Introduction to algorithms - Homework 1

Yorguin José Mantilla Ramos

Matricule 20253616

1 Asymptotic notation (25 points)

1.1 Question 1

(10 points) Using the definitions of O, Ω and Θ seen in class (without using limits), prove or disprove the following statements:

1.1.1 $\lfloor \frac{n}{10} \rfloor \in O(\sqrt{n})$

Answer:
By definition, $\lfloor n/10 \rfloor \leq n/10$ since the floor function just removes the fractional part, choosing the closest lower integer. In fact, we can refine this bound to make it from below rather than from above:

$$\frac{n}{10} - 1 \leq \lfloor n/10 \rfloor.$$

If we prove that $\frac{n}{10} - 1 \notin O(\sqrt{n})$, then it must also hold for $\lfloor n/10 \rfloor$ because of transitivity.
Assuming $\frac{n}{10} - 1 \in O(\sqrt{n})$, this means there exist constants $C > 0$ and $n_0 \in \mathbf{N}$ such that:

$$\frac{n}{10} - 1 \leq C\sqrt{n}, \quad \forall n \geq n_0.$$

Rearranging:

$$\frac{n}{10} - 1 \leq C\sqrt{n}.$$

Dividing by $\sqrt{n}$:

$$\frac{n}{10\sqrt{n}} - \frac{1}{\sqrt{n}} \leq C.$$

Simplifying:

$$\frac{\sqrt{n}}{10} - \frac{1}{\sqrt{n}} \leq C.$$

Notice that as n grows, the term $\frac{1}{\sqrt{n}}$ approaches 0, while $\frac{\sqrt{n}}{10}$ grows unbounded. Thus, the expression $\frac{\sqrt{n}}{10} - \frac{1}{\sqrt{n}}$ is eventually non-decreasing.
Since $\frac{\sqrt{n}}{10} - \frac{1}{\sqrt{n}}$ cannot be bounded by a constant when it reaches the eventually non-decreasing point, we reach a contradiction. Thus, we conclude:

$$\frac{n}{10} - 1 \notin O(\sqrt{n}) \quad \Rightarrow \quad \lfloor n/10 \rfloor \notin O(\sqrt{n}).$$

1.1.2 $n\sqrt{n}\log(n!) \in O(n^3 \log(n))$

Answer:
We want to determine whether:

$$n\sqrt{n}\log(n!) \in O(n^3 \log(n)).$$

Assuming this for some constants $C > 0$ and $n_0 \in \mathbf{N}$ such that $n \geq n_0$:

$$n\sqrt{n}\log(n!) \leq Cn^3 \log(n).$$

Note that we can upper-bound $log(n!)$:

$$\log(n!) = \log(1 \cdot 2 \cdot 3 \cdots n).$$

Since each term in the product is at most n, we can bound the sum:

$$\log(n!) \leq \log(n^n).$$

$$\log(n!) \leq n\log(n).$$

As n is positive, we can transform this inequality to:

$$n\log(n!) \leq n(n\log(n)).$$

$$n\log(n!) \leq n^2\log(n).$$

Note that $\sqrt{n} \leq n$ for positive n, and both sides of $n\log(n!) \leq n^2\log(n)$ involve terms that are all positive when $(n \geq 1)$, thus we can multiply both of this inequalities while preserving the inequality itself, obtaining:

$$n\sqrt{n}\log(n!) \leq n^3\log(n).$$

Remembering the Big-O definition, $n\sqrt{n}\log(n!)$ is in $O(n^3\log(n)$, if there exist positive constants C and n_0 such that for all $n \geq n_0$:

$$n\sqrt{n}\log(n!) \leq C \cdot n^3\log(n).$$

From the earlier inequality, we can see that a valid constants for the definition are $C = 1$ and $n_0 = 1$. We conclude then:

$$n\sqrt{n}\log(n!) \in O(n^3\log(n)).$$

1.2 Question 2

(10 points) Using the limit rule, determine the relative order (O, Ω or Θ) of the following functions:

1.2.1 $f(n) = \frac{n}{\sqrt[3]{n}}, \quad g(n) = \ln\sqrt{n}$

Answer:
Using the limit rule, let:

$$f(n) = \frac{n}{\sqrt[3]{n}}, \quad g(n) = \ln(\sqrt{n}).$$

Let express the functions in exponent form and simplify:

$$f(n) = \frac{n}{n^{1/3}} = n^{1-1/3} = n^{2/3}.$$

$$g(n) = \ln(\sqrt{n}) = \ln(n^{1/2}) = \frac{1}{2}\ln(n).$$

We can now evaluate the limit:

$$L = \lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{\frac{1}{1}n^{2/3}}{\frac{1}{2}\ln n} = \frac{2n^{2/3}}{\ln n}.$$

Applying L'Hôpital's rule (since both numerator and denominator tend to infinity):

$$L = \lim_{n\to\infty}\frac{\frac{d}{dn}(2n^{2/3})}{\frac{d}{dn}(\ln n)}.$$

We know the derivatives are:

$$\frac{d}{dn}(2n^{2/3}) = 2 \cdot \frac{2}{3}n^{-1/3} = \frac{4}{3}n^{-1/3},$$

$$\frac{d}{dn}(\ln n) = \frac{1}{n}.$$

Thus,

$$L = \lim_{n\to\infty}\frac{\frac{4}{3}n^{-1/3}}{\frac{1}{n}} = \lim_{n\to\infty}\frac{4}{3}n^{1-1/3} = \lim_{n\to\infty}\frac{4}{3}n^{2/3}.$$

Since $n^{2/3} \to \infty$, then $L \to +\infty$. This implies that: $\ln\sqrt{n} \in O\left(\frac{n}{\sqrt[3]{n}}\right)$ and $\frac{n}{\sqrt[3]{n}} \notin O\left(\ln\sqrt{n}\right)$.

1.2.2 $f(n) = 2^{bn}, \quad g(n) = 3^n$ **where** $b \in \mathbf{N}^{\geq 2}$

Answer:
Let $f(n) = 2^{bn}$ and $g(n) = 3^n$ for $b \in \mathbf{N}^{\geq 2}$, using the limit rule:

$$L = \lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{2^{bn}}{3^n}.$$

Rewriting the exponentials in terms of base e:

$$L = \lim_{n\to\infty} \frac{e^{(\ln 2)bn}}{e^{(\ln 3)n}}.$$

$$L = \lim_{n\to\infty} e^{(\ln 2)bn-(\ln 3)n}.$$

Lets define:

$$\Delta = (\ln 2)bn - (\ln 3)n = n(\ln 2 \cdot b - \ln 3).$$

The behavior of L depends on the sign of Δ:

- If $\Delta > 0$, then $L \to +\infty$.
- If $\Delta < 0$, then $L \to 0$.
- If $\Delta = 0$, then $L \to 1$

Since $b \in \mathbf{N}^{\geq 2}$, the smallest possible integer value of b is 2. Checking for the worst case $b = 2$, as all others possibilities with $b > 2$ will have a **larger and positive difference**:

$$2\log 2 - \log 3 \approx 0.28 > 0.$$

Since $\Delta > 0$ for the worst case with $b = 2$, we conclude that $L \to \infty$, meaning:

$$3^n \in O(2^{bn}) \quad \text{for } b \in \mathbf{N}^{\geq 2}.$$

$$2^{bn} \notin O(3^n) \quad \text{for } b \in \mathbf{N}^{\geq 2}.$$

1.3 Question 3

(5 points) Show that the following function is E.N.D.: $f(n) = 2n^2 - n\sin(n)$
Answer:
We know that:

$$-1 \leq \sin(n) \leq 1.$$

Multiplying by $-n$ reverses the inequality:

$$-n \leq -n\sin(n) \leq n.$$

Since we are given: $f(n) = 2n^2 - n\sin(n)$, we add $2n^2$:

$$2n^2 - n \leq 2n^2 - n\sin(n) \leq 2n^2 + n.$$

Thus, $f(n)$ is squeezed between $2n^2 - n$ and $2n^2 + n$.
The worst-case bound for $f(n)$ is $2n^2 - n$, as $2n^2 + n$ is clearly E.N.D since all of its terms are positive. We check for the worst case where the statement $f(n) \leq f(n+1)$ is true:

$$2n^2 - n \leq 2(n+1)^2 - (n+1).$$

Expanding:

$$2n^2 - n \leq 2(n^2 + 2n + 1) - (n+1),$$

$$2n^2 - n \leq 2n^2 + 4n + 2 - n - 1.$$

$$0 \leq 4n + 1.$$

Since $0 \leq 4n + 1$ holds for all $n \in \mathbf{N}$, we conclude:

$$f(n) \text{ is E.N.D. for all } n \in \mathbf{N}.$$

2 Recurrences (25 points)

2.1 Question 1

(10 points) Show the complete steps for finding the characteristic polynomial and the roots (with their multiplicities) of the recurrence below. After finding the roots, write the general form of the recurrence with generic constants c_i.

$$t_n = 4t_{n-1} - 4t_{n-2} + 4(n+1)4^n$$

Hint: Start by multiplying the recurrence by a constant and changing n to $n-1$.
Please note: You should not answer by simply using the formula on page 126 of the Bratley-Brassard reference book, but use it to validate your own answer.
Answer:
We start with the given recurrence:

$$t_n = 4t_{n-1} - 4t_{n-2} + 4(n+1)4^n.$$

We will attempt to rewrite the equation into an homogeneous one through succesive differencces. Separating the polynomic terms to one side of the equation, and noting that $4 \cdot 4^n = 4^{n+1}$

$$t_n - 4t_{n-1} + 4t_{n-2} = (n+1)4^{n+1}.$$

Expanding it:

$$t_n - 4t_{n-1} + 4t_{n-2} = 4^{n+1} + n4^{n+1} \tag{$\triangle$}$$

Multiply the entire equation by 4:

$$4t_n - 16t_{n-1} + 16t_{n-2} = 4^{n+2} + n4^{n+2}.$$

Shifting $n \to n-1$:

$$4t_{n-1} - 16t_{n-2} + 16t_{n-3} = 4^{n+1} + (n-1)4^{n+1} \tag{$\star$}$$

Subtracting the two equations ($\triangle - \star$):

$$t_n - 4t_{n-1} + 4t_{n-2} - (4t_{n-1} - 16t_{n-2} + 16t_{n-3}) = 4^{n+1} + n4^{n+1} - (4^{n+1} + (n-1)4^{n+1})$$

Expanding:

$$t_n - 4t_{n-1} - 4t_{n-1} + 4t_{n-2} + 16t_{n-2} - 16t_{n-3} = 4^{n+1} - 4^{n+1} + n4^{n+1} - n4^{n+1} + 4^{n+1}$$

$$t_n - 8t_{n-1} + 20t_{n-2} - 16t_{n-3} = 4^{n+1} \tag{$\sharp$}$$

Multiplying by 4,

$$4t_n - 32t_{n-1} + 80t_{n-2} - 64t_{n-3} = 4^{n+2}$$

Shifting again:

$$4t_{n-1} - 32t_{n-2} + 80t_{n-3} - 64t_{n-4} = 4^{n+1} \tag{$\flat$}$$

Subtracting the two equations ($\sharp - \flat$):

$$t_n - 8t_{n-1} + 20t_{n-2} - 16t_{n-3} - (4t_{n-1} - 32t_{n-2} + 80t_{n-3} - 64t_{n-4}) = 4^{n+1} - 4^{n+1}$$

Expanding:

$$t_n - 8t_{n-1} - 4t_{n-1} + 20t_{n-2} + 32t_{n-2} - 16t_{n-3} - 80t_{n-3} + 64t_{n-4} = 4^{n+1} - 4^{n+1}$$

$$t_n - 12t_{n-1} + 52t_{n-2} - 96t_{n-3} + 64t_{n-4} = 0$$

Thus the characteristic polynomial is:

$$x^n - 12x^{n-1} + 52x^{n-2} - 96x^{n-3} + 64x^{n-4} = 0$$

Multiplying by $\frac{1}{x^{n-4}}$

$$x^4 - 12x^3 + 52x^2 - 96x + 64 = 0$$

Using Ruffini's method:

$$\begin{array}{r|rrrrr} & 1 & -12 & 52 & -96 & 64 \\ 2 & & 2 & -20 & 64 & -64 \\ \hline & 1 & -10 & 32 & -32 & 0 \end{array}$$

$$\begin{array}{r|rrrr} & 1 & -10 & 32 & -32 \\ 2 & & 2 & -16 & 32 \\ \hline & 1 & -8 & 16 & 0 \end{array}$$

$$\begin{array}{r|rrr} & 1 & -8 & 16 \\ 4 & & 4 & -16 \\ \hline & 1 & -4 & 0 \end{array}$$

$$\begin{array}{r|rr} & 1 & -4 \\ 4 & & 4 \\ \hline & 1 & 0 \end{array}$$

So we can factorize the characteristic polynomial as:

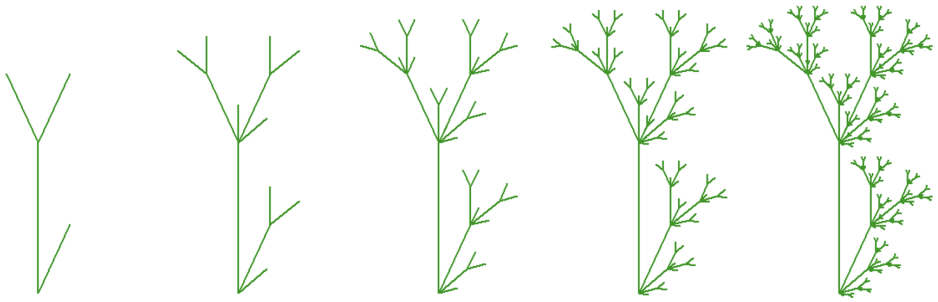
$$(x-4)^2(x-2)^2 = 0$$

Thus the solution must be of the form:

$$t_n = c_1 2^n + c_2 n 2^n + c_3 4^n + c_4 n 4^n.$$

2.2 Question 2

(10 points) *Consider the following code that draws a plant fractal with python turtle.*
What is the complexity of this code? How many "t.forward(length)" plots are made for a certain level? Model the recurrence and solve exactly. We want to have the exact form and complexity. Note that the code is also in a file with the assignment and a counter has been added to validate your own answer at the end.



Code

```
import turtle
t = turtle.Turtle()
t.shape("turtle")
t.speed(0)
count = 0
alpha = 2
angle = 25

t.penup()
t.lt(90)
t.backward(200)

def tree(length, level):
    global count
    if level > 0 :
        t.pendown()
        t.forward(length)
        count += 1
        t.penup()

        t.rt(angle)
        tree(length/alpha, level - 1)
        t.lt(angle*2)
        tree(length/alpha, level - 1)
        t.rt(angle)
        t.backward(length)
        t.rt(angle)
        tree(length/alpha, level - 1)
        t.lt(angle)

tree(200,4)
print(count)
turtle.exitonclick()
```

Answer:
From the code we can note that for a level the recurrence relation for the "t.forward" calls is:

$$t_n = 3t_{n-1} + 1$$

Let's find first the homogeneous solution. The characteristic polynomial is:

$$x^n - 3x^{n-1} = 0$$

Multiplying by $\frac{1}{x^{n-1}}$:

$$x - 3 = 0$$

Thus, we can write the homogeneous recurrence relation as:

$$t_n^h = C \cdot 3^n$$

As the original equation has a constant term, we guess that the particular solution is a constant, lets say K.

$$t_n^p = K \quad \forall n$$

Note that t_n^p must satisfy the recurrence relation $t_n = 3t_{n-1} + 1$, thus:

$$t_n^p = 3t_{n-1}^p + 1$$

Notice, $t_n^p = K \quad \forall n$, then:

$$K = 3K + 1 \implies K = -\frac{1}{2}$$

The full solution is the sum of the particular solution and the homogeneous solution $t_n = t_n^h + t_n^p$, thus:

$$t_n = C \cdot 3^n - \frac{1}{2}$$

From the code we see that for level=0, there is no call, thus:

$$t_0 = 0 = C - \frac{1}{2} \implies C = \frac{1}{2}$$

Finally, the exact form is:

$$t_n = \frac{1}{2}3^n - \frac{1}{2}$$

With the code we verify that for $n = 4$, the result is 40:

$$t_4 = \frac{1}{2}3^4 - \frac{1}{2} = \frac{1}{2}(3^4 - 1) = \frac{1}{2}(81 - 1) = \frac{1}{2} \cdot 80 = 40$$

2.3 Question 3

(5 points) *Use the master theorem to find the exact order of the following recurrences:*

2.3.1 $t(n) = t\left(\frac{n}{2}\right) + 4n$

Answer:
The Master's theorem tells us that given:

- $n_0 \geq 1 \in \mathbf{N}$
- $l \geq 1 \in \mathbf{N}$
- $b \geq 2 \in \mathbf{N}$
- $k \geq 0 \in \mathbf{N}$
- $c \in \mathbf{R}^+$

and a recurrence relation of the form:

$$T(n) = lT\left(\frac{n}{b}\right) + cn^k, \quad \forall n > n_0$$

Where $\frac{n}{n_0}$ is a power of b. Then,

$$T(n) \in \begin{cases} \Theta(n^k) & \text{if } l < b^k \\ \Theta(n^k \log_b n) & \text{if } l = b^k \\ \Theta(n^{\log_b l}) & \text{if } l > b^k \end{cases}$$

In this case $l = 1, b = 2, k = 1$, thus we have:

$$1 \textbf{ vs } 2^1 \implies 1 < 2^1 \implies l < b^k$$

Thus:

$$t\left(\frac{n}{2}\right) + 4n \in \Theta(n)$$

2.3.2 $t(n) = 2t\left(\frac{n}{2}\right) + 2n$

Answer:
In this case $l = 2, b = 2, k = 1$, thus we have:

$$2 \textbf{ vs } 2^1 \implies 2 = 2 \implies l = b^k$$

Thus:

$$t(n) = 2t\left(\frac{n}{2}\right) + 2n \in \Theta(n \log_2 n)$$

2.3.3 $t(n) = 3t\left(\frac{n}{3}\right) + 3n^3$

Answer:
In this case $l = 3, b = 3, k = 3$, thus we have:

$$3 \textbf{ vs } 3^3 \implies 3 < 27 \implies l < b^k$$

Thus:

$$3t\left(\frac{n}{3}\right) + 3n^3 \in \Theta(n^3)$$

2.3.4 $t(n) = 4t\left(\frac{n}{3}\right) + 2n$

Answer:
In this case $l = 4, b = 3, k = 1$, thus we have:

$$4 \textbf{ vs } 3^1 \implies 4 > 3 \implies l > b^k$$

Thus:

$$4t\left(\frac{n}{3}\right) + 2n \in \Theta(n^{\log_3 4})$$

2.3.5 $t(n) = 4t\left(\frac{n}{2}\right) + 2n^2$

Answer:
In this case $l = 4, b = 2, k = 2$, thus we have:

$$4 \textbf{ vs } 2^2 \implies 4 = 4 \implies l = b^k$$

Thus:

$$4t\left(\frac{n}{2}\right) + 2n^2 \in \Theta(n^2 \log_2 n)$$

3 Double pointers- Code Easy (10 points)

Question:
You have an ascending sorted list of n exam scores. Find the number of distinct pairs of scores that sum to the median of this list. For example, if we have $\{1, 1, 1, 3, 3, 3\}$, the median is 2 and the only distinct pair of scores is $(1, 1)$. Solve the problem in Python 3.
Desired complexity: *$O(n)$ where n is the input size. Your algorithm should be efficient to get all points.*
Code
Some of the code is already written. In the `Q3.py` file, you need to complete the `solve()` function (which takes a sorted list of elements and returns the number of pairs that satisfy the problem). You can write auxiliary functions if needed. A `Q3_test.py` file and input files are provided to help you test your code.
Example call (in the folder where `Q3.py` and `input.txt` are located):

```
python3 Q3.py input.txt
```

Submission: *Complete the file `Q3.py` provided, and submit **only** this file. Do not submit the file `Q3_test.py`, or any other test file.*
Answer:
To solve this in $O(n)$ we can use the two pointers method (Laaksonen, 2017). The idea is that in a single transversal of the list we scan from the beginning and end of the list with two pointers. We check if the sum of the two numbers is equal to the target sum. See Q3.py in the submission. The core of the code added is:

```python
def get_median(numbers):
    #print(numbers)
    n = len(numbers) # len is in O(1) time in the python list implementation
    if n == 0:
        return None
    if n % 2 == 0:
        return (numbers[n//2-1] + numbers[n//2]) / 2
    else:
        return numbers[n//2]
```

```python
def get_pairs_two_pointers(arr, target):
    left, right = 0, len(arr) - 1 # len is in O(1) time in the python list implementation
    count = 0
    pairs=[]

    while left < right:
        pair_sum = arr[left] + arr[right]

        if pair_sum == target:
            count += 1
            left += 1
            right -= 1  # Move both pointers inward
            pairs.append((arr[left], arr[right]))
            # Skip duplicates by moving past identical values
            while left < right and arr[left] == arr[left - 1]:
                left += 1
            while left < right and arr[right] == arr[right + 1]:
                right -= 1

        elif pair_sum < target:
            left += 1  # Increase sum by moving left pointer
        else:
            right -= 1  # Decrease sum by moving right pointer

    return pairs # im returning the pairs themselves because process_numbers does the len() call
```

4 Sliding Window- Code Easy (10 points)

Question:
A merchant sells children's toys. The n toys are displayed in a long row behind a glass case. Each toy i is labeled with a price $J[i] \in \mathbf{N}, \forall 0 \leq i < n$. *A child has been given a sum of money S and wants to buy as many toys as possible with this sum. However, the parents have challenged him to buy only a set of toys that are neighbors in the row. The child must therefore find the largest segment of toys that he can buy. How big is this segment? Solve the problem in C++.*
Desired complexity: $O(n)$ *where n is the number of toys. Your algorithm should be efficient to have all the points.*
Code
Example call:
Compilation:

```
g++ -o max_toys.exe max_toys.cpp MaxToysCalculator.cpp
```

Execution:

```
.\max_toys.exe
```

Submission: *Complete the provided* ***MaxToysCalculator.cpp*** *and* ***MaxToysCalculator.h*** *files, and only submit these files. Do not submit the* ***max_toys.cpp*** *file.*
Answer:
To solve this we can similarly use again the two pointer method but on a sliding window. The idea will be to start with a window of size 0 and expand it toy by toy. If the sum of the toys in the window exceeds the budget, we shrink the window from the left

Do note that because they toys are required to be contiguous the solution space really shrinks, and I actually believe this greedy approach gets the global optimum; I'm not completely certain, but my intuition is that since at each step, we are either expanding (exploring a new possibility) or shrinking (discarding a bad choice), we effectively explore all valid contiguous segments without missing any.

I think that because each element is either added at the right or discarded at the left, each one is processed 2 times at most. So the algorithm should be on $O(n)$.
In anycase, check the submitted files: max_toys.cpp and MaxToysCalculator.h .

The core of the code added is:

```
 int MaxToysCalculator::CalculateMaxToys(const vector<int>& Toys, int S) {

// Student: Yorguin José Mantilla Ramos - 20253616
// Sliding Window Approach to find the longest contiguous segment
// The idea will be to start with a window of size 0 and expand it toy by toy
// If the sum of the toys in the window exceeds the budget, we shrink the window from the left
// Moreover because they are required to be contiguous,
// I think this greedy approach actually returns the global optimum.
// But im not completely sure...

int left = 0;          // Left boundary of the window (start of toy segment)
int current_sum = 0;   // Sum of toy prices in the window
int max_toys = 0;      // The best (largest) segment we've found so far

// Right pointer expands the window toy by toy
for (int right = 0; right < Toys.size(); right++) {
    current_sum += Toys[right];  // Include the new toy in our segment

    // If we've exceeded the budget, shrink the window from the left
    while (current_sum > S) {
        current_sum -= Toys[left]; // Remove the toy at the left boundary
        left++;  // Move the left pointer to shrink the window
    }

    // Update the max length of a valid contiguous segment
    max_toys = max(max_toys, right - left + 1);
}

return max_toys;
}
```

5 Greedy Code Medium (15 points)

Question:
In a simplified and figurative world, a person is on a large ribbon of a planned career consisting of n squares. Each square of the ribbon R represents key stages of the career and allows the person to move up to R[i] squares forward. For example, reaching the end of primary school allows one to go to the end of secondary school, which in turn allows further advancements. The person starts on the first square of the ribbon and wants to go to the last square of the ribbon which represents the ultimate goal of his/her career plan. Please determine if this goal is achievable according to the planned stages of the career if the person makes the best possible choices. Solve the problem in C++.
Desired complexity: $O(n)$ *where n is the number of boxes. Your algorithm should be efficient to have all the points.*
Example call:
Compilation:

```
g++ -o career.exe career.cpp CareerCalculator.cpp
```

Execution:

```
.\career.exe
```

Submission: *Complete the provided* `CareerCalculator.cpp` *and* `CareerCalculator.h` *files, and submit* ***only*** *these files. Do not submit the* `career.cpp` *file.*
Answer:
I originally considered solving this problem using a local per-square greedy approach, where at each step, I would make the largest immediate jump possible. However, after re-reading the problem statement, I realized that the phrase "the person makes the best possible choice" refers to finding the global optimum, rather than just taking the best local move.
Based on this insight, I researched online and discovered that this is a well-known interview problem, often referred to as the Jump Game Problem. To solve it efficiently in $O(n)$, I will implement the approach from Stack Overflow User "cheeken" (2012).

The core idea is to traverse the array once using a for-loop while tracking the maximum reachable index at each step. If we ever reach a square beyond our maximum reachable index, it means that reaching the end is impossible. Otherwise, if we can reach or surpass the last square, the problem is solvable.
Check the submitted files `CareerCalculator.cpp` and `CareerCalculator.h` .
The core of the code added is:

```
bool CareerCalculator::CalculateMaxCareer(const vector<int>& Steps) {

    // Student: Yorguin José Mantilla Ramos - 20253616
    // Greedy approach to check if the last stage of the career can be reached
    int end_stage = Steps.size();  // Total number of career stages (squares)
    int max_reachable_square = 0;  // Maximum square that can be reached so far

    for (int square = 0; square < end_stage; square++) {
        if (square > max_reachable_square) {
            return false; // If we reached an unreachable square, return false
        }
        // Update the maximum reachable square from the current position
        max_reachable_square = max(max_reachable_square, square + Steps[square]);

        // If we can reach or surpass the last square, return true
        if (max_reachable_square >= end_stage - 1) {
            return true;
        }
    }

    return false; // If loop completes without reaching the last square, return false
}
```

6 Divide and Conquer- Code Hard (15 points)

Question:
You get 2 lists of numbers already sorted. You have to find the median of all these numbers, and do it as quickly as possible. Solve the problem in Python 3.
Desired complexity: $\mathcal{O}(\log(n+m))$ *where* n *and* m *are the lengths of the 2 lists respectively. Your algorithm should be efficient to get all the points. Start with a solution in* $\mathcal{O}((n+m)\log(n+m))$ *where we simply sort everything, followed by a solution in* $\mathcal{O}(n+m)$ *which sorts using the fact that the 2 lists are already sorted, to test your next solutions. Partial points may be given for these solutions if no solution is returned with the desired complexity.*
Code
Some of the code is already written. In the Q6.py file, you need to complete the function `solve()` *(which takes two sorted lists of elements and returns the median of the two lists). You can write auxiliary functions if needed.*
Example call (in the folder where Q6.py and input.txt are located):

```
python Q6.py input.txt
```

Submission: *Complete the file* `Q6.py` *provided, and only submit* **this** *file. Do not submit the file* `Q6-test.py`*, or any other test file.*
Hints

- *Start with an odd number of items in total.*
- *Perform a dichotomous search that uses the fact that the lists are sorted.*
- *A true median will separate the elements of the list into how many elements to the left and right?*
- *If we take a candidate for the median in a first list, where should he be located in the second list?*

Answer:
Given two sorted arrays, we need to find the median of their combined elements. I implemented two approaches. First, the naive approach of merging both arrays which finds the median with a time complexity of $O(n+m)$. Second, the divide and conquer approach, which achieves it in $O(\log(n+m))$. The divide and conquer approach is the one used by default (hard-coded, as the other one is commented).

Naive Approach

Basically, merge the lists noting that both are already sorted. So you construct a new list by grabbing from the current lowest element when traversing both lists at the same time. This uses two pointers.
The code for this approach is:

```python
## Naive approach with somewhat careful merge
def get_median(numbers):
    n = len(numbers) # len is in O(1) time in the python list implementation
    if n == 0:
        return None
    if n % 2 == 0:
        return (numbers[n//2-1] + numbers[n//2]) / 2
    else:
        return numbers[n//2]
def sorted_merge(nums1, nums2):
    # This one should be on O(n+m) time complexity
    size1 = len(nums1)
    size2 = len(nums2)

    if size1 > size2:
        nums1, nums2 = nums2, nums1  # Swap to ensure nums1 is the smaller list
        size1, size2 = size2, size1

    if size1 == 0:  # Edge case: if one list is empty
        size1 = size2
        nums1 = nums2
        size2 = 0
        nums2 = []

    i, j = 0, 0
    merged = []

    while True:
        if i == size1:
            merged += nums2[j:]  # Add the remaining elements of nums2
            break
        if j == size2:
            merged += nums1[i:]  # Add the remaining elements of nums1
            break
        if nums1[i] < nums2[j]:  # Merge step (sorted merge)
            merged.append(nums1[i])
            i += 1
        else:
            merged.append(nums2[j])
            j += 1
    return merged
```

Divide and Conquer Approach

The insight is that the middle elements of the merged array with T elements, is the same as finding the T//2+1 smallest element. This element cannot be in the first half of the list with the lowest median. Thus we can get rid of those elements, and reduce the search space. More precisely:

1. Suppose we need to find the k-th smallest element in the combined sorted arrays.
2. We compare the $\lfloor k/2 \rfloor$-th element from both arrays.
3. If the $\lfloor k/2 \rfloor$-th element of one array is smaller, we discard the first $\lfloor k/2 \rfloor$ elements of that array.
4. We then search for the $k - \lfloor k/2 \rfloor$-th smallest element in the remaining portion.
5. The process repeats until $k = 1$, at which point we return the minimum of the first elements of the remaining arrays.

Edge Cases

- If one of the arrays becomes empty, we directly return the k-th element from the other array.
- If $k = 1$, we return the smaller of the first elements from both arrays.
- If an array has fewer than $k/2$ elements, we treat its missing elements as $+\infty$ to avoid problems with the logic of finding the lower median.

Computing the Median

- If the total number of elements is odd, the median is the $\left\lfloor \frac{n+m}{2} \right\rfloor + 1$-th smallest element.
- If the total number of elements is even, the median is the average of the $\frac{n+m}{2}$-th and $\frac{n+m}{2} + 1$-th smallest elements.

Time Complexity Analysis

Each recursive step reduces the search space by half, leading to a time complexity of: $O(\log(n+m))$.
The code for the divide and conquer approach is:

```python
## Divide and conquer approach
def find_kth_smallest(nums1, nums2, k):
    #print('Nums1:', nums1, 'Nums2:', nums2, 'K:', k)
    # Function to find the k-th smallest element in two sorted arrays
    # k is always 1-based index (k=1 means the smallest element)

    # Base case: if nums1 is empty, return the k-th element from nums2
    if not nums1:
        #print('List 1 is empty')
        return nums2[k-1]  # k is 1-based, so we access k-1 (0-based indexing in Python)
    # Base case: if nums2 is empty, return the k-th element from nums1
    if not nums2:
        #print('List 2 is empty')
        return nums1[k-1]  # k is 1-based, so we access k-1 (0-based indexing in Python)
    # Base case: if k == 1, return the smallest element from both arrays
    if k == 1:
        #print('K=1', 'Getting the smallest between', nums1[0], nums2[0])
        return min(nums1[0], nums2[0])  # k=1 means smallest element

    # Get the k//2-th element from each list, if it exists
    # If the list has fewer than k//2 elements, we set mid to infinity (inf)
    # to effectively ignore it
    mid1 = nums1[k//2 - 1] if len(nums1) >= k//2 else float('inf')  # k is 1-based
    mid2 = nums2[k//2 - 1] if len(nums2) >= k//2 else float('inf')  # k is 1-based
    #print('Middle values:', mid1, mid2)

    # Remove k//2 elements from the list with the smaller mid value
    # Why? Because we know that the k-th smallest element **cannot** be
    # in the first k//2 elements of this array.
    # This is because:
    # - The k-th smallest element is greater than at least k//2 elements overall.
    # - If mid1 < mid2, then mid1 and all elements before it in nums1 are too small
    #   to be the k-th smallest element.
    if mid1 < mid2:
        # Remove the first k//2 elements from nums1
        # The total number of elements to consider is reduced by k//2
        # We now look for the (k - k//2)-th smallest element in the
        # remaining part of nums1 and full nums2
        return find_kth_smallest(nums1[k//2:], nums2, k - k//2)
    else:
        # Remove the first k//2 elements from nums2
        # Similarly, we now look for the (k - k//2)-th smallest element
        return find_kth_smallest(nums1, nums2[k//2:], k - k//2)
```

```python
def divide_and_conquer(nums1, nums2):
    # Intuition:
    # - The median is the middle element in a sorted list.

    # - If the total number of elements (k = len(nums1) + len(nums2)) is even,
    #   the median is the average of the two middle elements.
    #   (the k//2-th and (k//2 + 1)-th smallest in 1-based indexing).

    # - If the total number of elements is odd,
    #   the median is the middle element (the (k//2 + 1)-th smallest in 1-based indexing).

    # - The challenge is to find these elements without fully merging the two sorted lists.
    #   We use a *divide-and-conquer* approach (binary search on k-th smallest).

    # - Instead of merging, we repeatedly remove k//2 elements from one of the arrays.

    # - To do this:
    #   0. Suppose you are looking for the k-th smallest element.
    #   1. Compare the k//2-th element in each array.
    #   2. Remove the first k//2 elements from the array with the smaller k//2-th element.
    #      We can ignore these elements as the k-th element you are looking for
    #      is greater than k//2 elements in that array.
    #   3. Repeat this logic until k=1 (base case).

    total_len = len(nums1) + len(nums2)  # Compute total length of both arrays
    #print('Total Len:',total_len)

    # If the total length is odd, return the middle element
    if total_len % 2 == 1:
        #print('Odd Case')
        return find_kth_smallest(nums1, nums2, total_len // 2 + 1)
        # k is 1-based, so the middle element is at k//2 + 1

    # If the total length is even, return the average of the two middle elements
    else:
        #print('Even Case')
        right = find_kth_smallest(nums1, nums2, total_len // 2 + 1)  # Right median
        left = find_kth_smallest(nums1, nums2, total_len // 2)  # Left median, (the one before)
        return (left + right) / 2  # Compute the average
```

References

A. Laaksonen. Competitive programmer's handbook. *Preprint*, 5, 2017.

Stack Overflow User "cheeken". Interview Puzzle: Jump Game, 2012. URL https://stackoverflow.com/questions/9041853/interview-puzzle-jump-game.