

Analysis of Sequential Algorithms

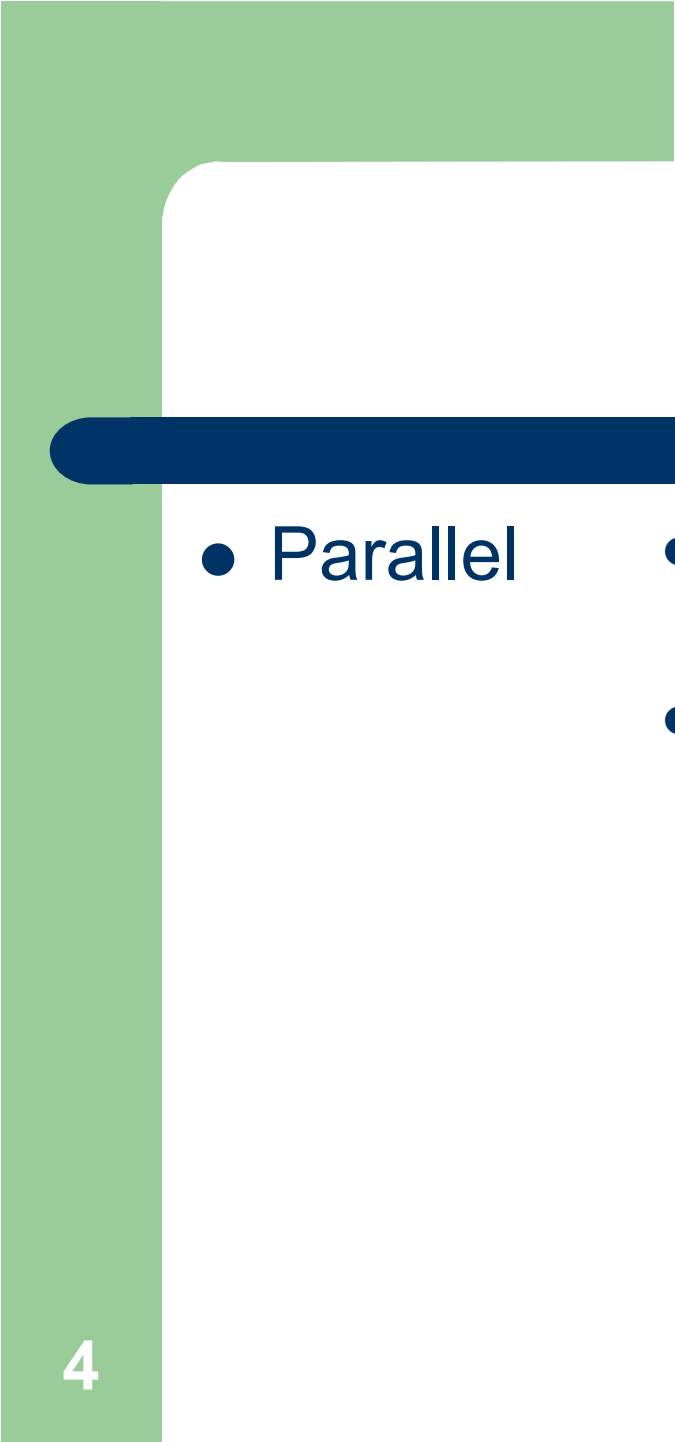

- What is an algorithm ?
- A set of well defined rules for a solution to a given problem
- Takes the input to a problem and transforms it to an output, which solves the problem
- Its concrete representation in a given programming language is called a program

Characteristics of an Algorithm

- Its length must be finite
- Each operation must be unambiguous
- Must acquire data as input and produce result as output, both input and output must belong to two predefined sets
- It must terminate for all inputs

Classes of Algorithm

- Sequential
 - Performs a single operation at a time
 - Implicitly directed to sequential processors

- 
- 
- Parallel
 - Performs several steps at the same time
 - Directed to multiple processors

Why Analysis of Algorithm?

- There are many algorithms for solving a problem
- Hence, we need a measure of choosing among them

Goals of Computer Program Design

- To design an algorithm that is easy to understand, code and debug. Concern of software engineering field of knowledge
- To design an algorithm that makes efficient use of the computer resources (time and space). Concern of data structure and algorithm design

Function Dominance

- A measure usually used to analyze algorithm
- A mathematical expression for computing $T(n)$
- $T(n)$ is an increasing function of size n of the input data
- $T(n)$ is called time-complexity, growth rate, time function or simply complexity
- The value of n in byte is the amount of memory needed to store the input data

Definition of Function Dominance

- Let $f, g : Z^+ \rightarrow R$ be any two integer functions. The function g dominates f or f is dominated by g if \exists constants $m \in R^+$ and $k \in Z^+$ such that

$$|f(n)| \leq m|g(n)| \quad \forall n \in Z^+, \text{ where } n \geq k \quad (11)$$

- Similarly, f dominates g or g is dominated by f if \exists constants $m \in \mathbb{R}^+$ and $k \in \mathbb{Z}^+$ such that

$$|f(n)| \geq m|g(n)| \quad \forall n \in \mathbb{Z}^+, \text{ where } n \geq k \quad (12)$$

- If g dominates f , then f is said to be of order $g(n)$ and the size of the quotient $|f(n)/g(n)|$ is bounded above by m for those $n \in \mathbb{Z}^+$ where $n \geq k$. When g dominates f , the words “big-oh”, “little-oh” and “equivalent” are often used as follows

Big-Oh

- A function $f(n)$ is defined to be of big-order $g(n)$ denoted $O(g(n))$, which is the set of all functions that dominate $f(n)$, if for large n ,

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k \neq 0 \quad (13)$$

Little-Oh

- A function $h(n)$ is said to be of little-order $z(n)$, denoted $o(z(n))$ for large n if

$$\lim_{n \rightarrow \infty} \frac{h(n)}{z(n)} = 0 \quad (14)$$

Equivalent

- If $k = 1$ in Equation (13), $f(n)$ is said to be equivalent to $g(n)$ and we write

$$f(n) \sim g(n)$$

Upper and Lower Bound on Time

- Measure of upper bound (Big-Oh)

$$T(n) = O(g(n)) \text{ or } T(n) \leq g(n) \text{ or } T(n) \in O(g) \quad (15)$$

- Measure of lower bound (Little-Oh, Omega)

$$T(n) = \Omega(g(n)) \text{ or } T(n) \geq g(n) \text{ or } T(n) \in \Omega(g) \quad (16)$$

- Measure of optimality (Equivalent, Theta)

$$T(n) = \theta(g(n)) \text{ or } T(n) = g(n) \quad (17)$$

Example

- Given that

$$f(n) = 100 \log_2 n, \quad g(n) = \left(\frac{1}{2}\right)^n$$

Show that g dominates f ?

Solution

$$|f(n)| = |100 \log_2 n| = \left| 200 \left(\frac{1}{2} \log_2 n \right) \right|$$

since for all $n \in \mathbb{Z}^+$, $0 \leq \log_2 n < n$, then

$$\begin{aligned} |f(n)| &< \left| 200 \left(\frac{1}{2} \right) n \right| \\ &= 200 |g(n)| \end{aligned}$$

This implies that $f \in O(g(n))$ and

$$\left| \frac{f(n)}{g(n)} \right| \text{ is bounded by } 200 \quad \forall n \geq k = 1$$

Asymptotic Analysis

- Finds a solution that closely approximates the exact solution
- Relative errors of the approximation is small
- Asymptotic tools: O-notation, bootstrapping, dissecting

Time Complexity of Algorithm

- Factors that affects actual running time of an algorithm: computer processor, programming language, programmer's skill, input data
- Running time should be measured only by basic steps the algorithm goes through and not esoteric factors

- These basic operations are: arithmetic operation (+, -, /, *), comparisons (>, <, >=, <=, ==, !=), assignment (=), predicate evaluation, number of swaps or moves
- Time complexity is defined as a function of input size(n), which returns the number of basic operations

Examples of Input Size and Frequently Used Orders

- List/array (array size), matrix (number of rows and column), Graph (number of vertices and edges), Integer (number of bits)
- Constant ($O(1)$), Logarithmic ($O(\log_2 n)$), Linear ($O(n)$), Polylogarithmic ($O(n \log_2 n)$), Quadratic ($O(n^2)$), Cubic ($O(n^3)$), Polynomial ($O(n^m)$, $m=0, 1, \dots$), Exponential ($O(e^n)$, $e>1$), Factorial ($O(n!)$)

Types of Running Times

- Worst Case (frequently used)
- Average/Expected Case
- Best Case

Steps in Analyzing Worst Case Time Complexity of an Algorithm

- Determine the input size n
- Count the numbers of basic operations for the worst case and represent it as a function of n
- Simplify the function using the asymptotic notation

Simplifications of Expressions in Asymptotic Analysis

- If c is a constant value, then
 $O(f(n) + c) = O(f(n))$, $O(cf(n)) = O(f(n))$
- If asymptotically, f grows faster than g , then
 $O(f(n) + g(n)) = O(f(n))$
- If f_1 is in $O(g_1)$ and f_2 is in $O(g_2)$ then
 $f_1(n) * f_2(n)$ is in $O(g_1(n) * g_2(n))$

Sum Rule for Consecutive Statements

- Consider the loop-free algorithm A

Algorithm A

Begin

Do statement 1 (S_1)

Do statement 2 (S_2)

...

Do statement k (S_k)

End A

$$\begin{aligned} T_A(n) &= T_{S_1}(n) + T_{S_2}(n) + \dots + T_{S_k}(n) \\ &= \max\{O(T_{S_1}(n)), O(T_{S_2}(n)), \dots, O(T_{S_k}(n))\} \end{aligned}$$

Product Rule

- If $T_1(n) = O(f_1(n))$, $T_2(n) = O(f_2(n))$, ..., $T_k(n) = O(f_k(n))$
 $T_1(n) \cdot T_2(n) \cdot \dots \cdot T_k(n) = O(f_1(n) \cdot f_2(n) \cdot \dots \cdot f_k(n))$

Control Flows

- If c then S_1 Else S_2
- The running time is the running time of c plus the more expensive (i.e. larger) of the running times for S_1 and S_2

Switch Statement

- Take the complexity of the most expensive case

Function Call

- Complexity of the function, including recursive call

Single Loops

- Consider the algorithm B

Algorithm B

Begin

 while (condition (n))

 statement (S)

End B

If the while loop is repeated $f(n)$ times then by product rule, $T_B(n) = f(n) * T_S(n)$

Nested Loops

- The running time is decided by the running time of the statement executed most in the innermost loop. For example,

For i = 1 to n

For j = 1 to n

statement (S)

- The complexity of the nested loop is accordingly given by the product rule of enumeration to be

$$T(n) \in O(n^2)$$

since statement is executed once for each cycle

Combinatorial Proofs for Loops

- Consider the following cases
- Case 1: $j = 0$

For $i = 1$ To n By $i = i + j$

Statement

$j = j + 1$

$$T_1(n) \in O(\sqrt{n})$$

- Case 2: For i = 1 To n By i = i + k
Statement

$$T_2(n) \in O(n)$$

- Case 3: For i = 1 To n By i = i * k
Statement

$$T_3(n) \in O(\log_k(n))$$

Combinatorial Proof for Case 3

- Statement (S) is executed between $i = 1$ to n .
Therefore,

When $i = k^0$

S is executed 1 time

$i = k^1$

S is executed 2 times

$i = k^2$

S is executed 3 times

...

$i = k^{T(n)-1}$

S is executed $T(n)$ times

- But i goes from 1 to n , therefore,

$$k^{T(n)-1} = n$$

Taking logarithm to base k of both sides, we have

$$T(n) - 1 = \log_k n$$

Hence

$$T(n) = \log_k n + 1$$

$$\therefore T(n) \in O(\log_k n)$$

Recursion

- What is recursion?
- A recursive procedure calls itself directly or indirectly
- Permits more lucid, elegant and concise descriptions of algorithm
- Applicable to any problem P that can be decomposed into $k \geq 1$ sub problems P_1, P_2, \dots, P_k identical to P

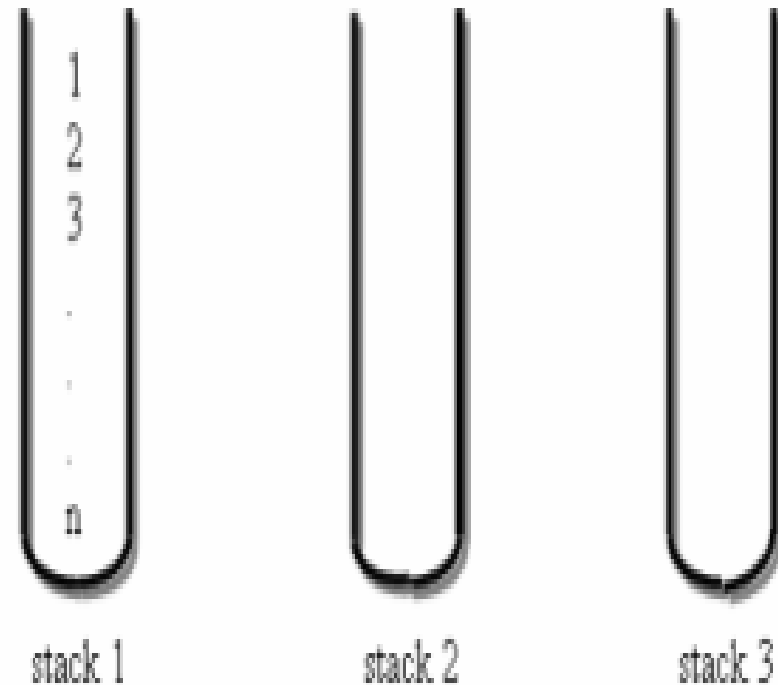
- Uses more memory than an iterative solution and may take longer time to run
- Can be transformed into iterative equivalent using technique called recursion removal
- Efficient for small size problem
- It's time complexity is expressed by recurrence equation

Guidelines for Recursive Problem Solving

- Solve the problem of the same form with smaller input
- Find the termination condition for the recursive method

Example – Tower of Hanoi

- Consider n unique weights stacked as shown. The objective of the problem is to transfer all the weights from stack 1 one at a time so that we end up with the original arrangement in stack 3. A temporary location is provided by stack 2 for any weight but at no time are we allowed to have a larger weight on top of a smaller one. What is the complexity of the algorithm to solve this problem?



Solution

- For $n \geq 0$, let T_n be the number of moves it takes to transfer n weights from stack 1 to stack 3. Then T_n is obtained as follows
- Step 1: Transfer the top $n-1$ weights from stack 1 to stack 2 using stack 3 as auxiliary location. This takes $n-1$ moves

- Step 2: Transfer the largest weight from stack 1 to stack 3. This takes 1 move
- Step 3: Finally, transfer the $n-1$ weights on stack 2 onto the largest weight now on stack 3 as auxiliary location. This requires another $n-1$ moves

- This procedure results in the recurrence relation $T_n = 2T_{n-1} + 1$, where $n \geq 0$ and $T_0 = 0$ (no weight is transferred when $n=0$).
- The solution to the equation, using the standard technique is given by

$$T_n = 2^n - 1, n \geq 0$$