Homework 1

This assignment reviews big-O notation, basic problem solving using loops, loop optimization, and counterexample finding.

When asked to submit graphs, drawings, or figures, you may do one of the following:

- Use Tikz or a related latex pacakge to create the graphics using LaTeX. See the LateX guide for examples of this. This is not as hard as it looks.
- Use photoshop / ms paint / etc other drawing software to draw an image and insert it into your pdf using the includegraphics command.
- Draw this on paper/whiteboard, take a picture, and include the image in your pdf using the includegraphics command.

Whichever option you choose, your images/drawings should be readable.

0. (0 points) (NOT OPTIONAL) Who did you work with on this homework? What sources did you consult? How long did this assignment take?

   Solution: I consulted my friend Yushiro and we both brainstormed ideas and came up with counterexamples. I also utilized the TA (Tien) to help me bounce around my ideas. Other than that, I used chatbot LLMs like Claude to help me format LaTeX and draw graphs. Also worth mentioning that I talked to my roommate Darshan about the bonus questions and found we had similar principles but different code. This assignment took me around 6–7 total hours to complete.

1. (5 points) Attend office hours for at least 2 minutes before this homework is due. My office is in Jepson 224. On busier days (Wednesday and Thursday evenings), I will likely move to the common area in front of Jepson 212.

   Office hours are tentatively scheduled for:
   M: 3-4:30 Tu: 2:00-6:00 Th: 2:30-4:30 F: 12-4

   When did you go to office hours?

   Solution: I went to office today at 5:50 PM right after our Wednesday's class. (Apologetic office hours). Other classmates who were present were Darshan, Meklete, Jordan, Tina.

2. (15 points) (Asymptotic practice) For each pair of functions below, decide which one of the following is true: (1) $f(n) = o(g(n))$, (2) $f(n) = \Theta(g(n))$, or (3) $g(n) = o(f(n))$. In all cases, show your work when calculating the appropriate limit. You may need to review some algebra / log and exponent rules / precalculus / calculus.

   (a) (Example) $f(n) = n^2$, $g(n) = n^3 - 9$.

   Solution:

   $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^2}{n^3 - 9} = \lim_{n\to\infty} \frac{1}{n - \frac{9}{n^2}} = \lim_{n\to\infty} \frac{1}{n} = 0$$

   Thus (1) $f(n) = o(g(n))$

   (b) (3 points) $f(n) = 3^{n+2}$, $g(n) = 2 * 3^n - 9$.

   Solution:

   $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3^{n+2}}{2 * 3^n - 9} = \lim_{n\to\infty} \frac{9 * 3^n}{2 * 3^n - 9} = \lim_{n\to\infty} \frac{\frac{9*3^n}{3^n}}{\frac{2*3^n}{3^n} - \frac{9}{3^n}} = \lim_{n\to\infty} \frac{9}{2 - \frac{9}{3^n}} = \frac{9}{2}$$

   Thus (2) $f(n) = \Theta(g(n))$

   (c) (3 points) $f(n) = 5^{2n+2}$, $g(n) = 1000000 * 5^n$.

   Solution:

   $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{5^{2n+2}}{1000000 * 5^n} = \lim_{n\to\infty} \frac{25 * 5^{2n}}{1000000 * 5^n} = \lim_{n\to\infty} \frac{25 * 5^n}{1000000} = \infty$$

   Thus (3) $g(n) = o(f(n))$

   (d) (3 points) $f(n) = 2^n + 3^n$, $g(n) = 4^n$.

   Solution:

   $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{2^n + 3^n}{4^n} = \lim_{n\to\infty} (\frac{1}{2})^n + (\frac{3}{4})^n = 0$$

   Thus (1) $f(n) = o(g(n))$

   (e) (3 points) $f(n) = \log_2(n^2)$, $g(n) = \log_3(n^4)$.

   Solution:

   $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\log_2(n^2)}{\log_3(n^4)} = \lim_{n\to\infty} \frac{2 * \log_2(n)}{4 * \log_3(n)} = \lim_{n\to\infty} \frac{2 * \frac{\cancel{\log_3(n)}}{\log_3(2)}}{4 * \frac{\cancel{\log_3(n)}}{\cancel{\log_3(3)}}1} = \frac{1}{2\log_3(2)}$$

   Thus (2) $f(n) = \Theta(g(n))$

   (f) (3 points) $f(n) = n^3$, $g(n) = 4^{(\log_2 n)}$

Solution:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^3}{4^{(\log_2 n)}} = \lim_{n\to\infty} \frac{n^3}{(2)^{2(\log_2 n)}} = \lim_{n\to\infty} \frac{n^3}{2^{\log_2(n^2)}} = \lim_{n\to\infty} \frac{n^3}{n^2} = \lim_{n\to\infty} n = \infty$$

Thus (3) $g(n) = o(f(n))$

3. (Example) (Polynomials) You are given an input integer array $P = [a_0, a_1, a_2, \ldots a_n]$ of length $n+1$ that stores the coefficients of an n degree polynomial p, and an integer value x. Your goal is to compute the value of $p(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \ldots a_nx^n$ for an input x. You may assume that all elementary integer operations (+,-,*,/) take $O(1)$ time.

   Consider the following algorithm, and convince yourself that it correctly evaluates the polynomial.

```
def evaluate(P,x):
  ans = 0
  for i in range(n+1): //0...n
    term = P[i]
    for j in range(i):
      term = term * x
    ans = ans + term
  return ans
```

   (a) What is the runtime of this algorithm? Choose between stating the answer as big-Θ or big-O as appropriate, choosing the *most* restrictive answer when possible.

   Solution: This takes $\Theta(n^2)$ time.

   (b) Find a faster algorithm for this problem (that still uses $O(1)$ space!). You may give pseudocode (copy/modify the above latex), use the verbatim environment (see examples in the latex guide), or submit real code in a separate file.

   Hint: Try to execute the above algorithm by hand on paper for an array of size 5. Identify any redundant or repetitive calculations.

   Solution: We can save time by not starting from scratch each time when computing the powers of x.

   function PolyEval2(Array $A[0 \ldots n]$, x)
   $ans = 0$
   $xpower = 1$
   for $i = 0$ to n **do**
   $xpower = xpower * x$
   $ans = ans + A[i] * xpower$
   return ans

   Note that this way requires two multiplications and an addition inside the for loop.

   A asymptotically equivalent, less intuitive, and really slick way to solve this problem needs only one multiplication and addition inside the for loop. Work out for yourself why the below psuedocode also produces the right answer.

   function PolyEval2(Array $A[0 \ldots n]$, x)
   $ans = A[n]$
   for $i = n - 1$ to 0 step -1 **do**
   $ans* = xpower$
   $ans+ = A[i]$
   return ans

4. (20 points) (Array Intersection)

   You have two very large arrays A of length a and B of length b, each with unsorted, distinct integers *with no repeated elements*. In this case, a is much larger than b. Your goal is to write an algorithm that can find the intersection of A and B - the set of elements present in both A and B.

   Example input: A = [3,8,10,2,4,9], B = [1,2,3,4,7]

   Expected output: 2,3,4

   Here's one of the simplest ways to implement this.

```
def Intersection(A, B):
  intersection = []
  for x in A:
    for y in B:
      if x == y:
        intersection.append(x)
        break #No need to continue this loop
```

   The runtime of this is $O(mn)$. We will try to do better than this, in both average and worst case situations. You should be reminded of the following facts. Not all of them may be useful for this problem.

   - Built-in or standard library functions and syntax ("in", "contains", "sort", "sum", "**", "union","intersection",etc.) are usually not $O(1)$ operations.
   - Array accesses to a known position in an array takes $O(1)$ time.
   - Iterating through an array of length n takes $\theta(n)$ time.
   - You can run binary search in a *sorted* list of n integers in $O(\log n)$ time using $O(1)$ space.
   - You can sort a list of n unsorted integers in $O(nlogn)$ time using only $O(1)$ extra space.
   - You can find the median in an unsorted list of integers in $O(n)$ time and $O(1)$ space.
   - In a balanced binary search tree of n elements, insertion, removal, and queries take $O(\log n)$ time and the tree takes $O(n)$ space.
   - In an unordered set/hash map/hash table of n elements, insertion, removal, and contains queries take $O(1)$ time in the average case, but $O(n)$ time in the worst case. The hash table takes $O(n)$ space.

This problem is continued on the next page:

For the below problems, you should give concrete pseudocode, like I did above, or actual code. You may submit high level pseudocode, where you use sentences in your pseudocode, provided those sentences aren't vague and could be easily turned into code by a CS 221 student. Probably don't do this unless you are comfortable being clear and not vague when writing proofs. If you aren't, stick to concrete pseudocode (or better yet, working code).

Your algorithms below should optimize for time as a first priority and extra space allocated as a second priority. Remember that $a >> b$.

Hint: For both of these, you can do better than sorting both arrays and doing the two pointed intersection check discussed in class.

(b) (10 points) Describe and analyze an algorithm that has, asmyptotically, the best *worst case* runtime for computing the intersection. What is the space usage of your algorithm?

Solution: Best Worst-Case Runtime Algorithm (that I found):

Approach: The intuition is that you can use binary search to search for an element in an array, which is faster than linear search, but requires the array to be sorted. So sort the smaller array B and use binary search to search for each element of A in B. (Each element in A is a target for the binary search function). Let b be the length of the much smaller array B, and a be the length of the much bigger array A.

Python Implementation:

```
def Intersection(A, B):
    intersection = []
    B.sort() # O(b*log(b)) time

    for x in A: # O(a) iterations
        index = bin_search(B, x) # O(log(b)) per search
        if index != -1:
            intersection.append(B[index])
    return intersection
```

Complexity Analysis:

- **Time Complexity:** $O(b \log b + a \log b) = O(a \log b)$
  - Sorting array B: $O(b \log b)$
  - Binary search for each of a elements: $O(a \log b)$
  - Since $a \gg b$, we have $b \log b < a \log b$, so total is $O(a \log b)$
- **Space Complexity:** $O(1)$ extra space (sorting in-place)
  - Output array size is $O(\min(a, b)) = O(b)$, but this is required output
  - Sorting B can be done in-place with $O(1)$ extra space

(c) (10 points) Give an algorithm that has, asmyptotically, the best *average case* runtime. What is the space usage of your algorithm?

Solution: Best Average-Case Runtime Algorithm (that I found):

Approach: The intuition is that in average-case, looking up in a hashset is $\theta(1)$. So just make a hashset of elements in array B and iterate through elements of array A and see if the hashset contains each element of A.

Python Implementation:

```
def Intersection(A,B):
    intersection = []
    hashset_b = set(B)
    for a in A:
        if a in hashset_b:
            intersection.append(a)
    return intersection
```

Complexity Analysis:

- **Time Complexity:** $O(b + a) = O(a)$ since $a \gg b$
  - Creating a hashset for B: $O(b)$
  - Iterating through elements of A: $O(a)$
  - Each look up in the hashset of B: $\theta(1)$
- **Space Complexity:** $O(b)$ extra space (hashset)
  - **Note:** I decided to hash B rather than A because either way the time complexity is still $O(a)$, but B is way smaller, so it's the most logical choice.

5. (20 points) A powerful concept in computer science is the idea of *precomputation*, computation that you do before the "main" part of the algorithm and store in a convenient format to make the job of the main part easier.

   For example, say you are given an array A. The "main" algorithm that runs on A is going to need to repeatedly compute the sum of all the elements in A between indices i and j for various i and j.

   If we used the following naive algorithm each time:

   ```
   s = 0
   for x in range(i, j+1):
     s += A[x]
   return s
   ```

   this would take $O(n)$ time per query.

   If the "main" algorithm makes lots of queries, this can get expensive.

   Figure out a way to do precomputation to make this faster. Your precomputation step should take $O(n)$ time and $O(n)$ space, and allow you to answer each query in $O(1)$ time after the precomputaiton.

   Hint: First figure out how to solve this problem for the special case queries where $i = 0$.

Solution:

Python Implementation:

```
def build_prefix_sum(arr):
    prefix_sum = [0] * len(arr)
    prefix_sum[0] = arr[0]

    for k in range(1, len(arr)):
        prefix_sum[k] = arr[k] + prefix_sum[k-1]

    return prefix_sum


def range_sum(prefix_sum, i, j):
    if i == 0:
        return prefix_sum[j]
    else:
        return prefix_sum[j] - prefix_sum[i-1]
```

6. (20 points) (Counter Examples)

   You run a daycare in charge of n children C. There are m nannies N present who can take care of the children. For each child i, there is a non-empty subset $C_i \subseteq N$ of the nannies, and that child is *happy* if at least one of their N_i nannies is working, otherwise that child will cry.

   You goal is to hire the smallest set of nannies such that all the children are happy.

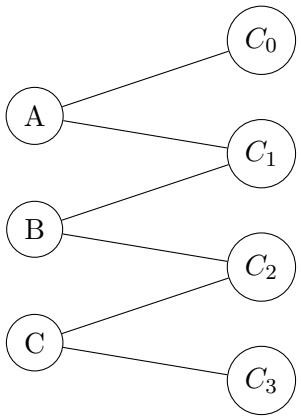
   For example, If $n = 4$, $C_0 = \{A, B, C\}$, $C_1 = \{C, D, E\}$, $C_2 = \{B, C, D, F\}$, $C_3 = \{B, E, F, G\}$. Then you can get away with hiring only 2 nannies: C,F is one possible solution.

   We propose the following sensible seeming algorithm, which can be described in English as: "Repeatedly add the nanny that will make the most unhappy children happy until all the children are happy". Described in high level pseudocode, this looks like:

```
def chooseNannies(Nannies,Children):
  solution = {}
  UnhappyChildren = Children
  while(UnhappyChildren isn't empty):
    bestNanny = the nanny that makes the most children in UnhappyChildren happy
    add bestNanny to Solution
    remove all the children that bestNanny makes happy from UnhappyChildren
  return solution
```

   Show that this algorithm is not always optimal by finding a counter example and describing the true optimal solution in your counter example versus what this algorithm might return.

Solution: If $n = 4$, $C_0 = \{A\}$, $C_1 = \{A, B\}$, $C_2 = \{B, C\}$, $C_3 = \{C\}$



Greedy algorithm's solution: The greedy algorithm might choose Nanny A or Nanny B, or Nanny C since they all have degree of 2. Let's say the algorithm chooses Nanny B, first. Nanny B makes children C_1 & C_2 happy. This leaves child C_0 with Nanny A and child C_3 with Nanny C. This means you will have to hire all **3 Nannies** to make all the children happy.

Optimal solution: Hire only Nannies A & C (**2 Nannies**): A makes C_0 and C_1 happy, C makes C_2 and C_3 happy. No need for an extra Nanny B.

7. (20 points) You have n bags of candy, and m children who are to receive these bags. Each bag i has $c[i]$ pieces of candy. You want to distribute the bags of candy to the children, but you cannot split a bag between multiple children.

   To distribute things as fairly as possible, your goal is to maximize the *fewest* pieces of candy received by any of the children. For example, if I have three children and five bags with $[3, 4, 2, 2, 3]$, then one allocation could look like: $[3, 4], [2, 2], [3]$ so each child gets at least 3 pieces, but a better allocation would look like $[4], [2, 3], [2, 3]$ so each child gets at least 4 pieces.

   One pretty reasonable algorithm works like this: sort the bags by amount of candy. Starting with the biggest bag, assign the bags one at a time to the child with currently the smallest amount of candy. (This is essentially the algorithm that many routers use to equally distribute traffic across multiple servers.)

   Show that this algorithm is not always optimal by finding a counter example and describing the true optimal solution in your counter example versus what this algorithm might return.

Solution: Consider the # of children = 2 and there are 7 bags of candy: [4,3,4,3,3,4,3].

Greedy algorithm's solution: The greedy algorithm will first sort, and distribute the candies as follows. Kid A gets 4, Kid B gets 4, then Kid A gets 4 again, so Kid B who has less gets 3. Kid B still has less than Kid A at a running sum of each at 7 and 8 in respective order, so Kid B gets another 3. Now that Kid A has less than Kid B, Kid A gets 3. Now kid B has less than kid A, so kid B gets 3. We have now exhausted all of our candy and they are all distributed between two kids. Ultimately, **Kid A gets 11 & Kid B gets 13.** So Kid A leaves with only 11 candies.

But we can do better!

Optimal solution: Give Kid A three of 4's and give Kid B four of 3's. So Kid A has [4,4,4] & Kid B has [3,3,3,3]. Now this is a better solution because not only this we maximized the *fewest* pices of candy received by any of the children, they were also distributed very fairly since both children got the same amount of candies. **Kid A gets 12 & Kid B gets 12.**

8. (10 points) (BONUS) Consider the following code. Input: array A and B of integers, both of length n. B is a sorted list of integers all between 0 and n, possibly with repeated values.

```
def WonkySum(A,B):
  s = 0
  for i in range(len(A)):
    for j in range(B[i]):
      s += A[j]
  return s
```

Example: If $A = [3, 7, -2, 9]$ and $B = [0, 3, 4, 4]$, the result would be: $0 + 8 + 17 + 17 = 42$.

Note that the inner loop goes to i instead of to n. The runtime of this is $O(n^2)$, because the elements of $B[i]$ are at most $O(n)$.

Figure out a way to compute this in $O(n)$ time with only $O(1)$ extra space! There are at least three different ways to do this. The below hints are for one of those ways.

Hint: Try running this algorithm on paper for n 6 for some inputs of B. What work is happening unnecessarily / repeatedly?

Hint2: First try to find a way to do this in $O(n)$ time with $O(n)$ space using a precomputation. Then, avoid doing the precomputation to get it to $O(1)$ space.

Hint3: Nested loops aren't always quadratic runtime.

Solution:

Python Implementation:

```
def WonkySum(A, B):
    total_sum = 0
    prev_sum = 0
    prev_b = 0
    for b in B:
        for index in range(prev_b, b):
            prev_sum += A[index]
        total_sum += prev_sum
        prev_b = b
    return total_sum
```

Complexity Analysis:

- **Time:** $O(n)$ - The inner loop collectively iterates over each element of A at most once since B is sorted and *prev_b* only increases.
- **Space:** $O(1)$ - Only constant extra space for variables.