HKUST-Department of Computer Science and Engineering
COMP3711: Design and Analysis of Algorithms
Final Examination - Fall 2018

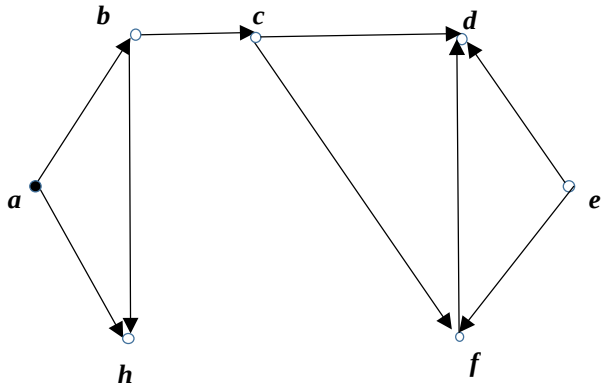
Name:	Student ID	ITSC account:

- This is a closed book exam. It consists of 10 pages and 8 questions.
- Answer all the questions within the space provided on the examination paper.
- ONLY WRITE ON THE FRONTS OF PAGES. SOLUTIONS WRITTEN ON THE BACKS OF PAGES WILL BE IGNORED.
- No math sheet is required for this exam.

Prob#	1	2	3	4	5	6	7	8	Total
points									

1. Topological Order and Breadth/Depth First Traversal [10 points]

Consider the following directed acyclic graph.



Part (a) [4 points]

Provide **two** different valid topological orders.

Topological order 1: a, b, h, c, e, f, d

Topological order 2: a, b, c, h, e, f, d

Marking Scheme:

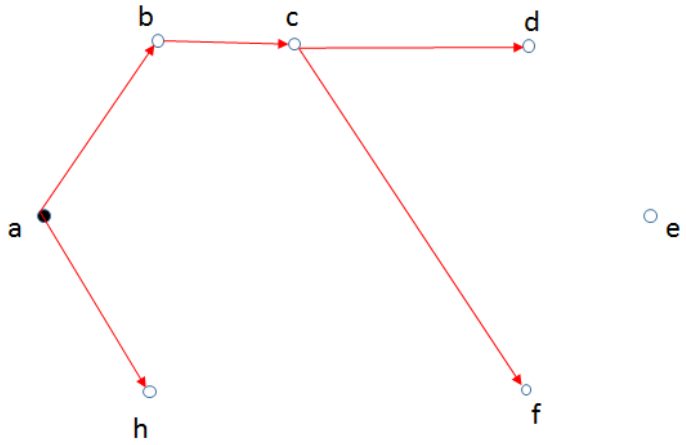
i. 2 points for each correct topological order. There are some more correct topological orders: [a, b, h, e, c, f, d], [e, f, d, a, b, h, c], [e, f, d, a, b, c, h], …, etc.

ii. If topological order 1 is identical to topological order 2, count only once.

iii. -1 point for any two out of order vertices for each topological order. E.g. if the answer of topological order 1 is [a, h, b, c, e, f, d], get 1 point out of 2.
0 point for more than 1 out of order pairs for each topological order.

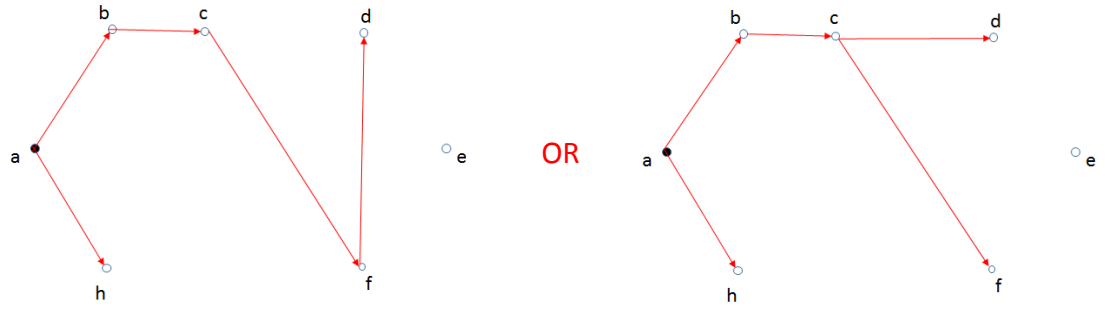
iv. If topological order 1 (or 2) is correct and topological order 2 (or 1) has exactly one out of order pair, and swapping the out of order vertices in topological order 2 (or 1) becomes identical to the topological order 1 (or 2), then the 1 point from (iii) will NOT be given because it's in fact two identical order.
   - Eg 1: topological order 1: [a, b, h, c, e, f, d] and topological order 2: [b, a, h, c, e, f, d]. Only 2 point for topological order 1, no point for topological order 2 because swapping (b, a) gives the same answer as topological order 1.
   - Eg 2: topological order 1: [a, b, h, c, e, f, d] and topological order 2: [e, d, f, a, b, h, c]. 2 point for topological order 1, 1 point for topological order 2 because topological order 2 has exactly one pair out of order (d, f). Swap (d, f) is different with topological order 1.

Part (b) [3 points]
Draw the BFS Tree if we start breadth first traversal from node *a*. (The tree only covers nodes reachable from *a*)



Marking Scheme:
-1 point for any extra or missing edge.

Part (c) [3 points]
Draw a DFS Tree if we start depth first traversal from node *a*. (The tree only covers nodes reachable from *a*)

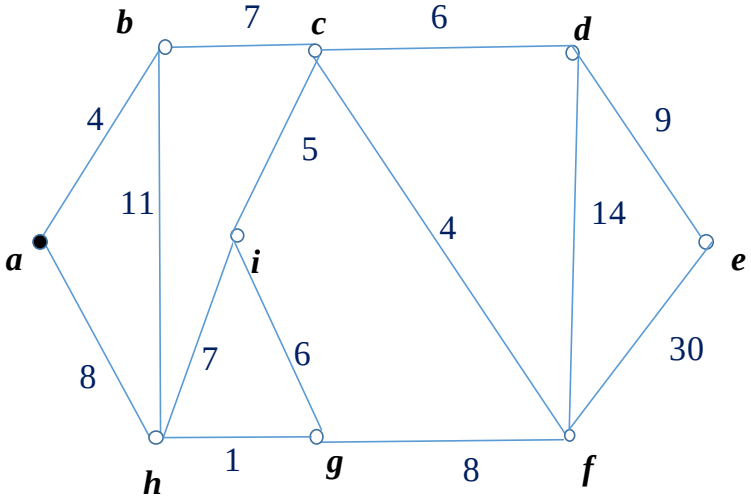


Because the adjacency list is not given, there are more than one (total should be 4) possible depth first search results. Two possible solutions are given above.

Marking Scheme:
-1 point for any extra or missing edge.

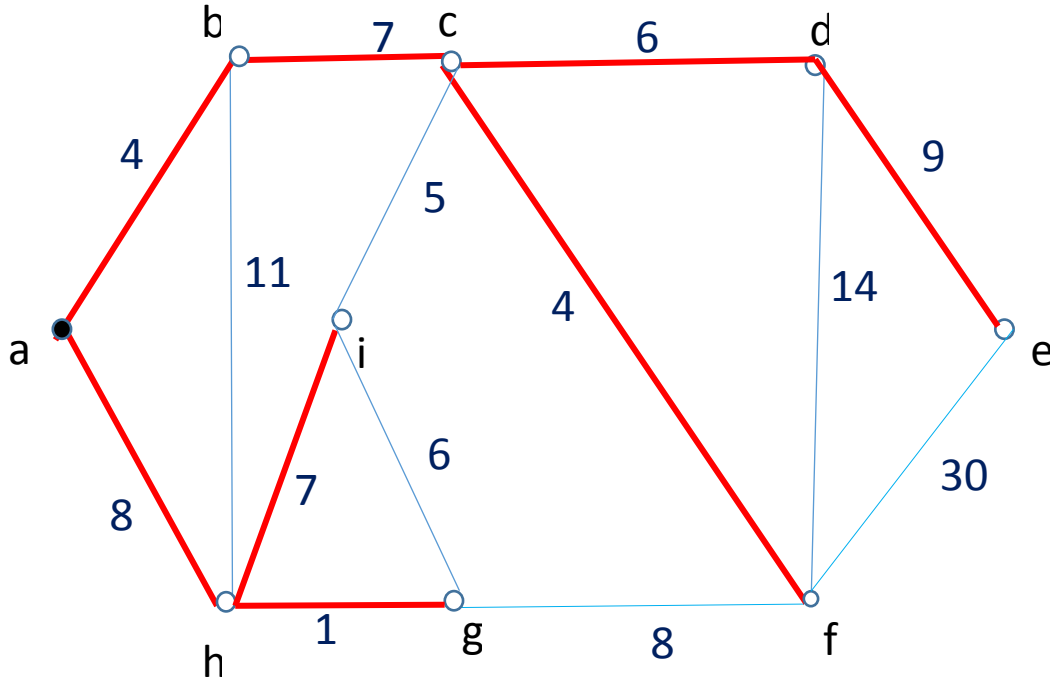
2. Shortest Path and Minimum Spanning Trees [10 points]

Consider the following graph.



Part (a) [5 points]

Draw the *shortest path tree* generated by applying *Dijkstra's algorithm* starting from node *a*.

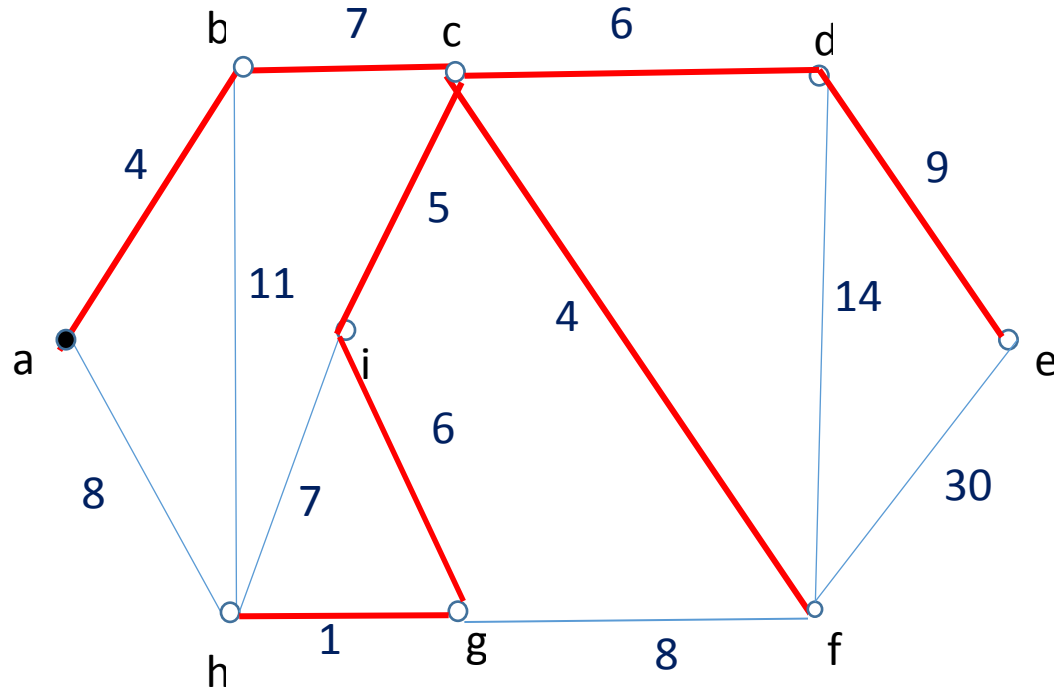


Marking Scheme:

i. 0 point if not a tree that covers all vertices.
ii. -1 point for each wrong edge if the answer is a tree that covers all vertices.

Part (b) [5 points]

Draw the *minimum spanning tree* generated by applying *Prim's algorithm* starting from node *a*.



Marking Scheme:

i. 0 point if not a tree that covers all vertices.

ii. -1 point for each wrong edge if the answer is a tree that covers all vertices.

3. Greedy Algorithms [10 points]

Let $X = \{x_1, x_2, \ldots, x_n\}$ be an input set of positive numbers such that $x_1 < x_2 < \cdots < x_n$. Describe a $\Theta(n)$ algorithm that returns the minimum number of unit intervals (i.e., with length equal to 1) required to cover all numbers. We say that an interval $i = [i_s, i_e]$ covers x_i, if $i_s \leq x_i \leq i_e$.

Solution:

Keep the points in an array. Walk through the array as follows.

1. Set $x = x_1$
2. Walk through the points in increasing order
   until finding the first j such that $x_j > x + 1$
3. Output
4. If there was no such j in (2) then stop. Otherwise, set $x = x_j$
5. Go to Step (2)

The running time is obviously $\Theta(n)$ because we just do a linear scan of the n numbers.

Marking Scheme:

i. 0 points if the answer is $\omega(n \log n)$.
ii. -7 points if the answer re-sort X (the given array is already sorted) or any correct $\Theta(n \log n)$ algorithm.
iii. -3 points if missing "set x = x_1".
iv. -5 points if interval start at x_{j-1} instead of x_j for j defined in the solution above.
v. -7 points if interval is not start at x_j for j defined in the solution above.
vi. -1 if missing/incorrect running time analysis.
vii. -1 point for each of any other careless mistake.
viii. Prove of correctness is not necessary.

4. Stable Matching [12 points]

Consider the following preference list for four men and women.

Man	1st Choice	2nd Choice	3rd Choice	4th Choice
A	W	X	Y	Z
B	X	W	Z	Y
C	W	Z	X	Y
D	Z	X	W	Y

Woman	1st Choice	2nd Choice	3rd Choice	4th Choice
W	B	D	A	C
X	A	C	D	B
Y	D	B	C	A
Z	A	D	C	B

Part (a) [6 points]

Provide a stable matching, which is *optimal* for the men.

A-X, B-W, C-Y, D-Z

Part (b) [6 points]

Provide a stable matching, which is *optimal* for the women.

A-X, B-W, C-Y, D-Z

Marking Scheme for both (a) and (b):

-1.5 for each incorrect matching.

5. Divide and Conquer [14 points]

You are given an array $A[1..n]$ that contains a sequence of 0 followed by a sequence of 1 (e.g., 0001111111). A contains at least one 0 and one 1.

Part (a) [6 points]
Design an O(log n) algorithm that finds the position k of the last 0, i.e. $A[k] = 0$ and $A[k + 1] = 1$.

Solution:
find-k($A[p..r]$)
if $r=p+1$, RETURN p
$mid=\lfloor (p+r)/2 \rfloor$
if $A[mid]=0$ and $A[mid+1]=1$, RETURN mid
if $A[mid]=0$, **find-k**($A[mid..r]$)
else **find-k**($A[p..mid]$)

This is similar to binary search: with a constant number of comparisons, we reduce the problem size by half: $T(n)=T(n/2)+c \Rightarrow T(n) = O(\log n)$

Marking Scheme:

i. At most 1 point will be given if using Selection, Partition or Sorting.
ii. -3 points for missing base case of both "if r = p+1, RETURN p" and "if $A[mid]=0$ and $A[mid+1]=1$, RETURN mid".
iii. -2 points for missing base case of "if r = p+1, RETURN p".
- Note that, missing "if $A[mid]=0$ and $A[mid+1]=1$, RETURN mid" is fine as long as "if r = p+1, RETURN p" is in the algorithm.

iv. If missing "if $A[mid]=0$ and $A[mid+1]=1$, RETURN mid",
- Some answer's base case might be "if r == p then, return p". In this case, the recursive call should be something like
"If A[mid] == 0, **find-k**(A[mid .. r]) else **find-k**(A[p..mid-1])"
Otherwise, the algorithm may never be terminated.

v. -1 point for missing a brief explanation of running time.
vi. -1 point for each of any other careless mistakes.
vii. Prove of correctness is not necessary.

Part (b) [8 points]
Suppose that k is much smaller than n. Design an O(log k) algorithm that finds the position k of the last 0.
Hint: You may re-use the solution of Part (a) in the solution of Part (b).

Solution:
$i \leftarrow 1$
while $A[i]=0$
$i \leftarrow 2i$
find-k($A[i/2+1..i]$)

The while loop will stop when it finds a 1. Since each time we double the value of *i*, the while loop performs log*k* iterations. The first 1 occurs somewhere between the positions *A*[*i*/2+1] and *A*[*i*]. To find it, we call **find-k**(*A*[*i*/2+1..*i*]), which has cost log(*k*/2)= *O*(log *k*). Therefore, the total cost is *O*(log *k*).

Marking Scheme:

It's fine to assume there exist a O(log n) algorithm to solve Part (a) and use here even though the answer of Part (a) is wrong.

i. 5 points for repeatedly multiply i by 2 until A[i] = 1.
ii. 3 points for running time analysis.
iii. -1 point for each of any other careless mistakes.
iv. Prove of correctness is not necessary.

6. Maximum Flow [16 pts]

A project is partitioned into m tasks $t_1, \dots, t_m$. There are n candidate workers $c_1, \dots, c_n$ and n lists $l_1, \dots\ l_n$. List l_j contains the tasks that candidate c_j can work for. For example, $l_1 = \{t_3, t_5\}$ means candidate c_1 can only be assigned to t_3 or t_5. You can assign at most a_i candidates to task t_i for $1 \le i \le m$. Each candidate can be assigned to at most 1 task. Design an algorithm for finding a candidate-task assignment that maximizes the number of working candidates under the limitations of a_i and l_j for $1 \le i \le m$ and $1 \le j \le n$. Note that some tasks may have no candidate.

Solution:

We first form a flow network *G* as follows:

- The *m* tasks form *m* vertices in the flow network, denote these *m* vertices by $t_1, \dots, t_m$. The *n* candidates form *n* vertices in the flow network, denote these *n* vertices by $c_1, \dots, c_n$.
- According to the list l_j,add edges from c_j to the vertices of the tasks in l_j with capacity 1.
- Add a source vertex *s* and a sink vertex *t*. Connect *s* to t_i with capacity a_i for all $1 \le i \le m$. Connect t to c_j for all $1 \le j \le n$ with capacity 1.

Apply Ford-Fulkerson algorithm with Edmonds Karp suggestion to find the maximum flow of *G*. The maximum ow value is the maximum number of candidates. The candidates who received a flow is assigned to the project. Trace the flow pushed to the vertex c_i, you know which task is assigned to candidate *i*.

The running time to construct the flow network is O(n + m + nm) time because there are $O(n + m)$ vertices and $O(nm)$ edges. The time for computing maximum flow is $O((n + m)(nm)^2)$.

Marking Scheme:

i. Any correct algorithm runs in $O((n + m)(nm)^2)$ is fine, not restricted to Max Flow.

If the answer is converted to max flow problem:

ii. -8 points if everything correct but without mention the edge capacities.
iii. If some edge capacities are mentioned:
    - -2 points if wrong/missing capacity is assigned for edges from candidates to sink.

- -3 points if wrong/missing capacity is assigned for edges from source to tasks.
- -3 points if wrong/missing capacity is assigned for edges from tasks to candidates.

iv. -2 if not single source single sink.
v. -3 if incorrectly connect to source and/or sink.
vi. -3 if incorrectly connect candidates and tasks.
vii. Prove of correctness and running time analysis are not necessary.

7. Dynamic Programming [18 pts]

Consider a version of the 0-1 knapsack problem, where there are infinite copies of each item. Specifically, the input is a set of n items, where item i has weight w_i and value v_i, and a knapsack with capacity W. The goal is to find $x_1, \dots, x_n \in \{0,1,2,3, \dots\}$ such that $\sum_{i=1}^{n} x_i w_i \leq W$ and $\sum_{i=1}^{n} x_i v_i$ is maximized. The difference with respect to the conventional 0-1 knapsack is that we can carry as many copies of the same item as we wish.

Part (a) [6 points]
Describe a greedy algorithm for solving the problem, and discuss if it is optimal or not.

Solution:
Take as many copies of the item with the largest $\frac{v_i}{w_i}$ as you can. Then, continue with the next most precious item that fits in the knapsack.
This will not give the optimal solution. Assume two items with $v_1 = 10$, $w_1 = 10$ and $v_2 = 3.5$, $w_2 = 4$. W = 12. The greedy solution will take item 1 with total value 10. The optimal solution will take 3 copies of item 2, with value 10.5.

Marking Scheme:
i. 0 point for non-greedy solution
- A greedy solution is not necessary to use $\frac{v_i}{w_i}$ as the key. It's fine to pick largest v_i or smallest w_i, etc, as long as it is always pick the largest or smallest element.

ii. -4 points if saying greedy algorithm can yield optimal solution.
iii. -2 points if saying greedy algorithm cannot yield optimal solution, but missing or incorrect example.
iv. Proof of correctness and running time analysis are not necessary.

Part (b) [6 points]
Describe and explain the recurrence for a dynamic programming algorithm that finds the optimal solution.

Solution:
If Optimal Solution for knapsack of size j chooses item i, remainder of optimal solution is optimal solution for subproblem of filling knapsack of size $j - w_i$.
Recursive case: $V[j] = \max(0,\ v_1 + V[j - w_1],\ v_2 + V[j - w_2], \dots,\ v_n + V[j - w_n])$
Base cases: $V[j] = 0, j = 0$
$V[j] = -\infty, j < 0$

Marking Scheme:
i. 0 point if EACH $V[j]$ is solved in exponential time (Note, $O(W)$ is also treated as exponential).

ii. 1 point if completely correct and EACH $V[j]$ is solve in time polynomial of n, but $\omega(n)$.
iii. -2 points for each missing/incorrect base case.
iv. -2 points for incorrect recursive case.
- Only -0.5 points if everything correct but missing 0, i.e.

$$V[j] = \max(\ v_1 + V[j - w_1],\ v_2 + V[j - w_2], \dots,\ v_n + V[j - w_n])$$

v. Proof of correctness and running time analysis are not necessary.

Part (c) [6 points]
Write the pseudo-code based on the recurrence of Part (b).

Solution:

$V[j] = 0$
for $j = 1$ to W
    $V[j] = 0$
    for $i = 1$ to n
        if $j \geq w_i$ and $v_i + V[j - w_i] \geq V[j]$
            $V[j] = v_i + V[j - w_i]$

Marking Scheme:

i. 0 point if completely not following the student's answer in Part (b).
ii. 0 point if Part (b) is completely not solving the problem.
iii. If Part (b) solves the problem (suffer minor mistake)
- Full marks for pseudo-code exactly follows the student's answer in Part (b).
- -1 point for any careless mistake.

8. Dynamic Programming over Intervals [10 points]

Given an array A of n non-negative integers and an integer m (m is much smaller than n), we wish to split A into m **non-empty continuous** subarrays, so that the largest sum among these m subarrays is minimized.

Example:

Assume that the input array is A=[7,2,5,10,8] and m = 2. There are four ways to split A into two subarrays: **(1)** [7],[2,5,10,8], **(2)** [7, 2], [5,10,8], **(3)** [7, 2, 5],[10,8], **(4)** [7, 2, 5, 10],[8]
The best way is to split it into [7,2,5] and [10,8] because the largest sum between the two subarrays is only 18.

Provide and explain the recurrence for the optimal solution. You do **not** need to include any pseudo-code.

Solution:

Let $s[j] = \sum_{k=1}^{j} A[k]$, s[j] can be computed in $O(n)$ time by
$s[0] = 0, s[j] = s[j-1] + A[j]$
With this pre-sum array s, $\sum_{k=i}^{j} A[k] = s[j] - s[i-1]$ can be computed in $O(1)$ time for $1 \leq i \leq j \leq n$.

For $1 \leq i \leq m$, $1 \leq j \leq n$, define $d[i][j]$ be the optimal value of the subproblem that we split $A[1..j]$ into i non-empty continuous subarrays so that the largest sum among these i subarrays is minimized.

Base case: $d[1][1] = A[1]$.
Recursive case: $d[i][j] = min_{i-1 \leq k < j}(max(d[i-1][k], s[j] - s[k]))$

The last split can be at position $i-1$ to $j-1$. When the last split is at position k, this means we have split the array into $i-1$ subarrays for the first k element, $d[i-1][k]$ stores the largest sum among these $i-1$ subarrays. The sum of the i-th subarray is $s[j] - s[k]$. So, $max(d[i-1][k], s[j] - s[k])$ is the largest sum among the i subarrays subject to the last split of $A[1..j]$ is at position k. We simply try all possibility (all possible k) and pick the minimum.

The running time of this algorithm is $O(n^2 m)$ because there are $O(nm)$ subproblems and each subproblem can be computed in $O(n)$ time. The time for computing s at the beginning is just $O(n)$.

Marking Scheme:

i. 0 point for any answer that runs in $\Omega(n^4 m)$.
ii. -2 points for missing the pre-sum array s. Because it results in a very slow algorithm.
iii. -2 points for missing/incorrect base case.
   - $d[0][0] = 0$ is a correct base case.
iv. -3 points for $d[i][j] = min_{i-1 \leq k < j}(d[i-1][k])$.
v. -3 points for $d[i][j] = min_{i-1 \leq k < j}(s[j] - s[k])$.
vi. -1 for each careless mistake. E.g. $min_{i \leq k < j}$ or $min_{i-1 \leq k \leq j}$, etc.
vii. Proof of correctness and running time analysis are not necessary.