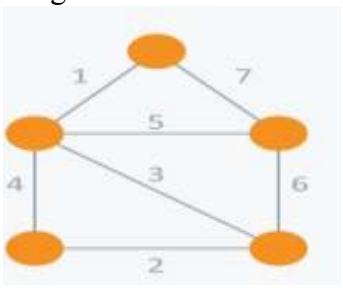
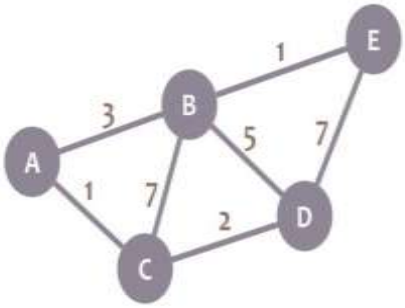
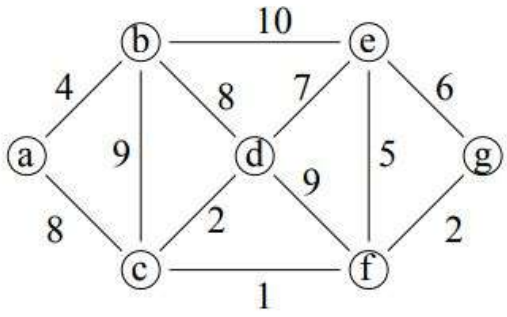
CMR INSTITUTE OF TECHNOLOGY	USN										

Second Internal Test

Sub:	DESIGN AND ANALYSIS OF ALGORITHMS							Code:	17CS43
Date:	16/04/2019	Duration:	90 mins	Max Marks:	50	Sem:	IV	Branch:	ISE

Answer Any **FIVE FULL** Questions

		Marks	OBE CO	OBE RBT
1 (a)	Write the algorithm for Quick Sort. Explain with example. Derive best case, worst case, average case time efficiency of the algorithm.	[10]	CO4, CO3	L2
2 (a)	Construct a Huffman code for the following data: Symbol: A, B, C, D, _ Frequency: 0.4, 0.1, 0.2, 0.15, 0.15 Encode ABACABAD using the code. Decode 100010111001010	[7]	CO3	L3
(b)	Explain the advantages and disadvantages of divide and Conquer Strategy	[3]	CO3	L2
3 (a)	Define MST. Explain KRUSKAL algorithm and apply it for the following graph to get MST. Show the intermediate steps. 	[10]	CO4	L3
4 (a)	Sort the following lists by heap sort by using the array representation heaps `4, 10, 3, 5, 1` (in increasing order).Analyze its complexity	[10]	C04, CO3	L2
5(a)	Explain Dijkstra's algorithm and apply this algorithm to find single source shortest path algorithm for the following graph 	[10]	CO4	L3
6.(a)	Define coin change problem. Write the greedy strategy for getting Optimal solution. If coins available are of values { 2, 5, 3, 6 }, find the least denominations for a) 55 b)77	[5]	CO4	L3
(b)	What is Job Scheduling Problem with deadline, Find	[5]	CO4	L3

	solution generated by Job Scheduling Problem with deadline where n=4, Deadlines are [D1,D2,D3,D4]=[4,1,1,1] and profits [P1,P2,P3,P4]=[20,10,40,30].			
7(a)	What are the different types of Decrease and Conquer Approaches? Explain in detail.	[5]	CO3	L2
(b)	What is Knapsack Problem? Obtain the solution for Knapsack problem For n=3, Knapsack Capacity, M = 50 and Items as (value, weight) pairs arr[] = {{60, 10}, {100, 20}, {120, 30}}	[5]	CO4	L3
8.(a)	Explain Stassen's Matrix multiplication with its analysis.	[5]	CO3	L3
(b)	Apply Prim's Algorithm for following graph and find Minimum spanning Tree 	[5]	CO4	L3

1. Explain Quick sort with example and analysis

Quicksort is the other important sorting algorithm that is based on the divide-and-conquer approach. Unlike mergesort, which divides its input elements according to their position in the array, quicksort divides (or partitions) them according to their value.

A partition is an arrangement of the array's elements so that all the elements to the left of some element A[s] are less than or equal to A[s], and all the elements to the right of A[s] are greater than or equal to it:

$$\underbrace{A[0] \ldots A[s-1]}_{\text{all are} \le A[s]} \quad A[s] \quad \underbrace{A[s+1] \ldots A[n-1]}_{\text{all are} \ge A[s]}$$

Obviously, after a partition is achieved, A[s] will be in its final position in the sorted array, and we can continue sorting the two subarrays to the left and to the right of A[s] independently (e.g., by the same method).

In quick sort, the entire work happens in the division stage, with no work required to combine the solutions to the subproblems.

ALGORITHM *Quicksort*$(A[l..r])$
//Sorts a subarray by quicksort
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices l and r
//Output: Subarray $A[l..r]$ sorted in nondecreasing order
if $l < r$
$s \leftarrow$ *Partition*$(A[l..r])$ //s is a split position
Quicksort$(A[l..s-1])$
Quicksort$(A[s+1..r])$

Partitioning

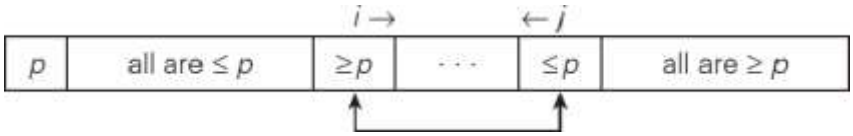
We start by selecting a pivot—an element with respect to whose value we are going to divide the subarray. There are several different strategies for selecting a pivot. We use the sophisticated method suggested by C.A.R. Hoare, the prominent British computer scientist who invented quicksort.

Select the subarray's first element: p = A[l].Now scan the subarray from both ends, comparing the subarray's elements to the pivot.

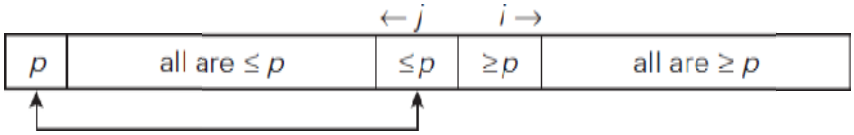
- The left-to-right scan, denoted below by index pointer i, starts with the second element. Since we want elements smaller than the pivot to be in the left part of the subarray, this scan skips over elements that are smaller than the pivot and stops upon encountering the first element greater than or equal to the pivot.
- The right-to-left scan, denoted below by index pointer j, starts with the last element of the subarray. Since we want elements larger than the pivot to be in the right part of the

subarray, this scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot.
After both scans stop, three situations may arise, depending on whether or not the scanning indices have crossed.

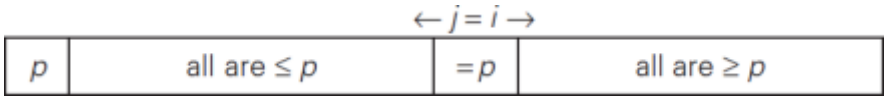
1. If scanning indices i and j have not crossed, i.e., i< j, we simply exchange A[i] and A[j] and resume the scans by incrementing I and decrementing j, respectively:



2. If the scanning indices have crossed over, i.e., i> j, we will have partitioned the subarray after exchanging the pivot with A[j]:



3. If the scanning indices stop while pointing to the same element, i.e., i = j, the value they are pointing to must be equal to p. Thus, we have the subarray partitioned, with the split position s = i = j :



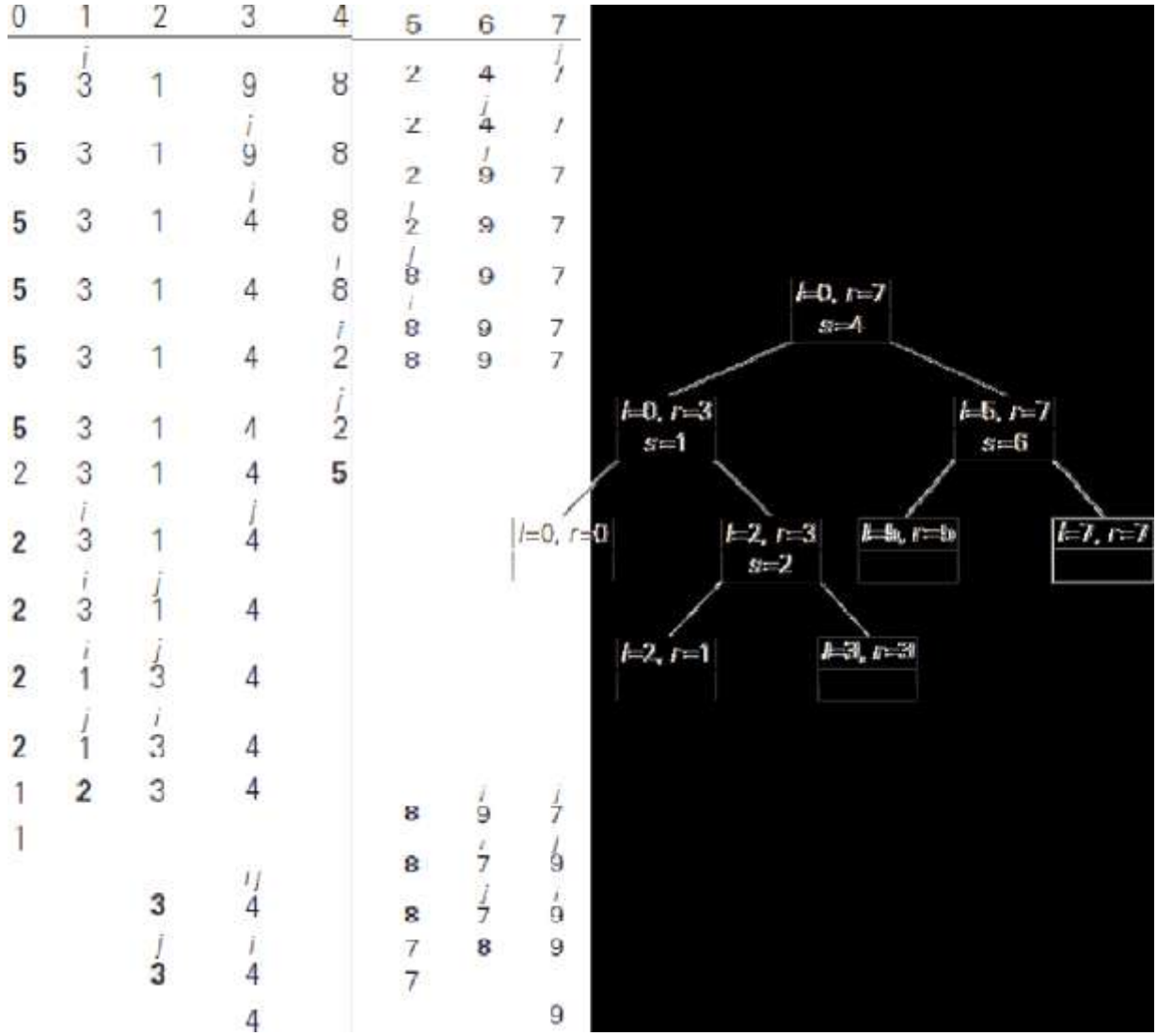
We can combine this with the case-2 by exchanging the pivot with A[j] whenever i≥j

ALGORITHM *HoarePartition*($A[l..r]$)
//Partitions a subarray by Hoare's algorithm, using the first element as a pivot
//Input: Subarray of array $A[0..n-1]$, defined by its left and right indices l and r ($l<r$)
//Output: Partition of $A[l..r]$, with the split position returned as this function's value

$p \leftarrow A[l]$
$i \leftarrow l$; $j \leftarrow r+1$
repeat
  repeat $i \leftarrow i+1$ **until** $A[i] \geq p$
  repeat $j \leftarrow j-1$ **until** $A[j] \leq p$
  swap($A[i]$, $A[j]$)
until $i \geq j$
swap($A[i]$, $A[j]$) //undo last swap when $i \geq j$
swap($A[l]$, $A[j]$)
return j

Note that index *i* can go out of the subarray's bounds in this pseudocode.

Example: Example of quicksort operation. (a) Array's transformations with pivots shown in bold. (b) Tree of recursive calls to Quicksort with input values land r of subarray bounds and split position s of a partition obtained.



Analysis

Best Case -Here the basic operation is key comparison. Number of key comparisons made before a partition is achieved is n + 1 if the scanning indices cross over and n if they coincide. If all the splits happen in the middle of corresponding subarrays, we will have the best case. The number of key comparisons in the best case satisfies the recurrence,

According to the Master Theorem, $C_{best}(n) \in \Theta(n \log_2 n)$; solving it exactly for $n = 2^k$ yields

$$C_{best}(n) = 2C_{best}(n/2) + n \quad \text{for } n > 1, \quad C_{best}(1) = 0.$$

$C_{best}(n) = n \log_2 n$.

Worst Case – In the worst case, all the splits will be skewed to the extreme: one of the two subarrays will be empty, and the size of the other will be just 1 less than the size of the subarray being partitioned. This unfortunate situation will happen, in particular, for increasing arrays. Indeed, if A[0..n − 1] is a strictly increasing array and we use A[0] as the

pivot, the left-to-right scan will stop on A[1] while the right-to-left scan will go all the way to reach A[0], indicating the split at position 0: So, after making n + 1 comparisons to get to this partition and exchanging the pivot A[0] with itself, the algorithm will be left with the strictly increasing array A[1..n − 1] to sort. This sorting of strictly increasing arrays of diminishing sizes will continue until the last one A[n−2.. n−1] has been processed. The total number of key comparisons made will be equal to

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2).$$

Average Case - Let $C_{avg}(n)$ be the average number of key comparisons made by quicksort on a randomly ordered array of size n. A partition can happen in any position s ($0 \le s \le n-1$) after n+1comparisons are made to achieve the partition. After the partition, the left and right subarrays will have s and n − 1− s elements, respectively. Assuming that the partition split can happen in each position s with the same probability 1/n, we get the following recurrence relation:

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1,$$

$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

Its solution, which is much trickier than the worst- and best-case analyses, turns out to be

$$C_{avg}(n) \approx 2n \ln n \approx 1.39n \log_2 n.$$

Thus, on the average, quicksort makes only 39% more comparisons than in the best case. Moreover, its innermost loop is so efficient that it usually runs faster than mergesort on randomly ordered arrays of nontrivial sizes. This certainly justifies the name given to the algorithm by its inventor.

Variations: Because of quicksort's importance, there have been persist nt efforts over the years to refine the basic algorithm. Among several improvements discovered by researchers are:

- Better pivot selection methods such as randomized quicksort that uses a random element or the median-of-three method that uses the median of the leftmost, rightmost, and the middle element of the array
- Switching to insertion sort on very small subarrays (between 5 and 15 elements for most computer systems) or not sorting small subarrays at all and finishing the algorithm with insertion sort applied to the entire nearly sorted array
- Modifications of partitioning algorithm such as the three-way partition into segments smaller than, equal to, and larger than the pivot

Limitations: 1. It is not stable. 2. It requires a stack to store parameters of subarrays that are yet to be sorted. 3. While Performance on randomly ordered arrays is known to be sensitive not only to the implementation details of the algorithm but also to both computer architecture and data type.

2.a

A 1 B 000 C 011 D 001 – 010

Code is

2.b Advantages and Disadvantages of Divide & Conquer

Advantages

- **Parallelism:** Divide and conquer algorithms tend to have a lot of inherent parallelism. Once the division phase is complete, the sub-problems are usually independent and can therefore be solved in parallel. This approach typically generates more enough concurrency to keep the machine busy and can be adapted for processor execution in multi-machines.

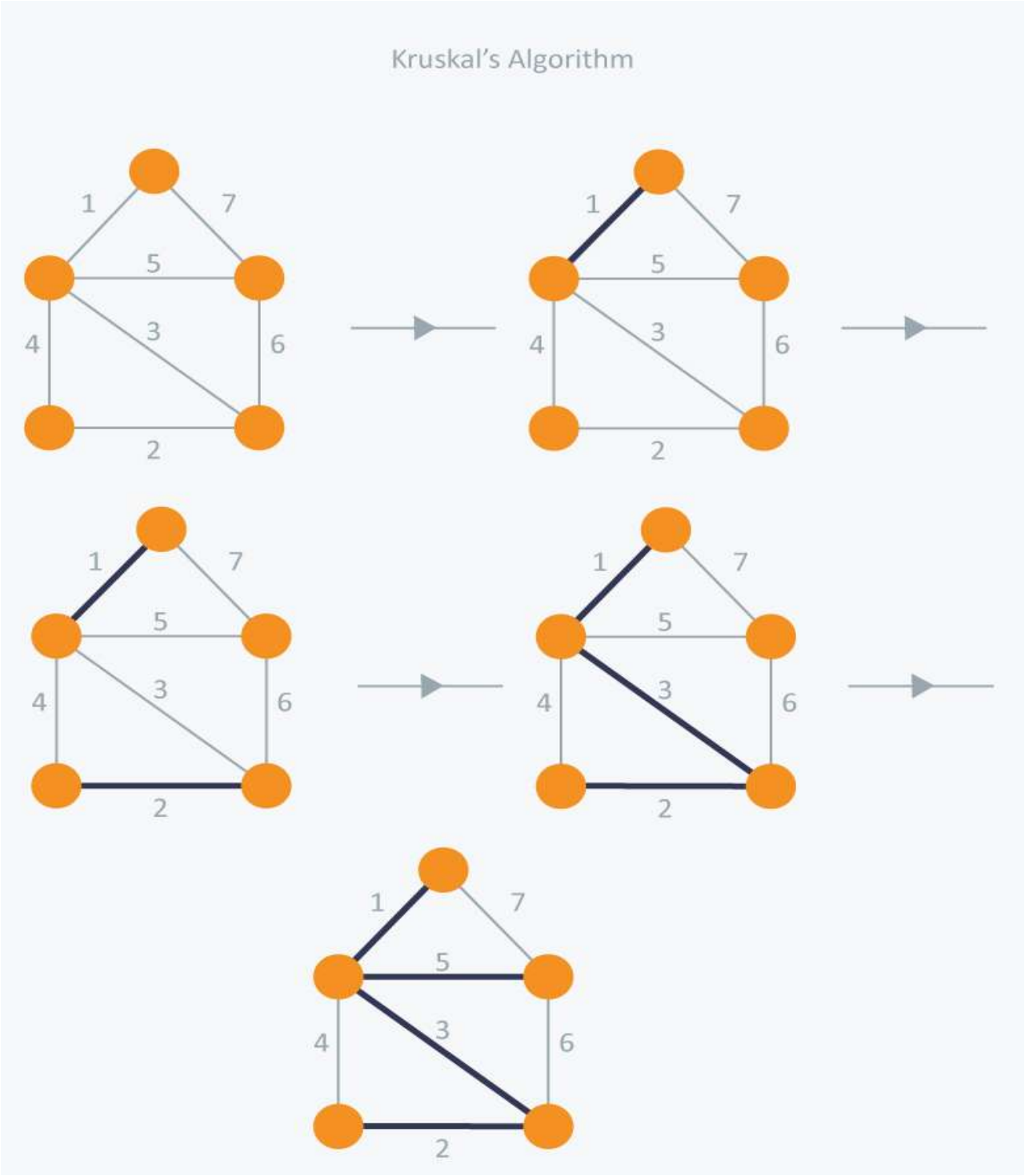
- **Cache Performance:** divide and conquer algorithms also tend to have good cache performance. Once a sub-problem fits in the cache, the standard recursive solution reuses the cached data until the sub-problem has been completely solved.
- It allows solving **difficult** and often impossible looking problems like the Tower of Hanoi. It reduces the degree of difficulty since it divides the problem into sub problems that are easily solvable, and usually runs faster than other algorithms would.
- Another advantage to this paradigm is that it often **plays a part in finding other efficient algorithms**, and in fact it was the central role in finding the quick sort and merge sort algorithms.

Disadvantages

- One of the most common issues with this sort of algorithm is the fact that the **recursion is slow**, which in some cases outweighs any advantages of this divide and conquer process.
- Another concern with it is the fact that sometimes it can become more **complicated than a basic iterative approach**, especially in cases with a large n. In other words, if someone wanted to add a large amount of numbers together, if they just create a simple loop to add them together, it would turn out to be a much simpler approach than it would be to divide the numbers up into two groups, add these groups recursively, and then add the sums of the two groups together.
- Another downfall is that sometimes once the problem is broken down into sub

problems, the same sub problem can occur many times. It is solved again. In cases like these, it can often be easier to identify and save the solution to the repeated sub problem, which is commonly referred to as memorization.

3a.



4 a. H**eap Sort Algorithm for sorting in increasing order:**

1. Build a max heap from the input data.

2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.

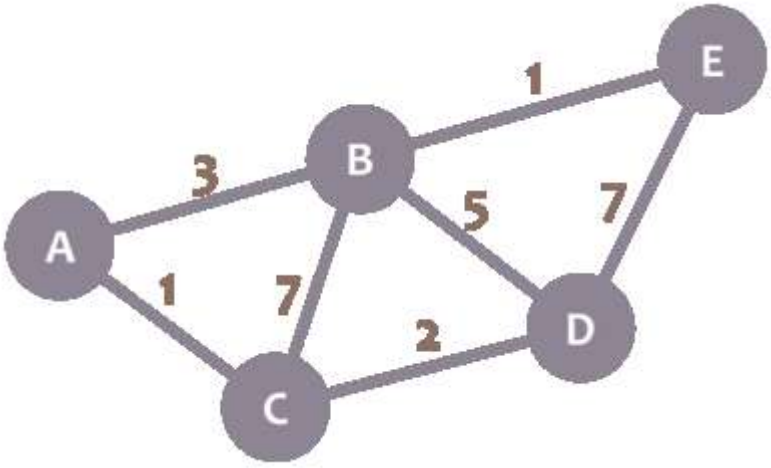
How to build the heap?

Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.
Lets understand with the help of an example:

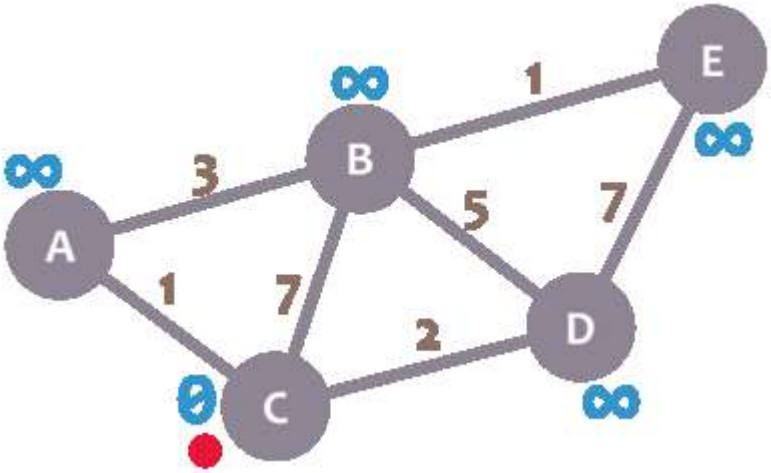
```
Input data: 4, 10, 3, 5, 1
         4(0)
        /   \
     10(1)   3(2)
    /   \
 5(3)    1(4)


The numbers in bracket represent the indices in the array
representation of data.


Applying heapify procedure to index 1:
         4(0)
        /   \
    10(1)    3(2)
    /   \
5(3)    1(4)


Applying heapify procedure to index 0:
        10(0)
        /  \
     5(1)  3(2)
    /   \
 4(3)    1(4)
The heapify procedure calls itself recursively to build heap
 in top down manner.
```

Complexity=O(nlogn)

5a.
Dijkstra's Algorithm allows you to calculate the shortest path between one node (you pick which one) and *every other node in the graph*. You'll find a description of the algorithm at the end of this page, but, let's study the algorithm with an explained example! Let's calculate the shortest path between node C and the other nodes in our graph:

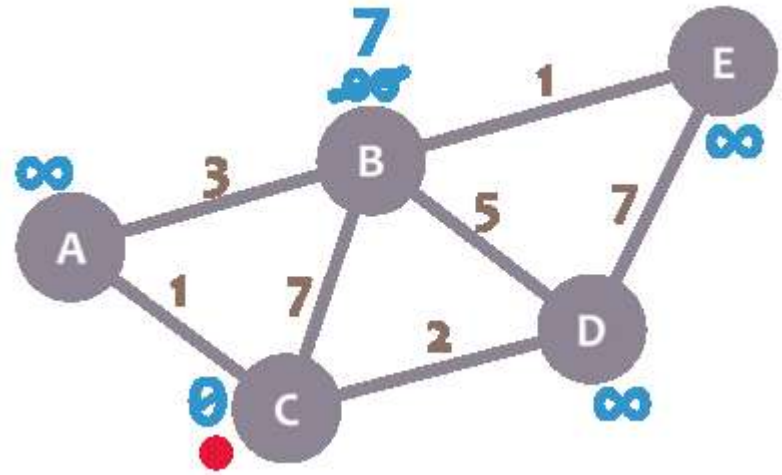


During the algorithm execution, we'll mark every node with its *minimum distance* to node C (our selected node). For node C, this distance is 0. For the rest of nodes, as we still don't know that minimum distance, it starts being infinity (∞):

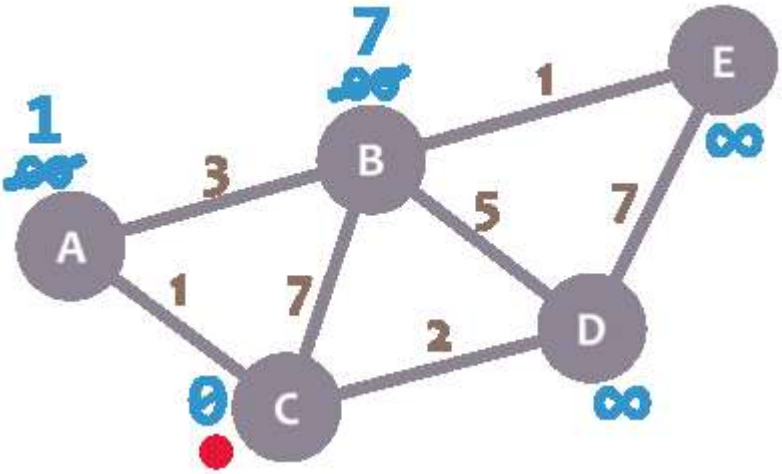


We'll also have a *current node*. Initially, we set it to C (our selected node). In the image, we mark the current node with a red dot.

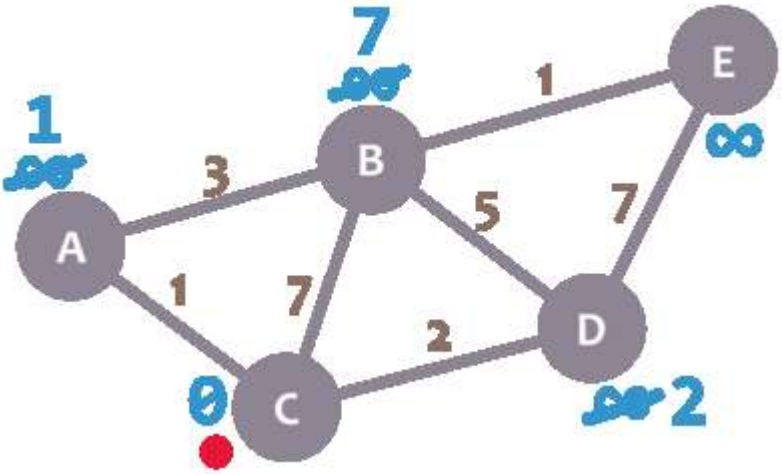
Now, we check the neighbours of our current node (A, B and D) in no specific order. Let's begin with B. We add the minimum distance of the current node (in this case, 0) with the weight of the edge that connects our current node with B (in this case, 7), and we obtain 0 + 7 = 7. We compare that value with the minimum distance of B (infinity); the lowest value is the one that remains as the minimum distance of B (in this case, 7 is less than infinity):



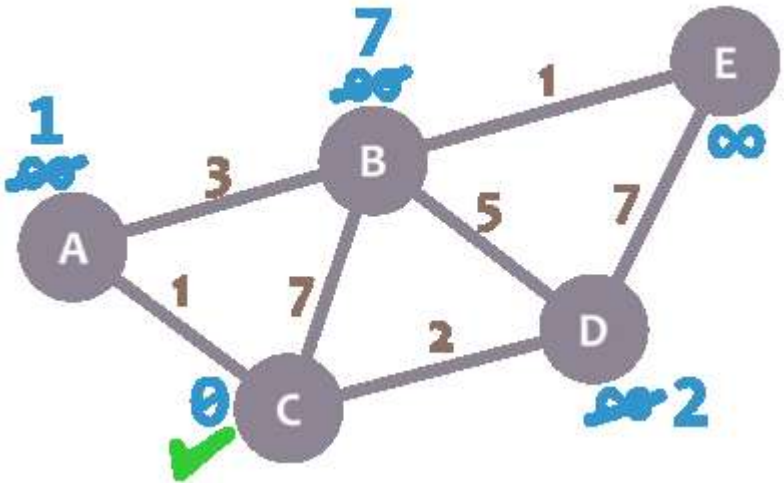
So far, so good. Now, let's check neighbour A. We add 0 (the minimum distance of C, our current node) with 1 (the weight of the edge connecting our current node with A) to obtain 1. We compare that 1 with the minimum distance of A (infinity), and leave the smallest value:



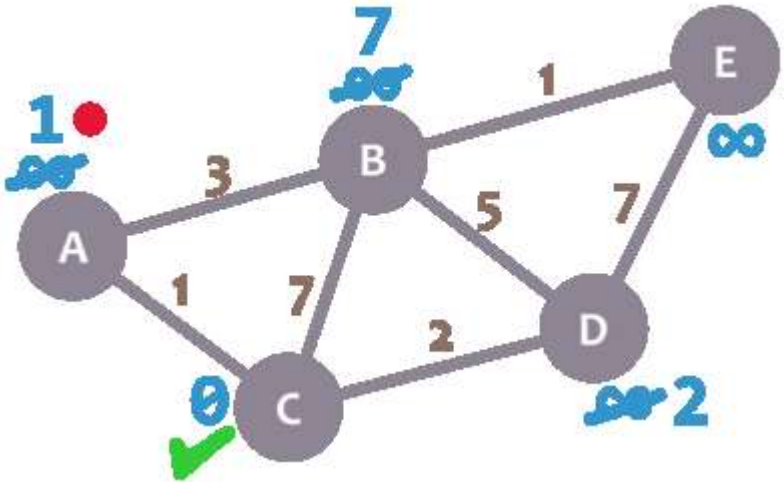
OK. Repeat the same procedure for D:



Great. We have checked all the neighbours of C. Because of that, we mark it as *visited*. Let's represent visited nodes with a green check mark:

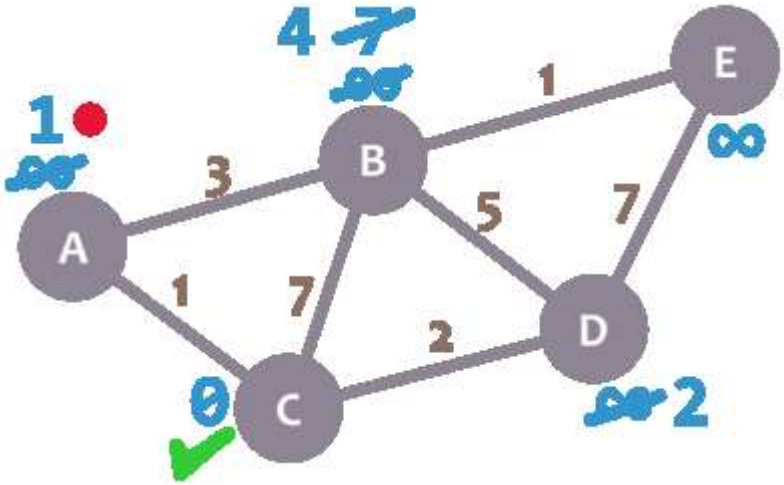


We now need to pick a new *current node*. That node must be the unvisited node with the smallest minimum distance (so, the node with the smallest number and no check mark). That's A. Let's mark it with the red dot:

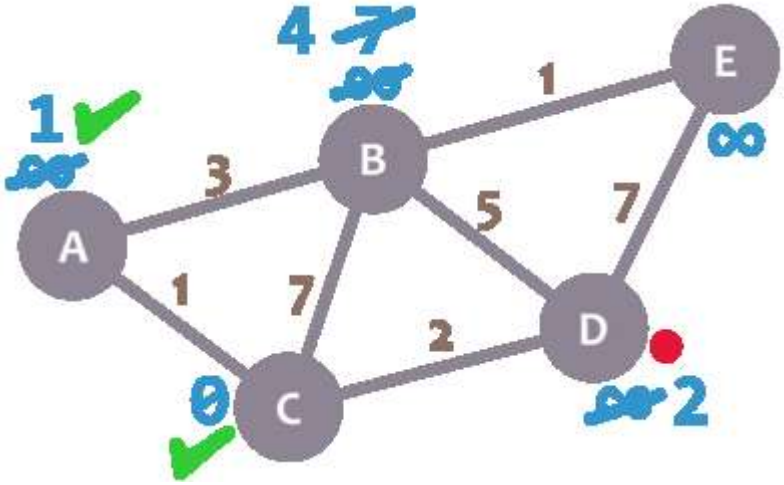


And now we repeat the algorithm. We check the neighbours of our current node, ignoring the visited nodes. This means we only check B.

For B, we add 1 (the minimum distance of A, our current node) with 3 (the weight of the edge connecting A and B) to obtain 4. We compare that 4 with the minimum distance of B (7) and leave the smallest value: 4.



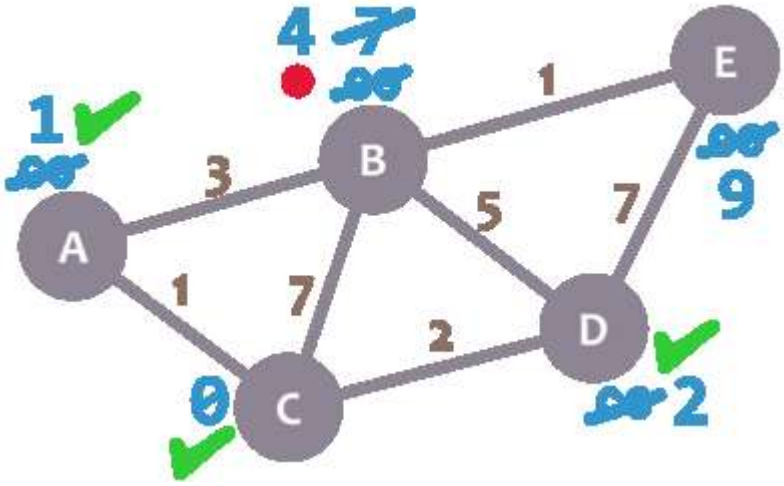
Afterwards, we mark A as visited and pick a new current node: D, which is the non-visited node with the smallest current distance.



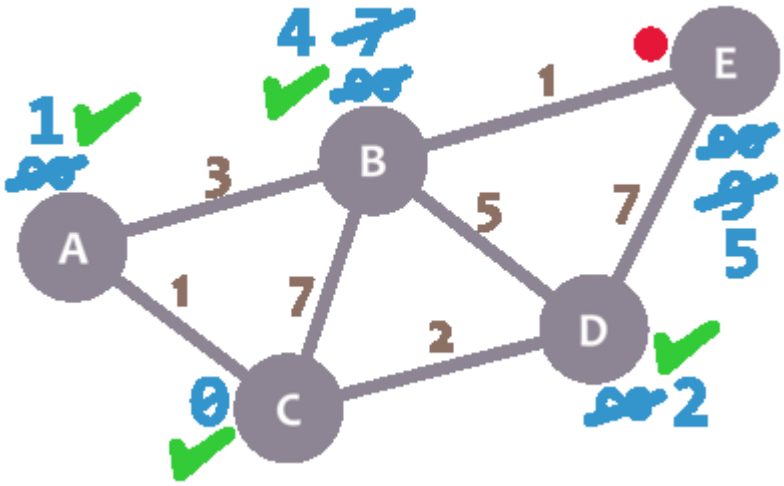
We repeat the algorithm again. This time, we check B and E.

For B, we obtain 2 + 5 = 7. We compare that value with B's minimum distance (4) and leave the smallest value (4). For E, we obtain 2 + 7 = 9, compare it with the minimum distance of E (infinity) and leave the smallest one (9).

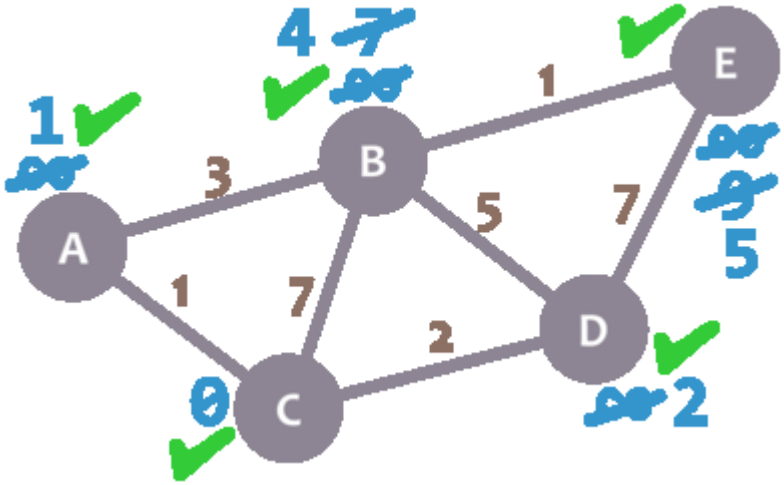
We mark D as visited and set our current node to B.



Almost there. We only need to check E. 4 + 1 = 5, which is less than E's minimum distance (9), so we leave the 5. Then, we mark B as visited and set E as the current node.



E doesn't have any non-visited neighbours, so we don't need to check anything. We mark it as visited.



As there are not univisited nodes, we're done! The minimum distance of each node now actually represents the minimum distance from that node to node C (the node we picked as our initial node)!

Here's a description of the algorithm:

1. Mark your selected initial node with a current distance of 0 and the rest with infinity.
2. Set the non-visited node with the smallest current distance as the current node `C`.
3. For each neighbour `N` of your current node `C`: add the current distance of `C` with the weight of the edge connecting `C-N`. If it's smaller than the current distance of `N`, set it as the new current distance of `N`.
4. Mark the current node `C` as visited.
5. If there are non-visited nodes, go to step 2.

6a.
1) Initialize result as empty.
2) find the largest denomination that is smaller than V.
3) Add found denomination to result. Subtract value of found denomination from V.
4) If V becomes 0, then print result. Else repeat steps 2 and 3 for new value of V

a)10 coins(1*2+8*6+1*5)
b)13 coins(12*6+1*5)
6b.

Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes single unit of time, so the minimum possible deadline for any job is 1. How to maximize total profit if only one job can be scheduled at a time.

Examples:

Input: Four Jobs with following deadlines and profits

JobID	Deadline	Profit
a	4	20
b	1	10
c	1	40
d	1	30

Output: Following is maximum profit sequence of jobs

c, a

Input: Five Jobs with following deadlines and profits

JobID	Deadline	Profit
a	2	100
b	1	19
c	2	27
d	1	25
e	3	15

Output: Following is maximum profit sequence of jobs

c, a, e

7.a. Decrease and Conquer Approach

Decrease-and-conquer is a general algorithm design technique, based on exploiting a relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem. Once such a relationship is established, it can be exploited either top down (usually recursively) or bottom up.

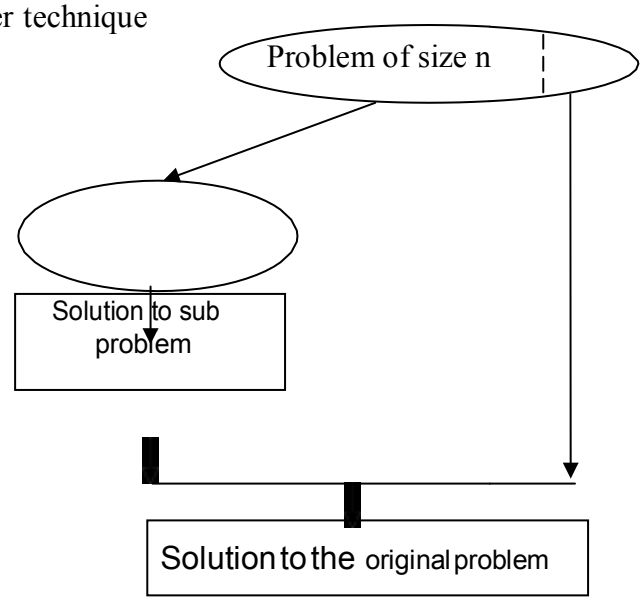
There are three major variations of decrease-and-conquer:

- decrease-by-a-constant, most often by one (e.g., insertion sort)
- decrease-by-a-constant-factor, most often by the factor of two (e.g., binary search)
- variable-size-decrease (e.g., Euclid's algorithm)

In the **decrease-by-a-constant** variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one although other constant size reductions do happen occasionally.

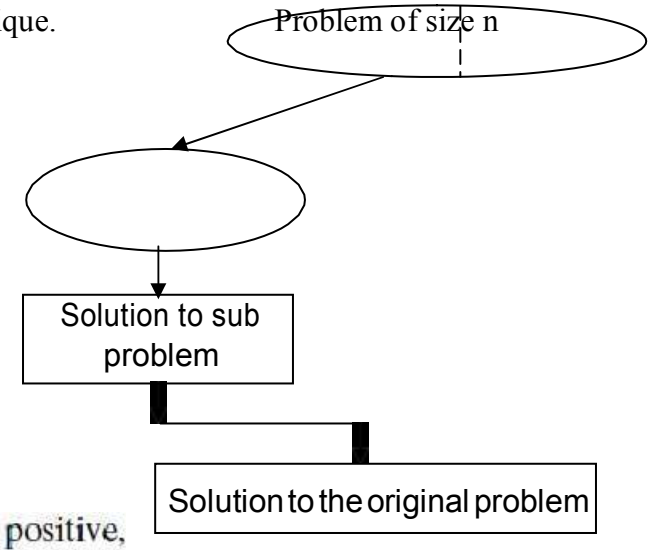
Figure: Decrease-(by one)-and-conquer technique

Example: $a^n = a^{n-1} \times a$



The **decrease-by-a-constant-factor** technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm. In most applications, this constant factor is equal to two.

Figure: Decrease-(by half)-and-conquer technique.



Example:

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive,} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd,} \\ 1 & \text{if } n = 0. \end{cases}$$

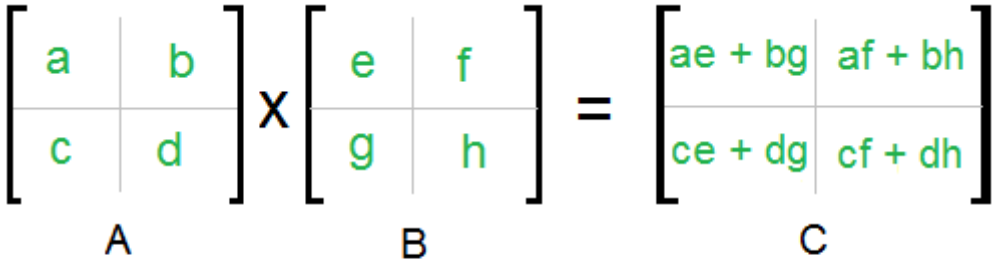
Finally, in the **variable-size-decrease** variety of decrease-and-conquer, the size-reduction pattern varies from one iteration of an algorithm to another.

Example: Euclid's algorithm for computing the greatest common divisor. It is based on the formula.
gcd(m, n) = gcd(n, m mod n).
Though the value of the second argument is always smaller on the right-hand side than on the left-hand side, it decreases neither by a constant nor by a constant factor.

7 b. In **Fractional Knapsack**, we can break items for maximizing the total value of knapsack. This problem in which we can break an item is also called the fractional knapsack problem.

```
Input :

   Same as above

Output :

   Maximum possible value = 240

   By taking full items of 10 kg, 20 kg and

   2/3rd of last item of 30 kg
```

8a. ***Divide and Conquer***
Following is simple Divide and Conquer method to multiply two square matrices.
1) Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.
2) Calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh.



A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

In the above method, we do 8 multiplications for matrices of size N/2 x N/2 and 4 additions. Addition of two matrices takes $O(N^2)$ time. So the time complexity can be written as

```
T(N) = 8T(N/2) + O(N²)

From Master's Theorem, time complexity of above method is O(N³)
which is unfortunately same as the above naive method.
```

Simple Divide and Conquer also leads to $O(N^3)$, can there be a better way?
In the above divide and conquer method, the main component for high time complexity is 8 recursive calls. The idea of **Strassen's method** is to reduce the number of recursive calls to 7.

Strassen's method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size N/2 x N/2 as shown in the above diagram, but in Strassen's method, the four sub-matrices of result are calculated using following formulae.

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

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

Time Complexity of Strassen's Method

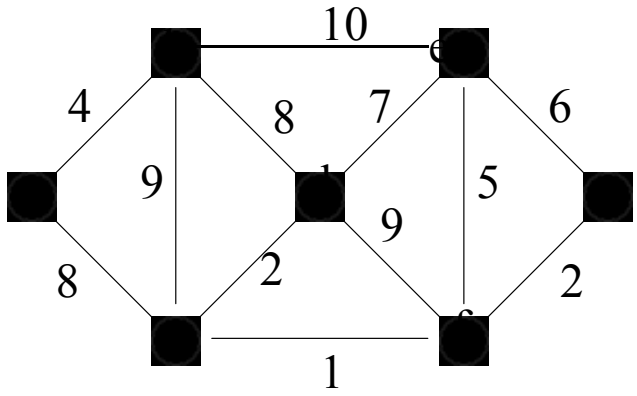
Addition and Subtraction of two matrices takes $O(N^2)$ time. So time complexity can be written as

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

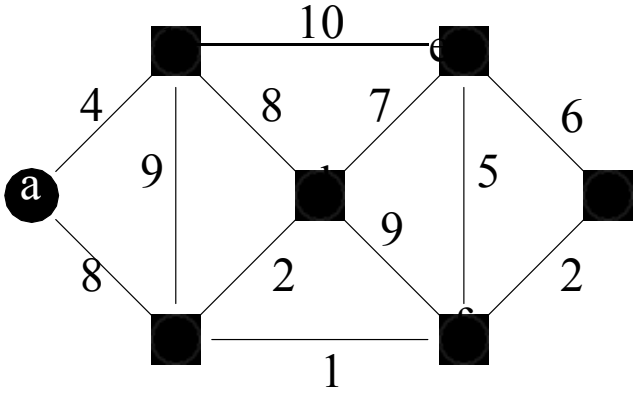
From Master's Theorem, time complexity of above method is $O(N^{Log^7})$ which is approximately $O(N^{2.8074})$

Generally Strassen's Method is not preferred for practical applications for following reasons.
1) The constants used in Strassen's method are high and for a typical application Naive method works better.
2) For Sparse matrices, there are better methods especially designed for them.
3) The submatrices in recursion take extra space.

8b.



Connected graph



Step 0

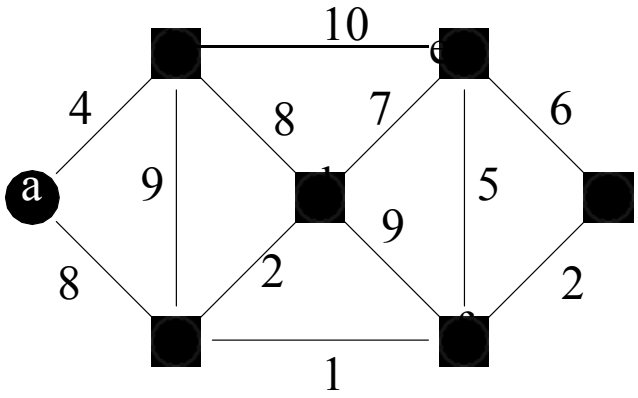
S={a}

V \ S = {b,c,d,e,f,g}

lightest edge = {a,b}

Prim's Algorithm

Prim's Example – Continued



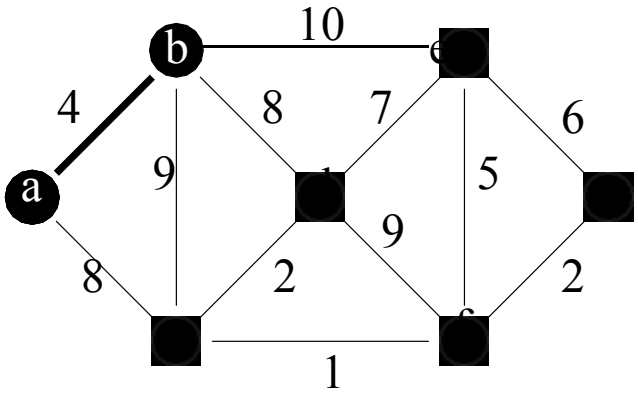
Step 1.1 before

S={a}

V \ S = {b,c,d,e,f,g}

A={}

lightest edge = {a,b}



Step 1.1 after

S={a,b}

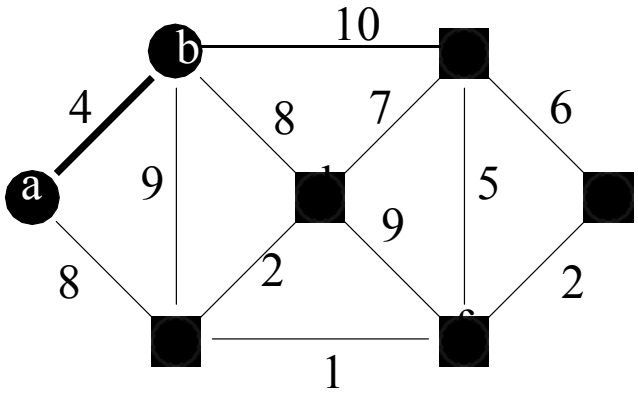
V \ S = {c,d,e,f,g}

A={{a,b}}

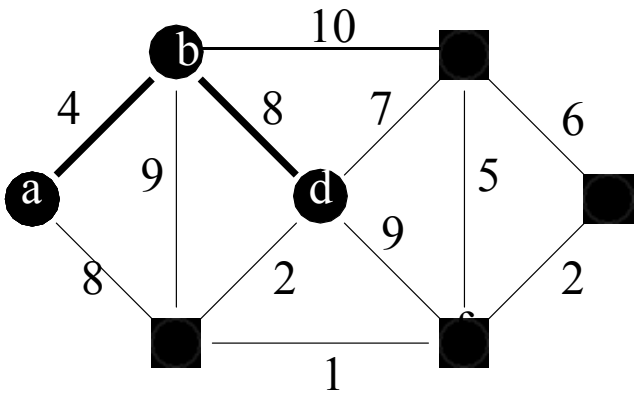
lightest edge = {b,d}, {a,c}

Prim's Algorithm

Prim's Example – Continued



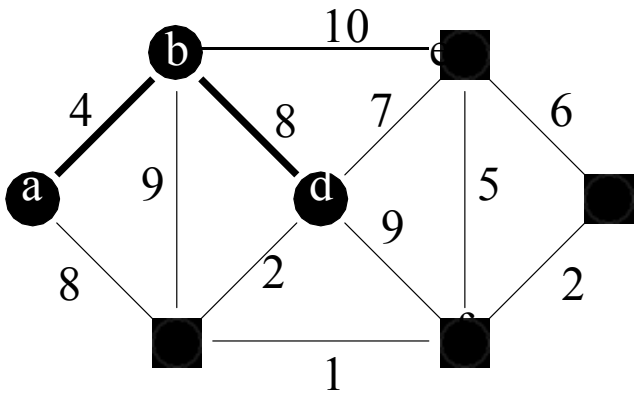
Step 1.2 before
S={a,b}
V \ S = {c,d,e,f,g}
A={{a,b}}
lightest edge = {b,d}, {a,c}



Step 1.2 after
S={a,b,d}
V \ S = {c,e,f,g}
A={{a,b},{b,d}}
lightest edge = {d,c}

Prim's Algorithm

Prim's Example – Continued



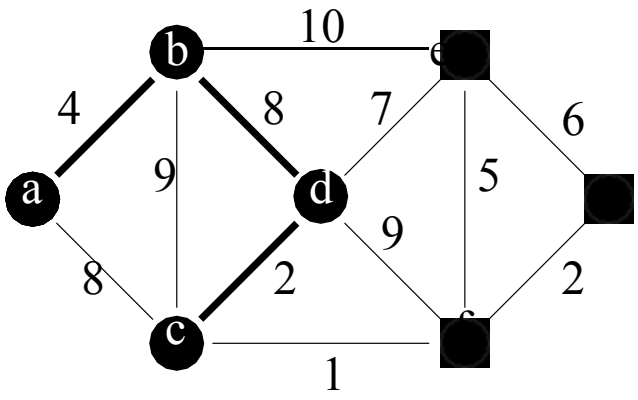
Step 1.3 before

S={a,b,d}

V \ S = {c,e,f,g}

A={{a,b},{b,d}}

lightest edge = {d,c}



Step 1.3 after

S={a,b,c,d}

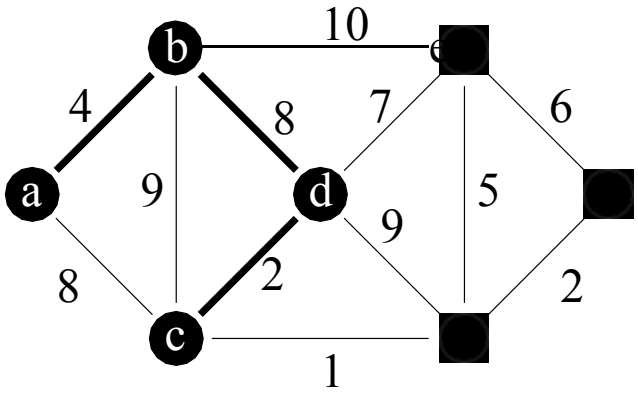
V \ S = {e,f,g}

A={{a,b},{b,d},{c,d}}

lightest edge = {c,f}

Prim's Algorithm

Prim's Example – Continued



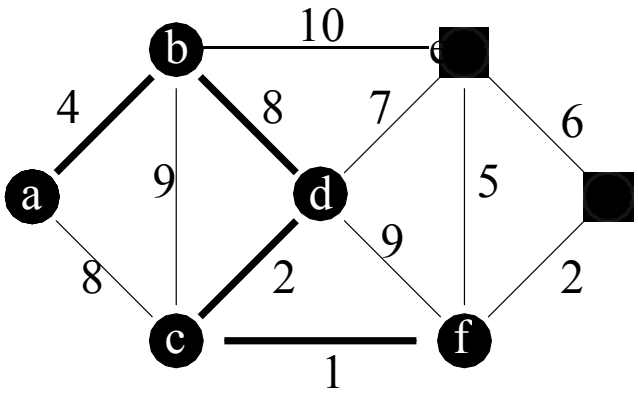
Step 1.4 before

S={a,b,c,d}

V \ S = {e,f,g}

A={{a,b},{b,d},{c,d}}

lightest edge = {c,f}



Step 1.4 after

S={a,b,c,d,f}

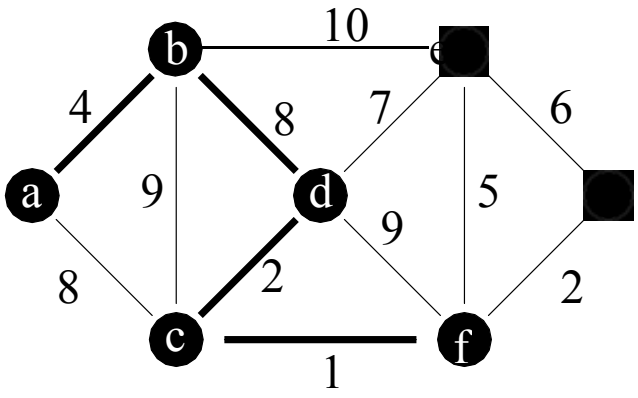
V \ S = {e,g}

A={{a,b},{b,d},{c,d},{c,f}}

lightest edge = {f,g}

Prim's Algorithm

Prim's Example – Continued



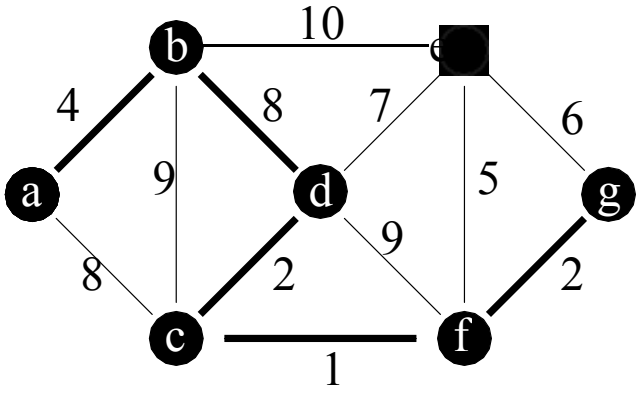
Step 1.5 before

S={a,b,c,d,f}

V \ S = {e,g}

A={{a,b},{b,d},{c,d},{c,f}}

lightest edge = {f,g}



Step 1.5 after

S={a,b,c,d,f,g}

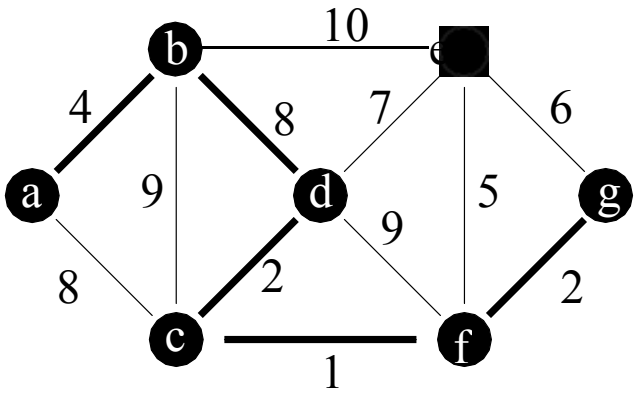
V \ S = {e}

A={{a,b},{b,d},{c,d},{c,f},
{f,g}}

lightest edge = {f,e}

Prim's Algorithm

Prim's Example – Continued



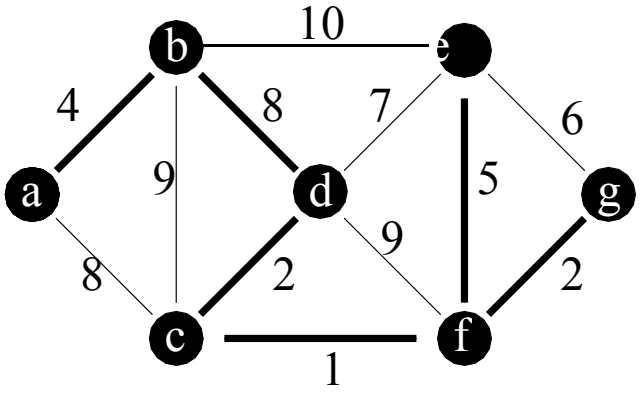
Step 1.6 before

S={a,b,c,d,f,g}

V \ S = {e}

A={{a,b},{b,d},{c,d},{c,f},
{f,g}}

lightest edge = {f,e}



Step 1.6 after

S={a,b,c,d,e,f,g}

V \ S = {}

A={{a,b},{b,d},{c,d},{c,f},
{f,g},{f,e}}

MST completed