time complexity	**April 13, 2025**

Homework 1

Spring 2025	*COSC 31: Algorithms*

1. We represent a degree n polynomial, $p(x) = a^0 + a_1x + a_2x^2 + \ + a_nx^n$, by an array, $A[0...n]$, where $A[i] = a_i$ for each index $i \in \{0, 1, ..., n\}$. In this problem, your goal is to design an algorithm *EvaluatePolynomial(A, x)* that takes such an array A (which represents a polynomial p) and a number x as input, and computes $p(x)$.

   For this problem,

   (a) Design an algorithm *SimpleEvaluatePolynomial(A, x)*, which performs at most $O(n)$ multiplications and $O(n)$ additions.

   Answer

   Algorithm SimpleEvaluatePolynomial

   ```
   Function SimpleEvaluatePolynomial(A, x):
       q ← 1
       p ← A[0]
       for i ← 1 to n do
           q ← q · x
           p ← p + q · A[i]
       end
       return p
   ```

   Analysis: The simple algorithm above does $2n$ multiplications and n additions. For each iteration of the loop, two multiplications are done, $A[i] \cdot q$ and $q \cdot n$. Ignoring the constants we have at most $O(n)$ multiplications and $O(n)$ additions

   (b) Design an algorithm *EfficientEvaluatePolynomial(A, x)*, which performs at most n multiplications and n additions.

   Answer

   Using the Horner's method, the polynomial can be written nested form as shown below

   $$p(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$$

   $$p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + a_x) \ldots))$$

   Example the following polynomial $p(x) = 3x^3 + 2x^2 + x + 1$ can be nested follows:

   $$p(x) = x(1 + x(2 + 3x)) + 1$$

   To evaluate $p(x)$ in the example above, we can start by evaluating the term $2 + 3x$ where is the 3 corresponds to $A[n]$

Algorithm EfficientEvaluatePolynomial

```
Function EfficientEvaluatePolynomial(A, x):
    p ← A[n]
    for i ← n − 1 to 0 do
        p ← p · x + A[i]
    end
    return p
```

Analysis: The above efficient algorithm doesn at most n multiplications and n additions, where each iteration of the loop does a 1 multiplication and 1 addition

2. Prove that $f(n) = \Omega(g(n))$ if and only if $g(n) = \mathcal{O}(f(n))$

Answer

This means $f(n) = \Omega(g(n)) \iff g(n) = \mathcal{O}(f(n))$

Starting with $f(n) = \Omega(g(n)) \Rightarrow g(n) = \mathcal{O}(f(n))$, we have that $f(n) = \Omega(g(n))$ if

$$\exists n_0, c > 0, \forall n > n_0 : f(n) \geq c \cdot g(n) \tag{1}$$

This can be re-arranged as follows:

$$\exists n_0, c > 0, \forall n > n_0 : \frac{1}{c} \cdot f(n) \geq g(n) \tag{2}$$

We see that statement(2) follows the definition of $g(n) = \mathcal{O}(f(n))$, hence $f(n) = \Omega(g(n)) \Rightarrow g(n) = \mathcal{O}(f(n))$

To show $g(n) = \mathcal{O}(f(n)) \Rightarrow f(n) = \Omega(g(n))$, we have $g(n) = \mathcal{O}(f(n))$ if

$$\exists n_0', c' > 0, \forall n > n_0' : g(n) \leq c' \cdot f(n) \tag{3}$$

This can be re-arranged as follows:

$$\exists n_0', c' > 0, \forall n > n_0' : \frac{1}{c'} \cdot g(n) \leq f(n) \tag{4}$$

We see that statement(4) follows the definition of $f(n) = \Omega(g(n))$, hence $g(n) = \mathcal{O}(f(n)) \Rightarrow f(n) = \Omega(g(n))$
Therefore, $f(n) = \Omega(g(n)) \iff g(n) = \mathcal{O}(f(n))$.

3. Prove that $\log_a n = \Theta(\log_b)n$ for all $a, b > 1$.

Answer

To show that $\log_a n = \Theta(\log_b n)$ for all $a, b > 1$, we must prove that:

$$\log_a n = \mathcal{O}(\log_b n) \quad \text{and} \quad \log_a n = \Omega(\log_b n)$$

Using the change of base formula:

$$\log_a n = \frac{\log_b n}{\log_b a} = \frac{1}{\log_b a} \cdot \log_b n$$

Big-O proof: We want to show that:

$$\exists n_0, c > 0, \forall n > n_0 : \log_a n \leq c \cdot \log_b n$$

Let $c = \frac{1}{\log_b a}$, and $n_0 = 1$. Then for all $n > 1$:

$$\log_a n = \frac{1}{\log_b a} \cdot \log_b n \leq c \cdot \log_b n$$

So $\log_a n = \mathcal{O}(\log_b n)$.
Big-Omega proof: We want to show that:

$$\exists n_0, c > 0, \forall n > n_0 : \log_a n \geq c \cdot \log_b n$$

Again, let $c = \frac{1}{\log_b a}$ and choose $n_0 = 1$. Then for all $n > 1$:

$$\log_a n = \frac{1}{\log_b a} \cdot \log_b n \geq c \cdot \log_b n$$

So $\log_a n = \Omega(\log_b n)$.
Since both bounds hold, we conclude:

$$\log_a n = \Theta(\log_b n)$$

4. Exponentials are of different orders
   (a) Prove that $2^n = \mathcal{O}(3^n)$

Answer

$2^n = \mathcal{O}(3^n)$ if $\exists n_0, c > 0, \forall n_0 > n : 2^n \leq c \cdot 3^n$. The inequality can be re-arranged as follows:

$$2^n \leq c \cdot 3^n \equiv \left(\frac{2}{3}\right)^n \leq c$$

As $0 < \left(\frac{2}{3}\right)^n < 1, \forall n$. Consider a suffiently large n_0 e.g, $n_0 = 10^5$, and $c = 1$, we see that $\left(\frac{2}{3}\right)^{10^5} < 1$ hence $2^n = \mathcal{O}(3^n)$.

   (b) Prove that $3^n \neq \mathcal{O}(2^n)$

Answer

Suppose not, that's $3^n = \mathcal{O}(2^n)$, then we have that $\exists n_0, c > 0, \forall n_0 > n : 3^n \leq c \cdot 2^n$. The inequality can be re-arranged as follows

$$3^n \leq c \cdot 2^n \equiv \left(\frac{3}{2}\right)^n \leq c$$

Consider,$n_0 = 3, c = 1$, we see that $1.5^3 > 1$ hence the contradiction that for $\forall n$, $\left(\frac{3}{2}\right)^n \leq c$.

Therefore, $3^n \neq \mathcal{O}(2^n)$.

5. For each pair of functions $(f(n), g(n))$ below, you must state and prove the strongest possible relationship between them
   (a) $6^{\lg n}$ and $5^{\lg n}$

Answer

The strongest possible relationship is:

$$6^{\lg n} = \omega(5^{\lg n})$$

This is because for suffiently large n, $6^{\lg n}$ grows strictly faster than $5^{\lg n}$. More formally, we see that:

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \left(\frac{5}{6}\right)^{\lg n} = 0$$

which satisfies the condition for the little-omega definition.

   (b) $f(n)$ and n^2, where f is defined as follows: $f(n) = 6n$ if n is even; else $f(n) = 5n^2$

Answer

We have

$$f(n) = \begin{cases} 6n & \text{if } n \mod 2 = 0 \\ 5n^2 & \text{otherwise} \end{cases}$$

$f(n) = \mathcal{O}(g(n))$. This means $\exists n_0, c > 0, \forall n > n_0 : f(n) \leq c \cdot n^2$.

Case 1), For $f(n) = 6n$, consider $n_0 = 6$, and $c = 1$, we see that $6 \cdot 6 \leq 1 \cdot 6^2$ which is true, and holds for a suffiently large n.
Case 2), For $f(n) = 5n^2$, consider $n_0 = 1$, and $c = 6$, we see that $5 \cdot 1^2 \leq 6 \cdot 1^2$.
Therefore, both cases $f(n) = \mathcal{O}(g(n))$

$f(n) \neq \Omega(g(n))$ globally for $f(n)$. Suppose for the sake of contradiction, $f(n) = \Omega(g(n))$, we have $\exists n_0, c > 0, \forall n > n_0 : f(n) \geq c \cdot g(n)$. Consider, $f(n) = 6n$ where $n_0 = 7, c = 1$, we see that $6 \cdot 7 < 1 \cdot 7^2$, hence the contradiction. Therefore $f(n) \neq \Omega(g(n))$, and $f(n) \neq \Theta(g(n))$.

$f(n) \neq o(g(n))$ because $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ varies, that's $\lim_{n\to\infty} \frac{6n}{n^2} = 0$, but $\lim_{n\to\infty} \frac{5n^2}{n^2} = 1$.

As shown above, the strongest relation is $f(n) = \mathcal{O}(g(n))$.

6. **Time Complexity Determination**: Express the time complexity of the following program fragment as a Big-Theta asymptotic, in terms of n. You must also prove that your answer is correct.

Answer

Time complexity is $\Theta(n^3)$. We see that for each k, $s \leftarrow s + i - j \cdot k$ has $\Theta(1)$ which executed j

```
for i ← 1 to n do
    for j ← 1 to i do
        for k ← 1 to j do
            s ← s + i − j · k
        end
    end
end
```

times for j^{th} iteration of the second loop.

$$
\begin{align}
T(n) &= \sum_{i=1}^{n} \sum_{j=1}^{i} \Theta(1) \cdot j \tag{5}\\
&= \sum_{i=1}^{n} \Theta\left(\frac{i^2+i}{2}\right) \tag{6}\\
&= \frac{1}{2}\left(\sum_{i=1}^{n} \Theta(i^2) + \sum_{i=1}^{n} \Theta(i)\right) \tag{7}\\
&= \frac{1}{2} \cdot \left(\frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2}\right) \tag{8}\\
&= \left(\frac{2n^3+3n^2+n}{12}\right) + \left(\frac{n^2+n}{4}\right) \tag{9}\\
&= \frac{2n^3+3n^2+n+3n^2+3n}{12} \tag{10}\\
&= \frac{2n^3+6n^2+4n}{12} \tag{11}\\
& \tag{12}
\end{align}
$$

To see that $T(n) = \mathcal{O}(n^3)$, we can see that:

$$
\begin{align}
T(n) &= \frac{2n^3+6n^2+4n}{12} \tag{13}\\
&\leq \frac{2n^3+6n^3+4n^3}{12} \tag{14}\\
&= \frac{12}{12} \cdot n^3 \tag{15}\\
&= n^3 \tag{16}
\end{align}
$$

Therefore, $T(n) = \mathcal{O}(n^3)$ for some n_0 and c such that $n_0 = 1$ and $c = 1$.
To see that $T(n) = \Omega(n^3)$, we can see that:

$$
\begin{align}
T(n) &= \frac{2n^3+6n^2+4n}{12} \tag{17}\\
&\geq \frac{2n^3}{12} \tag{18}\\
&= \frac{1}{6}n^3 \tag{19}
\end{align}
$$

Therefore, $T(n) = \Omega(n^3)$ for some n_0 and c such that $n_0 = 1$ and $c = \frac{1}{6}$, hence $T(n) = \Theta(n^3)$.

7. Suppose there are n players with IDs $1, 2, ..., n$. Every possible pair (i, j) of players competed and the results are recorded in an $n \times n$ matrix A. In particular, in the match between i and j, if i won against j, then $A[i, j]$ is set to 1 and $A[j, i]$ is set to 0; on the other hand, if the match resulted in a tie, both $A[i, j]$ and $A[j, i]$ are set to 0. For each i, since i does not compete with self, $A[i, i]$ has no meaning and its value can be anything. We call a player i a super athlete if i won against every other player. Here is a simple algorithm to identify a super athlete, if one exists.

Algorithm super-athlete

Input: A positive integer n and an $n \times n$ matrix A, as described above.
Output: The index i of a super athlete, or $\perp$ if no super athlete exists.

```
1 for i ← 1 to n do
2 |   if all entries in the ith row of matrix A, leaving out the entry A[i,i], are 1 then
3 |   |   return i
4 |   end
5 end
6 return ⊥
```

(a) Prove that there is at most one super athlete?

Answer

For a player at row i, and player any j where $1 \leq j \leq$ and $i \neq j$, if $A[i, j] = 1$, then $A[j, i] = 0$ because there can only be one winner in a match, therefore, there is atmost one supper player.

(b) In the algorithm above, Line 4 involves reading all entries of row i, with the exception of the ith entry of that row. Thus, Line 4 involves reading $n - 1$ entries of the matrix. Since Line 4 can be executed up to n times, the algorithm might read up to $\Theta(n^2)$ matrix entries. Design a cleverer algorithm that reads at most $O(n)$ entries of the matrix

Answer

Algorithm Super Player

```
Function SuperPlayer(n, A)
|   super ← 0
|   for i ← 1 to n do
|   |   if super ≠ i and A[super][i] = 0 then
|   |   |   super ← i
|   |   end
|   end
|   for m ← 1 to super − 1 do
|   |   if A[super][m] ≠ 1 then
|   |   |   return ⊥
|   |   end
|   end
|   return super
```

(c) Argue why your algorithm is correct. (You should argue both that the output is correct and that it reads at most $\mathcal{O}(n)$ entries of the matrix.)

Answer

For the optimized we do two inspections, 1) to identify the likely super players, 2) verify if the likey super player is indeed a super player or not.

In the first loop for the algorithm in part b), we keep track of some player at row *super*, we then check if $A[super, i] = 0$ for $1 \leq i \leq n$ and $i \neq super$, and update *super* to i if the condition is satisfied. We do so because we know that *super* has beaten players in the rows $super + 1$ to $i - 1$ hence they can't be super players. If $A[super, i] = 1$ we continue to inspect, until we find next player who has beaten *super*. Therefore, we have $\mathcal{O}(n)$ iterations.

Having identified a potential super player, we can verify if player at row *super* is a super player in the second loop. If some column m, $A[super, m] = 0$ then we return $\perp$, otherwise return player at row *super*. Similarly, we have $super - 1$ iterations where the worst case is $super = n$ hence we have $\mathcal{O}(n)$ for the second loop.

overall, the program has $2 \cdot \mathcal{O}(n)$ time complexity, and we can ignore the constant 2 so the time complexity is $\mathcal{O}(n)$.