Homework 1

Ting He
Foundations of Algorithms, Spring 2022

February 9, 2022

Proposition 1.

Problem Statement: Describe the time complexity of linear search algorithm. Choose the tightest asymptotic representation and argue why that is the tightest bound
Assumptions: $T(n)$ is the total time we needed for this linear search
Computations: Θ is the most tightest asymptotic representation since from theorem 3.1 we have, for any 2 functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \omega(g(n))$. The total tightest asymptotic running time can be written as:

$$\begin{aligned} T(n) &= \Theta(1) + \sum_{t=1}^{n-1} \Theta(1) + \Theta(1) \\ &= \Theta(n^2) \end{aligned}$$

where the $\sum_{t=1}^{n-1} \Theta(1)$ comes from the step 4 and 5 in our algorithms, the while loop and its increment of i; while the other steps in our algorithms are all constant running time as $\Theta(1)$
Conclusions: $\Theta(n^2)$

Proposition 2.

Problem Statement:Analyze the following binary search algorithm. Assume the input A is a sorted list of elements; x may or may not be in A.
(a) describe the time complexity of binary search and choose the tightest asymptotic representation. (b) make one small change to the algorithm to improve its run time, and give the revised tightest asymptotic representation
Assumptions: $T(n)$ is the total time we needed for this binary search
Computations:
(a) We can treat this binary search as a divide-and-conquer problem. We denoted that $n = len(A)$ and we are looking for place for x to insert. The problem itself can divided into

2 sub-problem with $\Theta(1)$ running time to search whether $x < A[m]$ or $x > A[m]$.

$$
\begin{aligned}
&If \\
&\quad n > 1, \\
&T(n) = T(\frac{n}{2}) + O(1) \\
&If \\
&\quad n = 1 \\
&T(n) = O(1)
\end{aligned}
$$

Based on master theorem,$T(n) = aT(n/b) + \Theta(n^k(log_2^n)^p)$, we know that $a = 1, b = 2, k = 0, p = 0$ in our case.

$$
\begin{aligned}
a = b^k &= 1 \\
p = 0 &> -1 \\
log_b(a) &= 0 \\
T(n) &= \Theta(n^0 log(n)) \\
&= \Theta(log(n))
\end{aligned}
$$

(b)

Algorithm 1. My Binary Search Algorithm

Input: sorted array A (indexed from 1), search item x
Output: index into A of item x if found, zero otherwise

```
1:  function BINARY-SEARCH(x, A)
2:      i = 1                                   ▷ lower search bound
3:      j = len(A)                              ▷ upper search bound
4:      while i < j do                          ▷ while places remain to be checked
5:          m = ⌊(i + j)/2⌋                     ▷ assign the midpoint
6:          if x = A[m] then
7:              loc = i
8:              return loc
9:          if x > A[m] then
10:             i = m + 1                       ▷ increase lower to mid
11:         else
12:             j = m                           ▷ decrease upper to mid
13:     if x = A[i] then
14:         loc = i
15:     else
16:         loc = 0
17:     return loc
```

Your writeup goes here. You can use labels and references, such as note the initialization in line 2, and the while-check in line 4.

3 lines have been added from line6 to line8 which can improve the runtime. The tightest asymptotic representation doesn't change, still $\Theta \log n$ but if the middle point equals to the value we are searching for, we can drump out of the while loop quicker than orginal pseudocode.
Conclusions: both $\Theta log(n)$

Proposition 3.

Problem Statement: use master theorem to find the asymptotoic bounds of $T(n) = 4T(n/4) + n^2$
Assumptions: none
Computations: Based on master theorem,$T(n) = aT(n/b) + \Theta(n^k(log_2^n)^p)$, we know that $a = 4, b = 4, k = 2, p = 0$ in our case.

$$\begin{aligned} a < b^k &= 16 \\ p &= 0 \\ then & \\ T(n) &= \Theta(n^k) \\ &= \Theta(n^2) \end{aligned}$$

Conclusions $T(n) = \Theta(n^2)$.

Proposition 4.

Problem Statement: use master theorem to find the asymptotic bounds of $T(n) = 3T((\frac{n}{3} + 1) + n)$
Assumptions: none
Computations: let $n = m + \frac{3}{2}$

$$\begin{aligned} T(n) &= T(m + \frac{3}{2}) \\ &= 3T(\frac{m}{3} + \frac{1}{2} + 1) + m + \frac{3}{2} \\ &= 3T(\frac{m}{3} + \frac{3}{2}) + m + \frac{3}{2} \end{aligned}$$

Based on master theorem,$T(n) = aT(n/b) + \Theta(n^k(log_2^n)^p)$, we know that $a = 3, b = 3, k = 1, p = 0$ in our case.

$$\begin{aligned} a = b^k &= 3 \\ p &= 0 > -1 \\ T(n) &= \Theta(n^{log_b^a} log_2^{p+1}(n)) \\ &= \Theta(n \log(n)) \end{aligned}$$

Conclusions: $T(n) = \Theta(n \log(n))$

Proposition 5.

Problem Statement: although merge sort runs in $\Theta(n \log n)$ worst-case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertions sort can make it faster in practice for small problem sizes on many machines
(a) prove that insertion sort can sort that $\frac{n}{k}$ sublists, each of length k in $\Theta(nk)$ worst-case time
(b) prove how to merge the sublists in Theta(nlg(n/k)) worst-case time
(c) given that the modified algorithm runs in Theta(nk + nlg(n/k)) worst-case time, what is the largest value of k as a function of n which the modified algorithm has the same running time as standard merge sort, in terms of Theta-notation
(d) why would we ever do this-why not just merge sort? argue it the other way, too - what are some problems with using our modified method
Assumptions: none
Computations:
(a) we have $\frac{n}{k}$ k-element sequences, therefore the worst-case running time for insertion sort will be $\frac{n}{k}\Theta(k^2)$. Based on linearity property,

$$T(n) = \Theta(\frac{n}{k}(k^2)) = \Theta(nk)$$

(b) we can see this as a divide-and-conquer problem where we have m, number of sequences, and each has k units. Denote the time to solve this problem as T(m), we can use the same algorithm to solve the first half and second half of these m sequences, and combine two results together. Therefore the equation ($\Theta(mk)$ is 2-pint comparison)

$$T(m) = 2T(m/2) + \Theta(mk)$$

Based on master theorem

$$\begin{aligned} T(m) &= \Theta(mk \log(m)) \\ m &= \frac{n}{k} \\ T(n) &= \Theta(n \log(\frac{n}{k})) \end{aligned}$$

(c) we want to let $\Theta(nk + nlg(n/k))$ to be equal to $\Theta(nlg(n))$ since the $\Theta nlg(n/k)$ is asymptotically close to $\Theta(nlg(n))$, to chose largest value of k, we need to make $\Theta(nk)$ somehow close to $\Theta(nlg(n))$ as well.
Therefore, let $k = \Theta(lgn)$, we can get that

$$\begin{aligned} \Theta(nk + nlg(n/k)) &= \Theta(nlgn + nlg(n/lgn)) \\ &= \Theta(nlgn + nlgn - nlglgn) \\ &= \Theta(2nlgn - nlglgn) \\ &= \Theta(nlgn) \end{aligned}$$

(d) the recursive method might take extra space then original merge sort **Conclusions**: the statements were proofed for (a) and (b) and (c)