Name: Jungbin Choi
Due September 8, 2023

COMP 550-001
Homework 1 (24 points)

---

**Notes.** Collaboration is allowed, and you may use any resources you'd like. However, you should write and submit your solutions individually, and each submission will be graded individually. In your submission, you should list your collaborators (including course staff) and cite every resource you use. For each problem, I encourage you to attempt it for at least 30 minutes before consulting any people or resources.

Your submission should be a single PDF file. I recommend using Overleaf; a template is on Canvas. If your answers are not clear, you might not get full credit. For this assignment, you should describe your algorithms in pseudocode similar to the examples given in the lectures/notes. I recommend also including a brief description in full sentences.

Throughout this course, you may not use hash tables (dictionaries) unless explicitly stated in the problem statement, because this course (and theoretical computer science in general) is about worst-case analysis.

**Problem 1 (3 points).** Prove that the $O(n^2)$-time Brute-Force algorithm for Two Sum given in lecture is correct. For your reference, it is given below:

```
ALG(A, t):
  for i = 1, ..., n-1:
    for j = i+1, ..., n:
      if A[i] + A[j] = t:
        return (i,j)
  return 0
```

**Solution.** Let $(i*, j*)$ equal the solution (if it exists) of the algorithm. Once the index $i$ reaches $i*$, all previous values have been checked with the values that come after it, therefore the algorithm now only needs to check the values after $i*$. This is exactly what happens when $j*$ start from the value with index $i*+1$ until it reaches index $n$ (i.e. the end). There, the solution value of $j*$ will be found (if it exists) and the algorithm will return the correct solution of $(i*, j*)$. If there is no solution, then the algorithm will continue on through the algorithm until the for loops reach the end and return a value of 0.

**Problem 2 (3 points).** Let $A$ be an array of $n$ integers. Our goal is to reverse the order of elements in $A$. For example, if $A = [13, 4, 7]$, the output should be $A = [7, 4, 13]$. For this problem, you may not create a new array (i.e., your algorithm should be "in place").

a. (2p) Give an $O(n)$-time algorithm that solves this problem.

b. (1p) Analyze the running time of your algorithm for (b).

**Solution.**

a.
```
ALG(A):
    for i = 1, ..., floor(n/2):
        temp = A[i]
        A[i] = A[n + 1 - i]
        A[n + 1 - i] = temp
    return A
```

b. The for loop will have a RT of $O(0.5n)$ due to the loop only needing to access half of the array. Within the loop: 3 assignments, 3 indexing, 4 arithmetic $\rightarrow$ it will result in the total number of primitive operations of $10x$, where $x =$ number of total iterations (i.e. $0.5n$). Therefore, the overall total for the primitive operations will be $5n$, which means the RT will be $O(5n)$, which simplifies to $O(n)$.

**Problem 3 (6 points).** Let $A$ be an array of $n$ integers, and $k$ be a positive integer where $k \leq n$. Our goal is to return an index $i \leq n - k + 1$ such that the sum of $A[i : i + k - 1]$ (i.e., the length-$k$ subarray of $A$ starting at index $i$) is maximized.

a. (2p) Give an $O(nk)$-time, brute-force algorithm that solves this problem and analyze its running time.

b. (2p) Give an $O(n)$-time algorithm that solves this problem.

c. (1p) Prove that your algorithm for (b) is correct.

d. (1p) Analyze the running time of your algorithm for (b).

**Solution.**

a.
```
ALG(A, k):
    m = 0
    result = 0
    for i = 1, ..., (n - k + 1):
        sum = 0
        for j = i, ..., (i + k - 1):
            sum += A[j]
        if sum > m:
            m = sum
            result = i
    return result
```

The first two assignments of $m$ and $result$ will be $O(1)$ as it will be constant no matter what the size of the array is. Next, the for loop for $i$ will be $O(n - k + 1)$, which simplifies down to $O(n)$. Per each loop in the $i$ loop, there will be a for loop $j$ which lasts $O(k)$. Everything else inside the $i$ and $j$ loops will have $O(1)$ constant time, so the total RT will be $O(n) * O(k)$, which is equal to $O(nk)$.

b.
```
ALG(A, k):
    beg = i
    end = n
    while True: //indefinite loop
        if (end - beg + 1) == k:
            return beg
        elif A[beg] >= A[end]:
            end -= 1
        else: //A[beg] < A[end]
            beg += 1
```

c. This algorithm will perform a sort of "survival of the fittest" approach to the values. Starting with the sum of the entire array, the algorithm will chip away at the total sum by removing the lower values from the extreme ends of the subarray (which starts as the whole array until the beginning or ending indices are moved). By only removing the smaller values until the subarray matches length $k$, the sum of the subarray will be at its maximum as only the values that contribute the most to the sum will be left.

d. The for loop will last $O(n)$ as the sum will be calculated by going that the array once. Then, the assignments for *beg* and *end* are $O(1)$ in total. Next, the while loop will take $n-k$ iterations (with each iteration only being $O(1)$ due to all of the operations inside being constant time) to reach the subarray size of $k$. Lastly, summing all the RT up, the total is $O(2n-k+2)$. Since $k$ is always $k \leq n$, the final simplified RT is $O(n)$.

**Problem 4 (6 points).** Let $A$ be an array of $m$ distinct integers, and let $B$ be an array of $n$ distinct integers where $m \leq n$. Our goal is to print every integer that appears in both $A$ and $B$. (We can print these numbers in any order).

a. (2p) Give an $O(mn)$-time, brute-force algorithm that solves this problem and analyze its running time.

b. (2p) Give an $O(n \log n)$-time algorithm that solves this problem.

c. (1p) Prove that your algorithm for (b) is correct.

d. (1p) Analyze the running time of your algorithm for (b).

**Solution.**

a.
```
ALG(A, B):
    for i = 1, ..., m:
        for j = 1, ..., n:
            if A[i] == B[j]:
                print(A[i])
```

The outer for loop will take $O(m)$ to loop through array $A$. The nested for loop will take $O(n)$ to loop through array $B$ per iteration of the outer for loop. The comparison, indexing, and printing will all be constant time $O(1)$ per iteration of the nested loop. Thus, the RT will be constant time per nested loop iteration per outer loop iteration. That would be $O(1) * O(n) * O(m)$, which equals $O(mn)$.

b.
```
ALG(A, B):
    sort(B)
    for i = 1, ..., m:
        if binary_search(B, A[i]): //if search was a success
            print(A[i])
```

c. First, we must sort array $B$ in order to binary search $B$. Assuming the sorting was done correctly, array $B$ will now be in numerical order. Using the binary search algorithm on array $B$ while setting the target value to be each value in array $A$, the value in $A$ will be either found or not found in $B$ (assuming the binary search was also correctly implemented). If found, the value will be printed; if not, then there would not be an output.

d. The sorting algorithm used will be a comparison-based sort where the RT of that sort is at most $O(n \log n)$. Next, the for loop will take $O(m)$ time to loop through array $A$. Per each iteration, the binary search function will take at most $O(\log n)$ RT to finish (array size decreases $\frac{n}{2^k}$ per $k$ iterations $\rightarrow \frac{n}{2^k} \leq 1$ when $k = \log_2 n \rightarrow n \leq 2^k \rightarrow \log_2 n \leq k$). Thus, the for loop will take a total of $O(m \log n)$ RT to finish. In total, the entire algorithm will take an RT of $O((n \log n) + (m \log n))$. This can be simplified due to the fact that $m \leq n$: the biggest term will be $O(n \log n)$ or if $m = n$, $O(2n \log n)$ will be simplified to $O(n \log n)$.

**Problem 5 (6 points).** Let $A$ be an array of $n$ integers sorted in non-decreasing order. Our goal is to return an array $B$ that contains the elements of $A$ squared; furthermore, $B$ should be sorted in non-decreasing order. For example, if $A = [-3, 1, 1, 5]$, then $B = [1, 1, 9, 25]$.

a. (2p) Give an $O(n \log n)$-time, brute-force algorithm that solves this problem and analyze its running time.

b. (2p) Give an $O(n)$-time algorithm that solves this problem.

c. (1p) Prove that your algorithm for (b) is correct.

d. (1p) Analyze the running time of your algorithm for (b).

**Solution.**

a.
```
ALG(A):
    B = int[]
    for i = 1, ..., n:
        B[i] = A[i] * A[i]
    sort(B)
    return B
```

The initialization of B only takes $O(1)$ RT. The for loop will take $O(n)$ RT to loop through array $A$. Per iteration, the assignment, multiplication, and indexing will only take $O(1)$ RT, thus the entire for loop will take $O(n)$. Next, the sorting algorithm will take at most $O(n \log n)$ RT to finish, if using a comparison-based sort such as merge. Finally, the total RT will be the sum of all the RT, which simplifies down to $O(n \log n)$ due to that term being the largest.

b.

```
ALG(A):
    i, j, k = 1, n, n
    B = int[n]
    while i != j:
        if (A[i] ** 2) >= (A[j] ** 2):
            B[k] = A[i] ** 2
            k -= 1
            i += 1
        else: //(A[i] ** 2) < (A[j] ** 2)
            B[k] = A[j] ** 2
            k -= 1
            j -= 1
    return B
```

c. Similar to the "Two-Pointer" method, the indices $i$ and $j$ will begin in the opposing ends. Since array $A$ is sorted, the biggest values (when squared) will be at the ends as well. Therefore, as the pointers $i$ and $j$ move closer together after checking which value is bigger and moving the index up or down, the squared values will also decrease in order.

d. The initialization of all the variables will be $O(1)$ constant time. The while loop will end when $i = j$, and since $i$ and $j$ start at distances $n - 1$ and get closer by 1, the while loop will iterate $n - 1$ times, thus a RT of $O(n - 1)$ or simplified as $O(n)$. Per iteration, all of the comparisons, assignments, arithmetic, and indexing will have a RT of $O(1)$ as it doesn't depend on the size of the array. Therefore, the entire while loop will have a RT of $O(n)$. Thus, when summing all the RT, the final RT will be $O(n)$

Collaborator: Sahra Arifi