Review for Algorithms Midterm Exam #1
Computer Science 140 & Mathematics 168
Harvey Mudd College
Computer Science Department
Instructor: B. Thom
Fall 2004

General Info:

- The first midterm will be given out on Tuesday, October 5, at the end of class and will be due back on Thursday, October 7, at the *beginning* of class. If you wish to download the corresponding LaTeX files, you will be able to use `scp` to get them, e.g.

  `scp bthom@cs.hmc.edu:public_html\cs140_exam.zip`.

  This file will only be available for copy within the dates that the exam can be open. After you are done taking the exam, you should destroy these files.

- The exam is closed-book and closed-notes, but you may prepare an 8×11 crib sheet (one side) with anything you like written on it (and this may be LateX'd if you like). Turn in your crib sheet (if you're using one) with your exam.

- This exam will be a comprehensive, covering all the topics we've discussed so far, including Dynamic Programming. Problems will be similar in spirit to those you've seen on prior homeworks, requiring a creative application of techniques presented in class to new situations. As a bare minimum, preparing should involve reviewing the solution sets you've gotten, focusing on their various derivations (as opposed to memorizing things).

- The exam will be designed to be completed in 2 to 3 hours, but you may spend as much time as you'd like within the open dates. I recommend you get your prep-work done before class on Tuesday so that when you receive the exam you can familiarize yourself with the problems immediately. Then, you can mull over the problems for an evening or so before you actually sit down to solve them and write up your answers. (Its amazing what inspiration has flashed upon students while showering!)

- You may not look previous years exams when preparing. However, some practice problems (taken from, among other places, prior years' exams) are included here if you'd like to try a few of them.

- Tuesday's lecture will be dedicated to answering any questions you might have—about the practice problems, topics covered in class, how to produce the best tasting biscotti and brew the best Latte, etc. When you run out of questions, I'll be out of answers :-).

Rough outline of topics covered so far:

1. Understand asymptotic notation. A good way to make sure you understand the differences between Ω, $\mathcal{O}$, and Θ is to derive why $\log n! \in \Theta(n \log n)$. Appreciate that bounds are always with respect to something (e.g. best, worst, average). Remember that for the lower bound constants, n_0 and c must often determined by hand. Why are lower bounds so valuable and what makes them so valuable? Why are upper bounds typically easier to derive? Why (and under what conditions) are restricted run-times (e.g. assuming that $n = a^k$) more generally applicable? Be comfortable with the thought processes associated with ranking various functions asymptotically.

2. Understand divide-and-conquer. Know how to set up these problems' recurrence relations and solve them using the recursion tree method. Understand how to deal with trees that aren't balanced and understand when a tree's height has no impact on run-time (e.g. recall the order statistics recursion when sorts are done on groups of 3 versus 5). Appreciate why a divide-and-conquer algorithm's runtime won't change when only its "combine" step is modified (and modified in such a way that it is no more asymptotically expensive than it was before modification).

3. Understand sorting and order statistics – insertion sort, heap-sort, quick-sort, counting sort, radix sort. Know how to analyze the runtime and argue the correctness of a particular sorting paradigm (often this requires that one argue a particular invariant is maintained throughout an algorithm's execution using induction). Which sorting algorithms rely on a partitioning technique and what is this (p.s. by he way, partitioning *can* be done in place, see page 146 of CLRS)? Why are heaps used so often (reason one: sorting, reason two: think certain very useful abstract data type operations!).

4. Understand how to derive a lower-bound for a comparison-based sorting algorithm (as well as other similar paradigms, e.g. HW 2b, Problems 4 and 5). Appreciate the difference between a particular algorithm's lower bound and a particular task's lower bound.

5. Understand dynamic programming. When can you use it to make a computation more efficient and when is it unable to help (recall the basic "components" required for dynamic programming to be useful, Section 15.3 in CLRS). As we did in class and on numerous HWs, know how to derive a worst-case running time for a recursive solution and be able to contrast this with a corresponding more-efficient dynamic programming solution. Aside: what does the term "programming" refer to here? Table lookup!

6. Understand abstract data types (ADT), e.g. heaps, queues, (dynamic) arrays, linked lists, (balanced) binary search trees, etc. Understand why Red-Black and B- trees are height balanced.

7. Understand how to use amortization to make an ADT look as good as possible. We looked at two techniques: aggregation, which charges a worst possible sequence of

n operations exactly and uses this to derive an average-cost fee per operation, and amortization, which assigns fees to operations up front and then argues why these fees are sufficient to pay for any sequence of n operations.

Some practice problems on these topics:

1. **Quicksort is Correct!**

   Use induction to prove that the Quicksort algorithm we saw in class is correct.

2. **A Quicksort Promise!**

   Recall that Quicksort had the recurrence relation

   $$T(1) = cT(n) = T(a-1) + T(n-a) + cn$$

   where a depends upon what value the pivot element has. Describe how Quicksort can be modified slightly to choose a different pivot so that the worst-case running time now becomes $O(n \log n)$ and carefully explain how this changes the recurrence relation so that $O(n \log n)$ results.

3. **Divide-n-Conquer!**

   Professor Hugh G. Messe has a recurrence relation that looks like:

   $$T(n) = aT(\frac{n}{b}) + cn$$

   where $T(1) = c$ and a and b are positive integers such that $a > b$.

   (a) Use the recurrence tree method to find a closed-form solution for the recurrence relation *under the assumption that* $a > b$. You may assume that n is some power of a particular integer. Show the details of your work! Don't skip steps in your analysis! Conclude by giving an asymptotic (Big-Oh) expression for $T(n)$ in the simplest form possible.

   (b) Does your analysis apply if $a = b$? If so, explain briefly. If not, what step of your analysis breaks down when $a = b$?

4. **Marxsort!** Professors Groucho, Chico, and Harpo of the Massachusetts Institute of Typography have been developing their own fast sorting algorithms.

   (a) Professor Groucho tells his colleagues that, under the assumption that a number can be converted from one base to any other in constant time, he has discovered a way to sort n integers in the range 1 to n^2 in $O(n)$ time. Explain how this can be done. Why doesn't Counting sort work here? (Go back and make sure you are completely happy with how Counting sort and Radix sort work and how much time they take.)

(b) Professor Chico announces that under the same base conversion assumption, he can sort n integers in the range 1 to n^k in $O(n)$ time for *any* fixed constant k. Explain how this too can be done.

(c) Professor Harpo is very excited! "We should be able to sort n integers in *any* range in $O(n)$ time," he mimes. Explain why the Groucho-Chico trick doesn't generalize as Professor Harpo suggests.

5. **Merge Lower Bounds!**

The problem of merging two sorted lists arises frequently. For example, it is used as a subroutine in Mergesort. One reasonable implementation of Merge (c.f. CLR pg. 29) is $\Theta(n)$.

In this problem, you will show that there is a lower bound of $2 \cdot n - 1$ on the worst-case number of comparisons required to merge 2 sorted lists, *each* containing n items.

First, you will show that there are $2 \cdot n - o(n)$ comparisons as follows:

(a) Show that, given $2 \cdot n$ numbers, there are $\binom{2n}{n}$ possible ways to divide them into two sorted lists, each with n numbers.

(b) Using a decision tree, show that *any* algorithm that correctly merges two sorted lists uses at least $2 \cdot n - o(n)$ operations. Note: One of the tricks with this problem is that care needs to be made in coming up with useful lower-bound assumptions—if you're "too sloppy" you can come up with useless results (e.g. $T(n) \geq$ some negative number).

6. **Sorting Variable-Length Items**

Solve Exercise 8.3 of Cormen *et al.* second edition (page 179).

7. **Hurts Car Rental!**

Hurts Car Rental is experimenting with a new type of vehicle. These vehicles use a modular fuel pack which allows the vehicle to travel exactly 100 miles. (The fuel is believed to consist of a mixture of pulverized Spam, Mountain Dew, and Cheetos, although the technology is proprietary and this is just speculation.) When the car needs to be refueled, the fuel pack is removed from the car and is replaced by a new one. Thus, it is possible that a fuel pack will be replaced before it is completely used up. Moreover, the driver gets no credit for the remaining fuel in the fuel pack and can only purchase a new fully charged 100 mile pack.

A driver may wish to travel on a long freeway from one point to another. Since only designated service stations sell the fuel packs, the driver needs to plan carefully where to refuel. Moreover, each service station charges a different amount for a fuel pack. Some drivers also don't want to stop very often to refuel. Consequently, these vehicles have a dial on the dashboard called the α dial. By turning the dial, the driver sets a value of α between 0 and 1. By doing so, the driver stipulates that she wishes to minimize the quantity $\alpha S + (1-\alpha)C$ where S is the number of fuel stops made and C is the total cost paid for the fuel packs. Notice that when α is set to 0, the objective becomes that of minimizing the total cost paid for the fuel packs. When α is set to 1, the objective becomes that of minimizing the number of stops. When α is set somewhere between 0 and 1, the objective is a linear combination of these.

Consider a highway represented by a line segment. Let $p_1, p_2, \ldots, p_n$ be n points on the line segment sorted from left to right where p_1 is the starting point, p_n is the destination point, and each of these points has a service station selling fuel packs. Let d_i be the distance from point p_1 to point p_i, for $1 \leq i \leq n$. Let c_i be the cost of a fuel pack at the station at point p_i.

(a) Professor I. Lai claims that the following greedy algorithm minimizes the total cost when $\alpha = 0$: Buy a fuel pack at p_1 (we have no choice there - we need fuel to depart on the trip). Travel to the furthest point which is at most 100 miles away. Purchase a fuel pack there. Now repeat this process of traveling as far as possible on the current fuel pack before purchasing a new fuel pack. Explain briefly why this approach does not guarantee an optimal solution when $\alpha = 0$.

(b) Assume that a given value of α has been established by the driver. Give pseudo-code for a recursive algorithm called `optimize(`$[p_1, p_2, p_3, \ldots, p_j]$`, `$p_k$`)` which returns the minimum value of $\alpha S + (1-\alpha)C$ for a trip from p_1 to p_k with fuel stops permitted at any of the consecutive service stations between p_1 and p_j. You should assume that $j < k$. Moreover, we must always purchase a fuel pack at p_1 to start the trip and this counts as one fuel stop. (Notice that once you've written this function, we can call it with `optimize`$([p_1, \ldots, p_{n-1}], p_n)$ to get our desired solution!)

Your pseudo-code will need to use the 100 mile travel limit per fuel pack and will need to refer to the d_i values which give the distances from p_1 to p_i and the c_i values which specify the cost of a fuel pack at point p_i. Finally, if there is no possible way to get from p_1 to p_k due to the distances between the given fuel stations, the function should return the value ∞.

(c) Describe how your algorithm could be converted into a dynamic programming algorithm. In particular:

i. Describe what the dynamic programming table looks like and its dimensions.

ii. Describe the order in which the cells in the table are filled in and the rule used to fill in each cell. Be sure to explain how to fill in the "easy" cells at the beginning as well as the rule for filling in the other cells.

(d) What is the running time of your dynamic programming algorithm? Explain very briefly.