Name: **EID:**

Exam #1 Practice Problems

Instructions. WRITE LEGIBLY.
No calculators, laptops, or other devices are allowed. This exam is **closed book**, but you are allowed to use a **one-page** reference sheet. Write your answers on the test pages. If you need scratch paper, use the back of the test pages, but indicate where your answers are. Write down your process for solving questions and intermediate answers that **may** earn you partial credit.

If you are unsure of the meaning of a specific test question, write down your assumptions and proceed to answer the question on that basis. **Questions about the meaning of an exam question will not be answered during the test.**

When asked to describe an algorithm, you may describe it in English or in pseudocode. If you choose the latter, make sure the pseudocode is understandable.
You have **75 minutes** to complete the exam. The maximum possible score is 100.

Some useful information:

Logarithms and Factorial:

$$\log(n!) = \Theta(n \log n)$$

Problem 1: Stable Job Offers

Consider a workforce economy made up of n available jobs and n people looking for work. Each worker has a list of preferences for jobs, and each job ranks all of the available workers. There are no ties in these lists. Half of the jobs are full-time jobs and half of the jobs are part-time jobs. We assume that every worker prefers any full-time job over any part-time job. Further, half of the workers are "hard workers" and the other half are "lazy." We assume that every employer prefers any hard worker over any lazy worker.

We assume a definition of stability identical to that we proved for stable marriage: a matching of workers to jobs is stable if it is perfect (i.e., every worker gets a job and every job gets filled) and there are no instabilities (i.e., there does not exist two pairs (j, w), (j', w') such that the employer offering job j prefers worker w' to w and w' also prefers jon j to j'.

Prove that, in every stable matching of workers to jobs, every hard worker gets a full time job.

Solution

Suppose not. Then there exists some stable matching of workers to jobs in which a hard worker (let's call h_w) is matched to a part time job. Because one of the hard workers has been given a part time job, and because there are equal numbers of hard workers, lazy workers, full time jobs, and part time jobs, then it must therefore be true that some lazy worker got a full time job (let's call this job j_f). But by the structure of the problem, h_w prefers j_f to the part time job he was assigned, and the employer with job j_f prefers h_w to the lazy worker it was assigned. This is exactly an instability, leading to a contradition.

Problem 2: Asymptotic Time Complexity

Consider each of the following pairs of functions. For each pair, either $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$ or $f(n) = \Theta(g(n))$. Determine which of these three options best captures the relationship and (briefly) explain or demonstrate why.

(a) $f(n) = \log n^2$; $g(n) = \log n + 5$

> **Solution**
>
> $f(n) = \Theta(g(n))$. Why? $\log n^2 = \log n$

(b) $f(n) = \sqrt{n}$; $g(n) = \log n^2$

> **Solution**
>
> $f(n) = \Omega(g(n))$. Why? Polynomials grow faster than logs.

(c) $f(n) = \log^2 n$; $g(n) = \log n$

> **Solution**
>
> $f(n) = \Omega(g(n))$. Why? Polylogarithmic functions grow faster than logs.

(d) $f(n) = n$; $g(n) = \log^2 n$

> **Solution**
>
> $f(n) = \Omega(g(n))$. Why? Polynomials grow faster than logs.

(e) $f(n) = n \log n + n$; $g(n) = \log n$

> **Solution**
>
> $f(n) = \Omega(g(n))$. Why? Polynomials grow faster than logs.

(f) $f(n) = 10$; $g(n) = \log 10$

> **Solution**
>
> $f(n) = \Theta(g(n))$. Why? They're both constants.

(g) $f(n) = 2^n$; $g(n) = 10n^2$

> **Solution**
>
> $f(n) = \Omega(g(n))$. Why? Exponentials grow faster than polynomials.

(h) $f(n) = 2^n$; $g(n) = 3^n$

> **Solution**
>
> $f(n) = O(g(n))$. Why? Consider the limit. $f(n)/g(n) = 0$ in the limit.

Problem 3: Decision Tree Lower Bounds

You are given a sequence of n elements to sort. The input sequence consists of n/k subsequences, each containing k elements. The elements in a given subsequence are all smaller than the elements in the succeeding subsequence and larger than the elements in the preceding subsequence. Thus, all that is needed to sort the whole sequence of length n is to sort the k elements in each of the n/k subsequences. Consider the following (simple) example, where n is 15 and k is 5: $\{2, 5, 4, 3, 1\}$, $\{8, 6, 9, 7, 10\}$, $\{11, 15, 12, 14, 13\}$.

(a) Show an $\Omega(n \log k)$ lower bound on the number of comparisons needed to solve this variant of the sorting problem.

Solution

We construct a decision tree for this variant of the sorting problem and show that it has height at least $n \log k$. Each leaf of the tree corresponds to a permutation of the original sequence. Each subsequence has $k!$ permutations, and there are n/k subsequences, so there are $(k!)^{\frac{n}{k}}$ permutations of the whole sequence. Thus the decision tree has $(k!)^{\frac{n}{k}}$ leaves. A binary tree with $(k!)^{\frac{n}{k}}$ leaves has height at least $\log((k!)^{\frac{n}{k}})$.

$$\log((k!)^{\frac{n}{k}}) = n/k \log(k!)$$

$$\log((k!)^{\frac{n}{k}}) = \Theta(n/k * k \log k)$$

$$\log((k!)^{\frac{n}{k}}) = \Theta(n \log k)$$

Therefore the lower bound on the number of comparisons needed to solve this variant of the sorting problem is $\Omega(n \log k)$.

(b) Design an optimal algorithm for performing this sorting task.

Solution

Merge-sort each of the subsequences. And then just concatenate them. Each merge-sort takes $O(k \log k)$. We have to do n/k such merge-sorts. Concatenation could be constant or take as much as $O(n)$ depending on storage. Therefore, we get $n/k * O(k \log k) + O(n)$ in the worst case. This is $O(n \log k) + O(n) = O(n \log k)$.

Scratch Page