CS 161
Summer 2017

Problem Set 2
Due: July 14, 2017, 11:59 p.m.

---

1. **Asymptotics redux** (5 points) For the following pairs of functions $f(n)$ and $g(n)$, indicate whether: (i) $f(n) = O(g(n))$, (ii) $f(n) = \Omega(g(n))$, or (iii) $f(n) = \Theta(g(n))$.

   If you believe $f(n) = O(g(n))$, show constants c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. If you believe $f(n) = \Omega(g(n))$, show constants c and n_0 such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.

   [We are expecting selection of (i), (ii), or (iii) and a choice of c and n_0.]

   (a) (1 point) $f(n) = 2n^4 - 3n^2 + 7$, $g(n) = n^5$

   (b) (1 point) $f(n) = \frac{\log_2 n}{n}$, $g(n) = \frac{1}{n}$

   (c) (1 point) $f(n) = 2^n$, $g(n) = 2^{2n}$

   (d) (1 point) $f(n) = \binom{n}{2}$, $g(n) = 4^{\log_2 n}$

   (e) (1 point) $f(n) = 2^{\sqrt{\log_2 n}}$, $g(n) = (\log_2 n)^{100}$

   SOLUTION: There are multiple solutions for c and n_0 for each example. Check that their particular c and n_0 satisfies the asymptotic bound for either $O(\cdot)$ or $\Omega(\cdot)$.

   (a) $f(n) = O(g(n))$, $c = 2$, $n_0 = 7$

   (b) $f(n) = \Omega(g(n))$, $c = 1$, $n_0 = 4$

   (c) $f(n) = O(g(n))$, $c = 1$, $n_0 = 0$

   (d) $f(n) = \Theta(g(n))$, $c_O = 1$, $n_{O0} = 0$, $c_\Omega = \frac{1}{4}$, $n_{\Omega 0} = 2$

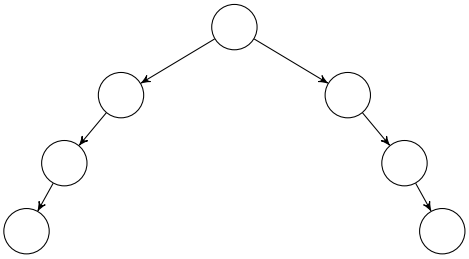
   (e) $f(n) = \Omega(g(n))$, $c = 1$, $n_0 = 2^{2^{24}}$

2. **Red, black, or neither?** (4 points) For each of the unlabeled binary trees, state whether or not it can be the structure of a red-black tree. If so, color the vertices red or black. If not, state which of the red-black tree invariants (1 to 5) cannot be satisfied and provide an explanation.

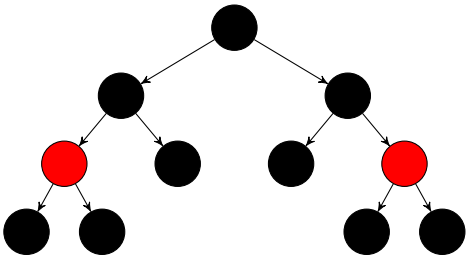
   [We are expecting a YES/NO, and either (1) a valid coloring or (2) a set of violated invariants and a brief explanation. For the colorings, feel free to redraw the trees by hand and upload the image.]

   (a) (1 point) No. We show this using proof by contradiction. Suppose that this is a valid red-black tree. The root node is, therefore, black. Consider the left child. If it is red, the only case in which invariant 5 is maintained is when all of its descendants are red. But, this violates invariant 4. So, the left child of the root node has to be black. In this case, its two descendants should be red to maintain invariant 5. But, this violates invariant 4. We have exhausted all possible ways to label the left child

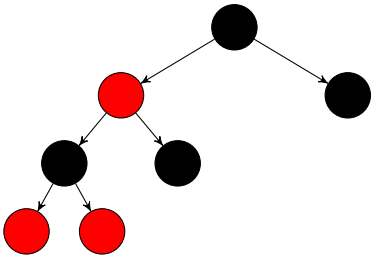
and none of them maintains a valid red-black tree. We have reached a contradiction and thus, this tree cannot be the structure of a red-black tree.



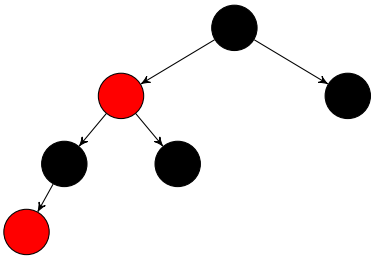
(b) (1 point) Yes.



(c) (1 point) Yes.



(d) (1 point) Yes.



3. **Finding k occurrences.** (7 points) Let L be an unsorted sequence of n numbers, where the numbers are not necessarily distinct. Assume you have an existing an $O(n \log n)$-time and $O(1)$-space sorting algorithm (i.e. heapsort) and the linear-time selection algorithm, which can may be used as subroutines. We want to find all elements in L that occur k or more times with using $O(1)$ extra space.

   (a) (1 point) Let $k = \lceil n/2 \rceil$; describe an $O(n)$ algorithm.

   [We are expecting a description of the algorithm (pseudocode optional), and a brief justification for its runtime.]

   (b) (1 point) Describe an $O(n \log n)$ algorithm that works for any k.

[We are expecting a description of the algorithm (pseudocode optional), and a brief justification for its runtime.]

(c) (3 points) Using divide-and-conquer, describe an algorithm with runtime $T(n,k) \leq O(n \log(n/k))$.

[We are expecting a description of the algorithm, pseudocode, and a brief justification for its runtime.]

(d) (2 points) Prove the lower bound $T(n,2) \geq \Omega(n \log n)$.

[We are expecting a brief, but convincing proof.]

SOLUTION:

(a) Let $x_1 \leq x_2 \leq ... \leq x_n$ be the numbers in L in the increasing order. We will refer to this sequence in our arguments, though we cannot compute it in linear time. Let us first consider the case where n is odd. If there is a subset of $k = \lceil n/2 \rceil$ equal elements, it contains the median, i.e., x_k. We can compute the median using select and compare it with the remaining elements; thus we find the number of occurrences of x_k in L and decide whether to report it or not. If n is even, we may do the same for the lower median x_k and the upper median x_{k+1} independently (though this is not optimal).

(b) We sort L and scan the sequence, counting occurrences of each number and reseting the counter when the current number changes.

(c) The pseudocode is as follows

```
find-repeating(L, k):
    n = size(L)
    if (n >= k):
        x = SELECT(L, ceil(n/2))
        (L1, L2, m) = divide-and-count(L, x)
        if (m >= k):
            print x
        report-repeating(L1, k)
        report-repeating(L2, k)

divide-and-count(L, x)
    m = # of occurrence of x in A
    # Swap in place to find the left and right subarray
    L1 = elements in A that are smaller than x
    L2 = elements in A that are larger than x
    return (L1, L2, m)
```

It is obvious that divide-and-count runs in linear time. The running time of `find-repeating` satisfy the recurrence

$$T(n,k) = \begin{cases} \Theta(1) & \text{if } n \leq k \\ 2T(n/2,k) + \Theta(n) & \text{if } n \geq k \end{cases}$$

Solving this recurrence, we get $T(n,k) \leq cn(log_2(n/k)+1)$ for $n \geq k$, where c is some constant.

(d) Any algorithm for the given problem obtains information about the input sequence by comparing its elements. Suppose that the elements in L are distinct. We claim that to check this condition, the algorithm must compare x_j with x_{j+1} for each j at some point during the computation. Indeed, if the algorithm fails to compare x_j with x_{j+1} for some j, it will not be able to tell the difference between L and the similar sequence L in which x_{j+1} is replaced with a copy of x_j. Thus, if we keep record of all comparison results, we will have enough information to sort the input sequence. But the sorting of an arbitrary permutation requires $\Omega(n \log n)$ comparisons.

4. **Word representations.** (5 points) As described in class, radix sort runs in $O(d(n+k))$ if the stable sort it uses runs in $O(n+k)$ time. For this problem, assume that radix sort uses bucket sort, which does in fact run in $O(n+k)$ time.

(a) (1 point) Pretend we want to use radix sort to sort a list of words W in lexicogrpahical (alphabetical) order. All words are padded with *space* characters (assume *space* comes before 'a' lexicographically) to make them the same length. For the following W, describe what the three variables d, n, and k refer to (your explanation should include some reference to W or its elements) and what their values would be for this problem. (For example, if the formula included a variable x that referred to the length of the first word in the list of items being sorted, you might say "x is the length of the first word in the list, and is equal to 3 for list W.")

W = [the, quick, brown, fox, jumps, over, the, lazy, dog]

[We are expecting values and descriptions for d, n, and k.]

(b) (1 point) Now suppose we convert each of the strings in W to their ASCII representations (8-bit binary strings, with *space* mapping to 00100000, so the word 'the' becomes 40 digits long–8 digits per character plus 8 digits per padding space to make it as long as the longest word in W). We still want to use radix sort, and we want to treat these bit strings as literal strings (i.e., do not try to interpret the 8 bit strings into decimal numbers). Now what are the values for d, n, and k?

[We are expecting values and descriptions for d, n, and k.]

(c) (1 point) Now we're back to using the character strings from part (a), but you happen to have the date (day, month, year) that each word was first published in an English dictionary. You want to sort first by date, then use lexicographical ordering to break ties. You will do this by converting each of the original words in W into words with date information (digits) prepended, appended, or inserted somewhere in the string. (Assume the digits 0-9 come before the *space* character and a-z). Write the string you would use to represent the word "jumps" (first published November 19, 1562) so that it will be correctly sorted by radix sort for the given objective.

[We are expecting a string.]

(d) (1 point) You decide that because you only ever use the words in a certain list V in everyday speech, you would like to save space and simply represent the first word in V with the binary value '0', the second word with '1', the third with '10', etc., continuing to increment by one in binary (and no longer including date information). All subsequent occurences of a particular word w receive the same binary assignment as the first occurence of w, all strings are padded with '0's to make them equal length. V has n words in it, where $n > 2$. Give the time complexity of radix sort on the list V with all words converted to their 0-padded binary strings and explain (informally) why that is correct. Simplify your answer as much as possible where values of d, n, or k are known.

[We are expecting a runtime, and a brief justification.]

(e) (1 point) Not wanting to mess with binary conversions, you decide instead to represent the words in your vocabulary V with "one-hot" vectors (vectors of length n with all 0's except for a single '1' in a position corresponding to a particular word. For example, in W, the word 'the' would be represented as '10000000', since there are eight unique words in the list). Give the new worst-case time complexity of radix sort on the list V, again simplifying as much as possible and explaining (informally) why that is the correct complexity.

[We are expecting a runtime, and a brief justification.]

SOLUTION:

(a) $d = 5$ (maximum word length in W), $n = 9$ (number of words in W), $k = 27$ (26 lowercase letters + space)

(b) $d = 40$ (8 bits * max word length of 5), $n = 9$ (number of words in W), $k = 2$ (all characters are either 0 or 1)

(c) 15621119jumps (year, month, day, word)

(d) $O(n \log n)$
radix sort complexity $= O(d(n + k))$, but $k = 2$ and $d \leq \log n + 1$.

$$O(d(n + k)) \leq O((\log n + 1)(n + 2)) = O(n \log n + n + 2 \log n + 2) = O(n \log n)$$

(e) $O(n^2)$.
Again, $k = 2$, but $d = O(n)$.
Linear sort isn't so linear with a bad data representation, eh?

5. **Centroid trees.** (7 points) We will now shift our attention to trees. Given a tree with N nodes, its centroid is defined by the node whose removal splits the tree into a forest of trees such that each of the resulting trees contains at most $N/2$ nodes.

(a) (2 points) Prove that, for any given tree, there exists a centroid.
[We are expecting a rigorous proof.]

(b) (2 points) Give an algorithm to find the centroid of a tree. Assume that you have an algorithm that can compute the size of a tree given its root in linear time (We

will actually learn about this later in the class. If you are curious, please check out the Depth-First-Search algorithm). Also, provide the runtime of this algorithm.

[We are expecting a description of the algorithm (pseudocode optional) and a brief justification for its runtime.]

(c) (2 points) Building upon the concept of a centroid, let us now explore centroid trees (not to be confused with the *centroid of a tree*. In a centroid tree, every node is the centroid of its own subtree (with this node as the root of the subtree). Give an algorithm to construct a centroid tree from a tree with N nodes. (Hint: use divide-and-conquer) with its runtime.

[We are expecting a description of the algorithm (pseudocode optional) and a brief justification for its runtime.]

(d) (1 point) What is the height (number of levels) of this centroid tree? Please support your claim with a brief explanation.

[We are expecting a height and a brief justification.]

SOLUTION:

(a) Choose an arbitrary node, v. If v satisfies the property of a centroid, return v. If not, there must exist a subtree with $> N/2$ nodes. There are a total of N node and since v is not the centroid, one of the subtrees must have violated the centroid property by having $> N/2$ nodes. Now, consider a node, u, adjacent to v in this subtree and follow the same procedure. At every iteration, we decrease the size of the centroid property-violating subtree by 1. Since there are finite number of nodes and we never revisit a node, the number of nodes in each tree will eventually become $\leq N/2$. Therefore, this algorithm should terminate and a centroid must exist.

(b) Pick an arbitrary root. Use the algorithm to find the size of each subtree. Move to node adjacent to root in the largest subtree. Repeat until no subtree has more than $N/2$ nodes. The root at that iteration is the centroid of the tree.
Proof of correctness: If root has the centroid property, we are done. If not, we move to the larger subtree because the remaining parts must have less than $N/2$ nodes. Since the size of the larger subtree decreases with each iteration and we have a finite number of nodes, it must terminate and produce a correct solution.
Time complexity is $O(N^2)$: We traverse $O(N)$ nodes and at each node, we run DFS which is also $O(N)$, making the overall runtime $O(N^2)$. Note that a faster solution is possible (but not required for full credit) if dynamic programming is used. In this case, DFS is only run at the root/in the beginning, making the runtime $O(N)$.

(c) **Note:** Some students proposed to disassemble the original tree and attach each node directly to the root node. This solution exploits a loophole in our problem formulation. We will accept this solution. If your solution was similar to this, we strongly encourage you to read the intended proof below for pedagogical reasons.

Centroid-Tree algorithm. Once we find the centroid of the given tree, we remove it, decomposing the tree into a number of subtrees where each subtree has fewer

than $N/2$ nodes. This centroid is the root of the centroid tree. We now recursively decompose each of the subtrees and attach their centroids as children of the root of our centroid tree. When the recursion terminates, we will have a centroid tree. Runtime is $O(N^2 \log N)$. If dynamic programming is used (again, not required for full credit), overall runtime is $O(N \log N)$.

(d) The centroid tree has at most $O(\log_2 N)$ levels. Recall that at each iteration, each subtree (formed as a result of removing the centroid) has at most $N/2$ nodes (where N is the number of nodes in the current tree).

6. **20-questions?** (8 points) Suppose you want to sort an array A of n numbers (not necessarily distinct), and you are guaranteed that all the numbers in the array are in the set $\{1, \ldots, k\}$. A **"20-question sorting algorithm"** is any deterministic algorithm that asks a series of YES/NO questions (not necessarily 20 of them, that's just a name) about A, and then *writes down* the elements of A in sorted order. (Specifically, the algorithm does not need to rearrange the elements of A, it can just write down the sorted numbers in a separate location).

   Note that there are many YES/NO questions beyond just comparison-questions—for example, the following are also valid YES/NO questions: "If I ignored A[3] and A[17] would the array be sorted?" and "Did it rain today?"

   (a) (3 points) Describe a 20-question sorting algorithm that, for every input, asks only $O(k \log n)$ questions. For now, assume that the algorithm accepts n and k as input.

   [We are expecting a description of the algorithm and a brief justification for its runtime.]

   (b) (1 point) Now suppose the algorithm does not accept n or k as input. Describe an algorithm that determines these values and still asks only $O(k \log n)$ questions.

   [We are expecting a description of the algorithm and a brief justification for its runtime.]

   (c) (4 points) Prove that for *every* 20-question sorting algorithm, there exists some array A consisting of n integers between 1 and k that will require $\Omega(k \log \frac{n}{k})$ questions, provided $k \leq n$.

   [We are expecting a mathematically rigorous proof (which does NOT necessarily mean something long and tedious).]

SOLUTION:

[Hints: Why is it sufficient for this problem to lower-bound the number of ordered arrays, instead of counting exactly? Once you have understood this, use a counting argument: how can you lower-bound the number of ordered arrays are there that consist of n integers $\{1, \ldots, k\}$ (not necessarily distinct)? There are a number of ways to do this; we suggest you do NOT use Stirling's approximation: you don't need this in order to prove the result, and it will be complicated.]

(a) To output the sorted array, we need to know how many occurrences there are of each value $1, 2, \ldots, k$. So for each of these we do a binary search on the number of

occurrences: for example, we ask "Are there $n/2$ or more 1's in the array?" and so on until we determine the number of occurrences for each of $1, 2, \ldots, k$. Each binary search takes $O(\log n)$ time and we do k searches, so in total this takes $O(k \log n)$ time.

Formally, the pseudocode is as follows.

```
def bsearch(value, low, high):
  if low >= high:
    return low
  mid = (low + high) / 2
  answer = ask("Are there more than ‘mid’ occurences of ‘value’ in the array?")
  if answer is "yes":
    return bsearch(value, mid + 1, high)
  else:
    return bsearch(value, low, mid)

def main():
  counts = []
  for i = 1 to k:
    counts.append(bsearch(i, 0, n))
  for i, count in enumerate(counts):
    for j = 1 to count:
      print i
```

(b) The above assumes knowledge of n and k. If you did not know what these values were, you could also do a binary search for them. For example, you could ask "Is $n < c$?" for $c = 2, 2^2, 2^3, \ldots$ If the answer is first "YES" for $c = 2^b$, then we binary search for n between 2^{b-1} and 2^b. This will take in total $O(\log n)$ time: $O(\log n)$ to find the range we need to binary search, and $O(\log n)$ to do the binary search. This same procedure can be used to find k.

(c) An algorithm that asks at most c questions can have at most 2^c outputs because each sequence of answers can correspond to only one output. For any series of c yes/no questions, there are 2^c sequences of possible answers. So if there are N distinct sorted arrays (for given values of n and k) then any algorithm must ask at least $\log N$ questions in the worst case.

A lower bound on N, the number of distinct sorted arrays of length n containing the elements $\{1, \ldots, k\}$, is given as follows: for each value $1, \ldots, k-1$ let it occur anywhere between 1 and n/k times. Let the value k occur the number of times that will fill out the array to n numbers. This results in $\left(\frac{n}{k}\right)^{k-1}$ distinct sorted arrays. Thus we have a lower-bound on the worst-case number of questions: $\log \left(\frac{n}{k}\right)^{k-1} = (k-1) \log \frac{n}{k} = \Omega(k \log \frac{n}{k})$.

Alternative way to count N ("stars and bars" approach): the number of distinct sorted arrays is given by the number of ways to distribute n balls into k boxes, which is $\binom{n+k-1}{k-1} = \frac{(n+k-1)(n+k-2)\cdots(n+1)}{(k-1)!} \geq \frac{n^{k-1}}{k^{k-1}} = \left(\frac{n}{k}\right)^{k-1}$. This simplification follows from the fact that $(n+k-1)(n+k-2)\cdots(n+1) > n^{k-1}$ and $(k-1)! < k^{k-1}$.

7. **How is the course so far?** (0.5 points extra credit)

   Please complete the poll at `https://goo.gl/forms/LnGR9BvTQj8MuRyH2`.