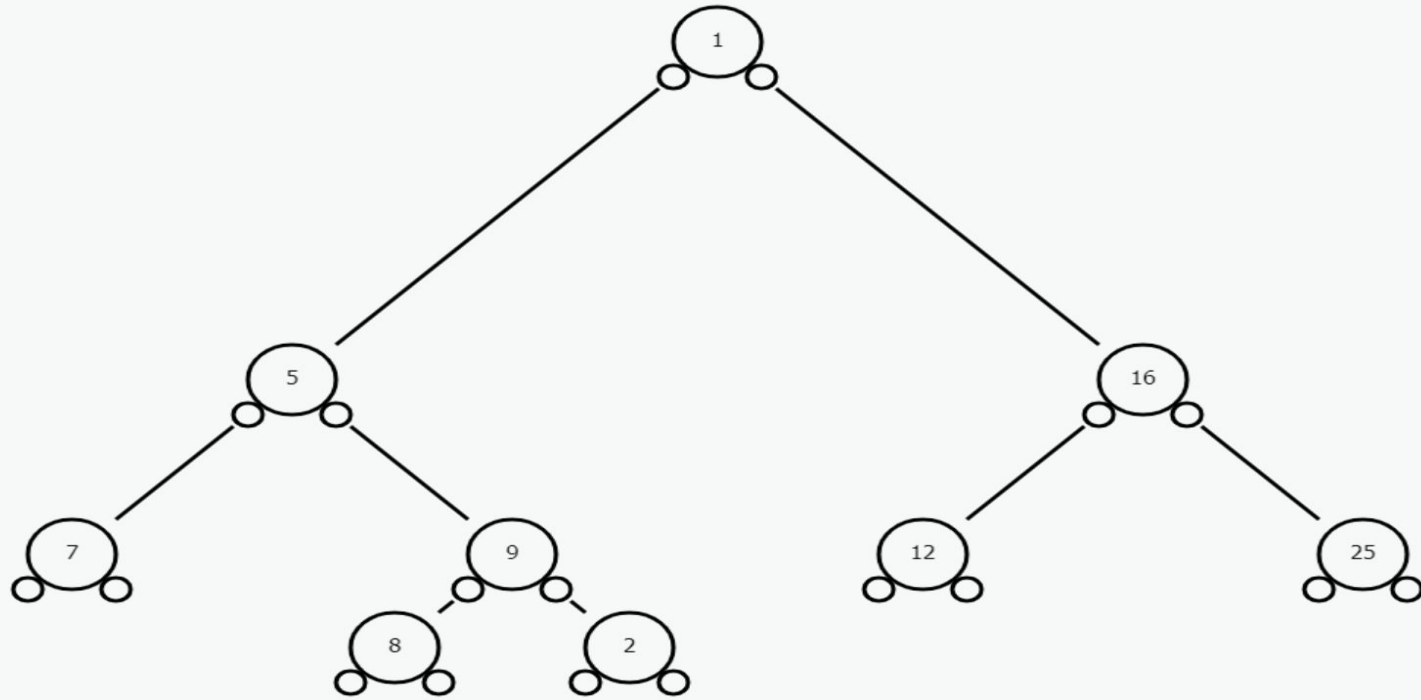


Asymptotic analysis and BFS/DFS

Question 1

We are given a Binary Tree T representing hierarchy. Let us assume each node in the binary tree represents a family member. Now, given any two random nodes, we need to find their first common ancestor. Design the most efficient algorithm.



Made using <https://binary-tree-builder.netlify.app/>

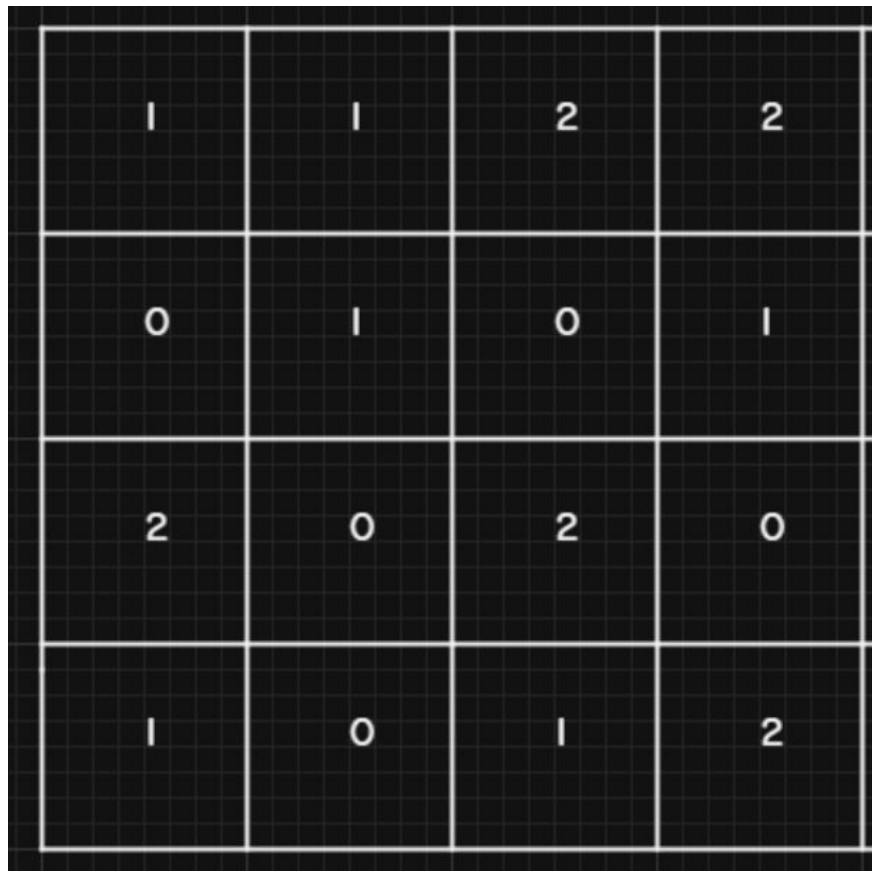
Solution

- 1) We start by checking the base case. If root is null, then we simply return Null, or if root is equal to either one of two nodes (let them be p and q), then return root.
- 2) Recursively traverse the left subtree and right subtree and store their values.
- 3) If we find leaf node, where both children are NULL, return NULL.
- 4) If left and right are not NULL, then return root.
- 5) If one of right or left is NULL, then return the other one.

Time complexity: $O(n)$ as we traverse the tree.

Question 2

We are given a Grid of size $n \times m$. Some cells contain a person stressed due to the 570 exam. Each second a stressed person makes other happy people around them stressed (4-dimensionally) by giving them reminders that the exam is next week. Let a stressed person be represented by 2, a happy person by 1 and 0 represents someone who just doesn't care. Design an efficient algorithm to return the minimum number of minutes until every happy person is stressed. If not possible return -1.



1	1	2	2
0	1	0	1
2	0	2	0
1	0	1	2

Made using <https://reallysketch.com/app/>

Solution

Given the size of the grid, we can loop through it to find all the 2's. So, create an empty queue and insert all the 2 coordinates into it. Let this queue be called q.

We also need to keep track of all the 1's in variable count. If we do not find any 1's then the answer is 0, Mark all the 2's as visited.

Now follow the below steps until q is empty:

- 1) Find the current size of q;
- 2) For each iteration (<size) pop the front element of the q and search its 4 directions, of course while checking that those lie within our range, i.e. ≥ 0 and $< n/m$ and they are not 0 and are not visited;
- 3) This means we are left with only 1's. Add those in the queue and mark them as visited. We also need to check if the count of 1's we have encountered so far is same as total 1's. If it's true then return the iteration number (here iteration number is the number of times we have iterated the entire q).

In the end just return -1, as 2's could never stress all the 1's.

Time complexity: $O(n*m)$ as we traverse the grid.

Question 3

Arrange the following in increasing order:

$\log(n)$, $2^{(3n)}$, $3^{(2n)}$, $n^{(n^2)}$, $n^{(n\log(n))}$, $2^{(\log(n))}$, $2^{n\log n}$, $n^{\log n}$.

First 5 have base e, last 3 have base 2.

Solution

$$\log_e(n) = \log_e(n)$$

$$2^{(3n)} = 8^n$$

$$3^{(2n)} = 9^n$$

$$n^{(n^2)} = n^{(n^2)}$$

$$n^{(n \cdot \log_e(n))} = n^{(n \cdot \log_e(n))}$$

$$2^{(\log_2(n))} = n$$

$$2^{(n \log_2(n))} = n^n$$

$$n^{(\log_2(n))} = 2^{(\log_2(n))^2}$$

Answer:

$$\log_e(n), 2^{(\log_2(n))}, n^{(\log_2(n))}, 2^{(3n)}, 3^{(2n)}, 2^{(n \log_2(n))}, n^{(n \cdot \log_e(n))}, n^{(n^2)}$$

Question 4

If $f(n) = O(n)$ and $g(n) = O(n)$, then $f(g(n)) = \theta(n^2)$
T/F?

Source: Fall'23 Midterm.

Solution

False

Take $f(n) = g(n) = \text{any constant}$

SHORTEST PATH

PRACTICE QUESTIONS

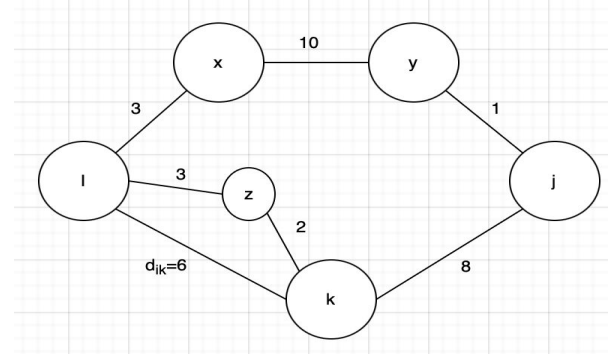
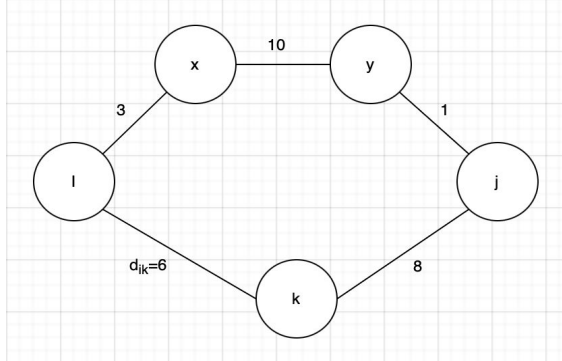
MCQ

Suppose the shortest path from node i to node j goes through node k and that the cost of the subpath from i to k is D_{ik} . Consider these two statements:

- I. Every shortest path from i to j must go through k .
- II. Every shortest path from i to k has cost D_{ik} .

- A. I and II are both true.
- B. Only I is true.
- C. Only II is true.
- D. I and II are both false.

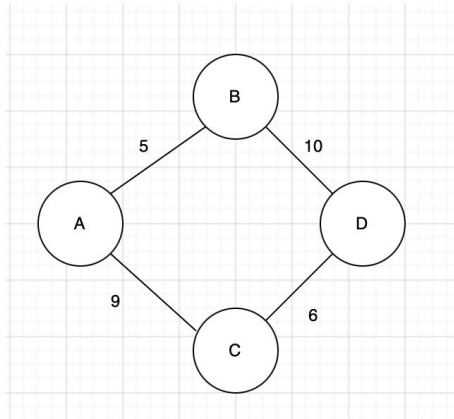
Option C



True or False ?

If all edges in a connected undirected graph have distinct positive weights, the shortest path between any two vertices is unique.

FALSE



True or False ?

Dijkstra's algorithm may not terminate if the graph contains negative-weight edges, i.e. the iterations may continue forever.

FALSE

- Even in graph with negative weight edges, Dijkstra's Algorithm will terminate.
- But it is possible that the shortest path found using Dijkstra's Algorithm may not be correct.

Solve the following Question

You want to go from node s to node t in a connected undirected graph $G(V,E)$ with positive edge costs. You would also like to stop at node u if it is possible to do so without increasing the cost of your path by more than a factor of k .

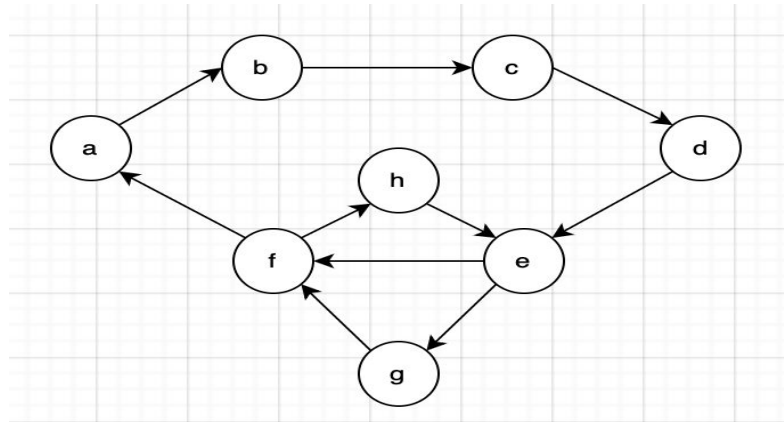
Design an efficient Algorithm to find the optimal path of going from node s to node t considering your preference of stopping at node u if possible.

Solution

- Run Dijkstra's Algorithm from node s to find the shortest path to node u [$d(su)$] and node t [$d(st)$]
- Also run Dijkstra's Algorithm from node u to find shortest path to node t [$d(ut)$]
- Compare the value of $d(su) + d(ut)$ with $d(st)$. If it is less than a factor of k then the path will contain node u else it will not
- Since we are running Dijkstra's Algorithm twice, total run time of the algorithm is $O(|E| \log |V|)$

Solve the following Question

Consider a positively weighted directed graph. Design the most efficient algorithm you can for finding a minimum weight simple cycle in the graph. Be sure to prove that your algorithm is correct and find its running time.



Solution

Consider a minimum weight simple cycle and consider any edge $t \rightarrow s$ within it. The portion of the cycle from s to t is a simple path and it must be of minimum cost: if it were not then one could concatenate the better path from s to t with the edge $t \rightarrow s$ and obtain a cycle of lesser cost, a contradiction.

So we have shown that it certainly suffices to compute the shortest path from s to each vertex t and then check the cost of the cycle this path makes with the edge $t \rightarrow s$ (if it exists). By checking over all possible sources s and all edges to s we are guaranteed of seeing a minimum cycle at least once.

Solution

PSEUDO CODE :

$\text{best} \leftarrow \infty$

For each vertex s do

{

Run Disjkstra's algorithm from s

For each edge $t \rightarrow s$ do

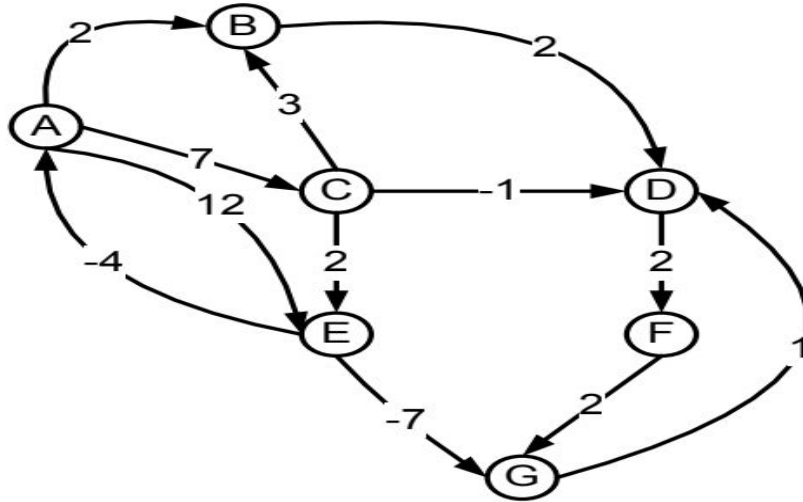
$\text{best} \leftarrow \min(\text{best}, \text{weight}(t \rightarrow s) + \text{dist}[t])$

}

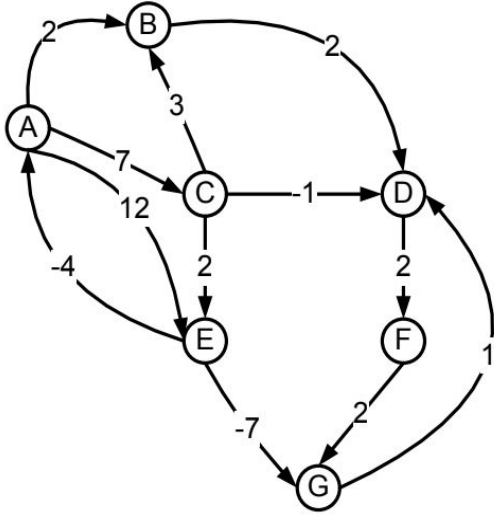
The algorithm clearly takes $O(VE \log V)$ time if one uses the heap implementation of Dijkstra,

Solve the following Question

Consider the following graph. Find the shortest path from Node A to all other nodes using Dijkstra's Algorithm.



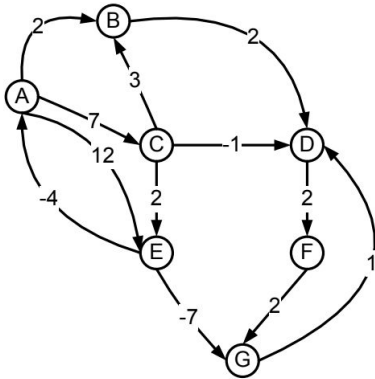
Solution



Nodes	Distance	Path
A	0	A
B	2	A-B
D	4	A-B-D
F	6	A-B-D-F
C	7	A-C
G	8	A-B-D-F-G
E	9	A-C-E

Solve the following Question

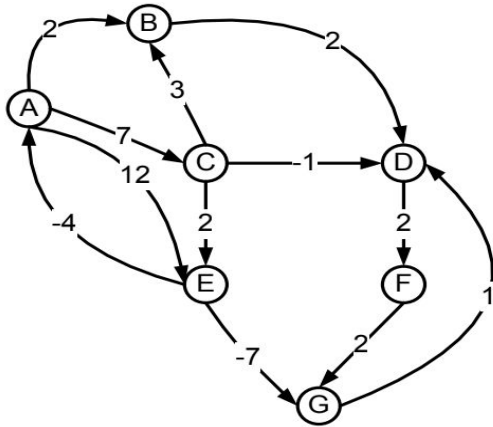
Dijkstra's algorithm found the wrong path to some of the vertices. For just the vertices where the wrong path was computed, indicate both the path that was computed and the correct path.



Nodes	Distance	Path
A	0	A
B	2	A-B
D	4	A-B-D
F	6	A-B-D-F
C	7	A-C
G	8	A-B-D-F-G
E	9	A-C-E

Solution

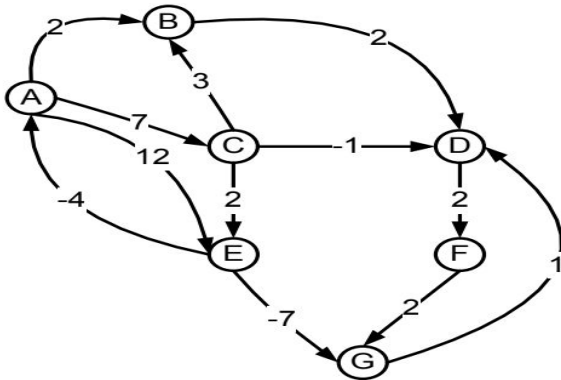
Computed path to G is A,B,D,F,G but shortest path is A,C,E,G (2).
Computed path to D is A,B,D but shortest path is A,C,E,G,D (3).
Computed path to F is A,B,D,F but shortest path is A,C,E,G,D,F (5).



Nodes	Distance	Path
A	0	A
B	2	A-B
D	4	A-B-D
F	6	A-B-D-F
C	7	A-C
G	8	A-B-D-F-G
E	9	A-C-E

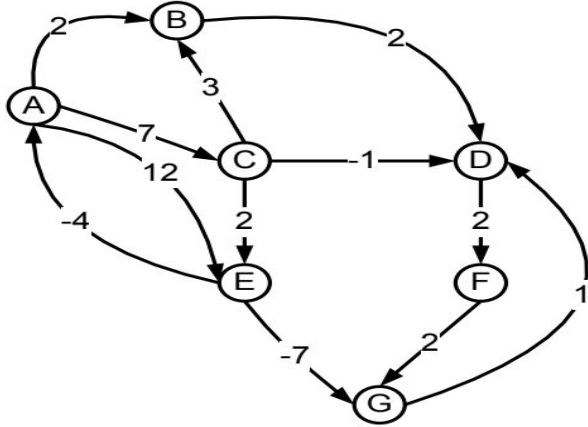
Solve the following Question

List the minimum number of edges that could be removed from the graph so that Dijkstra's algorithm would happen to compute correct answers for all vertices in the remaining graph.



Solution

Removing the edge from E to G will allow Dijkstra's algorithm to compute correct shortest path for all vertices in the remaining graph.



Greedy

Amirmohammad Nazari

Scheduling

Input:

Jobs j_1, j_2, \dots, j_n .

Durations t_1, t_2, \dots, t_n .

Penalties Per Hour p_1, p_2, \dots, 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:

$$t_A * p_A + (t_A + t_B) * p_B \leq t_B * p_B + (t_B + t_A) * p_A$$

$$t_A * p_A + t_A * p_B + t_B * p_B \leq t_B * p_B + t_B * p_A + t_A * z$$

$$x * p_B \leq t_B * p_A$$

$$t_A / p_A \leq t_B / p_B$$

$$p_B / t_B \leq p_A / t_A$$

Scheduling

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

For all pairs j_i and j_{i+1} in the optimal solution, switch j_i and j_{i+1} if $p_i/t_i \leq p_{i+1}/t_{i+1}$. 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 ,

Activity Selection

Input:

Activities a_1, a_2, \dots, a_n .

Rewards r_1, r_2, \dots, r_n .

Deadlines d_1, d_2, \dots, d_n .

Firstly, each activity takes one day. Secondly, you will lose r_i if you pass d_i .

Output:

What is an optimal order for activities to maximize rewards?

Activity Selection

❖ Algorithm:

❖ $\text{today} = \max(\text{deadlines})$

❖ For today, choose the activity that has the largest reward between activities with deadline greater than or equal to today.

❖ $\text{today} = \text{today} - 1$

❖ Repeat.

Activity Selection

Input:

Activities a_1, a_2, a_3, a_4 .

Rewards 9, 8, 5, 6.

Deadlines 2, 2, 4, 4.

Activity Selection

Input:

Activities a_1, a_2, a_3, a_4 .

Rewards 9, 8, 5, 6.

Deadlines 2, 2, 4, 4.

Output:

Day 4: a_4

Activity Selection

Input:

Activities a_1, a_2, a_3, a_4 .

Rewards 9, 8, 5, 6.

Deadlines 2, 2, 4, 4.

Output:

Day 4: a_4

Day 3: a_3

Activity Selection

Input:

Activities a_1, a_2, a_3, a_4 .

Rewards 9, 8, 5, 6.

Deadlines 2, 2, 4, 4.

Output:

Day 4: a_4

Day 3: a_3

Day 2: a_1

Activity Selection

Input:

Activities a_1, a_2, a_3, a_4 .

Rewards 9, 8, 5, 6.

Deadlines 2, 2, 4, 4.

Output:

Day 4: a_4

Day 3: a_3

Day 2: a_1

Day 1: a_2

Activity Selection

Optimal Solution: $a_1, a_2, a_3, a_4, \dots, a_n$

Assume that i is the last day of deadlines:

Optimal Solution: $a_1, a_2, a_3, a_4, \dots, a_i, \dots, a_n$

Assume A has the highest reward between activities with the deadline greater than or equal to i :

Optimal Solution: $a_1, a_2, a_3, a_4, \dots, A, \dots, a_i, \dots, a_n$

Optimal Solution: $a_1, a_2, a_3, a_4, \dots, a_i, \dots, A, \dots, a_n$

Switch a_i and A . The switch can only make the solution better.

Activity Selection

Now, let's go one step back:

Optimal Solution: $a_1, a_2, a_3, a_4, \dots, a_{i-1}, A, \dots, a_n$

Now, assume B has the highest reward between activities with the deadline greater than or equal to $i-1$:

Optimal Solution: $a_1, a_2, a_3, a_4, \dots, B, \dots, a_{i-1}, A, \dots, a_n$

Optimal Solution: $a_1, a_2, a_3, a_4, \dots, a_{i-1}, A, \dots, B, \dots, a_n$

Switch a_{i-1} and B. The switch can only make the solution better.

Repeat this procedure.

Minimum Spanning Tree

Question 1

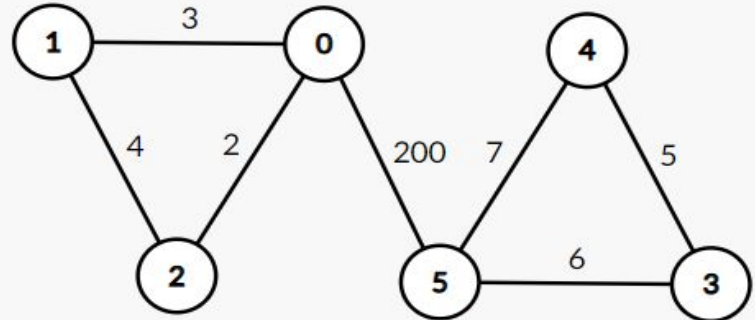
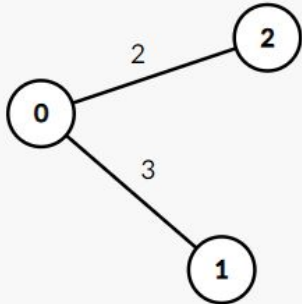
True or False

In an undirected, connected, weighted graph with at least three vertices and unique edge weights, the heaviest edge in the graph cannot be in any minimum spanning tree.

True or False

In an undirected, connected, weighted graph with at least three vertices and unique edge weights, the heaviest edge in the graph cannot be in any minimum spanning tree.

False: A tree is also a connected graph. The MST for a tree is the tree itself. Another example would be where the heaviest edge is the only one connecting 2 components.



Question 2

True or False

If all edge weights of a given graph are the same, then every spanning tree of that graph is minimum.

True or False

If all edge weights of a given graph are the same, then every spanning tree of that graph is minimum.

Solution. True.

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

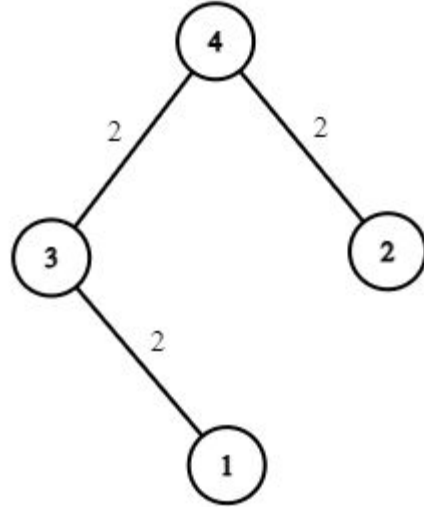
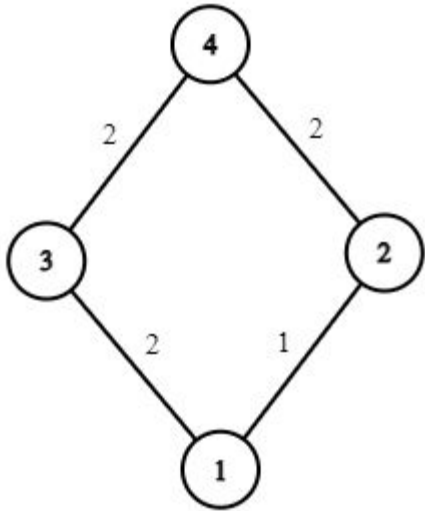
Question 3

MST

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.

MST - Solution

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)



Question 4

MST

Suppose you have n objects with defined distances $d(u, v)$ between each pair of them. $d(u, v)$ may be an actual distance, or some abstract representation of how dissimilar two objects are. Your task is to group these objects in such a way that objects within a single group have small distances between them, and objects across groups have larger distances between them. This problem is called clustering. Propose an algorithm to separate given n objects into $k \leq n$ clusters (groups), so that the minimum distance between items in different groups is maximized.

MST

Run Kruskal's but stop when you have k components.

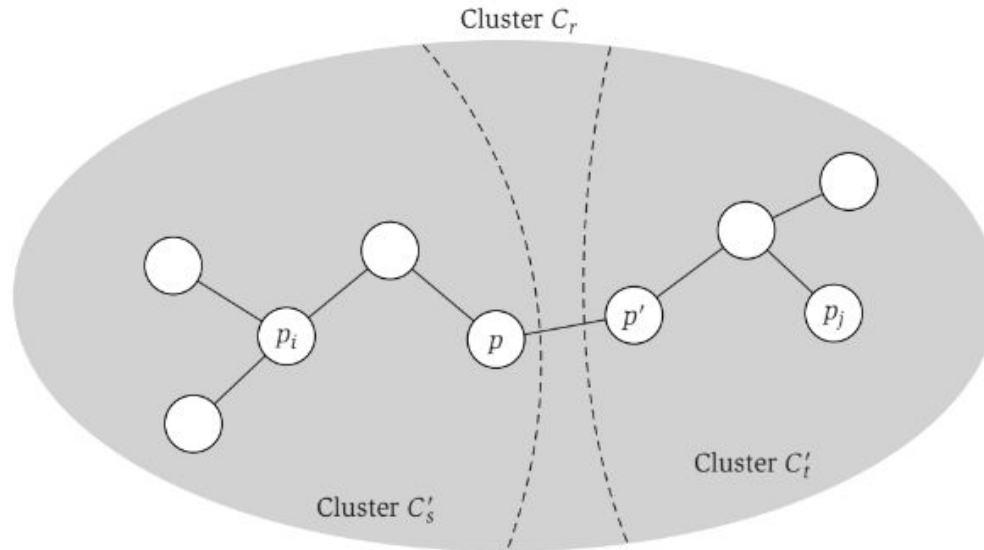
So we basically run Kruskal's till we have added $n-k$ edges.

This works because all the individual components have smaller edge-weights than the edges that weren't considered to be in the first $n-k$ edges.

We can also look at the clustering as the k components we would get after removing the $k-1$ most expensive edges from the MST.

Let C_r be one of the clusters given by our k -clustering algorithm, and let C_s' and C_t' be 2 clusters given in a different k -clustering. Let p_i and p_j be points that belong to C_s' and C_t' respectively, and they also belong to C_r given by our algorithm. Since p_i and p_j belong to the same cluster C_r given by our algorithm, there definitely is a path from p_i to p_j in the component generated by our algorithm, and all of these edges have a length of at most that of the $(k-1)^{\text{th}}$ most expensive edge in the tree generated by Kruskal's on the graph.

Let p and p' be 2 nodes on the path such that p belongs to C_s' and p' belongs to C_t' . Since the distance between these 2 points is at most that of the $(k-1)^{\text{th}}$ most expensive edge, the partitioning between them can't be better than the one given by our algorithm.



Amortized Analysis

Question 1: Calculate amortized time complexity of Incrementing a Binary Counter. Consider implementing a k-bit binary counter that counts upwards from 0.

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	1	1	0
7	0	0	0	0	0	1	1	1
8	0	0	0	0	1	0	0	0

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Figure 17.2 An 8-bit binary counter as its value goes from 0 to 16 by a sequence of 16 INCREMENT operations. Bits that flip to achieve the next value are shaded. The running cost for flipping bits is shown at the right. Notice that the total cost is always less than twice the total number of INCREMENT operations.

Solution:

The key to analyzing the amortized time complexity of the INCREMENT operation is to observe the frequency at which each bit in the counter is flipped:

- The least significant bit ($A[0]$) is flipped in every operation, resulting in n flips over n operations.
- The second least significant bit ($A[1]$) flips every alternate operation, resulting in $n/2$ flips over n operations.
- Similarly, the i th bit is flipped $n/2^i$ times over n operations.

Summing up the flips for all bits yields a total number of flips that is less than $2n$ for n INCREMENT operations. This can be seen by summing the geometric series $\sum (n/2^i)$ for i from 0 to $k-1$, which converges to a value less than $2n$. Hence, the amortized cost per operation, is $O(n)/n = O(1)$.

HEAPS

Question: You are given an array of tasks, representing the tasks a CPU needs to execute. Each letter in the array represents a different task. Tasks can be performed in any order, and each task takes one unit of time to complete. However, tasks are subject to a cooldown period defined by a non-negative integer n , meaning there must be at least n units of time between identical tasks. The CPU can be idle during this cooldown period.

Given the tasks and the cooldown period n , your goal is to determine the minimum number of units of time required for the CPU to finish all tasks.

Example 1:

Input: tasks = ["A", "A", "A", "B", "B", "B"], $n = 2$

Output: 8

Explanation:

A -> B -> idle -> A -> B -> idle -> A -> B

There is at least 2 units of time between any two same tasks.

Solution:

1. Populate the map with the frequency of each task
2. Create a max heap to store task frequencies
3. Loop until the heap is not empty
4. One cycle is $n+1$ iterations long.
5. Make a temporary list to store tasks executed in the current cycle
6. Increment time counter by the number of tasks popped from the heap in the current cycle.
7. Decrement frequencies by 1 and added back to heap if the frequency exceeds 0.
8. For the last loop avoid adding the cool-down period

```

class Solution {
public:
    function leastInterval(tasks: list of characters, n: integer) -> integer:
        // Initialize a histogram for task frequencies
        initialize h[26] with 0
        // Populate the histogram with the frequency of each task
        for each task t in tasks:
            h[t - 'A'] += 1

        // Create a max heap to store task frequencies
        create a max heap

        // Add all non-zero frequencies to the heap
        for i from 0 to 25:
            if h[i] > 0:
                heap.push(h[i])

        // Initialize the answer (total units of time)
        initialize ans with 0

        // Loop until the heap is empty
        while heap is not empty:
            // Temporary list to store tasks executed in the current cycle
            create a list v of size n+1 with 0

            // Index for tasks in the current cycle
            initialize i with 0

            // Fill the cycle with tasks from the heap
            for i from 0 to n:
                if heap is empty:
                    break
                v.push_back(heap.top())
                heap.pop()

            // Try to re-add tasks with decremented frequencies back to the heap
            for each frequency i in v:
                if i > 1:
                    heap.push(i - 1)

            // If the heap is empty, adjust the answer with the actual number of tasks executed
            // in the last cycle, otherwise, a full cycle of n+1 units was completed
            if heap is not empty:
                ans += n + 1
            else:
                ans += actual number of tasks executed in the last cycle

        // Return the total units of time
        return ans
}

```

Divide & Conquer, Master Theorem

Chenghao Wang

True or False

Now we have $T(n) = 2T(n/2) + n + \log(n) + \sqrt{n}$, then $T(n) = O(n \log n)$

True or False

Now we have $T(n) = 2T(n/2) + n + \log(n) + \sqrt{n}$, then $T(n) = \theta(n \log n)$

Answer:

True.

Since $n + \log(n) + \sqrt{n} = \theta(n)$, $T(n) = 2T(n/2) + O(n)$, according to master theorem, $a = 2$, $b = 2$, $f(n) = \theta(n^1)$, we can get $T(n) = \theta(n \log n)$.

Find Peak Element

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

Given a 0-indexed integer array `nums`, 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 $\text{nums}[-1] = \text{nums}[n] = -\infty$. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.

$\text{nums}[i] \neq \text{nums}[i + 1]$ 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 $\text{nums}[-1] = \text{nums}[N] = -\infty$, we know that $\text{nums}[-1] < \text{nums}[0]$, $\text{nums}[N-1] > \text{nums}[N]$. If there's no peak, since $\text{nums}[-1] < \text{nums}[0]$, we know that $\text{nums}[0] < \text{nums}[1]$ must hold. And similarly, we will get $\text{nums}[1] < \text{nums}[2]$, $\text{nums}[2] < \text{nums}[3]$, ..., $\text{nums}[N-1] < \text{nums}[N]$. But we have known that $\text{nums}[N-1] > \text{nums}[N]$. We reached a contradiction.

Count Reverse Pair

Given an array of integers *nums*, count the number of pairs (i, j) that $i < j$ and $\text{num}[i] > \text{num}[j]$. You should use Divide and Conquer to solve this problem and the time complexity should be $\theta(N \log N)$.

Tips: Consider counting the number of valid pairs when merging two parts in Merge Sort.

Solution

We can count the number of valid pairs when performing Merge Sort.

The merge part in Merge Sort selects the smaller of the first elements from two sorted sequences each time (if they are equal, it selects the one from the left sequence) and places it at the end of a new sequence. If the selected element is from the right sequence, then the remaining elements in the left sequence are exactly those greater than it. At this point, we add the count of remaining elements in the left sequence to the answer.

Solution (Continued)

The time complexity is the same as Merge Sort.

We have $T(n) = 2T(n/2) + \theta(n)$

According to the master theorem, $a = 2$, $b = 2$, $f = \theta(n^1)$, the time complexity for our algorithm is $\theta(N\log N)$.