Design and Analysis of Algorithms CS218M

Asymptotic Complexity

Paritosh Pandya

Indian Institute of Technology, Bombay

Autumn, 2022

Resources

Textbook

- Introduction to Algorithms, T.H. Cormen, C.E. Leicserson, R.L. Rivest, C. Stein, Third Edition, PHI, 2014. CLRS

Course Webpage

Course URL: https://cse.iitb.ac.in/~pandya58/CS218M/algo.html

Reference for Today's Class: CLRS Ch 2.1,2.2 and Ch 3.1

Self Study Topics: CLRS Ch 3.2

Lectures
Tutorials
Quizes
Assignment
Midterm
EndSem
} Exams.

} 10%

} 40%

50%

Recap:

- Algorithm V/s Program
  Prog = (Code, PL, Compiler, OS, Machine)
- Analysis of Algo
  - Correctness
  - Efficiency
- Design Paradigms
- Inherent Complexity of Problems
  Complexity Classes
  - NP-Complete class

5 S

5

Pseudo Code Language

{A array of int, x : int}

Procedure

```
Linear_Search(A,x):
    found=false
    for i from 1 to n
        if A[i]==x
            found=true
            return
```

- Indentation gives the block structure.

Pseudo Code Language

```
Linear_Search(A,x):
    found=false
    for i from 1 to n
        if A[i]==x
            found=true
            return
```

- Indentation gives the block structure.
- Control structures: `if-else`, `while`, `for`, `repeat-until`

Pseudo Code Language

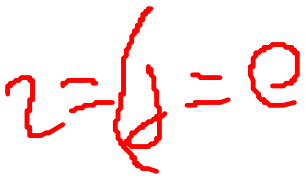
```
Linear_Search(A,x):
    found=false
    for i from 1 to n
        if A[i]==x
            found=true
            return
```

- Indentation gives the block structure.
- Control structures: `if-else`, `while`, `for`, `repeat-until`
- Comments `// this is a comment.`

Pseudo Code Language

```
Linear_Search(A,x):
    found=false
    for i from 1 to n
        if A[i]==x
            found=true
            return
```

- Indentation gives the block structure.
- Control structures: `if-else`, `while`, `for`, `repeat-until`
- Comments `// this is a comment.`
- Expressions and comditions.
  Condition $x + 1 > y$
  Assignments `i = e`, and multiple assignments `i,j = j,i`.



Pseudo Code Language (2)

- All variables are local to procedures.

Pseudo Code Language (2)

- All variables are local to procedures.
- Primitive types integers, booleans, ...

Pseudo Code Language (2)

- All variables are local to procedures.
- Primitive types integers, booleans, ...
- Objects and arrays are of reference type.

Pseudo Code Language (2)

- All variables are local to procedures.
- Primitive types integers, booleans, ...
- Objects and arrays are of reference type.
- Object have attributes: E.g. `f` has attribute `f.x`

Pseudo Code Language (2)

- All variables are local to procedures.
- Primitive types integers, booleans, ...
- Objects and arrays are of reference type.
- Object have attributes: E.g. `f` has attribute `f.x`
- Array `A[1..n]`
  Array length attribute `A.Length`
  Array slice `A[i..j]` denotes list (slice) of elements `A[i]` to `A[j]`
  Example: Let `A=[5,4,3,2,1]`. Then, `A[2..4]=[4,3,2]`.

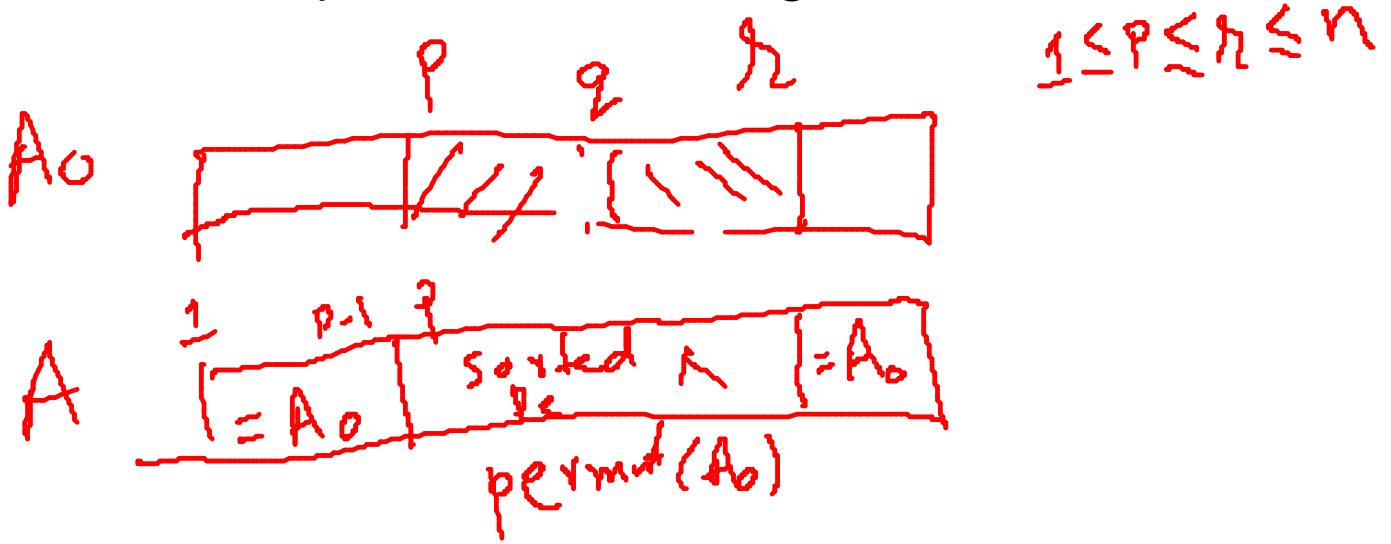
Pseudo Code Language (3)

{Pre: $p \le r$, $A = A_0$}

{Post: Permut($A[p..r]$, $A_0[p..r]$) $\wedge$ Sorted($A[p..r]$) $\wedge$ $A[1..p-1] = A_0[1..p-1]$}

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

- Procedures with Parameters. The `return e1,e2,e3` statement will exit the procedure returning three values.



Pseudo Code Language (3)

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

- Procedures with Parameters. The `return e1,e2,e3` statement will exit the procedure returning three values.
- Parameter passing for primitive types is **by value**

Pseudo Code Language (3)

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

- Procedures with Parameters. The `return e1,e2,e3` statement will exit the procedure returning three values.
- Parameter passing for primitive types is **by value**
- Parameter passing of Arrays and Objects is **by reference**

Pseudo Code Language (3)

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

- Procedures with Parameters. The `return e1,e2,e3` statement will exit the procedure returning three values.
- Parameter passing for primitive types is **by value**
- Parameter passing of Arrays and Objects is **by reference**
- Recursive procedure invocations are permitted.

Pseudo Code Language (3)

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

- Procedures with Parameters. The `return e1,e2,e3` statement will exit the procedure returning three values.
- Parameter passing for primitive types is **by value**
- Parameter passing of Arrays and Objects is **by reference**
- Recursive procedure invocations are permitted.
- Boolean operators `and`, `or` are short-circuiting.
  E.g. boolean condition `x!=nil and x.f=y`

Model of Computation: RAM

Random Access Memory Model.

- Memory is organized as words.

Model of Computation: RAM

Random Access Memory Model.

- Memory is organized as words.
- Each primitive type variable (int, bool,...) is stored in one word.

Model of Computation: RAM

Random Access Memory Model.

- Memory is organized as words.
- Each primitive type variable (int, bool,...) is stored in one word.
- Variable access (load, store) takes constant time.

Model of Computation: RAM

Random Access Memory Model.

- Memory is organized as words.
- Each primitive type variable (int, bool,...) is stored in one word.
- Variable access (load, store) takes constant time.
- Each primitive operation takes constant time.

Model of Computation: RAM

Random Access Memory Model.

- Memory is organized as words.
- Each primitive type variable (int, bool,...) is stored in one word.
- Variable access (load, store) takes constant time.
- Each primitive operation takes constant time.
- Program execution consists of a sequence of loads, stores, primitive operations (e.g. +,*, and) as specified by the pseudo code control flow.

Model of Computation: RAM

Random Access Memory Model.

- Memory is organized as words.
- Each primitive type variable (int, bool,...) is stored in one word.
- Variable access (load, store) takes constant time.
- Each primitive operation takes constant time.
- Program execution consists of a sequence of loads, stores, primitive operations (e.g. +,*, and) as specified by the pseudo code control flow.
- Each array access takes constant time.

Model of Computation: RAM

Random Access Memory Model.

- Memory is organized as words.
- Each primitive type variable (int, bool,...) is stored in one word.
- Variable access (load, store) takes constant time.
- Each primitive operation takes constant time.
- Program execution consists of a sequence of loads, stores, primitive operations (e.g. +,*, and) as specified by the pseudo code control flow.
- Each array access takes constant time.
- There is no concurrency. We do not have cache memory, paging, pipelining etc.

Modelling Execution time of an Algorithm

{A, B array[1..n][1..n] of int}

cost count

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows                          c1   1
2  let C be a new n × n matrix         c2   1
3  for i = 1 to n                      c3   n
4      for j = 1 to n                  c4   n^2
5          c_ij = 0                    c5   n^2
6          for k = 1 to n              c6   n^3
7              c_ij = c_ij + a_ik · b_kj   c7   n^3
8  return C
```

Growth Function

$$T(n) = c_1 + c_2 + c_3 n + c_4 n^2 + c_5 n^2 + c_6 n^3 + c_7 n^3$$

$$= (c_6 + c_7) n^3 + (c_4 + c_5) n^2 + c_3 n + (c_1 + c_2)$$

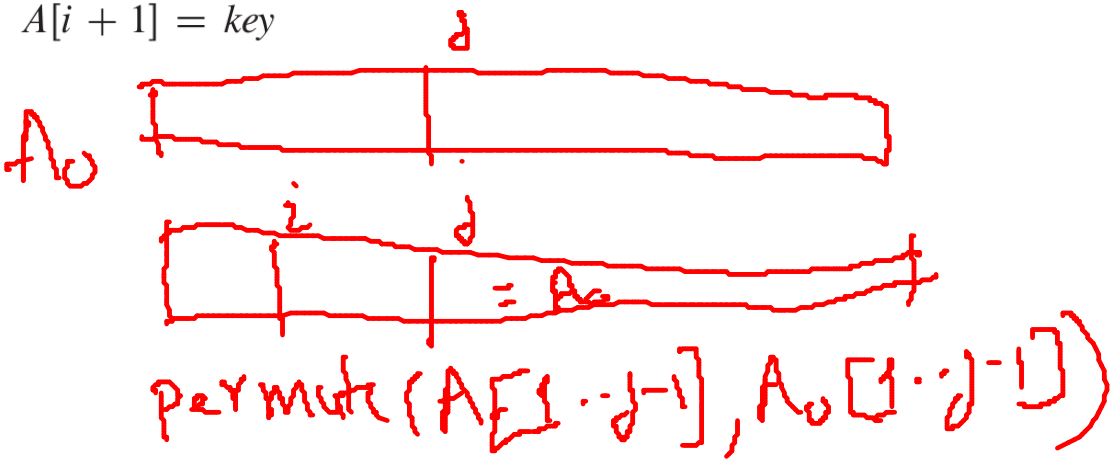
$$= an^3 + bn^2 + cn + d$$

Time Complexity of Insertion Sort Algorithm

INSERTION-SORT(A)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Cost	Count
c_1	n
c_2	$n-1$
0	$n-1$
c_4	$n-1$
c_5	t_j



Time Complexity of Insertion Sort Algorithm

INSERTION-SORT(A)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Growth Function

Time Complexity of Insertion Sort Algorithm (2)

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

Time Complexity of Insertion Sort Algorithm (2)

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] = key$	c_8	$n-1$

Growth Function in General Case

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j-1) + c_7\sum_{j=2}^{n}(t_j-1) + c_8(n-1)\,.$$

Best Case Execution Time of Insertion Sort

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

Best Case Execution Time of Insertion Sort

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

Growth Function in Best Case

- Array is already sorted in correct order.
- While loop terminates immediately $t_j = 1$.

Time Complexity of Insertion Sort Algorithm (4)

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\
&\quad + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \, .
\end{aligned}
$$

Time Complexity of Insertion Sort Algorithm (4)

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\
&\quad + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \, .
\end{aligned}
$$

Best Case Execution Time $t_j = 1$

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\
&= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \, .
\end{aligned}
$$

$= an + b$

Worst Case Execution Time of Insertion Sort

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

Worst Case Execution Time of Insertion Sort

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

Growth Function in Worst Case

- Array is already sorted in reverse order.
- While loop in line five executes j times. Hence $t_j = j$.

Time Complexity of Insertion Sort Algorithm (4)

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\
&\quad + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \, .
\end{aligned}
$$

Time Complexity of Insertion Sort Algorithm (4)

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

Worst Case Execution Time

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) \\ &\quad + c_6 \left(\frac{n(n-1)}{2} \right) + c_7 \left(\frac{n(n-1)}{2} \right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

$= an^2 + bn + c$

Asymptotic Order of Growth

The efficiency of two algorithms with growth functions $T_1(n)$ and $T_2(n)$:

- Asymptotic growth: We compare $T_1(n)$ and $T_2(n)$ as n grows large.

Asymptotic Order of Growth

The efficiency of two algorithms with growth functions $T_1(n)$ and $T_2(n)$:

- Asymptotic growth: We compare $T_1(n)$ and $T_2(n)$ as n grows large.
- Only the highest order terms dominate.
  Simplify $a * n^2 + b * n + c$ to $a * n^2$. (why?)

Asymptotic Order of Growth

The efficiency of two algorithms with growth functions $T_1(n)$ and $T_2(n)$:

- Asymptotic growth: We compare $T_1(n)$ and $T_2(n)$ as n grows large.
- Only the highest order terms dominate.
  Simplify $a * n^2 + b * n + c$ to $a * n^2$. (why?)
- Constant of the highest order term is less important as n grows large. Simplify $a * n^2 + b * n + c$ to its order of growth $\Theta(n^2)$.

Asymptotic Order of Growth

The efficiency of two algorithms with growth functions $T_1(n)$ and $T_2(n)$:

- Asymptotic growth: We compare $T_1(n)$ and $T_2(n)$ as n grows large.
- Only the highest order terms dominate.
  Simplify $a * n^2 + b * n + c$ to $a * n^2$. (why?)
- Constant of the highest order term is less important as n grows large. Simplify $a * n^2 + b * n + c$ to its order of growth $\Theta(n^2)$.

Example

Let $T_1(n) = 10^4 * n^2 + 10^3 * n + 10^6$ and $T_2(n) = 10^{-4} * n^4$.

$\Theta(T_1(n)) = n^2$ $\quad$ $\Theta(T_2(n)) = n^4$

$n = 10^6$

Insertion Sort: Simplified Worst Case Execution Time Analysis

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

$\theta(n)$ $\theta(n)$ $\theta(n)$ $\theta(n)$ $\theta(n^2)$ $\theta(n^2)$ $\theta(n)$

$\theta(T(n)) = \theta(n^2)$

Growth Function in Worst Case

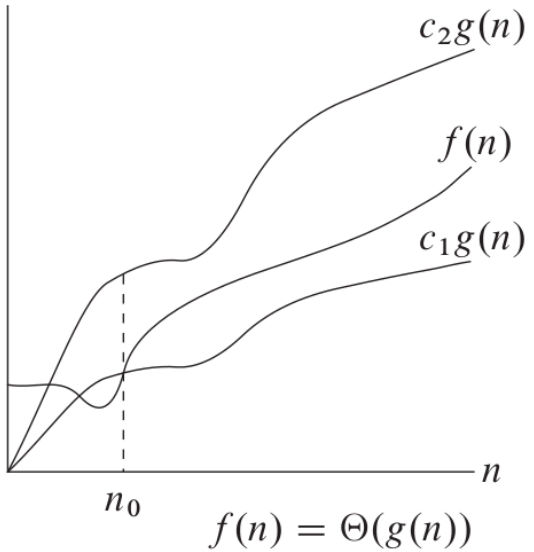
- Array is already sorted in reverse order.Hence $t_j = j$.
- We keep only the maximum order term.
- We disregard constants.

Order of Growth: Mathematical Definition

Big Theta

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}\ .^{1}$$

- We write $f(n) = \Theta(g(n))$ instead of $f(n) \in \Theta(g(n))$
- Pronounced $g(n)$ is asymptotically a tight bound for $f(n)$ OR $f(n)$ is of order Θ of $g(n)$.



Examples

To show $f(n) = \Theta(g(n))$ choose positive c_1, c_2 and n_0 such that for all $n > n_0$ we have $0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

- Show that $(1/2) * n^2 - 3^n = \Theta(n^2)$
- Show that $6(n^3) \neq \Theta(n^2)$

$3n$

$\exists c_1, c_2, n_0 > 0$ s.t.

$\forall n > n_0$

$c_1 n^2 \leq \frac{1}{2} n^2 - 3n \leq c_2 n^2$

$\forall n > n_0$

$0 \leq c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$

$n_0 = 6$

$\frac{1}{14} \leq \frac{1}{2} - \frac{3}{n} \leq \frac{1}{2}$

$6n^3 \neq \Theta(n^2)$

Assume to contrary.

$\exists c_2 > 0, n_0 > 0$ s.t.

$6n^3 \leq c_2 n^2 \quad \forall n \geq n_0$

$\Rightarrow 6n \leq c_2 \quad \forall n \geq n_0$

Contradiction.

Asymptotic Upper and Lower Bounds

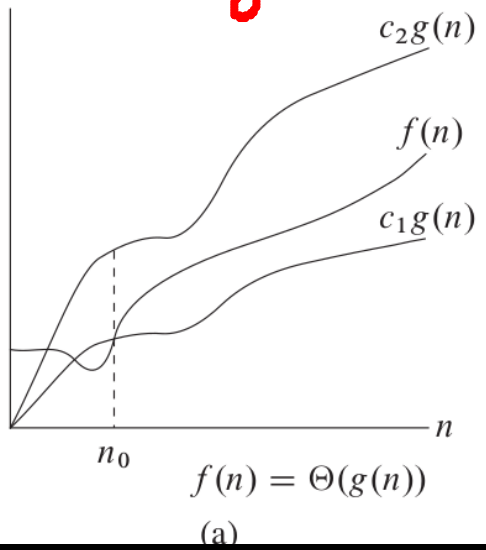
Asymptotic Upper Bound

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$

Asymptotic Lower Bound

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}.$$

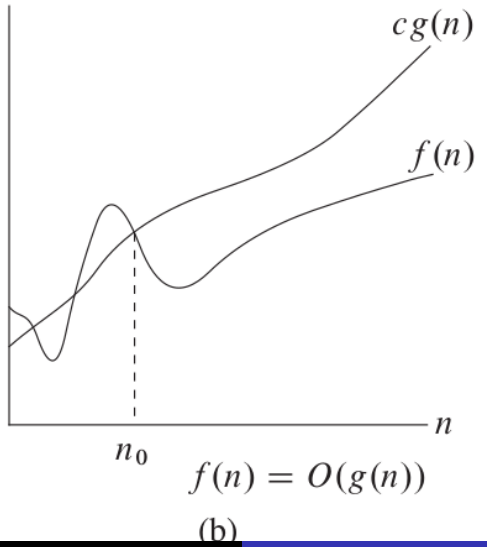
Tight



$f(n) = \Theta(g(n))$

(a)

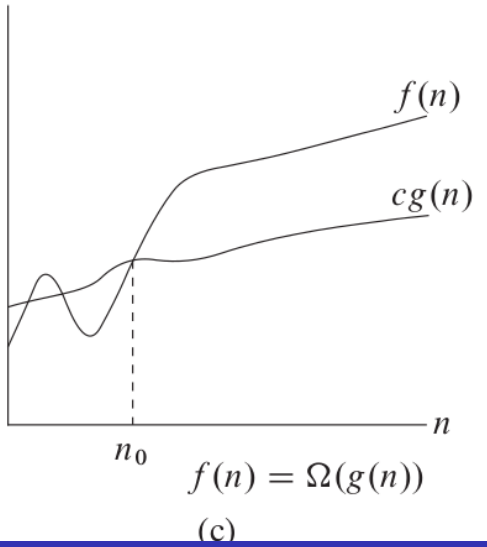
Upper



$f(n) = O(g(n))$

(b)

Lower



$f(n) = \Omega(g(n))$

(c)

Insertion Sort

$O(n^2)$ $\Omega(n)$

Properties

- $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
- Symmetry: $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$
- Reflexivity, Transitivity of $= \Theta$, $= O$ and $= \Omega$.

$f(n) = \Theta(f(n))$ reflexivity

$f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n))$

$\Rightarrow f(n) = \Theta(h(n))$

$n^2 = O(n^4)$

$n^4 \neq O(n^2)$

Standard Mathematical Functions and their Properties

For Self Study.

$\Theta(n^2)$ dominated by $\Theta(n^3)$

$O(n^2)$

CLRS Section 3.2