CSCI 570 Exam 1 Review

Stable Matching

Ziyi Wu

Stable Matching

- We have a complete bipartite graph with two sides, M and W
- Each member on one side has a strict preference order for each member of the other side.
- Our goal is a stable matching
  - No instabilities: where in pairs (*m, w*) and (*m'*, *w'*)
    - *m* prefers *w* to *w'* and
    - *w'* prefers *m* to *m'*

Gale Shapley Algorithm

```
Initialize all m in M and w in W to be free
    While there is a man $m$ that is free and has not proposed to every woman:
        Let $w$ be $m$'s highest ranking women he has not proposed to
            If $w$ is free or prefers $m$ to her current partner $m'$:
                engage($m$,$w$)
                $m'$ becomes free if exists
            Endif
    Endwhile
```

Gale Shapley Properties

Runtime: n^2

GS will always correctly return a stable matching

The proposing side receives their best possible stable matches

The accept/reject side receives their worst possible stable matches

Question 1

A zoo wishes to pair up their warthogs and manatees based on their preferences.

Run the GS algorithm with the warthogs proposing in alphabetical order, and then again with the manatees proposing. What are the stable matchings returned by the algorithm?

Let *improvement* be a measure of how many ranks better a match is. How many ranks of improvement did manatees get due to proposing?

Manatee	**1**	**2**	**3**
Ally	Bethany	Cho	Alex
Brad	Alex	Bethany	Cho
Charlie	Bethany	Cho	Alex

Warthog	**1**	**2**	**3**
Alex	Ally	Brad	Charlie
Bethany	Brad	Charlie	Ally
Cho	Charlie	Ally	Brad

Solution 1

1) Alex -> Ally (e)
2) Bethany -> Brad(e)
3) Cho -> Charlie(e)

Final Matching (Alex, Ally), (Bethany, Brad), (Cho, Charlie)

1) Ally -> Bethany (e)
2) Brad -> Alex (e)
3) Charlie -> Bethany (e) Ally (f)
4) Ally -> Cho(e)

Final Matching (Alex, Brad), (Bethany, Charlie), (Cho, Ally) - Improvement: 3

Question 2

Prove that in the male-propose version of the GS algorithm, at most one man ends up with his worst possible choice.

Solution 2

Suppose that two men m_1 and m_2 end up with their worst possible choices, w_1 and w_2 respectively. WLOG, let m_2 be the second man to propose to his worst woman. This means that he must have proposed to every woman excluding w_2.

And given that w_1!= w_2, and that m_1 must have proposed to his entire preference list, we know that w_2 must have been proposed to by m_1 at some point previously as well. This means that by the last round, every woman has been proposed to.

In GS, we know that once a woman is proposed to, she stays engaged for the rest of the run because she can only reject because she can only reject or engage. So we now know that every woman is engaged.

But there are the same number of men and women so every man must be engaged as well. However our premise was that m_2 is proposing to w_2 which means he must be free.

This is a contradiction, thus it is not possible for two men to end up with their worst possible partners

Asymptotic Notation

Ziyang Guo

Asymptotic Notation

Examples:

	f(n)	g(n)
f(n) = **O**(g(n))	n	n^2
f(n) = **Ω**(g(n))	n	log(n)
f(n) = **θ**(g(n))	$2n^2$	$5n^2+3n$

Asymptotic Notation

Arrange these functions under the O notation using only = (equivalent) or < (a strict subset of):

(a) n^2; (b) n; (c) $\log(n)$; (d) $n\log(n)$; (e) 2^n; (f) n^n

Logarithmic < Polynomial < Exponential

Sol:

$c < b < d < a < e < f$

Amortized Analysis

Devarsh Amin

Amortized Analysis

- If we calculate the time complexity for a series of operations which are not unique , we tend to take time complexity for all the operations equal to the most time taken by an operation in that sequence.
- This seems very pessimistic approach as the actual time taken will be much less than this, since some operations take less time than the worst case time taken per each step.
- On the other hand, taking best case for each operation seems too much optimistic, therefore we need to find worst case time taken for each step or operation and then amortize it or disburse it among ‘n’ steps.

Problem-1

Suppose we perform sequence of n operations where i^{th} operation costs i if it is a power of 3 or costs 1 otherwise. Using aggregate method calculate the amortized cost for each operation.

Solution:

Let us prepare a table for representing cost for sequence of operations .

We consider number of operations such that it is equal to power of 3 .

If the index of operation is power of 3 we take its ‘ = index , else we take 1

Aggregate Method:-

Operation	**Cost**
1	1 (since its first operation)
2	1
3	3
4	1
5	1
6	1
7	1
8	1
9	9

Total cost : Cost for steps with index as power of 3 + Cost for steps not power of 3

$= (3^n - 1)*3 / 2 \quad + \quad (3^n - \log(3^n))$

Amortized Cost = $\Sigma_{\lim n \to \infty} ((3^n - 1)*3 / 2 \quad + \quad (3^n - \log(3^n))) / 3^n$

A.C = O(3)

Accounting Method:-

We find the amortized cost for each operation in this method by charging each operation by certain amount. The main emphasis of the accounting method is that, our account should never go negative of the total cost till the point, in the other words we should be charging each operation enough to cover for its cost and future operations cost if they exceed what the operation is assigned.

Let us calculate how much tokens should we assign to each of our operation.

Total cost for n operations = Cost for steps with index as power of 3 + Cost for steps not power of 3

Cost $< n + \Sigma^{\log n}_{i=0} 3^i = n + 3n - 1 = 4n - 1$

Hence , we can assign each step cost of 4 since , it would be amortized by slightly less than 4n for n steps

Index	Token	Token Used	Account
1	4	1	3
2	4	1	6
3	4	3	7
4	4	1	10
5	4	1	13

One can never exceed the token in bank or go negative in this method, Since the Amortized cost per operation is constant , Total cost for n operations , O(n)

Problem -2

Suppose we have a dynamic array which we expand , when the array is not completely filled, but filled till certain percentage. We do this to make sure we never run out of memory. Prove that the amortized cost for an operation still remains constant.

Solution:

Let us denote k be the factor representing the filled part of array, where 0<k<1.

For such problem , we always break down total cost in 2 parts:-

1. Cost for inserting an element
2. Cost for copying the elements from previous array to new array.

Cost of Insertion will always remain constant and will be equal to number of operations.

The cost of copying the elements can at most increase in geometric progression when k =1 and can never exceed that, which in turn will never be upper bound of total number of operations. Therefore, A.C for such operation would always be constant.

Let us understand math behind by taking k =¾.

Insert	Old	New	Cost for Copy
1	1	-	-
2	1	2	1
3	2	4	2
4	4	4	-
5	4	8	4
6	8	8	-
7	8	8	-
8	8	16	7
9	16	16	-

Therefore, Total Cost = Total number of Insertions + Cost to copy

Here we have Total Elements= 2^n+1

Cost to Copy till $2^n+1 = 1+2+4+7... < 1+2+4+8+..$

As we can see the cost to copy will always be less than sum of G.P till the last insert for $0<k<1$

Therefore, Cost to Copy till $2^n+1 \leq 2^{n+1}-1$

Therefore, Total Cost = $2^n+1 + 2^{n+1}-1$

Therefore Amortized cost = $3.2^n/(2^n+1) = O(3)$

Hence , Amortized cost for insertion in such schema will always be constant

.

Problem - 3

Suppose you have a data structure such that sequence of n operations has Worst-Case Cost of $n^2\log n$, what can be the highest actual cost per operation?

Solution: Since the Worst-Case Cost for n operations is $n^2\log n$, this is the highest cost an operation can possibly have in that sequence.

For example, 1,n,1,nlogn,.....$n^2\log n$

Problem - 4

(T/F) The actual cost of an operation in a sequence of n operations , can be higher than A*n, where A is the amortized cost for that sequence of operations

Solution : False

If A is the amortized (average) cost of the operations in that sequence, then A*n represents the total cost of that sequence of operations. So a given operation cannot have a cost greater than that.

Greedy Part 1

TA: Alex Wang

Greedy: 2 Common Methods of Proof of Optimality

- **stay-ahead:** an inductive method to show that for each step you take, you are always ahead of any other solution according to some metric

- **exchange:** show that for your greedy solution, any swaps between steps/elements cannot achieve a better result

Problem 1

For a graph G = (V, E), a vertex cover of G is a set of vertices that includes at least one endpoint of each edge of G. Assuming our graph G is a tree, find an algorithm that finds a minimum vertex cover of G.

Solution 1

Let S be the set of vertices that will be the vertices we return. If our graph G has no vertices or only 1 vertex, return S. Otherwise, add all vertices 1 layer above all our leaf vertices in the tree to S, and remove the vertices and its adjacent edges from G, as well as the leaf vertices. Recursively apply this step until we end up with 1 or no vertices in G. Return S. This is optimal because we have to cover the lead edges regardless, so our cover will always either include the leaf node itself or its parent. If we include its parent, we cover just as much if not more than if we select the leaf node. The problem then reduces to a vertex cover for a subset of the original tree. This subset is smaller than the one we would have gotten from choosing the leaf node, and a vertex cover of tree is at least as large as the vertex cover of a subset of it.

Problem 2

Given an array of distinct integers A = [a_1, a_2, …, a_n] and a sequence p of numbers p = [1, 2, …, n], rearrange the numbers in p such that $\sum_{i=1}^{n} |a_i - p_i|$ is minimized.

Solution 2

Consider 2 real numbers ai and aj, and 2 other real numbers, pi and pj, where ai < aj and pi < pj. WLOG, assume that all 4 of these numbers are distinct. There are 6 possible orderings given these constraints: ai < aj < pi < pj, ai < pi < aj < pj, ai < pi < pj < aj, pi < ai < aj < pj, pi < ai < pj < aj, and pi < pj < ai < aj. For each of these cases, we want to show that |ai - pi| + |aj - pj| ≤ |ai - pj| + |aj - pi|, which basically means that we cannot get a smaller distance if we exchange elements pi and pj. Therefore if we have ai < aj if and only if pi < pj for all i, j, we will have the minimum distance. We can store the index of the elements in A in a hashmap h. We then sort A and pair it with p, and reorder p according to indices of h. Since our algorithm produces elements ai and pi with these properties, our algorithm produces a minimum distance.

In the previous proof, we left out details about showing $|ai - pi| + |aj - pj| \le |ai - pj| + |aj - pi|$ for each of 6 cases. Here is an example of showing that this is true for the case for $ai < pi < aj < pj$. Define the distance between ai and pi, pi and aj, and aj and pj, as d1, d2, and d3 respectively. Then $|ai - pi| + |aj - pj| = d1 + d3 \le d1 + 2*d2 + d3 = |ai - pj| + |aj - pi|$. You can verify that the other cases are true as well. In the exam, there will not be this much detail involved in a proof, and if there is, we will state which parts can be skipped or assumed to be true.

Greedy Part 2

TA: Tiancheng Lin

T/F questions (3 min)

1. A greedy algorithm always makes the choice that looks best at the moment.

1. The shortest path between two points in a graph will not change if the weight of each edge is increased by an identical number.

1. Gale Shapley algorithm is based on greedy technique.

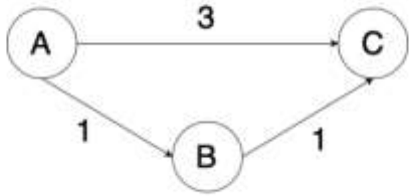
T/F questions

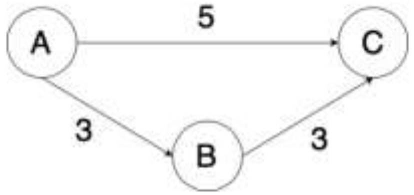
1. A greedy algorithm always makes the choice that looks best at the moment.

   Answer: True

1. The shortest path between two points in a graph will not change if the weight of each edge is increased by an identical number.

   Answer: False



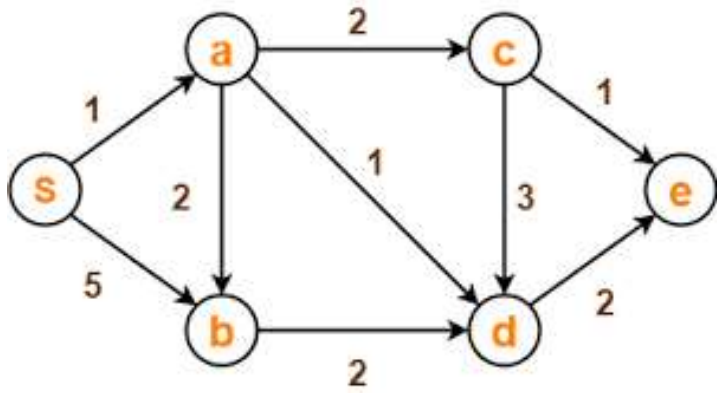


1. Gale Shapley algorithm is based on greedy technique.

   Answer: True

Question 1 (3 min)

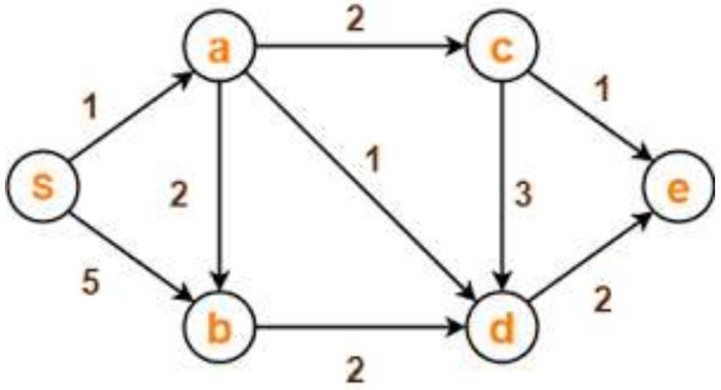
Find shortest path for each vertex from S using Dijkstra’s Algorithm.



Question 1

Find shortest path for each vertex from S using Dijkstra’s Algorithm.

a	b	c	d	e
1	5	INF	INF	INF
	3	3	**2**	INF
	3	3		4
		3		4
				4



Question 2 (5 min)

Given an array of meeting time intervals where intervals[i] = [start_i, end_i], find the minimum number of conference rooms required. i.e. [[0,30], [5,10], [15,20]]

Example: meeting 1: [0,30],meeting 2: [5,10], meeting 3: [15,20], minimum number of conference rooms required is 2, because meeting 2 and meeting 3 can share the same room but meeting 1 must reserve its own room.

(Hints: Greedy + priority queue)

Question 2

Given an array of meeting time intervals where intervals[i] = [start_i, end_i], find the minimum number of conference rooms required. i.e. [[0,30], [5,10], [15,20]]

Algorithm:

1. Sort the given meetings by their **start time**.
2. Initialize a new min-heap and add the first meeting's **ending time** to the heap. We simply need to keep track of the ending times as that tells us when a meeting room will get free.
3. For every meeting room check if the minimum element of the heap i.e. the room at the top of the heap is free or not, which means we **assign this meeting to the earliest-available room**.
4. If the room is free, then we extract the topmost element and add it back with the ending time of the current meeting we are processing; If not, then we allocate a new room and add it to the heap.
5. After processing all the meetings, the size of the heap will tell us the number of rooms allocated. This will be the minimum number of rooms needed.

Time complexity:

O(nlogn) for sorting, N meetings * (O(1) for getMin + O(logn) for extractMin +O(logn) for insert) = **O(nlogn)**

Question 2

Given an array of meeting time intervals where intervals[i] = [start_i, end_i], find the minimum number of conference rooms required. i.e. [[0,30], [5,10], [15,20]]

Proof of correctness:

1. Assume that the optimal solution does not assign [x1,y1] to the room of the earliest-end meeting: it assigns [x1,y1] to another free room ended in x0. If we can assign [x1,y1] to this free room, then the later meeting, [x2,y2] can also assign to the earliest-end meeting because x0<=x1<=x2, the result of both of the strategy ends with two rooms ended in y1 and y2, so our solution is the same as optimal solution, which means that this strategy is optimal.
2. Assume that when we assign [x1,y1], an optimal solution assigns [x2,y2] first where x1 < x2.
   a. y1 <= x2. Then in our solution, we don't need to assign extra meeting room, but optimal solution needs an extra room because x1 < y1 <= x2 < y2, which means that the meeting [x1,y1] cannot reuse the same meeting room in the optimal solution.
   b. y1 > x2. Then both of solutions needs an extra room and ends with two meeting rooms, which end with y1 and y2. So optimal solution is not better than our solution.
   c. We prove that our solution is not worse than optimal solution, hence this strategy is optimal.

MST

TA: Rong Wang

MST 1

[T/F] If all edge weights of a given graph is the same, then every spanning tree of that graph is minimum.

True.

Spanning tree: Any tree that covers all nodes of a graph is called a spanning tree.

MST 2

[T/F] If the weight of each edge in a connected graph is distinct, then the graph contains exactly one unique minimum spanning tree.

True.

Proof: Suppose two MSTs T1 and T2 of G,

E = set of edges that is in either T1 or T2 but not both.

e is the min edge in E.

Suppose e is not in T2. Adding e to T2 creates a cycle. Then in this cycle, at least one edge, say f, is not in T1. Since e is te min edge in E, then w(e) <= w(f). And all edges have distinct weights, w(e) < w(f). Replacing f with e results in a new spanning tree with less weight than T1 and T2. Therefore, contradiction.

MST 3

There are n cities and existing roads R_u. The government plans to build more roads such that any pair of cities are connected by roads. The candidate roads are R_p and each road has a cost c_i (c_i > 0). Design an algorithm to decide the roads to be built such that all cities are connected by road(s) and the cost is minimized.

MST 3

1: Construct a graph with n nodes.

2: Add R_u to the graph and set edge weight to be 0.

3: Add R_p to the graph and set edge weights to be the cost.

4: Run MST algorithm, return roads that are in both R_p and MST.

1: Get connected components of the graph with R_u.

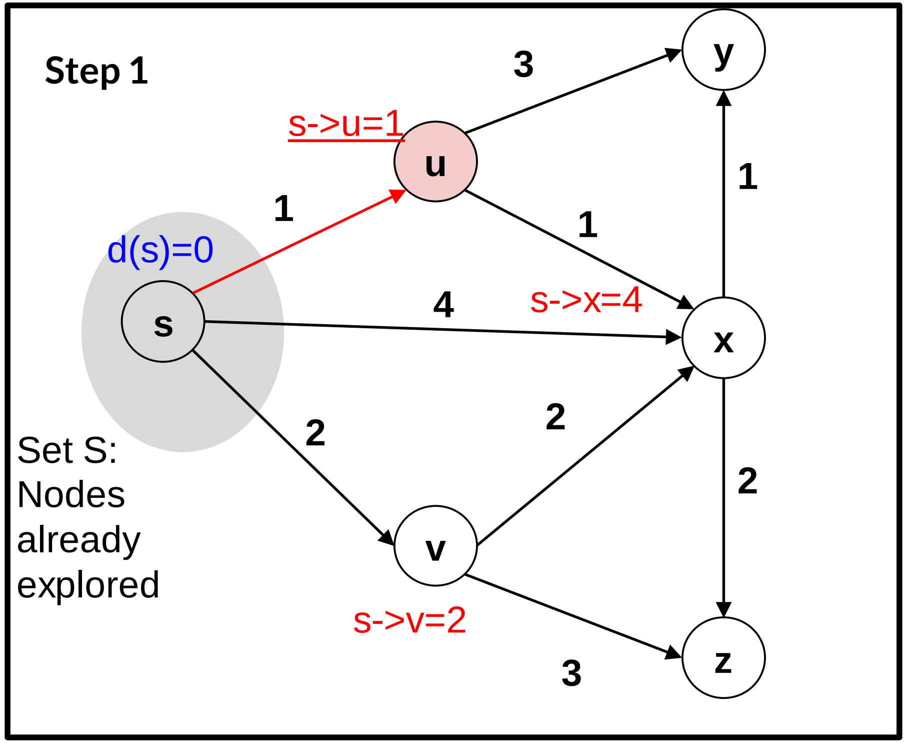
2: Merge nodes and edges.

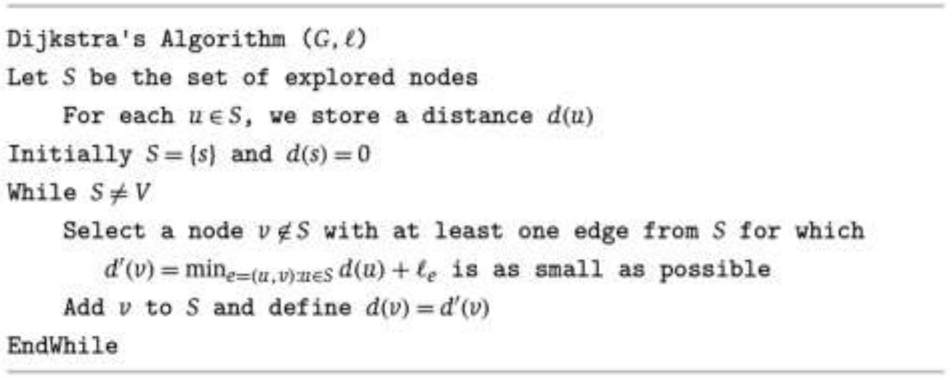
3: Add R_p, run MST.

Shortest Path

Nan Xu

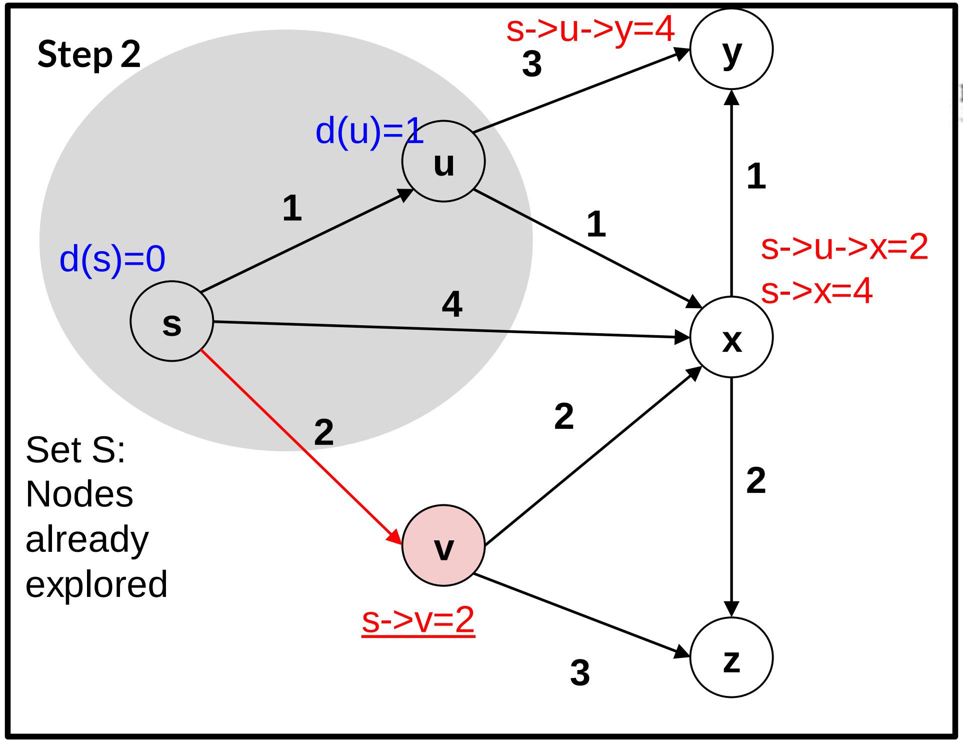
Recap: Dijkstra's Algorithm

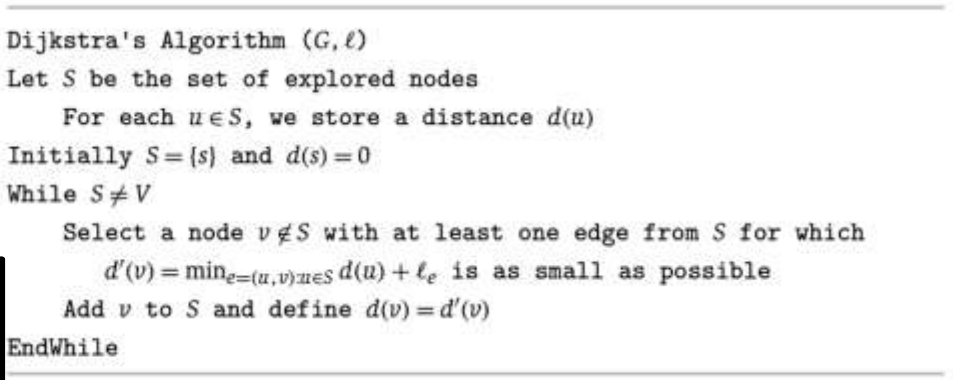




```
Dijkstra's Algorithm $(G, \ell)$
Let $S$ be the set of explored nodes
    For each $u \in S$, we store a distance $d(u)$
Initially $S = \{s\}$ and $d(s) = 0$
While $S \neq V$
    Select a node $v \notin S$ with at least one edge from $S$ for which
        $d'(v) = \min_{e=(u,v):u \in S} d(u) + \ell_e$ is as small as possible
    Add $v$ to $S$ and define $d(v) = d'(v)$
EndWhile
```

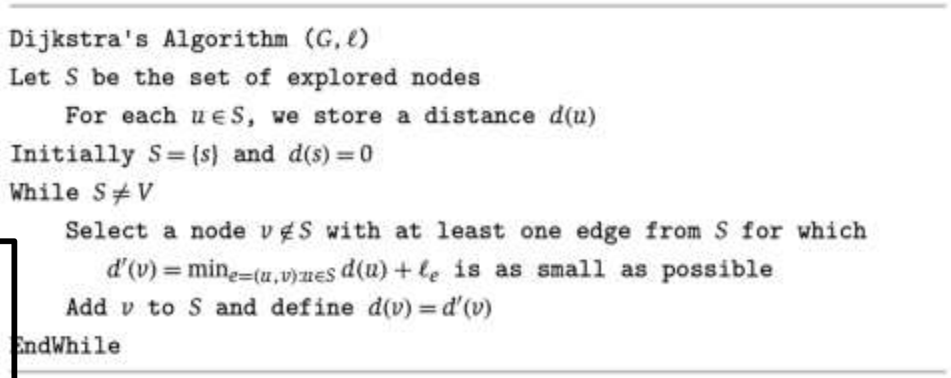
Recap of Dijkstra's Algorithm



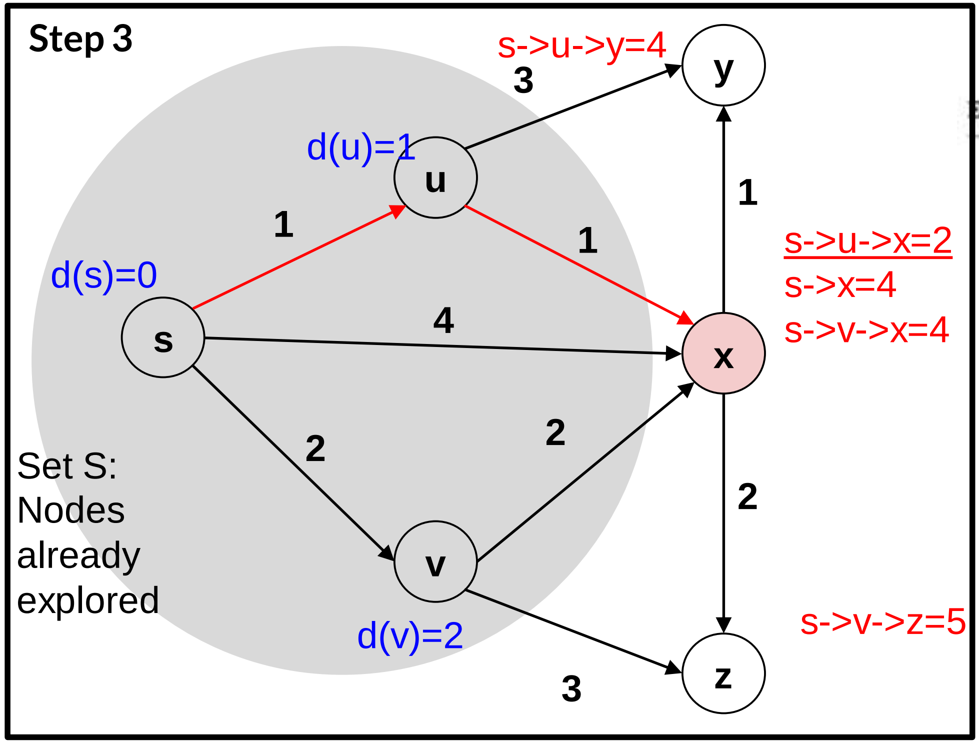


```
Dijkstra's Algorithm $(G, \ell)$
Let $S$ be the set of explored nodes
    For each $u \in S$, we store a distance $d(u)$
Initially $S = \{s\}$ and $d(s) = 0$
While $S \neq V$
    Select a node $v \notin S$ with at least one edge from $S$ for which
        $d'(v) = \min_{e=(u,v):u \in S} d(u) + \ell_e$ is as small as possible
    Add $v$ to $S$ and define $d(v) = d'(v)$
EndWhile
```

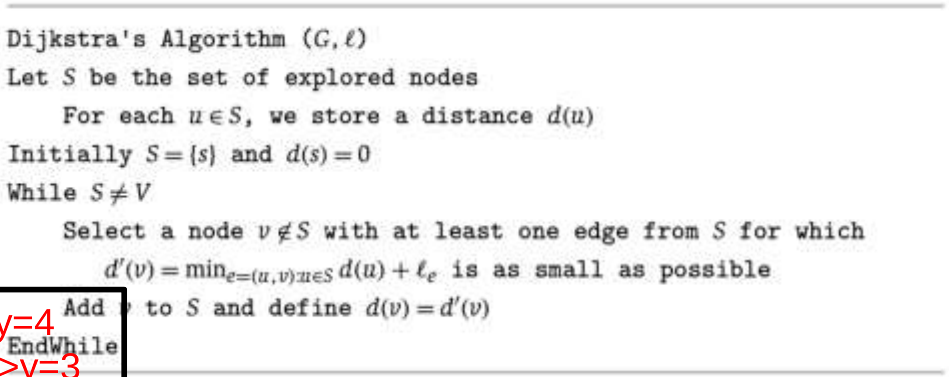
Recap of Dijkstra’s Algorithm



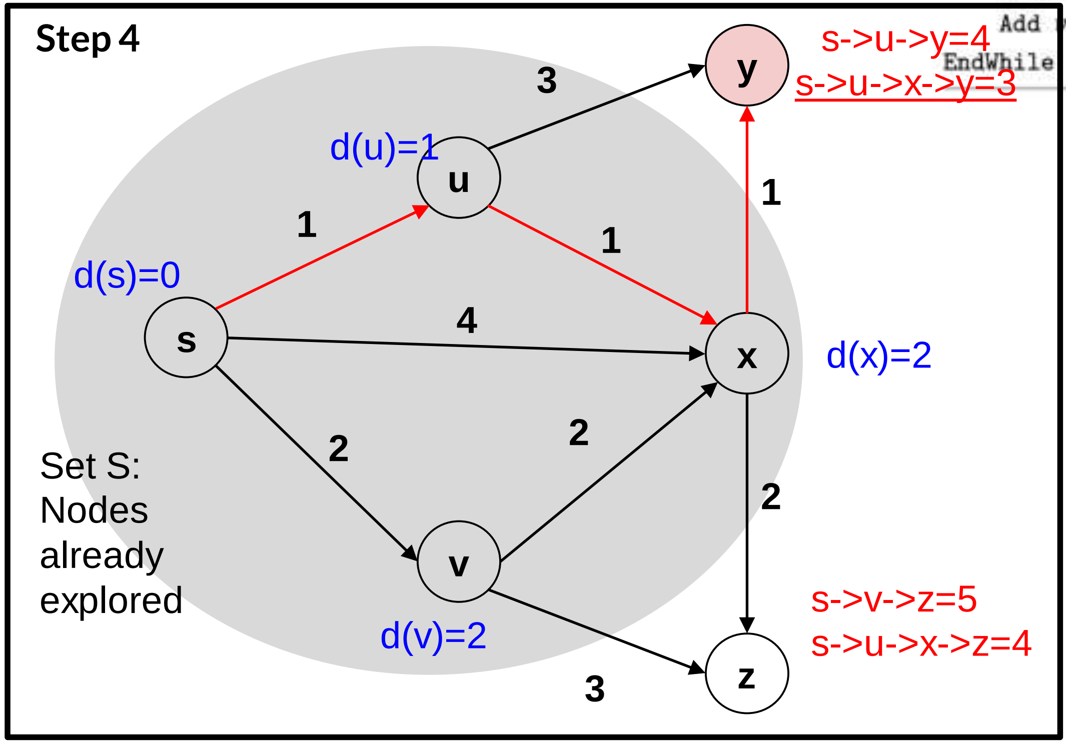
```
Dijkstra's Algorithm $(G, \ell)$
Let $S$ be the set of explored nodes
    For each $u \in S$, we store a distance $d(u)$
Initially $S = \{s\}$ and $d(s) = 0$
While $S \neq V$
    Select a node $v \notin S$ with at least one edge from $S$ for which
        $d'(v) = \min_{e=(u,v):u \in S} d(u) + \ell_e$ is as small as possible
    Add $v$ to $S$ and define $d(v) = d'(v)$
EndWhile
```



Recap of Dijkstra’s Algorithm

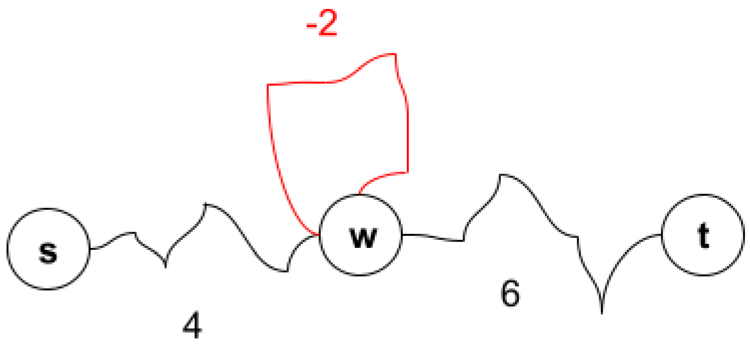


```
Dijkstra's Algorithm $(G, \ell)$
Let $S$ be the set of explored nodes
    For each $u \in S$, we store a distance $d(u)$
Initially $S = \{s\}$ and $d(s) = 0$
While $S \neq V$
    Select a node $v \notin S$ with at least one edge from $S$ for which
        $d'(v) = \min_{e=(u,v):u \in S} d(u) + \ell_e$ is as small as possible
    Add $v$ to $S$ and define $d(v) = d'(v)$
EndWhile
```



NOTE: Negative Weights - Negative Cycles

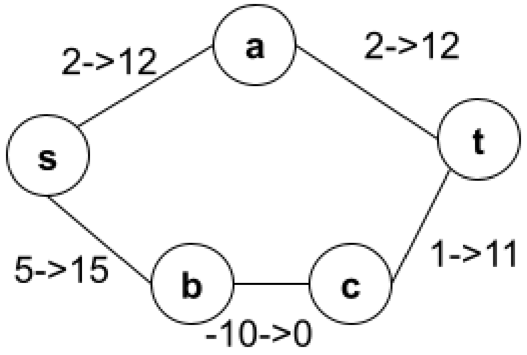
- Negative cycles: if some cycle has a negative total cost, we can make the s-t path as low cost as we want



s->w->t: 4+6=10
s->w (cycle once)->t: 4-2+6=8
s->w (cycle twice)->t: 4-2*2+6=6
…
s->w (cycle N times)->t: 4-2*N+6=-∞, when N->∞

NOTE: Negative Weights - Solve with a Big Number?

- Adding a large number M to each edge DOES NOT work
  - New cost of path P: M*len(P)+old_cost(P)
  - When M is big, the number of hops (path length) will dominate rather than old path cost

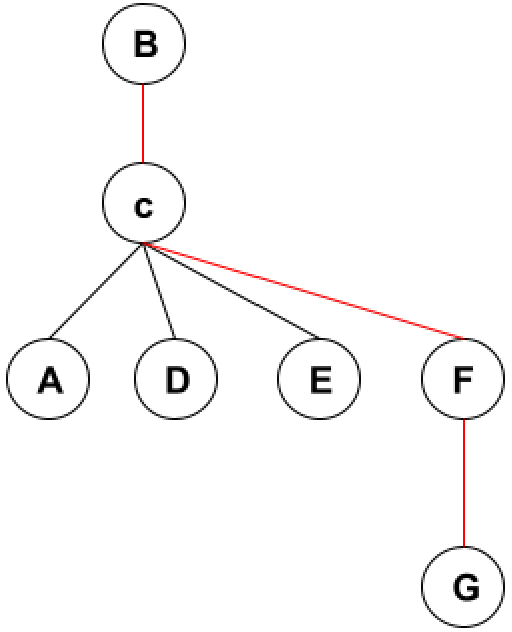


Before: s-b-c-t (-4) is shorter than s-a-t (4)
After adding 10 to all edges: s-a-t (24) is shorter than s-b-c-t (26)

Solution: Bellman-Ford in Dynamic Programming Section

Problem I

- The diameter of a graph is the maximum of the shortest paths' lengths between all pairs of nodes in graph G.
- Design an algorithm which computes the diameter of a connected, undirected, unweighted graph in O(mn) time, and explain why it has that runtime.



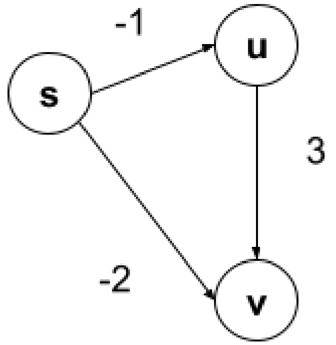
Diameter of this graph is: 3

Solution I

- Unweighted graph => BFS can be used for shortest path search for each source node in O(m+n)
    - For one source node s, report the maximum layer reached
- Repeat BFS for each node in O(n)
- Total time complexity: O(n(m+n))
    - Connected graph, the number of edges are at least n-1, at most n(n-1)/2, hence n=O(m)
    - => O(m+n) is O(m)
    - total time complexity: O(nm)

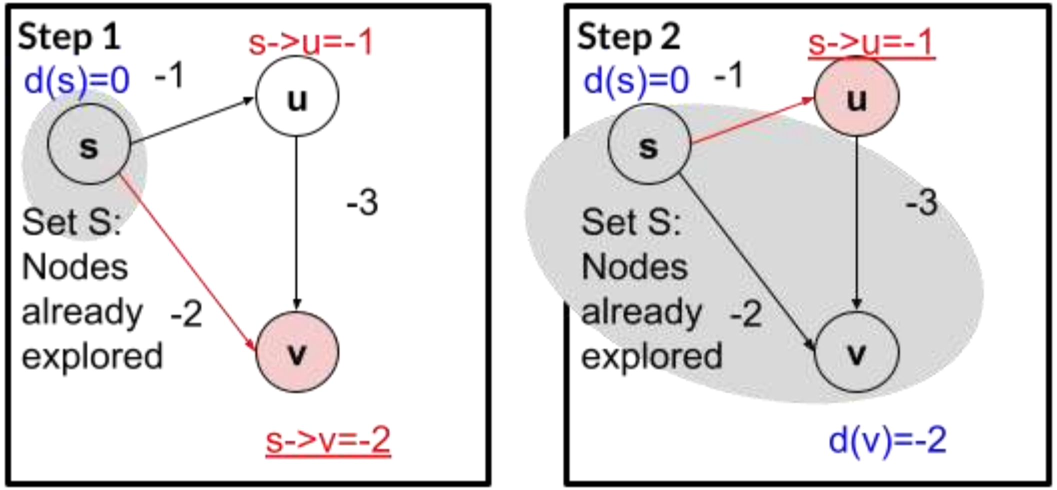
Problem II:

- Dijkstra’s algorithm works correctly on a directed acyclic graph even when there are negative-weight edges.
- TRUE or FALSE



Solution II

- False

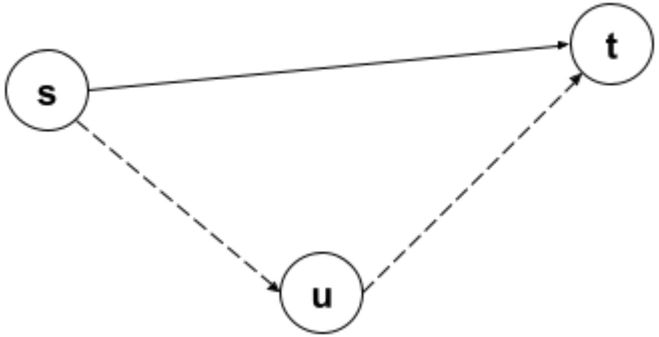


Dijkstra's algorithm: d(v)=-2
Actual shortest path: s-u-v, with cost -4

Problem III

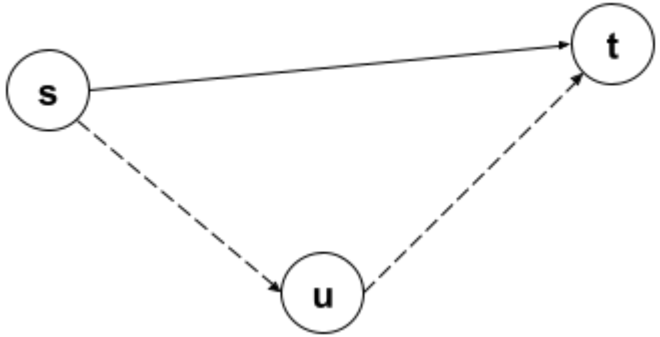
- Suppose that you want to get from vertex s to vertex t in a connected undirected graph G = (V; E) with positive edge costs, but you would like to stop by vertex u (imagine that there are free burgers at u) if it is possible to do so without increasing the length of your path by more than a factor of **α**

- Describe an efficient algorithm in O(|E| log |V|) time that would determine an optimal s-t path given your preference for stopping at u along the way if doing so is not prohibitively costly. (In other words, your algorithm should either return the shortest path from s to t, or the shortest path from s to t containing u, depending on the situation.)



Solution III

- Positive edges => Dijkstra's algorithm for shortest path from source node s, and from source node u
  - shortest path from s => we know $d_s(t)$ and $d_s(u)$
  - Shortest path from u => we know $d_u(t)$
- Compare $d_s(u) + d_u(t)$ and $\alpha^* d_s(t)$
  - If $d_s(u) + d_u(t) <= \alpha^* d_s(t)$, stop by u for burger!
  - If $d_s(u) + d_u(t) > \alpha^* d_s(t)$, go directly from s to t
- Time complexity
  - Connected graph: $|V|=O(|E|)$
  - Running Dijkstra's algorithm twice:
    - Binary heap: $O((|V| +|E|) \log |V|) => O(|E| \log |V|)$
    - Fibonacci heap: $O(|E|+|V|\log|V|) => O(|E| \log |V|)$



DIVIDE AND CONQUER

TA: Rutu Desai

Question 1

A polygon is called convex if all of its internal angles are less than 180°and none of the edges cross each other. We represent a convex polygon as an array V with n elements, where each element represents a vertex of the polygon in the form of a coordinate pair (x, y). We are told that V [1] is the vertex with the least x coordinate and that the vertices V [1], V [2], . . . , V [n] are ordered counter-clockwise. Assuming that the x coordinates (and the y coordinates) of the vertices are all distinct, do the following:

(a) Give a divide and conquer algorithm to find the vertex with the largest x coordinate in O(log n) time.

(b) Give a divide and conquer algorithm to find the vertex with the largest y coordinate in O(log n) time.

Solution

(a). Since V[1] is known to be the vertex with the minimum x-coordinate (leftmost point), moving counter-clockwise to complete a cycle must first increase the x-coordinates and then after reaching a maximum (rightmost point), should decrease the x-coordinate back to that of V[1]. Thus, $V_x[1 : n]$ is a unimodal array, and so it is synonymous with detecting the location of the maximum element in the array.

Algorithm:

(1) If $n = 1$, return $V_x[1]$.

(2) If $n = 2$, return $\max\{V_x[1], V_x[2]\}$.

(3) $k \leftarrow \lceil n/2 \rceil$.

(4) If $V_x[k] > V_x[k - 1]$ and $V_x[k] > V_x[k + 1]$, then return $V_x[k]$.

(5) If $V_x[k] < V_x[k - 1]$ then call the algorithm recursively on $V_x[1 : k - 1]$, else call the algorithm recursively on $V_x[k + 1 : n]$.

Solution

(b). Let p denote the index at which the x-coordinate was maximized. Observe that joining V[1] and V[p] by a straight line divides the polygon into an upper half and a lower half and the vertex achieving the maximum y-coordinate must lie above this line. The counter-clockwise traversal $V_y[p] \rightarrow V_y[p+1] \rightarrow \cdots \rightarrow V_y[n] \rightarrow V_y[1]$ is unimodal and we need to detect the location of the maximum element of this array. So, considering the same algorithm as above with this new array $[V_y[p], V_y[p+1], \ldots, V_y[n], V_y[1]]$ will return the vertex with the maximum y-coordinate.

The recurrence equation is $T(n) \leq T(n/2) + \Theta(1)$.

Invoking Master’s Theorem gives $T(n) = O(\log n)$.

DIVIDE AND CONQUER

TA: Saumil Dedhia

Question 2

Each ship is located at an integer point on the sea represented by a cartesian plane, and each integer point may contain at most 1 ship.

You have a function `Sea.hasShips(topRight, bottomLeft)` which takes two points as arguments and returns `true` If there is at least one ship in the rectangle represented by the two points, including on the boundary.

Given two points: the top right and bottom left corners of a rectangle, return the number of ships present in that rectangle. It is guaranteed that there are at most 10 ships in that rectangle.

Example :

```
  ships = [[1,1],[2,2],[3,3],[5,5]], topRight = [4,4], bottomLeft = [0,0]
```

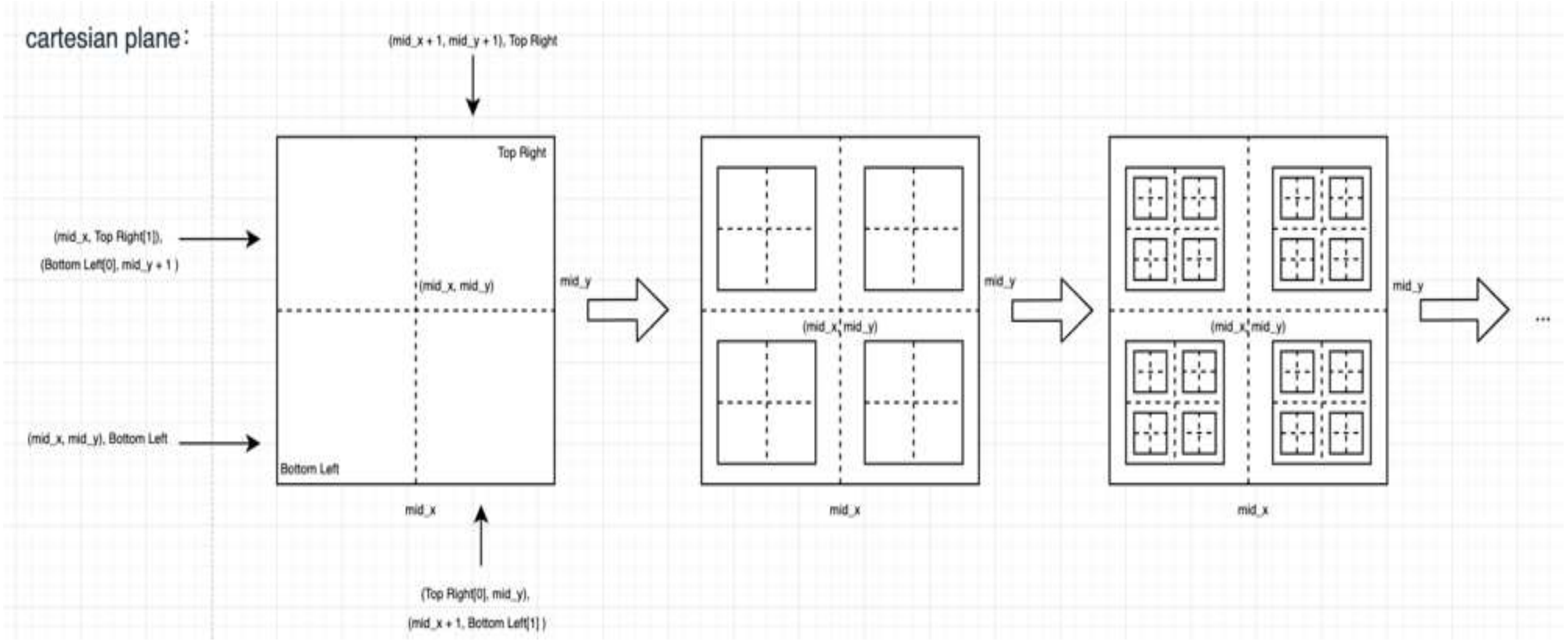
Algorithm should output = 3

Solution

Hint 1 : Divide the query rectangle into 4 rectangles.

Hint 2 : Use recursion to continue with the rectangles that has ships only.

Solution - Visualization



Solution

Step 1 : Base Condition:

1. if (!sea.hasShips(topRight, bottomLeft))

            return 0;

1. if (topRight[0] == bottomLeft[0] && topRight[1] == bottomLeft[1])

             return 1;

Step 2 : Divide

```
int midX = (topRight[0] + bottomLeft[0])/2;

int midY = (topRight[1] + bottomLeft[1])/2;
```

Solution

Step 3 : Conquer

```
countShips(sea, new int[]{midX, midY}, bottomLeft) +
countShips(sea, topRight, new int[]{midX+1, midY+1}) +
countShips(sea, new int[]{midX, topRight[1]}, new int[]{bottomLeft[0], midY+1}) +
countShips(sea, new int[]{topRight[0], midY}, new int[]{midX+1, bottomLeft[1]});
```

Time Complexity Analysis using Master’s Theorem

Time complexity: O(n) n is the side of the rectangle

T(n) = 4xT(n/2) + O(1)

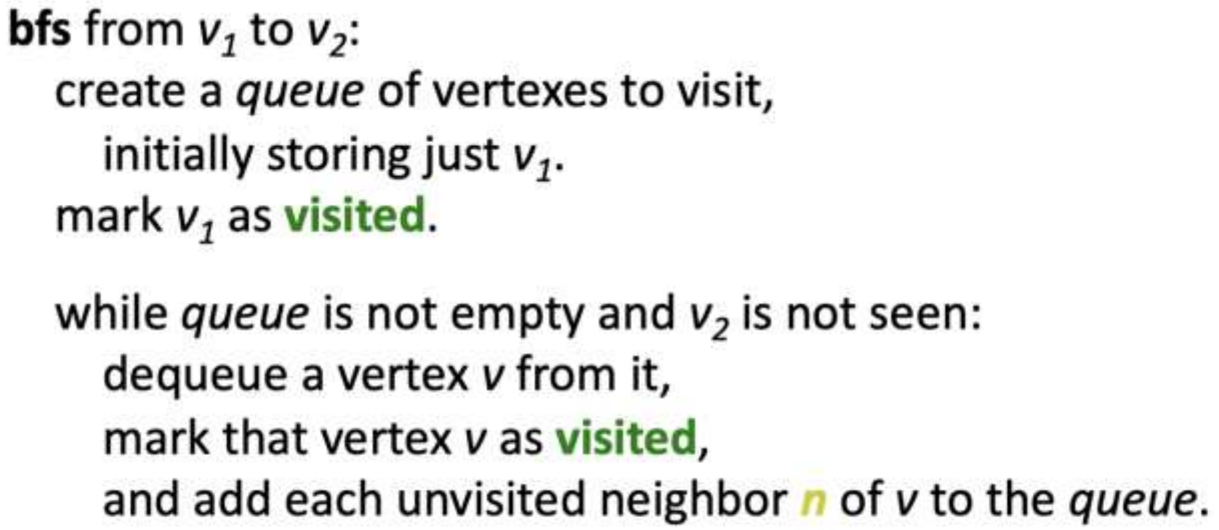
Apply master theorem: n^(log(4)4) = n^2 is O(O(1)). So T(n) = O(n^2)

Complexity is still the same but, we significantly improve the time complexity practically since we would not check any of the blocks that don’t have any ships.

BFS and DFS

TA: Adwaita Jadhav

BFS



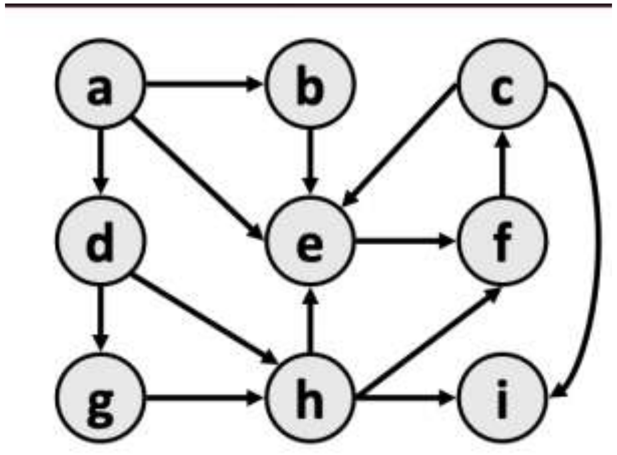
```
bfs from $v_1$ to $v_2$:
    create a queue of vertexes to visit,
        initially storing just $v_1$.
    mark $v_1$ as visited.

    while queue is not empty and $v_2$ is not seen:
        dequeue a vertex $v$ from it,
        mark that vertex $v$ as visited,
        and add each unvisited neighbor $n$ of $v$ to the queue.
```

Time Complexity: O(V+E)

Question: BFS traversal of following graph.

- a->b (since e is already visited we stop)
- a->d->g (since h is already visited we stop)
- ->h->i (since e and i are already visited)
- a->e->f->c



Some Observations

optimality:
 – always finds the shortest path (fewest edges).
– in unweighted graphs, finds optimal cost path.
– In weighted graphs, not always optimal cost.

retrieval: harder to reconstruct the actual sequence of vertices or edges in the path once you find it

– conceptually, BFS is exploring many possible paths in parallel, so it's not easy to store a path array/list in progress

– solution: We can keep track of the path by storing predecessors for each vertex (each vertex can store a reference to a previous vertex).

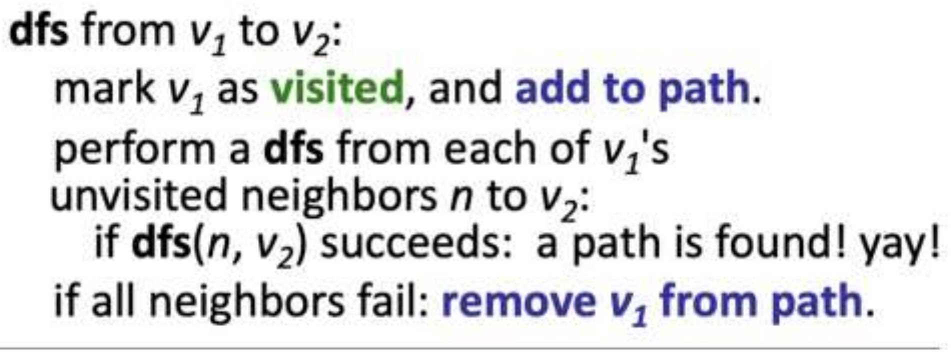
DFS

• Depth-first search (DFS): Finds a path between two vertices by exploring each possible path as far as possible before backtracking.

– Often implemented recursively.

– Many graph algorithms involve visiting or marking vertices.

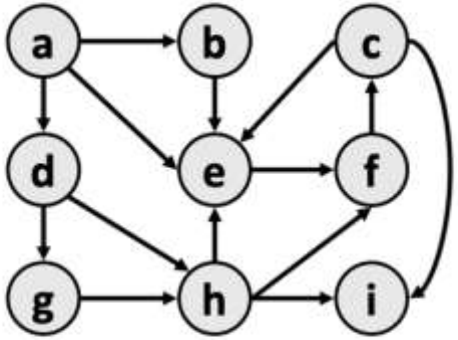
– Time Complexity: O(V+E)



```
dfs from $v_1$ to $v_2$:
    mark $v_1$ as visited, and add to path.
    perform a dfs from each of $v_1$'s
    unvisited neighbors n to $v_2$:
        if dfs(n, $v_2$) succeeds:  a path is found! yay!
    if all neighbors fail: remove $v_1$ from path.
```

Question: Find a path from a to h

- a->e->f->c->i
- Backtrack to c, no more unvisited neighbors
- Backtrack to f, no more unvisited neighbors
- Backtrack to e, no more unvisited neighbors
- Backtrack to a, univisited neighbors b and d
- a->d->g->h



Observation: we could also go from d->h and that would lead to shortest path. But DFS does not guarantee shortest path, but it guarantees if path exists or not.

Heaps

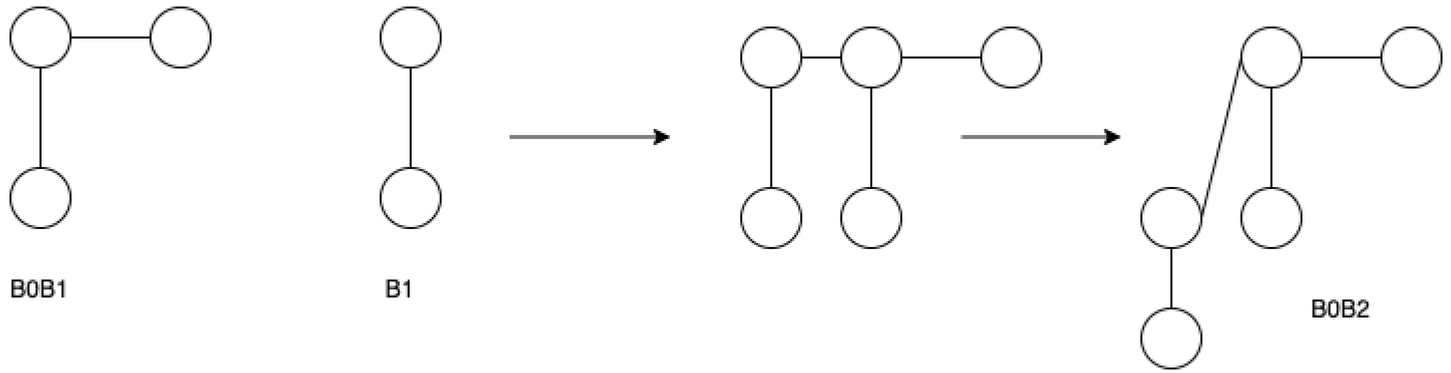
Rachitha Kagalvadi

Question 1

Merge two binomial heaps

B0B1B2B4 and B1B4

Example-



$11 + 10 = 101 \rightarrow$ B2B0

Solution

B0B1B2B4 → 10111

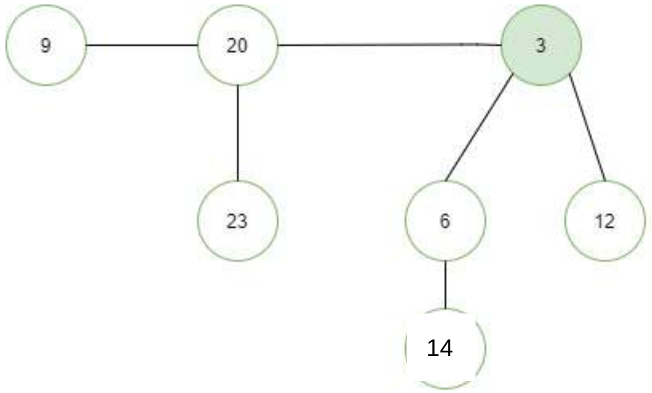
B1B4 → 10010

Perform Binary addition

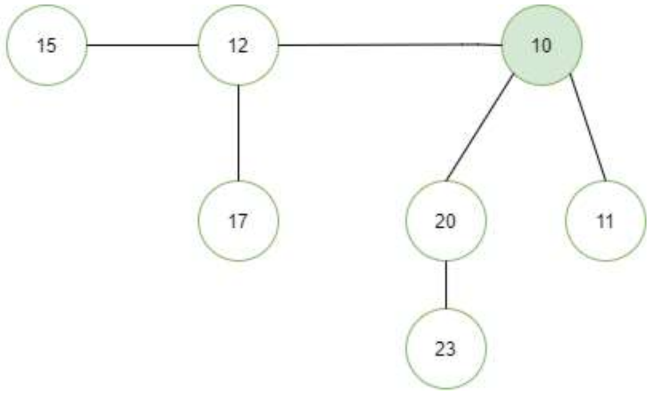
Result → 101001

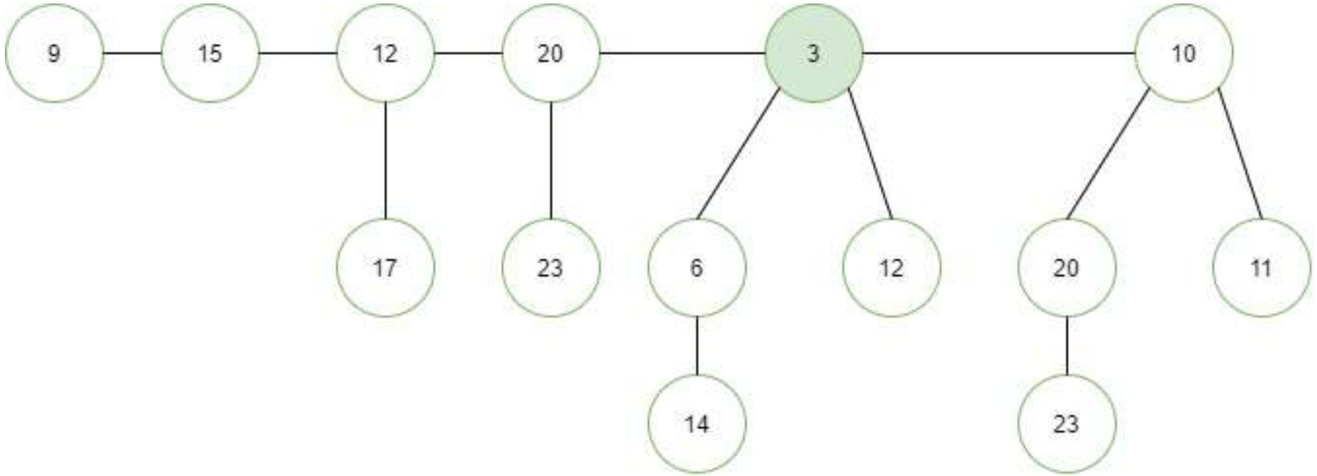
Heap after merging → B5B3B0

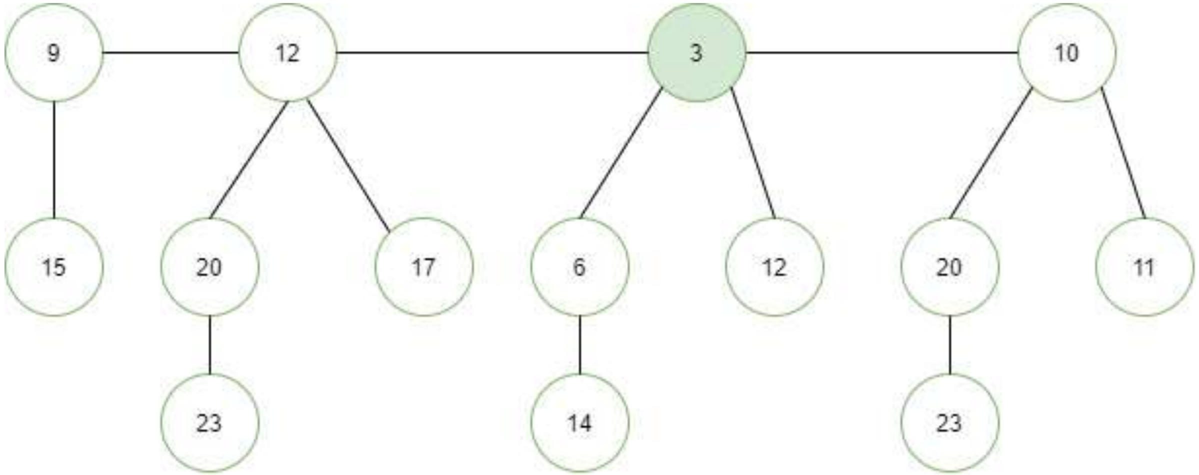
Question 2

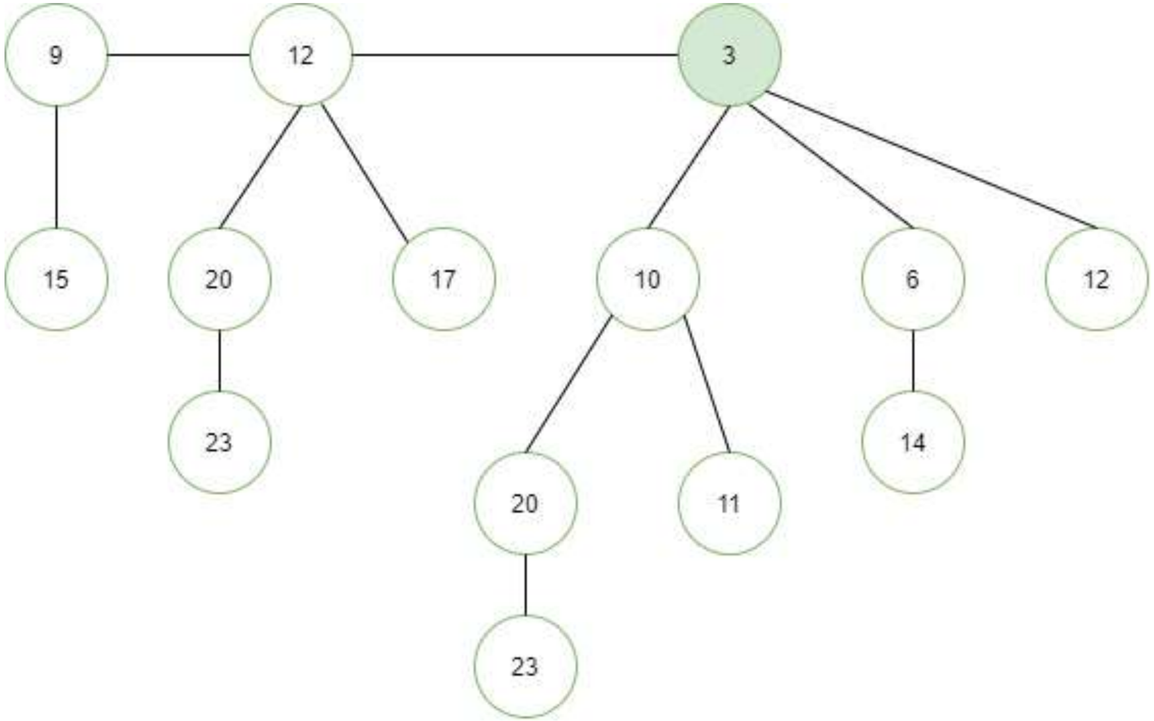


Merge with

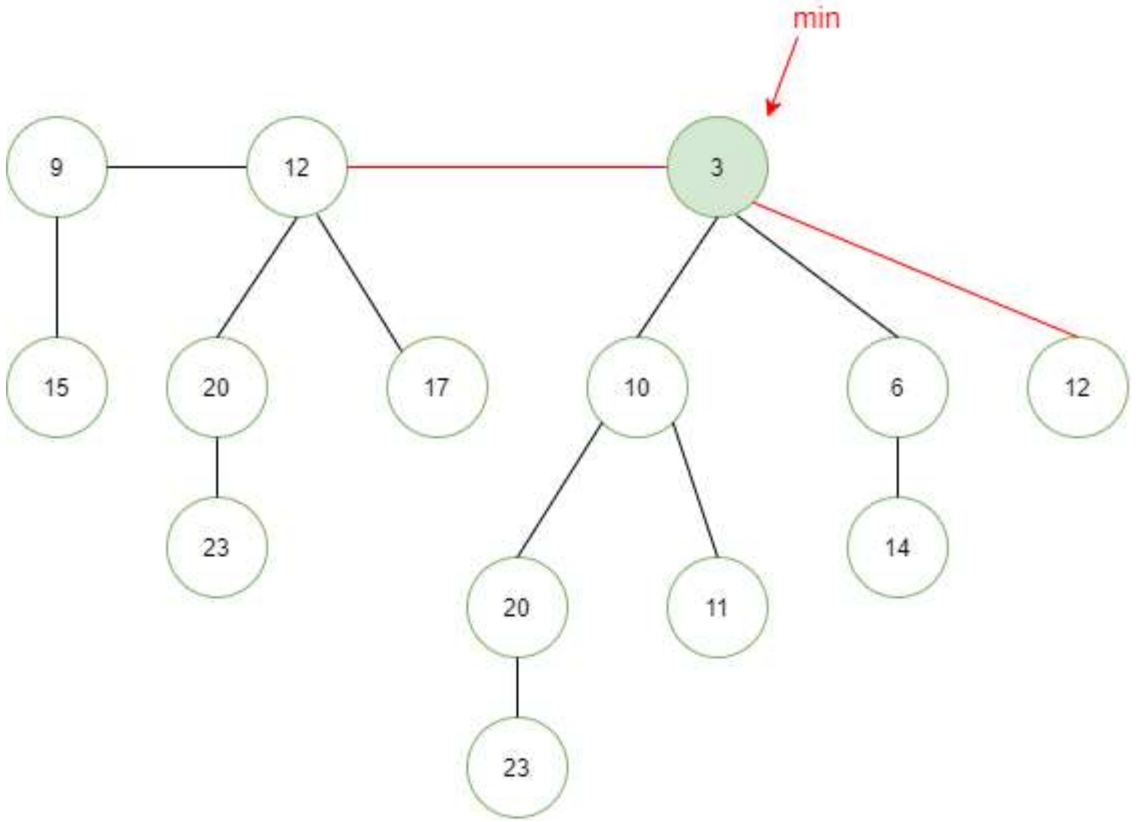


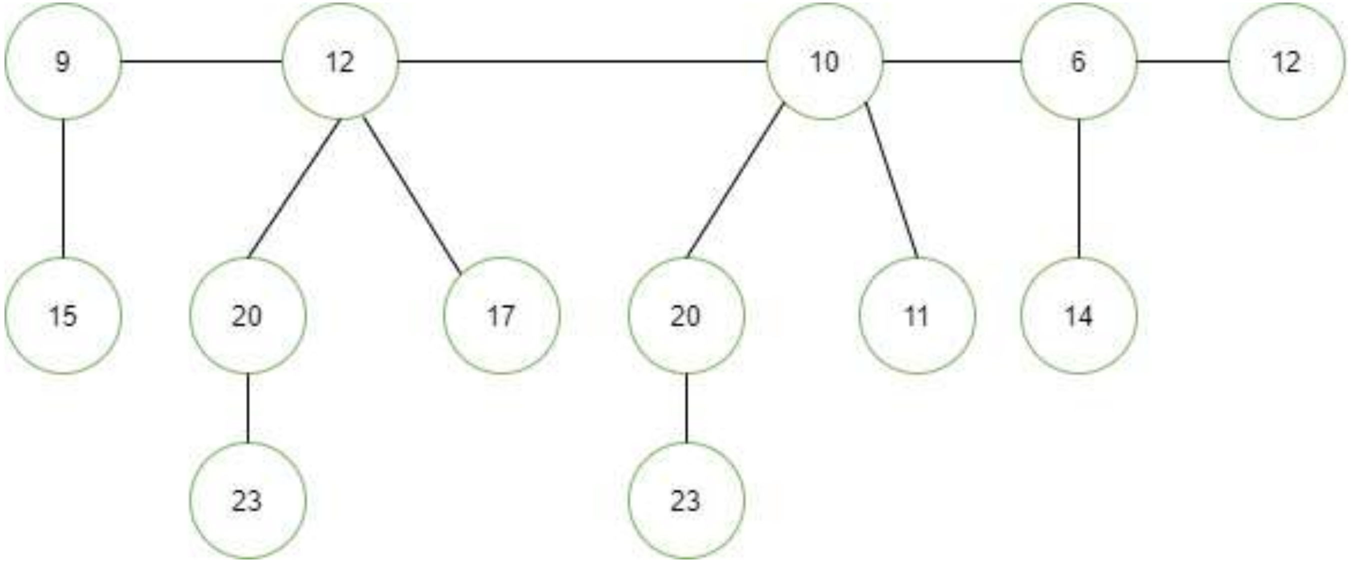

9
15
12
20
3
10
17
23
6
12
20
11
14
23

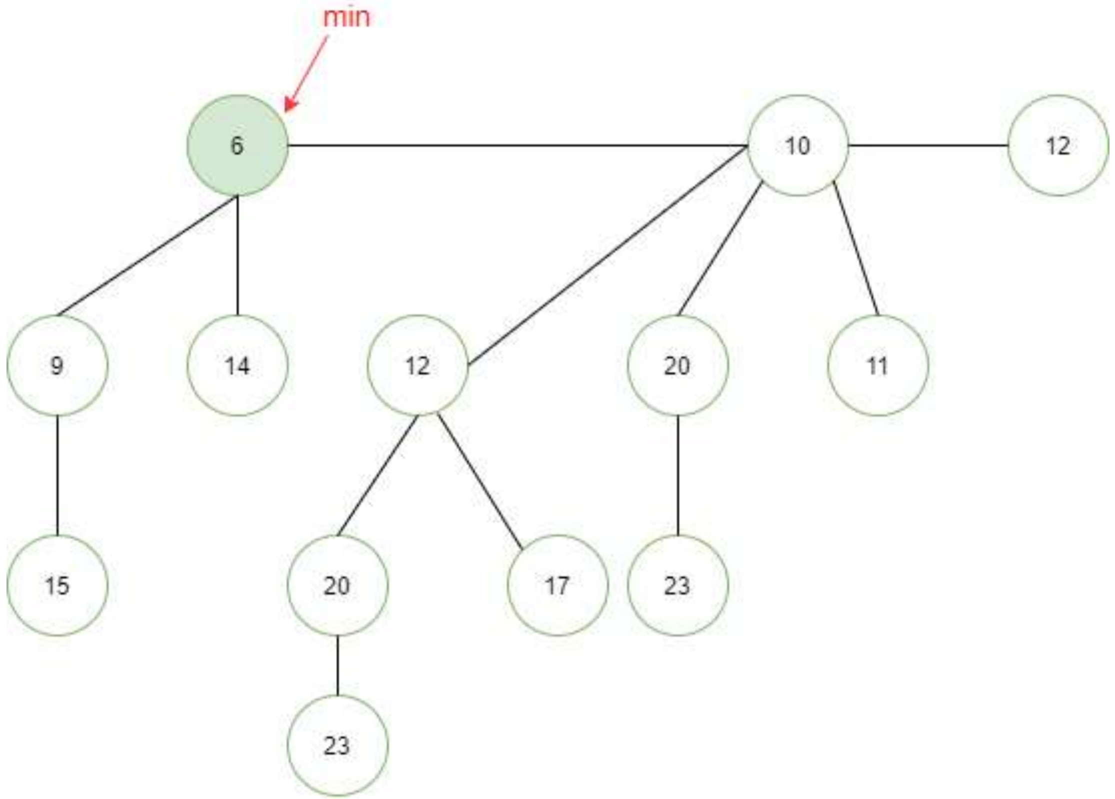

9
12
3
10
15
20
17
6
12
20
11
23
14
23


9
12
3
15
20
17
10
6
12
23
20
11
14
23

deleteMin()




9
12
10
6
12
15
20
17
20
11
14
23
23


min
6
10
12
9
14
12
20
11
15
20
17
23
23

Question 3 (T/F)

The smallest element in a binary max-heap of size n can be found with at most n/2 comparisons.

Solution

True. In a max heap, the smallest element is always present at a leaf node. So we need to check for all leaf nodes for the minimum value → how many leaf nodes are there?

ceil(n/2) leaf nodes. → Therefore, we will only have to do at most n/2 comparisons.