Sheetal Parikh
Foundations of Algorithms
EN. 605.621.81.SU19
Homework #1

Foundations of Algorithms Homework #1

1. *[20 points] Analyze the following algorithms for performing searches.*

(a) [10 points] Describe the complexity of the linear search algorithm.

In linear search we search for an element by traversing an array until the element is found. Since you must go through each element of the array, the time complexity depends on the length of the array or n in this example. The more elements in the array, the higher the *n-value.* Since the upper bound would be described as *n*, the time complexity or worst-case scenario is $O(n)$. The best case scenario would be $O(1)$ in which the value we are searching for is the first element in the array.

(b) [10 points] Describe the time complexity of the binary search algorithm in terms of number of comparisons used (ignore the time required to compute m = b(i + j)/2c in each iteration of the loop in the algorithm) as well as showing the worst case complexity. Note: Do not use the algorithm on page 799 of the book, use the following algorithm.

In the given binary search algorithm, we approximately check the middle element in the array. If the middle element is smaller than the target, then the target is in the first half of the array. If the target is greater than the middle element, then the target is in the 2nd half of the array. Overall, one comparison divides the number of elements to search in the array by half. The best case would occur if there is only one comparison which would only occur if the target was the middle element. Therefore, the best case scenario would have a time complexity of $O(1)$. The worst case would occur if the target element is not in array and we keep having to divide the elements in the array by half until there is only 1 element left. With each division, you add 1 comparison and divide the list you are searching by half which results in a worst case time complexity of $O(log_2 N)$. For example, let say that we have an array that has a length of $N = 32$:

# of Elements	# of Comparisons
32	0
16	1
8	2
4	3

2 4
1 5

As shown above, the pattern of the number of elements(n) and the number of comparisons (x) can be described by $X = log_2 N$.

2. *[10 points] Give asymptotic upper and lower bounds for* $T(n) = 3T(n/2 + nlogn)$ *assuming* $T(n)$ *is constant for sufficiently small* n *. Make your bounds as tight as possible. Prove that your bounds are correct*.

The master method can be used to find the bounds for $T(n)$.

Step 1: $a = 3$

$b = 2$

$f(n) = nlogn$

Step 2: $n^{log_b a}$

$= n^{log_2 3}$

$= n^{1.5850}$ $2^x = 3$

$= n^{log_2 3}$ $x \approx 1.5850$

$log_2 3 > 1$

Step 3: comparing $n^{log_b a}$ $and f(n)$

$f(n) = nlogn$

$n^{log_b a} = n^{1.5850}$

$nlogn < n^{1.5850}$

$f(n) < n^{log_b a}$

Step 4: since $f(n) < n$, $T(n)$ is Case 1

Therefore, $f(n) = n^{1.5850 - \varepsilon}$

$T(n) = \theta(n^{log_2 3})$

3. *[15 points] Give asymptotic upper and lower bounds for* $T(n) = T(\sqrt{n}) + 1$ *, assuming* $T(n)$ *is constant for sufficiently small* n *. Make your bounds as tight as possible. Prove that your bounds are correct.*

The substitution method can be be used to change the format of T(n) so that it's easier to use the master method.

$m = log_2 n$ $T(n) = T(\sqrt{n}) + 1$

$n = 2^m$ $T(n) = T(n^{1/2}) + 1$

$S = T(2^m)$ $T(2^m) = T(2^m) + 1$

$$S(m) = S(m/2) + 1$$

The master method can be used to evaluate $S(m) = S(m/2) + 1$

Step 1: $a = 1$

$b = 2$

$f(m) = 1$

Step 2: $m^{log_b a}$

$= m^{log_2 1}$

$= m^0$

$= 1$

Step 3: comparing $m^{log_b a}$ $and f(m)$

$f(m) = m^{log_b a} = 1$

Step 4: since $f(m) = m$, $S(m)$ is evenly distributed and thus Case 2

$S(m) = \theta(m^{log_b a} logm)$

$= \theta(m^0 logm)$

$= \theta(logm)$ $\qquad m = logn$

After substituting m into the equation above:

$T(n) = \theta(loglogn)$

4. *[20 points] Give asymptotic upper and lower bounds for T(n) = 2T(n/3 + 1)+ n, assuming T(n) is constant for sufficiently small n. Make your bounds as tight as possible. Prove that your bounds are correct.*

We can assume that *T(n) = 2T(n/3 + 1)+ n* will have the same time complexity as *T(n) = 2T(n/3)+ n.* The "+ 1" in the equation shouldn't affect the solution to the recurrence because when n is sufficiently large, the difference between *n/3 + 1 and n/3* is negligible.

The master method can be used to find the time complexity of *T(n) = 2T(n/3)+ n*:

Step 1: $a = 2$

$b = 3$

$f(n) = n$

Step 2: $n^{log_b a}$

$= n^{log_3 2}$

$= n^{0.6309}$ $\qquad 3^x = 2$

< 1 $\qquad x \approx 0.6309$

Step 3: comparing $n^{log_b a}$ $and f(n)$

$$f(n) = n$$
$$n^{log_b a} = n^{0.6309}$$
$$n > n^{0.6309}$$
$$f(n) > n^{log_b a}$$

Step 4: since $f(n) > n^{log_b a}$, $T(n)$ is Case 3

Checking the regularity expression:

$af(n/b) \le cf(n)$ where $c < 1$

$f(n/b) = (n/b)log(n/b)$ $\quad a(n/b)log(n/b) \le cn$

$2(n/3)log(n/3) \le cn$

$(2/3)nlog(n/3) \le cn$ condition satisfied for $c = 2/3$

$$f(n) = \Omega(n^{log_3 2 + \varepsilon})$$
$$T(n) = \theta f(n)$$
$$= \theta(n)$$

Therefore, we can assume that *T(n) = 2T(n/3 + 1)+ n* will also run in $\theta(n)$. To prove that *T(n) = 2T(n/3 + 1)+ n* will also run at $\theta(n)$ we should prove that the recurrence runs in $O(n)$ and $\Omega(n)$.

To prove that *(n/3 + 1) is* $\Omega(n)$ we need to find values for n_o and c such that *c(g(n))* ≤ *f(n)* in which *g(n)* is the lower bounds on *f(n).*

Let's say n_o = 3 and c = ⅔, $n \ge n_o$

$n/3 + 1 \ge n/3 + n/3$

$n/3 + 1 \ge 2n/3 \rightarrow c(n) \rightarrow O(n)$

$f(n) \ge cg(n)$

This is true so we can say that the lower bounds on f(n) is $\Omega(n)$

To prove that *(n/3 + 1) is* $O(n)$ we need to find values for n_o and c such that *c(g(n))* ≥ *f(n)* in which *g(n)* is the upper bounds on *f(n).*

Let's say n_o = 30 and c = 11/30, $n \ge n_o$

$n/3 + 1 \le n/3 + n/30$

$n/3 + 1 \le 10n/30 + n/30$

$$n/3 + 1 \leq 31n/30 \rightarrow c(n) \rightarrow O(n)$$

$$f(n) \leq cg(n)$$

This is true so we can say that the upper bounds on f(n) is $O(n)$.

Because the upper bounds is $O(n)$ *and the* lower bounds is $\Omega(n)$ *we can say that the recurrence runs at* $\theta(n)$. It was correct to assume that the "+1" in the equation wouldn't affect the solution to the recurrence.

5. *[35 points] Collaborative Problem –CLRS 2-1: Although merge sort runs in Θ(n lg n) worst-case time and insertion sort runs in Θ(n 2) worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to coarsen the leaves of the recursion by using insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which n/k sublists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.*

(a) [10 points] Prove that insertion sort can sort the n/k sublists, each of length k, in Θ(nk) worst-case time.

We know that the worse case for insertion sort can be described as $an^2 + bn + c$ *or specifically for this example using length k,* $ak^2 + bk + c$. Since we have n/k sublists we can apply or multiple the number of sublists to $ak^2 + bk + c$:

$$= (n/k)(ak^2 + bk + c)$$
$$= ank + bn + cn/k$$

By focusing on just the higher ordered term ank *and ignoring the coefficient we see that insertion sort can sort the n/k sublists each of length k in* $\theta(nk)$ worst-case time.

b) [10 points] Prove how to merge the sublists in Θ(n lg (n/k)) worst-case time.

We know that each merge occurs at $\theta(n)$ when merging n elements. Therefore, if you merge 2 sublists into one list it should also occur at $\theta(n)$ worst-case time. We need to next determine the number of merges. The number of merges would appear as the following: n/k ➡ $n/2k$ ➡ $n/4k$ ➡.........1 (in which each step occurs at $\theta(n)$). The number of merges would occur at $log(n/k)$, worst-case time. By applying the worst-case time of each merge, $\theta(n)$, to the number of steps, $log(n/k)$, we get final sublist merge worst-case time of $\theta(nlog(n/k))$.

c) [10 points] Given that the modified algorithm runs in Θ(nk + n lg (n/k)) worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge sort, in terms of Θ-notation?

We know that standard merge sort in terms of θ - notation occurs at $\theta(nlogn)$. Therefore, k cannot be faster than $logn$ because than the modified algorithm would run faster than standard merge sort. Therefore, we can assume that the largest possible value of k can be $logn$.

assuming $k = logn$ $\quad \theta(nk + nlog(n/k))$

$$= \theta(nlogn + nlogn/logn)$$
$$= \theta(nlogn + n)$$

By focusing on the higher ordered terms we get $\theta(nlogn)$.

(d) [5 points] How should we choose k in practice?

Since insertion sort has a “worse” worst-case time of $\theta(n^2)$ compared to that of merge sort, $\theta(nlogn)$, we would chose a value of k that optimizes insertion sort used in the modified algorithm. We are also using merge sort so we wouldn’t want to pick a value that is too small that would cause merge sort to become inefficient. Insertion sort is more efficient with smaller lists whereas merge sort is more efficient with larger lists. Therefore, we would chose the largest value of k in which insertion sort would have a better time complexity than merge sort.