CS41 Lab 8: Greedy "Staying Ahead" and Recurrence Relations

In typical labs this semester, you'll be working on a number of problems in groups of 3-4 students. You will not be handing in solutions; the primary purpose of these labs is to have a low-pressure space to discuss algorithm design. However, it will be common to have some overlap between lab exercises and homework sets.

The goal for lab this week is to get some additional practice with greedy algorithms, as well as to get some practice working with recurrence relations.

1 Greedy Algorithms and "Staying Ahead"

In class, we looked at the interval scheduling problem: We are given n requests, numbered 1 through n. We are given for each request i a start time $s(i)$ and a finish time $f(i)$. We saw several greedy approaches that turn out not to work, but the "earliest finish time last" rule does work.

In class, we saw the basic idea to the proof of its correctness. Now, we will work out the details carefully. Suppose the greedy algorithm chooses the requests $i_1, i_2, i_3, \ldots, i_k$, in this order (so i_1, being chosen first has the earliest finish time, then i_2, and so on, so that $f(i_1) < f(i_2) < \ldots f(i_k)$. Then consider an optimal solution that contains m requests $j_1, j_2, \ldots, j_m$, also given in order of finishing time (so that $f(j_1) < f(j_2) < \cdots < f(j_m)$).

First, note that since $j_1, j_2, \ldots, j_m$ is optimal, no valid schedule can include more than m requests, so $k \leq m$; our goal will be to prove that in fact $k = m$ (so that our greedy schedule has just as many requests as an optimal schedule, and is thus itself optimal).

1. In class, I claimed that for all positive integers $r \leq k$, $f(i_r) \leq f(j_r)$. Prove this by induction on r.

2. Use part (a) to prove that $k = m$, so the greedy algorithm is optimal.

   (*Hint:* Use a proof by contradiction. Suppose that $m > k$, then consider part (a) with $r = k$. What can we conclude about request j_{r+1}?)

3. Our textbook calls this argument the "greedy stays ahead" style of argument. In what sense does the greedy algorithm "stay ahead"?

4. The proof of correctness for Dijkstra's Algorithm is based on proving that, if R is the "known region" of vertices whose `dist` values have been fixed, then every time it selects a node v to add to R, the path the algorithm has found at this point is shorter than every other path to v (see (4.14) in Kleinberg-Tardos).

   In what sense can the correctness argument for Dijkstra's Algorithm also be considered a "stays ahead" argument?

2 Recurrences, Divide and Conquer

The following equalities might be helpful:

- For any $a, b > 0$, $\log(a^b) = b \log a$.
- For any $a, b \geq 1$, $a^{\log b} = b^{\log a}$.
- For any $a, b \geq 1$, $\log(a \cdot b) = \log(a) + \log(b)$.
- For any $a, b \geq 1$, $\log(a/b) = \log(a) - log(b)$.
- For any $x \in \mathbb{R}$, $2^{\log x} = x$.

The following summation appears often in recurrence relations.

Fact 1. *Fix any constant $c > 0$. For all $m > 0$, we have*

$$\sum_{k=0}^{m-1} c^k = \frac{c^m - 1}{c - 1} \ .$$

Recursion Tree Recap

In lecture, we considered the recurrence relation that describes an upper bound on the running time for mergesort:

$$T(n) = 2T(n/2) + cn$$
$$T(1) = c$$

where $c > 0$ is some constant. This comes from the fact that mergesort, in the recursive case, splits the list exactly in half and recursively calls mergesort on each (so we have two lists of size $n/2$, and we call mergesort on each, which takes time $T(n/2)$, hence the $2T(n/2)$); and from the merging step that takes linear time (hence the $+ cn$).

We saw how to analyze this recurrence using a Recursion Tree. We also looked at similar cases. Let's consider the general form:

$$T(n) = aT(n/2) + cn$$
$$T(1) = c$$

where mergesort is the case of $a = 2$, and we also considered $a = 1$ and $a = 3$ in class.

Below is a table that is useful for working out the total running time using the Recursion Tree method.

Level	Number of nodes in level	Input size at each node	Work per node	Total work at level
0				
1				
2				
ℓ				

1. Consider the case where $a = 4$. Fill out the table above, determine how many total levels are in the tree, and conclude the final running time in terms of Big-Oh.
2. Now, consider the recurrence $T(n) = aT(n/2) + n^2$. What is the running time when $a = 2$? What about $a = 4$? $a = 5$?

More Recurrences

Another method for solving recurrences is the Substitution Method, where we repeatedly substitute the recurrence in for itself to try to reach a base case, and look for a pattern. We will see an example of this.

Try to use either the Substitution Method or the Recursion Tree Method to solve these recurrence relations. You do not need to solve these exactly, only asymptotically. e.g. if a recurrence relation has an exact solution of $T(n) = 3n^2 - 19\sqrt{n}$, then an answer of $T(n) = O(n^2)$ suffices.

1. $S(n) = S(n-1) + 3n$,
   $S(1) = 3$
2. $M(n) = 3M(n/2) + 2n$,
   $M(1) = 1$
3. $W(n) = 3W(n/3) + n^2$,
   $W(1) = 1$
4. $H(n) = 4H(n/2) + 2n^2$,
   $H(4) = 5$
5. $T(n) = 3T(n/3) + 10\sqrt{n}$,
   $T(1) = 5$.