EXAM

Confidential

THE UNIVERSITY OF SYDNEY

Faculty of Engineering and IT
School of Information Technologies
COMP 2007: Algorithms and Complexity

Student Details (to be filled in by the candidate)	
seat number	
full name	
other names you use	
SID	

Exam information and instructions:

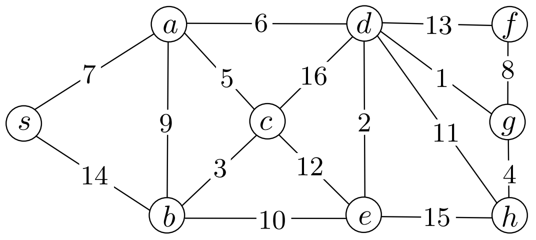
- 10 minutes reading time, 2:30 hours exam
- the paper comprises 11 pages
- you are not allowed to use any electronic devices
- this exam paper must not be removed from exam room

Results (for office use only)						
Question 1	Question 2	Question 3	Question 4	Question 5	Question 6	Total
/10	/10	/10	/10	/10	/10	/60

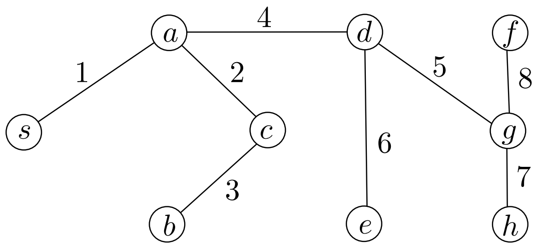
EXAM

- **Question 1 (10 points): Graphs**

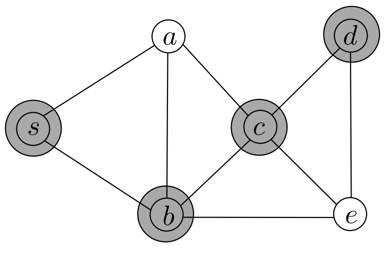
(a) Draw a minimum spanning tree of the weighted graph using Prim's algorithm, starting at node s. Indicate the order in which the algorithm adds the edges to the solution.



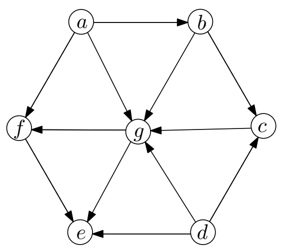
Answer:



(b) Show a minimum vertex cover of the below graph.



(c) Draw a topological order of the below graph.



Answer:

a,b,d,c,g,f,e or
any similar order where d comes before g

- **Question 2 (10 points): Interval scheduling**

(a) Describe a greedy algorithm that solves the interval scheduling problem. We have a set of requests $\{1, \ldots, n\}$; the ith request corresponds to an intervalof time starting at $s(i)$ and finishing at $f(i)$. We say that a subset of the requests is compatible if no two of them overlap in time. The goal is to accept asa large a compatible subset as possible.

Solution: Sort the request with respect to increasing finishing time. Use a simple greedy algorithm to process the requests in the sorted order. Once the first request i_1 is selected, reject all requests that are not compatible with i_1. Then select the next request i_2, remove all requests that are not compatible with i_2. Continue in this fashion until all requests have been handled. The rule for picking the next request is to select the request that has the smallest finishing time.

(b) Argue why the greedy algorithm outputs a correct solution.

Solution: Let A be the solution produced by the greedy algorithm, and let $i_1, \ldots, i_k$ be the set of requests in A in the order they were added to A. Similarly let OPT be an oiptimal solution, and let $j_i, \ldots, j_m$ be the set of requests in OPT ordered from left-to-right.
If $k = m$ then we are done, hence, assume $m > k$. Assume that OPT and A are identical up to index r (note that r could be 0), that is, index $r + 1$ is the leftmost index where OPT and A selected different jobs. The greedy algorithm will pick the job with the earliest finishing time, thus $f(i_{r+1}) \leq f(j_{r+1})$. This observation allows us to make a simple exchange argument. Since $f(j_r) = f(i_r)$ we can exchange j_{r+1} in OPT with the request i_{r+1} to get OPT'. And since $f(i_{r+1}) \leq f(j_{r+1})$ we can guarantee that OPT' is at least as good as OPT. As a result we now have an optimal solution OPT' that is equal to A up until index $r + 1$. We can continue this argument for every index iteratively which shows that A cannot be worse than an optimal solution.

Question 2(c) →

(c) Given the input instance shown below (Fig. 1(c)), state the optimal schedule produced by your algorithm for Question 2(a).

Solution: c, f, j, l

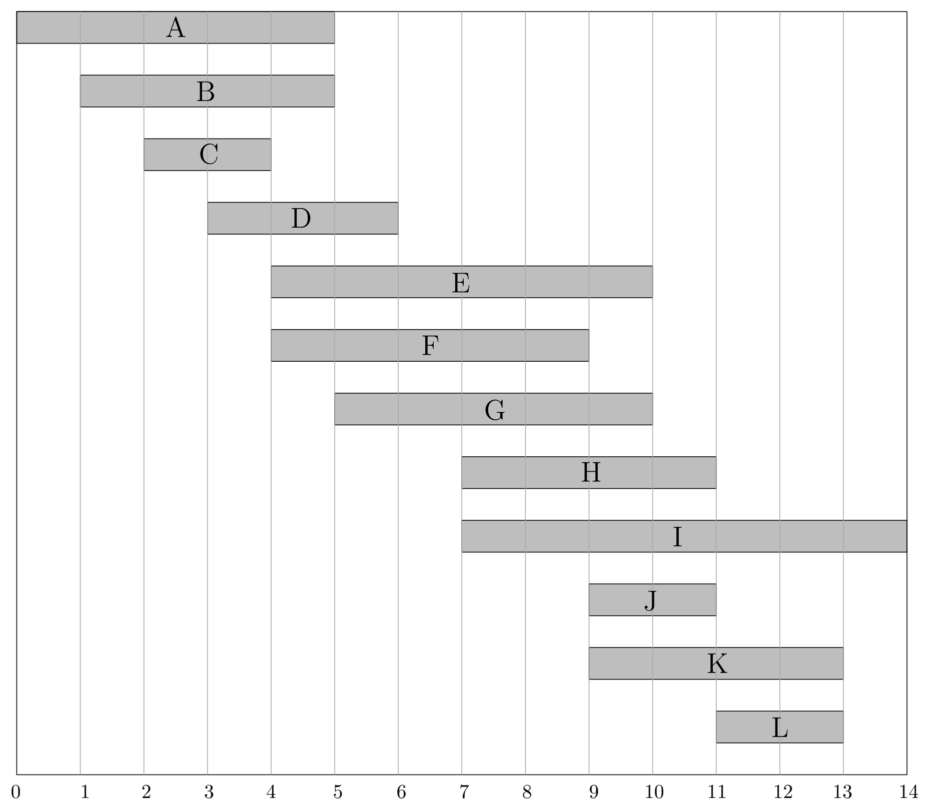


Figure 1: The input instance to Question 2.

- **Question 3 (10 points): Divide and Conquer**

  Given an array $A[1..n]$ of integers, an element x in A is said to have the *majority* if and only if the number of x's in A is greater than $n/2$. Consider the following algorithm (also given in pseudocode below). We split the array A into two subarrays A_L and A_R of half the size. We choose the majority element M_L of A_L and the majority element M_R of A_R, if they exist. After that we check if M_L is a majority element of A. If not then we check if M_R is a majority element of A. If none of these are true then the algorithms returns *'no majority'*.

Algorithm 1 MAJORITY

```
function MAJORITY(A[1..n])          ▷ $A$ is an array of $n$ integers
    if $n = 1$ then
        return $A[1]$
    end if
    Let $A_L$ be the first half of $A$
    Let $A_R$ be the second half of $A$
    $M_L$ = MAJORITY($A_L$)
    $M_R$ = MAJORITY($A_R$)
    if $M_L$ is a majority element of $A$ then
        return $M_L$
    end if
    if $M_R$ is a majority element of $A$ then
        return $M_R$
    else
        return "no majority"
    end if
end function
```

Question 3(a–b) →

(a) State and solve the recurrence of the algorithm. You can assume lines 9 and 11 requires $O(n)$ time.

Solution: Breaking the main problem into the two subproblems and testing the two candidate majority elements can be done in $O(n)$ time. Thus we get the following recurrence

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

which solves to

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

(b) Argue the correctness of the Majority algorithm.

Solution:

1. If x is a majority element in the original array then, by the pigeon-hole principle, x must be a majority element in at least one of the two halves. Suppose that n is even. If x is a majority element at least $n/2 + 1$ entries equal x. By the pigeon hole principle either the first half or the second half must contain $n/4 + 1$ copies of x, which makes x a majority element within that half.

2. We scan the array counting how many entries equal x. If the count is more than $n/2$ we declare x to be a majority element. The algorithm scans the array and spends $O(1)$ time per element, so $O(n)$ time overall.

3. We break the input array A into two halves, recursively call the algorithm on each half, and then test in A if either of the elements returned by the recursive calls is indeed a majority element of A. If that is the case we return the majority element, otherwise, we report that there is "no majority element".

   To argue the correctness, we see if there is no majority element of A then the algorithm must return "no majority element" since no matter what the recursive call return, we always check if an element is indeed a majority element. Otherwise, if there is a majority element x in A we are guaranteed that one of our recursive call will identify x and the subsequent check will lead the algorithm to return x.

- **Question 4 (10 points): Knapsack**

Consider the knapsack problem as we discussed in class. You are not given the actual input, instead you are given the trace of the execution of the knapsack algorithm. The dynamic programming table B is given below. Recall that $B[k, w]$ is the optimal solution that can be obtained using only the first k items and a maximum allowed total weight of w.

$k \setminus w$	0	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	25	25	25	25
2	0	0	15	15	15	15	15	25	25	40	40
3	0	0	15	20	20	35	35	35	35	40	45
4	0	0	15	20	20	35	36	36	51	56	56

(a) What is the number of items in the input instance?

Solution: This is the given instance:

i	1	2	3	4
b_i	25	15	20	36
w_i	7	2	3	6

Answer to question (a) is 4

(b) What is the maximum weight limit of the knapsack?

Solution: 10

(c) What is the weight of item 1?

Solution: 7

Question 4(d-f) →

(d) What is the value of item 2?

Solution: 15

(e) What is the value of the best packing having a total weight of at most 8.

Solution: 51

(f) Which items are included in an optimal solution for $w = 10$?

Solution: 3 and 4

- **Question 5 (10 points): NP-completeness**

  Consider a set $A = \{a_1, a_2, \ldots, a_n\}$ and a collection $B_1, B_2, \ldots, B_m$ of subsets of A, $B_i \subseteq A$. We say that a set $H \subseteq A$ is a hitting set for the collection $B_1, \ldots, B_m$ if H contains at least one element of from each B_i. The hitting set problem is defined as follows: Given a set $A = \{a_1, a_2, \ldots, a_n\}$ and a collection of subsets $B_1, B_2, \ldots, B_m$, and a positive integer k decide whether there exists a hitting set $H \subseteq A$ for $B_1, B_2, \ldots, B_m$ so that the size of H is at most k. Prove that the hitting set problem is NP-complete. Hint: A suggestion is to use the Vertex Cover problem in your proof.

Solution: To prove hitting set is NP-complete, we will show that Hitting Set is in NP and it is NP-hard by reducing the Vertex Cover problem to Hitting Set.
Hitting Set is in NP: The polynomial time verifier will take (A, B, k) and H as certificate. The algorithm will check if (a) $|H| = k$, (b) H is a subset of A; (c) for every set B_i, B_i and H have at least one common element. The time complexity is $O(|A|^2|B|)$.
Hitting Set is NP-hard because we can reduce the known NP-complete problem, Vertex Cover, to Hitting Set.
Problem: vertex cover
Input: A graph $G = (V, E)$ and a positive integer $k \leq |V|$.
Output: Is there a subset S of at most k vertices such that every edge in E has at least one vertex in S?
Reduction: Given a graph $G = (V, E)$ and a number k, we define the instance (A, B, k) of Hitting Set as follows: $k = k$, $A = V$ and $B = \{u, v | (u, v) \in E\}$. The construction of (A, B, k) takes linear time.
Claim: G has a vertex cover of size k iff (V, B, k) has a hitting set of size k.
Proof of the claim: (Only if part): If G has a vertex cover S of size k, then S is also a hitting set for (V, B, k) because for each edge (u, v) of E, either u or v is in S. So for each set $\{u, v\}$ of B, $\{u, v\}$ has at least one element in S. Thus S is a hitting set.
(If part): If X is a hitting set of (A, B, k), then X is also a vertex cover of G because for each set $\{u, v\}$ of B, u or v is in X, so the edge $\{u, v\}$ is covered by X.

- **Question 6 (10 points): Dynamic Programming**

Let $G = (V, E)$ be an undirected graph with edge weights $w : E \to Z^+$ (positive integers). Recall that a matching $M \subseteq E$ is a subset of edges such that no two edges in M are incident on the same vertex. The maximum weight matching problem is to find a matching M maximizing the sum of the weights of the edges in M, that is, maximize $\sum_{e \in M} w(e)$.

Your task is to design a polynomial time algorithm for solving the maximum weight matching on *binary* trees using dynamic programming. Remember to:

(a) Clearly define your DP states.

Solution: Pick an arbitrary vertex $r \in V$ as the root of the tree, denote this rooted tree by T. For every vertex $v \in V$ let $T(v)$ denote the subtree of T rooted at v.
For each vertex $v \in V$ we maintain two states:

- $M_{in}[v]$: The weight of a maximum weight matching $M(v)$ of $T(v)$ where v is incident to an edge in $M(v)$.
- $M_{out}[v]$: The weight of a maximum weight matching $M(v)$ of $T(v)$ where v is not incident to any edge in $M(v)$.

(b) State the recurrence (base and recursive cases).

Solution: For a vertex v let v_l and v_r denote the left and right child of v, respectively (if they exist, otherwise they are set to *null*). We have the following recurrence:

$$M_{in}[v] = \begin{cases} 0 & \text{if } v \text{ is a leaf or null} \\ \max[w(v, v_l) + M_{out}[v_l] + \max(M_{in}[v_r], M_{out}[v_r]), & \\ w(v, v_r) + M_{out}[v_r] + \max(M_{in}[v_l], M_{out}[v_l])] & \text{otherwise} \end{cases}$$

$$M_{out}[v] = \begin{cases} 0 & \text{if } v \text{ is a leaf or null} \\ \max[\max(M_{in}[v_l], M_{out}[v_l]) + \max(M_{in}[v_r], M_{out}[v_r])] & \text{otherwise} \end{cases}$$

Question 6(c) →

(c) Analyze the time complexity of your algorithm.

Solution: If the DP states are computed bottom-up, starting from the leaves, then each vertex can be processed in $O(1)$ time, thus $O(n)$ in total.

(end of exam)