Program Design

- An algorithm is a finite sequence of logically related instructions to solve a computational problem.
- Examples:
  – Given an integer x, test whether x is prime or not.
  – Given a program P, check whether P runs into an infinite loop.
- Properties - Input, Output, Finiteness, Definiteness

- The word **algorithm** (Latin algorithmus *)
  algorism - the art of computing using Arabic numerals
  Mohammed ibn-Musa al-Khwarizmi
  (Persian mathematician)
  arithmós - number (Greek word)
- Types of Algorithm
  – Top-down approach (Iterative algorithm)
  – Bottom-up approach (Recursive algorithm)

---

* Dictionary of Logic as Applied in the Study of Language, pp 14-18

Introduction

- Iterative

```
Fact(n)
{
    for i = 1 to n
        fact = fact * i;
    return fact;
}
```

- Recursive

```
Fact(n)
{
    if n = 1
        return 1;
    else
        return n * fact(n-1);
}
```

1. Correctness For any algorithm, a proof of correctness is important which will exhibit the fact that the algorithm indeed output the desired answer.
2. Amount of work done (time complexity) The time complexity does not refer to the actual running time of an algorithm, instead it is the step count: the number of times each statement in an algorithm is executed.

⋆ The step count focuses on primitive operations along with basic operations.

Input Size and Primitive Operations

Common Problems	Associated Primitive Operations	Input Size
(Search Problem) Find x in an array *A*	Comparison of x with other elements of A	Array size
Multiply 2 matrices A and B	Multiplication and Addition	Dimension of the matrix
Sorting	Comparison	Array size
Graph Traversal	Number of times an edge is traversed	The number of vertices and Edges
Any Recursive Procedure	Number of recursive calls + spent on each recursive call	Recursive depth
Finding Factorial	Multiplication	The input number
Finding LCM(a,b)	Basic arithmetic (div. sub.)	Number of bits to represent a and b

- The number of primitive operations increases with the problem size.
- Space complexity is a related complexity measure that refers to the amount of space used by an algorithm.
- Among the many many algorithms for a given combinatorial problem, a natural question is to find the best algorithm (efficient).

Analysis of algorithms using step count method

1. There is no count for { and }.
2. Each basic statement like assignment and return have a count of 1.
3. If a basic statement is iterated, then multiply by the number of times the loop is run.
4. The loop statement is iterated n times, it has a count of (n + 1). Here the loop runs n times for the true case and an additional check is performed for the loop to exit (the false condition).

1 Sum of elements in an array

```
                                  Step-count (T.C)       Step-count (Space)
Algorithm Sum(a,n)
{                                       0
     sum = 0;                           1                  1 word for sum
     for i = 1 to n do                 n+1                 1 word each for i and n
         sum = sum + a[i];              n                  n words for the array a[]
     return sum;                        1
}                                       0

                            Total:     2n+3                (n+3) words
```

2 Adding two matrices of order m and n

```
Algorithm  Add(a, b, c, m, n)            Step Count
{
  for i = 1 to m do                       ----  m + 1
    for j = 1 to n do                     ----  m(n + 1)
      c[i,j] = a[i,j] + b[i,j]            ----  m.n
}                                               -------------
                     Total no of steps=  2mn + 2m + 2
```

3 Fibonacci series

```
algorithm Fibonacci(n)                        Step Count
{
  if n <= 1 then                                 ---- 1
    output 'n'
  else
    f2 = 0;                                      ---- 1
    f1 = 1;                                      ---- 1

    for i = 2 to n do                            ---- n
    {
      f = f1 + f2;                               ---- n - 1
      f2 = f1;                                   ---- n - 1
      f1 = f;                                    ---- n - 1
    }
    output 'f'                                   ---- 1
}                                                    --------
                          Total no of steps=     4n + 1
```

4 Recursive sum of elements in an array

```
algorithm RecursiveSum(a, n)                Step Count
{
  if n <= 0 then                            ---- 1
    return 0;                               ---- 1
  else
    return RecursiveSum(a, n-1) + a[n];     ---- 2 + Step Count of recursive call
}
```

- Order of Growth or Rate of Growth - A simple characterization of the algorithms efficiency by identifying relatively significant term in the step count.
- Asymptotic analysis is a technique that focuses analysis on the "significant term".

  For example, an algorithm with a step count $2n^2 + 3n + 1$, the order of growth depends on $2n^2$ for large n.

1. Big-oh Notation ($\mathcal{O}$) - To express an upper bound on the time complexity as a function of the input size.
2. Omega (Ω) - To express a lower bound on the time complexity as a function of the input size.
3. Theta (Θ) - To express the tight bound on the time complexity as a function of the input size.

Asymptotic Upper Bounds

- Big-oh Notation ($\mathcal{O}$)
- The function $f(n) = \mathcal{O}(g(n))$ if and only if there exist positive constants c, n_0 such that $f(n) \leq c \cdot g(n), \forall n \geq n_0$.
- Big-oh can be used to denote all upper bounds on the time complexity of an algorithm.
- Big-oh also captures the worst case analysis of an algorithm.

Examples

1. $3n+2$
   $3n+2 \le 4n, c=4 \;\; \forall n \ge 2 \implies \mathcal{O}(n)$
   Note that $3n+2$ is $\mathcal{O}(n^2), \mathcal{O}(n^3), \mathcal{O}(2^n), \mathcal{O}(10^n)$ as per the definition
2. $100n+6$
   $100n+6 \le 101n, c=101 \;\; \forall n \ge 6 \implies \mathcal{O}(n)$
3. $10n^2+4n+2$
   $10n^2+4n+2 \le 11n^2, c=11 \;\; \forall n \ge 5 \implies \mathcal{O}(n^2)$
4. $6 \cdot 2^n + n^2 + 2$
   $6 \cdot 2^n + n^2 + 2 \le 7 \cdot 2^n, c=7 \;\; \forall n \ge 7 \implies \mathcal{O}(2^n)$

Asymptotic Lower Bounds

- Big-Omega Notation (Ω)
- The function $f(n) = \Omega(g(n))$ if and only if there exist positive constants c, n_0 such that $f(n) \geq c.g(n), \forall n \geq n_0$.
- Omega can be used to denote all lower bounds of an algorithm.
- Omega notation also denotes the best case analysis of an algorithm.

Examples

1. $3n + 2$
   $3n + 2 \geq n, \forall n \geq 1 \Rightarrow 3n + 2 = \Omega(n)$
2. $10n^2 + 4n + 2 = \Omega(n^2)$
   $10n^2 + 4n + 2 \geq n^2,\ c = 1\ \forall n \geq 1$
3. $n^3 + n + 5 = \Omega(n^3)$
   $n^3 + n + 5 \geq n^3, c = 1, \forall n \geq 0$
4. $2n^2 + nlogn + 1 = \Omega(n^2)$
   $2n^2 + nlogn + 1 \geq 2.n^2, c = 2, \forall n \geq 1$
5. $6.2^n + n^2 = \Omega(2^n) = \Omega(n^2) = \Omega(n) = \Omega(1)$
   $6.2^n + n^2 \geq 2^n,\ c = 1\ \forall n \geq 1$

- $3n^2 + 2 \neq \Omega(n^3)$.
  Reason: There does not exist a positive constant c such that $3n^2 + 2 \geq c.n^3$ for every $n \geq n_0$ as we cannot bound n by a constant. That is, on the contrary if $3n^2 + 2 \geq c.n^3$, then $n \leq \frac{1}{c}$ is a contradiction.
- $3.2^n \neq \Omega(n!)$.
- $5 \neq \Omega(n)$.

Asymptotic Tight Bound

- Theta notation (Θ)
- The function $f(n) = \Theta(g(n))$ if and only if there exist positive constants c_1, c_2, n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0$.
- Theta can be used to denote tight bounds of an algorithm. i.e., $g(n)$ is a lower bound as well as an upper bound for $f(n)$.
- Note that $f(n) = \Theta(g(n))$ if and only if $f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$.

Examples

1 $3n + 10^{10}$
$3n \leq 3n + 10^{10} \leq 4n, \forall n \geq 10^{10} \Rightarrow 3n + 10^{10} = \Theta(n)$
Note that the first inequality captures $3n + 10^{10} = \Omega(n)$ and the later one captures $3n + 10^{10} = O(n)$.

2 $10n^2 + 4n + 2 = \Theta(n^2)$
$10n^2 \leq 10n^2 + 4n + 2 \leq 20n^2, \forall n \geq 1, c_1 = 10, c_2 = 20$

3 $6(2^n) + n^2 = \Theta(2^n)$
$6(2^n) \leq 6(2^n) + n^2 \leq 12(2^n), \forall n \geq 1, c_1 = 6, c_2 = 12$

4 $2n^2 + nlogn + 1 = \Theta(n^2)$
$2n^2 \leq 2n^2 + nlog_2 n + 1 \leq 5.n^2, \forall n \geq 2, c_1 = 2, c_2 = 5$

5 $n\sqrt{n} + nlog_2(n) + 2 = \Theta(n\sqrt{n})$
$n\sqrt{n} \leq n\sqrt{n} + nlog_2(n) + 2 \leq 5.n\sqrt{n},\ \forall n \geq 2, c_1 = 1, c_2 = 5$

Remarks

- $3n + 2 \neq \Theta(1)$. Reason: $3n + 2 \neq O(1)$
- $3n + 3 \neq \Theta(n^2)$. Reason: $3n + 3 \neq \Omega(n^2)$
- $n^2 \neq \Theta(2^n)$. Reason: $n^2 \neq \Omega(2^n)$
  Proof: Note that $f(n) \leq g(n)$ if and only if $log(f(n)) \leq log(g(n))$.
  Suppose $n^2 = \Omega(2^n)$, then by definition $n^2 \geq c.2^n$ where c is a positive constant.
  Then, $log(n^2) \geq log(c.2^n)$, and $2log(n) \geq nlog(2)$, which is a contradiction.

- The asymptotic upper bound provided by O-notation may or may not be asymptotically tight.
- The bound $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not.
- We use o-notation (Little oh) to denote an upper bound that is not asymptotically tight.
- We formally define as $f(n) = o(g(n))$ if for any positive constant $c > 0$, there exists a positive constant $n_0 > 0$ such that $0 \leq f(n) < c.g(n)$ for all $n \geq n_0$.
- Note that in the definition the inequality works for any positive constant $c > 0$.
- This is true because $g(n)$ is a loose upper bound, and hence $g(n)$ is polynomially larger than $f(n)$ by $n^{\epsilon}, \epsilon > 0$.
- Due to this n^{ϵ}, the contribution of c to the inequality is minimal which is why the quantifier in o notation is universal whereas in O is existential.

O-NOTATION EXAMPLES

1. $2n = o(n^2)$, but $2n^2 \neq o(n^2)$. Note that here n^2 is polynomially larger than $2n$ by $n^\epsilon, \epsilon = 1$.
2. $100n + 6 = o(n^{1.2})$ Here $n^{1.2}$ is polynomially larger than $100n + 6$ by $n^\epsilon, \epsilon = 0.2$. For any positive constant c, there exist n_0 such that $\forall n \geq n_0, 100n + 6 \leq c.n^{1.2}$
3. $10n^2 + 4n + 2 = o(n^3)$ Here n^3 is polynomially larger than $10n^2 + 4n + 2$ by $n^\epsilon, \epsilon = 1$
4. $6.2^n + n^2 = o(3^n)$ Note that 3^n is $1.5^n \times 2^n$. So for any $c > 0$, $2^n \leq c.3^n$. The value of c is insignificant as 1.5^n dominates any $c > 0$.
5. $3n + 3 = o(n^{1.00001})$ Here $n^{1.00001}$ is polynomially larger than $3n + 3$ by $n^\epsilon, \epsilon = 0.00001$
6. $n^3 + n + 5 = o(n^{3.1})$ Here $n^{3.1}$ is polynomially larger than $n^3 + n + 5$ by $n^\epsilon, \epsilon = 0.1$

- We use ω-notation to denote a lower bound that is not asymptotically tight.
- We define $\omega(g(n))$ (little-omega) as $f(n) = \omega(g(n))$ if for any positive constant $c > 0$, there exists a positive constant $n_0 > 0$ such that $0 \leq c.g(n) < f(n)$ for all $n \geq n_0$.

1. $n^2 = \omega(n)$ but $n^2 \neq \omega(n^2)$.
2. $3n + 2 = \omega(log(n))$
3. $10n^3 + 4n + 2 = \omega(n^2)$
4. $5n^6 + 7n + 9 = \omega(n^3)$
5. $2n^2 + nlogn + 1 = \omega(n^{1.9999999})$
6. $15 \times 3^n + n^2 = \omega(2^n) = \omega(n^2) = \omega(n) = \omega(1)$

- In o-notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity; that is, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
- The relation $f(n) = \omega(g(n))$ implies that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, if the limit exists.
  That is, $f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity.

1. Reflexivity :

$f(n) = O(f(n)) \qquad f(n) = \Omega(f(n)) \qquad f(n) = \theta(f(n))$

2. Symmetry :

$f(n) = \theta(g(n))$ if and only if $g(n) = \theta(f(n))$
Proof:
Necessary part: $f(n) = \theta(g(n)) \Rightarrow g(n) = \theta(f(n))$
By the definition of θ , there exists positive constants c_1, c_2, n_o such that $c_1.g(n) \leq f(n) \leq c_2.g(n)$ for all $n \geq n_o$

$\Rightarrow g(n) \leq \frac{1}{c_1}.f(n)$ and $g(n) \geq \frac{1}{c_2}.f(n)$

$\Rightarrow \frac{1}{c_2}f(n) \leq g(n) \leq \frac{1}{c_1}f(n)$

Since c_1 and c_2 are positive constants, $\frac{1}{c_1}$ and $\frac{1}{c_2}$ are well defined.

Therefore, by the definition of θ , $g(n) = \theta(f(n))$

Sufficiency part: $g(n) = \theta(f(n)) \Rightarrow f(n) = \theta(g(n))$

By the definition of θ , there exists positive constants c_1, c_2, n_o such that $c_1.f(n) \leq g(n) \leq c_2.f(n)$ for all $n \geq n_o$
$\Rightarrow f(n) \leq \frac{1}{c_1}.g(n)$ and $f(n) \geq \frac{1}{c_2}.g(n)$
$\Rightarrow \frac{1}{c_2}.g(n) \leq f(n) \leq \frac{1}{c_1}.g(n)$
By the definition of θ , $f(n) = \theta(g(n))$ This completes the proof of Symmetry property.

3. Transitivity :

$f(n) = O(g(n))$ and $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$

Proof:

$f(n) = O(g(n))$ and $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$

By the definition of Big-Oh(O) , there exists positive constants c, n_o such that $f(n) \leq c.g(n)$ for all $n \geq n_o$

$\Rightarrow f(n) \leq c_1.g(n)$

$\Rightarrow g(n) \leq c_2.h(n)$

$\Rightarrow f(n) \leq c_1.c_2 h(n)$

$\Rightarrow f(n) \leq c.h(n)$, where, $c = c_1.c_2$

Properties of Asymptotic notation

4. Transpose Symmetry:

$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$
Proof:
Necessity: $f(n) = O(g(n)) \Rightarrow g(n) = \Omega(f(n))$

By the definition of Big-Oh (O)

$\Rightarrow f(n) \leq c.g(n)$ for some positive constant c

$$\Rightarrow g(n) \geq \frac{1}{c} f(n)$$

By the definition of Omega (Ω) , $g(n) = \Omega(f(n))$

Sufficiency: $g(n) = \Omega(f(n)) \Rightarrow f(n) = O(g(n))$

Lemma

Let $f(n)$ and $g(n)$ be two asymptotic non-negative functions. Then, $max(f(n), g(n)) = \theta(f(n) + g(n))$

Proof.

Without loss of generality, assume $f(n) \leq g(n)$,
$\Rightarrow max(f(n), g(n)) = g(n)$
Consider, $g(n) \leq max(f(n), g(n)) \leq g(n)$

$\Rightarrow g(n) \leq max(f(n), g(n)) \leq f(n) + g(n)$

$\Rightarrow \frac{1}{2}g(n) + \frac{1}{2}g(n) \leq max(f(n), g(n)) \leq f(n) + g(n)$

From what we assumed,we can write
$\Rightarrow \frac{1}{2}f(n) + \frac{1}{2}g(n) \leq max(f(n), g(n)) \leq f(n) + g(n)$

$\Rightarrow \frac{1}{2}(f(n) + g(n)) \leq max(f(n), g(n)) \leq f(n) + g(n)$

By the definition of θ ,
$max(f(n), g(n)) = \theta(f(n) + g(n))$ $\square$

Lemma

For two asymptotic functions $f(n)$ and $g(n)$,
$O(f(n)) + O(g(n)) = O(max(f(n), g(n)))$

Proof.

Without loss of generality, assume $f(n) \leq g(n)$

$\Rightarrow O(f(n)) + O(g(n)) = c_1 f(n) + c_2 g(n)$

From what we assumed,we can write

$O(f(n)) + O(g(n)) \leq c_1 g(n) + c_2 g(n)$

$\leq (c_1 + c_2) g(n) \leq \text{c } g(n)$

$\leq \text{c } max(f(n), g(n))$

By the definition of Big-Oh(O),

$O(f(n)) + O(g(n)) = O(max(f(n), g(n)))$

$\square$

- If $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = c, c \in \mathbb{R}^+$ then $f(n) = \theta(g(n))$
- If $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} \leq c, c \in \mathbb{R}$ (c can be 0) then $f(n) = O(g(n))$
- If $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = 0,$ then $f(n) = O(g(n))$ and $g(n) \neq O(f(n))$
- If $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} \geq c, c \in \mathbb{R}$($c$ can be ∞) then $f(n) = \Omega(g(n))$
- If $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = \infty,$ then $f(n) = \Omega(g(n))$ and $g(n) \neq \Omega(f(n))$
- **L$'$Hôpital Rule** :

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

where $f'(n)$ and $g'(n)$ are differentiates of $f(n)$ and $g(n)$.

Lemma

Show that $\log n = O(\sqrt{n})$*, however,* $\sqrt{n} \neq O(\log n)$

Proof.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\log n}{\sqrt{n}}$$

Applying L$'$Hôpital Rule,

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

$$= \lim_{n\to\infty} \frac{2}{\sqrt{n}} = 0$$

From Remark 3, $f(n) = O(g(n)) => \log n = O(\sqrt{n})$.

contd... $\square$

Proof.

Proof for $\sqrt{n} \neq O(\log n)$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\sqrt{n}}{\log n}$$

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

$$= \lim_{n\to\infty} \frac{\sqrt{n}}{2} = \infty$$

From Remark 3,
$f(n) = \Omega(g(n)) \Rightarrow \sqrt{n} = \Omega(\log n). \Rightarrow \sqrt{n} \neq O(\log n)$ $\square$