COSC 302: Analysis of Algorithms — Spring 2018
Prof. Darren Strash
Colgate University

Problem Set 4 — Divide and Conquer II, Average-Case Analysis, Heaps, Non-Comparison Sorting

Due by 4:30pm Friday, Feb. 23, 2018 as a single pdf via Moodle (either generated via LaTeX, or concatenated photos of your work). Late assignments are not accepted.

This is an *individual* assignment: collaboration (such as discussing problems and brainstorming ideas for solving them) on this assignment is highly encouraged, but the work you submit must be your own. Give information only as a tutor would: ask questions so that your classmate is able to figure out the answer for themselves. It is unacceptable to share any artifacts, such as code and/or write-ups for this assignment. If you work with someone in close collaboration, you must mention your collaborator on your assignment.

Suggested practice problems (not to be turned in): 4.3-1, 4.3-8, 4.4-2, 4.4-4, 4.4-6, 4-3

1. *Hobbies.* Suppose you are given a group of n people each of which have exactly one hobby (though their specific hobbies are unknown to you). Suppose that you can find out if any two people have the same hobby by asking them to compare hobbies and then they tell you *yes* or *no*—in the process you do not learn what their hobbies are, only that they are the same or different. Now, you are tasked with determining if more than $n/2$ people in the group have the same hobby.

   Describe an algorithm that determines if more than $n/2$ people have the same hobby by asking $O(n \lg n)$ pairs of people to compare hobbies.

   Hints: Consider the following when designing your algorithm.

   (a) The combine step here is critical: the recursion fairy must return more than just "yes" or "no" as the solution to a subproblem.

   (b) Consider all 4 combinations of "yes"/"no" from the recursion fairy.

      i. Is a "yes" answer possible when both subproblems are "no"?

      ii. How many subproblems must return "yes" for the total problem to be "yes"?

      iii. Do you need to compare additional hobbies in case of a "yes" answer? How many comparisons can you do and still have $O(n \lg n)$ 'running time'? (i.e., what should your recurrence be?)

Solution on next page →

Solution: Assume that n is a power of 2.

Divide: Divide people into 2 subgroups of size $n/2$.

Conquer: Each subgroup that has a majority with the same hobby (a subset of people with size $> n/4$, we get a representative person who has that hobby, otherwise, we get that there is no majority subset. Note that there can only be one such majority subset from each subgroup, otherwise the subgroup would have $> n/4 + n/4 = n/2$ people. Note that for the base case, we have subsets of size 1, which always have a majority.

Combine:

Case 1: If neither subgroup has a representative, then there is no majority in the current group of n people. Otherwise, 2 groups of size $n/4$ or less cannot sum to a group of size greater than $n/2$.

Case 2: Some subgroup has a majority (one or both). Then we pick the representative person, and compare hobbies with all n people in the group. This takes $n-1$ comparisons. If the other subgroup has a majority, we perform the same $n-1$ comparisons with this representative. If either representative matches with $n/2$ other people, then the current group has a majority, otherwise, we don't.

Time Analysis: We perform at most $2n-2$ comparisons in our combine step, and recursively solve two subproblems of size $n/2$. Therefore our running time $T(n)$ is represented by the recurrence

$$T(n) = 2T(n/2) + O(n)$$

and $T(n) = O(n \lg n)$.

2. Suppose we are given a set S of n items, each with a value and a weight. For any element $x \in S$, we define two subsets

   - $S_{<x}$ is the set of all elements of S whose value is smaller than the value of x.
   - $S_{>x}$ is the set of all elements of S whose value is larger than the value of x.

   For any subset $R \subseteq S$, let $w(R)$ denote the sum of the weights of elements in R. The weighted median of S is any element x such that $w(S_{<x}) \leq w(S)/2$ and $w(S_{>x}) \leq w(S)/2$. Describe and analyze an algorithm to compute the weighted median of a given weighted set in $O(n)$ time. Your input consists of two unsorted arrays $S[1..n]$ and $W[1..n]$, where for each index i, the i-th element has value $S[i]$ and weight $W[i]$. You may assume that all values are distinct and all weights are positive.[1]

   Solution: We design a divide and conquer algorithm that uses the linear-time selection algorithm as a subroutine. Linear-time selection will be used to pick the median, and then binary search is performed using the weights of those values less than or greater than the median.

   Preprocessing: Compute $w(S)/2$. Make a copy of S, called S_{copy} on which we can run selection, so that we do not move elements of our original S (becomes clear in the divide step). These computations take $\Theta(n)$ time.

   Divide: Compute the median of S, by calling linear time selection with $i = n/2$, getting the median x. We run selection on S_{copy} so that partition does not move elements of S around. We then partition S, while at the same time swapping elements in W, so that $S[k]$ and $W[k]$ represent the value and weight of the same item. The divide step takes $\Theta(n)$ for all copying, median finding, and partitioning.

   Combined, we will perform $\Theta(n)$ operations making copies and partitioning the array, then look at one subproblem of size $n/2$. Therefore our running time $T(n)$ is represented by the recurrence
   $$T(n) = T(n/2) + \Theta(n)$$
   and $T(n) = \Theta(n)$ by the master theorem.

   We compute the weights $w(S_{<x})$ and $w(S_{>x})$ in $\Theta(n)$ time by summing weights that have corresponding values less than x and greater than x, where x is the median element.

   If $w(S_{<x}) \leq w(S)/2$ and $w(S_{>x}) \leq w(S)/2$, then we return x as the weighted median. Otherwise, one of $w(S_{<x})$ or $w(S_{>x})$ is greater than $w(S)/2$.

Continued on next page →

[1]From Jeff Erickson's *Algorithms and Models of Computation* (`http://www.cs.illinois.edu/~jeffe/teaching/algorithms`); Chapter 1: Recursion.

Conquer: We recursively call our algorithm on $S_{<x}$ or $S_{>x}$, whichever set has weight more than $w(S)/2$. These are stored in $S[1..n/2-1]$ and $S[n/2+1..n]$, respectively, with matching weights in $W[1..n/2-1]$ and $W[n/2+1..n]$. They are likewise stored in the same indices of S_{copy}, since linear time selection partitioned around the pivot x. However, note that the weighted median of the subproblems is not necessarily the weighted median of the full problem at this point—this is because we have eliminated weights which are used to compute the full weighted median. We therefore must also pass two weights along with the subset of elements, in order to capture the weight of the full problem: we pass the weight of all elements to the left of the set, call it w_l, and the weight of elements to the right of the set, call this w_r. At the top level, if we recurse on $S[1..n/2-1]$ then we pass $l = 0$ as the weight to the left of the set, and $w_r = w(S_{>x} \cup \{x\})$ as the weight to the right of the set. Likewise, if we recurse on $S[n/2+1..n]$, then we pass $w_l = w(S_{<x} \cup \{x\})$ as the weight left of the set, and $w_r = 0$. These weights are updates throughout recursion. For ease of description, we always create new copies of S. See pseudocode below.

Combine: Nothing to do, like binary search.

Pseudocode:

Algorithm 1 Compute the weighted median.

```
proc Weighted-Median($S$, $W$, $w_l$, $w_r$, $w_{\text{total}}$)
 1: $S_{\text{copy}} \leftarrow S$
 2: $x \leftarrow$ Select($S_{\text{copy}}$, $n/2$)
 3: Partition($S$,$W$)
 4: $w(S_{<x}) \leftarrow$ sum($W[1..n/2-1]$)
 5: $w(S_{>x}) \leftarrow$ sum($W[n/2+1..n]$)
 6: if $w_l + w(S_{<x}) \leq w_{\text{total}}/2$ and $w_r + w(S_{>x}) \leq w_{\text{total}}/2$ then
 7:     return x
 8: else if $w_l + w(S_{<x}) > w_{\text{total}}/2$ then
 9:     $S \leftarrow$ copy($S[1..n/2-1]$)
10:     $W \leftarrow$ copy($W[1..n/2-1]$)
11:     $w_r \leftarrow w_r + w(S_{>x}) + W[n/2]$
12: else
13:     $S \leftarrow$ copy($S[n/2+1..n]$)
14:     $W \leftarrow$ copy($W[n/2+1..n]$)
15:     $w_l \leftarrow w_l + w(S_{<x}) + W[n/2]$
16: return Weighted-Median($S$, $W$, $w_l$, $w_r$, $w_{\text{total}}$)

    // To solve the full problem, we call Weighted-Median with $w_l = 0$, $w_r = 0$, $w_{\text{total}} = w(S)$
17: Weighted-Median($S$, $W$, 0, 0, $w(S)$)
```

3. Suppose we are given an array $A[1..n]$ of distinct integers, and we wish to use linear search to find an element with maximum value. Prove that the expected number of comparisons made by linear search until the maximum element is *encountered*[2] is approximately[3] $n/2$. Use the following template for guidance.

   (a) We will assume that the input consists of distinct integers, and that all permutations of these integers are equally likely. What is the probability that a particular element $A[i]$ is the maximum element in A?

   Solution: Since all permutations are equally likely, the probability that any particular value $A[i]$ is maximum is $1/n$. We can see this by dividing the number of permutations of A where $A[i]$ is maximum by the total number of permutations of A:

$$Pr\{A[i] \text{ is maximum}\} = \frac{(n-1)!}{n!} = \frac{1}{n}.$$

   (b) We need an indicator random variable. Recall that indicator random variables are used to *count* events, being 1 when you encounter an event, and 0 when you don't. What do we want to count here? What does a summation look like for this problem? Give a summation as a template, and temporarily use this summation to guide you in the next steps.

   Hint: You don't **need** a double summation for this problem. However, you **can** solve it with a double summation.

   Solution: We want to count the number of comparisons made until the maximum element is found. This is the same as the index of the maximum element in A. Linear search advances one index at a time from 1 to n, so the summation could have the form:

$$X = \sum_{i=1}^{n} X_i,$$

   where X_i is the indicator random variable.

   (c) Devise an indicator random variable. Your indicator random variable should be 1 when a given element meets your criteria, and 0 otherwise.

   Solution: Given the probability from (a), the event is probably that a particular element is maximum. That is,

$$X_i = \begin{cases} 1 & A[i] \text{ is maximum,} \\ 0 & \text{otherwise.} \end{cases}$$

Continued on next page →

[2] Note that this is different that finding the maximum element. As we may encounter the maximum element, but not be assured it is truly maximum until we have evaluated all elements in the array.

[3] Note that I am not giving you the exact number, just something close (intuitively). Your number should be exact, and not asymptotic.

(d) Is it sufficient to simply to have a summation that includes your indicator random variable and no other multiplicative factors? Why or why not? If you need some additional multiplicative factor, what is it?

Solution: No, the guess from (b) always sums to 1, since only one of the X_i's can be 1 (there can only be one maximum, elements are distinct!). My multiplicative factor is i, since this selects the index of the maximum element. Now the sum is

$$X = \sum_{i=1}^{n} iX_i.$$

(e) Now compute the expected value of your summation by combining together your indicator random variable, your probability, and any multiplicative factor you might have.

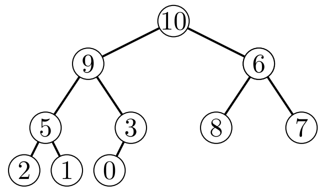
$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} iX_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[iX_i] && \text{linearity of expectation} \\
&= \sum_{i=1}^{n} i\mathrm{E}[X_i] && \text{linearity of expectation} \\
&= \sum_{i=1}^{n} i \Pr\{A[i] \text{ is maximum}\} \\
&= \frac{1}{n}\sum_{i=1}^{n} i \\
&= \frac{1}{n} \cdot \frac{n(n+1)}{2} \\
&= \frac{n+1}{2}.
\end{aligned}$$

The expected number of comparisons until the maximum element is encountered is $\frac{n+1}{2}$.

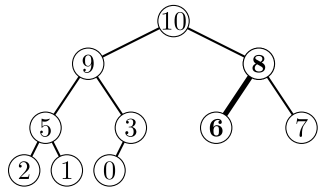
4. In this problem, you will execute a sequence of operations on a given array, which is not yet a heap. The operations are cumulative: each operation is executed on the result of the previous operation. For each operation, draw the binary tree representation of the output. The first operation is applied to the following array:

$$A = [10, 9, 6, 5, 3, 8, 7, 2, 1, 0].$$

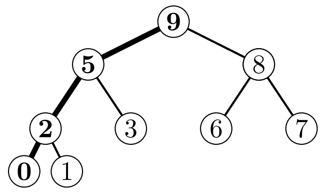
(a) First, draw the contents of the array A in the binary tree representation.



(b) MAX-HEAPIFY$(A, 3)$

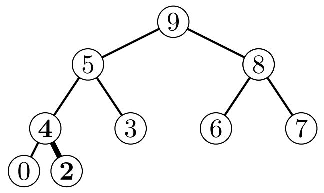


(c) HEAP-EXTRACT-MAX(A)



Key 10 is extracted (removed) from the heap and and key 0 is moved to the root $A[1]$. Key 0 is then swapped with its largest child until the max-heap property is restored.

(d) HEAP-INCREASE-KEY$(A, 9, 4)$



Increase the key of $A[9]$ to 4 and iteratively swap the new key with its parent until the max-heap property is restored.

5. (Problem 8.2-4 from CLRS) Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a..b]$ in $O(1)$ time. Your algorithm should use $\Theta(n+k)$ preprocessing time.

(Solution to be given with homework 5)