CSE 6140 / CX 4140 Assignment 1
due Sep 4, 2020 at 11:59pm on Canvas
Solutions

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)

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 ≤ n do
    j = 0;
    k = i;
    while k % 3 == 0 do
        k = k/3 ;
        j++ ;
    end
    i++;
    print i,j;
end
```

Solution:

1. (6 pts)
   (a) $f = (n + 1000)^4$, $g = n^4 - 3n^3$ $\quad$ $\mathbf{f \in \Theta(g)}$

(b) $f = \log_{1000} n$, $g = \log_2 n$ $\quad \mathbf{f \in \Theta(g)}$

(c) $f = n^{1000}$, $g = n^2$ $\quad \mathbf{f \in \Omega(g)}$

(d) $f = 2^n$, $g = n!$ $\quad \mathbf{f \in O(g)}$

(e) $f = n^n$, $g = n!$ $\quad \mathbf{f \in \Omega(g)}$

(f) $f = \log n!$, $g = n \log n$ $\quad \mathbf{f \in \Theta(g)}$

2. (5 pts)

Acceptable Solution (+3 pts): $O(n \log n)$. Outer loop goes from 1 to n, the inner loop takes $O(\log_3 n)$ time in the worst case.

Correct Solution (+4 pts): It turns out that because we only enter the loop when k is **divisible** by 3, the complexity reduces to $O(n)$. (Note: the complexity would be $O(n \log n)$ if the condition in the loop was $k/3 > 1$).

To get $O(n)$, we'll need to count exactly the number of operations that are taking place. One way to do that is for each number i, count the number of iterations the inner loop goes through. This proves difficult, however, since there's no closed form solution to counting this number.

Instead, we can tackle it from another perspective, where we count the number of times the inner loop is entered a first time, a second time, etc... For example, if $n = 20$, then i goes through 1, 2, ... 20. The inner loop is entered (at least) once by $i = 3, 6, 9, 12, 15, 18$ (i.e. $\lfloor 20/3 \rfloor = 6$ times). The inner loop is entered a second time by $i = 9, 18$ (i.e. $\lfloor 20/3^2 \rfloor = 2$ times). Finally, the inner loop is entered a third time by $i = 9$ (i.e. $\lfloor 20/3^3 \rfloor = 1$ time). Therefore, the number of times the inner loop is entered is

$$\lfloor 20/3 \rfloor + \lfloor 20/3^2 \rfloor + \lfloor 20/3^3 \rfloor = 9$$

More generally, for $i = 1 \ldots n$, the inner loop is entered

$$\begin{aligned}
T(n) &= \lfloor \frac{n}{3} \rfloor + \lfloor \frac{n}{3^2} \rfloor + \ldots + \lfloor \frac{n}{3^{\lfloor \log_3 n \rfloor}} \rfloor \\
&\leq \frac{n}{3} + \frac{n}{3^2} + \ldots + \frac{n}{3^{\lfloor \log_3 n \rfloor}} \\
&= n \sum_{k=1}^{\lfloor \log_3 n \rfloor} \left(\frac{1}{3} \right)^k
\end{aligned}$$

Let $r = \frac{1}{3}, p = \lfloor \log_3 n \rfloor$, then

$$\begin{aligned}
T(n) &= n \sum_{k=1}^{p} r^k && \text{geometric series} \\
&= n \left(\frac{r(1 - r^p)}{1 - r} \right) \\
&= n \left(\frac{\frac{1}{3}(1 - \frac{1}{3}^{\lfloor \log_3 n \rfloor})}{1 - \frac{1}{3}} \right) \\
&= \frac{n}{2} \left(1 - \frac{1}{3}^{\lfloor \log_3 n \rfloor} \right) \\
&\leq \frac{n}{2} \left(1 - \frac{1}{3}^{\log_3 n} \right) \\
&= \frac{n-1}{2} = O(n) && (1)
\end{aligned}$$

Output (+1 pt): The code prints pairs i, j where j is the number of times i has 3 as a factor.A sample run output (for $n = 30$) is presented in the next page to help you understand the process.

```
 i = [ 1 ]  =>  k = [ 1 ]    (k % 3 =  1 )

[STD OUT]  2 0

 i = [ 2 ]  =>  k = [ 2 ]    (k % 3 =  2 )

[STD OUT]  3 0

 i = [ 3 ]  =>  k = [ 3 ]    (k % 3 =  0 )

        ------- inside the inner loop -------
         >> inner-loop counter (j) =  1
         >> updated k              =  1.0

[STD OUT]  4 1

 i = [ 4 ]  =>  k = [ 4 ]    (k % 3 =  1 )

[STD OUT]  5 0

 i = [ 5 ]  =>  k = [ 5 ]    (k % 3 =  2 )

[STD OUT]  6 0

 i = [ 6 ]  =>  k = [ 6 ]    (k % 3 =  0 )

        ------- inside the inner loop -------
         >> inner-loop counter (j) =  1
         >> updated k              =  2.0

[STD OUT]  7 1

 i = [ 7 ]  =>  k = [ 7 ]    (k % 3 =  1 )

[STD OUT]  8 0

 i = [ 8 ]  =>  k = [ 8 ]    (k % 3 =  2 )

[STD OUT]  9 0

 i = [ 9 ]  =>  k = [ 9 ]    (k % 3 =  0 )

        ------- inside the inner loop -------
         >> inner-loop counter (j) =  1
         >> updated k              =  3.0

        ------- inside the inner loop -------
         >> inner-loop counter (j) =  2
         >> updated k              =  1.0

[STD OUT]  10 2

 i = [ 10 ]  =>  k = [ 10 ]    (k % 3 =  1 )

[STD OUT]  11 0

 i = [ 11 ]  =>  k = [ 11 ]    (k % 3 =  2 )

[STD OUT]  12 0

 i = [ 12 ]  =>  k = [ 12 ]    (k % 3 =  0 )

        ------- inside the inner loop -------
         >> inner-loop counter (j) =  1
         >> updated k              =  4.0

[STD OUT]  13 1

 i = [ 13 ]  =>  k = [ 13 ]    (k % 3 =  1 )

[STD OUT]  14 0

 i = [ 14 ]  =>  k = [ 14 ]    (k % 3 =  2 )

[STD OUT]  15 0

 i = [ 15 ]  =>  k = [ 15 ]    (k % 3 =  0 )

        ------- inside the inner loop -------
         >> inner-loop counter (j) =  1
         >> updated k              =  5.0

[STD OUT]  16 1

 i = [ 16 ]  =>  k = [ 16 ]    (k % 3 =  1 )

[STD OUT]  17 0

 i = [ 17 ]  =>  k = [ 17 ]    (k % 3 =  2 )

[STD OUT]  18 0

 i = [ 18 ]  =>  k = [ 18 ]    (k % 3 =  0 )

        ------- inside the inner loop -------
         >> inner-loop counter (j) =  1
         >> updated k              =  6.0

        ------- inside the inner loop -------
         >> inner-loop counter (j) =  2
         >> updated k              =  2.0

[STD OUT]  19 2

 i = [ 19 ]  =>  k = [ 19 ]    (k % 3 =  1 )

[STD OUT]  20 0

 i = [ 20 ]  =>  k = [ 20 ]    (k % 3 =  2 )

[STD OUT]  21 0

 i = [ 21 ]  =>  k = [ 21 ]    (k % 3 =  0 )

        ------- inside the inner loop -------
         >> inner-loop counter (j) =  1
         >> updated k              =  7.0

[STD OUT]  22 1

 i = [ 22 ]  =>  k = [ 22 ]    (k % 3 =  1 )

[STD OUT]  23 0

 i = [ 23 ]  =>  k = [ 23 ]    (k % 3 =  2 )

[STD OUT]  24 0

 i = [ 24 ]  =>  k = [ 24 ]    (k % 3 =  0 )

        ------- inside the inner loop -------
         >> inner-loop counter (j) =  1
         >> updated k              =  8.0

[STD OUT]  25 1

 i = [ 25 ]  =>  k = [ 25 ]    (k % 3 =  1 )

[STD OUT]  26 0

 i = [ 26 ]  =>  k = [ 26 ]    (k % 3 =  2 )

[STD OUT]  27 0

 i = [ 27 ]  =>  k = [ 27 ]    (k % 3 =  0 )

        ------- inside the inner loop -------
         >> inner-loop counter (j) =  1
         >> updated k              =  9.0

        ------- inside the inner loop -------
         >> inner-loop counter (j) =  2
         >> updated k              =  3.0

        ------- inside the inner loop -------
         >> inner-loop counter (j) =  3
         >> updated k              =  1.0

[STD OUT]  28 3

 i = [ 28 ]  =>  k = [ 28 ]    (k % 3 =  1 )

[STD OUT]  29 0

 i = [ 29 ]  =>  k = [ 29 ]    (k % 3 =  2 )

[STD OUT]  30 0

 i = [ 30 ]  =>  k = [ 30 ]    (k % 3 =  0 )

        ------- inside the inner loop -------
         >> inner-loop counter (j) =  1
         >> updated k              =  10.0

[STD OUT]  31 1
```

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

Please explain your algorithms clearly.

Solution:

1. The complexity in $O(\log(n))$ is a hint: One should use a binary search. Indeed, if we have $k \geq \lceil \log(n) \rceil$, we know the result for the floors whose indices range from i to j by dropping a box from the m-th floor where $m = \lfloor \frac{i+j}{2} \rfloor$ and then by iterating with floors i to $m-1$ if the box broke, and by iterating with floors m to j otherwise. The principle of the binary search guarantees that we will obtain the desired result (when $i = j$) and in at most $\lceil \log(n) \rceil$ steps, and, thus, after having broken at most $\lceil \log(n) \rceil$ boxes.

2. As we have only $k < \lceil \log(n) \rceil$ boxes, we cannot directly apply a binary search. We, however, will solve this problem in a simple way. We apply the binary search using $k-1$ boxes in order to narrow as much as possible the search interval around the desired floor. We then use the last box to scan the remaining interval floor by floor, from the lowest to the highest. After throwing $k-1$ boxes, if the target floor has not been identified, there are at most $n/2^{k-1}$ floors in the search interval, hence a worst-case complexity of $O(k + n/2^{k-1})$.

3. When $k = 2$ we do not want to have to test each floor one after the other, thereby ending up with a linear complexity. We, therefore, will adapt the idea of narrowing the search interval. We partition the set of floors into "slices" of $\sqrt{n}$ floors (assume that n is a square without loss of generality, or use ceil functions). Then we throw the first box from the first floor of each slice until it breaks, starting with the lowest slice. Then, we return to the last tested floor, m, from which the box was dropped but did not break. We then test one by one the floors using the second box. We start with floor $m+1$ and, in the worst case, we go up to the floor from which the first box broke. There are, by construction, $\sqrt{n}$ floors in that slice. Therefore, we have two series of tests, with $O(\sqrt{n})$ tests in each of them. Hence, the overall complexity is $O(\sqrt{n})$.

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.

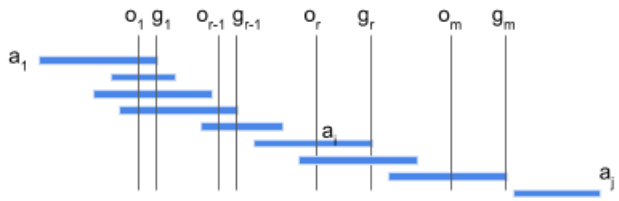
Solution:

For each given point p, we can compute in constant time the two points on l, p_l and p_r , such that $d(p, p_l) = d(p, p_r) = r$. Clearly, p is covered by some disk, if and only if there exists one disk whose center is between p_l and p_r. Thus, the problem is equivalent to the following problem: given n intervals on one line, compute a minimum number of points on the same line such that each interval contains at least one point.

Sort the intervals according to their right endpoints; also, maintain a list of points, initialized as an empty set. We scan the intervals one by one from left to right. If the left position of the current interval is on the left of the last point in the list, which implies that the last point in the list is inside the current interval, then we just skip this interval; otherwise, we add the rightmost endpoint of the current interval to the list as the last point. The running time of this algorithm is dominated by sorting, which is $O(n \log n)$.

We now prove that this greedy algorithm returns an optimal solution. We use a similar procedure as the proof for the interval scheduling problem used in class. Denote the intervals sorted by the right end (finish time) as $a_1, a_2, ..., a_n$, their left ends as $s_1, s_2, ..., s_n$, and their right ends as $f_1, f_2, ..., f_n$. The positions of the points given by greedy algorithm are denoted by $g_1, g_2, ..., g_k$ (sorted from left to right), and the points given by optimal is $o_1, o_2, ..., o_m$ (sorted from left to right).

First, we prove the "greedy stays ahead" argument, that is, we show that for all $r \leq m$, $g_r \geq o_r$ by induction. Base case: when $r = 1$, to cover a_1, $g_r \geq o_r$ by greedy choice. Inductive hypothesis: $g_{r-1} \geq o_{r-1}$, and we are to show that $g_r \geq o_r$. Let a_i be the next interval with $s_i > g_{r-1}$. In order to cover a_i, greedy creates another point $g_r = f_i$. Since $o_{r-1} \leq g_{r-1} < s_i$, o_{r-1} doesn't cover s_i. So o_r has to cover a_i which means $s_i \leq o_r \leq f_i$. Thus $o_r \leq g_r$.



Similar reasoning can be used to show that $k = m$. Suppose $k > m$. If greedy needs to add another point after g_m, it means there exists an interval a_j with $s_j > g_m$. Since $o_m \leq g_m$, $o_m < s_j$. So o_m does not cover a_j. Since o_m is the rightmost point from optimal, this contradicts that $o_1, o_2, ..., o_m$ is a feasible solution.

There can be other proofs which also show the optimality of the algorithm.

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 $\max(a_1, a_2, a_3)$)

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

Solutions:

The problem is: given a list of athletes, $[a_1, ..., a_n]$, where each athlete, a_i, has an associated swimming, running, and biking time: s_i, r_i, and b_i, respectively, what ordering of athletes minimizes the maximum finish time (time at which last athlete finishes), given that only one athlete can swim at a time, and that each athlete has to swim, then run, then bike?

Observe that no matter what the ordering of the athletes is, the last athlete will start the running + biking portion of the race at time $t = \sum_{i=1}^{n} s_i$, that is, after everyone is done swimming. From this we can look at the n^{th} and $(n-1)^{th}$ spots in the schedule and see that we should always put the fastest person at running and biking combined in the n^{th} spot, because if we put them in the $(n-1)^{th}$ spot our schedule can only be slower. With this intuition we can construct a greedy algorithm and associated exchange argument to prove its correctness!

Our greedy algorithm will be to sort the athletes in terms of $r_i + b_i$ (call this sum t_i) and them schedule them in order from slowest to fastest.

Our exchange argument is as follows:

- Assume that an optimal schedule exists. If it does not contain an inversion (i.e. an inversion is when athlete i is scheduled before athlete j although $t_i < t_j$), then we are done: the optimal is the greedy. (Technically, we need to demonstrate that all solutions without inversions have the same end time, in cases where the optimal time does not have a unique schedule, but points were not deducted if this case was not addressed.)
- If the optimal does contain an inversion, then we can assume (from lecture) that this leads to a point at which two consecutive athletes are inverted. Let these two consecutive inverted athletes in the optimal solution be i followed by j (with $t_i < t_j$).
- Let f_i be the finish time for athlete i, and f_j be the finish time for athlete j (ignoring all the swimming time that comes before them, because in both cases it will be the same). In this case, $f_i = s_i + t_i$ and $f_j = s_i + s_j + t_j$. Because $t_i < t_j$, the finish time of athlete j will always be larger: $f_j > f_i$.
- If we swap the positions of the two athletes (so that i goes after j), we get $f'_i = s_i + s_j + t_i$ and $f'_j = s_j + t_j$

  Now we can see that the finish times for i and j in this schedule are better than their larger finish time from the previous schedule (f_j), i.e. (1) $f'_i < f_j$ and (2) $f'_j < f_{i+1}$.

  (1) $f'_i = (s_i + s_j) + t_i, f_j = (s_i + s_j) + t_j$
  $t_i < t_j \Rightarrow f'_i < f_j$

  (2) $f'_j = (s_j) + t_j, f_j = s_i + (s_j) + t_j$
  $t_j < s_i + t_j \Rightarrow f'_j < f_j$
- Swapping the inverted pair thus decreases the total number of inversions and does not lead to a worse finish time than the original optimal time.
- We can iteratively swap all pairs of inversions and continuously decrease the overall number of inversions without loss of optimality, until we arrive at a solution without inversions (shown to give the same results as the greedy) that is at least as good as the given optimal solution. The total number of swaps is at most $n(n-1)/2$.
- $\Rightarrow$ Thus, greedy is an optimal solution (or, greedy is at least as good as anything you can come up with).