CS 344
Fall 2022

Problem Set 1
Due: October 10, 2022, 11:59 p.m.

Name: (Dong Shu) **NetID**: (ds1657)

Honor Code: Students may discuss homework problems with peers. 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 encourage you to use LaTex to write your answer, because it's particularly suitable to type equations and it's frequently used for writing academic papers. We have provided the homework1.tex file for you, you can write your answer to each question in this .tex file directly after the corresponding question, and then compile the .tex file into a PDF file for submission. As a quick introduction to LaTex, please see this *Learn LaTeX in 30 minutes* [1]. Compiling a .tex file into a PDF file needs installing the LaTex software on your laptop. It's free and open source. But if you don't want to install the software, you can just use this website `https://www.overleaf.com/`, which is also free of charge. You can just create an empty project in this website and upload the homework1.zip, and then the website will compile the PDF for you. You can also directly edit your answers on the website and instantly compile your file.

You can also use Microsoft Word or other software if you don't want to use LaTex. If so, please clearly number your answers so that we know which answer corresponds to which question. You also need to save your answers as a PDF file for submission.

Finally, please submit your PDF file only. You should name your PDF file as "Firstname-Lastname-NetID.pdf".

Late Policy: The homework is due on 10/10 (Monday) at 11:59pm. We will release the solutions of the homework on Canvas on 10/14 (Friday) 11:59pm. If your homework is submitted to Canvas before 10/10 11:59pm, there will no late penalty. If you submit to Canvas after 10/10 11:59pm and before 10/14 11:59pm (i.e., before we release the solution), your score will be penalized by 0.9^k, where k is the number of days of late submission. For example, if you submitted on 10/13, and your original score is 80, then your final score will be $80 * 0.9^3 = 58.32$ for $13 - 10 = 3$ days of late submission. If you submit to Canvas after 10/14 11:59pm (i.e., after we release the solution), then you will earn no score for the homework.

Make best use of picture in Latex: If you think some part of your answer is too difficult to type using Latex, you may write that part on paper and scan it as a picture, and then insert that part into Latex as a picture, and finally Latex will compile the picture into the final PDF

[1] `https://www.overleaf.com/learn/latex/Learn_LaTeX_in_30_minutes`

output. This will make things easier. Regarding how to insert pictures in Latex, you may refer to the Figure 1 in the following, which is an example of inserting pictures in Latex.

Discussion Peers (People with whom you discussed ideas used in your answers if any): Qi-Hang Wang, TianLe Chen,

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

---

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) (14 points) Suppose n is the input size, we have the following commonly seen functions in complexity analysis: $f_1(n) = 1, f_2(n) = \log n, f_3(n) = n, f_4(n) = n \log n, f_5(n) = n^2, f_6(n) = 2^n, f_7(n) = n!, f_8(n) = n^n$. Intuitively, the growth rate of the functions satisfy $1 < \log n < n < n \log n < n^2 < 2^n < n! < n^n$. Prove this is true.

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

   Answer: my answer to the question.

   We all know that 1 is a constant number. Thus, $f_1'(n) = 0$. For $f_2(n) = \log n$ it will become $f_2'(n) = 1/n$. We can tell that, $f_1(n) \leq f_2(n)$ when $n \geq 10$, and $f_1'(n) < f_2'(n)$ for $n \geq 1$. Therefore, $f_2(n)$ is upper bounded by $f_1(n)$. In this case $\log n$ is upper bounded by 1. So $1 = O(\log n)$.
   Same idea goes to $\log n = O(n)$. We know that $f_2'(n) = 1/n$. For the $f_3(n) = n$, it becomes $f_3'(n) = 1$. Since $f_2(1) < f_3(1)$ and $f_2'(n) \leq f_3'(n)$ for $n \geq 1$. Therefore, $f_3(n)$ is upper bounded by $f_2(n)$. In this case, n is upper bounded by $\log n$. So $\log n = O(n)$.
   For comparison between $f_3(n) = n$ and $f_4(n) = n \log n$, we can cancel the n from both side. In this case $f_3(n)$ will $= 1$, and $f_4(n)$ will $= \log n$. Since we already proved that $\log n$ is upper bounded by 1 from the first example. Thus, we can say that $n \log n$ is upper bounded by n. Therefore, we proved that $n = O(n \log n)$.
   For $f_4(n) = n \log n$ and $f_5(n) = n^2$, we can also cancel n from both side. In this case $f_4(n)$ will $= \log n$, and $f_5(n)$ will $=$ n. Since we already proved that n is upper bounded by $\log n$ from the second example. Thus we can say that n^2 is upper bounded by $n \log n$. Therefore, we proved that $n \log n = O(n^2)$.
   For $f_5(n) = n^2$ and $f_6(n) = 2^n$, We can use induction to prove 2^n is upper bounded by n^2. Base case: when n = 5, $f_5(n) = 5^2 = 25$ and $f_6(n) = 2^5 = 32$. So $f_5(5) < f_6(5)$, base case is true. Inductive step: let $k > 4$, we assume f(k) is true, and $k^2 < 2^k$. We need to prove $f(k+1)$, which is $(k+1)^2 < 2^{k+1}$, is also true. Assume $2^k > k^2$, we double two side, then we get $2^{k+1} > 2k^2$. Since we assume before that $n > 4$, so we

can separate $2k^2$ into $k^2+k*k > k^2+4k = k^2+2k+2k > k^2+2k+8 > k^2+2k+1$. Therefore, we can say that $2^{k+1} > k^2+2k+1$. To point out that $k^2+2k+1 = (k+1)^2$. So we can say that $2^{k+1} > (k+1)^2$. And now we successfully proved that for $2^k > k^2$ for $k > 4$. Therefore, we know that 2^n is upper bounded by n^2 when $n > 4$. So we can say that $n^2 = O(2^n)$.

For $f_6(n) = 2^n$ and $f_7(n) = n!$, we can prove $n!$ is upper bounded by 2^n by induction. Base case: when n = 4, we have $f_6(4) = 2^4 = 16$ and $f_7(4) = 4! = 24$. $f_6(4) < f_7(4)$, so base case is true. We assume that $2^k < k!$ for some integer $k \geq 4$. Inductive step: we need to prove that $2^{k+1} < (k+1)!$. Since we assumed that $2^k < k!$, so 2^{k+1} can be separate into $2^k * 2 < k! * 2$. Also, since we assume $k \geq 4$, so $k > 2$, so $k+1 > 2$. Now, we multiple $k!$ on both side of $2 < k+1$, we get $k! * 2 < k!(k+1)$. From the previous step, we can say that $k! * 2 < k!(k+1)$ is same as $2^{k+1} < k!(k+1)$ and is same as $2^{k+1} < (k+1)!$. And now we successfully proved that $2^k < k!$ is true follows that $2^{k+1} < (k+1)!$ is true. Therefore, we can say that $2^n < n!$ when $n \geq 4$. So $n!$ is upper bounded by 2^n, and $2^n = O(n!)$.

For $f_7(n) = n!$ and $f_8(n) = n^n$, we can prove n^n is upper bounded by $n!$ by induction. Base case: when n = 1, $f_7(1) = 1! = 1$ and $f_8(1) = 1^1 = 1$, so $f_7(1) \leq f_8(1)$. When n = 2 $f_7(2) = 2! = 2$ and $f_8(2) = 2^2 = 4$, so $f_7(2) < f_8(2)$. So base case is true. We assume that $k! < k^k$. We need to prove $(k+1)! < (k+1)^{k+1}$. First, we know that $(k+1)! = (k+1) * k * (k-1) * ... * 2 * 1 = (k+1)(k!)$. Since we assumed that $k! < k^k$, so we can say that $(k+1)(k!) < (k+1)k^k$. Since $k < k+1$, so $(k+1)(k!) < (k+1)k^k < (k+1)(k+1)^k = (k+1)^{k+1}$, which means $(k+1)! < (k+1)^{k+1}$. And now we successfully proved that $k! < k^k$ is true follows that $(k+1)! < (k+1)^{k+1}$ is true. Therefore, we can say that $n! < n^n$ when $n \geq 1$. So n^n is upper bounded by $n!$, and $n! = O(n^n)$.

From all the previous answers, we know that every question is the upper bound of the previous question. Therefore we have successfully proved $1 < \log n < n < n \log n < n^2 < 2^n < n! < n^n$ when n become larger and pass certain threshold.

(b) (3 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:

When we select a maximum value from $f(n), g(n)$, no matter which value we select, it is still smaller or equal to f(n) + g(n). Since we know that, $\max\{f(n), g(n)\} \leq f(n)+g(n)$, we can say that $\max\{f(n), g(n)\}$ is lower bound by $f(n)+g(n)$. However, if we multiple the maximum value in $f(n), g(n)$ by 2, then it should be bigger or equal to f(n) + g(n). Therefore, we can say that $f(n)+g(n) \leq 2 \cdot \max\{f(n), g(n)\}$. So that we can say that $f(n)+g(n)$ is lower bound by $2 \cdot \max\{f(n), g(n)\}$. Since $f(n)+g(n)$ is the lower bound, we can say that $O(f(n) + g(n)) = O(2 \cdot \max\{f(n), g(n)\})$, and we can ignore the 2, since it is a constant number. Therefore, we can say $O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$.

(c) (3 points) Let $f, g : N \to R^+$, prove that $\Omega(f(n) + g(n)) = \Omega(\max\{f(n), g(n)\})$.

[**Note**: Proving this will help you understand that we can also leave out the insignificant parts in big-Ω notation and the result is still a lower bound, e.g., $\Omega(n^2 + n \log n + n) = \Omega(n^2)$.]

Answer:

We all know that big-Ω is the opposite of big-O, which means lower bound. Since we already proved $\max\{f(n), g(n)\} \leq f(n) + g(n)$ from the previous question. Which means $\max\{f(n), g(n)\}$ is lower bounded by $f(n) + g(n)$. Therefore, we can say that $\Omega(f(n) + g(n)) = \Omega(\max\{f(n), g(n)\})$.

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

Algorithm 1: `Sort a list in descending order`

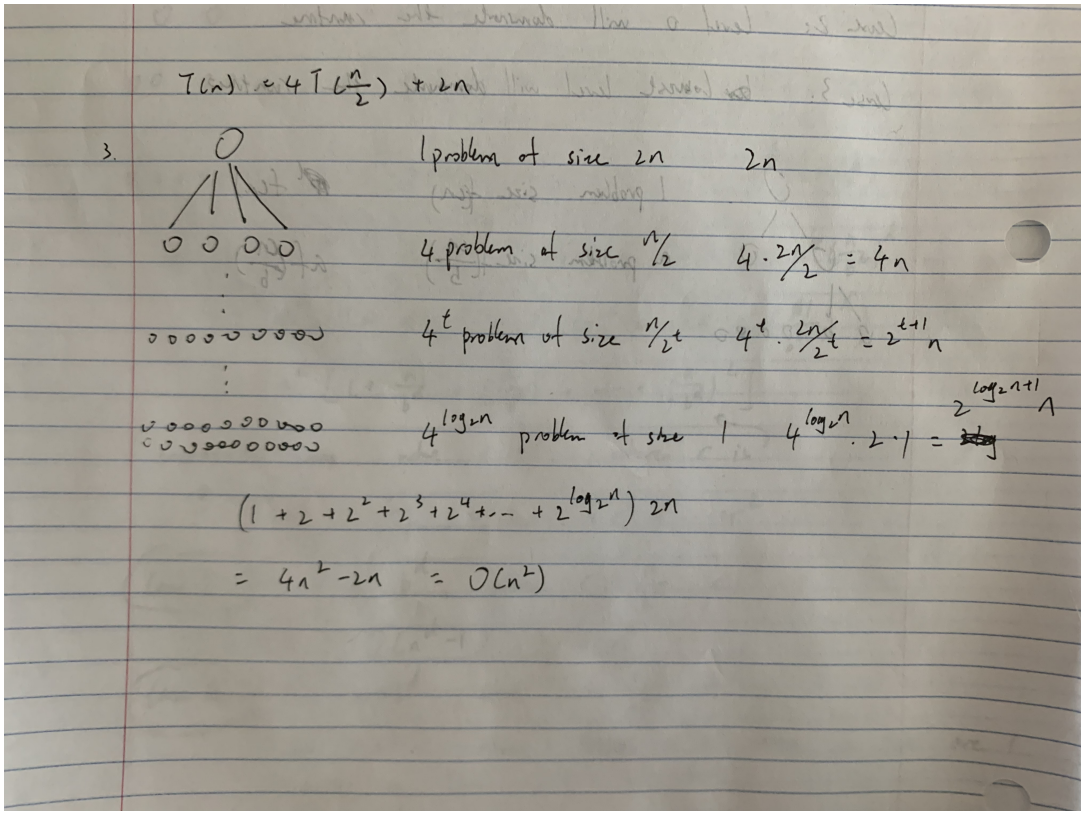
Input: Unsorted list $A = [a_1, \ldots, a_n]$ of n items
Output: Sorted list $A' = [a'_1, \ldots, a'_n]$ of n items in descending order
for $i = 0$, $i < n - 1$, i++ **do**
    // Find the maximum element
    max_index = i
    for $j = i + 1$, $j < n$, j++ **do**
        if $A[j] > A$[min_index] **then**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

Answer:
"*min_index*" should change to "*max_index*"
Theorem: Algorithm 1 will sort a list in descending order
Outer invariant: At the end of the outer for loop at iteration i, elements from A[0:i] are sorted, and elements in A[i+1:n] are all smaller than elements in A[0:i].
Inner invariant: At the end of the inner for loop at iteration j, the function will find the maximum index in A[i+1:j]. And it will store in the max_index.
Lemma: At the beginning, the A[0:i] is sorted since i = 0, and the array only have one value. If array A[0:i-1] is sorted at the end of iteration i-1, then A[0:i] will be sorted at the end of iteration i of the loop.
The invariant holds at the start of the iteration j = i+1 of the inner while-loop. To see why, at the start of the inner for loop at iteration j = i+1, we can see that max_index is already equate to i. Because, we are choosing a maximum index in A[i:i], which only contains one element. Therefore, we will have a max_index at in A[i:i].
Now, we will prove the inductive step. Suppose that the invariant holds at the start of an arbitrary iteration j = y (inductive hypothesis). We prove that it still holds by the end of iteration y (i.e., at the start of iteration j = y-1)

There are two cases of the while-loop condition to consider at the start of y:
The condition returns True.
First, A[j] will be the new max_index. That means A[j] will be the biggest number in A[i:j]. In other word the old max_index in A[i:y-1] will change to a new max_index in A[i:y]. Which means that we are maintaining the inner invariant.
The condition returns False.
In this case, the element in max_index is still the maximum index in A[i:y], and we do not need to change anything. In this case, we are still maintaining the inner invariant.
By the end of the 0-th iteration of the outer for-loop, A[0:0] (a single element) is trivially sorted.
By our lemma, if A[0:i-1] is sorted by the end of iteration i-1 of the loop, then A[0:i] will be sorted by the end of iteration i of the loop.
At the termination of the loop, our max_index will finally contain the biggest element in A[i:j], in this case, we can swap the A[i] with our max_index. We know that because our outer invariant, when A[o:i-1] is sorted, the elements in A[i+1:n] are all smaller than the elements in A[0:i], because we just swap the new max_index in to A[i]. Therefore, we know that A[0:i] will be sorted after the swap.
The loop terminates by the end of iteration length(A)-1, which implies that A[0:length(A)-1] is sorted when the loop ends, which proves the theorem.

3. **Practice the recursion tree.** (10 points) We have already had a recurrence relation of an algorithm, which is $T(n) = 4T(n/2) + 2n$. 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 6.]

Answer:



4. **Practice with the iteration method.** We have already had a recurrence relation of an algorithm, which is $T(n) = 4T(n/2) + n \log n$. We know $T(1) \leq c$.

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

   Answer:

   Since we know that $T(1) \leq c$, so that $T(n) \leq 4T(n/2) + n \log n$. Then we can reapplying the recurrence relation to $T(n/2)$ by assuming the problem size is $n/2$. Then we have $T(n) \leq 4(4T(n/4)+(n \log n)/2)+n \log n = 16T(n/4)+3n \log n$. Then we can recursively apply the recurrence relation on $T(n/4)$ by assuming the problem size is $n/4$. In this case, we have $T(n) \leq 16(2T(n/8) + (n \log n)/4) + 3n \log n =$

$32T(n/8) + 7n\log n$. Now we can discover some pattern of this recurrent relation, and the equation will looks like this: $T(n) \leq 4^k T(n/2^k) + (2^k - 1(n\log n))$, where k is the number of times to divide n by 2 to get 1.
But what is the final level of k? We want the question size in final level is 1, that means $n/2^k = 1 => 2^k = n => k = log_2 n$. So we can replace k with $log_2 n$. Now we have $T(n) \leq 4^k T(n/2^k) + (2^k - 1(n\log n)) = 4^{log_2 n} T(n/2^{log_2 n}) + (2^{log_2 n} - 1(n\log n)) = 2nT(1) + (n - 1(n\log n)) \leq 2cn + (n - 1(n\log n)) = O(n(n\log n)) = O(n^2\log n)$.

(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:

Since we already have the general form of T(n) from previous question, $T(n) \leq 4^k T(n/2^k) + (2^k - 1(n\log n))$. We are now ready to prove the base case:
T(0) = 0, which is right
$T(1) = 4^1 T(n/2^1) + (2^1 - 1(n\log n)) = 4T(n/2) + n\log n$, which is right.
Induction hypothesis: we assume for some n that $T(n) = 4^k T(n/2^k) + (2^k - 1(n\log n))$ is true.
Inductive step: Now we need to know does k+1 case also works.
$T(n) = 4T(n/2) + n\log n$
$T(n) \leq 4k(n/2)^2 + n^2\log n$
$T(n) \leq kn^2 + n^2\log n$
Since we know that n^2 is a subset of $(n^2 * \log n)$, where $\log n$ is a increasing function, so we know that $n^2\log n$ is upper bounded than n^2.
$T(n) \leq kn^2\log n + n^2\log n$
$T(n) = (k+1)n^2\log n$, which is correct
So we proved our induction

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) $T(n) = 4T(n/4) + \sqrt{n}$

Answer:
We apply the master theorem with a = 4, b = 4, and d = 1/2. We have $a > b^d$, and so the running time is $O(n^{\log_4(4)}) = O(n)$.

(b) (5 points) $T(n) = 6T(n/3) + \Theta(n^3)$

Answer:
We apply the master theorem with a = 6, b = 3, and d = 3. We have $a < b^d$, and so the running time is $O(n^d) = O(n^3)$.

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

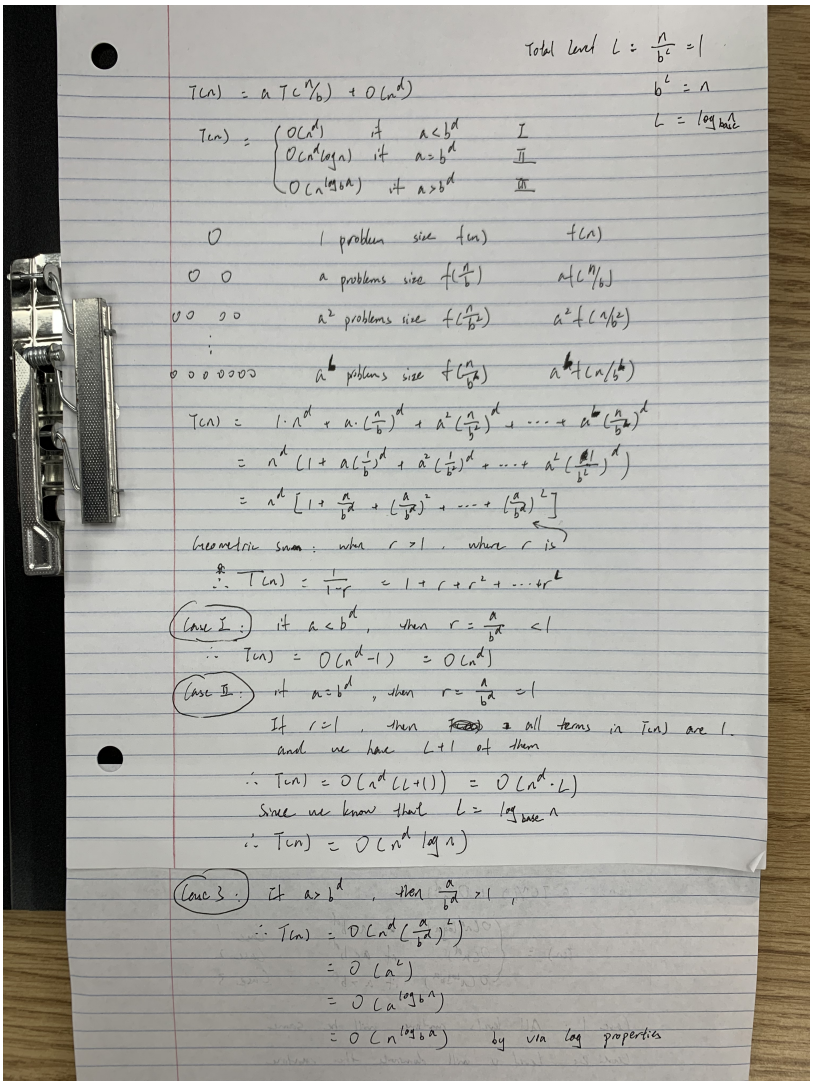
Answer:
We apply the master theorem with a = 2, b = 3, and $d \geq 1 = c$. We have $a < b^d$, and so the running time is $O(n^c)$.

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:



7. **Algorithm design with sorting.** 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	2
2	1	4
3	2	5
4	7	8
5	9	10
6	6	10

   Then, the number of distinct pairs of users who are on the site at the same time is five:

these pairs are $(1,2)$, $(1,3)$, $(2,3)$, $(4,6)$, $(5,6)$. (Drawing the intervals on a number line may make this easier to see).

(a) (5 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:

Algorithm 2: `Find every distinct pair of user`

Input: List A = n users
Output: List $answer = [(a_1, b_1), ..., (a_n, b_n)]$every distinct pair of users
Create an empty list(answer)
for $i = 0$, $i < n - 1$, i++ **do**
    user_a = i
    for $j = i + 1$, $j < n$, j++ **do**
        user_b = j
        //check if user_a leave time is greater or equal to user_b enter time
        if $user\_a\_b_i \geq user\_b\_a_i$ **then**
            append(user_a, user_b) into answer
return *answer*

We can see there is an outer for-loop and an inner for-loop. The running time for outer for-loop is n, because we need to iterate every user in the array. For each iteration of the outer for loop, we need to iterate n-i+1 elements in the inner loop, where i start at 0. Therefore total time is $\Theta(n^2)$.

(b) (5 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:

We can see that we are first using the merge sort or quick sort algorithm to sort the user array. The running time for those algorithms are $\Theta(n \log n)$. We also used only one for loop outside of the sort, and the running time is $\Theta(n)$. Therefore, the total running time is $\Theta(n \log n + n) = \Theta(n \log n)$

Algorithm 3: `Find every distinct pair of user`

Input: List A = n users
Output: List $answer = [(a_1, b_1), ..., (a_n, b_n)]$every distinct pair of users
Create an empty list(answer)
merge_sort or quick_sort(interval) use enter time
On_media = 0
for $i = 0$, $i < n - 1$, i++ **do**
    //check if leave time[i] is greater or equal to enter time[i+1]
    if $b_i[i] \geq a_i[i + 1]$ **then**
        On_media += 1
        append(On_media) into answer
return *answer*

8. **Algorithm design with divide and conquer.** Given an array $A[0 : n-1]$ of n integers, compute the maximum sum of non-empty consecutive subsequence present in the array.

   For example, if the input is $A = [3, 2, 5, 1, 6]$, then the output max_sum is 17 (the corresponding maximum sum consecutive subsequence is $[3, 2, 5, 1, 6]$. If the input is $A = [-2, 11, -4, 13, -5, -2]$, then the output max_sum is 20 (the corresponding maximum sum consecutive subsequence is $[11, -4, 13]$.

   (a) (5 points). Given input $A[0 : n - 1]$ and n, design a divide-and-conquer algorithm which outputs the maximum sum of non-empty consecutive subsequence in the array. You only need to output the sum value and do not need to output the exact subsequence. [**We are expecting a brief justification for the intuitive idea of your algorithm and the pseudocode of the algorithm.**]
   Answer:

   We first create some if conditions for n = 0 or n = 1. Then if the n is longer than 1, we first separate n into left part and right part. Then we calculate the sum of the left part and right part and store them in the sumL and sumR. We also store the sum of the whole array into sumWhole. After that we use a for loop to iterate the left subarray, which is i start from the middle and goes to 0. Every time we iterate 1 step, we add the current value into the leftTemp, and if the leftTemp is bigger than sumLeft, we store the value in the sumLeft. In this case, we will have the largest sum of consecutive subarray inside of left subarray, which is sumLeft. And we do the same thing with the right part. After that we should have two largest sum of consecutive subarray from left and right. Therefore when we add those two largest sum together, we should have the largest sum of consecutive subarray from the middle. After that we use max function to find the maximum value from sumWhole, sumL, sumR, sumLeft, sumRight and middle.

Algorithm 4: `algorithm divide_and_conquer`

```
Input: an array A[0:n-1] of n integers
Output: The maximum sum of non-empty consecutive subsequence present in the
        array
middle = 0
//sum of all number from the left
sumL = 0
//sum of all number from the right
sumR = 0
//sum of whole array
sumWhole = 0
//number one by one added from the left
sumLeft = 0
//number one by one added from the right
sumRight = 0
if n = 0 then
    return array cannot be empty
if n = 1 then
    return A[0]
else
    let left = first half of A
    let right = second half of A
    sumWhole = sum(left) + sum(right)
    sumL = sum(left)
    sumR = sum(right)
    //add all number one by one from the left
    leftTemp = 0
    for i = length(A)/2; i > 0; i − − do
        leftTemp += A[i]
        if leftTemp > sumLeft then
            sumLeft = leftTemp
    //add all number one by one from the right
    rightTemp = 0
    for i = length(A)/2; i < n − 1; i + + do
        rightTemp += A[i]
        if rightTemp > sumRight then
            sumRight = rightTemp
    middle = sumRight + sumLeft
    return max(sumWhole, sumL, sumR, sumLeft, sumRight, middle)
```

(b) (5 points). Let $T(n)$ be the runtime of your algorithm when the input size is n. Establish the recurrence relation of $T(n)$ for your algorithm, and then solve the recurrence relation to provide the big-O runtime of your algorithm. [**You can use any method we introduced in class to solve your recurrence relation.**]
Answer:

Since we separate the array into two parts, and have a for loop for each part, so our recurrence relation of T(n) will be: $2T(n/2) + cn$**.**
Therefore, we apply the Master Theorem with a = 2, b = 2, and d = 1.
We have $a = b^d$**, and so the running time is** $O(n \log n)$