CSE 6010 / CX 4010 Assignment 1
due Sep 24, 2020 at 11:59pm on Canvas

Ting Liao GTID: 903278773 (Chin Wang GTID: 903506612)

08/30/2020

Please type your answers (LaTeX highly recommended) and upload a single PDF including all your answers. If you want to draw a graph by hand you can take a picture of your drawing and insert it to your PDF file. Please make sure that your inserted picture can be clearly read. Do not forget to acknowledge your collaborators.

1 Simple Complexity (11 pts)

1. (6 pts) For each pair of functions f and g, write whether f is in $\mathbb{O}(g)$, $\Omega(g)$, or $\Theta(g)$.

   (a) $f = (n + 1000)^4$, $g = n^4 - 3n^3$

   (b) $f = \log_{1000} n$, $g = \log_2 n$

   (c) $f = n^{1000}$, $g = n^2$

   (d) $f = 2^n$, $g = n!$

   (e) $f = n^n$, $g = n!$

   (f) $f = \log n!$, $g = n \log n$ (hint: Stirling's approximation)

   Ans:

   (a) f is in $\Theta(g)$

   (b) f is in $\Theta(g)$

   (c) f is in $\Omega(g)$

   (d) f is in $\mathbb{O}(g)$

   (e) f is in $\Omega(g)$

   (f) f is in $\Theta(g)$

2. (5pts) Determine the Big-O time complexity for the algorithm below (**show your analysis**). Also, very briefly explain (in one or two sentences) what the algorithm outputs (note: the % symbol is the modulo operator):

```
Data: n
i = 1;
while $i \leq n$ do
    j = 0;
    k = i;
    while $k$ % $3 == 0$ do
        k = k/3 ;
        j++ ;
    end
    i++ ;
    print i,j;
end
```

Ans: Analysis steps

1. In this problem, we have n data.
2. When the code executes, the first while-loop (outer loop) will iterate n times (since i is initialized as 1, $i <= n$, i++), so the run time complexity is $O(n)$.
3. Then, for the inner while-loop, due to that it executes when the number is the multiple of 3, it will run $O(\log_3 n)$. So, the run time complexity here is $O(\log n)$.
4. Finally, we now consider the whole program and multiply the outer and inner while-loop. Therefore, we can know that the total time complexity is $O(n \log n)$.
5. If now we have n = 5, i will start from 1 until i = 5. Then, the inner loop will execute when k % 3 == 0. The program will iterate 5 times and the result is, (2,0) (3,0) (4,1) (5,1) (6,1) (if we let the result i,j shown as tuple).

2 Algorithm design and complexity (15 pt)

The problem consists of finding the lowest floor of a building from which a box would break when dropping it. The building has n floors, numbered from 1 to n, and we have k boxes. There is only one way to know whether dropping a box from a given floor will break it or not. Go to that floor and throw a box from the window of the building. If the box does not break, it can be collected at the bottom of the building and reused.

The goal is to design an algorithm that returns the index of the lowest floor from which dropping a box will break it. The algorithm returns $n + 1$ if a box does not break when thrown from the n-th floor. The cost of the algorithm, to be kept minimal, is expressed as the number of boxes that are thrown (note that re-use is allowed).

1. For $k \geq \lceil \log(n) \rceil$, design an algorithm with $O(\log(n))$ boxes thrown.

   Ans:

   Considering $O(\log n)$ time complexity, using a binary search tree algorithm might be a good choice. Also, due to that the number of boxes k is greater than $(\log n)$, we should do binary search of the floor n. For example, if we have a building with 11 floors, we can first throw a box at 6th floor ((1+11)/2 = 6). Then, if the box broke, we know that throwing boxes at any floor higher than 6th floor will break the box. So, we can then try (1+5)/2 = 3. However, if the box didn't break, then we could try (7+11)/2 = 9th floor. Based on this idea, we can find that the worst case will happen when the box breaks at 11th floor or 1st floor.

However, even meeting the worst case, we only need to try $O(\log_2 n)$ times. Also, we know that we have enough boxes ($k \geq \lceil \log(n) \rceil$) to throw. Therefore, by using this algorithm, the throwing time complexity will be $O(\log n)$.

2. For $k < \lceil \log(n) \rceil$, design an algorithm with $O\left(k + \frac{n}{2^{k-1}}\right)$ boxes thrown.
   Ans:

   Compared with last question, now we only have boxes which are less than ($\log n$). That means we can still implement the binary search tree algorithm, but we should do binary search of k boxes. In other word, we can't do binary search to n floors because if we met the worst case, we would not have enough boxes to throw. Another reason we do binary search to k boxes is that before we only have 1 box left and can only do linear search, we still can utilize $(k-1)$ boxes to do binary search.

   After throwing from 1 box to k-1 boxes, we will have 1 box and $\frac{n}{2^{k-1}}$ floors left. Because when every time we throw a box, the number of floors n will be divided by 2 ($=> n/2$). The time complexity here is $O(k-1)$, or we can say $O(k)$. Then, due to that we only have 1 box left, we can only do linear search, which will need to run $O(\frac{n}{2^{k-1}})$ times. As a result, the total throwing time complexity will be $O(k) + O(\frac{n}{2^{k-1}})$ equals to $O\left(k + \frac{n}{2^{k-1}}\right)$.

3. For $k = 2$, design an algorithm with $O(\sqrt{n})$ boxes thrown.
   Ans:

   Now we can only try once and then use the last box to do linear search. First, assume the maximum floor for throwing the box is N. Also, after each throw, if the box didn't break, we moved to the floor $2N-1$. This is because, at Nth floor we throw the first time. Then, at $2N-1th$ floor, we throw the second time. If the box broke at this floor, then we could do linear search from floor $N+1$ to $2N-2$. The total throwing times will be same as if the box broke at Nth floor.

   For example, assume $N = 10$ floors. If the box broke at 10th floor, we have to do linear search from 1st to 9th floor, which means we have to throw 10 times. However, if the box didn't break, we moved to the 19th floor ($2*10-1 = 19$). If the box broke at 19th floor, we could do linear search from 11th to 18th floor, and the total throwing times will be ($1+1+8 = 10$) same as the maximum throwing times 10. So the pattern will be $P : N...N+N-1...N+N-1+N-2...1$ and thus $P = \frac{N(N+1)}{2}$. To proof it, let's say N=10, so P = 10+9+8+7+6+5+4+3+2+1 = $\frac{10(10+1)}{2}$ = 55.

   Now we know that our P should be greater than floor n in order to check for all floors.

   $=> P = \frac{N(N+1)}{2} \geq n$

   $=> N(N+1) \geq 2n$

   $=> N \geq \frac{-1}{2} + (\sqrt{2n + \frac{1}{4}})$

   From above, the throwing time complexity should be $O(\sqrt{2n})$, and we could also write as $O(\sqrt{n})$. Therefore, we can proof that this is an algorithm with $O(\sqrt{n})$ boxes thrown.

Please explain your algorithms clearly.

3 Greedy - points on a 2D plane (12pt)

You are given n distinct points and one line l on the plane and some constant $r > 0$. Each of the n points is within distance at most r of line l (as measured along the perpendicular). You are to place disks of radius r centered along line l such that every one of the n points lies within at least one disk. Devise a greedy algorithm that runs in $O(n \log n)$ time and uses a minimum number of disks to cover all n points; prove its optimality.

Ans:

From the problem, I know that each point has a distance with the line smaller than r. Then, I can assume that each point should have a left point (Pl) and a right point (Pr) on the line, and it means that when a disk's center is located between Pl and Pr, the point will lie within the disk. Now I find that the problem seems to be similar to a topic mentioned in the course - interval scheduling problem. For example, Pl and Pr are pretty similar to the concept of the start time and the finish time.

Based on this knowledge, I try to sort all the points by considering their (Pr). Let's say I get a set of Pl and Pr => (Pl_1, Pr_1), (Pl_2, Pr_2) ... $(Pl_n - 1, Pr_n - 1)$, (Pl_n, Pr_n). Then, I will set the first interval ($v1$) at Pr_1 and loop through each point. If $Pl < v1$, I can know that this point is covered by the disk. However, whenever the Pr of a point (Let's say Ps) is greater than $v1$, I will set a new interval ($v2$) using the position of the point Pr_s. Due to doing sorting, the running time will be $O(n \log n)$ and thus should get the minimum number of disks covering all the points.

Now we have to prove that Greedy returns the optimal solution.

Proof:

Assume:

Point left: $Pl_1, Pl_2, Pl_3 ... Pl_{n-1}, Pl_n$

Point right: $Pr_1, Pr_2, Pr_3 ... Pr_{n-1}, Pr_n$

Greedy interval: $gv_1, gv_2.gv_3 ... gv_{n-1}, gv_n, gv_{last}$

Optimal interval: $o_1, o_2.o_3 ... o_{n-1}, o_n, o_{last}$

First, I have to claim that when $last \geq n$, $gv_n \geq o_n$ by the induction proof method. That is, when considering covering $P1$, $gv_1 \geq o_1$. So based on the assumption of induction, I assume that $gv_{n-1} \geq o_{n-1}$ and now I have to proof that $gv_n \geq o_n$. If a point's Pl_s is greater than gv_{n-1}, the Greedy interval has to create a new interval to cover it, let's say $gv_n = Pr_s$. Due to that $Pr_s \geq gv_{n-1} \geq o_{n-1}$, o_{n-1} can't cover Pl_s. Then, o_n needs to cover Pl_s, which means that $Pr_s \geq o_n \geq Pl_s$, and thus $gv_n \geq o_n$. As a result, it proves that Greedy returns the optimal solution.

4 Greedy - Why is the pool always so busy anyway? (12 pt)

Georgia Tech is trying to raise money for the Technology Square Research Building and has decided to host the "Tech Swim Run Bike" (TSRB) Triathalon to start off the fundraising campaign!

Usually in a triathalon all athletes will perform the three events in order (swimming, running, then biking) asynchronously, but unfortunately due to some last minute planning, the race committee was only able to secure the use of one lane of the olympic pool in the campus recreation center. This means that there will be a bottleneck during the first portion (the swimming leg) of the race where only one person can swim at a time. The race committee needs to decide on an ordering of athletes where the first athlete in the order will swim first, then as soon as the first athlete completes the swimming portion the next athlete will start swimming, etc. The race committee, not wanting to wait around for an extremely long time for everyone to finish, wants to come up with a schedule that will minimize the time it takes for everyone to finish the race. Luckily, they have prior knowledge about how fast the athletes will complete each portion of the race, and they have you, an algorithm-ista, to help!

Specifically, for each athlete, i, they have an estimate of how long the athlete will take to complete the swimming portion, s_i, running portion, r_i and biking portion, b_i. A schedule of athletes can be represented as a list of athletes (e.g. $[athlete_7, athlete_4, ...]$) that indicate in which order the athletes will start the race. Using this information, they want you to find an ordering for the athletes to start the race that will minimize the time taken for everyone to finish the race, assuming all athletes perform at their estimated time. More precisely, give an efficient algorithm that produces a schedule of athletes whose completion time is as small as possible and prove that it gives the optimal solution using an exchange argument.

Keep in mind that once an athlete finishes swimming, they can proceed with the running and biking portions of the race, even if other athletes are already running and biking. Also note that all athletes have to swim first (i.e. some athletes won't start off running or biking). For example: if we have a race with 3 athletes scheduled to go in the order $[2, 1, 3]$, and the finishing time for athlete i is represented as a_i, then the finishing times would be as follows: (with a total finishing time of $\max(a_1, a_2, a_3)$)

$$a_2 = (s_2) + r_2 + b_2$$
$$a_1 = (s_2 + s_1) + r_1 + b_1$$
$$a_3 = (s_2 + s_1 + s_3) + r_3 + b_3$$

Ans:

First, from the paragraph, I know that no matter how we arrange the athletes' schedule, the total time of swimming will always remain the same. So, if I let the athlete who takes longer time on running and biking to go swimming first, then I should be able to prevent idle time of the sport place.

Assume there were an ideal schedule which doesn't have any idle time.
=> Prove that our schedule has the minimum completion time.

Also, all ideal schedules with no idle time should have the same finish time. For example, several athletes' running+biking time is the same $(1 + 8, 2 + 7, 3 + 6, 4 + 5...)$, but this won't impact the finish time of our ideal schedule. While making the ideal schedule, if an athlete whose running+biking time is more than the previous athlete. It would definitely cause some idle time, but I can simply exchanged their position to solve this problem. However, I have to prove that after the exchange, the total time will not greater than the ideal schedule time.

Proof:

if $a_i = (s_i + ...) + r_i + b_i$ and $a_{i+1} = (s_i + s_{i+1} + ...) + r_{i+1} + b_{i+1}$

Then I found that $r_{i+1} + b_{i+1} \geq r_i + b_i$, so I exchanged a_i with a_{i+1}.

So the schedule will be modified to => $a_{i+1} = (s_{i+1} + ...) + r_{i+1} + b_{i+1}$ and $a_i = (s_i + s_{i+1} + ...) + r_i + b_i$ and thus prove that the time won't increase because after modified ($a_i = (s_i + s_{i+1} + ...) + r_i + b_i$) is smaller than previous ($a_{i+1} = (s_i + s_{i+1} + ...) + r_{i+1} + b_{i+1}$).

Therefore, after proving that our Greedy algorithm will not have any idle time and no inversion, our Greedy algorithm can generate the ideal schedule.