COMPSCI 3AC3

Assignment 1

January 23 2022

1 Part I

Exercises 3, 4, 5, 8 from Chapter 2.

1.1 Q3

The ordering of these function from smallest to largest is as follows:

f_2, f_3,f_6, f_1, f_4, f_5

1.2 Q4

From smallest to largest the right ordering is: $g_1, g_5, g_3, g_4, g_2, g_7, g_6$.

1.3 Q5

(i) False, as it is possible that $g(n) = 1$ for all $n, f(n) = 2$ for all n, subsequently $\log_2 g(n) = 0$, therefore $\log_2 f(n) \leq c \log_2 g(n)$ cannot be written. Conversely, if we need $g(n) \geq 2$ for all n beyond n_1, this then remains true. Considering $f(n) \leq cg(n)$ for all $n \geq n_0$, then $\log_2 f(n) \leq \log_2 g(n) + \log_2 c \leq (\log_2 c) (\log_2 g(n))$ once $n \geq \text{mrax}(n_0, n_1)$.

(ii) False. Consider $f(n) = 2n$ and $g(n) = n$. Thus, $2'^{(n)} = 4''$, and $2^{g(n)} = 2^n$.

(iii) True. As $f(n) \leq cg(n)$ for all $n \geq n_0$, we have $(f(n))^2 \leq c^2(g(n))^2$ for all $n \geq n_0$.

1.4 Q8

(a) Let us assume for a moment that n is a perfect square. The first jar is dropped from heights of multiples of $\sqrt{n}$, until it breaks. If it is dropped from the top and doesn't break from height, then no more work needs to be done. However, if it breaks from a height $j\sqrt{n}$, we then know that the highest safe rung lies between $(j-1)\sqrt{n}$ and $j\sqrt{n}$, as such the second jar is dropped from rung $1 + (j-1)\sqrt{n}$ upward, increasing by one progressively. We drop each of the two jars at most $\sqrt{n}$ times, for a total of at most $2\sqrt{n}$. If n isn't a perfect square, the first jar is dropped from heights which are multiples of $\lfloor \sqrt{n} \rfloor$ and then subject the second jar to the aforementioned rule. Thus, the first jar is dropped at most $2\sqrt{n}$ times and the second jar at most $\sqrt{n}$, which is bounded by $O(\sqrt{n})$.

(b) By induction, $f_k(n) \le 2kn^{1/k}$. The first jar is dropped from heights of multiples of $\lfloor n^{(k-1)/k} \rfloor$. Thus, the first jar is dropped at most $2n/n^{(k-1)/k} = 2n^{1/k}$ times, which reduces the sets of potential rungs to intervals of length of at most $n^{(k-1)/k}$.

For $k-1$ jars, we apply this process recursively. We claim by induction that it uses at most $2(k-1)\left(n^{(k-1)/k}\right)^{1/(k-1)} = 2(k-1)n^{1/k}$ drops. Completing the inductive step, adding $\le 2n^{1/k}$ drops by the first jar, we obtain a bound of $2kn^{1/k}$.

2 Part II

Exercises 2, 3, 4, 6, 7 from Chapter 5.

2.1 Q2

We can solve this by defining a recursive divide and conquer algorithm DC, taking a sequence of distinct numbers $a_1, \dots, a_n$ returning N and $a'_1, \dots, a'_n$ where N represent the number of significant inversion and $a'_1, \dots, a'_n$ represents the sequence in an increasing order.

The formal definition of this algorithm is as follows, where for $n = 1$ DC returns $N = 0$ and $\{a_1\}$ and for $n > 1$:

- $k = \lfloor n/2 \rfloor$
- Recursive call to DC $(a'_1, \dots, a'_k)$. Returns $b_1, \dots, b_k$.
- Recursive call to DC $\left(a'_{k_1}, \dots, a'_n\right)$. Returns N_2 and $b_{k+1}, \dots, b_n$.
- Calculate the amount N_3 of significant inversions $\left(a_i, a_j\right)$ where $i \le k < j$.
- Return $N - N_1 + N_2 + N_3$ and $a'_1, \dots, a'_n - \text{MERGE}\left(b_1, \dots, b_k; b_{k+1}, \dots, b_n\right)$

For MERGE, a version of merge-count of $b_1, \dots, b_k$ and $2b_{k+1}, \dots, 2b_n$ can be implemented as follows.

- Instantiate counters: $i \leftarrow k, j \leftarrow n, N_3 \leftarrow 0$.
- If $b_i \le 2b_j \rightarrow$
  - if $j > k+1$ decrease j by 1
  - if $j = k+1$ return N_3
- If $b_i > 2b_j$, increase N_3 by $j - k$. $\rightarrow$
  - if $i > 1$ decrease i by 1
  - if $i = 1$ return N_3

For every i we increment N between b_i and all b_j. If $b_i \le 2b_j$ no significant inversions exist between b_i and any b_m s.t. $m \ge j$; j decreases. If $b_i > 2b_j$ then $b_i > 2b_m$ for all m s.t. $k < m \le j$. This means that we have found $j - k$

significant inversions with b_i; we increase N_3 by $j-k$. Once we reach $i = 1$ with b_1 the algorithm completes.

2.2 Q3

We can solve this problem with a divide and conquer solution. First, let $v_1, \dots, v_n$ denote the equivalence of the cards; cards a and b are equivalent if $v_i = v_j$. The problem is solved if we can find a value x such that more than $\frac{n}{2}$ of the indices have $v_i = x$. We first divide the set of cards into two approximately equal piles: one set of $\lfloor n/2 \rfloor$ cards and the other a set of $\lceil n/2 \rceil$ cards. On both sets the algorithm will run recursively, with the directive that, if it finds an equivalence class with more than half of the cards total, it returns a sample card in the class. If there are more than $\frac{n}{2}$ cards that are equivalent in the entire set; namely equivalence class x, then at the least one of the sides will possess cards with more than half equivalent to x. As such, one of the recursive calls will return a card with equivalence class x.

The reverse of this does not hold. It is possible for there to exist, on one side, a majority of equivalence cards without possessing more than $\frac{n}{2}$ cards in total. Therefore, if a card with a majority equivalence is found in either set of cards, we have to test this card with all the other cards.

The algorithm has two recursive calls, at most $2n$ tests performed outside recursive calls. The following recurrence is then derived:

$$T(n) \leq 2T(n/2) + 2n.$$

Which implies that $T(n) = O(n \log n)$.

2.3 Q4

We can solve this through a convolution. One vector is defined as $(q_1, q_2, \dots, q_n)$ and the other vector is defined as $b = (n^{-2}, (n-1)^{-2}, \dots, 1/4, 1, 0, -1, -1/4, \dots - n^{-2})$. For each j the convolutions a and b will have entries:

$$\sum_{i<j} \frac{q_i}{(j-i)^2} + \sum_{i>j} \frac{-q_i}{(j-i)^2}$$

We then multiply this term by Cq_j to obtain the net force F_j. This process takes $O(n \log n)$ time, and obtaining F_j takes $O(n)$ time.

2.4 Q6

First some definitions: - u is smaller than v, or $u \prec v$, if $x_u < x_v$. - If S is a set of nodes, $u \prec S$ if u is smaller than any node in S.

The algorithm is defined as follows: Starting at the root r of the tree, detect if r is smaller compared to its children. If this is proven true, the root is then a local minimum; else we proceed to smaller children and iterate from there. If a (1) node v is reached that is smaller than both children, (2) or a leaf w is reached, the algorithm terminates.

The runtime of this algorithm is in $O(d) = O(\log n)$ probes of the tree; with a local minimum as a return value, which is proven as follows:

- As explained above, if root r is return, then it follows that it is a local minimum
- If the algorithm terminates in case (1), v is determined to be a local minimum as v is smaller than its two children and its parent.
- In case (2) if we terminate, w is a local minimum because w is smaller than its parent.

2.5 Q7

First some definitions:

- B will signify the set of nodes on the border of the grid G
- G has Property $(*)$ if it has a node $v \notin B$ adjacent to a node B which satisfies $v \prec B$
- In grid G with Property $(*)$, the global minimum is not on the border B, as such G has at least one local minimum that is not on the border B. These can be denoted as a local minimum and an internal local minimum.

This problem can be solved with a recursive algorithm which takes a grid satisfying Property $(*)$ returning an internal local minimum, with $O(n)$ probes. Let G satisfy Property $(*)$, and $v \notin B$ is adjacent to a node B and smaller than every node in B. C will denote the union of nodes in the middle row and column of G (excepting nodes on border G). Let $S = B \cup C$; removing S from G allocates G into four sub-grids. Then let T denote all nodes adjacent to S.

With $O(n)$ probes, we detect a minimum value node $u \in S \cup T$. As $u \notin B$, since $v \in S \cup T$ and $v \prec B$; there are two cases. If $u \in C$ then u is an internal local minimum, as all neighbors of u are in $S \cup T$, u is also smaller than all of them; else $u \in T$. G' will denote a sub-grid with u, with parts of S on its border. As u is adjacent to the border of G' and is also smaller than all nodes on the border of G', G' satisfies Property $(*)$. Therefore, G' possesses an internal local minimum, that is an internal local minimum of G as well. The algorithm is recursively called on G' to find said minimum.

If $I'(n)$ represents the amount of probes required to detect an internal local minimum, the recurrence $T(n) = O(n) + T(n/2)$ occurs, which is $T(n) - O(n)$. To find a local minimum of grid G, using $O(n)$ probes, the node v on the border B with a minimum values is detected. If there exists a corner node v, we have found a local minimum and the algorithm terminates; else there is a unique neighbor of u of v that doesn't exist on B. If $v \prec u$, again we detect a local minimum v and the algorithm terminates, else G satisfies Property $(*)$, and we call the algorithm on G.