Introduction to Analysis of Algorithms

Partha P Chakrabarti
Indian Institute of Technology Kharagpur

Overview

- ❑ Algorithms and Programs
- ❑ Pseudo-Code
- ❑ Algorithms + Data Structures = Programs
- ❑ Initial Solutions + Analysis + Solution Refinement + Data Structures = Final Algorithm
- ❑ Use of Recursive Definitions as Initial Solutions
- ❑ Recurrence Equations for Proofs and Analysis
- ❑ Solution Refinement through Recursion Transformation and Traversal
- ❑ Data Structures for saving past computation for future use

<u>Sample Problems:</u>

1. Finding the Largest
2. Largest and Smallest
3. Largest and Second Largest
4. Fibonacci Numbers
5. Searching for an element in an ordered / unordered List
6. Sorting
7. Pattern Matching
8. Permutations and Combinations
9. Layout and Routing
10. Shortest Paths

Time and Space Complexity of an Algorithm

Time Complexity is an estimate of the Execution Time of an Algorithm

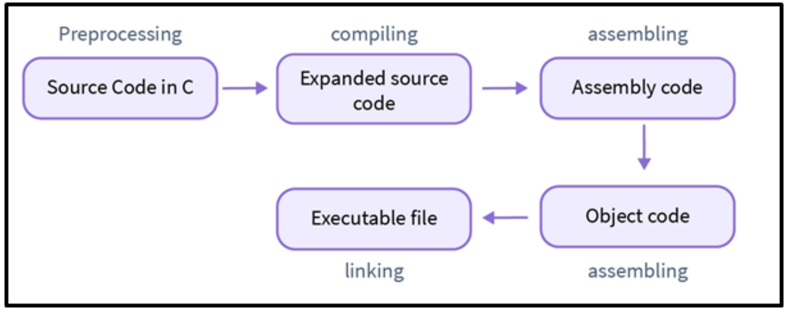
- ❑ Varies for individual computer architectures, compilers used to implement the algorithm
- ❑ Varies as size of input varies, as well as for different inputs of the same size, etc.

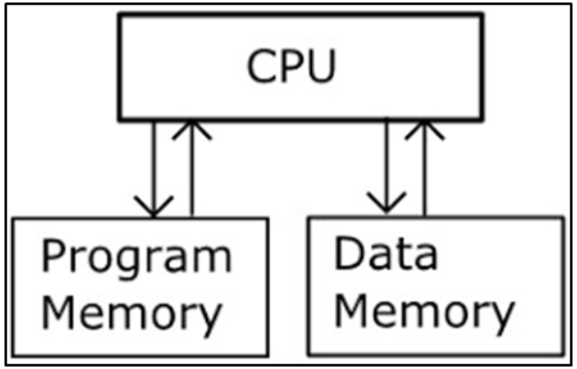
Space Complexity is an estimate of the Memory Requirement of a Program

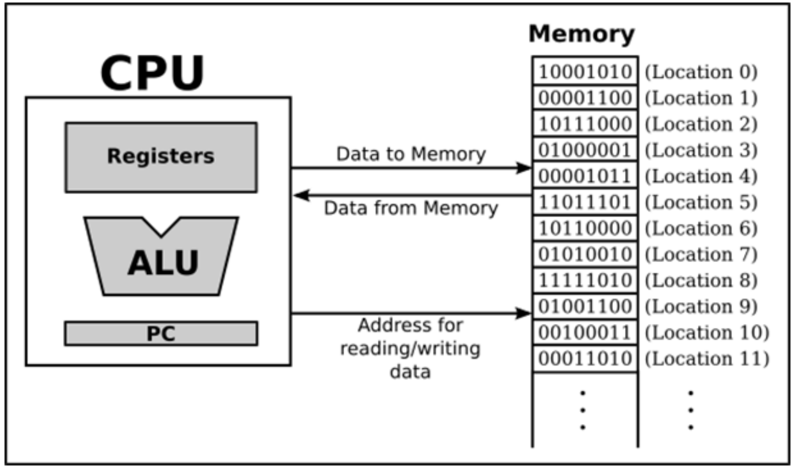
- ➢ Depends on how code and data are stored in different architectures and what happens when function calls are made, handling of global and local variables, etc.
- ➢ Access speeds of data also affect Time Complexity

How do we develop a measure that can be used to determine the time efficiency of an algorithm independently and compare various algorithms for the same problem across input sizes?

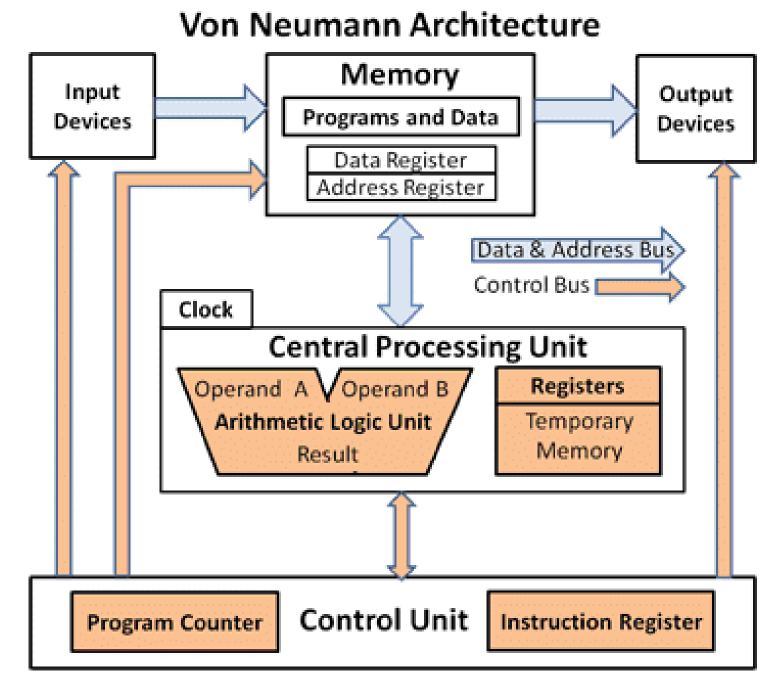
Conversion of High Level Programs

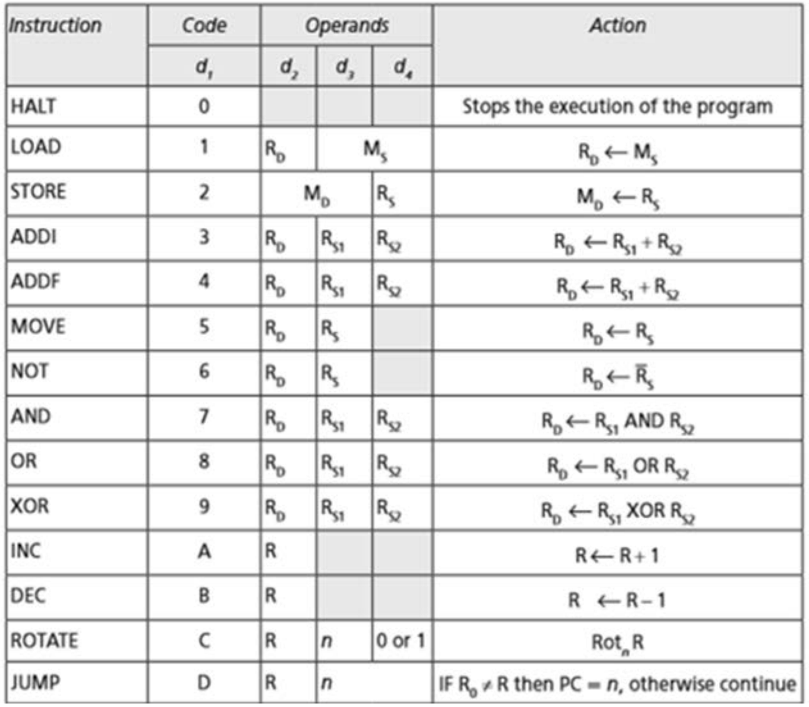






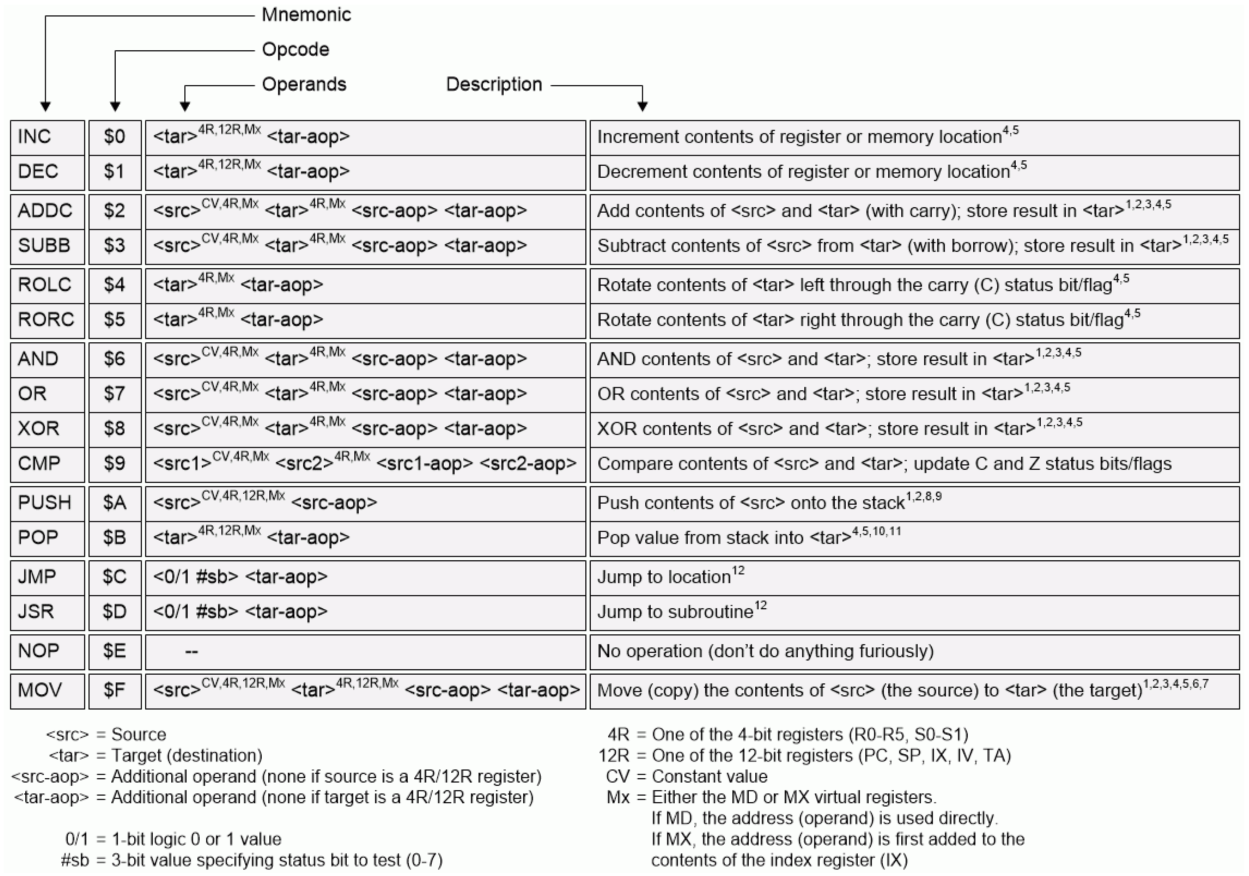
Computer Architecture, Sample Instruction Set





Instruction	*Code*	*Operands*			*Action*
	d_1	d_2	d_3	d_4	
HALT	0				Stops the execution of the program
LOAD	1	R_D	M_S		$R_D \leftarrow M_S$
STORE	2	M_D		R_S	$M_D \leftarrow R_S$
ADDI	3	R_D	R_{S1}	R_{S2}	$R_D \leftarrow R_{S1} + R_{S2}$
ADDF	4	R_D	R_{S1}	R_{S2}	$R_D \leftarrow R_{S1} + R_{S2}$
MOVE	5	R_D	R_S		$R_D \leftarrow R_S$
NOT	6	R_D	R_S		$R_D \leftarrow \overline{R}_S$
AND	7	R_D	R_{S1}	R_{S2}	$R_D \leftarrow R_{S1}$ AND R_{S2}
OR	8	R_D	R_{S1}	R_{S2}	$R_D \leftarrow R_{S1}$ OR R_{S2}
XOR	9	R_D	R_{S1}	R_{S2}	$R_D \leftarrow R_{S1}$ XOR R_{S2}
INC	A	R			$R \leftarrow R + 1$
DEC	B	R			$R \leftarrow R - 1$
ROTATE	C	R	*n*	0 or 1	$\text{Rot}_n R$
JUMP	D	R	*n*		IF $R_0 \neq R$ then PC = *n*, otherwise continue

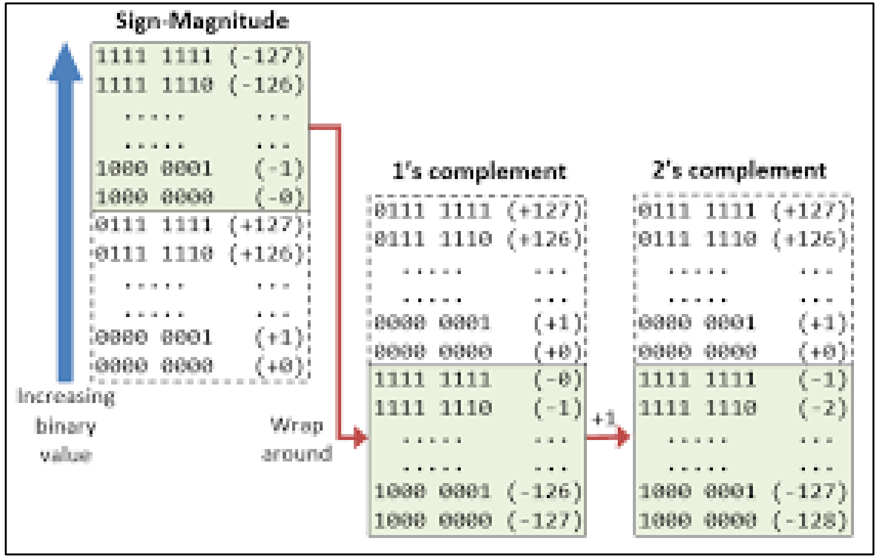
Another Instruction Set with Format

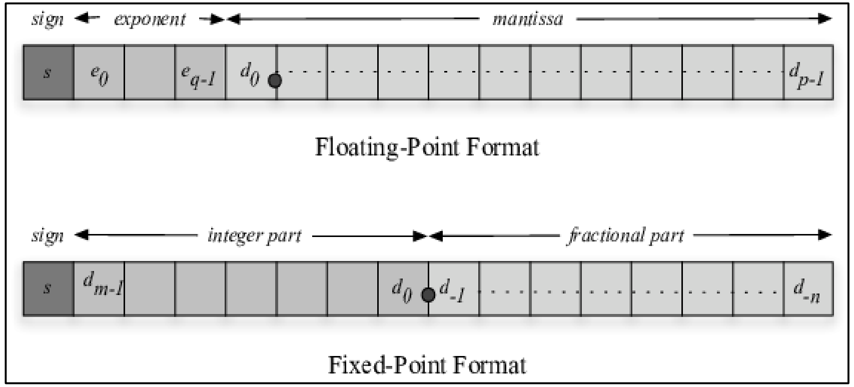


Mnemonic	Opcode	Operands	Description
INC	$0	<tar>^{4R,12R,Mx} <tar-aop>	Increment contents of register or memory location[4,5]
DEC	$1	<tar>^{4R,12R,Mx} <tar-aop>	Decrement contents of register or memory location[4,5]
ADDC	$2	<src>^{CV,4R,Mx} <tar>^{4R,Mx} <src-aop> <tar-aop>	Add contents of <src> and <tar> (with carry); store result in <tar>[1,2,3,4,5]
SUBB	$3	<src>^{CV,4R,Mx} <tar>^{4R,Mx} <src-aop> <tar-aop>	Subtract contents of <src> from <tar> (with borrow); store result in <tar>[1,2,3,4,5]
ROLC	$4	<tar>^{4R,Mx} <tar-aop>	Rotate contents of <tar> left through the carry (C) status bit/flag[4,5]
RORC	$5	<tar>^{4R,Mx} <tar-aop>	Rotate contents of <tar> right through the carry (C) status bit/flag[4,5]
AND	$6	<src>^{CV,4R,Mx} <tar>^{4R,Mx} <src-aop> <tar-aop>	AND contents of <src> and <tar>; store result in <tar>[1,2,3,4,5]
OR	$7	<src>^{CV,4R,Mx} <tar>^{4R,Mx} <src-aop> <tar-aop>	OR contents of <src> and <tar>; store result in <tar>[1,2,3,4,5]
XOR	$8	<src>^{CV,4R,Mx} <tar>^{4R,Mx} <src-aop> <tar-aop>	XOR contents of <src> and <tar>; store result in <tar>[1,2,3,4,5]
CMP	$9	<src1>^{CV,4R,Mx} <src2>^{4R,Mx} <src1-aop> <src2-aop>	Compare contents of <src> and <tar>; update C and Z status bits/flags
PUSH	$A	<src>^{CV,4R,12R,Mx} <src-aop>	Push contents of <src> onto the stack[1,2,8,9]
POP	$B	<tar>^{4R,12R,Mx} <tar-aop>	Pop value from stack into <tar>[4,5,10,11]
JMP	$C	<0/1 #sb> <tar-aop>	Jump to location[12]
JSR	$D	<0/1 #sb> <tar-aop>	Jump to subroutine[12]
NOP	$E	--	No operation (don't do anything furiously)
MOV	$F	<src>^{CV,4R,12R,Mx} <tar>^{4R,12R,Mx} <src-aop> <tar-aop>	Move (copy) the contents of <src> (the source) to <tar> (the target)[1,2,3,4,5,6,7]

<src>	= Source	4R	= One of the 4-bit registers (R0-R5, S0-S1)
<tar>	= Target (destination)	12R	= One of the 12-bit registers (PC, SP, IX, IV, TA)
<src-aop>	= Additional operand (none if source is a 4R/12R register)	CV	= Constant value
<tar-aop>	= Additional operand (none if target is a 4R/12R register)	Mx	= Either the MD or MX virtual registers. If MD, the address (operand) is used directly. If MX, the address (operand) is first added to the contents of the index register (IX)
0/1	= 1-bit logic 0 or 1 value		
#sb	= 3-bit value specifying status bit to test (0-7)		

Number Representation





Asymptotic Analysis of Algorithms

- ❑ All data is stored in finite sized locations bounded by a constant.
- ❑ All instructions take finite amount of time that is bounded by a constant.
- ❑ If the number of instructions executed are around the same, but speeds of the computer are different, then their comparative execution times will be bounded by a constant.
- ❑ To make it computer independent, we assume that each operation of an algorithm takes unit time.
- ❑ We are interested to compare algorithms when they take large amounts of time.

ASYMPTOTIC ANALYSIS

- ❑ An approximate measure that tries to estimate the growth pattern of the execution time of an algorithm with respect to increased input / output size.
- ❑ We wish to analyze upper and lower limits of this growth.
- ❑ We wish to look at Worst Case, Average Case Growth Performance of the algorithm

Approximation of Growth

```
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
        if (a[i] + a[j] == 0)
            count++;
```

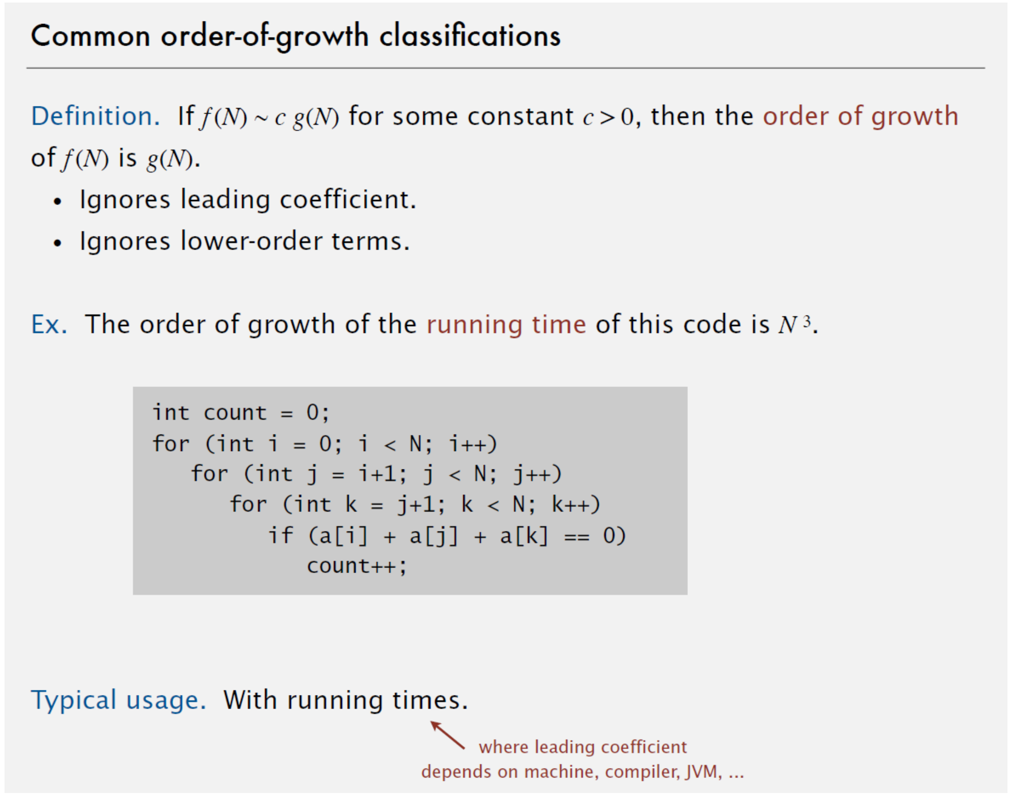
- Estimate running time (or memory) as a function of input size N.
- Ignore lower order terms.
  - when N is large, terms are negligible
  - when N is small, we don't care

Operation	Frequency	Approximation
variable declaration	$N + 2$	$\sim N$
assignment statement	$N + 2$	$\sim N$
less than compare	$\frac{1}{2}(N+1)(N+2)$	$\sim \frac{1}{2}N^2$
equal to compare	$\frac{1}{2}N(N-1)$	$\sim \frac{1}{2}N^2$
array access	$N(N-1)$	$\sim N^2$
increment	$\frac{1}{2}N(N-1)$ to $N(N-1)$	$\sim \frac{1}{2}N^2$ to $\sim N^2$

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

$$\lim_{N\to\infty} \frac{f(n)}{g(n)} = 1$$

Order of Growth



Common order-of-growth classifications

Definition. If $f(N) \sim c\, g(N)$ for some constant $c > 0$, then the order of growth of $f(N)$ is $g(N)$.

- Ignores leading coefficient.
- Ignores lower-order terms.

Ex. The order of growth of the running time of this code is N^3.

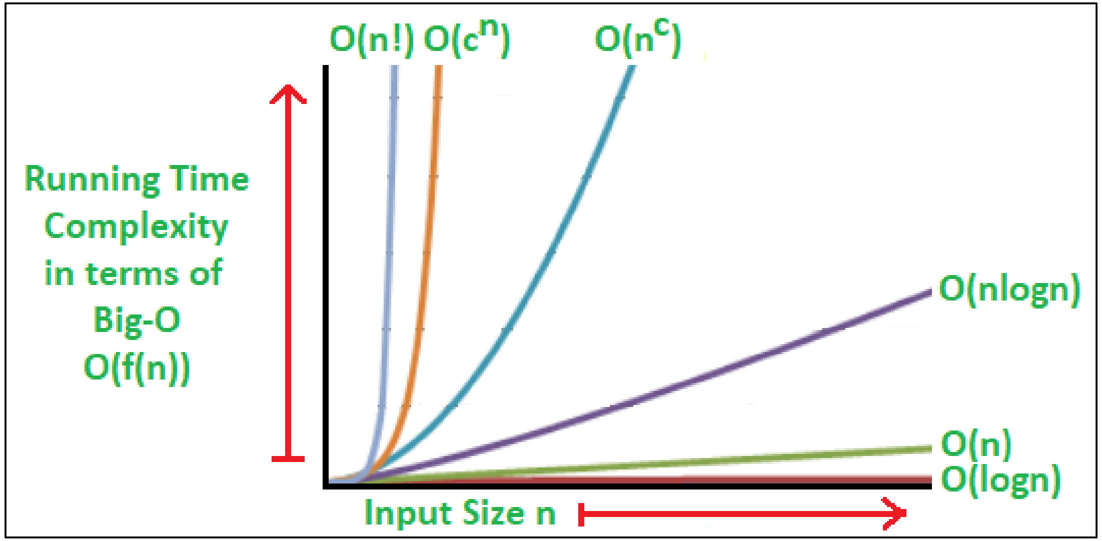
```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      for (int k = j+1; k < N; k++)
         if (a[i] + a[j] + a[k] == 0)
            count++;
```

Typical usage. With running times.

where leading coefficient depends on machine, compiler, JVM, ...

Courtesy: Algorithms by Robert Sedgewick & Kevin Wayne

Order of Growth: Graphically



$O(n!), O(c^n), O(n^c)$ - Worst

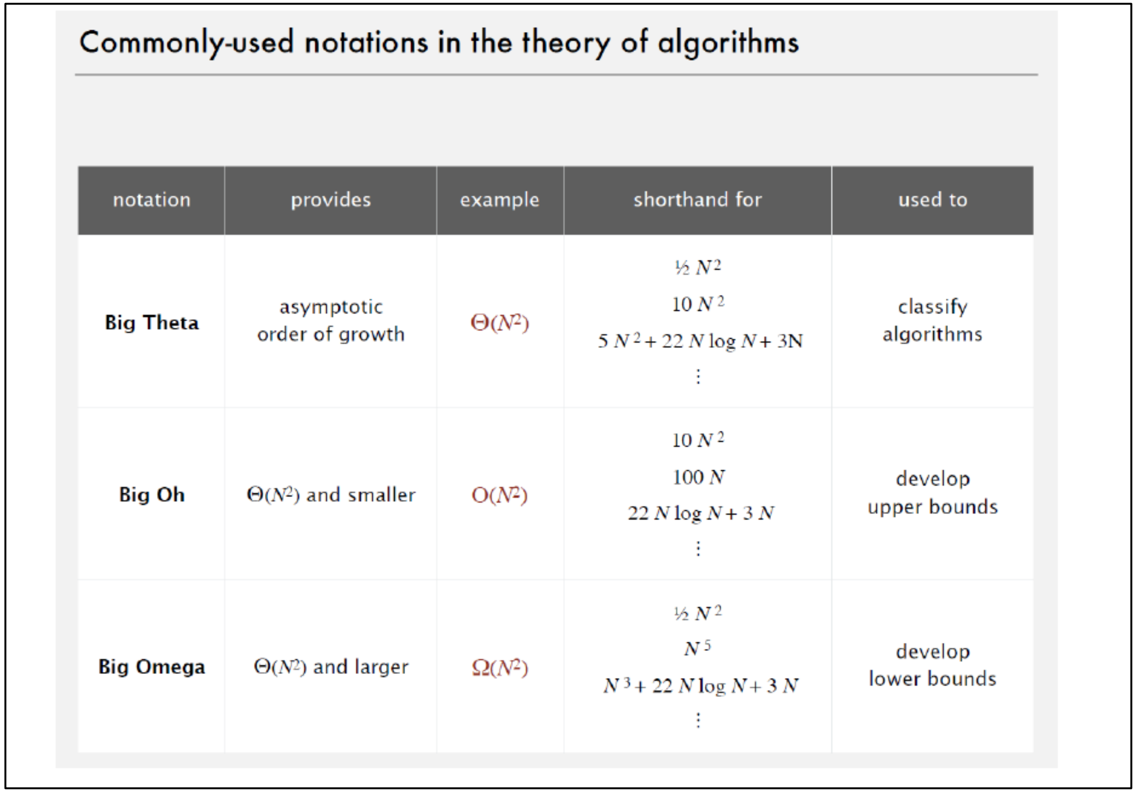
$O(n\log n)$ - Bad

$O(n)$ - Fair

$O(\log n)$ - Good

$O(1)$ - Best

Notations for Order of Growth

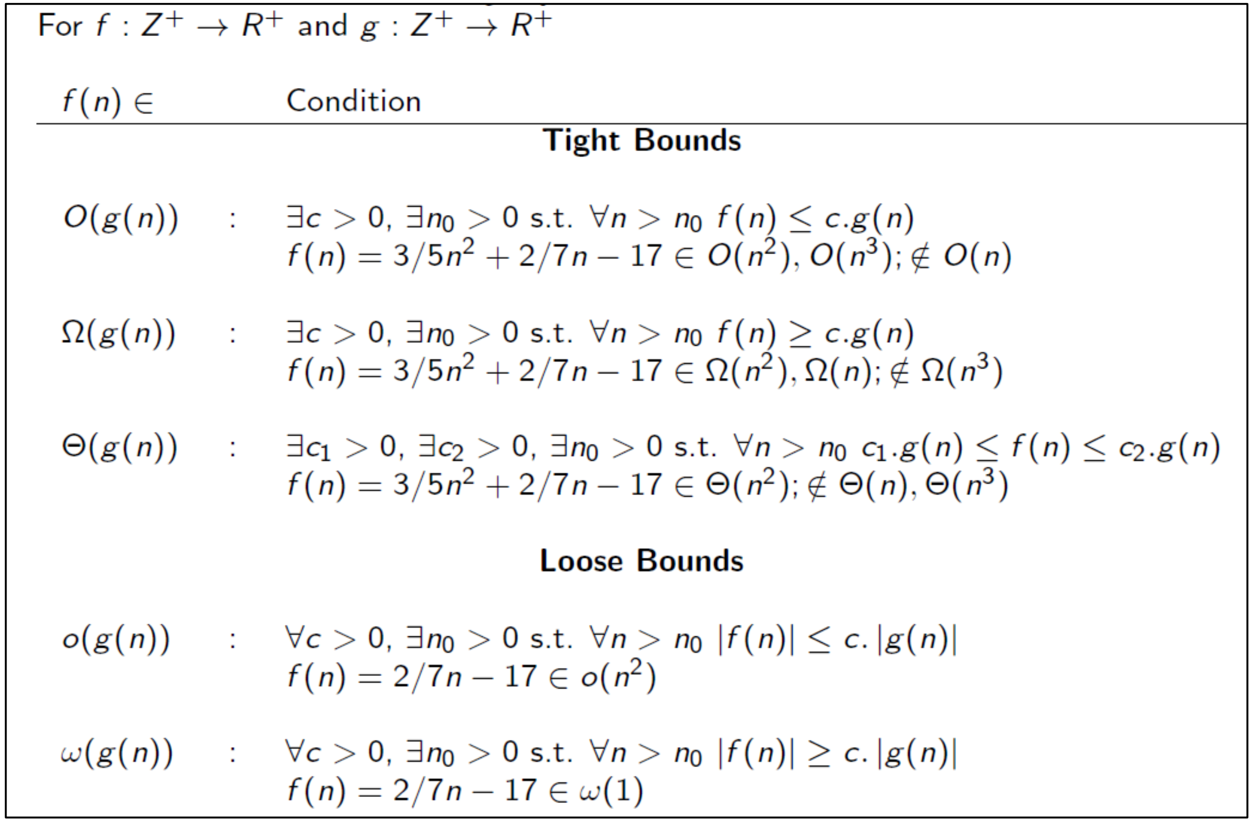


Commonly-used notations in the theory of algorithms

notation	provides	example	shorthand for	used to
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$ $\vdots$	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$ $\vdots$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2} N^2$ N^5 $N^3 + 22 N \log N + 3 N$ $\vdots$	develop lower bounds

Courtesy: Algorithms by Robert Sedgewick & Kevin Wayne

Formal Definitions of Asymptotic Notations



For $f : Z^+ \to R^+$ and $g : Z^+ \to R^+$

$f(n) \in$		Condition
		Tight Bounds
$O(g(n))$	:	$\exists c > 0,\ \exists n_0 > 0$ s.t. $\forall n > n_0\ f(n) \le c.g(n)$ $f(n) = 3/5n^2 + 2/7n - 17 \in O(n^2), O(n^3); \notin O(n)$
$\Omega(g(n))$	:	$\exists c > 0,\ \exists n_0 > 0$ s.t. $\forall n > n_0\ f(n) \ge c.g(n)$ $f(n) = 3/5n^2 + 2/7n - 17 \in \Omega(n^2), \Omega(n); \notin \Omega(n^3)$
$\Theta(g(n))$	:	$\exists c_1 > 0,\ \exists c_2 > 0,\ \exists n_0 > 0$ s.t. $\forall n > n_0\ c_1.g(n) \le f(n) \le c_2.g(n)$ $f(n) = 3/5n^2 + 2/7n - 17 \in \Theta(n^2); \notin \Theta(n), \Theta(n^3)$
		Loose Bounds
$o(g(n))$	:	$\forall c > 0,\ \exists n_0 > 0$ s.t. $\forall n > n_0\ \lvert f(n)\rvert \le c.\lvert g(n)\rvert$ $f(n) = 2/7n - 17 \in o(n^2)$
$\omega(g(n))$	:	$\forall c > 0,\ \exists n_0 > 0$ s.t. $\forall n > n_0\ \lvert f(n)\rvert \ge c.\lvert g(n)\rvert$ $f(n) = 2/7n - 17 \in \omega(1)$

Asymptotic Solutions to Recurrences

- ❑ Max
- ❑ Max and Min
- ❑ Sorting by Max Removal

- ❑ Coins
- ❑ Searching in an Unordered List
- ❑ Searching in an Ordered List

Master Theorem

Let $T(n)$ be a monotonically increasing function that satisfies

$$\begin{aligned} T(n) &= aT(n/b) + f(n) \\ T(1) &= c \end{aligned}$$

where $a \geq 1$, $b > 1$, $c > 0$, and $f(n)$ asymptotically positive. If $f(n) \in \Theta(n^d)$ where $d \geq 0$, then

$$T(n) \in \begin{cases} \Theta(n^{\log_b a}) & \text{Case 1: if } a > b^d \text{ or } d < \log_b a \\ \Theta(n^d \log n) & \text{Case 2: if } a = b^d \text{ or } d = \log_b a \\ \Theta(n^d) & \text{Case 3: if } a < b^d \text{ or } d > \log_b a \end{cases}$$

Master Theorem, Examples

Let $T(n)$ be a monotonically increasing function that satisfies

$$\begin{aligned} T(n) &= aT(n/b) + f(n) \\ T(1) &= c \end{aligned}$$

where $a \geq 1$, $b > 1$, $c > 0$, and $f(n)$ asymptotically positive.
If $f(n) \in \Theta(n^d)$ where $d \geq 0$, then

$$T(n) \in \begin{array}{ll} \Theta(n^{\log_b a}) & \text{Case 1: if } a > b^d \text{ or } d < \log_b a \\ \Theta(n^d \log n) & \text{Case 2: if } a = b^d \text{ or } d = \log_b a \\ \Theta(n^d) & \text{Case 3: if } a < b^d \text{ or } d > \log_b a \end{array}$$

$T(n) =$	a	b	d	Condition	Case	$T(n) =$
$3T(\frac{n}{2}) + \frac{3}{4}n + 1$	3	2	1	$3 > 2^1$	1	$\Theta(n^{\log_b a}) = \Theta(n^{\log_2 3})$
$2T(\frac{n}{4}) + \sqrt{n} + 42$	2	4	$\frac{1}{2}$	$2 = 4^{\frac{1}{2}}$	2	$\Theta(n^d \log n) = \Theta(\sqrt{n} \log n)$
$T(\frac{n}{2}) + \frac{1}{2}n^2 + n$	1	2	2	$1 < 2^2$	3	$\Theta(n^d) = \Theta(n^2)$

Master Theorem: Corollary

Let $T(n)$ be a monotonically increasing function that satisfies

$$\begin{aligned} T(n) &= aT(n/b) + f(n) \\ T(1) &= c \end{aligned}$$

where $a \geq 1$, $b > 1$, $c > 0$, and $f(n)$ asymptotically positive. If $f(n) \in \Theta(n^d)$ where $d \geq 0$, then

$$T(n) \in \begin{cases} \Theta(n^{\log_b a}) & \text{Case 1: if } a > b^d \text{ or } d < \log_b a \\ \Theta(n^d \log n) & \text{Case 2: if } a = b^d \text{ or } d = \log_b a \\ \Theta(n^d) & \text{Case 3: if } a < b^d \text{ or } d > \log_b a \end{cases}$$

Corollary of Master Theorem

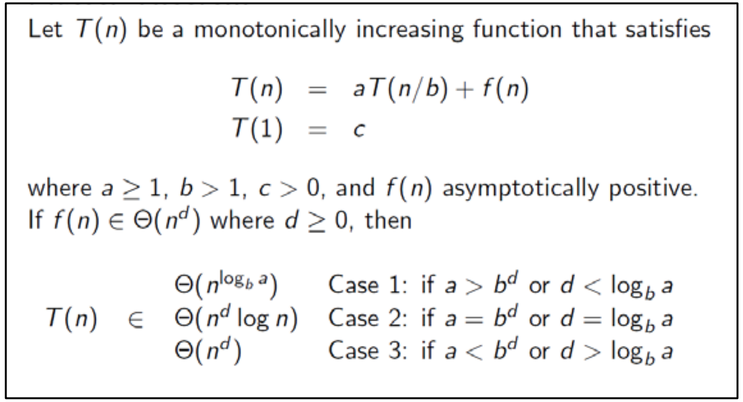
If

$$f(n) \in \Theta(n^{\log_b a} \log^k n)$$

then

$$T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$$

Master Theorem & Recursion Tree

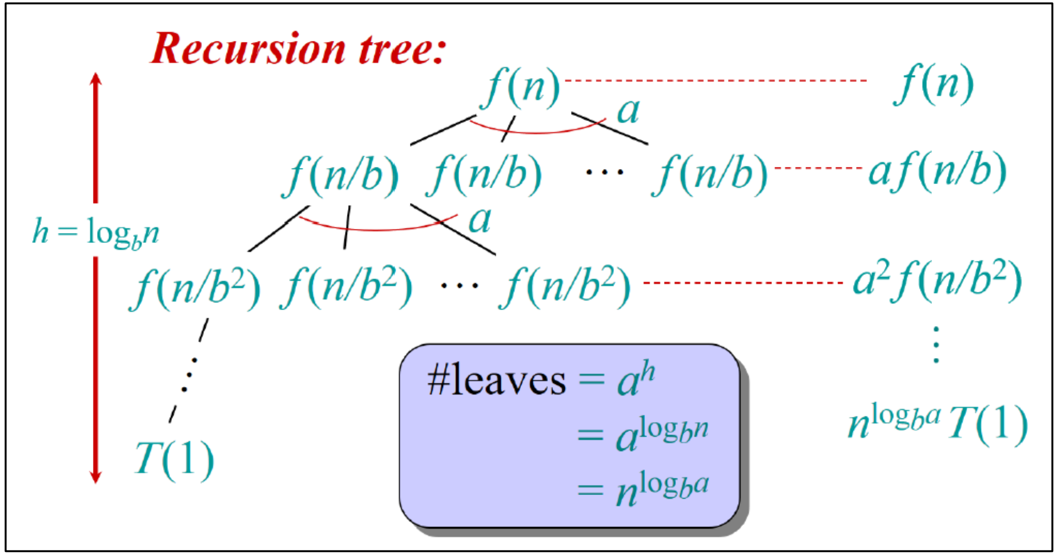


Let $T(n)$ be a monotonically increasing function that satisfies

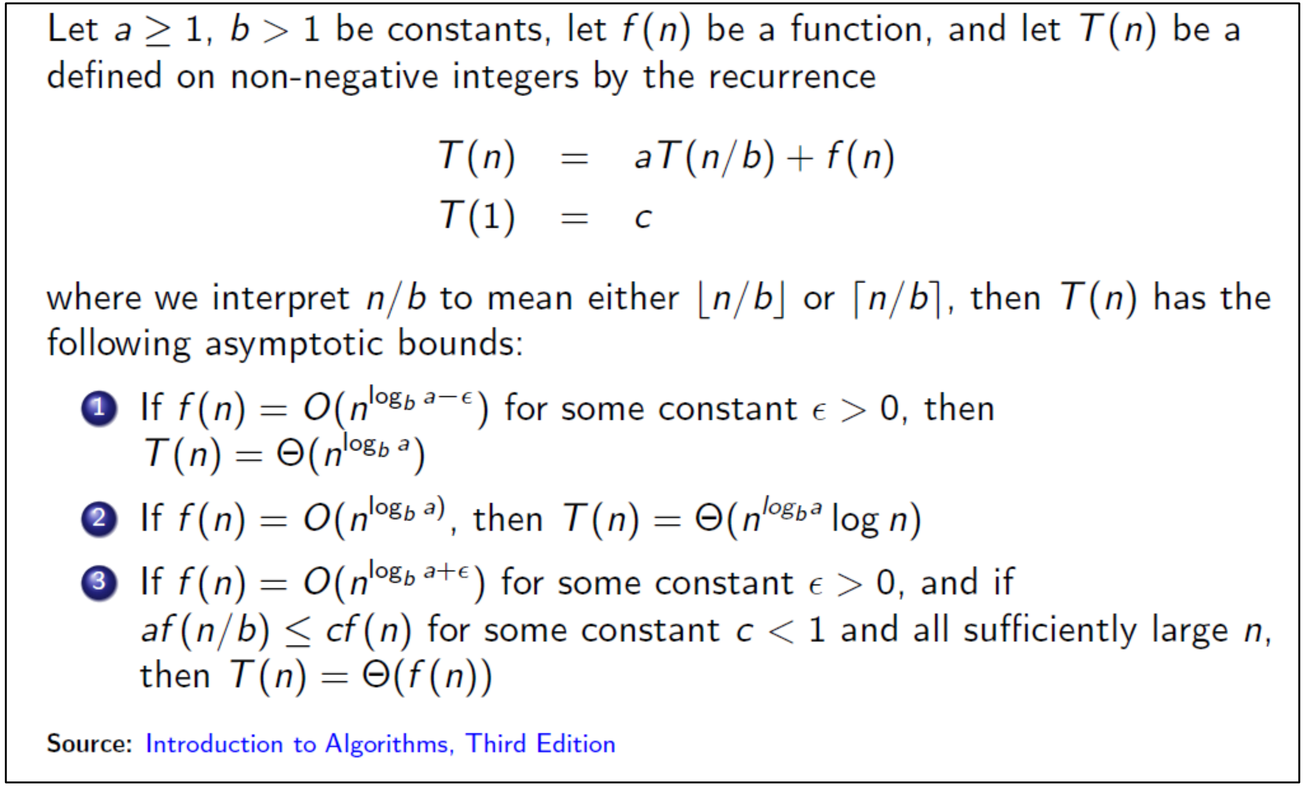
$$\begin{aligned} T(n) &= aT(n/b) + f(n) \\ T(1) &= c \end{aligned}$$

where $a \geq 1$, $b > 1$, $c > 0$, and $f(n)$ asymptotically positive. If $f(n) \in \Theta(n^d)$ where $d \geq 0$, then

$$T(n) \in \begin{cases} \Theta(n^{\log_b a}) & \text{Case 1: if } a > b^d \text{ or } d < \log_b a \\ \Theta(n^d \log n) & \text{Case 2: if } a = b^d \text{ or } d = \log_b a \\ \Theta(n^d) & \text{Case 3: if } a < b^d \text{ or } d > \log_b a \end{cases}$$



Refined Statement of Master Theorem



Let $a \geq 1$, $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be a defined on non-negative integers by the recurrence

$$\begin{aligned} T(n) &= aT(n/b) + f(n) \\ T(1) &= c \end{aligned}$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$, then $T(n)$ has the following asymptotic bounds:

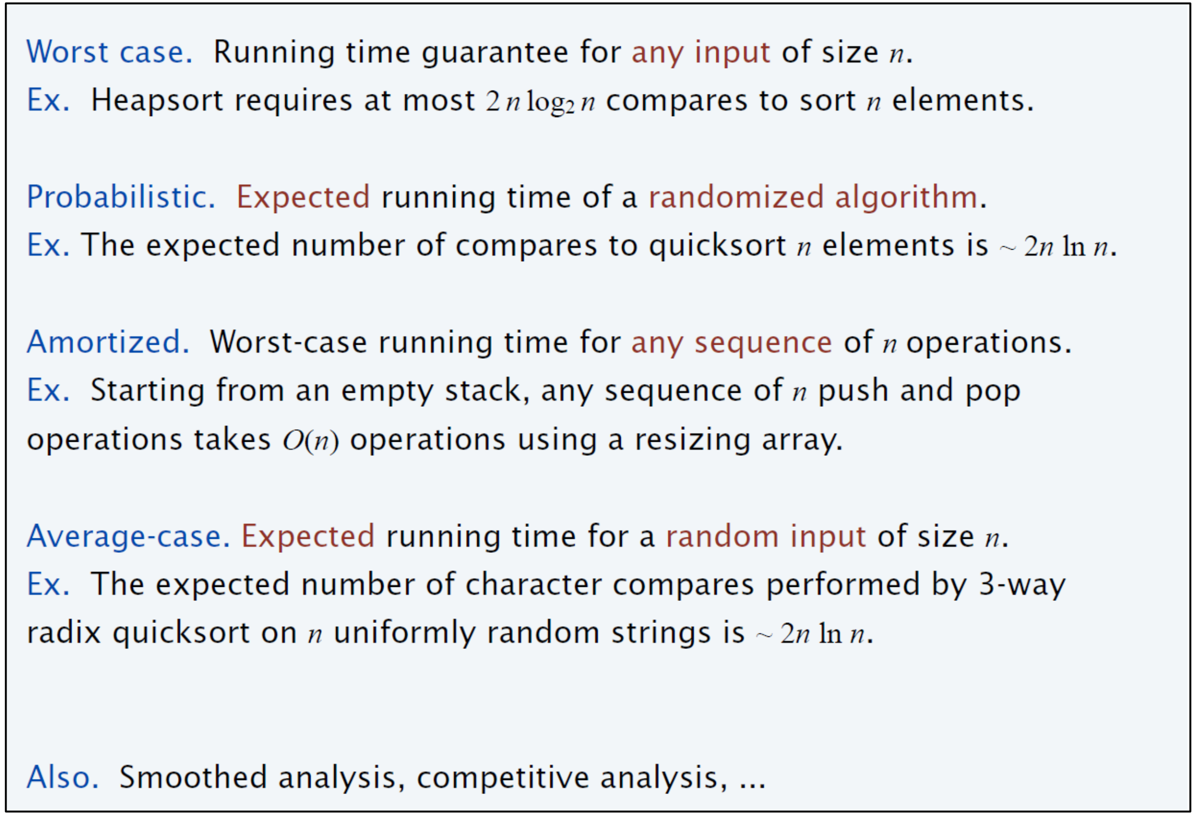
1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = O(n^{\log_b a)}$, then $T(n) = \Theta(n^{\log_b a} \log n)$
3. If $f(n) = O(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$

Source: Introduction to Algorithms, Third Edition

Examples

- [Case 1]: $T(n) = 4T(n/2) + n \Rightarrow T(n) = \Theta(n^2)$
- [Case 2]: $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2 \log n)$
- [Case 3]: $T(n) = 4T(n/2) + n^3 \Rightarrow T(n) = \Theta(n^3)$

Types of Analysis



Worst case. Running time guarantee for any input of size n.
Ex. Heapsort requires at most $2\,n \log_2 n$ compares to sort n elements.

Probabilistic. Expected running time of a randomized algorithm.
Ex. The expected number of compares to quicksort n elements is $\sim 2n \ln n$.

Amortized. Worst-case running time for any sequence of n operations.
Ex. Starting from an empty stack, any sequence of n push and pop operations takes $O(n)$ operations using a resizing array.

Average-case. Expected running time for a random input of size n.
Ex. The expected number of character compares performed by 3-way radix quicksort on n uniformly random strings is $\sim 2n \ln n$.

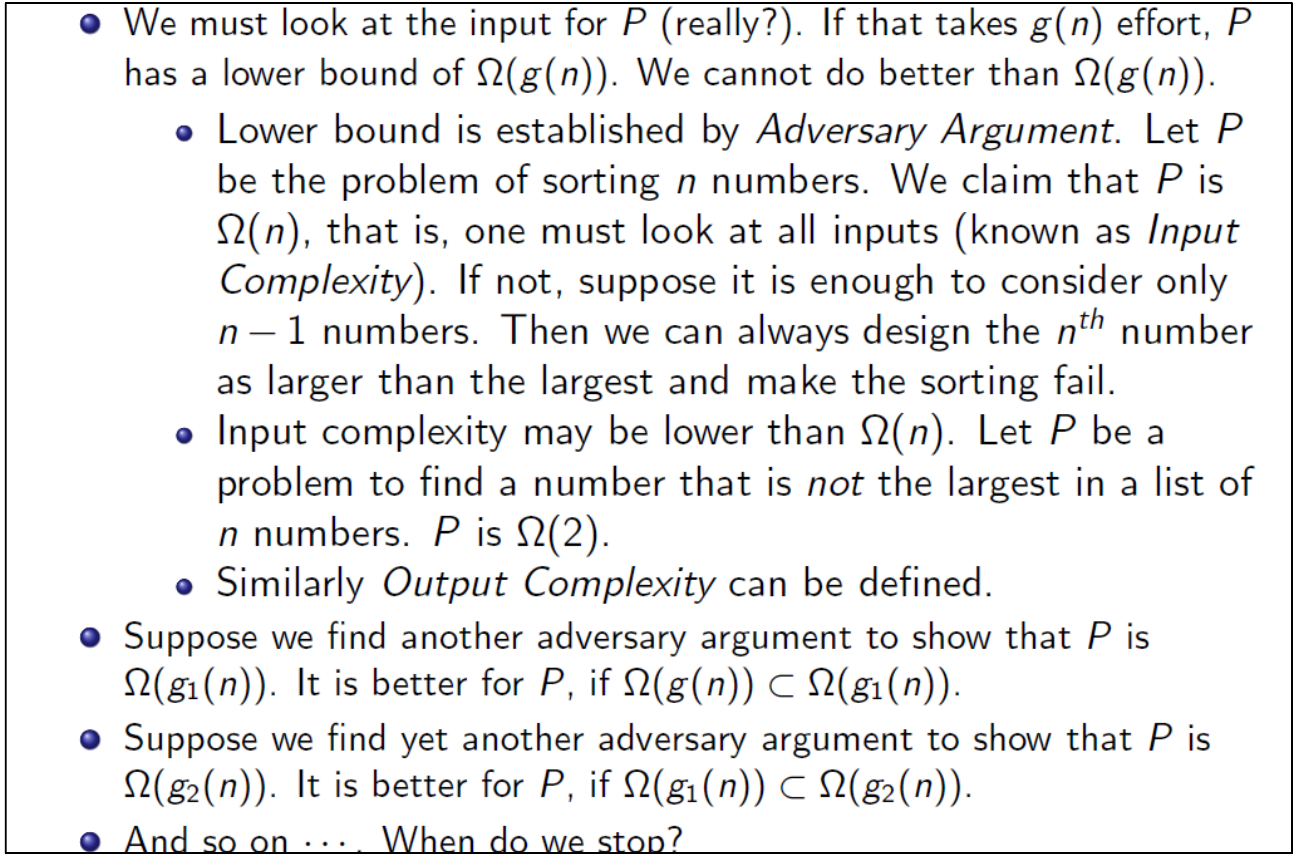
Also. Smoothed analysis, competitive analysis, ...

Courtesy: Algorithms by Robert Sedgewick & Kevin Wayne

Upper Bounds

- If A is $O(f(n))$, then A is an $O(f(n))$ upper bound for P because we already know an algorithm (that is, A) that can solve P in $O(f(n))$.
- Let A_1 be another algorithm that solves P in $O(f_1(n))$. Naturally, A_1 is a better upper bound for P, if $O(f_1(n)) \subset O(f(n))$.
- Let A_2 be yet another algorithm that solves P in $O(f_2(n))$. Naturally, A_2 is a better upper bound for P, if $O(f_2(n)) \subset O(f_1(n))$.
- Let A_3 be $\cdots$. When do we stop?

Lower Bounds



- We must look at the input for P (really?). If that takes $g(n)$ effort, P has a lower bound of $\Omega(g(n))$. We cannot do better than $\Omega(g(n))$.
  - Lower bound is established by *Adversary Argument*. Let P be the problem of sorting n numbers. We claim that P is $\Omega(n)$, that is, one must look at all inputs (known as *Input Complexity*). If not, suppose it is enough to consider only $n-1$ numbers. Then we can always design the n^{th} number as larger than the largest and make the sorting fail.
  - Input complexity may be lower than $\Omega(n)$. Let P be a problem to find a number that is *not* the largest in a list of n numbers. P is $\Omega(2)$.
  - Similarly *Output Complexity* can be defined.
- Suppose we find another adversary argument to show that P is $\Omega(g_1(n))$. It is better for P, if $\Omega(g(n)) \subset \Omega(g_1(n))$.
- Suppose we find yet another adversary argument to show that P is $\Omega(g_2(n))$. It is better for P, if $\Omega(g_1(n)) \subset \Omega(g_2(n))$.
- And so on $\cdots$. When do we stop?

Lower Bounding Methods

- Information Theoretic Analysis
- Adversary Argument
- Reduction of Problems

Optimal Algorithm

- A is *Optimal* for P, if $O(f(n)) = \Omega(f(n))$, that is, the lower bound matches the upper bound.
- Complexity of A is $\Theta(f(n))$.
- Till an optimal algorithm is found we continuously work on the algorithm to lower the upper bound and devise adversary arguments to upper the lower bound.

Summary

Algorithm Design by Recursion Transformation

- ❑ Algorithms and Programs
- ❑ Pseudo-Code
- ❑ Algorithms + Data Structures = Programs
- ❑ Initial Solutions + Analysis + Solution Refinement + Data Structures = Final Algorithm
- ❑ Use of Recursive Definitions as Initial Solutions
- ❑ Recurrence Equations for Proofs and Analysis
- ❑ Solution Refinement through Recursion Transformation and Traversal
- ❑ Data Structures for saving past computation for future use

1. Initial Solution
   a. Recursive Definition – A set of Solutions
   b. Inductive Proof of Correctness
   c. Analysis Using Recurrence Relations
2. Exploration of Possibilities
   a. Decomposition or Unfolding of the Recursion Tree
   b. Examination of Structures formed
   c. Re-composition Properties
3. Choice of Solution & Complexity Analysis
   a. Balancing the Split, Choosing Paths
   b. Identical Sub-problems
4. Data Structures & Complexity Analysis
   a. Remembering Past Computation for Future
   b. Space Complexity
5. Final Algorithm & Complexity Analysis
   a. Traversal of the Recursion Tree
   b. Pruning
6. Implementation
   a. Available Memory, Time, Quality of Solution, etc

Overview of Algorithm Design

1. Initial Solution
   a. Recursive Definition – A set of Solutions
   b. Inductive Proof of Correctness
   c. Analysis Using Recurrence Relations
2. Exploration of Possibilities
   a. Decomposition or Unfolding of the Recursion Tree
   b. Examination of Structures formed
   c. Re-composition Properties
3. Choice of Solution & Complexity Analysis
   a. Balancing the Split, Choosing Paths
   b. Identical Sub-problems
4. Data Structures & Complexity Analysis
   a. Remembering Past Computation for Future
   b. Space Complexity
5. Final Algorithm & Complexity Analysis
   a. Traversal of the Recursion Tree
   b. Pruning
6. Implementation
   a. Available Memory, Time, Quality of Solution, etc

1. Core Methods
   a. Divide and Conquer
   b. Greedy Algorithms
   c. Dynamic Programming
   d. Branch-and-Bound
   e. Analysis using Recurrences
   f. Advanced Data Structuring
2. Important Problems to be addressed
   a. Sorting and Searching
   b. Strings and Patterns
   c. Trees and Graphs
   d. Combinatorial Optimization
3. Complexity & Advanced Topics
   a. Time and Space Complexity
   b. Lower Bounds
   c. Polynomial Time, NP-Hard
   d. Parallelizability, Randomization

Thank you

Any Questions?