CS3340 Assignment 1

Bradley Assaly-Nesrallah (250779140)

January 24, 2019

1 2-1 Insertion sort on small arrays in merge sort

We consider a modification to mergesort 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. We know that insertion sort runs $\Theta(n^2)$ worst case time, thus for each list with k elements insertion sort takes $\Theta(k^2)$ worst case time. Thus when sorting n/k sublists of k elements each must take $\Theta(k^2n/k) = \Theta(kn)$ in the worst case time.

b. To merge the sublists in $\Theta(nlg(n/k))$ worst case time, suppose we have courseness k, so we merge as usual however we start at the level in which each array has size at most k. Thus the depth of the merge tree is $lg(n) - lg(k) = lg(n/k)$, so this is worst case $\Theta(lg(n/k))$. We know that each level of merging is $\Theta(n)$ worst case. Therefore combining these the worst case to merge the sublists with this method is $\Theta(nlg(n/k))$.

c. We know that the modified algorithm runs in $\Theta(nk + nlg(n/k))$ worst case run time. Note that $\Theta(nk + nlg(n/k)) = \Theta(nk + nlgn - nlgk)) = \Theta(nlgn)$. So to satisfy this, k cannot be larger that lgn otherwise nk will run worse that $\Theta(nlgn)$, so $k <= \Theta(lgn)$. Thus the largest asymptotic value of k as a function of n that satisfies the condition is $= \Theta(lgn)$.

d. In practice we should choose k by manually putting different values and timing to find the largest list length on which insertion sort is faster than merge sort. Note that insertion sort $T(n) = c1n^2$ and merge sort is $T(n) =$

$c2nlgn$, so to find k, we rearrage $c1k^2 <= c2klgk$, so $k <= (c2/c1)lgk$, in practice this would be done by timing to find such k as stated above.

2 3-2 Relative asymptotic growths

A	B	O	o	Ω	ω	Θ
$lg^k n$	n^c	yes	yes	no	no	no
k	c^n	yes	yes	no	no	no
$\sqrt{n}$	n^{sinn}	no	no	no	no	no
2^n	$2^n/2$	no	no	yes	yes	no
n^{lgc}	c^{lgn}	yes	no	yes	no	yes
$lg(n!)$	$lg(n^n)$	yes	no	yes	no	yes

3 4-2 Parameter-passing costs

a. We consider the binary search algorithm, with N the size of the original problem and n the size of a subproblem, we will give the recurrence for the worst case running time and the upper bound of the solution of the recurrences for each parameter passing case:
1. $T(n) = T(n/2) + c$. Solving the recursion we have $T(N) = \Theta(lgN)$ by master method, we let $a = 1, b = 2, f(n) = c$. so $n^{log_b a} = n^{log_2 1} = n^0 = 1$, so we have $f(n) = \Theta(n^{log_b a}) = \Theta(1) = c$ so we can apply case 2 of the master theorem, which gives $T(N) = \Theta(lgN)$
2. $T(n) = T(n/2) + \Theta(N)$. Solving the recursion we have $T(N) = cNlgN = \Theta(NlgN)$.
3. $T(n) = T(n/2) + \Theta(n/2)$. Solving the recursion we have $T(N) = \Theta(N)$ by master method, we let $a = 1, b = 2, f(n) = cn = n$. so $n^{log_b a} = n^0 = 1$, since $f(n) = \Omega(n^{log_b a + \epsilon})$ where $\epsilon = 1$, not $1(n) <= cn$ where $c > 1$ so we can apply case 3 of the master theorem, which gives $T(N) = \Theta(N)$.
b. We consider the merge sort algorithm as in a), and give the recurrence for the worst case running time and the upper bound of the solution of the recurrence for each case:
1. $T(n) = 2T(n/2) + cn$. Solving the recursion we have $T(N) = \Theta(NlgN)$ by master method, we let $a = 2, b = 2, f(n) = cn = n$. so $n^{log_b a} = n^{log_2 2} = n^1 = n$, so we have $f(n) = \Theta(n^{log_b a}) = \Theta(n)$, so we can apply case 2 of the master theorem, which gives $T(N) = \Theta(NlgN)$.
2. $T(n) = 2T(n/2) + cn + 2\Theta(N)$. Solving the recursion we have $T(N) = \Theta(NlgN) + \Theta(N^2) = \Theta(N^2)$.

3. $T(n) = 2T(n/2) + cn + 2c'n/2$. Solving the recursion we have $T(N) = \Theta(NlgN)$ by master theorem, we let $a = 2, b = 2, f(n) = cn = n$. so $n^{log_b a} = n^1 = n$, since $f(n) = \Theta(n^{log_b}) = n$ we can apply case 2 of the master theorem, which gives $T(N) = \Theta(NlgN)$.

4 Recurrence

Suppose that the running time of a program has the following recurrence relation:
$T(2) = 1$
$T(n) <= 2T(n/2) + nlog_2(n)$
We compute the time complexity using a recurrence tree then prove the answer by using the master theorem:
First we create a recurrence tree to determine the complexity of the relation $T(n) <= 2T(n/2) + nlog_2(n)$:, the recurrence tree is at the end of the document
We have determined that the running time is $\Theta(nlg^2n)$ with a recurrence tree, now we prove it using the master method. From the recurrence tree it is clear each branch is being added $\Theta(log_b n)$ times. The result is that the time complexity is $T(n) = \Theta(n^{log_b a} log_b^{k+1} n)$. So we have $a = b = 2, k = 1$, where $a = 2 = 2^1 = b^k$ so we may apply the master theorem case 2 to obtain that $T(n) = \Theta(nlg^2n)$, since by master theorem case 2 $T(n) = \Theta(n^{log_b a} log^{b+1} n) = \Theta(nlg^2n)$. Therefore we have determined the complexity of the recurrence via the recurrence tree method and proved it with the master method case 2 so we are done.

5 Sorting algorithm time complexity comparison

a. Insertion sort: implemented in python output time: n=2000 t=0.285s, n=20000, t= 25.33s
b. Merge sort: implemented in python output time: n=2000 t=0.049s, n=20000 t=0.137s, n=20000000, t=1m57.881s
c. Mergesort with insertionsort for size¡=k: we time output for n=20000000, and different k: k=2 t=1m54.029s, k=4 t=1m47.549s, k=8 t=1m45.506s, k=16 t=1m42.719s, k=32 t=1m47.820s, k=64 t=2m3.987s
d. We do not want to run insertion sort for large input sizes as it is $O(n^2)$, so that as input sizes grow very large the runtime increases exponentially, so

an input of 20000000 in python would take extremely long, as you can see from the growth of increasing from 2000 to 20000, a factor of 10, the time is almost 100x as large.
When comparing the results of b and c we can see that using the modified merge sort from c is faster than b for certain sizes of k . The fastest k tested was when k=16 where the time was around 1m42s compared to the regular megesort which was around 1m58s when both had an input size of n. This shows that the best value of k is 16. This shows that when using insertion sort for small lists, despite being $O(n^2)$ while mergesort is $O(nlogn)$ however insertionsort has a smaller constant and is thus faster for smaller input sizes, so we can use it when k¡= a certain size to save time, by reducing the number of calls necessary by the mergesort algorithm it can reduce the runtime, which is clearly observable by the testing for size n=20000000.

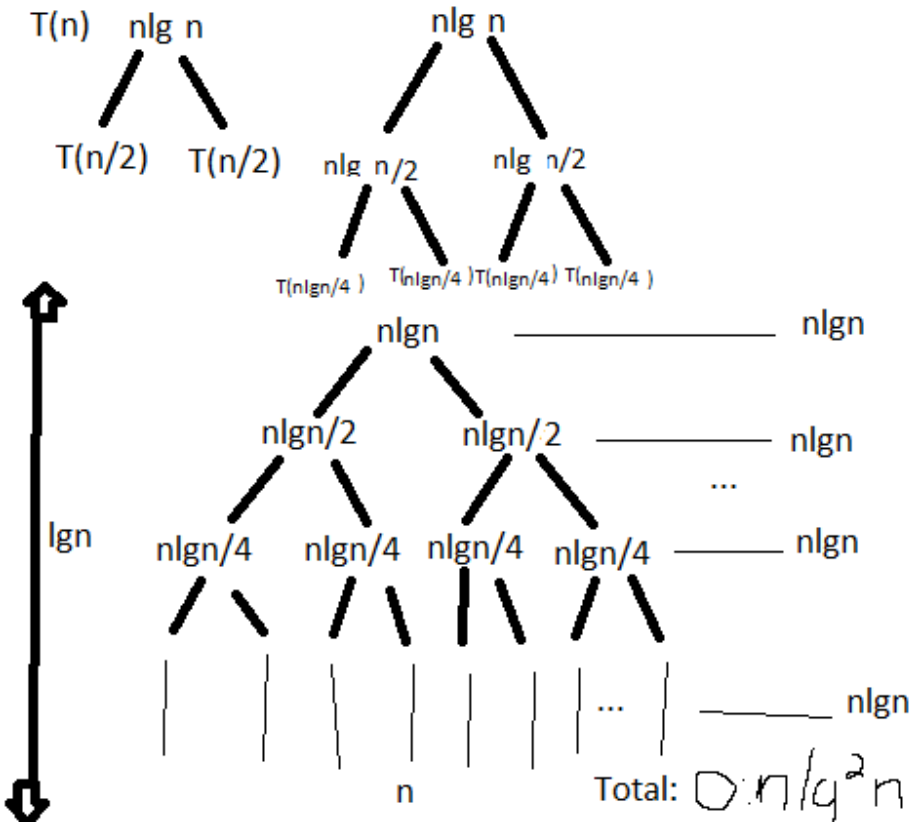

Figure 1: recursive tree for the function, from q4.