CS 344
Fall 2020

Problem Set 1
Due: Sep 28, 2020, 11:59 p.m.

Name: (Yuting Chen) **NetID**: (yc1071)

Honor Code: Students may discuss and work on homework problems in groups, which is encouraged. However, each student must write down their solutions independently to show they understand the solution well enough to reconstruct it by themselves. Students should clearly mention the names of the other students who offered discussions. We check all submissions for plagiarism. We take the honor code seriously and expect students to do the same.

Instruction for Submission: This homework has a total of 100 points, it will be rescaled to 10 points as the eventual score. We have provided the homework1.tex file for you, please write your answer to each question in this latex file directly after the corresponding question, and submit the complied PDF file only. You should name your PDF file as "Firstname-Lastname-NetID.pdf".

Discussion Group (People with whom you discussed ideas used in your answers if any):Xudong Jiang

I acknowledge and accept the Honor Code. Please type your initials below:
Signed: (YC)

---

1. **Big-O notation.** We have learnt big-O notation to compare the growth rates of functions, this exercise helps you to better understand its definition and properties.

   (a) (10 points) Suppose n is the input size, we have the following commonly seen functions in complexity analysis: $f_1(n) = 1, f_2(n) = \log n, f_3(n) = n, f_4(n) = n \log n, f_5(n) = n^2, f_6(n) = 2^n$. Intuitively, the growth rate of the functions satisfy $1 < \log n < n < n \log n < n^2 < 2^n$. Prove this is true.

   [**Hint**: You are expected to prove the following asymptotics by using the definition of big-O notation: $1 = O(\log n), \log n = O(n), n = O(n \log n), n \log n = O(n^2), n^2 = O(2^n)$. **Note**: Chap 3.2 of our textbook provides some math facts in case you need.]

   Answer:
   Using The Definition:$f(n)$ is $O(g(n))$ if and only if there exists positive constants "C" and "k" such that $|f(n)| <= C * |g(n)|$ for all $n >= k$.
   Note: All of the log is base on 10.
   let $f(n) = 1, g(n) = \log n$
   $1 \leq C * \log n$ (for all $n \leq k$)
   choose $k = 10$
   $1 \leq C * \log n$ (for all $n \leq 10$)
   $\frac{1}{\log n} \leq C$ (for all $n \leq 10$)
   choose $C = 1$

$1 \leq 1 * \log n$ (for all $n \leq 10$)
$1 \leq \log n$ (for all $n \leq 10$) that always true.
So $f(n) = O(g(n))$ implies $1 = O(\log n)$

let $f(n) = \log n, g(n) = n$
$\log n \leq C * n$ (for all $n \leq k$)
choose $k = 10$
$\log n \leq C * n$ (for all $n \leq 10$)
$\frac{logn}{n} \leq C$ (for all $n \leq 10$)
choose $C = \frac{1}{10}$
$\log n \leq C * n$ (for all $n \leq 10$)
$\log n \leq \frac{1}{10} * n$ (for all $n \leq 10$)
$1 \leq \frac{n}{10 \log n}$ (for all $n \leq 10$) that always true.
So $f(n) = O(g(n))$ implies $\log n = O(n)$

let $f(n) = n, g(n) = n \log n$
$n \leq C * n \log n$(for all $n \leq k$)
choose $k = 10$
$n \leq C * n * \log n$(for all $n \leq 10$)
$1 \leq C \log n$ (for all $n \leq 10$)
choose $C = 1$
$n \leq C * n * \log n$(for all $n \leq 10$)
$1 \leq \log n$ (for all $n \leq 10$) that always true.
So $f(n) = O(g(n))$ implies $n = O(n * \log n)$

let $f(n) = n \log n, g(n) = n^2$
$n \log n \leq C * n^2$ (for all $n \leq k$)
choose $n = 10$
$n \log n \leq C * n^2$ (for all $n \leq 10$)
$\log n \leq C * n$ (for all $n \leq 10$)
$\frac{\log n}{n} \leq C$ (for all $n \leq 10$)
choose $C = \frac{1}{10}$
$n \log n \leq \frac{1}{10} * n^2$ (for all $n \leq 10$)
$\log n \leq \frac{n}{10}$ (for all $n \leq 10$)
$1 \leq \frac{n}{10 \log n}$ (for all $n \leq 10$) that always true.
So $f(n) = O(g(n))$ implies $n \log n = O(n^2)$

let $f(n) = n^2, g(n) = 2^n$
$n^2 \leq C * 2^n$ (for all $n \leq k$)
choose $k = 1$
$n^2 \leq C * 2^n$ (for all $n \leq 1$)
$\frac{n^2}{2^n} <= C$ (for all $n \leq 1$)
choose $C = \frac{1}{2}$
$n^2 \leq \frac{1}{2} * 2^n$ (for all $n \leq 1$)

$1 \leq \frac{\frac{2^n}{2}}{n^2}$ (for all $n \leq 1$)
$1 \leq \frac{2^n}{2*n^2}$ (for all $n \leq 1$) that always true.
So $f(n) = O(g(n))$ implies $n^2 = O(2^n)$
In sum, we can get that he growth rate of the functions satisfy $1 < \log n < n < n \log n < n2 < 2n$.

(b) (10 points) Let $f, g : N \to R^+$, prove that $O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$. [**Hint**: The key is $\max\{f(n), g(n)\} \leq f(n) + g(n) \leq 2 \cdot \max\{f(n), g(n)\}$. **Note**: Proving this will help you to understand why we can leave out the insignificant parts in big-O notation and only keep the dominate part, e.g., $O(n^2+n \log n+n) = O(n^2)$.]
Answer:

Given $f, g : N \to R^+$
$f(n) \leq f(n) + g(n), g(n) \leq f(n) + g(n)$
Therefore, $\max\{f(n), g(n)\} \leq f(n) + g(n)$
Let $\max\{f(n), g(n)\} = f(n)$
Therefore, $g(n) \leq \max\{f(n), g(n)\}$
Thus, $f(n) + g(n) \leq 2 \max\{f(n), g(n)\}$

Let $\max\{f(n), g(n)\} = g(n)$
Therefore, $f(n) \leq \max\{f(n), g(n)\}$
Thus, $f(n) + g(n) \leq 2 \max\{f(n), g(n)\}$
From above, we can get that $\max\{f(n), g(n)\} \leq f(n) + g(n) \leq 2 \max\{f(n), g(n)\}$

From $f(n) + g(n) \leq 2 \max\{f(n), g(n)\}$ for every $n \geq 1$
We can get that $f(n) + g(n) = O(\max\{f(n), g(n)\}$.
From $\max\{f(n), g(n)\} \leq f(n) + g(n)$ for every $n \geq 1$
We can get that $\max\{f(n), g(n)\} = f(n) = g(n)$

In sum, $O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$, for $f, g : N \to R^+$.

2. **Proof of correctness.** (10 points) We have the following algorithm that sorts a list of integers to ascending order. Prove that this algorithm is correct. [**Hint:** You are expected to use mathematical induction to provide a rigorous proof.]

Answer:
Invariant of outer loop: At start of iteration the outer for loop,$A[0 : i - 1]$ is sorted.
Invariant of inner loop: at start of iteration the inner for loop, $A[min_index]$ is the smallest value from $A[i : j]$.

Algorithm 1: `Sort a list`

Input: Unsorted list $A = [a_1, \ldots, a_n]$ of n items
Output: Sorted list $A' = [a'_1, \ldots, a'_n]$ of n items in ascending order
for $i = 0$, $i < n - 1$, i++ **do**
    // Find the minimum element
    min_index = i
    for $j = i + 1$, $j < n$, j++ **do**
        if $A[j] < A$[min_index] **then**
            min_index = j
    // Swap the minimum element with the first element
    swap(A, i, min_index)
return A

Initialization:
Prior to the first iteration of outer loop, A[0] is the $A[min - index]$, the array $A[1 : i - 1]$ is empty.
Prior to the first iteration of inner loop, the sub-array $A[j : n]$ contains the single value.

Maintenance:
The invariant holds at the start of the iteration $j = i + 1$ of the inner for loop. At the start of the first iteration, A[i]is the $A[min_indx]$, then comparing the all elements with $A[min_indx]$ from $A[j : n]$. The smallest value instead of the $A[min_indx]$, became the new $A[min_indx]$.
At the end of each outer loop, swapping the $A[i], A[min_indx]$. Therefore, the sub array is sorted from $A[0 : i]$.

Termination:
When $i = n - 1$,the outer for loop is ended. All values in $A[0 : n - 2]$ are sorted and $A[length - 1]$ is biggest value from $A[0 : n - 1]$, therefore, all values in $A[0 : n - 1]$ are sorted.
when $i = n - 1, j = n$, the inner for loop is ended. $A[n - 1]$ is smallest value from $a[n - 1 : n]$.

3. **Practice the recursion tree.** (10 points) We have already had a recurrence relation of an algorithm, which is $T(n) = 2T(n/2) + 3n$. Solve this recurrence relation, i.e. express it as $T(n) = O(f(n))$, by using the recursion tree method. [**Note**: If you find it difficult to draw a picture using Latex directly, you can draw it using Microsoft PowerPoint and then save it as a picture file (.png or .jpg), or you can draw on a piece of paper by hand, and take a picture of it using your phone. Then, you can insert the picture into your latex code and compile it into the final PDF submission. The following code shows an example of how to insert figures in latex code, as shown in Figure 1.]

Answer:

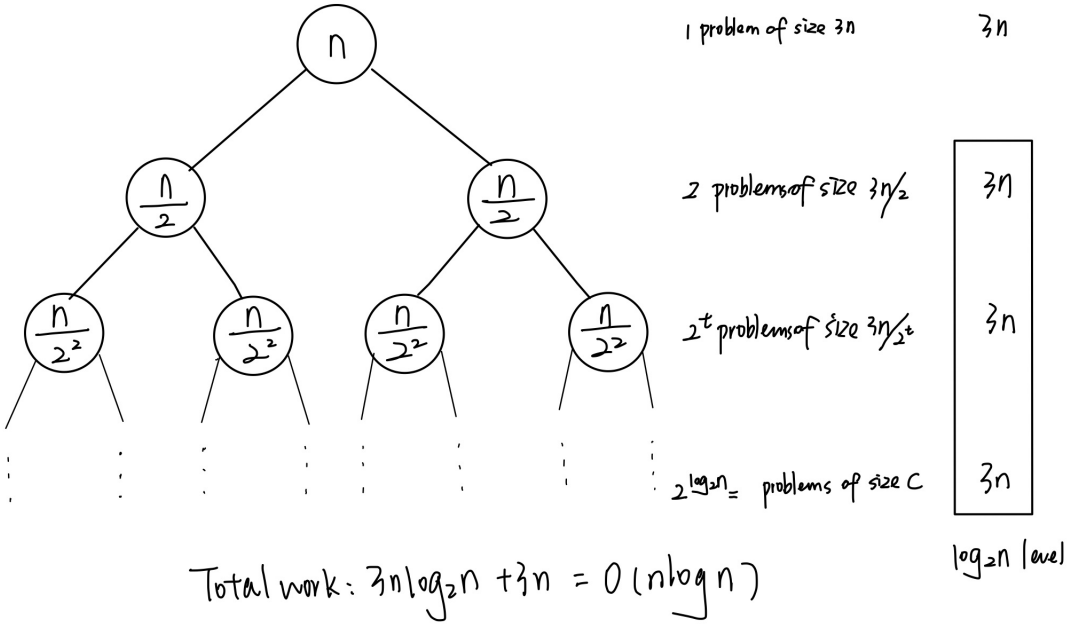

Figure 1: recursion tree.

4. **Practice with the iteration method.** We have already had a recurrence relation of an algorithm, which is $T(n) = 4T(n/2) + n^2 \log n$.

   (a) (5 points) Solve this recurrence relation, i.e., express it as $T(n) = O(f(n))$, by using the iteration method.

   Answer:
   Note: all the log is base on 2.
   Given $T(n) = 4T(n/2) + n^2 \log n$.
   Using the Iteration method: $T(n/2) = 4T(n/2^2) + (n/2)^2 \log(n/2)$.
   $T(n) = 4T(n/2) + n^2 \log n$
   $T(n) = 4[4T(n/2^2) + (n/2)^2 \log(n/2)] + n^2 \log n$.
   $T(n) = 4^2 T(n/2^2) + n^2 \log(n/2) + n^2 \log n$
   $T(n) = 4^2 T(n/4) + n^2[\log(n/2) + \log n]$
   As same we get that: $T(n/4) = 4T(n/2^3) + (n/2^2)^2 \log(n/2^2)$
   $T(n) = 4^2 T(n/4) + n^2(\log(n/2) + \log n)$
   $T(n) = 4^2[4T(n/2^3) + (n/2^2)^2 \log(n/2^2)] + n^2 \log(n/2) + n^2 \log n$
   $T(n) = 4^3 * T(n/2^3) + n^2 \log(n/2^2) + n^2 \log n/2) + n^2 \log n$
   $T(n) = 4^3 * T(n/2^3) + n^2[\log(n/2^2) + \log n/2) + \log n]$
   In sum,we can find that $T(n) = 4^k[T(n/2^k)] + n^2[\sum_{i=0}^{k-1} log(n/2^i)]$

Let $n = 1, T(1) = n/2^k = 1$
$n = 2^k, k = \log n$
$\sum_{i=0}^{k-1} log(n/2^i)$
$= \log n + \log(n/2) + \log(n/2^2) + ... + \log(n/2^{k-1})$
$= \log(\frac{n^k}{2^{1+2+3+4+...+k-1}}$
$= \log(\frac{n^k}{2^{\frac{k(k-1)}{2}}})$
$= \log(\frac{n}{2^{\frac{(k-1)}{2}}})^k$
$= k\log(\frac{n}{2^{\frac{(k-1)}{2}}})$
$= \log n * k\log(\frac{n}{2^{\frac{(k-1)}{2}}})$
$T(n) = 4^k * T(\frac{n}{2^k)+n^2(\sum_{i=0}^{k-1} log(n/2^i))}$
$T(n) = 4^{\log n} * T(\frac{n}{2^{\log n}}) + n^2 * \log n * \log(n/2^{\frac{(k-1)}{2}})$
$T(n) = n^2 * T(1) + n^2 * (\log n * \log(n/2^{\frac{(k-1)}{2}})$
$T(n) = O(n^2 * \log^2(n))$

(b) (5 points) Prove, by using mathematical induction, that the iteration rule you have observed in 4(a) is correct and you have solved the recurrence relation correctly. [**Hint**: You can write out the general form of $T(n)$ at the iteration step t, and prove that this form is correct for any iteration step t by using mathematical induction. Then by finding out the eventual number of t and substituting it into your general form of $T(n)$, you get the $O(\cdot)$ notation of $T(n)$.]

Answer:
Using the mathematical introduction
Base case: let $n = 2, T(2) = 4T(2/2) + 4\log 2 = 4T(1) + 4 = 4c + 4 \leq k(4\log^2(2))$
So, $T(2) = O(n^2\log^2(2))$is true.
Assume $T(n) = O(n^2 * \log^2 n)$is true.
let$n = k + 1$
$T(k+1) = 4T(k+1/2) + (k+1)^2 * \log(k+1)$
$T(k+1) = 4[(k+1/2)^2\log^2(k+1/2)] + (k+1)^2 * \log(k+1)$
$T(k+1) = (k+1)^2\log^2(k+1/2) + (k+1)^2 * \log(k+1)$
$T(k+1) = (k+1)^2 * (\log^2(k+1/2)] + \log(k+1)) \leq k(k+1)^2 * \log^2(k+1)$
$(\log^2(k+1/2) + \log(k+1)) \leq k\log^2(k+1)$
$(\log^2(k+1) - \log^2(2) + \log(k+1)) \leq k\log^2(k+1)$ is true.
So, we get that when$n = k+1, T(k+1) = O((k+1)^2 * \log^2(k+1))$
Thus, $T(n) = O(n^2 * \log^2(n))$ is true.

5. **Practice with the Master Theorem**. Solve the following recurrence relations; i.e. express each one as $T(n) = O(f(n))$ for the tightest possible function $f(n)$ using the

Master Theorem, and give a short justification. Unless otherwise stated, assume $T(1) = 1$. **[To see the level of detail expected, we have worked out the first one for you.]**

(z) $T(n) = 6T(n/6) + 1$. We apply the master theorem with $a = b = 6$ and with $d = 0$. We have $a > b^d$, and so the running time is $O(n^{\log_6(6)}) = O(n)$.

(a) (5 points) $T(n) = 3T(n/4) + \sqrt{n}$

Answer:
Using master method $T(n) = a * T(n/b) + O(n^d)$
$T(n) = 3T(n/4) + \sqrt{n}$
we get that $a = 3, b = 4, d = 1/2$
$b^d = 4^{1/2} = 2, a > b^d$
so, $O(n^{\log_b(a)} = O(n^{\log_4(3)})$

(b) (5 points) $T(n) = 7T(n/2) + \Theta(n^3)$

Answer:
$T(n) = 7T(n/2) + \Theta(n^3)$
$a = 7, b = 2, d = 3$
$b^d = 2^3 = 8, a < b^d$
So, $O(n^d) = O(n^3)$

(c) (5 points) $T(n) = 2T(n/3) + n^c$, where $c \geq 1$ is a constant that doesn't depend on n.

Answer:
$T(n) = 2T(n/3) + n^c$
$a = 2, b = 3, d = c \geq 1$
$b^d \geq 3$
so, $a < b^d$, $O(n^d) = O(n^c)$

6. **Proof of the Master Theorem.** (15 points) Now that we have practiced with the recursion tree method, the iteration method, and the Master method. The Master Theorem states that, suppose $T(n) = a \cdot T(n/b) + O(n^d)$, we have:

$$T(n) = \begin{cases} O(n^d \log n), & if\ a = b^d \\ O(n^d), & if\ a < b^d \\ O(n^{\log_b a}), & if\ a > b^d \end{cases}$$

Prove that the Master Theorem is true by using either the recursion tree method or the iteration method.

Answer:
Assume $n = b^k and O(n^d) = n^d$
$T(n) = T(b^k) = a * T(b^{k-1} + b^{k*d} = a(a * T(b^{k-2} + b^{d(k-1)}) + b^{k*d}$
$T(n) = a^2 * T(b^{k-2}) + ab^{d(k-1)} + b^{k*d}$
$T(n) = a^3 * T(b^{k-3} + a^2 * T(b^{d(k-2)}) + ab^{d(k-1)} + b^{k*d}$
In sum, $T(n) = a^k * T(1) + \sum_{i=0}^{k-1} a^i * b^{d(k-i)}$
let $p = b^d, q = a/r$
$p^k = b^{b*k} = (b^k)^d = n^d, a^k = a^{\log_b(n)} = n^{\log_b(a)}$

So, we get that $T(n) = c * n^{\log_b(a)} + n^d \sum_{i=0}^{k-1} q^i$
There are three situations in $X = \sum_{i=0}^{k-1} q^i$ for different values of q.
when $a = b^d, d = \log_b(a), p = a$
Therefore, $q = 1, \sum_{i=0}^{k-1} q^i = k, X = O(n^d \log n)$
when $a < b^d, d > \log_b(a), p > a$
Therefore, $q < 1, X = n^d * \frac{q^k - 1}{q-1} = O(r^k * q^k) = O(n^d)$
when $a > b^d, d < \log_b(a), p < a$
Therefore, $q > 1, X = n^d * \frac{q^k - 1}{q-1} = O(r^k * q^k) = O(a^k) = O(n^{log_b(a)}$

7. **Algorithm design.** Each of n users spends some time on a social media site. For each $i = 1, \ldots, n$, user i enters the site at time a_i and leaves at time $b_i \geq a_i$. You are interested in the question: how many distinct pairs of users are ever on the site at the same time? (Here, the pair (i, j) is the same as the pair (j, i)).

   Example: Suppose there are 5 users with the following entering and leaving times:

User	Enter time	Leave time
1	1	4
2	2	5
3	7	8
4	9	10
5	6	10

   Then, the number of distinct pairs of users who are on the site at the same time is three: these pairs are $(1, 2)$, $(4, 5)$, $(3, 5)$. (Drawing the intervals on a number line may make this easier to see).

   (a) (10 points) Given input $(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)$ as above in no particular order (i.e., not sorted in any way), describe a straightforward algorithm that takes $\Theta(n^2)$-time to compute the number of pairs of users who are ever on the site at the same time, and explain why it takes $\Theta(n^2)$-time. [**We are expecting pseudocode and a brief justification for its runtime.**]

   Answer:
   input: List containing the time interval of each users $(a1, b1), (a2, b2), (a3, b3), ...(an, bn)$.
   There are n users.

   for $i = 0, i < n - 1, i++$ (outer loop)
   for $j = i + 1, j < n, j++$ (inner loop)
   if $A[ai] >= A[bj] or A[bi] <= A[aj]$
   Output: (i, j)
   We compare the each user's time intervals with all the time intervals of the left users. if we find the overlap, we will output the serial numbers of two users with overlapping time.
   Time complexity $= n - 1 + n - 2 + n - 3 + n - 4 + ... + 1 = n * (n - 1)/2 = O(n^2)$, because outer loop need to run from $i = 0$ to $i = n - 2$. the inner loop need to run from $j = i + 1 to j = n - 1$.

(b) (10 points) Give an $\Theta(n \log(n))$-time algorithm to do the same task and analyze its running time. (**Hint:** consider sorting relevant events by time). [**We are expecting pseudocode and a brief justification for its runtime.**]

Answer:
Input: List containing the time interval of each users $(a1, b1), (a2, b2), (a3, b3), ...(an, bn)$
$A = [a1, a2, a3...an]$ of n users. (All of the start time)
$B = [b1, b2, b3..bn]$ of n users. (All of the end time)

Procedure insert(valueA, root)
if $(root == null)$
root= new node(valueA)
return
if $(root.data > valueA)$
$root.left = newnode(valueA)$
return
if$(root.data < valueA)$
if $(root.right == null)$
$root.right = newnode(valueA)$
return
insert$(valueA, root.right$
for $i = 0, i < n, i++$ (outer loop)
for $j = 0, j < n, j++$ (inner loop)
if $A[bi] <= A[aj]$
Output: $(i, j)(i + 1, j)..(n, j)$
end procedure
First, We put the start time and end time in two different arraysThen, we used the binary search tree to sort the array A with ascending. final, compare the user's end time with all the start time of the users, when we find the end time is bigger than start time, the left start time have overlap with end time.
Time complexity $O(logn) * O(n) = O(logn * n)$, because the big O of binary search tree is $O(logn)$,every time, we need compare the two array n times.