Exam 1 Review

By: Sajjad Shahabi & Omkar Thakoor

Stable Matching

We have a set M of *n* men and a set W of *n* women. Suppose all women have identical rankings. Propose a simpler variant of GS that does not have to break any engagements and computes a stable matching.

Hint: Consider the order of proposals.

Solution

Algorithm: Similar to GS, but the men will propose in the order of the women’s rankings (top-ranked man proposes first, bottom-ranked man proposes last).

Proof (no engagements broken): by Contradiction → There is a woman, w, who breaks an engagement by choosing a man, m’, over her previous engagement, m. This means that w must prefer m’ over m. However, since men propose in the order of the women’s rankings and m proposed before m’, m must be ranked higher than m’. Contradiction.

Proof (stable matching): by Contradiction → There exists an unmatched pair, (m, w') such that m prefers w' to w, his assigned partner, and w' prefers m to m', her assigned partner. m must have proposed to w’ before w. If w’ was single: she would have matched with m, and since she will not break the engagement, she cannot end up with m’. Else, w’ would have already been engaged with an m’’ > m > m’, and will not break the engagement with m’’. Contradiction.

Asymptotic

Problem

Give upper bound for the following code.

```
k = 1
for i = 1 to n
    j = 1
    while j <= i
        k = k + 1
        j = j * 2
```

Solution

Each inner loop i takes O(log(i))

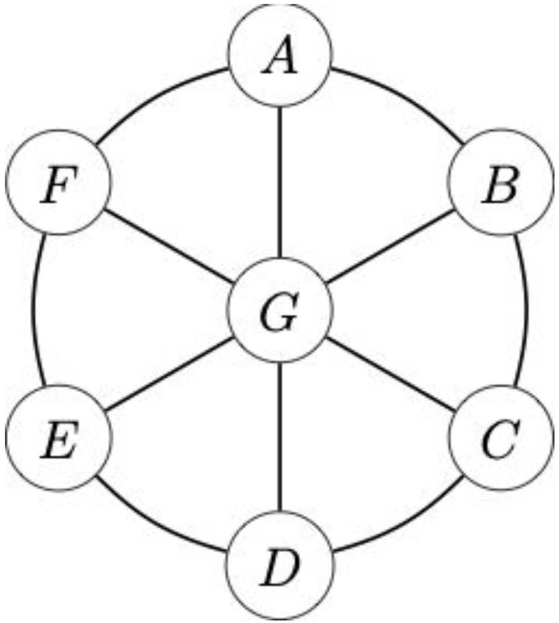
Total = Σ_i O(log(i)) = O(nlog(n))

```
k = 1
for i = 1 to n
    j = 1
    while j <= i
        k = k + 1
        j = j * 2
```

BFS/DFS & Shortest Paths

Problem 1

List the order in which vertices are visited when executing BFS and DFS starting from vertex E and breaking ties alphabetically.



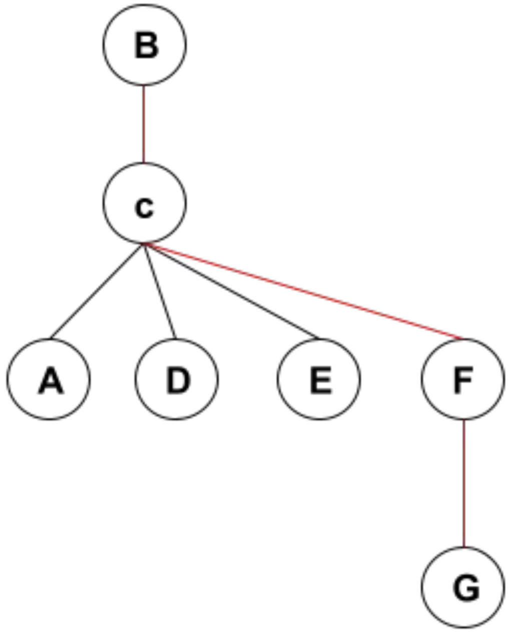
Answer:

BFS: *E, D, F, G, C, A, B*

DFS: E, D, C, B, A, F, G

Problem 2

- 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

- 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 3

Because of a bad update, you (the CTO of SpaceX) are in a hurry to fix the ongoing issue with the Starlink satellite. Your talented engineers have prepared the urgent fix, which needs to be sent to all the satellites as soon as possible.

The fix is first sent from the base station situated in California to satellite *S1*. From there, you can send the fix from satellite *Si* to a satellite *Sj* if there is a link (edge) between *Si* and *Sj*. You know the time required to send the fix over this link from satellite *Si* to satellite *Sj* is *Tij*. We can assume that if a satellite *Si* needs to send the fix to *k* of its neighbors the *k* messages can be sent out at the same time. In other words, messages from a satellite to its neighbors go out in parallel not sequentially. However, a satellite can send the fix to its neighbors only AFTER it has fully received it from a neighbor.

a) Provide an algorithm to determine the minimum time required to propagate the update to all satellites. In other words, we need to minimize the time between the first message leaving the base station and when the last satellite completes receiving the update

b) Assuming that each satellite requires a single processing time *Di* before broadcasting the fix to any nearby satellites that need to receive the fix, how would you modify the solution in part a to determine the minimum time required to propagate the fix to all satellites.

Solution 3

a) Run Dijkstra's on the undirected graph and find the shortest distance to the farthest node from the California base station.

b) Turn the satellite network into a directed graph G (one undirected edge -> two directed edges in opposite directions). For every (S_i, S_j) in E, let the edge weight be E_(S_i, S_j) = T_ij + D_i. Run Dijkstra's on this directed graph and find the shortest distance to the farthest node from the California base station.

Greedy

Problem 1

Given an array of distinct integers A = [a1, a2, …, an] 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

$|ai - pi| + |aj - pj| \leq |ai - pj| + |aj - pi|$

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

$|ai - pi| + |aj - pj| = d1 + d3 \leq d1 + 2*d2 + d3 = |ai - pj| + |aj - pi|$.

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.

Scheduling

Input:

Jobs $j_1, j_2, \ldots, j_n$.

Durations $t_1, t_2, \ldots, t_n$.

Penalties Per Hour $p_1, p_2, \ldots, p_n$.

Firstly, all jobs are due at t=0. Secondly, j_i has penalty p_i per hour.

Output:

What is an optimal order for tasks to minimize penalties?

Scheduling

Lemma:

Given jobs so that j_i takes time t_i with penalty p_i, there is an optimal schedule so that the first job is the one that maximizes the ratio p_i/t_i.

Scheduling

Lemma:

Given jobs so that j_i takes time t_i with penalty p_i, there is an optimal schedule so that the first job is the one that maximizes the ratio p_i/t_i.

Algorithm:

Pick the job that has the largest p_i/t_i.

Include it in your job list.

Repeat.

Scheduling

Algorithm:

Pick the job that has the largest p_i/t_i.

Include it in your job list.

Repeat.

Example:

Jobs j_1, j_2, j_3, j_4.

Times 3, 5, 2, 4.

Penalties 10, 7, 5, 2.

Scheduling

Algorithm:

Pick the job that has the largest p_i/t_i.

Include it in your job list.

Repeat.

Example:

Jobs j_1, j_2, j_3, j_4.

Times 3, 5, 2, 4.

Penalties 10, 7, 5, 2.

p_i/t_i 3.3, 1.4, 2.5, 0.5.

Scheduling

Algorithm:

Pick the job that has the largest p_i/t_i.

Include it in your job list.

Repeat.

Example:

Jobs j_1, j_2, j_3, j_4.

Times 3, 5, 2, 4.

Penalties 10, 7, 5, 2.

p_i/t_i 3.3, 1.4, 2.5, 0.5.

Output j_1, j_3, j_2, j_4.

Scheduling

A: t_A hours, p_A penalties per hour

B: t_B hours, p_B penalties per hour

A then B is better than B then A when:

$$t_A*p_A+(t_A+t_B)*p_B <= t_B*p_B+(t_B+t_A)*p_A$$

$$t_A*p_A+t_A*p_B+t_B*p_B <= t_B*p_B+t_B*p_A+t_A*p_A$$

$$t_A*p_B <= t_B*p_A$$

$$t_A/p_A <= t_B/p_B$$

$$p_B/t_B <= p_A/t_A$$

Scheduling

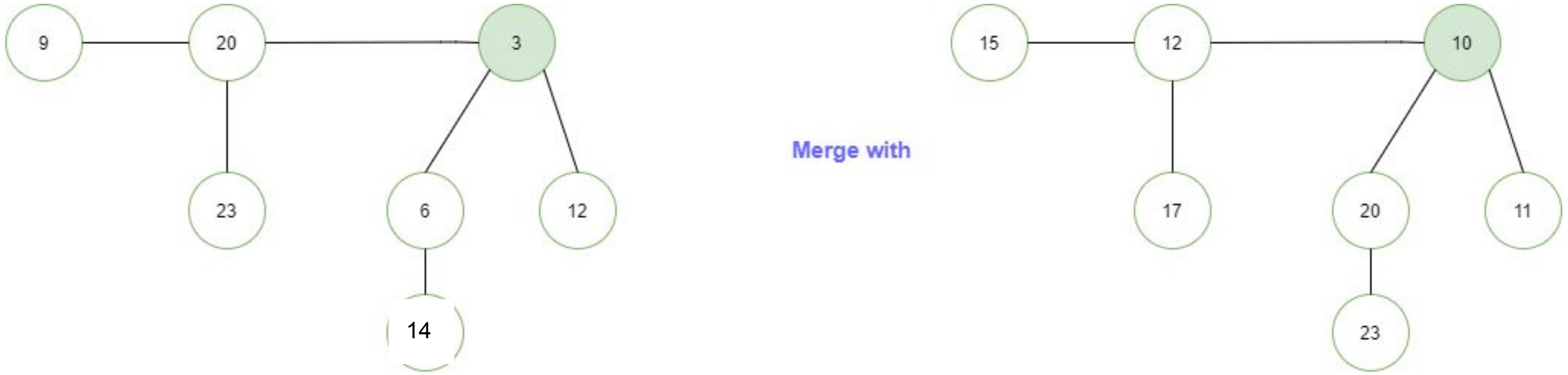
Optimal Solution: $j_1, j_2, j_3, j_4, \ldots$

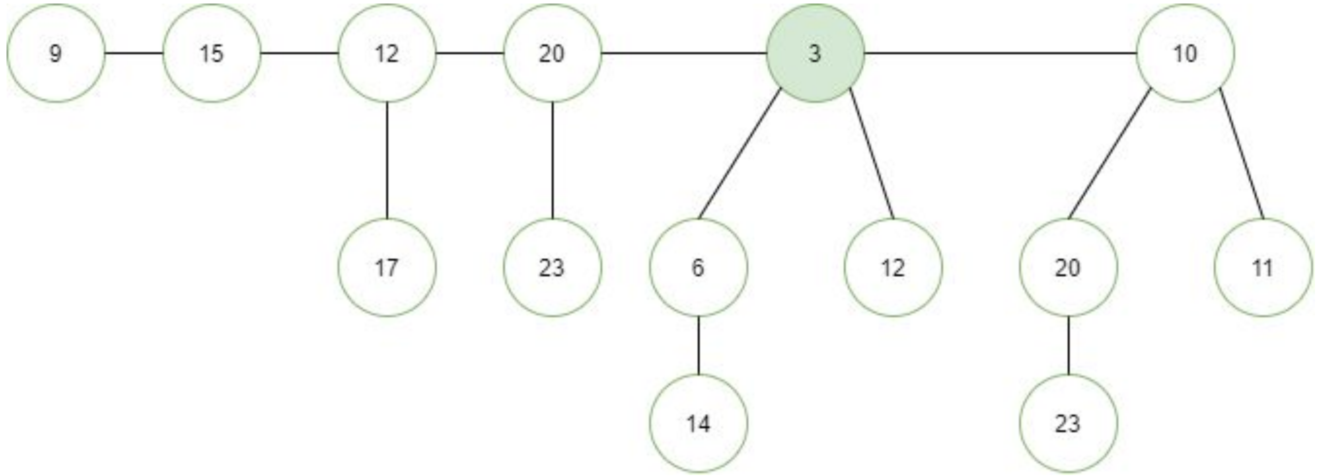
For all pairs j_i and j_{i+1} in the optimal solution, switch j_i and j_{i+1} if $p_i/t_i <= p_{i+1}/t_{i+1}$. The switch can only make the solution better. Repeat this procedure.

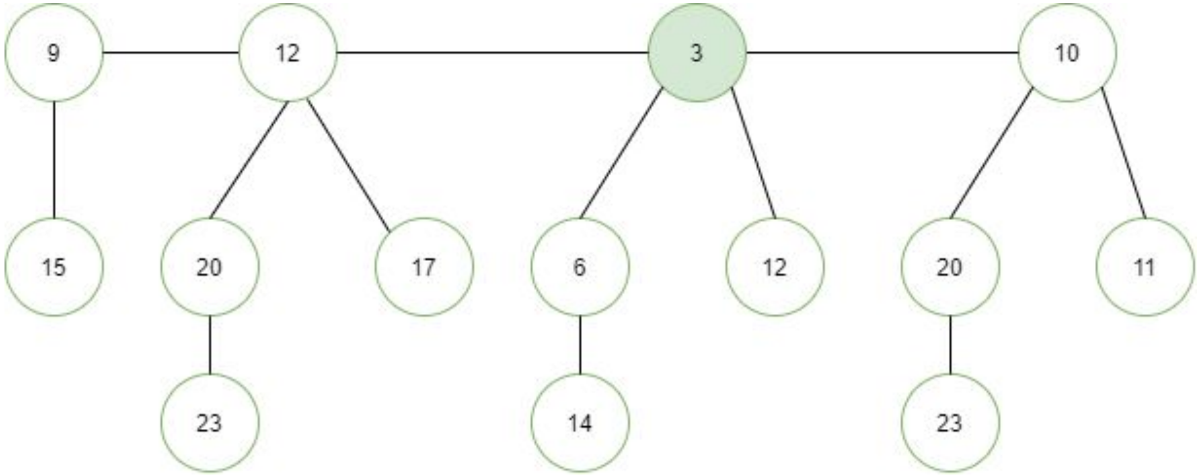
With this procedure, the first job in the new solution has the first highest p/t, and the second job in the new solution has the second highest p/t, ….

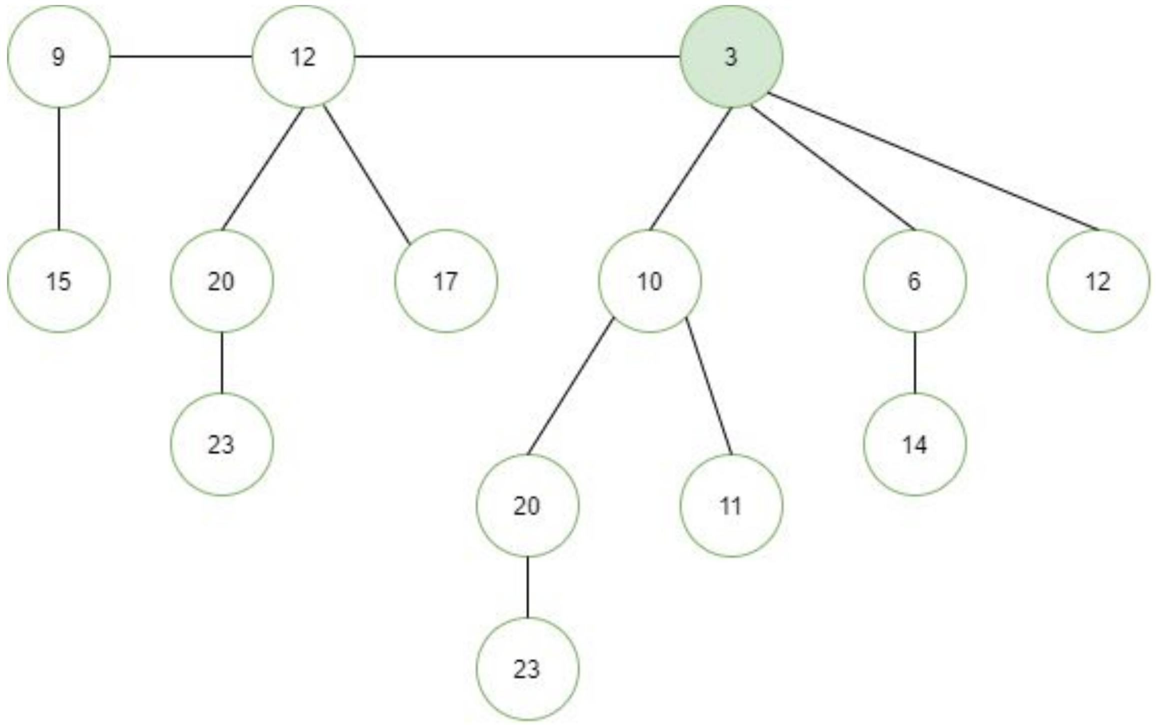
Heap

Question 1




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

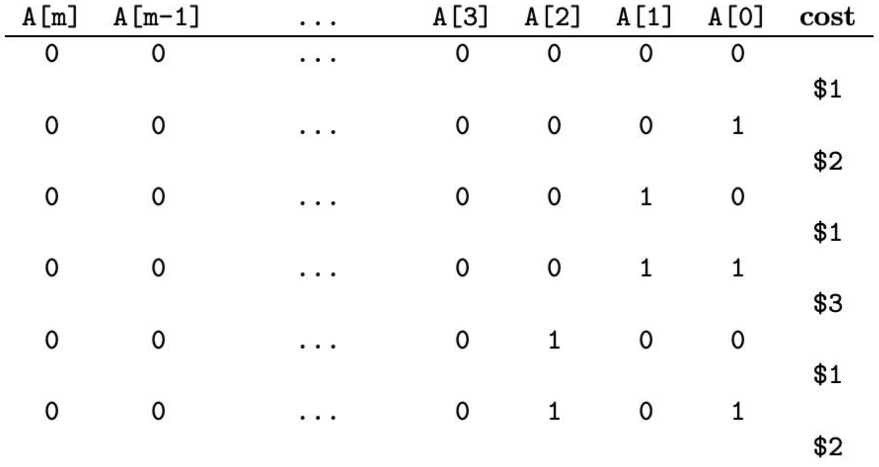

9
12
3
10
15
20
17
6
12
20
11
23
14
23


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

Amortized cost

Binary Counter

Imagine we want to store a big binary counter in an array A. Each element of the array holds a bit which is either marked 0 or 1. All the entries start at 0 and at each step we will be simply incrementing the counter. Let's say our cost model is: whenever we increment the counter, we pay $1 for every bit we need to flip.
For instance, here is a trace of the first few operations and their cost:
What is our amortized cost per increment?



A[m]	A[m-1]	...	A[3]	A[2]	A[1]	A[0]	cost
0	0	...	0	0	0	0	
							$1
0	0	...	0	0	0	1	
							$2
0	0	...	0	0	1	0	
							$1
0	0	...	0	0	1	1	
							$3
0	0	...	0	1	0	0	
							$1
0	0	...	0	1	0	1	
							$2

In a sequence of n increments, the worst-case cost per increment is O(log n), since at worst we flip lg(n) + 1 bits

Aggregate analysis

First, how often do we flip A[0]?
Answer: every time.

How often do we flip A[1]?
Answer: every other time.

How often do we flip A[2]? Answer: every 4th time, and so on.

So, the total cost spent on flipping A[0] is n,
the total cost spent flipping A[1] is at most n/2,
the total cost flipping A[2] is at most n/4, etc.

For n increments, we'll flip till log(n) bits

$n + n/2 + n/4 + \ldots. + 1 = O(2n)$

Summing these up, the total cost spent flipping all the positions in our n increments is at most 2n
for n increments.

Amortized cost = $2n/n = O(2) = O(1)$

Accounting method

Every time you flip 0 → 1, pay the actual cost of $1, plus put $1 into a piggy bank.
So the total amount spent is $2.

Now, every time you flip a 1 → 0, use the money in the bank to pay for the flip.

Clearly, by design, our bank account cannot go negative.

The key point now is that even though different increments can have different numbers of 1 → 0 flips, each increment has exactly one 0 → 1 flip.

So, we just pay $2 (amortized) per increment

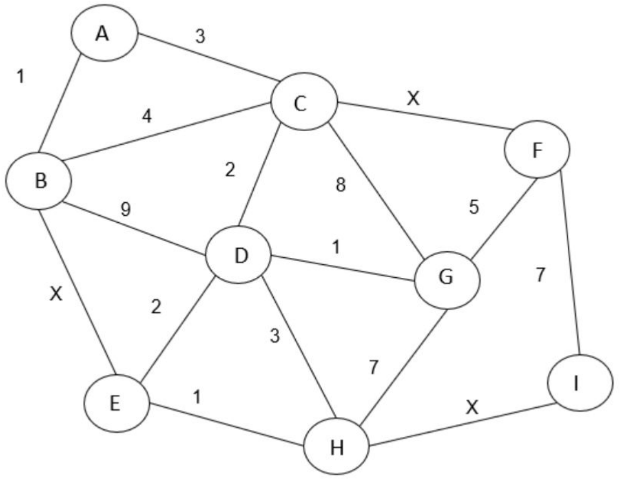
operation #
where coin is used

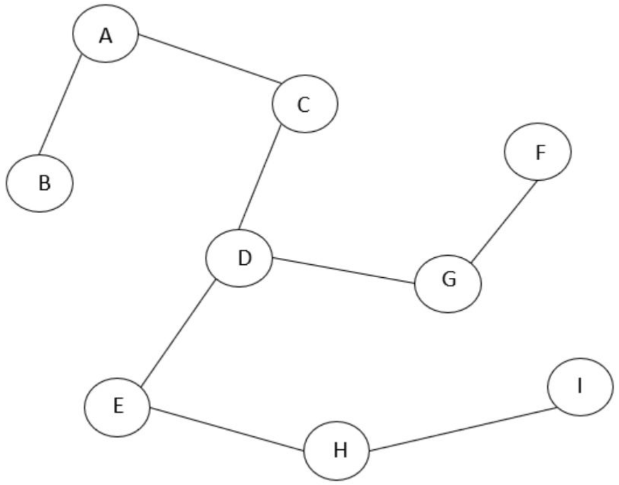
	5	4	3	2	1	0	①	②
oper^n 1						$\boxed{1}$	1	2
2					$\boxed{1}$	0	2	4
3					1	$\boxed{1}$	3	4
4				$\boxed{1}$	0	0	4	8
5				1	0	$\boxed{1}$	5	6
6				1	$\boxed{1}$	0	6	8
7				1	1	$\boxed{1}$	7	8
8			$\boxed{1}$	0	0	0	8	16
9			1	0	0	$\boxed{1}$	9	10
10			1	0	$\boxed{1}$	0.	10	12
11			1	0	1	$\boxed{1}$	11	12
12			1	$\boxed{1}$	0	0	12	16
13			1	1	0	$\boxed{1}$	13	14
14			1	1	$\boxed{1}$	0	14	16
15			1	1	1	$\boxed{1}$	15	16
16		$\boxed{1}$	0	0	0	0	16	

MST

Problem 1

Consider the weighted undirected graph *G* on the left (see below graphs). Suppose the graph *T* on the right is the unique MST of *G.* Find the value of X, assuming X is an integer. You must provide the reasoning for your answer.





Solution 1

1. If $X < 5$ then FG will not be the lowest connection cost between F and the rest of the MST. So, $X \geq 5$. But if $X=5$ then the MST will not be unique since FC can replace FG without affecting the cost of the MST. So it must be that $X > 5$.

2. If $X > 7$ then FI will be the lowest connection cost between I and the rest of the MST, the MST will include the edge FI instead of HI. So, $X \leq 7$. But if $X=7$ then the MST will not be unique since FI can replace HI without affecting the cost of the MST. So it must be that $X < 7$.

3. Since edge costs are integer and $X < 7$ and $X > 5$ then $X=6$.

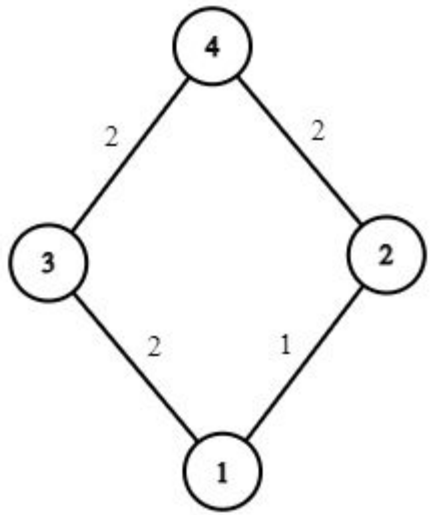
4. Arguments that use a specific MST algorithm are also acceptable.

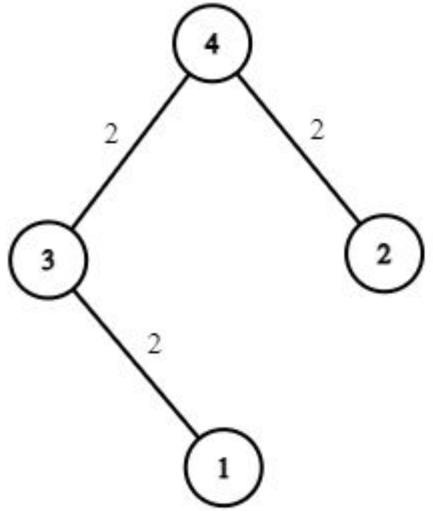
Problem 2

Consider the Minimum Spanning Tree Problem on an undirected graph $G = (V, E)$, with a cost $c_e \geq 0$ on each edge, where the costs may not all be different. If the costs are not all distinct, there can in general be many distinct minimum-cost solutions. Suppose we are given a spanning tree $T \subseteq E$ with the guarantee that for every $e \in T$, e belongs to some minimum-cost spanning tree in G. Can we conclude that T itself must be a minimum-cost spanning tree in G? Give a proof or a counterexample with explanation.

Solution 2

Let’s look at this graph. Is the second graph an MST for the first graph? Note that all of its edges belong to an MST, but each MST must include the edge (1,2)





Problem 3

There are n houses in a village. We want to supply water for all the houses by building wells and laying pipes.

For each house i, we can either build a well inside it directly with cost wells[i], or pipe in water from another well to it. The costs to lay pipes between houses are given by the array pipes where each pipes[j] = [house1j, house2j, costj] represents the cost to connect house1j and house2j together using a pipe. Connections are bidirectional, and there could be multiple valid connections between the same two houses with different costs.

Return *the minimum total cost to supply water to all houses*.

Solution 3

Create a graph where each house is represented by a node. For every pipes[j], we create an edge between house1j and house2j with edge weight as costj.

We also create a node representing the well and connect to each house such that the edge weight of the edge between the new node and the ith house node is wells[i]

Find the MST in this graph and return the cost of the tree.

Divide & Conquer, Master theorem

Find Peak Element

A peak element is an element that is strictly greater than its neighbors.

Given a 0-indexed array nums of distinct integers, find a peak element, and return its index. If the array contains multiple peaks, return the index to any of the peaks.

You may imagine that nums[-1] = nums[n] = -∞. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.

nums[i] != nums[i + 1] for all valid i.

You must write an algorithm that runs in O(log N) time.

Solution

Let N be the size of nums.

First, there’s at least one peak element in this problem. Let’s prove it.

Since nums[-1] = nums[*N*] = -∞, we know that nums[-1] < nums[0], nums[N-1] > nums[N]. If there’s no peak, since nums[-1] < nums[0], we know that nums[0] < nums[1] must hold. And similarly, we will get nums[1] < nums[2], nums[2] < nums[3], …, nums[N-1] < nums[N]. But we have known that nums[N-1] > nums[N]. We reached a contradiction.

Solution

Split the array in half and compare the two middle elements nums[i] and nums[i+1]: (i = n/2)

```
    If nums[i] > nums[i+1]:
        Then a peak is within [0, i] elements
    If nums[i] < nums[i+1]:
        Then a peak is within [i+1, n] elements
```

Repeat the same process in the selected half

Solution

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

$$\log_b a = \log_2 1 = 0 \Rightarrow f(n) = O(1) = \Theta(n^0) = \Theta(1)$$

Master’s theorem second case:

$$T(n) = \Theta(\log n)$$

Q2: find *x* in array

- Consider a two-dimensional array A[1:n,1:n] of integers. In the array each row is sorted in ascending order and each column is also sorted in ascending order. Our goal is to determine if a given value x exists in the array.

Design a divide-and-conquer algorithm to solve this problem, and state the runtime of your algorithm.

Your algorithm should take strictly less than O(n^2) time to run, and should make use of the fact that each row and each column is in sorted order (i.e., don't just call binary search on each row or column). State the run-time complexity of your solution.

Q2: find *x* in array

Let m be the middle element of the full matrix.

If x == m return True

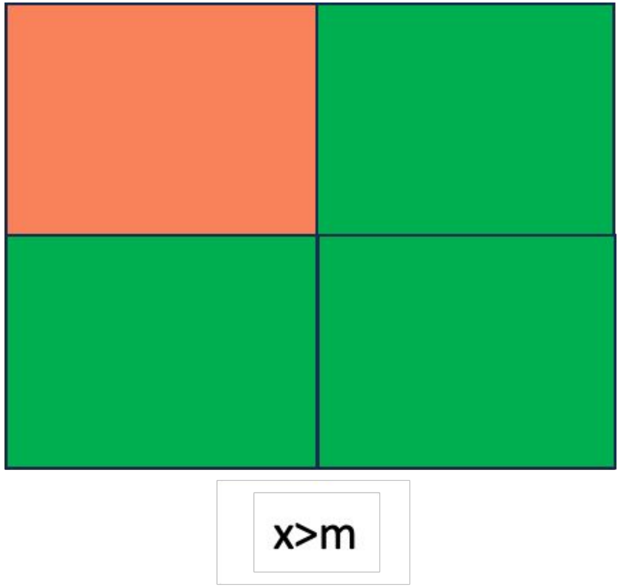
1. If x < m:

   Eliminate A[$\frac{n}{2}..n, \frac{n}{2}..n$]

2. If x > m:

   Eliminate A[1..$\frac{n}{2}$,1..$\frac{n}{2}$]

recursively search in the remaining three $\frac{n}{2} \times \frac{n}{2}$



Q2: find *x* in array

Master Theorem

T(n) = $3T(\frac{n}{2})$+O(1)

f(n) = 1

a = 3, b =2, $log_b^a = log_2^3 \Rightarrow n^{log_b^a} = n^{log_2^3}$

Case 1 of Master Theorem → f(n) = $O(n^{log_2^3})$

T(n) = $\theta(n^{\log_2 3})$

Q2: find *x* in array

Alternate solution:

Let m be the top-right-corner element. Compare x with m

- If x==m, return the element (Base case)
- If m>x: Eliminate rightmost column since each element there would also be bigger than x.
- If m<x: Eliminate topmost row since each element there would also be smaller than x.

The new subproblem has one less row/column. The recurrence is
$T(r) = T(r-1) + O(1)$, where r = #rows+columns.
This simplifies to $O(n)$ since initially $r = 2n$ (n rows + n columns)