# HW1 - S15
# Warmuth

Justin Wong (jujwong)
4/14/15

1. Binary Min Heaps

   a) $O(n \log k)$

   Build a max-heap of the first $k$ elements this takes time O($k$). Then each element after the $k^th$ elemenet compare it to the root of the max-heap, if it is less than the root than make that the new root and heapify otherwise ignore the new element. The root will be the smallest $k^th$ element.

   The runtime for this algorithm ends up being O($nlogk$) because in order to find the $k^{th}$ smallest element it will have to first build the heap taking O($k$) time. Searching through the rest of the elements require O($nlogk$) time, due to the number of elements and the time it takes to compare them and possibly reorder the data structure to maintain the properties of a max-heap. O($k + nlogk$) reduces down to O($nlogk$) given that $k > 1$.

   b) $O(n + k \log n)$

   Build a min-heap using the $n$ elements. Then remove the root from the heap $k-1$ times, each time you remove the root the min-heap must reorder itself to ensure it maintains min-heap properties. Then just return the root, which is now the $kth$ smallest element.

   The runtime is $O(n + k \log n)$ due to the time to build the heap O($n$) and the number of times $k - 1$ that the root is removed taking O($logn$).

   c) $O(n + k \log k)$

   Build a min-heap of size $n$. Create a second initially empty min-heap of size $k$, the root of this heap will be the root from the original min-heap. Then we keep adding to the new min-heap by deleting the current root and inserting children of the current root from the original heap, this happens $k$ times. The second heap will be at capacity of $k$ nodes and the root will be the $k^th$ smallest element.

   The runtime is $O(n + k \log k)$. The time it takes to build a heap O($n$). The number of times an element is inserted $k$ and the time it takes to insert at $logk$. Getting the smallest element, the children, is a function that runs at constant time O(1).

2. Two-Thirds Sorting Algorithm

   This algorithm begins by taking the first 2/3s of the array, and sorting that. This is the preliminary run, where it ensures that each element in the first third is less than or equal to the elements in the second third. The next step is to take the now sorted 2nd third and the unsorted final third and sort those. Now all the largest elements must be in the 3rd third of the array, because if they were in the first sort then they would have been moved into the 2nd third and now on the second sort they would be moved into the 3rd third. The final step is to sort the the 1st 2/3s again as the middle (2nd third) elements could have changed. This final step allows elements to gain their proper order, and places the now ensured smaller elements into the 1st third. Even if the array was reverse sorted (max to min) this algorithm would, in step 1 take all the highest elements move them to the second 3rd, in step 2 move those to the third 3rd and move the smallest to the second 3rd, then finally replace the lowest elements in the second 3rd to the first 3rd.

The problem creates 3 sub-problems all of size $\frac{2}{3}$n The recurrence relation is:

$$T(n) = 3T(\frac{2}{3}n) + 1$$

.

$$= 1 + 3 + 9T(4/9)n$$

$$...$$

$$= 1 + 3 + 3^2 + ... + 3^{log_{3/2}n}$$

$$= \frac{3^{log_{3/2}n+1} - 1}{3 - 1}$$

$$= \theta(3^{log_{3/2}n})$$

$$= \theta(n^2.71)$$

3. KT, problem 1, pg 246

   We have our two arrays $m_1$ and $m_2$. We begin with a query to both to find their respective medians element where k = $\frac{n}{2}$. The median must be between the two arrays. There are at least $n$ values in both $m_1$ and $m_2$ which are less than or equal to the max of either $m_1$ and $m_2$. The median, by definition, is not able to be larger than the max of $m_1$ and $m_2$, nor can it be smaller than the min of $m_1$ and $m_2$. This allows us to know that we don't have to look through the total amount of either database $m_1$ or $m_2$. This algorithm will continue recursively until there are two options left, the $n^{th}$ and the $n^{th} + 1$, and then we would return the smaller of the two.

   The problem splits in half each time through the use of two queries The recurrence relation is:

$$T(n) = T(\frac{n}{2}) + 2$$

$$...$$

$$T(n) = O(logn)$$

4. Running Time Comparisons

   (a) Algorithm $A$ solves problems by dividing them into 5 subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.

$$T(n) = 5T(\frac{n}{2}) + O(n)$$

   $log_2 5 \approx 2.321$
   Case 1 of the Master Theorem: $f(n) = n = O(n^{2-\epsilon})$ when $\epsilon = 1.321$.
   Thus, $T(n) = \theta(n^{log_2 5}) \approx \theta(n^{2.321})$

(b) Algorithm $B$ solves problems of size n by recursively solving 2 subproblems of size $n-1$ and the combining the solutions in constant time.

$$T(n) = 2T(n-1) + C$$

We must expand the reccurence and recursively expland $T(i)$ until $i = 0$ , since we are unable to perform Master Theorem:

$$T(n) = 1 + 2 + 2^2 + 2^3T(n-4) + 2^4T(n-5)...$$
$$= 1 + 2 + 2^2 + 2^3 + ... + 2^{n-1}T(0)$$

Thus the running time is O($2^n$).

(c) Algorithm $C$ solves problems of size $n$ by dividing them into nine sub-problems of size $n/3$, recursively solving each sub-problem, and the combining the solution in $O(n^2)$ time.

$$T(n) = 9T(\frac{n}{3}) + O(n^2)$$
$$log_3 9 = 2$$

Case 2 of the Master Theorem: $f(n) = \theta(n^{log_b a})$
$O(n^2) = \theta(n^{log_3 9})$
Thus the running time is $\theta(n^2 logn)$

The fastest algorithm is algorithm C.

5. Hadamard Matrices

The column vector x of length n, let $l_1$ be the length of the first n/2 and $l_2$ to be the rest of the elements also length n/2. To find:

$(H_k v)_1 = H_{k-1}v_1 + H_{k-1}v_2 = H + k - 1(v_1 + v_2)$

and

$(H_k v)_2 = H_{k-1}v_1 - H_{k-1}v_2 = H + k - 1(v_1 - v_2)$

To get $H_k v$ we only need to find

$v_1$ + $v_2$ and $v_1$ - $v_2$ and compute $H_{k-1}(v_1 + v_2)$ and $H_{k-1}(v_1 - v_2)$ recursively.

So we really only need to compute: $H_{k-1}(v_1$ and $H_{k-1}(v_2$ then we can combine these using addition and subtraction, reducing the amount of repetive multiplication that is required, and doing addition and subtraction which requires less computation.

This recurrence is represented by 2 subproblems each with (1/2)n, with addition/subtraction being used as the combining in O(n).

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

This is the classic divide and conquer recurrence, and solved using Master Theorem. The runtime is O(nlogn).

$$log_2 2 = 1$$
$$f(n) = \theta(n^{log_b a})$$
$$O(n) = \theta(n^{log_2 2}) = \theta(n)$$