University of Toronto, Department of Computer Science
CSC 373 Midterm Exam, July 3^{rd}, 2014
Lalla Mouatadid
Duration: 50 minutes
No Aids Allowed

PLEASE COMPLETE THE SECTION BELOW:

First Name: ____

Last Name: ____

Exam Instructions

- **Check that your exam book has 13 pages** (including this cover page and 2 blank pages at the end). The last 2 pages are for rough work only, *they will* **not** *be marked.* Please bring any discrepancy to the attention of an invigilator.
- There are 4 questions worth a total of 20 points. Answer all questions on the question booklet.
- For Questions C and D, if you do not know how to answer any part of the question then you can leave that part blank or write "I DON'T KNOW." to receive 10 percent of the points of the question.

Course Specific Notes

- Unless stated otherwise, you can use the standard data structures and algorithms discussed in CSC263 and in the lectures without describing their implementation by simply stating their standard name (e.g. min-heap, merge- sort, DFS, Dijkstra). You do not need to provide any explanation or pseudo-code for their implementation. You can also use their running time without proof. For example, if you are using the merge-sort in your algorithm you can simply state that merge-sort's worst-case running time is $\mathcal{O}(n \log n)$. If you modify a data structure or an algorithm from class, you must describe the modification and its effects.
- In some questions you will be given a computational problem and then asked to design an efficient algorithm for it. Unless stated otherwise, for data structures and algorithms that you design you should provide a short high-level explanation of how your algorithm works in plain English, and the pseudo-code for your algorithm in a style similar to those we have seen in the lectures. If you miss any of these the answer might not be marked. If you give the name of the country you're cheering for this world cup on the back of this exam you will receive one extra bonus mark. Your answers will be marked based on the efficiency of your algorithms and the clarity of your explanations. State the running time of your algorithm with a brief argument supporting your claim and prove that your algorithm works correctly (e.g. finds an optimal solution).

PLEASE PRINT YOUR STUDENT NUMBER AND YOUR NAME

Student Number: ____________________

First Name: ____________________

Last Name: ____________________

The section below is for marker's use only. Do NOT use it for answering or as scratch paper.

Questions	**Points**
A	/3
B	/6
C	/4
D	/7
Total	/20

A. Definitions [3]

Give the inputs and outputs of the following problems.

1. **Minimum Weight Spanning Tree** [1]

   Input:
   An edge weighted graph $G(V, E, w)$ where $w : E \to \mathbb{N}$.

   Output:
   A spanning tree T of G that minimizes $\sum\limits_{e \in T} w(e)$.

2. **A Maximum-Flow** [1]

   Input:
   A flow network (G, s, t, c) with source s, terminal t and edge capacity c.

   Output:
   The value of a maximum flow f on G, $val(f) = \sum\limits_{(s,u) \in E} f(s, u)$ is maximized.

3. **A minimum cost prefix-free code** [1]

   Input:
   An alphabet $\Sigma = \{a_1, a_2, ..., a_n\}$ and a corresponding list of probabilities (or frequencies) $\{p_1, p_2, ..., p_n\}$ where p_i is a_i's probability.

   Output:
   An encoding $\mathcal{C} = \{b_1, b_2, ..., b_n\}$ that minimizes $\sum p_i * length(b_i)$ where b_i is the encoding of a_i.

B. Algorithmic Paradigms [6]

1. What is the difference between an *adaptive* and a *non-adaptive* greedy algorithm? [2]

   We've been exposed to the adaptive vs non-adaptive greedy algorithms when dealing with minimum spanning tree algorithms. An adaptive greedy algorithm is one that would "adapt", rearrange, readjust its input at every iteration to its own convenience. A non-adaptive greedy algorithm on the other hand only re-orders it's inputs once, and then it blindly takes it one by one until it has been completely exhausted.

2. What distinguishes *dynamic programming* from *memoization*? [2]

   Dynamic programming is a bottom-up approach, where we solve all related subproblems first (i.e. filling up the table), then extract the solution of the original problem. Memoization on the hand is a top down approach, we start by solving the "top" problems, which typically recurses down to solve the subproblems.

3. What is the difference between a *greedy* algorithm and a *local search* algorithm? [2]

   A greedy algorithm is an algorithm where at every iteration, we make a myopic decision. That is, we make the choice that is best at the time, without worrying about the future. And decisions are irrevocable; you do not change your mind once a decision is made. Local search starts with an arbitrary solution to the problem, then keep refining this solution by making small repeated local changes that will increase the quality of the solution. Once a solution converges to a local optimum, then no number of small changes will increase its quality and we are done

C. Maximizing Payoff [4]

Suppose you are given two sets A and B, each containing n positive integers. You can choose to reorder each set however you like. After reordering, let a_i be the i^{th} element of set A, and let b_i be the i^{th} element of set B. You then receive a payoff of $\prod_{i=1}^{n} a_i^{b_i}$.

1. Give a greedy algorithm that will maximize your payoff. Your algorithm must run in $\mathcal{O}(n \log n)$ time. [1.5]

Algorithm 1 MaximizingPayoff

```
1: Sort $A$ in nondecreasing order $a_1 \geq a_2 \geq ... \geq a_n$
2: Sort $B$ in nondecreasing order $b_1 \geq b_2 \geq ... \geq b_n$
3: return $\prod_{i=1}^{n} a_i^{b_i}$                ▷ Pair $a_i$ with $b_i$
```

2. Give an argument about its running time. [0.5]

   If the two sets A and B are already sorted, the time complexity is $O(n)$.
   If the sets are not sorted, then sort them first and the time complexity is $O(n \log n)$.

3. Prove that your algorithm is optimal. [2]

Proof. Suppose the optimal payoff is not produced from the above solution. Let S be the optimal solution, in which a_1 is paired with b_p and a_q is paired with b_1. Note that $a_1 \geq a_q$ and $b_1 \geq b_p$.

Consider another solution S' in which a_1 is paired with b_1, a_q is paired with b_p, and all other pairs are the same as S. Then

$$\begin{aligned}\frac{\text{Payoff}(S)}{\text{Payoff}(S')} &= \frac{\prod\limits_{S} a_i^{b_i}}{\prod\limits_{S'} a_i^{b_i}} \\ &= \frac{(a_1)^{b_p}(a_q)^{b_1}}{(a_1)^{b_1}(a_q)^{b_p}} \\ &= (\frac{a_1}{a_q})^{b_p - b_1}\end{aligned}$$

Since $a_1 \geq a_q$ and $b_1 \geq b_p$, we have

$$\frac{\text{Payoff}(S)}{\text{Payoff}(S')} \leq 1$$

This contradicts the assumption that S is the optimal solution. Therefore a_1 should be paired with b_1. Repeating the argument for the remaining elements completes the proof. □

D. Ski Allocation [7]

A set S of n students from UofT organized a sking trip to Whistler. A ski rental agency has m pairs of skis, $m \geq n$, where the height of the i^{th} pair of skis is s_i.

Ideally, each skier should obtain a pair of skis whose height matches his/her own height as closely as possible. Let h_j denote the height of the j^{th} student.

The agency's goal is to assign skis to students as to minimize the cost $\sum_i |h_i - s_i|$.

Input: A list $A = \{s_1, s_2, ..., s_m\}$ where s_i denotes the length of i^{th} pair of skis and a list $B = \{h_1, h_2, ..., h_n\}$ where h_j denotes the height of the j^{th} student.

Output: The minimum possible cost to match n pairs of skis to the n students.

Give a dynamic programming algorithm to solve this problem. Your algorithm must run in $\mathcal{O}(n \log n + m \log m + (m - n)m)$ time.

Hint: How can sorting help you? Suppose you have two students with heights h_1, h_2 and two pairs of skis with length s_1, s_2, what's the optimal matching?

1. First, show that if $m = n$, a simple greedy algorithm solves the problem in $\mathcal{O}(n \log n)$ time. Be sure to give an argument about the running time. [1]
   If $m = n$, we just sort the students and the skis in increasing heights and lengths and assign correlatively.

   Both sortings take $\mathcal{O}(n \log n)$ time each, and the matching take $\mathcal{O}(n)$ time, so the algorithm has a total runtime of $\mathcal{O}(n \log n)$.

2. Prove that your greedy algorithm is correct. [0.5] The proof follows from the following lemma, that we will also use for when $m \neq n$.

   Lemma: For any pair of skis with $s_i < s_j$ and any pair of students with $h_i < h_j$, there exists an optimal solution that assigns s_i to h_i, s_j to h_j.

   Proof. **Case** $s_i \leq h_i < h_j \leq s_j$:
   1. If we match s_i to h_i and s_j to h_j then $h_i - s_i + s_j - h_j = (s_j - s_i) - (h_j - h_i)$.
   2. If we match s_i to h_j and s_j to h_i then $h_j - s_i + s_j - h_i = (s_j - s_i) + (h_j - h_i)$.
   As $(h_j - h_i) > 0$, the matching (1) costs less than (2).

   Case $s_i \leq h_i \leq s_j \leq h_j$:
   1. If we match s_i to h_i and s_j to h_j then $h_i - s_i + h_j - s_j = (h_j - s_i) - (s_j - h_i)$.
   2. If we match s_i to h_j and s_j to h_i then $h_j - s_i + s_j - h_i = (h_j - s_i) + (s_j - h_i)$.
   As $(s_j - h_i) \geq 0$, the matching (1) costs less than (2).

   In the same manner we can argue with the case $s_i < s_j \leq h_i < h_j$. $\square$

   Now we focus on giving a dynamic program for the general case when $m > n$.

3. Give a high-level description of each entry in the recurrence used by your algorithm. [0.5]

   Let $S[i,j]$ be the optimal cost of matching the first j pairs of skis with the first i students.

4. Give a formal definition of the recurrence you defined in part (1). [1]

$$S[i,j] = \begin{cases} 0 \qquad If i = 0 \text{ or } j = 0 \\ \min_i(S[i,j-1], S[i-1,j-1] + |h_j - s_i|) \qquad If 1 \leq i \leq j \\ \infty \qquad If i > j > 1 \end{cases}$$

5. Give a dynamic programming algorithm implementing the recurrence relation. Give a brief, high-level description of your algorithm, a pseudocode implementation, and an argument about the running time. [2]

   $S[1...n, 1...m]$ is a table with n rows (students) and m columns (skis). We sort both lists in increasing order of heights and lengths, and at every stage i,j decide whether it's best to assign the j^{th} pair of skis to the i^{th} students or not.

Algorithm 2 Ski Rental

```
Input: : A list $A = \{s_1, s_2, ..., s_m\}$ where $s_i$ denotes the length of $i^{th}$ pair of skis and a
    list $B = \{h_1, h_2, ..., h_n\}$ where $h_j$ denotes the height of the $j^{th}$ student.
Output: : The minimum possible cost to match $n$ pairs of skis to the $n$ students.
    $n \leftarrow |A|$, $m \leftarrow |B|$
    Sort $A$ and $B$ by increasing height and length.
    for $i \leftarrow 0$ to n do
        $S[i,0] = 0$
    end for
    for $j \leftarrow 0$ to m do
        $S[0,j] = 0$
    end for
    for $i \leftarrow 0$ to n do
        for $j \leftarrow 0$ to m do
            if $j < i$ then
                $S[i,j] = \infty$
            else
                $S[i,j] = S[i,j-1], S[i-1,j-1] + |h_j - s_i|$
            end if
        end for
    end for
    return $S[n,m]$
```

Complexity: We have $\mathcal{O}(n \log n + m \log m)$ time for both sortings, and $\mathcal{O}(nm)$ time for the last loop. Notice however that we don't need to store the whole table: The bottom "triangle" is all ∞'s and the top one is all 0's. Therefore, it suffices to compute a table of $(m-n)m$ entries. And so the total running time is $\mathcal{O}(n \log n + m \log m + (m-n)m)$ time.

6. Prove that your recurrence relation from Part (2) is correct. [2]

Consider the first i students and the first j pairs of skis. To write the recurrence defining $S[i,j]$, we are interested in the j^{th} pair and the i^{th} skier. We have two case:
1. If the optimal solution for i, j does not use the j^{th} pair, then $S[i,j] = S[i, j-1]$.
2. Otherwise, if we use the j^{th} pair, then by the Lemma above, it is best to assign it to the i^{th} students, since both input lists are sorted in increasing order of height and length. And thus $S[i,j] = S[i-1, j-1] + |h_j - s_i|$
The recurrence computes both cases and returns the smallest of the two.