Assignment 1
CS 3340B

Name: Maryam Karimi Firozjaei
Student #: 250999590

---

1. Insertion sort on small arrays in merge sort

Although merge sort runs in $\Theta(n \lg n)$ worst-case time and insertion sort runs in $\Theta(n^2)$ 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. Show that insertion sort can sort the n/k sublists, each of length k, in $\Theta(nk)$ worst-case time.

Solution: Insertion sort runs in $\Theta(n^2)$ worst-case time. For sublists of length k it will take $\Theta(k^2)$ worst-case time. Therefore, sorting n/k sublists, each of length k, takes $\frac{n}{k} \times \Theta(k^2) = \Theta(nk)$.

---

b. Show how to merge the sublists in $\Theta(n \lg (n/k))$ worst-case time.

Solution:

$\lg (n/k)$	Merging n/k sublists into n/2k sublists	$\Theta(n)$ worst-case time
	Merging n/2k sublists into n/4k sublists	$\Theta(n)$ worst-case time
	⋮	
	Merging 2 sublists into 1 list	$\Theta(n)$ worst-case time

To merge the sublists, we need to take 2 sublists at a time and merge them. To merge all of the sublists, there will be $\lg (n/k)$ steps. Every merge takes $\Theta(n)$ so the whole process will run at $\lg (n/k) \times \Theta(n) = \Theta(n \lg (n/k))$ worst-case time.

---

c. Given that the modified algorithm runs in $\Theta(nk + n \lg (n/k))$ 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?

Solution: Merge sort runs in $\Theta(n \lg n)$ worst-case time and we want:
$\Theta(n \lg n) = \Theta(nk + n \lg (n/k))$
For the above statement to be true, either $\Theta(n \lg n) = nk$ or $\Theta(n \lg n) = n \lg (n/k)$.
Thus, the largest value of k is $\lg(n)$. If we substitute, we get:
$\Theta(nk + n \lg (n/k)) = \Theta(n \lg(n) + n \lg (n/ \lg(n))) = \Theta(n \lg n)$

---

d. How should we choose k in practice?

We should use k in practice such that insertion sort is faster than merge sort for lists of that size and k is the largest value possible. (For example if insertion sort is faster than merge sort for both sizes of 8 and 16, we pick the largest one: 16).

Name: Maryam Karimi Firozjaei
Student #: 250999590

2. Relative asymptotic growths

Indicate, for each pair of expressions (A, B) in the table below, whether A is O, o, Ω, ω, or Θ of B. Assume that $k \geq 1$, $\epsilon > 0$, and $c > 1$ are constants. Your answer should be in the form of the table with "yes" or "no" written in each box.

	A	B	O	O	Ω	ω	Θ
a.	$\lg^k n$	n^{ϵ}	yes	yes	no	no	no
b.	n^k	c^n	yes	yes	no	no	no
c.	$\sqrt{n}$	$n^{\sin n}$	no	no	no	no	no
d.	2^n	$2^{n/2}$	no	no	yes	yes	no
e.	$n^{\lg c}$	$c^{\lg n}$	yes	no	yes	no	yes
f.	$\lg(n!)$	$\lg(n^n)$	yes	no	yes	no	yes

3. Parameter-passing costs

Throughout this book, we assume that parameter passing during procedure calls takes constant time, even if an N-element array is being passed. This assumption is valid in most systems because a pointer to the array is passed, not the array itself. This problem examines the implications of three parameter-passing strategies:

1. An array is passed by pointer. Time = $\Theta(1)$
2. An array is passed by copying. Time = $\Theta(N)$ where N is the size of the array.
3. An array is passed by copying only the subrange that might be accessed by the called procedure. Time = $\Theta(q - p + 1)$ if the subarray $A[p..q]$ is passed.

a. Consider the recursive binary search algorithm for finding a number in a sorted array (see Exercise 2.3-5). Give recurrences for the worst-case running times of binary search when arrays are passed using each of the three methods above, and give good upper bounds on the solutions of the recurrences. Let N be the size of the original problem and n be the size of a subproblem.

1. $T(n) = T(n/2) + c = \Theta(\lg n)$ (master method)
2. $T(n) = T(n/2) + cN = 2cN + T(n/4) = \sum_{i=0}^{\lg n-1}(2^i cN/2^i) = \sum_{i=0}^{\lg n-1}(cN) = cN\sum_{i=0}^{\lg n-1}(1) = cN\lg(n)\text{-}1+1 = cN\lg(n) = \Theta(n\lg n)$
3. $T(n) = T(n/2) + cn = \Theta(n)$ (master method)

Name: Maryam Karimi Firozjaei
Student #: 250999590

---

b. Redo part (a) for the MERGE-SORT algorithm from Section 2.3.1.

1. $T(n) = 2T(n/2) + cn = \Theta(n \lg n)$ (master method)

2. $T(n) = 2T(n/2)cn + 2N = 4N + cn + 2c(n/2) + 4T(n/4) = \sum_{i=0}^{\lg n-1}(cn + 2^i N) = \sum_{i=0}^{\lg n-1}(cn + N\sum_{i=0}^{\lg n-1} 2^i) = cn \lg n + N\frac{2^{\lg n}-1}{2-1} = cn \lg n + nN - N = \Theta(nN) = \Theta(n^2)$

3. $T(n) = 2T(n/2) + cn + 2n/2 = 2T(n/2) + (c+1)\ n = \Theta(n \lg n)$ (master method)

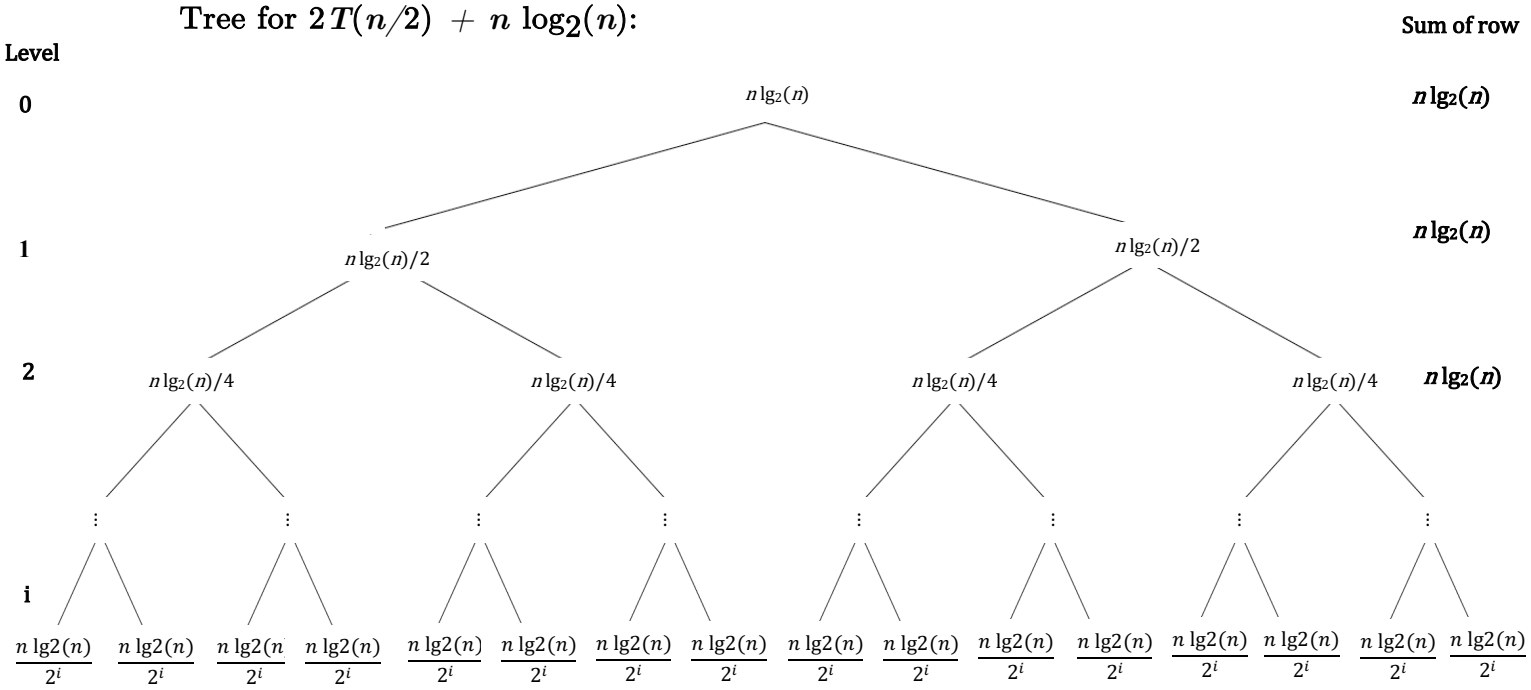
---

4. Suppose that the running time of a recursive program is represented by the following recurrence relation:

$T(2) = 1$
$T(n) \leq 2T(n/2) + n \log_2(n)$

Determine the time complexity of the program using recurrence tree method and then prove your answer.

Tree for $2T(n/2) + n \log_2(n)$:



Time Complexity $= \Theta(n(\lg_2(n))^2)$

Proof: First, we look at the tree for $2T(n/2) + n \log_2(n)$. We see that every row has a run time of $n \lg_2(n)$ worst-case. The base case is when n =2. So, to find the height of this tree, we solve the following line:

$\frac{n \lg 2(n)}{2^i} = 2 \Rightarrow n \lg_2(n) = 2^{i+1} \Rightarrow \lg_2(n \lg_2(n)) = i+1 \Rightarrow i = \lg_2(n \lg_2(n))-1$

Now, we calculate the following:
$\sum_{i=0}^{\lg2(n\,\lg2(n))-1} n\,\lg2(n) = n\,\lg2(n)\sum_{i=0}^{\lg2(n\,\lg2(n))-1} 1 = n\,\lg2(n)\times(\lg2(n\,\lg2(n)) - 1 + 1)$
$= n\,\lg2(n) \times \lg2(n\,\lg2(n)) = n\,\lg2(n) \times [\lg2(n) + \lg2(\lg2(n))] = n\,\lg2(n)\,\lg2(n) +$
$n\,\lg2(n)\,\lg2(\lg2(n)) = \Theta(\,n\,\lg2(n)\,\lg2(n)) = \Theta(n\,(\lg2(n))^2)$
Since $T(n) \leq 2T(n/2) + n\,\log_2(n)$, $T(n) \leq \Theta(n\,(\lg2(n))^2)$.
Therefore, the running time of this program is $\Theta(n\,(\lg2(n))^2)$

---

5. Comparison of three sorting algorithms with time complexities of $O(n^2)$, O(n log(n)) and O(n log(n)). The difference of the last two methods are between constants.

a) Implement InsertionSort algorithm and then run it on reversely sorted integer input of sizes 20000 and 200000 (2000 and 20000 if you are using python). The output is the first 20 integers before and after sorting.

Please refer to asn1_a.java for my source code. Using terminal, you can run it using the following codes:

javac asn1_a.java
java asn1_a size

```
bash-4.4$ time java asn1_a 20000
--------------------------------
First 20 elements BEFORE sorting:
[20000,19999,19998,19997,19996,19995,19994,19993,19992,19991,19990,19989,19988,19987,19986,19985,19984,19983,19982,19981]
First 20 elements AFTER sorting:
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
--------------------------------

real    0m0.200s
user    0m0.218s
sys     0m0.022s
bash-4.4$ time java asn1_a 200000
--------------------------------
First 20 elements BEFORE sorting:
[200000,199999,199998,199997,199996,199995,199994,199993,199992,199991,199990,199989,199988,199987,199986,199985,199984,199983,199982,199981]
First 20 elements AFTER sorting:
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
--------------------------------

real    0m6.623s
user    0m6.575s
sys     0m0.038s
```

b) Implement MergeSort algorithm and then run it on reversely sorted integer input of sizes 20000, 200000, and 200,000,000 (2000, 20000, and 20,000,000 if you are using python). The output is the first 20 integers before and after sorting.
Please refer to asn1_b.java for my source code. Using terminal, you can run it using the following codes:

javac asn1_b.java
java asn1_b size

```
bash-4.4$ time java asn1_b 20000
---------------------------------
First 20 elements BEFORE sorting:
[20000,19999,19998,19997,19996,19995,19994,19993,19992,19991,19990,19989,19988,19987,19986,19985,19984,19983,19982,19981]
First 20 elements AFTER sorting:
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
---------------------------------

real    0m0.149s
user    0m0.153s
sys     0m0.025s
bash-4.4$ time java asn1_b 200000
---------------------------------
First 20 elements BEFORE sorting:
[200000,199999,199998,199997,199996,199995,199994,199993,199992,199991,199990,199989,199988,199987,199986,199985,199984,199983,199982,199981]
First 20 elements AFTER sorting:
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
---------------------------------

real    0m0.188s
user    0m0.214s
sys     0m0.037s
bash-4.4$ time java asn1_b 200000000
---------------------------------
First 20 elements BEFORE sorting:
[200000000,199999999,199999998,199999997,199999996,199999995,199999994,199999993,199999992,199999991,199999990,199999989,199999988,199999987,199999986,199999985,199999984,199999983,199999982,199999981]
First 20 elements AFTER sorting:
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
---------------------------------

real    0m17.940s
user    0m24.655s
sys     0m2.376s
```

c) Implement your algorithm of Question 1, i.e. MergeSort when size greater than k and InsertionSort when size less than or equal to k, and then run it with k = 2, 4, 8, 16, 32, 64 on reversely sorted integer input of size 200,000,000 (20,000,000 if you are using python). The output is the first 20 integers before and after sorting.
Please refer to asn1_c.java for my source code. Using terminal, you can run it using the following codes:

javac asn1_c.java
java asn1_c size k

```
bash-4.4$ time java asn1_c 200000000 2
---------------------------------
First 20 elements BEFORE sorting:
[200000000,199999999,199999998,199999997,199999996,199999995,199999994,199999993,199999992,199999991,199999990,199999989,199999988,199999987,199999986,199999985,199999984,199999983,199999982,199999981]
First 20 elements AFTER sorting:
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
---------------------------------

real    0m17.469s
user    0m23.980s
sys     0m2.301s
bash-4.4$ time java asn1_c 200000000 4
---------------------------------
First 20 elements BEFORE sorting:
[200000000,199999999,199999998,199999997,199999996,199999995,199999994,199999993,199999992,199999991,199999990,199999989,199999988,199999987,199999986,199999985,199999984,199999983,199999982,199999981]
First 20 elements AFTER sorting:
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
---------------------------------

real    0m15.244s
user    0m21.483s
sys     0m2.187s
bash-4.4$ time java asn1_c 200000000 8
---------------------------------
First 20 elements BEFORE sorting:
[200000000,199999999,199999998,199999997,199999996,199999995,199999994,199999993,199999992,199999991,199999990,199999989,199999988,199999987,199999986,199999985,199999984,199999983,199999982,199999981]
First 20 elements AFTER sorting:
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
---------------------------------

real    0m14.340s
user    0m19.259s
sys     0m2.170s
bash-4.4$ time java asn1_c 200000000 16
---------------------------------
First 20 elements BEFORE sorting:
[200000000,199999999,199999998,199999997,199999996,199999995,199999994,199999993,199999992,199999991,199999990,199999989,199999988,199999987,199999986,199999985,199999984,199999983,199999982,199999981]
First 20 elements AFTER sorting:
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
---------------------------------

real    0m14.278s
user    0m19.105s
sys     0m2.134s
bash-4.4$ time java asn1_c 200000000 32
---------------------------------
First 20 elements BEFORE sorting:
[200000000,199999999,199999998,199999997,199999996,199999995,199999994,199999993,199999992,199999991,199999990,199999989,199999988,199999987,199999986,199999985,199999984,199999983,199999982,199999981]
First 20 elements AFTER sorting:
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
---------------------------------

real    0m16.220s
user    0m22.529s
sys     0m2.422s
bash-4.4$ time java asn1_c 200000000 64
---------------------------------
First 20 elements BEFORE sorting:
[200000000,199999999,199999998,199999997,199999996,199999995,199999994,199999993,199999992,199999991,199999990,199999989,199999988,199999987,199999986,199999985,199999984,199999983,199999982,199999981]
First 20 elements AFTER sorting:
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
---------------------------------

 real    0m16.975s
 user    0m23.750s
 sys     0m2.186s
```

d) Use the UNIX bash command "time" to track the time needed to compute for each run of the three algorithms.

• Compare the results for a) and b).
Answer: For inputs of size 20000 and 200000, b (MergeSort) is faster. It can be seen that the difference of run times for 200000 is significant, and this is because MergeSort has a better run-time for bigger inputs which makes this algorithm suitable for larger values.

• Why we do not want to run a) on input of size 200,000,000 (20,000,000 for python).
Answer: Because InsertionSort has a poor performance for big values. However, it can have a better run-time for smaller sizes such as 16.

• Compare the results for b) and c).
Answer: The algorithm in part c performs better than b for all values of k. When we use InsertionSort for smaller array sizes in MergeSort, it improves the run-time. For example, for input size of 200000000, when we use the algorithm in part c instead of the one in part b, the run time decreases 20% (14.278 as opposed to 17.940).

• What is the best value for k? And why with this k, c) is faster than b)?
Answer: The best value for k is 16. (10.8) / 8 is Also a good k (they are close)
It's because inside our MergeSort, we're using InsertionSort. Whenever the size of an array is smaller or equal to 16, InsertionSort is called. This is helpful because we are taking advantage of the fact that InsertionSort performs superior for arrays of smaller size.