NOTE TO STUDENTS: This file contains sample solutions to the term test together with the marking scheme and comments for each question. Please read the solutions and the marking schemes and comments **carefully**. Make sure that you understand why the solutions given here are correct, that you understand the mistakes that you made (if any), and that you understand *why* your mistakes were mistakes.

Remember that although you may not agree completely with the marking scheme given here it was followed the same way for all students. We will remark your test only if you clearly demonstrate that the marking scheme was not followed correctly.

For all remarking requests, please submit your request **in writing** directly to your instructor. For all other questions, please don't hesitate to ask your instructor during office hours or by e-mail.

Question 1. [16 MARKS]

It's nearing the end of the semester and you have n final projects to complete. Your goal, of course, is to maximize your total grade on these projects! For simplicity, say that you have a total of H hours (a positive integer) to work on the projects cumulatively, and you'll spend an integer number of hours on each project. To figure out how best to divide up your time, you've come up with a set of non-decreasing estimate functions $\{e_1, e_2, \ldots, e_n\}$, one for each project: if you spend h hours on project i (where $0 \leqslant h \leqslant H$), you'll get a grade of $e_i(h)$ on that project.

Give a dynamic programming algorithm that takes the estimate functions $\{e_1, \ldots, e_n\}$ as input and that generates non-negative numbers of hours $\{h_1, \ldots, h_n\}$ such that $h_1 + h_2 + \cdots + h_n = H$ and your total grade $e_1(h_1) + e_2(h_2) + \cdots + e_n(h_n)$ is maximum. *For full marks, follow clearly the steps outlined in class for dynamic programming, justify your recurrence, and briefly analyze your algorithm's running time.*

SAMPLE SOLUTION:

Step 1: *Define an array.*

For $0 \leqslant k \leqslant n$ and $0 \leqslant h \leqslant H$, let $M[k, h]$ represent the maximum total grade on projects $1, \ldots, k$, using at most h hours in total.

Step 2: *Give a recurrence for the array values.*

$M[0, h] = 0$ for $0 \leqslant h \leqslant H$ (no project means no grade).

$M[k, h] = \max_{i=0}^{h} M[k-1, h-i] + e_k(i)$ (the best grade overall involves spending some number of hours i on project k, and the remaining hours to get the maximum grade out of projects $1, \ldots, k-1$).

Step 3: *Write a bottom-up algorithm to compute the array values.*

```
for h ← 0, 1, ..., H:
    M[0, h] ← 0
for k ← 1, 2, ..., n:
    for h ← 0, 1, ..., H:
        M[k, h] ← M[k − 1, h] + e_k(0)
        for i ← 1, 2, ..., h:
            if M[k − 1, h − i] + e_k(i) > M[k, h]:
                M[k, h] ← M[k − 1, h − i] + e_k(i)
```

Step 4: *Reconstruct a solution from the array values.*

Use a second array $N[k, h]$ to store the number of hours to spend on project k in order to achieve grade $M[k, h]$, for all k, h.

```
for h ← 0, 1, ..., H:
    M[0, h] ← 0
for k ← 1, 2, ..., n:
    for h ← 0, 1, ..., H:
        M[k, h] ← M[k − 1, h] + e_k(0)
        N[k, h] ← 0
        for i ← 1, 2, ..., h:
            if M[k − 1, h − i] + e_k(i) > M[k, h]:
                M[k, h] ← M[k − 1, h − i] + e_k(i)
                N[k, h] ← i

h ← H
for k ← n, n − 1, ..., 1:
    h_k ← N[k, h]
    h ← h − N[k, h]
return h_1, h_2, ..., h_n
```

The total running time is $\Theta(nH^2)$.

MARKING SCHEME:

- **Form** [5 marks]: clear attempt to define an array, to give a recurrence relation for the array values, to justify the recurrence based on the problem's structure, to give a bottom-up algorithm to compute the array values, to reconstruct a solution from the array values, and to analyze the algorithm's running time
- **Array** [2 marks]: clear and correct array definition
- **Recurrence** [4 marks]: clear and correct recurrence relation, including base cases and good justification
- **Bottom-up** [1 mark]: clear and correct algorithm to compute array values bottom-up
- **Solution** [3 marks]: clear and correct algorithm to generate solutions from the array values
- **Time** [1 mark]: correct analysis of algorithm's running time

MARKER'S COMMENTS:

- **common error** [−2 to −3]: forgetting to reconstruct an optimal solution, or omitting details of how to do it
- **common error**: using "$A[0, h] = \infty$": this makes the recurrence fail
- **common error** [−1 for Justification]: no justification/explanation provided for the recurrence

Question 2. [14 MARKS]

You have a network of wireless sensors that you would like to make more reliable, by selecting some number of "backup" sensors for each sensor in the network. Formally, consider the following problem.

Input: Sensors $s_1, s_2, \ldots, s_n$ (where each sensor s_i has coordinates (x_i, y_i)), distance parameter $d > 0$, positive integer redundancy parameter r, and positive integer backup parameter $b \geqslant r$.

Output: Backup sets $B_1, \ldots, B_n$ where $B_j \subseteq \{s_1, \ldots, s_n\} - \{s_j\}$ for each j, every sensor in B_j is within distance d of s_j, every B_j contains *exactly* r elements, and every sensor belongs to at most b backup sets. (If this is not possible, output the special value NIL.)

Give an efficient algorithm to solve this problem, based on network flow techniques. Justify that your algorithm is correct (in particular, explain how backup sets and flows correspond to each other) and analyze the worst-case running time of your algorithm. (HINT: Consider using *two* nodes for each sensor.)

SAMPLE SOLUTION:

Create a network $N = (V, E)$, where $V = \{s, t, a_1, \ldots, a_n, b_1, \ldots, b_n\}$ and

$$\begin{aligned} E = &\{(s, a_i) : 1 \leqslant i \leqslant n\} \text{each with capacity } b, \\ &\cup \{(b_j, t) : 1 \leqslant j \leqslant n\} \text{each with capacity } r, \\ &\cup \{(a_i, b_j) : 1 \leqslant i \neq j \leqslant n \text{ and distance(server } i\text{, server } j) \leqslant d\} \text{each with capacity 1.} \end{aligned}$$

Then, find a maximum flow f in the network. If $|f| = kn$, let $B_i = \{j \in \{1, \ldots, n\} : f(a_j, b_i) = 1\}$ for $i = 1, 2, \ldots, n$; otherwise, output NIL.

Worst-case running time is $\mathcal{O}(n^3)$, for finding a maximum flow.

The algorithm is correct because:

- Every selection of backup sets $B_1, \ldots, B_n$ corresponds to a valid flow in N, by setting $f(s, a_i) =$ number of backup sets that a_i belongs to (cannot be more than b for valid backup sets), $f(b_j, t) =$ size of backup set B_j (equal to r for valid backup sets), and $f(a_i, b_j) = 1$ if $i \in B_j$ (0 otherwise).
  This means the maximum flow value is at least as large as kn.
- Every valid flow in N corresponds to a selection of backup sets $B_1, \ldots, B_n$ by putting i in B_j iff $f(a_i, b_j) = 1$. No sensor can belong to more than b different backup sets (because $c(s, a_i) = b$), and no sensor has a backup set larger than r (because $c(b_j, t) = r$).
  This means the maximum flow in N determines a collection of backup sets that is as large as possible.

MARKING SCHEME:

- **Form** [5 marks]: clear attempt to give a network-based algorithm to solve the problem (constructing a network from the problem input, solving the maximum flow problem on this network, and extracting a solution from the network flow), to argue that the algorithm is correct, and to analyze the algorithm's running time
- **Algorithm** [5 marks]: clear algorithm, correct network generated by the algorithm, and correct backup sets constructed from the network flow
- **Justification** [4 marks]: correct analysis of algorithm's runtime, and correct justification that network flow values and backup sets correspond to each other (in both directions)

MARKER'S COMMENTS:

- **error code S**: sometimes used instead of "J" for Justification
- **common error** [−1 to −3]: missing or poor justification
- **common error** [−1 for Algorithm]: did not explain how to construct solution from flow
- **common error** [−1 for Justification]: did not analyze running time

Question 3. [10 MARKS]

We know how to generate Minimum Spanning Trees efficiently. In this question, you will explore how to *update* a M.S.T. when changes are made to the input graph. Formally, consider the following problem.

Input: An undirected graph $G = (V, E)$ with positive integer costs $c(e)$ for every edge $e \in E$. A minimum spanning tree $T \subseteq E$ for G. One edge $e_0 \in E - T$ (*i.e.*, e_0 is an edge of G that is *not* in T). A new cost $c'(e_0) < c(e_0)$ for edge e_0.

Output: A minimum spanning tree T' of the graph G', where G' is the same as G except for the cost $c'(e_0)$ of edge e_0.

Give a linear-time algorithm to solve this problem. Justify briefly that your algorithm runs in linear time and that it is correct, for example, by stating properties of M.S.T.s that you rely on. (HINT: Think about the endpoints of edge e_0.)

SAMPLE SOLUTION:

Known fact (from the proof of correctness of Kruskal's algorithm): Adding any edge to a M.S.T. creates one cycle and removing any edge with maximum cost from such a cycle yields another M.S.T. We can apply this directly to T along with edge e_0:

- Use DFS to find a path in T from one endpoint of e_0 to the other, and remember an edge e' with maximum cost $c(e')$ on this path. (This path forms a cycle together with e_0.)
- If $c(e') <= c'(e_0)$, then output $T' = T$ (e_0 is an edge with maximum cost on the cycle).
- Else, output $T' = T - \{e'\} \cup \{e_0\}$ (swapping e_0 for e' reduces the total cost).

This clearly runs in linear time, because DFS takes linear time and the algorithm only carries out a constant amount of work outside of DFS.

MARKING SCHEME:

- **Form** [3 marks]: clear attempt to give an explicit algorithm, to analyze its running time, and to justify that it is correct based on known facts about M.S.T.s
- **Algorithm** [4 marks]: algorithm is written clearly, correctly finds a spanning tree with minimum total cost, and runs in linear time
- **Runtime** [1 mark]: correct analysis of algorithm's running time
- **Correctness** [2 marks]: good justification for the correctness of the algorithm

MARKER'S COMMENTS:

- **common error**: searching for edges adjacent to e_0 only
- **common error**: removing an edge with maximum cost from T without checking that it belongs to a cycle with e_0

Bonus. [4 MARKS]

Give an algorithm to find the *maximum* sum of any consecutive sublist $L[i] + L[i+1] + \cdots + L[j]$ of a list of numbers L (numbers could be positive, negative, or zero). Your algorithm must run in time $o(n^2)$ (*i.e.*, asymptotically *less* than n^2). Justify that this is the case and that your algorithm is correct.

SAMPLE SOLUTION: (Not provided for bonus...)

MARKING SCHEME: *Be particularly picky when marking the bonus!*

- **Array** [1 mark]: clear and correct array definition for dynamic programming solution
- **Recurrence** [2 marks]: clear and correct recurrence relation, with detailed justification
- **Solution and Runtime** [1 mark]: correct solution returned, correct runtime analysis, and runtime is $o(n^2)$

MARKER'S COMMENTS: *(None)*