CS180: Homework 5, Section 1B

Ryan Sharif 204-351-724

February 17, 2016

1 Significant inversions

We can almost use the same algorithm introduced in this chapter for counting the number of inversions in a sequence of non-decreasing integers. That is, we implement a divide-and-conquer merge-sort algorithm. For this problem, however, we need to keep track of the positions that are more than twice as far. Thus, begin the algorithm as we did in the chapter: Divide the sequence in half, and recursively do so until our sequence is sorted, i.e., there is only one item in each subsequence.

We then begin the task of merging our sequence back together. Comparing the two lists we have in front of us, keep a pointer pointed at the beginning of each list. If we merge from the right, and if $a_i > 2a_j$, we increment our significant inversion variable. If not, then we just merge. We continue to do this until we have completely merge our sequence. We can discard the sorted list and maintain the variable that contains the significant number of inversions. And since, we haven't added any more work than our original algorithm, this runs in $O(n \log n)$.

2 Bank fraud

2.1 Overview of the problem

The biggest problem that our bank faces is that it does not have a way of recording the account numbers read from our advanced 'equivalence tester', i.e., it only looks at two cards and reports *true* or *false*. We can still find out whether there is a set of cards greater than $\frac{n}{2}$ that are equivalent.

We begin by stating that if there is an equivalence majority in the set, and if we pair up every card in the set, at least one pair must have two such equivalent cards. We can use a proof by contradiction to show why this is so. Suppose that we have paired each card in the set but there is not a set with two equivalent cards x_1, x_2, then if there were an equivalence set of cards, there would be a matching non-equivalence card for each equivalence card, since we assumed that no pair had such an equivalence.

Then there are no more than $\frac{n}{2}$ cards that share an equivalence relation, since we have matched each card with one that is non-equivalent. But our bank wants to know if such an equivalence set exists, and since there are no more than $\frac{n}{2}$ equivalence cards, we have a contradiction, since we said we are looking at an equivalence majority, where such as class has greater than $\frac{n}{2}$, but we don't have at least $\frac{n}{2}$ cards.

2.2 Algorithm

Given the facts above, we only need to find such pairs of equivalent cards.

- *pair up every card in our set*
- *check each pair of cards for equivalence*
- *if neither card is equivalent, throw both cards in a GOOD pile*
- *if both cards are equivalent, throw one in a BAD pile, discard the other*
- *repeat step 2 on our BAD pile*

- *if one card remains we have such a BAD equivalence class*
- *check each card in our entire set with the BAD card*
- *if we have $\frac{n}{2}$ bad cards, we have found the fraud*
- *else, there was no such fraud*

2.3 Runtime

We will perform $\frac{n}{2}$, initial comparisons to derive a bad pile. We will perform a maximum of $\log n$ comparisons on our bad pile. Finally, we will perform a final test on $(n-1)$ cards. Thus, we have $O(\frac{n}{2} + \log n + (n-1))$ or $O(n)$.

3 This tree has a minimum

3.1 A little lemma

Before providing an $O(n \log n)$ algorithm that solves this problem, lets consider a lemma that will help us. Since we have a complete binary tree, we either have the case where a parent node is less than the value of its two child nodes, or one of its child nodes is less than the value of its parent node. We can use a proof by contradiction to see why this is so. Suppose for the purpose of contradiction, this is not so, that is suppose that a parent p has a value less than its two child nodes. Furthermore, suppose that one of the child nodes c_x is less than its sister node c_y and its parent node. Then, we have $p < c_x$ and $p < c_y$ and $c_x < p$ and $c_x < c_y$. But, this is a contradiction, since we cannot have $p < c_x$ and $c_x < p$. With this fact now in hand, we can write our algorithm:

3.2 Algorithm

- *Begin at the root node*
- *If the root node is smaller than both its children, then*
  - *return the root node*
- *Else if, a child node is smaller than its parent, then*
  - *check this nodes children*
  - *If both children are larger then return this node*
  - *Else, return to step 3*
- *If there are no child nodes, then*
  - *we have arrived at a leaf, return this node*

3.3 Correctness

We can use our little lemma above to see why our algorithm will always return a local minimum. Recall that a local minimum is one such that its label x_v is less than the label x_w for all nodes w connected to a node v by an edge. Every node in our complete binary tree will have a parent node, except for the root node.

Suppose we have arrived at our i^{th} step in the algorithm. By our lemma above, we will either have a parent whose value is smaller than its children, or a child node that is smaller than its sister and parent node. In the former case, our algorithm will return the parent node, and since we got to this node because it was smaller than its parent node, we have a local minimum. Similarly for a child node c, if its children are greater than c, we return c, since we have found a local minimum.

Finally, if we reach a leaf, then we got there because it was smaller than its parent node. And since by definition, a leaf node has no children it must be smaller than all nodes it is connected to, namely its parent node.

3.4 Runtime

Our algorithm runs within the specification of the problem, $O(\log n)$, since at every point, we make a decision to follow the binary tree in one direction, or return the current node. Since we never examine more than one path in the tree, our algorithm performs at most $\log n$ probes.

4 Two eggs

We now turn to our problem of dropping two eggs from a building. From the outset, we can consider what we would do if we only had one egg. The only way to ensure a reliable measure for the egg would be to drop it from each floor, one by one starting from the first floor. With two eggs, we could improve our speed by climbing floors in multiples of two, retreating to the previous floor when our first egg broke to determine if the maximum height was the previous floor, and not two floors below.

We can do better than this simple multiplier by trying several other multipliers. In each case, we will find floors where the multiplier achieves fewer steps than others but performs worse on other floors:

floor	2x	5x	10x	20x	25x
8	5	5	9	9	9
23	13	8	6	5	24
47	25	12	12	9	22
71	37	16	9	15	24
89	46	22	18	14	18

Thus, it appears that there is no optimal solution, if we arbitratily assign a multiplier. What we'd like to do is set a maximum number of drops no matter where any egg should break between the first floor and the last floor. We can achieve such an even spacing by noting that everytime we drop our first egg we are using up our maximum number of drops, decreasing it by one. We can express this decrease in available drops using the equation: $n + (n-1) + (n-2) + ... \leq 100$. This looks like the formula for obtaining the sum of the first n integers: $\frac{n(n+1)}{2}$, which we can solve:

$$\frac{n(n+1)}{2} = 100 \tag{1}$$

$$n(n+1) = 200 \tag{2}$$

$$n^2 + n = 200 \tag{3}$$

$$n^2 + n - 200 = 0 \tag{4}$$

Using the quadratic formula, we get a number greater than 13. Thus, if we space our drops in increments starting from 14 and decreasing the maximum floor by 1 and returning to our last known good drop if our first egg breaks, we arrive at the maximum number of drops for any n, such that $1 \leq n \leq 100$ is 14.

5 Local minimum on a chess board

Although this problem deals with a chess board, we can convert it into a problem where we want to corral a local minimum. That is we want to search the easiest place first, the place that takes the least amount of work and divide and conquer after that. We know that if there is a local minimum on the outside edge of our graph, we will perform fewer probes, since every outer node is connected to a maximum of three other nodes, the corner nodes only connect to two other nodes.

Thus, we perform a search for the local minimum on the outside edge of our graph G, which takes $O(n)$ probes. If our local minimum, is there, we're done. Else, a local minimum is foud on the inside, of the graph. We can demarcate another edge to find a local minimum, by searching the middle horizontal row of our graph G. This, search again, can accomplished in $O(n)$ probes, and if our local minimum is there, we are done.

Finally, if a local minimum is not in our 'fence', then one must reside in one of our 'fence' quarters. Perform a search for a local minimum in an arbitrarily chosen fence. Since, our local minimum was not found on the 'fence', i.e., since $v \notin F$, there must exist one inside the quarter. Suppose, not, that is suppose there is no local minimum inside the quarter. Then, that means for every node $w \in Q$, that is our quarter, there is a smaller $v \in F$, but we just said $v \in F$, thus, we have a contradiction. Searching a quarter of our graph, also takes $O(n)$ time. Thus, the maximum number of probes is $O(3n)$ or $O(n)$.