, $max(|e|) = n(n - 1)/2$

Representation of graphs: There are many different efficient ways in which a graph can be represented than the two ways which we will be using/ discussing below.

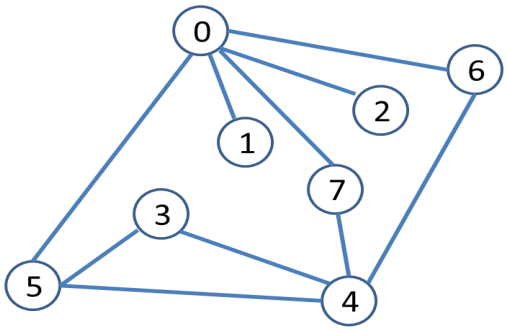
1. Adjacency matrix
2. Adjacency list

Adjacency matrix: If there are n number of vertices then the adjacency matrix M consists of $n \times n$ matrix such that,

$M[v_1][v_2] \quad = 1 \quad if\ there\ is\ an\ edge\ between\ v_1\ and\ v_2$

$\quad\quad\quad\quad\ = 0 \quad if\ there\ is\ no\ edge\ between\ v_1\ and\ v_2$

Example:

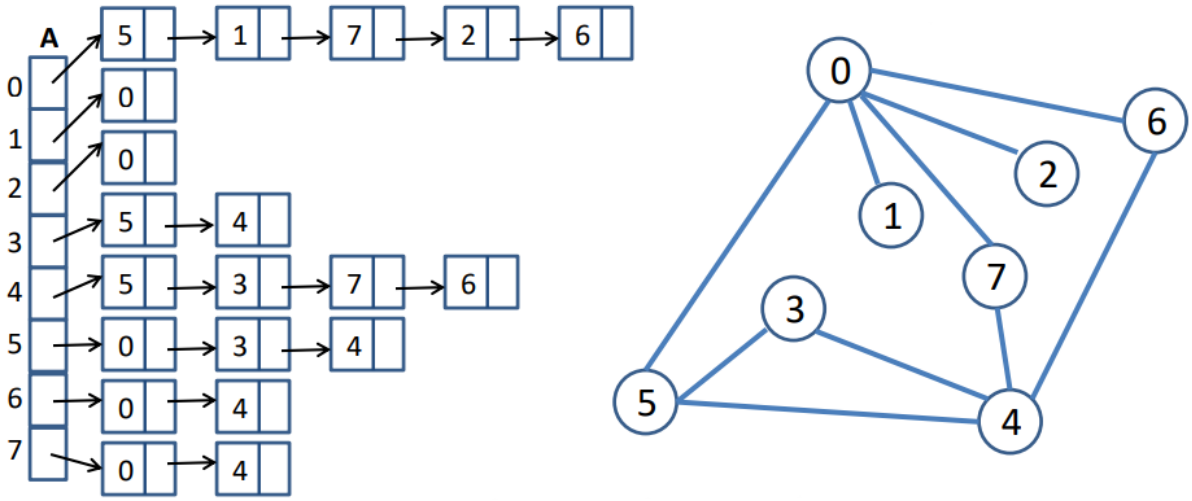


	0	1	2	3	4	5	6	7
0	0	1	1	0	0	1	1	1
1	1	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0
4	0	0	0	1	0	1	1	1
5	1	0	0	1	1	0	0	0
6	1	0	0	0	1	0	0	0
7	1	0	0	0	1	0	0	0

Space complexity is $O(n^2)$ and the time complexity is as follows:

1. To add/ remove/ check edge: $O(1)$
2. To find neighbors: $O(n)$

Adjacency lists: The edges of a graph are represented by an array of linked lists. If the name of the array is A then $A[v_1]$ consists of a linked list consisting of the neighbors of vertex v_1.



Space complexity: If m are the number of edges and n are the number of nodes/ vertices then the space complexity is $O(m + n)$. This is because the array A will require $O(n)$ space and every edge is represented as a node in linked lists, therefore $O(m)$ for directed and $O(2m) = O(m)$ for an undirected graph. Thus in total the space complexity is $O(m + n)$.

Time complexity:

1. To add/ remove/ check edge: $O(n)$. In worst case scenario, the node might contain edges to all n nodes and thus to check if edge exists it will take $O(n)$. To add or remove, we first need to check if the edge exists or not which will require $O(n)$ running time and then add or remove edge in $O(1)$.
2. To find neighbors: $O(k)$ where k is the number of neighbors. This is because we just need to go through the linked list corresponding to that particular node.

Graph traversal: In graph traversal we are given a graph and a source node and we need to find the path such that we go through every node in the graph.
Input: Start/ source vertex
Output: A sequence of nodes resulting from graph traversal.

There are different types of graph traversal techniques. Few of them which we will be discussing are listed below:

1. Level order or Breadth first traversal
2. Depth first traversal

Breadth first Search/ Traversal: In this traversal technique, the nodes are traversed level by level and thus is also called as level order traversal. (A great visualization for BFS and DFS is at this link: https://visualgo.net/en/dfsbfs)
In BFS we first explore all the neighbors of the source node and then explore neighbors of neighboring nodes and so on. The pseudocode of BFS is as follows. The implementation requires a queue (FIFO) in which all neighbors are stored one by one. Each node consists of an explored flag which is set if the node is explored.

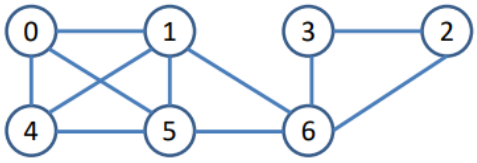
Pseudocode:

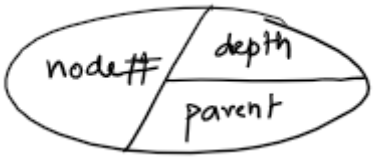
```
Breadth_First_Search(G,s)  // G is the given graph and s is the source node
    mark s as explored
    Q <= a queue data structure initialized with s
    while Q is not empty
        v = Dequeue(Q)
        for edge(v, w) in G's adjacency list corresponding to node v
            if w is unexplored
                mark w as explored
                Enqueue(Q, w)
```

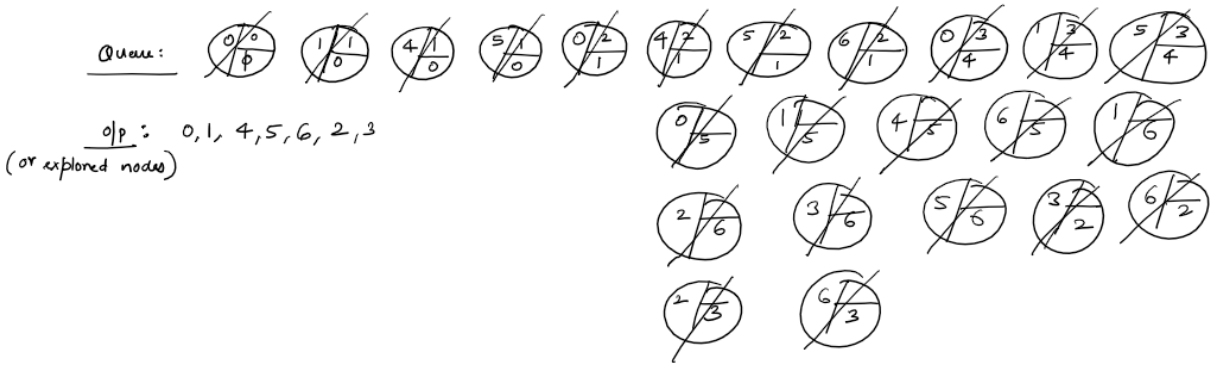
Time complexity of BFS depends on the representation of the graph used i.e. adjacency list or adjacency matrix.

1. Adjacency list: $O(n + m)$
2. Adjacency matrix: $O(n^2)$

And space complexity is $O(n)$.



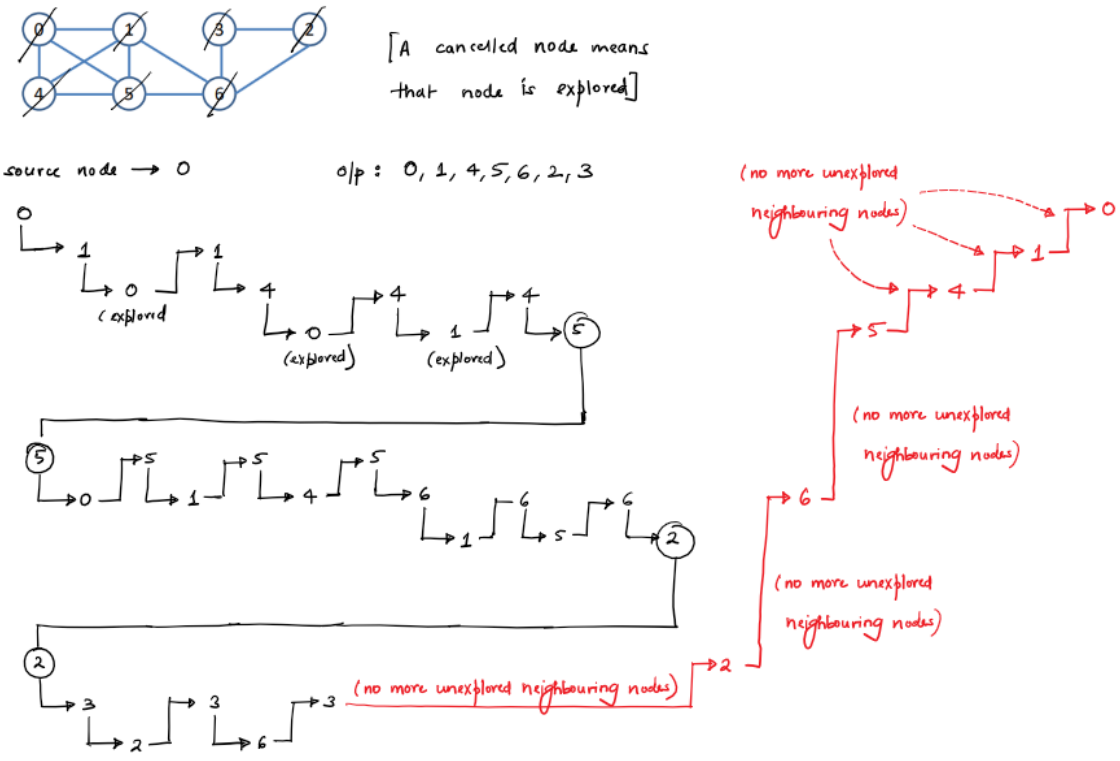




Depth First Search/ Traversal: In DFS, we first explore the neighboring nodes of neighbors of the current node. Here we are using a LIFO queue (basically stack) instead of a FIFO queue.

Pseudocode:

```
Depth_First_Search(G,s)
    stack = stack data structure initialized with s
    while stack is not empty
        v = pop(stack)
        if v is unexplored
            mark v as explored
            for each neighbor node w of node v in graph G
                push(stack,w)
```



Time complexity of DFS is $O(n^2)$ where n is the number of nodes in the given graph.

Length of shortest path: The BFS algorithm with few modifications can be used to find the length (number of hops) of shortest path between source node and destination node. This is because BFS will first check the neighbors at a distance of one hop, then the ones at 2 hops and so on. Basically the modified BFS code will check if the destination node is at distance of i hops for source node where $i = 1, 2, 3, 4$….

Pseudocode:

```
Shortest_path(G,s,d)  // G is the given graph, s is the source node and d is destination node
    mark s as explored
    l(s) = 0
    l(v) = ∞ (∀v ≠ s)
    Q <= a queue data structure initialized with s
    while Q is not empty
        v = Dequeue(Q)
        for edge(v, w) in G's adjacency list corresponding to node v
            if w is unexplored
                mark w as explored
                l(w) = l(v) + 1
                if w == d then return l(w)
                Enqueue(Q, w)
    return "error: node d is not reachable from node s."
```

Greedy algorithms - Minimum spanning tree using Prim’s algorithm and Kruskal’s algorithm.

Reference: Algorithms by S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani and Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein

Greedy algorithms: A game like chess can be won only by thinking ahead: a player who is focused entirely on immediate advantage is easy to defeat. But in many other games, such as Scrabble, it is possible to do quite well by simply making whichever move seems best at the moment and not worrying too much about future consequences.
This sort of myopic behavior is easy and convenient, making it an attractive algorithmic strategy. Greedy algorithms build up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Although such an approach can be disastrous for some computational tasks, there are many for which it is optimal. Our first example is that of minimum spanning trees.

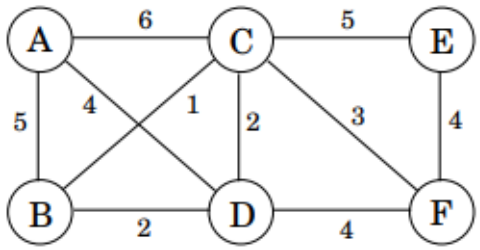
Minimum-cost Spanning Trees (MST): A graph can consist of multiple edges between nodes and cycles. Such a graph can be converted to MST by removing selective edges that are creating cycles in the graph.
Property: Removing a cycle edge can not disconnect a graph.
Thus an MST is an acyclic and connected graph with minimum edge weights. It can be formally defined as,

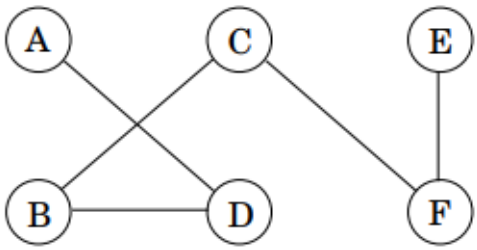
Input: An undirected graph $G = (V, E)$; edge weights w_e

Output: A tree $T = (V, E')$, with $E' \subseteq E$, that minimizes $weight\ (T) = \sum_{e \in E'} w_e$.

Prim's algorithm: Prim's algorithm finds the minimum spanning tree from a given graph. It does so by first selecting a random node, let's say A and adds it to a list called visited. Then it searches among all the edges connecting the nodes that are not present in the visited list and selects the edge with minimum weight. It adds the node connected by that edge in the visited nodes list. It repeats the same steps until all the nodes are added to the visited list.

```
procedure prim(G, w)
Input:    A connected undirected graph G = (V, E) with edge weights w_e
Output:   A minimum spanning tree defined by the array prev

for all u ∈ V:
    cost(u) = ∞
    prev(u) = nil
Pick any initial node u_0
cost(u_0) = 0

H = makequeue(V)      (priority queue, using cost-values as keys)
while H is not empty:
    v = deletemin(H)
    for each {v, z} ∈ E:
        if cost(z) > w(v, z):
            cost(z) = w(v, z)
            prev(z) = v
            decreasekey(H, z)
```





Set S	A	B	C	D	E	F
$\{\}$	0/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
A		$5/A$	$6/A$	$4/A$	∞/nil	∞/nil
A, D		$2/D$	$2/D$		∞/nil	$4/D$
A, D, B			$1/B$		∞/nil	$4/D$
A, D, B, C					$5/C$	$3/C$
A, D, B, C, F					$4/F$	

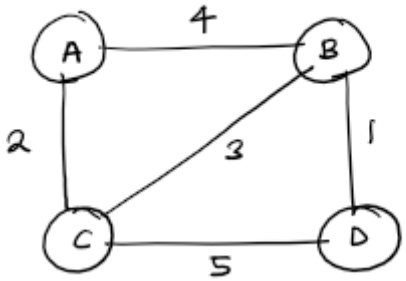
If the weights of every edge are unique then the minimum spanning tree will be unique orelse not. The total edge weights of the MST can be calculated by adding the weights of all edges of the MST. For the above MST, the total edge weight is $2 + 1 + 4 + 4 + 3 = 14$.
The time complexity of Prim's algorithm depends on what data structures are used for implementation.

1. Adjacency matrix, searching: $O(n^2)$ where n is the number of vertices/ nodes
2. Binary heap and adjacency list: $O(nlogn + mlogn)$ where n is the number of nodes and m is the number of edges.

Kruskal's algorithm: Kruskal's algorithm uses a different approach for finding an MST. It searches for edges with minimum weights. It sorts the edges in ascending order and selects them such that the edges do not form a cycle. For doing so, we use set operations.

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3      MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
9  return A
```

Let's take a small and quick example.



Steps:
1. Create a set for every node in the graph. $sets = \{A\}, \{B\}, \{C\}, \{D\}$.
2. Create an empty set to add edges that make the MST. Let that set be $a = \{\}\ or\ \phi$.
3. Arrange all sets in non-decreasing order of their weights.

Edge	Weight
$B - D$	1
$A - C$	2
$B - C$	3
$A - B$	4
$C - D$	5

4. For every edge (in above non decreasing order) check if end nodes of that edge belong to the same set in $sets$ defined in step 1. If they belong to the same set then ignore and go to the next edge. If they belong to different set then add that edge to set a and combine/ union the sets that they both belong to.

i. Edge $B - D$:
$Current\ sets = \{A\}, \{B\}, \{C\}, \{D\}$
B belongs to set $\{B\}$ and D belongs to set $\{D\}$. And $\{B\} \neq \{D\}$. Therefore,
$a = a \cup \{(B, D)\}$
$\therefore a = \{(B, D)\}$
$sets = (sets - \{B\} - \{D\}) \cup \{B, D\}$
$\therefore sets = \{A\}, \{B, D\}, \{C\}.$

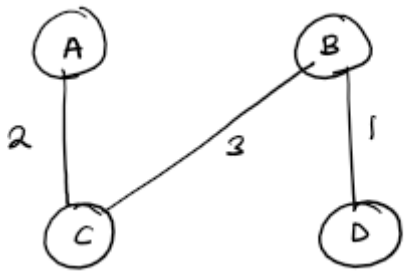
ii. Edge $A - C$:
$Current\ sets = \{A\}, \{B, D\}, \{C\}$
A belongs to set $\{A\}$ and C belongs to set $\{C\}$. And $\{A\} \neq \{C\}$. Therefore,
$a = a \cup \{(A, C)\}$
$\therefore a = \{(B, D), \{A, C\}\}$
$sets = (sets - \{A\} - \{C\}) \cup \{A, C\}$
$\therefore sets = \{A, C\}, \{B, D\}.$

iii. Edge $B - C$:
$Current\ sets = \{A, C\}, \{B, D\}$
B belongs to set $\{B, D\}$ and C belongs to set $\{A, C\}$. And $\{B, D\} \neq \{A, C\}$. Therefore,
$a = a \cup \{(B, C)\}$
$\therefore a = \{(B, D), \{A, C\}, (B, C)\}$
$sets = (sets - \{A, C\} - \{B, D\}) \cup \{A, C, B, D\}$
$\therefore sets = \{A, B, C, D\}.$

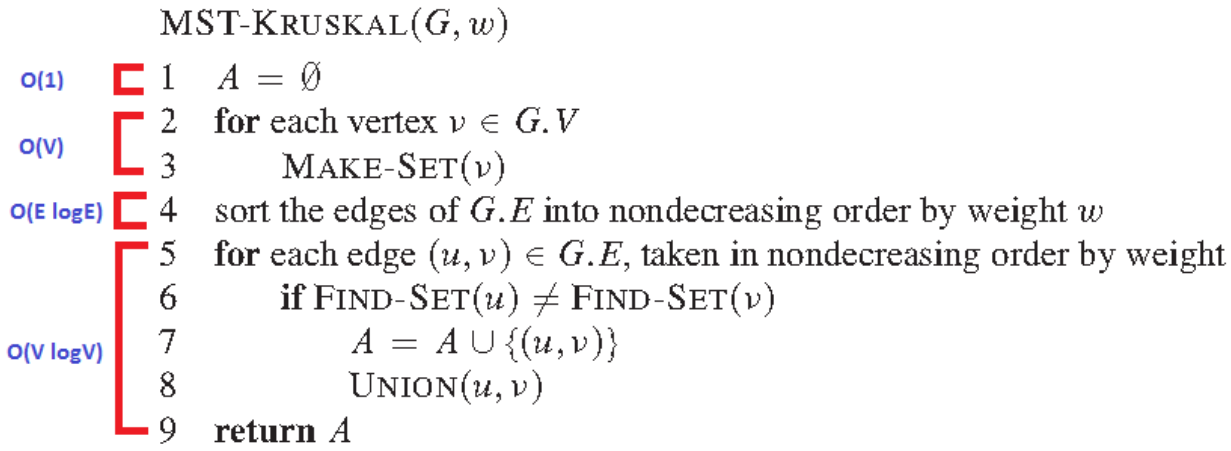
There are no more sets to be merged, therefore we stop.

There are two stopping conditions i.e. if there is a single set left in $sets$ or if there are no more edges left. The latter occurs when the graph is a disconnected graph.

The minimum-cost spanning tree obtained from above is as follows:



Time complexity:
In worst case scenario, the given graph can be a connected graph where the total number of edges are n^2 where n are the number of nodes in the given graph. The time complexity is as follows,



(Reference: https://stackoverflow.com/questions/20432801/time-complexity-of-the-kruskal-algorithm)

$T(n) = O(m.log(m) + n.log(n))$

As mentioned in the beginning, in worst case scenario $m = n^2$.

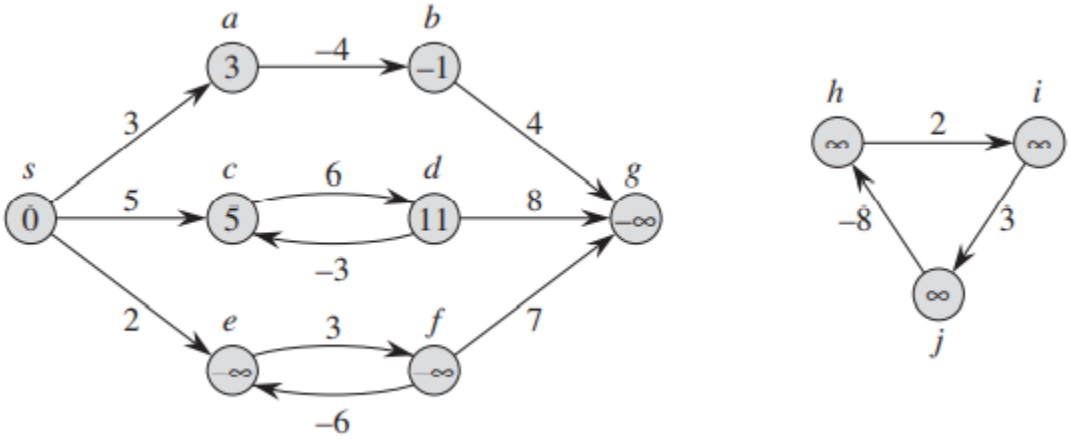
$T(n) = O(m.log(n^2) + n.log(n)) = O(2m.log(n) + n.log(n))$

$= O(m.log(n) + n.log(n)) = O(m.log(n))$ ($\because m > n$ i.e. number of edges > number of nodes in complete graph)

Single Source Shortest Path (SSSP) - Dijkstra's algorithm and Bellman Ford algorithm.

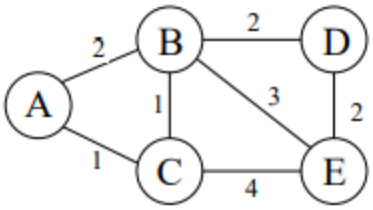
Shortest path/ Single Source Shortest Path (SSSP): In this lecture, we are considering that we have only one source in the given graph $G = (V, E)$ where source vertex/ node is s such that $s \in V$.

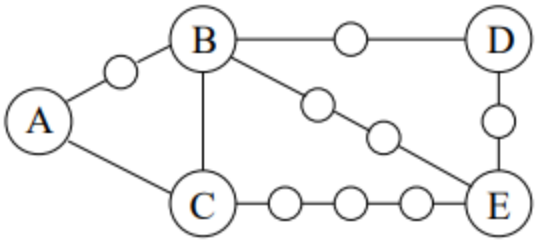
Negative weight cycle: Consider the following graph,



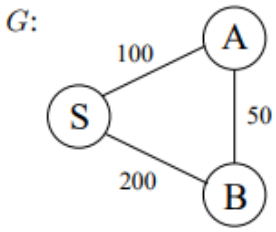
The distance between s and a is 3 as there is only one possible path to reach a from s. But it is not true for node c and node e. The possible paths between node s and node c are $s \rightarrow c$, $s \rightarrow c \rightarrow d \rightarrow c$, $s \rightarrow c \rightarrow d \rightarrow c \rightarrow d \rightarrow c$ and so on. Therefore the possible distances between node s and c are 5, (5 + 6 - 3) = 8, (5 + 6 - 3 + 6 - 3) = 11 and so on. As this cycle has positive weight we can easily find that the shortest path is $s \rightarrow c$ with a weight of 5.
But this is not true for the path from node s to e. The possible paths between node s and node e are $s \rightarrow e$, $s \rightarrow e \rightarrow f \rightarrow e$, $s \rightarrow e \rightarrow f \rightarrow e \rightarrow f \rightarrow e$ and so on. Therefore the possible distances between node s and e are 2, (2 + 3 - 6) = -1, (2 + 3 - 6 + 3 - 6) = -4 and so on. This weight keeps on decreasing as the path moves around this cycle. Such a cycle is called the negative weight cycle. Negative cycle will eventually cause the distance between nodes s and e to reduce to $-\infty$. And due to this the distance between all the node on the path $s \rightarrow e \rightarrow f \rightarrow g$ will reduce to $-\infty$.
Due to this reason, if there is a negative cycle in the graph we can not find the shortest path from a single source. There are different ways to remove negative cycles from the graph or at least detect that there is a negative cycle in the graph.
Algorithms such as Dijkstra's algorithm do not allow negative weights whereas Bellman Ford allows negative weights.

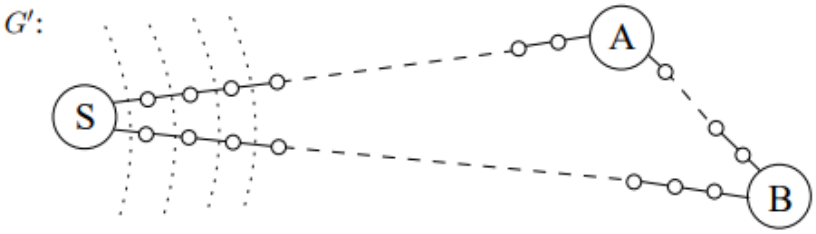
Can we use BFS to search the minimum cost path? The answer is yes. We can use BFS to search min cost path but only if the distance between every node is of unit length. But it is not necessary that the weight of all paths in the graph will be of unit length.
So can we convert the weighted paths in the graph to unit length? Yes, we can do that by splitting the path and adding additional nodes in the path as follows,





But the issue with this is the time complexity to find shortest path increases with the increased number of nodes. If the weight of paths are small then the number of nodes added will be less but if they weight is 100 then 98 nodes will be added i.e. for $path\ weight = x$ the number of nodes added will be $(x - 2)$.





The time complexity of this modified graph will be $O((\sum_{e=1}^{n} w_e)^2)$ for adjacency list.

Dijkstra's algorithm: Dijkstra's algorithm is used to find the shortest path from a single source. Following is a naive approach for implementing Dijkstra's algorithm.

Pseudocode:

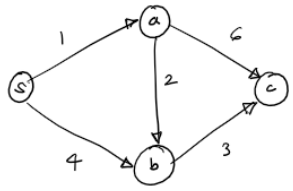
```
Dijkstra_naive(G, s)
    X = {s}
    len(s) = 0
    len(v) = ∞
    while (v, w) such that v ∈ X and w ∈ X
        select (v*, w*) such that it minimizes (len(v) + l_vw)
        add w* to X
        len(w*) = len(v*) + l_v*w*
```

The time complexity of the above approach is $O(n.m)$ where n is the number of vertices/ nodes and m is the number of edges in the graph. The time complexity can be further improved by using min heap (priority queue) for finding the (v, w) pair that minimizes the cost.

Faster approach for Dijkstra's algorithm:

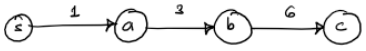
```
Dijkstra_faster_approach(G, s)
    X = { } or ϕ
    H <= min heap
    key(s) = 0
    for every v ≠ s
        key(v) = ∞
    H = build_min_heap(G.V)   // with key(v) as key for min heap where v ∈ V
    while H is not empty
        w* = extract_min(H)
        add w* to X
        len(w*) = key(w)
        for every edge (w*, y)
            key(y) = min(key(y), len(w*) + cost(w*, y))
```

The time complexity of the above approach is $T(n) = O(n.log(n) + m.log(n)) = O(m.log(n))$.

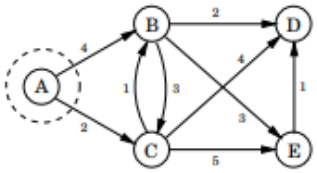


X	s	a	b	c	edges
{ }	0	∞	∞	∞	
{s}	0	1/s	4/s	∞	len(a) < len(b) ∴ add node a to X
{s, a}			3/a	7/a	len(b) < len(c) ∴ add node b to X
{s, a, b}				6/b	only c left ∴ add c to X
{s, a, b, c}					

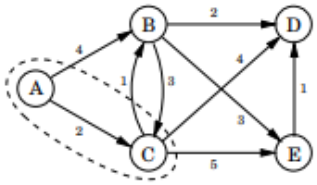
Shortest path found using Dijkstra's algorithm from source node s:



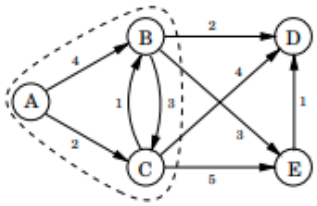
```
procedure dijkstra(G, l, s)
Input:    Graph G = (V, E), directed or undirected;
          positive edge lengths {l_e : e ∈ E}; vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
    dist(u) = ∞
    prev(u) = nil
dist(s) = 0

H = makequeue(V)   (using dist-values as keys)
while H is not empty:
    u = deletemin(H)
    for all edges (u, v) ∈ E:
        if dist(v) > dist(u) + l(u, v):
            dist(v) = dist(u) + l(u, v)
            prev(v) = u
            decreasekey(H, v)
```



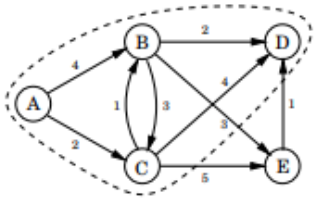
A: 0	D: ∞
B: 4	E: ∞
C: 2	



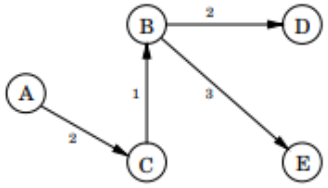
A: 0	D: 6
B: 3	E: 7
C: 2	



A: 0	D: 5
B: 3	E: 6
C: 2	

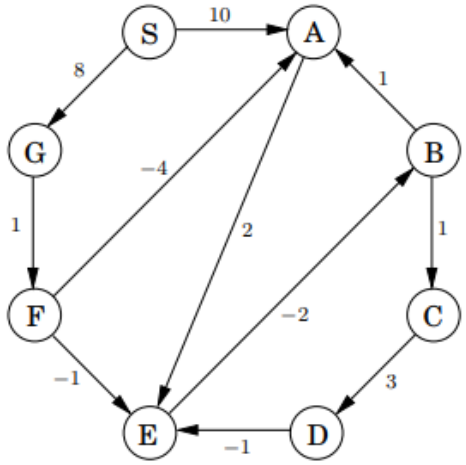


A: 0	D: 5
B: 3	E: 6
C: 2	



Bellman Ford algorithm:

```
procedure update((u, v) ∈ E)
dist(v) = min{dist(v), dist(u) + l(u, v)}

procedure shortest-paths(G, l, s)
Input:    Directed graph G = (V, E);
          edge lengths {l_e : e ∈ E} with no negative cycles;
          vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil

dist(s) = 0
repeat |V| − 1 times:
   for all e ∈ E:
      update(e)
```



	Iteration							
Node	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

The time complexity of the Bellman ford algorithm is $O(n.m)$ where n is the number of nodes and m is the number of edges in the given graph.

.

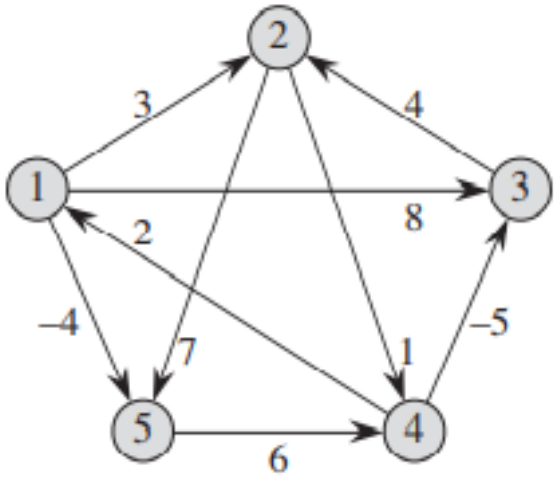
All Pairs Shortest Path - Floyd Warshall algorithm and Johnson's algorithm.

All-pairs shortest path: The single source shortest path algorithms such as Dijkstra's algorithm and Bellman-Ford can give us the shortest path from a single source to multiple sources in $O(n.m)$ time complexity. This can then be repeated for all the source nodes available to find the shortest path from every source node to all the other nodes in the network and consider the lowest weight. This will require a time complexity of $O(n.m) * O(n) = O(n^2 m)$ or $O(m.log(n)).O(n) = O(m.n.log(n))$.

Floyd Warshall algorithm: The Floyd Warshall algorithm is as given below. It generates n number of matrices of size $n \times n$ where n is the number of vertices/ nodes in the given graph.

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D^(0) = W
3  for k = 1 to n
4      let D^(k) = (d_ij^(k)) be a new n × n matrix
5      for i = 1 to n
6          for j = 1 to n
7              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
8  return D^(n)
```

Given graph:



$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

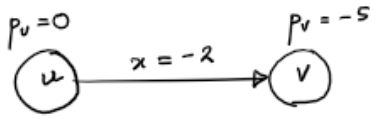
The time complexity of this algorithm is $O(n^3)$.
https://www.cs.usfca.edu/~galles/visualization/Floyd.html

Johnson's algorithm: In Johnson's algorithm, we consider that the given graph is directional and contains negative weights over the edges. We convert this graph to a graph with non-negative weight using the reweighting technique as follows.

1. Add a new source node and connect it to all the nodes with link weight = 0.
2. Then compute all the shortest distances to every node using Bellman-Ford algorithm.
3. If there are two nodes u and v. And the edge (u, v) has original weight x and the shortest path distance calculated in step 2 for node u is p_u and that of node v is p_v then the reweighting of the edge can be calculated as follows,

$$weight = x + p_u - p_v$$

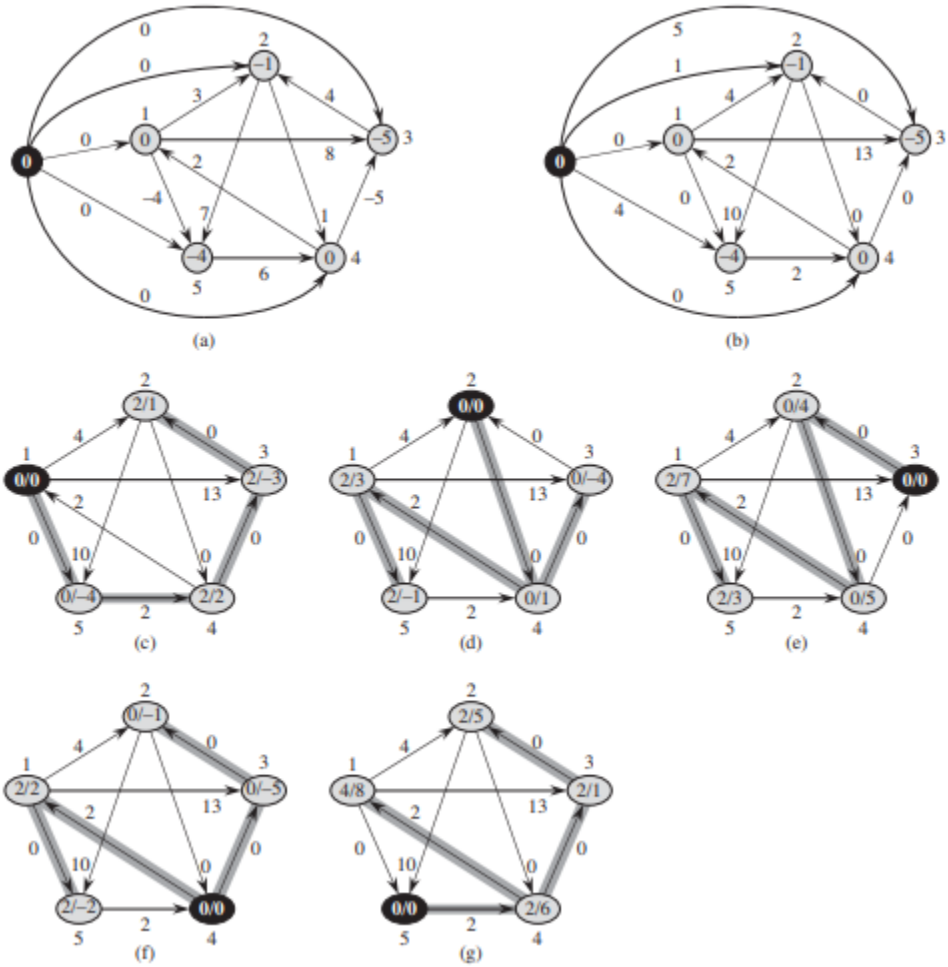
Example: If the edge calculations after step 2 are as follows,



$\therefore$ weight $= x + p_u - p_v$
$= -2 + (0) - (-5)$
$= -2 + 5$
$\therefore$ weight $= 3$ //

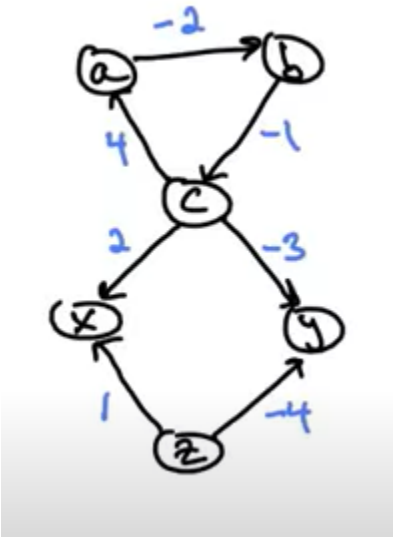
JOHNSON(G, w)

```
1   compute G', where G'.V = G.V ∪ {s},
        G'.E = G.E ∪ {(s, v) : v ∈ G.V}, and
        w(s, v) = 0 for all v ∈ G.V
2   if BELLMAN-FORD(G', w, s) == FALSE
3       print "the input graph contains a negative-weight cycle"
4   else for each vertex v ∈ G'.V
5           set h(v) to the value of δ(s, v)
                computed by the Bellman-Ford algorithm
6       for each edge (u, v) ∈ G'.E
7           ŵ(u, v) = w(u, v) + h(u) − h(v)
8       let D = (d_uv) be a new n × n matrix
9       for each vertex u ∈ G.V
10          run DIJKSTRA(G, ŵ, u) to compute δ̂(u, v) for all v ∈ G.V
11          for each vertex v ∈ G.V
12              d_uv = δ̂(u, v) + h(v) − h(u)
13      return D
```

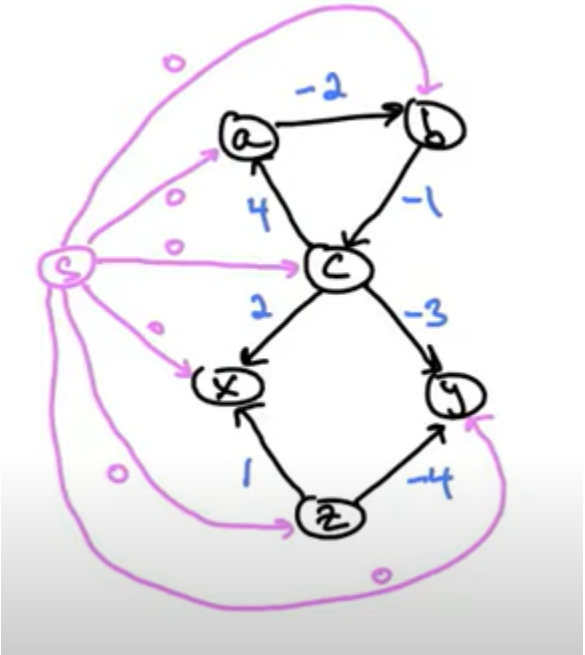


The reason that Johnson's algorithm is better for sparse graphs is that its time complexity depends on the number of edges in the graph, while Floyd-Warshall's does not. Johnson's algorithm runs in $O(n^2.log(n) + n.m)$ time. So, if the number of edges is small (i.e. the graph is sparse), it will run faster than the $O(n^3)$ runtime of Floyd-Warshall.

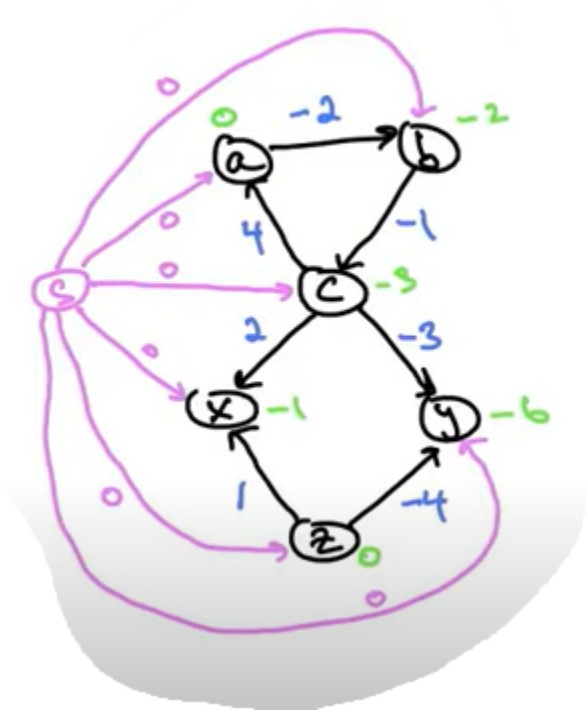
Example from youtube video: https://www.youtube.com/watch?v=xc2ua8sQAoE
Given graph:



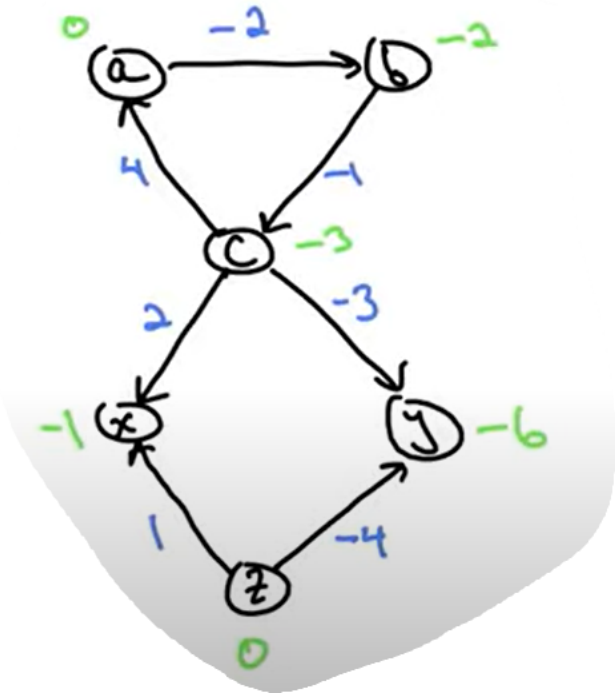
Add source node to every node with link weight = 0



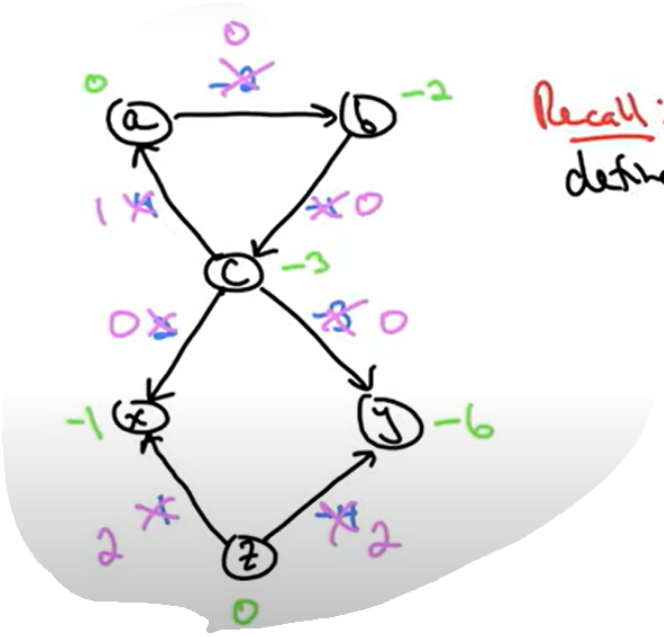
Calculate the shortest path distance to every node using Bellman-Ford algorithm.



Graph after removing the newly added source node.

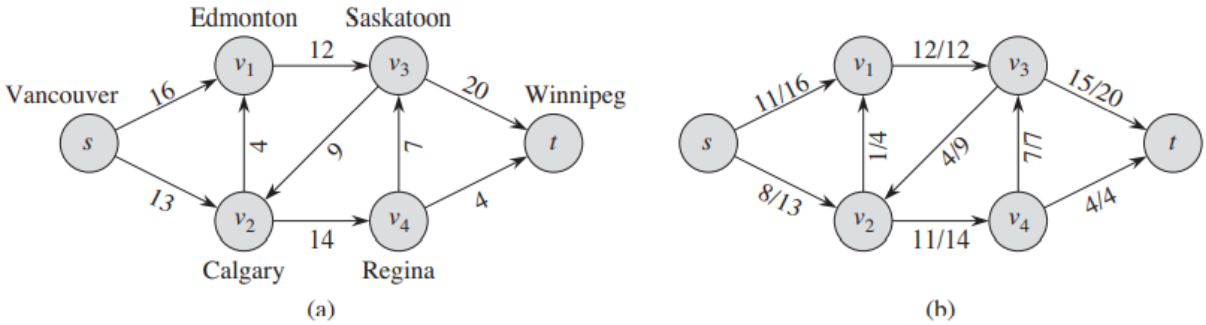


Calculate the reweighted edge weight using the given formula.



Flow networks -
Maximum flow problem,
Residual graphs,
Ford-Fulkerson algorithm,
Edmond Karp algorithm,
Max-flow, min-cut theorem,
Multi-source, multi-sink and
Maximum bipartite matching.

Flow network: Flow network $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a non-negative capacity $c(u, v) \geq 0$ and consists of at least one source node s (node without incoming edges) and one sink node t (node without outgoing edges).



Flow through an edge consumes the capacity of the edge. For example, if there is an edge (u, v) with capacity $c(u, v) = 4$ and there is flow through that edge of 2 units then the residual capacity of the link is 4 - 2 = 2. The capacity function is generally denoted as $c(u, v)$ and flow as $f(u, v)$. Flow is a real valued function $f: V \times V \rightarrow \Re$ that satisfies the following two properties:

1. Capacity constraint: For all $u, v \in V$, we require $0 \leq f(u, v) \leq c(u, v)$.
2. Flow conservation: For all $u \in (V - \{s, t\})$, we require

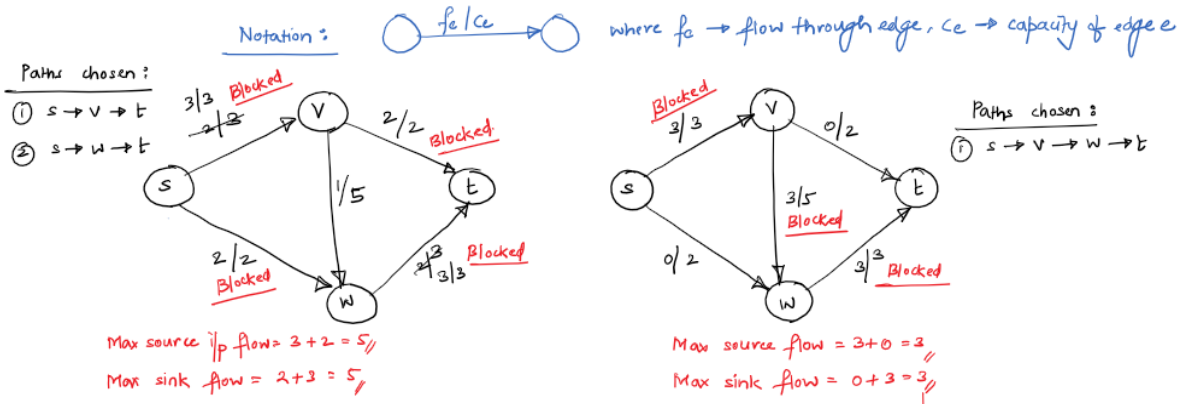
$$\sum_{v \in V} f(u, v) = \sum_{w \in V} f(w, u)$$

That is, for all nodes u except s and t, the amount of flow entering u equals the amount leaving.

Maximum flow problem: In the maximum-flow problem we are given a flow network G with source and sink and the output is maximum flow permissible in the network.
Input: Graph G_f with source node s and sink node t.
Output: Maximum output flow.

Basic approach to find the maximum flow of the flow network is to find different paths in the graph such that there are no blocked or completely consumed edges in the selected path. And then find the maximum flow through the links from the source node or the links to the sink node.

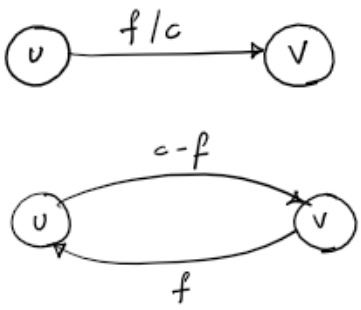
Here the order in which the paths are chosen is important. For example check the flow network in the following image.



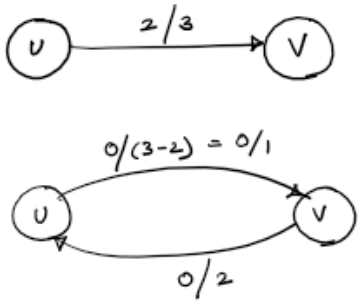
We can see that the maximum flow obtained in the first graph is 5 and that obtained in the second graph is 3 because the paths chosen are different here.
Note that here the path chosen can be in any order. We can use either BFS or DFS for doing so.

To overcome this issue we use residual graphs. This is used by the Ford-Fulkerson algorithm as well to find the maximum flow of flow graphs.

Residual graphs: This is the additional graph G_f that we maintain along with the original graph G. If we have a flow f from node u to node v in graph G then we update that edge as follows:

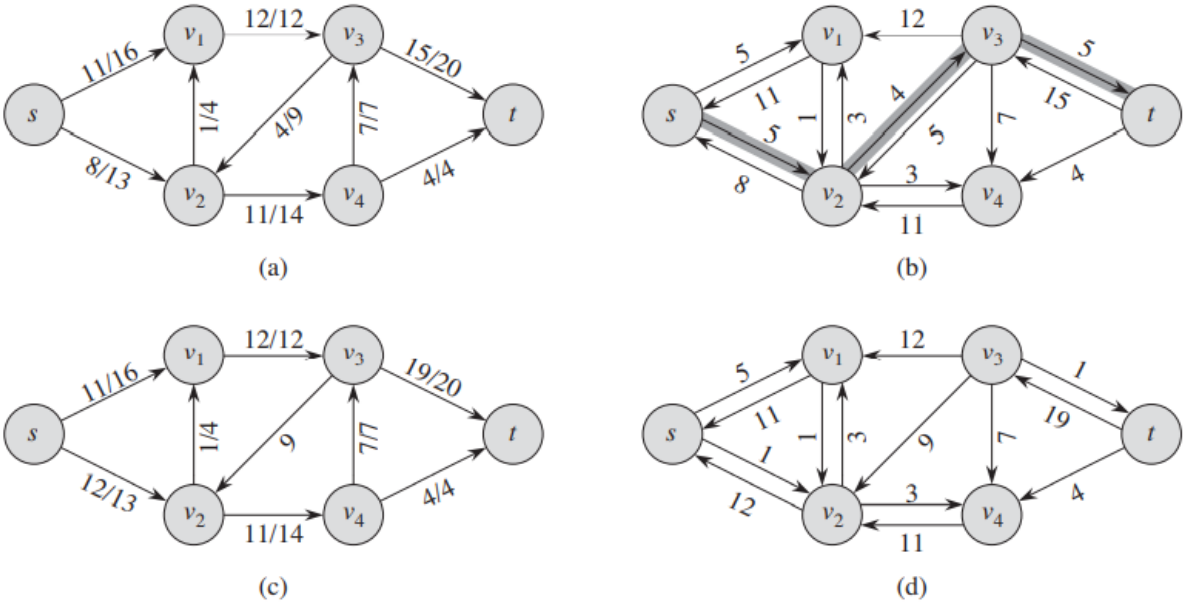


For example:



The residual capacity c_f can be formally defined as,

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E\ , \\ f(v,u) & \text{if } (v,u) \in E\ , \\ 0 & \text{otherwise}\ . \end{cases}$$



Ford-Fulkerson algorithm:

FORD-FULKERSON-METHOD(G, s, t)

1 initialize flow f to 0
2 **while** there exists an augmenting path p in the residual network G_f
3 augment flow f along p
4 **return** f

The basic Ford-Fulkerson algorithm can be summarized as follows:

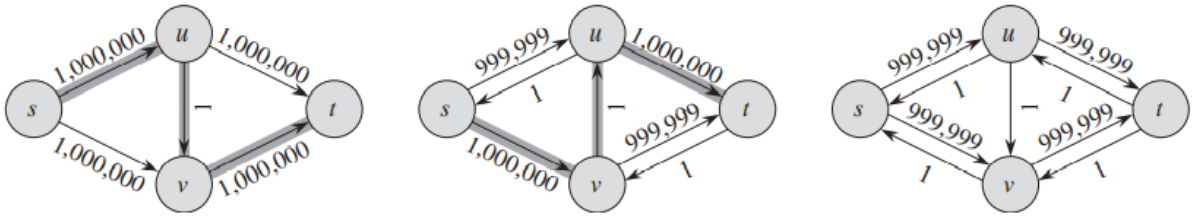
1. Find an augmenting path. (Time complexity: $O(m)$)
2. Compute the bottleneck capacity of the path
3. Augment each edge and the total flow and repeat from step 1. (Time complexity: $O(f)$)

The total time complexity can be given as, $T(n) = O(m.f)$.

As per Ford-Fulkerson algorithm, for finding the augmenting path we can either use DFS or BFS.

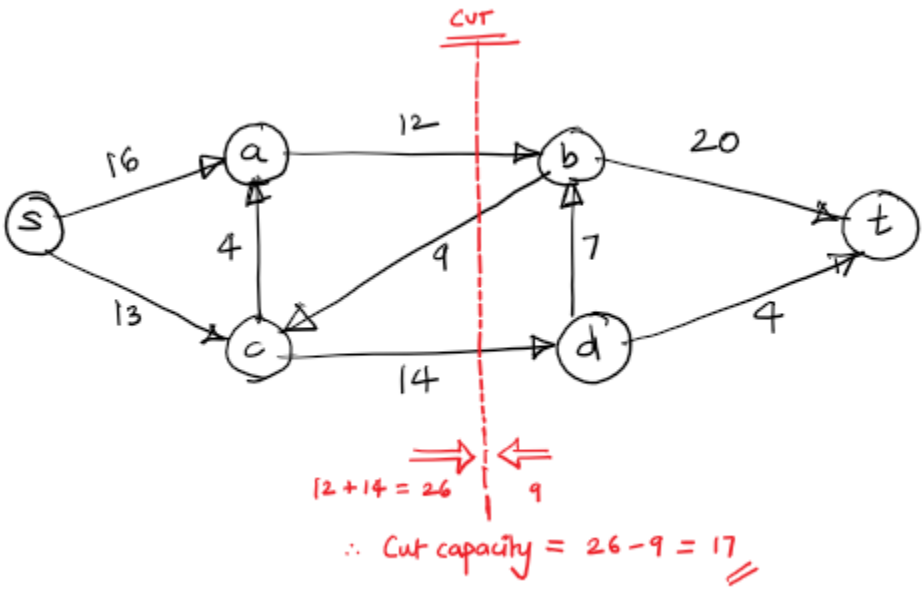
```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ G.E
2      (u, v).f = 0
3  while there exists a path p from s to t in the residual network G_f
4      c_f(p) = min {c_f(u, v) : (u, v) is in p}
5      for each edge (u, v) in p
6          if (u, v) ∈ E
7              (u, v).f = (u, v).f + c_f(p)
8          else (v, u).f = (v, u).f − c_f(p)
```

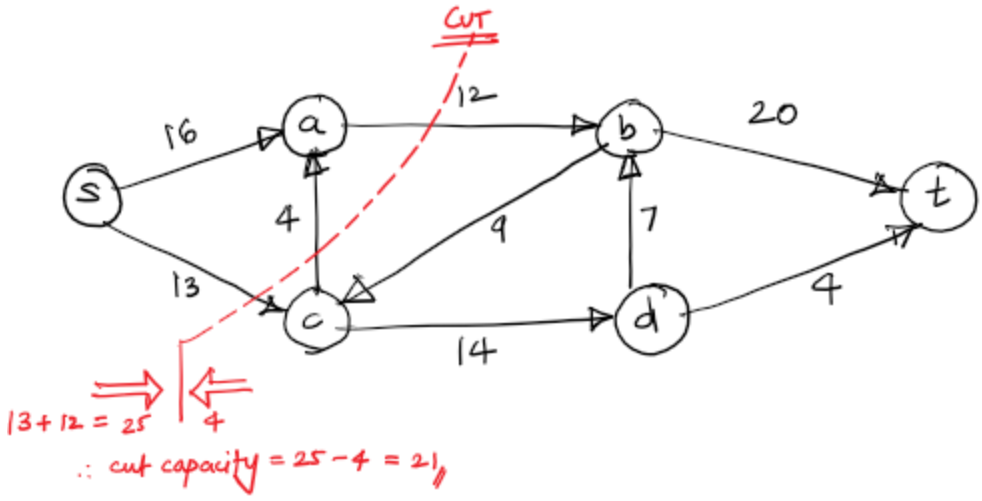
Edmond-Karp algorithm: We can improve the bound on FORD-FULKERSON by finding the augmenting path p in line 3 with a breadth-first search. That is, we choose the augmenting path as a shortest path from s to t in the residual network, where each edge has unit distance (weight). We call the Ford-Fulkerson method to implement the Edmonds-Karp algorithm.



The time complexity of the Edmond-Karp algorithm is $O(m^2.n)$.

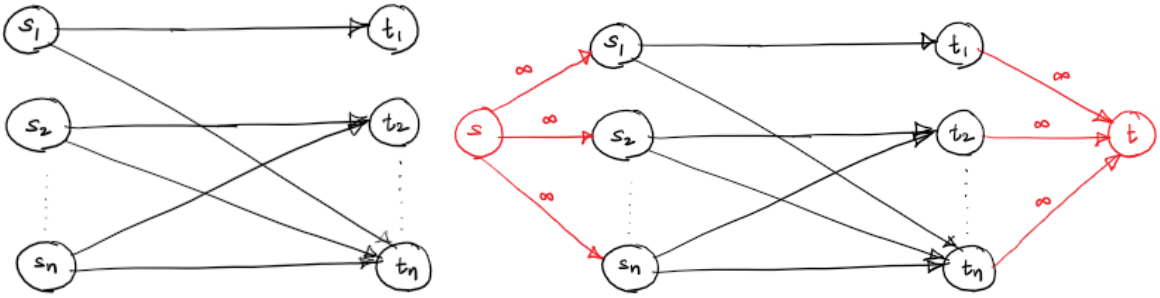
Cut: If a network is cut into two parts then we can calculate the cut capacity of the cut. Lets consider the following example.



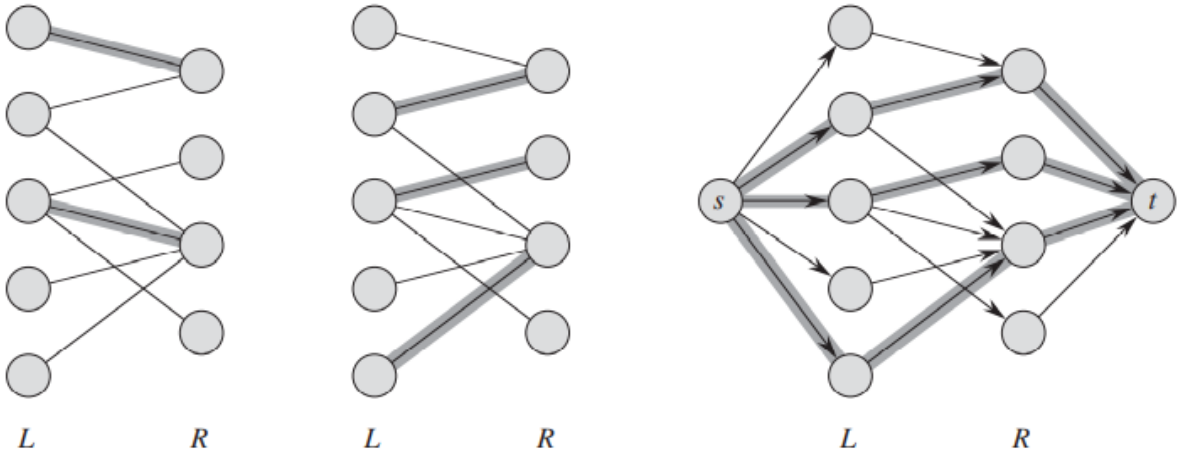


Max-flow, min-cut theorem: This theorem says that the size of the maximum flow in a network equals the capacity of the smallest cut. Or in other words, the value of a maximum flow is in fact equal to the capacity of a minimum cut.

Multi-source, multi-sink: In this problem, we have multiple sources and multiple sinks. In this scenario, we can easily solve the problem by adding a single source and single sink. These newly added source and sink can then be connected as follows.



Maximum bipartite matching: For maximum bipartite matching, we can consider the nodes on the left side as multiple sources and the ones on the right as multiple sinks. As the links are undirected, we can consider their direction from the source nodes to sink nodes with weight/ capacity as 1. Finally add a single source and single sink and connect them as mentioned in the previous section but the weights of the links should be 1. And in the end just run Ford-Fulkerson or Edmond-Karp algorithm to find the maximum bipartite matching.



The time complexity of this algorithm is $O(n.m)$.

String matching - Knuth-Morris-Pratt approach and Rabin-Karp matcher.

String matching: In string matching, we are given 2 strings, one is text string and another one is pattern, and we try to find the location at which the pattern string is present in the text string (if text string contains pattern string).
Input: $Text\ string\ T,\ Pattern\ string\ P$
Output: index of 1st match.

For example:
Input: $T = abcabaacabac.\ P = abaa$
Output: $s = 3$ (assuming that the index starts at 0)

Naive approach: In this approach, we iterate over the text string T of length n from 0 to $(n - m)$ and check if we have a match with pattern string P of length m.

Pseudocode:

```
NAIVE-STRING-MATCHER(T, P)
1  n = T.length
2  m = P.length
3  for s = 0 to n − m
4       if P[1..m] == T[s + 1..s + m]
5            print "Pattern occurs with shift" s
```

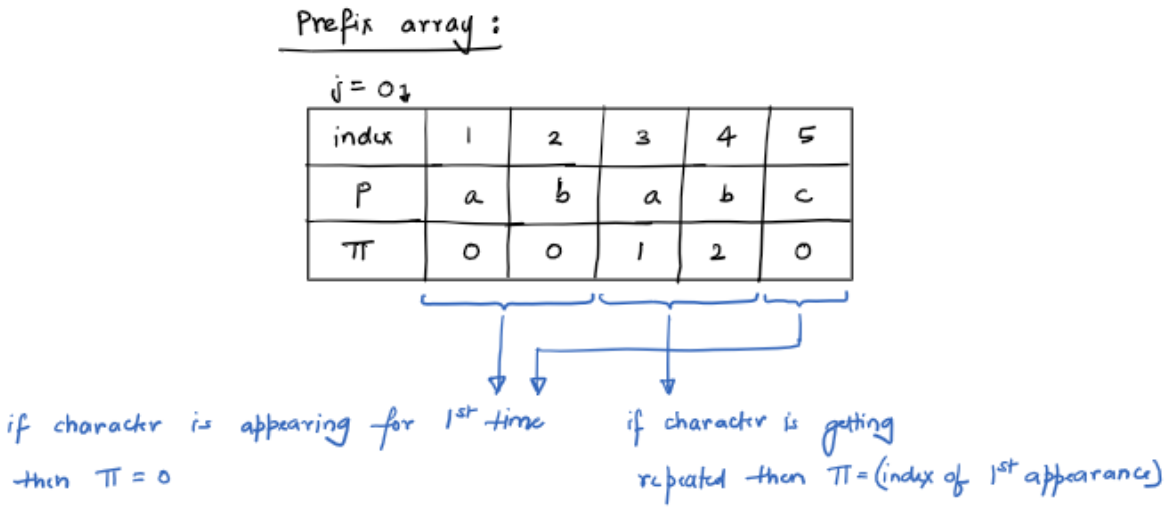
This approach is easy to code but the time complexity is $O(n.m)$. Note that the if statement on line 4 has a time complexity of $O(m)$ as it involves comparison of m elements of pattern string.

Knuth-Morris-Pratt approach: In this approach, we do some preprocessing before we start scanning the text string for pattern string. The preprocessing is done in order to take advantage of repetitions of characters present in pattern string.
For example, consider the pattern string as $P = ababc$ and text string as $T = abababcabaa$.

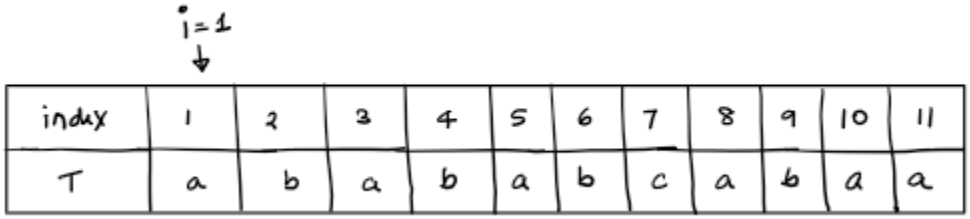
Now here when we compare the pattern string with text string, the first 4 characters will match and at the 5th character there will be a mismatch. We can see that in the pattern string ab is repeating twice. So when there is a mismatch, instead of checking the pattern string from the first character, we can just go back to the 3rd character in the pattern string since ab is already matching in the text string.

For doing so, we create a prefix array which contains a row π which indicates the index at which the pointer on pattern string should return to.

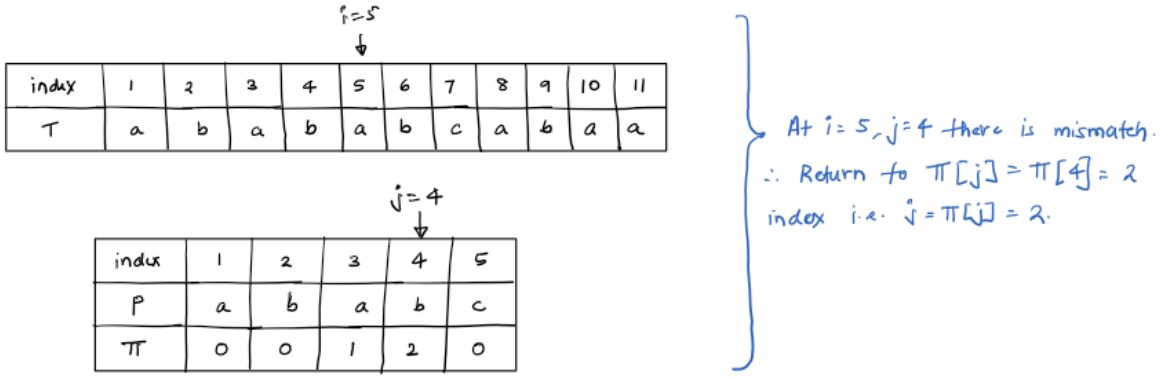
Creating a prefix array:



Here we are assuming that the array indexes start from 1. The pointer j is initialized on Prefix array shown above and $j = 0$ initially. The pointer i is initialized to 1 initially on the text string array.
The text string array is as follows,



We compare $P[j + 1]$ with $T[i]$. And if there is a match then we increment both i and j. If there is a mismatch then the j pointer is return to π value i.e. $j = \pi[j]$. Following is an example of the mismatch condition.



As you can see, j returns to index 2 and thus we don't need to compare ab again since ab was already matched in the text string.

Pseudocode:

```
KMP-MATCHER(T, P)
 1  n = T.length
 2  m = P.length
 3  π = COMPUTE-PREFIX-FUNCTION(P)
 4  q = 0                                  // number of characters matched
 5  for i = 1 to n                         // scan the text from left to right
 6      while q > 0 and P[q + 1] ≠ T[i]
 7          q = π[q]                       // next character does not match
 8      if P[q + 1] == T[i]
 9          q = q + 1                      // next character matches
10      if q == m                          // is all of P matched?
11          print "Pattern occurs with shift" i − m
12          q = π[q]                       // look for the next match
```

The time complexities of $O(n)$ is achieved using the KMP approach.

Rabin-Karp matcher: In this approach, we use hash values for characters in the text and pattern strings. The hash values are generally unique single digit values.
For example: $T = ababcabcababd$, $P = ababd$. And we assign following hash values to the characters a, b, c and d.

Character	Hash value
a	0
b	1
c	2
d	3

So the first step is to convert all the characters in the text and pattern string arrays to values as per hash values. Therefore the text and pattern string arrays will be,
$T = 0101201201013$ and $P = 01013$.

One approach can be find the sum of pattern string and then calculate the sum of numbers in text string as sets of 5 i.e. $sum(T[1\ to\ 5])$ and compare it with $sum(P[1\ to\ 5])$, check if sums are same, if not then go to next five i.e. $sum(T[2\ to\ 6])$ and compare again with $sum(P[1\ to\ 5])$.

Now we know that $sum(P[1\ to\ 5]) = 0 + 1 + 0 + 1 + 3 = 5$. And this matches with $sum(T[5\ to\ 9])$ but the string is not matching. Such a hit is called a spurious hit. Due to this, even after sum matches we need to match all characters of the pattern string with the text string which increases the time complexity.

Another approach is to convert the pattern string array hash values to decimal numbers (or any other radix). And similarly convert every set of the text array to decimal numbers. In this case the comparison will just involve comparison of 2 decimal numbers.
$to\_number(P[1\ to\ 5]) = 1013_{10}\ and\ to\_number(T[1\ to\ 5]) = 1012_{10}$.
This conversion is done using Horner's rule. It is as follows,
$p = P[m] + 10(P[m-1] + 10(P[m-2] + \ldots + 10(P[2] + 10(P[1]) \ldots)))$

The issue with this approach is that, if the length of the pattern string array is too long (comparable to n) then the comparison of numbers will be computationally expensive.
To overcome this we reduce the pattern string number using mod function. But this will again cause the wrong comparison hits i.e. spurious hits.

```
RABIN-KARP-MATCHER(T, P, d, q)
 1  n = T.length
 2  m = P.length
 3  h = d^(m−1) mod q
 4  p = 0
 5  t_0 = 0
 6  for i = 1 to m                    // preprocessing
 7      p = (dp + P[i]) mod q
 8      t_0 = (dt_0 + T[i]) mod q
 9  for s = 0 to n − m                // matching
10      if p == t_s
11          if P[1..m] == T[s + 1..s + m]
12              print "Pattern occurs with shift" s
13      if s < n − m
14          t_(s+1) = (d(t_s − T[s + 1]h) + T[s + m + 1]) mod q
```

The approximate time complexity of Rabin-Karp-Matcher function is $O(n)$.