Design and Analysis Algorithm

Adnan YAZICI
Dept. of Computer Engineering
Middle East Technical Univ.
Ankara - TURKEY

Design and Analysis of Algorithms

- An ***algorithm*** is a set of instructions to be followed to solve a problem. Another word, an algorithm is a complete set of rules that transform the input into the output in a finite number of steps.
  - There can be more than one solution (more than one algorithm) to solve a given problem.
  - An algorithm can be implemented using different programming languages on different platforms.
- An algorithm should correctly solve the problem.
  - e.g., for sorting, this means even if (1) the input is already sorted, or (2) it contains repeated elements.
- Once we have a correct algorithm for a problem, we have to determine the efficiency of that algorithm.

Design and Analysis of Algorithms

- **<u>Aspects of studying algorithms:</u>**
- *1. Designing algorithms:*
  - putting the pieces of the puzzles together,
  - choosing data structures,
  - selecting the basic approaches to the solution of the problem,
- The most popular design strategies are *divide&conquer, greedy, dynamic prog., backtracking,* and *branch&bound*.

- *2. Expressing and implementing the algorithm*
- Concerns are:
  - clearness
  - conciseness
  - effectiveness

Design and Analysis of Algorithms

♦*3. Analyzing the algorithm*

Algorithm analysis is assessing the time and space resources required by an algorithm as a function of the size of the problem, without actually implementing the algorithm.

♦*4. Compare UB and LB to see if your solution is good enough*

- ♦ Analyzing the algorithm gives us the upper bound to solve the problem
- ♦ Analyzing the problem gives us the lower bound to solve the problem

♦*5. Validate the algorithm*

We show that the algorithm computes the correct answer for all possible legal (or given) inputs

Design and Analysis of Algorithms

6. Verifying the algorithm (or program)

An algorithm is said to be *correct* (*verified*) if, for every input instance, it halts with the correct output.

7. Testing algorithms

There are two phases;

♦ *Debugging:* The process of executing programs on sample data sets to determine if faulty results occur, and if so, to correct them.

"Debugging can only point to the presence of errors, but not to their absence"

♦ *Profiling:* the process of executing a correct program on various data sets and measuring the time (and space) it takes to compute the results.

Algorithmic Performance

There are *two aspects* of algorithmic performance:

- Time
  - Instructions take time.
  - How fast does the algorithm perform?
  - What affects its runtime?
- Space
  - Data structures take space
  - What kind of data structures can be used?
  - How does choice of data structure affect the runtime?

➢ We will focus on time:

– How to estimate the time required for an algorithm

– How to reduce the time required

Analysis of Algorithms

- ***Analysis of Algorithms*** is the area of computer science that provides tools to analyze the efficiency of different methods of solutions.

- How do we compare the time efficiency of two algorithms that solve the same problem?

  Naïve Approach: implement these algorithms in a programming language (i.e., C++), and run them to compare their time requirements.

Analysis of Algorithms

- Comparing the programs (instead of algorithms) has difficulties.
  - *What data should the program use?*
    - Any analysis must be independent of specific data. Execution time is sensitive to the amount of data manipulated, grows as the amount of data increases.
  - *What computer should we use?*
    - We should compare the efficiency of the algorithms independently of a particular computer. Because of the execution speed of the processors, the execution times for an algorithm on the same data set on two different computers may differ.
  - *How are the algorithms coded?*
    - Comparing running times means comparing the implementations.
    - We should not compare implementations, because they are sensitive to programming style that may cloud the issue of which algorithm is inherently more efficient.
- *∴ absolute measure for an algorithm is not appropriate*

Analysis of Algorithms

- When we analyze algorithms, we should employ mathematical techniques that analyze algorithms independently of *specific implementations, computers,* or *data.*

- To analyze algorithms:
  - First, we start to count the number of significant operations in a particular solution to assess its efficiency.
  - Then, we express the efficiency of algorithms using growth functions.

What is Important?

- An array-based list `retrieve` operation is O(1), a linked-list-based list `retrieve` operation is O(n).
- But insert and delete operations are much easier on a linked-list-based list implementation.

  ➔ When selecting the implementation of an Abstract Data Type (ADT), we have to consider how frequently particular ADT operations occur in a given application.

- If the problem size is always very small, we can probably ignore the algorithm's efficiency.
  - In this case, we should choose the simplest algorithm.

What is Important? (cont.)

- We have to weigh the trade-offs between an algorithm's time requirement and its memory requirements.
- We have to compare algorithms for both style and efficiency.
  - The analysis should focus on gross differences in efficiency and not reward coding tricks that save small amount of time.
  - That is, there is no need for coding tricks if the gain is not too much.
  - Easily understandable program is also important.
- Order-of-magnitude analysis focuses on large problems.

Analysis of Algorithms

- A *running time function*, T(n), yields the time required to execute the algorithm of a problem of size 'n.'
- Often the analysis of an algorithm leads to T(n), which may contain unknown constants, which depend on the characteristics of the ideal machine. So, we cannot determine this function exactly.

- $T(n) = an^2 + bn + c$, where a,b,and c are unspecıfied constants

General Rules for Estimation

- **Loops**: The running time of a loop is at most the running time of the statements inside of that loop times the number of iterations.
- **Nested Loops**: Running time of a nested loop containing a statement in the inner most loop is the running time of statement multiplied by the product of the sized of all loops.
- **Consecutive Statements:** Just add the running times of those consecutive statements.
- **If/Else**: Never more than the running time of the test plus the larger of running times of c1 and c2.

The Execution Time of Algorithms

Example: Simple Loop

	Cost	Times
`i = 1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`  i = i + 1;`	c4	n
`  sum = sum + i;`	c5	n
`}`		

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*c5

➔ The time required for this algorithm is proportional to n

The Execution Time of Algorithms (cont.)

Example: Nested Loop

```
                                    Cost        Times
i=1;                                c1          1
sum = 0;                            c2          1
while (i <= n) {                    c3          n+1
    j=1;                            c4          n
    while (j <= n) {                c5          n*(n+1)
        sum = sum + i;              c6          n*n
        j = j + 1;                  c7          n*n
    }
    i = i +1;                       c8          n
}
```

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8

➔ The time required for this algorithm is proportional to n^2

The Execution Time of Algorithms (cont.)

Example: Simple If-Statement

	Cost	Times
`if (n < 0)`	c1	1
`  absval = -n`	c2	1
`else`		
`  absval = n;`	c3	1

Total Cost <= c1 + max(c2,c3)

Analysis of Algorithms

- We measure the complexity of an algorithm by identifying a basic operation and then counting how any times the algorithm performs that basic operation for an input size n.

problem	input of size n	basic operation
searching a list	lists with n elements	comparison
sorting a list	lists with n elements	comparison
multiplying two matrices	two n-by-n matrices	multiplication
traversing a tree	tree with n nodes	accessing a node
Towers of Hanoi	n disks	moving a disk

Algorithm Growth Rates

- We measure an algorithm's time requirement as a function of the *problem size*.
  - Problem size depends on the application: e.g. number of elements in a list for a sorting algorithm, the number disks for towers of hanoi.
- So, for instance, we say that (if the problem size is n)
  - Algorithm A requires **$5*n^2$** time units to solve a problem of size n.
  - Algorithm B requires **$7*n$** time units to solve a problem of size n.
- The most important thing to learn is how quickly the algorithm's time requirement grows as a function of the problem size.
  - Algorithm A requires time proportional to **n^2**.
  - Algorithm B requires time proportional to **n**.
- An algorithm's proportional time requirement is known as ***growth rate***.

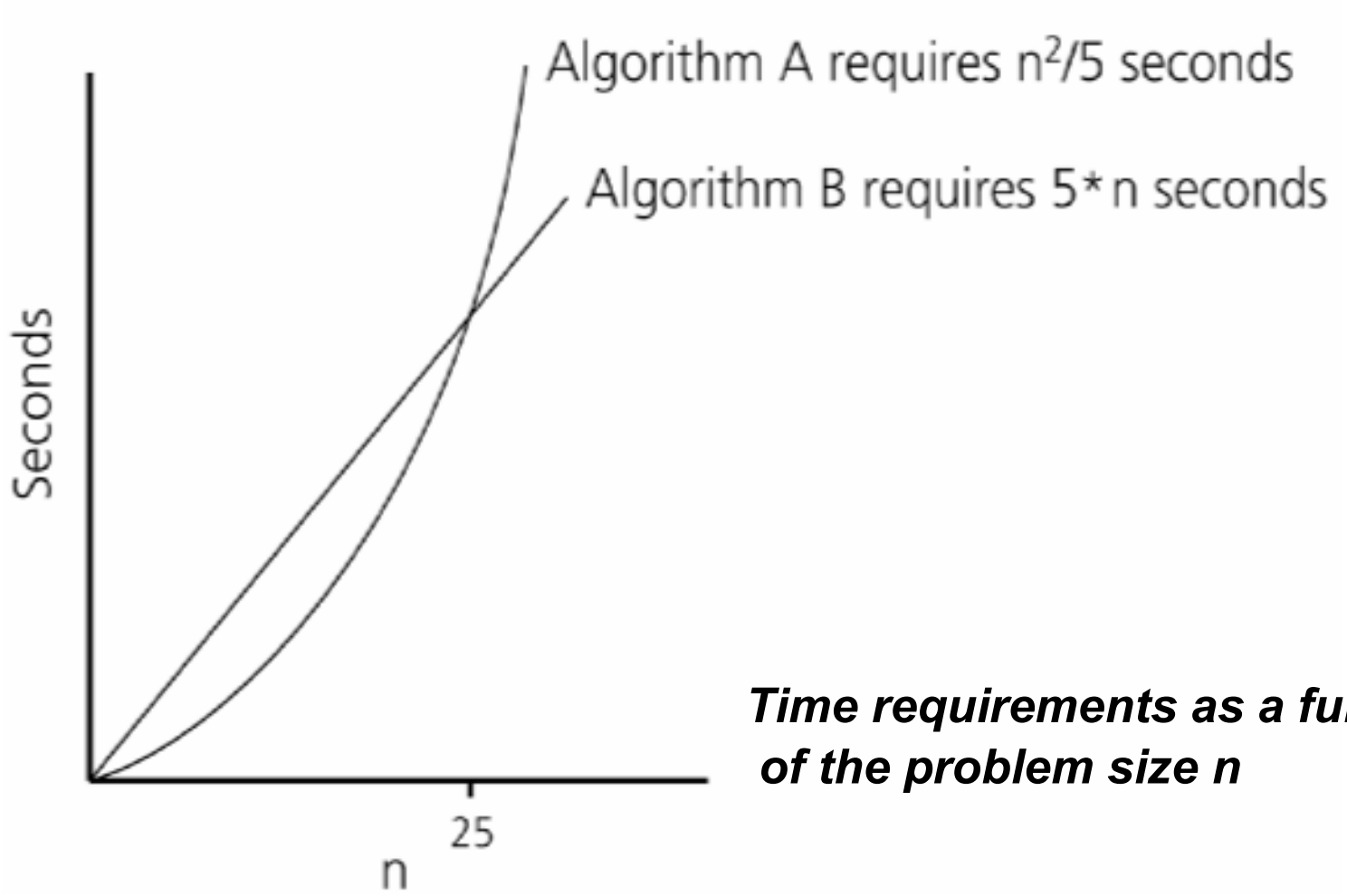
Analysis of Algorithms

- The *growth rate* of T(n) is referred to the *computational complexity* of the algorithm.
- The *computational complexity* gives a concise way of saying how the running time, T(n), varies with n and is independent of any particular implementation.
- We can compare the efficiency of two algorithms by comparing their growth rates.
- The lower the growth rate, the faster the algorithm, at least for large values of n.

- *For example;* $T(n) = an^2 + bn + c$, the growth rate is $O(n^2)$

- The *goal* of the *algorithm designer* should be an algorithm with as low a growth rate of the running time function, T(n), of that algorithm as possible.

Algorithm Growth Rates (cont.)



Time requirements as a function of the problem size n

Common Growth Rates

Function	Growth Rate Name
c	Constant
$log\ N$	Logarithmic
$log^2 N$	Log-squared
N	Linear
$N\ log\ N$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Growth-Rate Functions

O(1) Time requirement is **constant**, and it is independent of the problem's size.

O($\log_2 n$) Time requirement for a **logarithmic** algorithm increases increases slowly as the problem size increases.

O(n) Time requirement for a **linear** algorithm increases directly with the size of the problem.

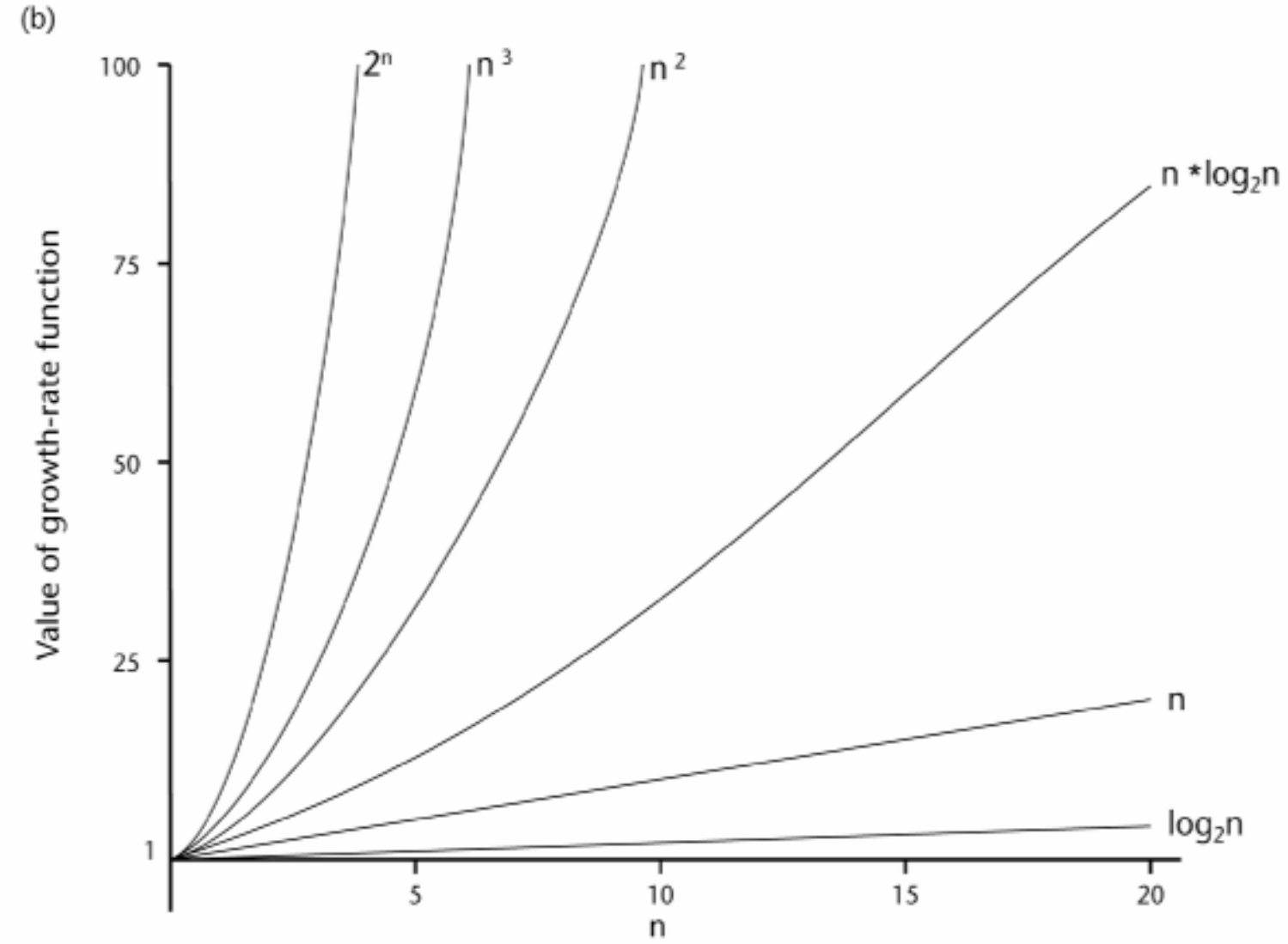
O($n*\log_2 n$) Time requirement for a **$n*\log_2 n$** algorithm increases more rapidly than a linear algorithm.

O(n^2) Time requirement for a **quadratic** algorithm increases rapidly with the size of the problem.

O(n^3) Time requirement for a c**ubic** algorithm increases more rapidly with the size of the problem than the time requirement for a quadratic algorithm.

O(2^n) As the size of the problem increases, the time requirement for an **exponential** algorithm increases too rapidly to be practical.

A Comparison of Growth-Rate Functions (cont.)



Properties of Growth-Rate Functions

1. We can ignore low-order terms in an algorithm's growth-rate function.
   - If an algorithm is $O(n^3+4n^2+3n)$, it is also $O(n^3)$.
   - We only use the higher-order term as algorithm's growth-rate function.

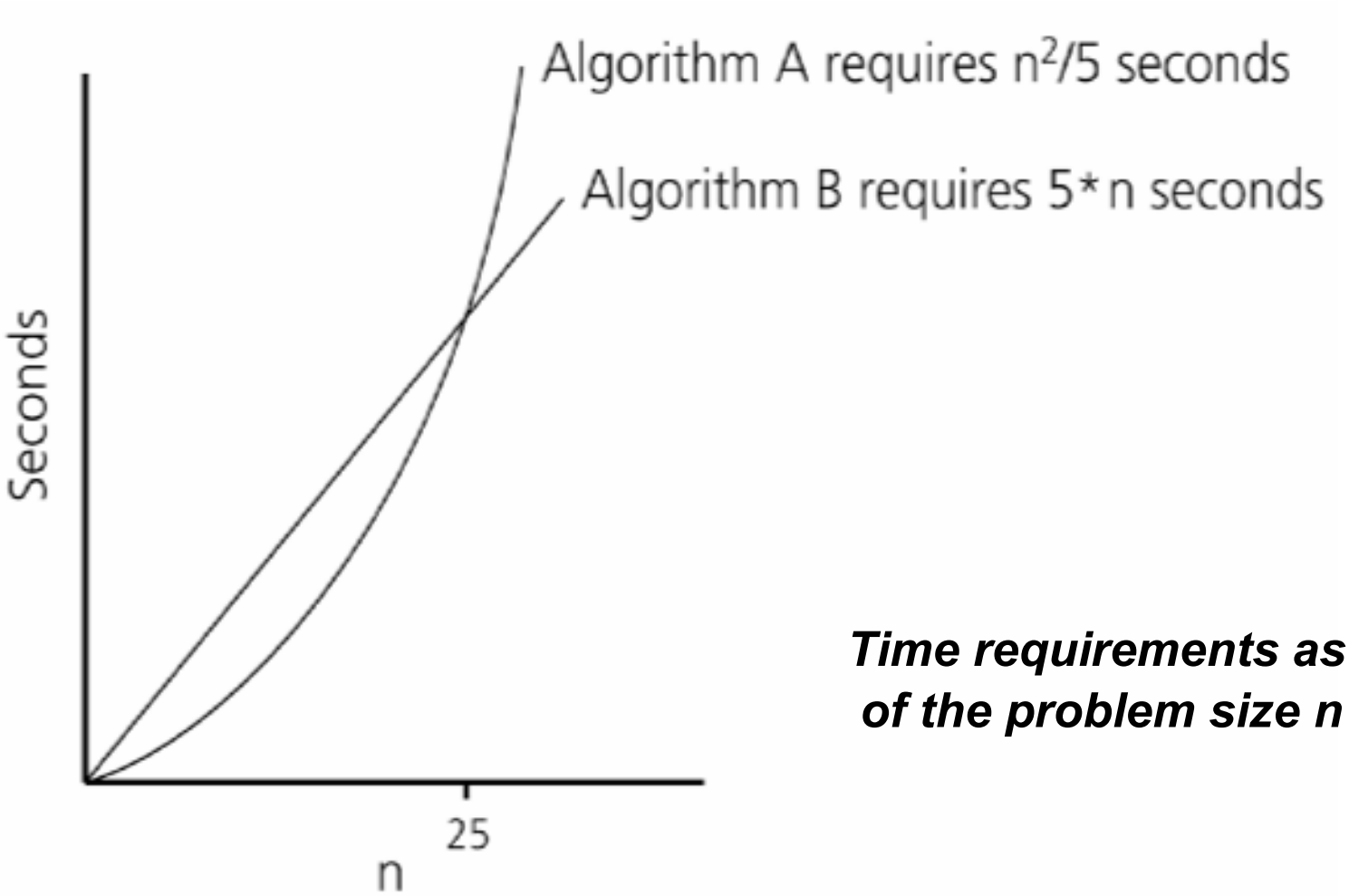
2. We can ignore a multiplicative constant in the higher-order term of an algorithm's growth-rate function.
   - If an algorithm is $O(5n^3)$, it is also $O(n^3)$.

3. $O(f(n)) + O(g(n)) = O(f(n)+g(n))$
   - We can combine growth-rate functions.
   - If an algorithm is $O(n^3) + O(4n)$, it is also $O(n^3 +4n^2)$ ➔ So, it is $O(n^3)$.
   - Similar rules hold for multiplication.

Problems with growth rate analysis

- An algorithm with a smaller growth rate will not run faster than one with a higher growth rate for any particular n, but only for n ‘large enough’
- Algorithms with identical growth rates may have strikingly different running times because of the constants in the running time functions.
  - The value of n where two growth rates are the same is called the *break-even point*.

Algorithm Growth Rates (cont.)



Time requirements as a function of the problem size n

Figure 6.1

Running times for small inputs

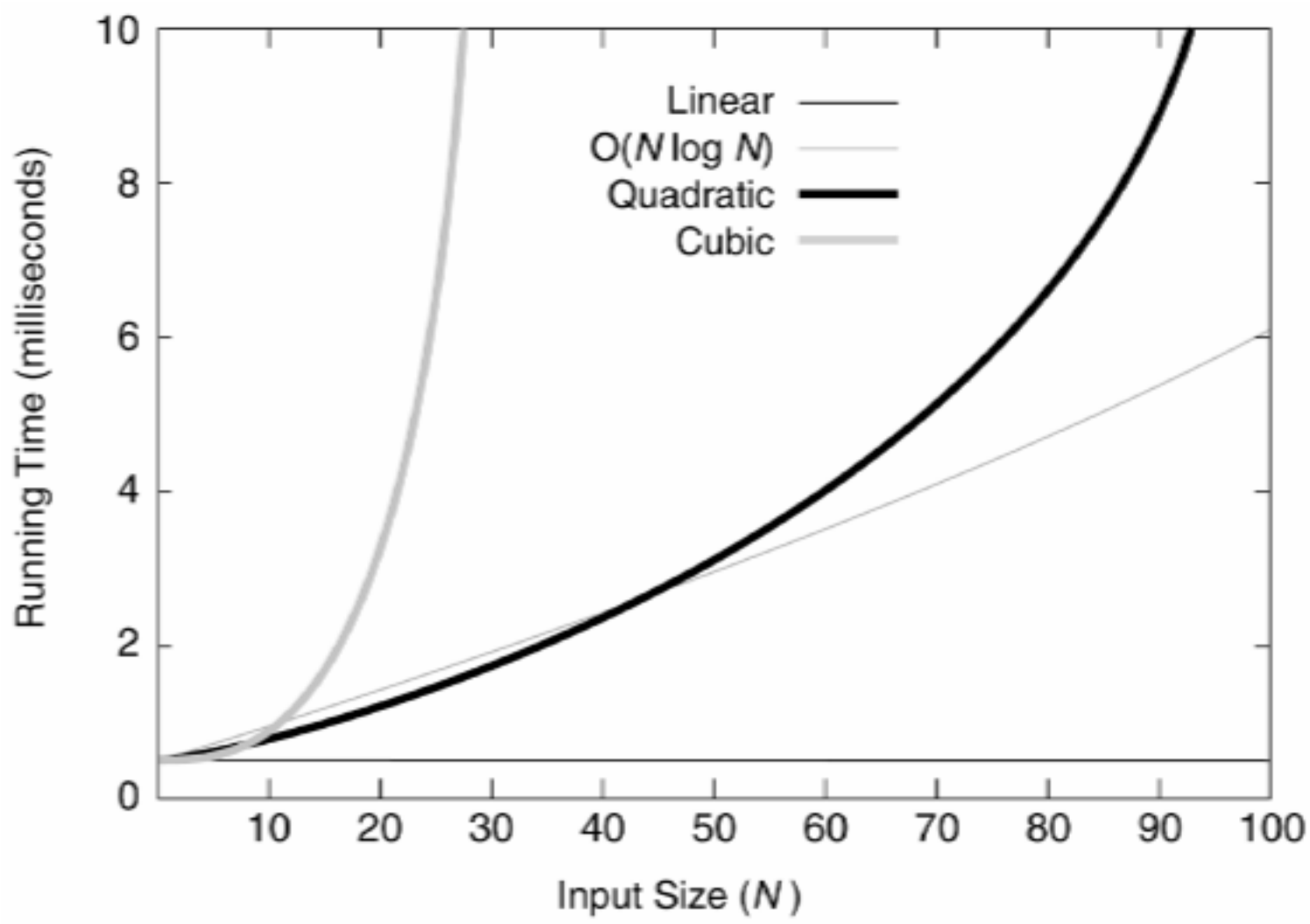
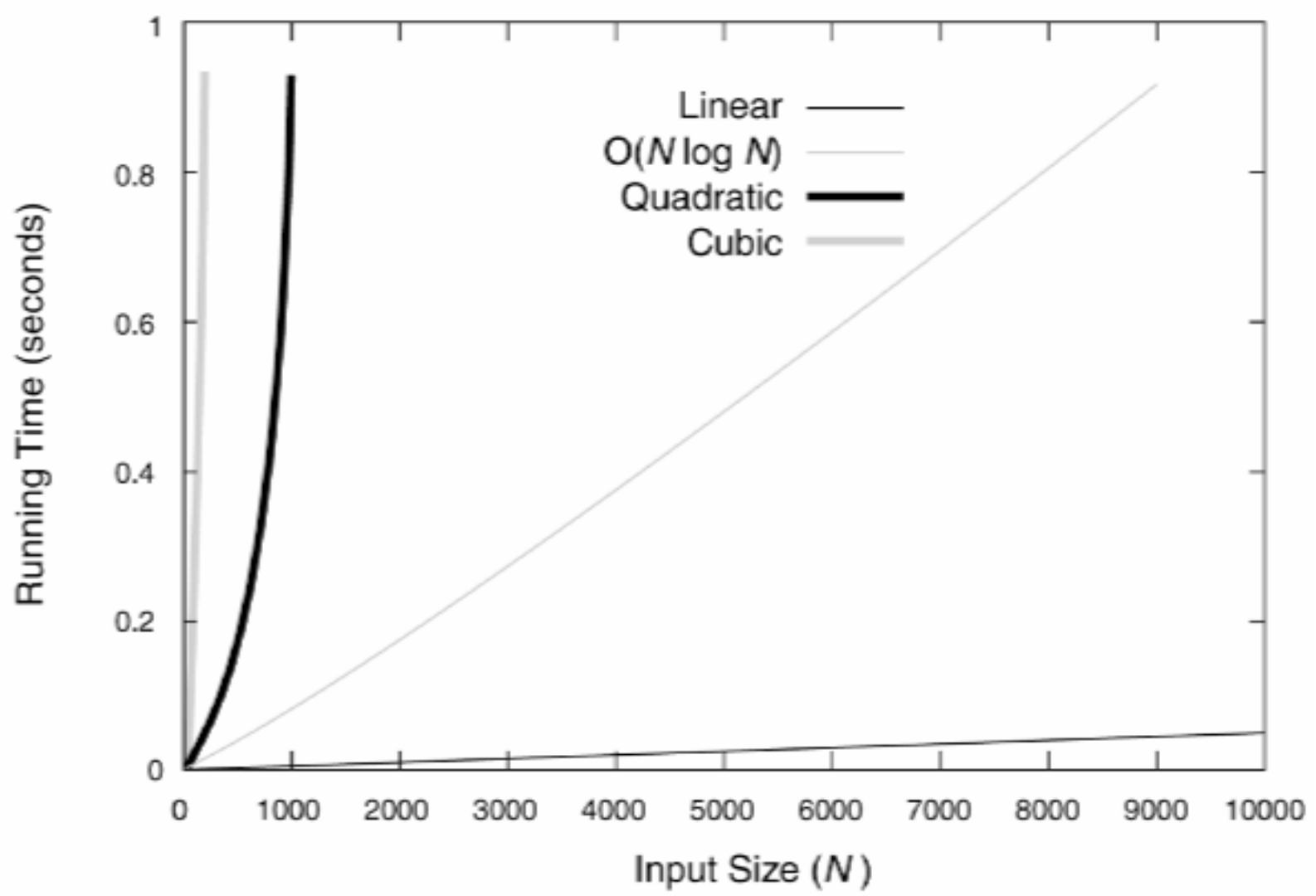


Figure 6.2

Running times for moderate inputs



A Comparison of Growth-Rate Functions

(a)

Function	n = 10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

A Comparison of Growth-Rate Functions

Importance of developing Efficient Algorithms:

Sequential search vs Binary search

Array size	**No. of comparisons by seq. src**	**No. of comparisons by bin. Srch**
128	128	8
1,048,576	1,048,576	21
~4.10^9	~4.10^9	33

Execution times for algorithms with the given time complexities:

n	**f(n)=n**	**nlgn**	**n^2**	**2^n**
20	0.02 μs	0.086 μs	0.4 μs	1 ms
10^6	1μs	19.93 ms	16.7 min	31.7 years
10^9	1s	29.9s	31.7 years	!!! centuries

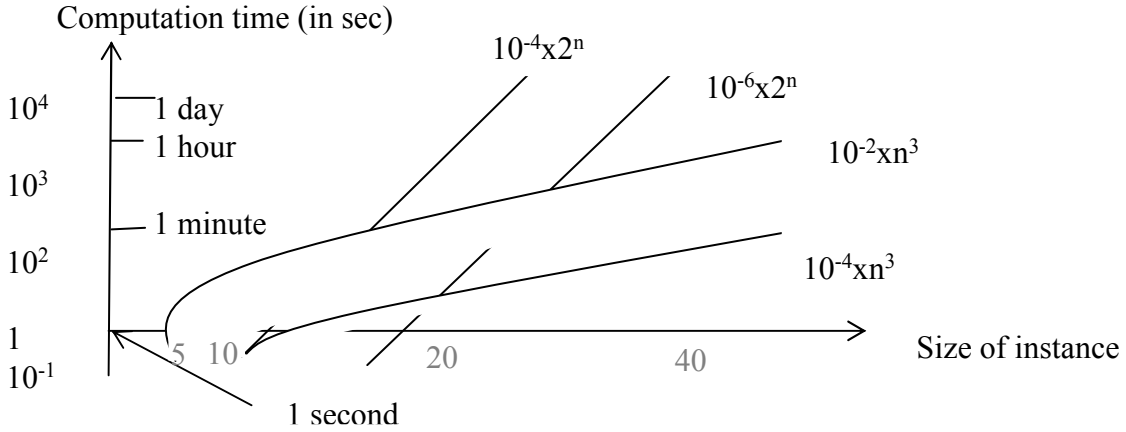


Figure: Algorithms versus hardware

Definition of the Orders of an Algorithm

O-Notation: Given two running time functions f(n) and g(n), we say that f(n) is O(g(n)) if there exists a real constant $c \geq 0$, and a positive integer n_0, such that $f(n) \leq c.g(n)$ for all $n \geq n_0$

O-notation gives an upper bound for a function to within a constant factor.

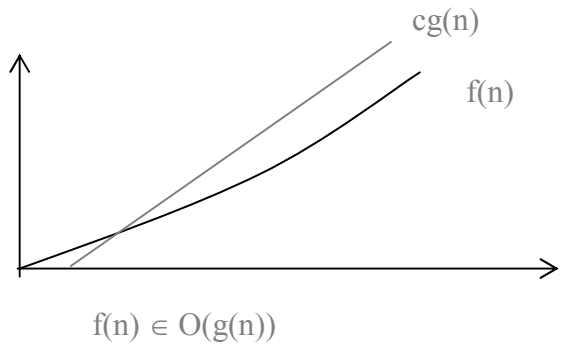
Example: We want to show that $1/2n^2 + 3n \in O(n^2)$

$f(n) = 1/2n^2 + 3n$ $\qquad$ $g(n) = n^2$

to show desired result, need c and n_0 such that $0 \leq 1/2n^2 + 3n \leq c.n^2$

try c =1

$1/2n^2 + 3n \leq n^2$ $\qquad$ $3n \leq 1/2n^2$ $\qquad$ $6 \leq n$ i.e. $n_0 = 6$.



Order of an Algorithm

- **Example**: If an algorithm requires $n^2–3*n+10$ seconds to solve a problem size n. If constants k and n_0 exist such that

  $k*n^2 \geq n^2–3*n+10$ for all $n \geq n_0$.

  the algorithm is order n^2 (In fact, k is 3 and n_0 is 2)

  $3*n^2 \geq n^2–3*n+10$ for all $n \geq 2$.

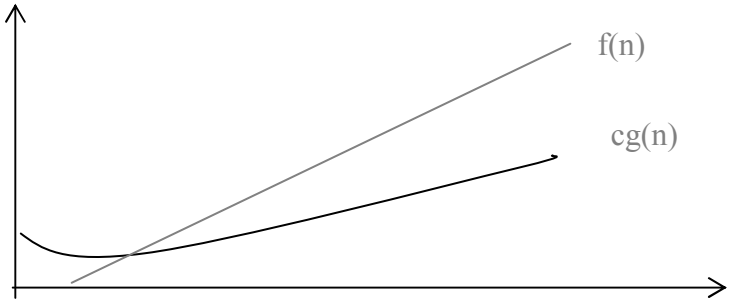
  Thus, the algorithm requires no more than $k*n^2$ time units for $n \geq n_0$, So it is **$O(n^2)$**

Definition of the Orders of an Algorithm

Ω- Notation: We say that f(n) is Ω(g(n)) if there exists a real constant $c \geq 0$, and positive integer n_0, such that

$$c.g(n) \leq f(n) \text{ for all } n \geq n_0$$

Example: We show that $n^2 + 10n \in \Omega(n^2)$. Because for $n \geq 0$, $n^2 + 10n \geq n^2$, we can take $c = 1$ and $n = 0$ to obtain the result.



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

Relationships among O, Ω, and Θ - Notations

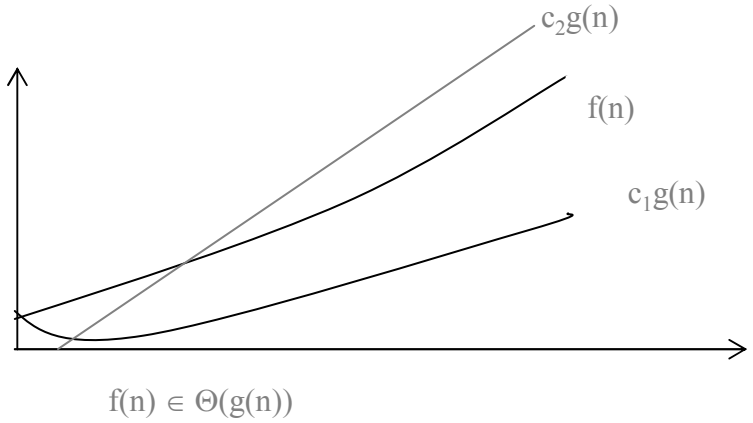
- f(n) is O(g(n)) iff g(n) is Ω(f(n))
- f(n) is Θ(g(n)) iff f(n) is O(g(n)) and f(n) is Ω(g(n)),
- Θ(g(n)) = O(g(n)) ∩ Ω(g(n)),

Definition of the Orders of an Algorithm

Θ-Notation: If f(n) and g(n) are running time functions, then we say that $f(n) \in \Theta(g(n))$ if there exists positive real constants c_1 and c_2, and a positive integer n_0, such that

$$c_1.g(n) \leq f(n) \leq c_2.g(n)$$

for all $n \geq n_0$. That is, f(n) is bounded from above and below by g(n). That is, Θ-notation bounds a function within constant factors. f(n) always lies between $c_1.g(n)$ and $c_2.g(n)$ inclusive.



Analysis of Algorithms

***Example*:** We want to show that $1/2n^2 - 3n = \Theta(n^2)$.

***Solution*:** $f(n) = 1/2n^2 - 3n$ $g(n) = n^2$

To show desired result, we need determine positive constants c_1, c_2, and n_0 such that

$0 \le c_1 . n^2 \le 1/2n^2 - 3n \le c_2 . n^2$ for all $n \ge n_0$.

Dividing by n^2, we get $0 \le c_1 \le 1/2 - 3/n \le c_2$

$c_1 \le 1/2 - 3/n$ holds for any value of $n \ge 7$ by choosing

$c_1 \le 1/14$

$1/2 - 3/n \le c_2$ holds for any value of $n \ge 1$ by choosing

$c_2 \ge$ ½.

Thus, by choosing $c_1 = 1/14$ and $c_2 =$ ½ and $n_0 = 7$, we can verify $1/2n^2 - 3n = O(n^2)$. Certainly other choices for the constants exist.

Properties of Θ-notation

- f(n) is Θ(f(n)) (reflexivity)
- f(n) is Θ(g(n)) iff g(n) is Θ(f(n)) (symmetry)
- If f(n) is Θ(g(n)) and g(n) is Θ(h(n)) then f(n) is Θ(h(n)) (transitivity)
- For any c > 0, the function c.f(n) is Θ(f(n))
- If f_1 is Θ(g(n)) and f_2 is Θ(g(n)),

  then $(f_1 + f_2)(n)$ is Θ(g(n))
- If f_1 is $\Theta(g_1(n))$ and f_2 is $\Theta(g_2(n))$,

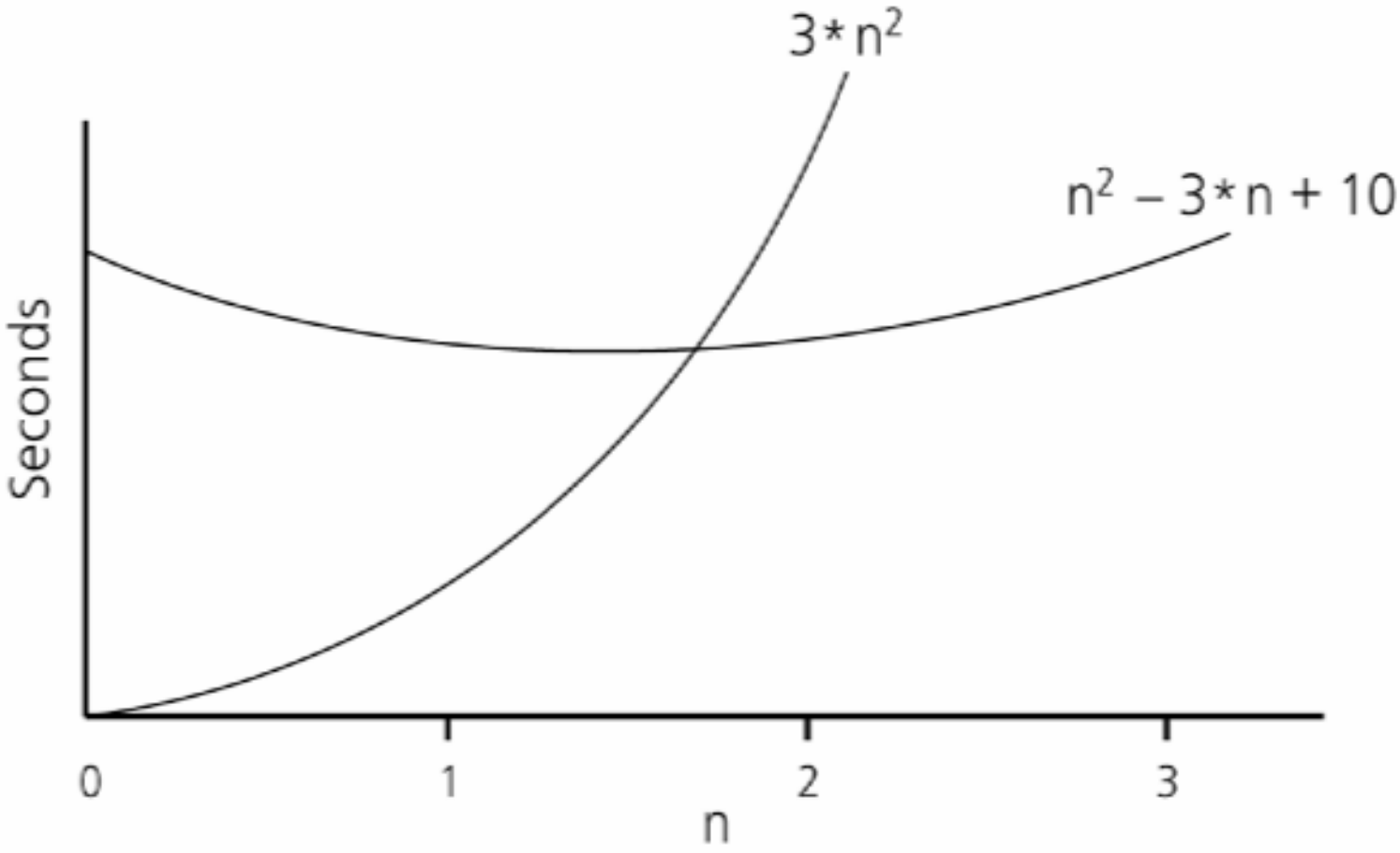
  then $(f_1 . f_2)(n)$ is $\Theta(g_1 . g_2(n))$

Definition of the Orders of an Algorithm

- **O-Notation** *(for two variables):* Given two real-valued functions g(n,m) and f(n,m) of two nonnegative variables n and m, we say that g(n,m) and f(n,m) are eventually nonnegative if there exists a positive number n_0 such that $g(n,m) \geq 0$ whenever $n \geq n_0$ and $m \geq n_0$. We say that f(n,m) is O(g(n,m) if the set O(g(n,m)) consists of those eventually nonnegative functions $f(n,m) \in O(g(n,m))$ if, and only if, there exists positive integers c and n_0, such that

- $f(n,m) \leq c.g(n,m)$ for all $n,m \geq n_0$.
- The sets $\Theta(g(n,m))$ and $\Omega(g(n,m))$ are similarly defined. The notions also extend to functions of more than two variables

Order of an Algorithm (cont.)



Best, Worst and Average Case Analysis

An algorithm can require different times to solve different problems of the same size.

Example: Searching an item in a list of n elements using sequential search. ➔ Cost: 1,2,...,n

We use the concepts of *best-case, worst-case,* and *average* complexity of an algorithm.

Best, Worst and Average Cases

Best-Case Analysis –The minimum amount of time that an algorithm require to solve a problem of size n.

The best case behavior of an algorithm is NOT so useful.

Let Ψn denote the set of all inputs of size n to an algorithm and $\tau(I)$ denotes the number of basic operations that are performed when the algorithm is executed with input I, that varies over all inputs of size n.

Defn: *Best-case complexity* of an algorithm is the function $B(n)$ such that $B(n)$ equals the minimum value of $\tau(I)$. That is, $B(n) = min \{ \tau(I) \mid I \in \Psi n \}$

Best, Worst and Average Cases

Worst-Case Analysis –The maximum amount of time that an algorithm require to solve a problem of size n.

This gives an upper bound for the time complexity of an algorithm.

Defn: *Worst-case* complexity of an algorithm is the function $W(n)$ such that $W(n)$ equals the maximum value of $\tau(I)$. That is, $W(n) = max\{\tau(I) \mid I \in \Psi n\}$

Normally, we try to find worst-case behavior of an algorithm.

Best, Worst and Average Cases

Average-Case Analysis –The average amount of time that an algorithm require to solve a problem of size n. Sometimes, it is difficult to find the average-case behavior of an algorithm.

We have to look at all possible data organizations of a given size n, and their distribution probabilities of these organizations.

Defn: *Average* complexity of an algorithm with finite set Ψn is defined as $A(n) = \sum \tau(I)\, p(I);\ I \in \tau_n$

Average complexity *A(n)* is defined as the expected number of basic operations performed.

• *p(I)* is the probability of occurring each $I \in \Psi n$ as the input to the algorithm. Alternatively,

• $A(n) = (t1+\ldots+tg(n)) / g(n)$, $(t1+\ldots+tg(n))$ covers all different cases.

Worst-case analysis is more common than average-case analysis.

Analysis of Algorithms

- We shall usually concentrate on finding only the *worst-case running time (W(n)),* that is, the longest running time for any input of size n. The reasons for that:
  - The worst-case running time of an algorithm is an upper bound on the running time for any input. Algorithm will not take longer (we hope).
  - For some algorithms, the worst-case occurs fairly often, e.g., searching databases, absent information.
  - The average case is often roughly as bad as the worst case.

Algorithm	*B(n)*	*A(n)*	*W(n)*
Linear Search	1	n	n
BinarySearch	1	logn	logn
Max,Min,MaxMin	n	n	n
InsertionSort	n	n^2	n^2
QuickSort	nlogn	nlogn	n^2
Towers	2^n	2^n	2^n

Figure: Order of best-case, average, and worst-case complexities

Analysis of Algorithms

An Example: Sorting Problem

Input: A sequence of n numbers $<a_1, a_2, \ldots, a_n>$.

Output: A permutation (reordering) $<a'_1, a'_2, \ldots, a'_n>$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

An example sorting algorithm: *Insertion sort:*

Insertion-Sort uses an ***incremental design*** approach: having sorted the subarrays A[1..j-1],

we insert the singles element A[j], into its proper place, yielding the sorted subarray A[1..j].

Array A[1..n] contains a sequence of length n that is to be sorted.

An example:

5	2	4	6	1	3
2	5	4	6	1	3
2	4	5	6	1	3
1	2	4	5	6	3
1	2	3	4	5	6

'_' represents the position of index j.

Insertion-Sort (A)	cost	times
// sort in increasing order //		
1 **for** j ← 2 **to** length[A]	c1	n
2 **do** key ← A[j]	c2	n-1
3 // insert A[j] ınto the sorted sequence A[1..j-1] //		
4 i ← j –1	c4	n-1
5 **while** i > 0 and A[i] > key	c5	$\sum_{2<j<n} t_j$
6 **do** A[i+1] ← A[i]	c6	$\sum_{2<j<n} (t_j-1)$
7 i ← i –1	c7	$\sum_{2<j<n} (t_j-1)$
8 A[i+1] ← key	c8	n-1

Analysis of Insertion sort algorithm:

T(n), the running time of insertion sort is the sum of products of the cost and times.

$T(n) = c1n+c2(n-1)+c4(n-1)+c5\sum_{2<j<n} (t_j)+c6\sum_{2<j<n} (t_j-1)+c7\sum_{2<j<n} (t_j-1)+c8(n-1)$

t_j =1 for j = 2,..,n for the *best* case, array is already sorted,

T(n) = c1n + c2(n-1) + c4(n-1) + c5(n-1) + c8(n-1 = (c1+c2+c4+c5+c8)n - (c1+c2+c4+c5+c8), that is, T(n) $\Theta(n)$

t_j =j for j = 2,..,n for the worst case, array is reverse sorted,

$\sum_{2<j<n} j = (n(n+1)/2) –1$ and $\sum_{2<j<n} j-1 = n(n-1)/2$

$T(n) = c1n + c2(n-1)+c4(n-1)+c5((n(n+1)/2)–1)+c6(n(n-1)/2)+c7(n(n-1)/2)+c8(n-1)$

$= (c5/2+c6/2+c7/2)n^2+(c1+c2+c4+c5/2-c6/2-c7/2+c8)n–(c2+c4+c5+c8)$,which is $\Theta(n^2)$.

Definition of the Orders of an Algorithm

o- Notation: f(n) is o(g(n)), "little-oh of g of n" as the set o(g(n)) = {f(n) : for any positive constant c > 0, there exists a constant $n_0 \geq 0$ such that

$$0 \leq f(n) < c.g(n) \text{ for all } n \geq n_0\}$$

- We use *o*-notation to denote an upper bound that is not asymptotically tight, whereas O-notation may be asymptotically tight. Intuitively, in the *o*-notation, the function f(n) becomes insignificantly relative to g(n) as n approaches infinity, that is,

$$\lim_{n \to \infty} f(n)/g(n) = 0$$

Ex: $2n = o(n^2)$, but $2n^2 \neq o(n^2)$

Ex: $n^2/2 \in o(n^3)$, since $\lim_{n \to \infty} (n^2/2)/ n^3 = \lim_{n \to \infty} 1 / 2n = 0$

Proposition: $f(n) \in o(g(n)) \Rightarrow O(f(n)) \subset O(g(n))$

Definition of the Orders of an Algorithm

ω - Notation: f(n) is ω(g(n)), "little-omega of g of n" as the set ω(g(n)) = {f(n) : for any positive constant c > 0, there exists a constant $n_0 \geq 0$ such that

$$0 \leq c.g(n) < f(n) \text{ for all } n \geq n_0\}$$

ω-notation denotes a lower bound that is not asymptotically tight. The relation f(n) = ω (g(n)) implies that,

$$\lim_{n \to \infty} f(n)/g(n) = \infty, \text{ if the limit exists.}$$

That is, f(n) becomes arbitrarily large relative to g(n) as n approaches infinity.

Ex: $n^2/2 = \omega(n)$, but $n^2/2 \neq \omega(n^2)$

Definition of the Orders of an Algorithm

Example: Factorials, $n!$, $n \geq 0$

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n*(n-1)! & \text{if } n > 0 \end{cases}$$

- A weak upper bound on the factorial function is $n! \leq n^n$, since each n terms in the factorial product is at most n.
- Stirling's approximation is $n! = \sqrt{2\pi n}\ (n/e)^n\ (1+\Theta(1/n))$, which gives a tighter upper bound, and a lower bound as well. Using Stirling's approximation, one can prove

  $n! = o(n^n)$, $n! = \omega(2^n)$, $\lg(n!) = \Theta(n \lg n)$

The following bounds also hold for all n: $\sqrt{2\pi n}\ (n/e)^n \leq n! \leq \sqrt{2\pi n}\ (n/e)^{n+(1/12n)}$

Definition of the Orders of an Algorithm

***Definition*:** Given the function g(n), we define ~(g(n) to be the set of all functions f(n) having the property that

$$\lim_{n \to \infty} f(n)/g(n) = 1,$$

If $f(n) \in \sim g((n))$, then we say that f(n) is strongly asymptotic to g(n) and denote this by writing $f(n) \sim g(n)$.

Ex: $n^2 = \sim (n^2)$, since $\lim_{n \to \infty} n^2/ n^2 = 1$,

Property: $f(n) \sim g(n) \Rightarrow f(n) \in \Theta(g(n))$

Definition of the Orders of an Algorithm

- ***Definition*:** A function f(n) is said to grow slower than a function g(n) (or, g(n) grows faster than f(n)), denoted by $f(n) \prec g(n)$, where the relation $\prec$ is a partial ordering, if $\lim_{n \to \infty} f(n)/g(n) = 0$.

For example, $g(n) = n^2$ grows faster than $f(n) = n$
since $\lim_{n \to \infty} n/ n^2 = \lim_{n \to \infty} 1/ n = 0$.

- We write $f(n) = o(g(n))$ if $f(n) \prec g(n)$, and
- $f(n) = O(g(n))$ if for some constant c and almost all $n \geq 0$ (for all, but finitely many $n \geq 0$), $f(n) \leq c.g(n)$.

Definition of the Orders of an Algorithm

- ✓ The following sequences of functions appear very often in the study of computational complexity:
  - Poly-log sequence: $\{(\mathbf{lgn})^{\mathbf{i}} \mid i = 1, 2, \ldots\}$
  - Polynomial sequence: $\{\mathbf{n}^{\mathbf{i}} \mid i = 1, 2, \ldots\}$
  - Subexponential sequence: $\{_{\mathbf{n}}\mathbf{(logn)}^{\mathbf{i}} \mid i = 1, 2, \ldots\}$
  - Exponential sequence: $\{(2^{\mathbf{ni}} \mid i = 1, 2, \ldots\}$
  - Superexponential sequence: $\{_{\mathbf{2}}\mathbf{n}^{\mathbf{i}} \mid i = 1, 2, \ldots\}$

Definition of the Orders of an Algorithm

- These sequences satisfy the following properties:
  - ✓ In each sequence, if i < j, then i^{th} function grows slower than the j^{th} function. For instance, $\mathbf{_n(logn)^3 \prec _n(logn)^4}$ **.**
  - ✓ For each sequences, every function in the former sequence grows slower than any function in the latter sequence (except for the first function of the last sequence). For instance, $lgn^{64} \prec n^{10} \prec \mathbf{_n(logn)^4} \prec 2^{3n} \prec {}_2n^2$.
- These five sequences do not contain all possible measurement for growth rates. There are other examples not in these sequences.
  - ✓ For example, the function $T(n) = 2^{\sqrt{logn}}$ grows slower than every function in the polynomial sequence, but faster than every function in the poly-log sequence. That is, $(logn)^i \prec 2^{\sqrt{logn}} \prec n$.

  We can show this by the following:

  $\lim_{n \to \infty} (logn)^i / 2^{\sqrt{logn}} = 0$ and $\lim_{n \to \infty} 2^{\sqrt{logn}} / n = 0$

Orders of Algorithms

Theorem: Let the polynomial

$P(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_2 n^2 + a_1 n + a_0$

be degree of k, $a_k > 0$. Then, P(n) is $\Theta(n^k)$.

Proof: It is sufficient to show that $P(n) \in O(n^k)$ and $P(n) \in \Omega(n^k)$. We first show that $P(n) \in O(n^k)$.

$P(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_2 n^2 + a_1 n + a_0$

$\leq a_k n^k + |a_{k-1}| n^k + \ldots + |a_1| n^k + |a_0| n^k \leq (|a_0| + |a_1| + |a_2| + \ldots + |a_{k-1}| + a_k) n^k$

Thus, if $c = (|a_0| + |a_1| + |a_2| + \ldots + |a_{k-1}| + a_k)$, then $P(n) \leq c.n^k$ for all $n \geq n_0 = 1$, so that $p(n) \in O(n^k)$.

We now show that $P(n) \in \Omega(n^k)$.

$P(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_{k-1} n^{k-1} + a_k n^k$

$\geq a_k n^k - |a_{k-1}| n^{k-1} - \ldots - |a_1| n^1 - |a_0| n^0 = (a_k/2) n^k + [(a_k/2) n^k - |a_{k-1}| n^{k-1} - \ldots - |a_2| n^2 - |a_1| n - |a_0|]$

$\geq (a_k/2) n^k + [(a_k/2) n^k - |a_{k-1}| n^{k-1} - \ldots - |a_2| n^{k-1} - |a_1| n^{k-1} - |a_0| n^{k-1}]$

$= (a_k/2) n^k + [(a_k/2) n - (|a_{k-1}| + \ldots + |a_2| + |a_1| + |a_0|)] n^{k-1}$

Thus, if $c = a_k/2$ and $n_0 = (|a_{k-1}| + \ldots + |a_2| + |a_1| + |a_0|)] (2/a_k)$, the term inside the bracket in the last equation is nonnegative for all $n \geq n_0$, and hence $P(n) \geq c.n^k$ for all $n \geq n_0$. This shows that $P(n) \in \Omega(n^k)$.

Therefore, P(n) is $\Theta(n^k)$.

Orders of Algorithms

Proposition: $P(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$ be any polynomial of degree of k, $a_k > 0$.
Then, $P(n) \sim a_k n^k$

Proof: Consider the ration $(a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0) / a_k n^k$ as n tends to go infinity:

$$\lim_{n \to \infty} (a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0) / a_k n^k$$
$$= \lim_{n \to \infty} (1 + a_{k-1} / a_k n + \ldots + a_1 / a_k n^{k-1} + a_0 / a_k n^k) = 1.$$

Proposition: For any nonnegative real constants k and a, with a >1, $O(n^k) \subset O(a^n)$.

Proof: Consider the ration n^k/a^n. Repeated applications of L'Hopital's Rule yields

$$\lim_{n \to \infty} n^k/a^n = \lim_{n \to \infty} kn^{k-1}/(\ln a)a^n$$
$$= \lim_{n \to \infty} k(k-1)n^{k-2}/(\ln a)^2 a^n$$
…
…
$$= \lim_{n \to \infty} k!/(\ln a)^k a^n$$
$$= 0.$$

Hence, $n^k \in o(a^n)$.

Since, if $f(n) \in o(g(n)) \Rightarrow O(f(n) \subset O(g(n))$, it is true that $O(n^k) \subset O(a^n)$.

Orders of Algorithms

Proposition: $L(n) = \log(n!) \in \Theta(n\log n)$.

Proof: $L(n) = \log 1 + \log 2 + \ldots + \log n$.

Clearly, $L(n) \in O(n\log n)$, since

$L(n) = \log 1 + \log 2 + \ldots + \log n \leq \log n + \log n + \ldots + \log n = n\log n$,

So that $L(n) \in O(n\log n)$.

It remains to show that $L(n) \in \Omega(n\log n)$. Let $m = \lfloor n/2 \rfloor$. We have

$L(n) = (\log 1 + \log 2 + \ldots + \log m) + [\log(m+1) + \log(m+2) + \ldots + \log n]$

$\geq \log(m+1) + \log(m+2) + \ldots + \log n$

$\geq \log(m+1) + \log(m+1) + \ldots + \log(m+1)$

$= (n-m)\log(m+1)$

$\geq (n/2)\log(n/2) = (n/2)(\log n - \log 2)$

Clearly, $(n/2)(\log n - \log 2) \geq (n/2)[\log n - (\tfrac{1}{2})\log n)$ for all $n \geq n_0$, where n_0 is sufficiently large (the constant n_0 depends on the base chosen).

Thus, $L(n) \geq (1/4)(n\log n)$, so that $L(n) \in \Omega(n\log n)$.

➢ Therefore, $\log(n!) \in \Theta(n\log n)$.

Orders of Algorithms

Theorem: The function $h(n) = \sum_{1 \le i \le n} (1/i) = 1 + ½ + \ldots + 1/n$ is P(n) is $\Theta($ lgn).

Proof: Values c_1, c_2, and n_0 can be arrived at by finding upper and lower bounds for h(n) using only terms of the form $1/2^j$ as follows:

$1 + ½ + ¼ + ¼ + 1/8 + 1/8 + 1/8 + 1/8 + \ldots + 1/2^k$

$\le 1 + ½ + 1/3 + ¼ + 1/5 + \ldots + 1/2^k = h(2^k)$

$\le 1 + ½ + ½ + ¼ + ¼ + ¼ + ¼ + 1/8 + \ldots. + 1/2^k$

The first and third sums can be evaluated and inequalities become:

$$1 + k/2 \le h(2^k) \le k + 1/2^k$$

For an integer n, let $k = \lfloor \lg n \rfloor$, $2^k = n$; then

$$1 + \lg n / 2 \le h(n) \le \lg n + 1/n,$$

If we use the following inequilities: $\lg n > 1/n$ and $\lg n / 2 < 1 + k/2$, then

$$½(\lg n) \le h(n) \le 2(\lg n)$$

Therefore, $h(n) = \Theta(\lg n)$, $c_1 = ½$, $c_2 = 2$, $n_0 = 1$.

Orders of Algorithms

Arithmetic sequence: $S(n) = \sum_{1 \le i \le n-1} a + id = na + d \sum_{1 \le i \le n-1} i$,

where a and d are real constants; d is called the difference of the sequence, since consecutive terms differ by d. If we solve this, we find $S(n) = na + d.n.(n-1)/2$. Here S is a polynomial of degree 2, which implies that S is $\Theta(n^2)$

Geometric sequence: $G(n) = \sum_{0 \le i \le n-1} ar^i = ar^0 + ar + \ldots + ar^{n-1}$ where r is a real constant, called the ratio of the sequence. Multiplying both sides by r,

$rG = \sum_{0 \le i \le n-1} ar^{i+1} = ar + ar^2 + \ldots + ar^n = -a + a + ar + ar^2 + \ldots + ar^n$

$rG = G + ar^n - a$

$G = (ar^n - a)/(r-1) = a(r^n - 1) / (r-1)$ if $r \neq 1$.

Note that $G = an$ if $r = 1$.

In conclusion

G is $\Theta(n)$ if $r = 1$

G is $\Theta(r^n)$ if $r \neq 1$

G is $O(1)$ if $r < 1$

Orders of Algorithms

Growth rates of some other functions

1. $f(n) = \sum_{1 \le i \le n} i = n(n+1)/2$
2. $f(n) = \sum_{1 \le i \le n} i^2 = n(n+1)(2n+1)/6$

2'. $f(n) = \sum_{1 \le i \le n} i^k = [1/(k+1)]\ (n^{k+1})$

3. $f(n) = \sum_{0 \le i \le n} x^i = (x^{n+1} - 1)/\ (x - 1)$
4. $f(n) = \sum_{0 \le i \le n} 2^i = (2^{n+1}-1)/\ (2-1) = 2^{n+1}-1$

Sample Algorithm analysis:

The sum rule:

Given the a functions $f_1, f_2, \ldots, f_a$, where a is constant, if $f_i = O(g_i)$, $\forall i \le n$, then

$$\sum_{1 \le i \le a} f_i = O(g_m)$$

where g_m is the fastest growing of the functions $g_1, g_2, \ldots, g_a$.

The product rule:

If $fi = O(g_i)$, $\forall i \le n$, where a is constant, then

$\Pi_{1 \le i \le a} f_i = O(\Pi_{1 \le i \le a} g_i)$

Some Mathematical Facts

- Some mathematical equalities are:

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n = \frac{n*(n+1)}{2} \approx \frac{n^2}{2}$$

$$\sum_{i=1}^{n} i^2 = 1 + 4 + \ldots + n^2 = \frac{n*(n+1)*(2n+1)}{6} \approx \frac{n^3}{3}$$

$$\sum_{i=0}^{n-1} 2^i = 0 + 1 + 2 + \ldots + 2^{n-1} = 2^n - 1$$

Growth-Rate Functions – Example1

	Cost	Times
`i = 1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`  i = i + 1;`	c4	n
`  sum = sum + i;`	c5	n
`}`		

T(n) = c1 + c2 + (n+1)*c3 + n*c4 + n*c5

= (c3+c4+c5)*n + (c1+c2+c3)

= a*n + b

➔ So, the growth-rate function for this algorithm is **O(n)**

Growth-Rate Functions – Example2

	Cost	Times
`i=1;`	c1	1
`sum = 0;`	c2	1
`while (i <= n) {`	c3	n+1
`  j=1;`	c4	n
`  while (j <= n) {`	c5	n*(n+1)
`  sum = sum + i;`	c6	n*n
`  j = j + 1;`	c7	n*n
`  }`		
`  i = i +1;`	c8	n
`}`		

T(n) = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8

= $(c5+c6+c7)*n^2 + (c3+c4+c5+c8)*n + (c1+c2+c3)$

= $a*n^2 + b*n + c$

➔ So, the growth-rate function for this algorithm is **$O(n^2)$**

Growth-Rate Functions – Example3

	Cost	Times
`for (i=1; i<=n; i++)`	c1	n+1
`  for (j=1; j<=i; j++)`	c2	$\sum_{j=1}^{n}(j+1)$
`    for (k=1; k<=j; k++)`	c3	$\sum_{j=1}^{n}\sum_{k=1}^{j}(k+1)$
`      x=x+1;`	c4	$\sum_{j=1}^{n}\sum_{k=1}^{j}k$

$$T(n) = c1*(n+1) + c2*\left(\sum_{j=1}^{n}(j+1)\right) + c3*\left(\sum_{j=1}^{n}\sum_{k=1}^{j}(k+1)\right) + c4*\left(\sum_{j=1}^{n}\sum_{k=1}^{j}k\right)$$

$$= a*n^3 + b*n^2 + c*n + d$$

➔ So, the growth-rate function for this algorithm is **$O(n^3)$**

Orders of Algorithms

Example:

```
For k = 1 to n/2 do
  {
   ….
  }
For j = 1 to n*n do
  {
   ----
  }
```

$\sum^{n/2}_{k=1} c + \sum^{n*n}_{j=1} c = c.n/2 + c.n^2 = O(n^2)$

Example:

```
For k = 1 to n/2 do
{
    For j = 1 to n*n do
    {
----
    }
}
```

$\sum^{n/2}_{k=1} \sum^{n*n}_{j=1} c = c.n/2 * n^2 = O(n^3)$

Orders of Algorithms

Example:

```
i = n
while i > 1 do
{
   i = i div 2
}
```

64 32 16 8 4 2 1

$\sum^{n}_{i=2}(1/2^i) = O(\log_2 n)$

Example:

```
For i = 1 to n do
   For j = i to n do
         For k = i to j do
                  m =m + i + j + k
```

$\sum^{n}_{i=1}\sum^{n}_{j=i}\sum^{j}_{k=i}3 \approx n^3/10$ additions, that is $O(n^3)$

$\sum^{n}_{i=1}\sum^{n}_{j=i}3(j-i+1) = \sum^{n}_{i=1}[3\sum^{n}_{j=i}j - 3\sum^{n}_{j=i}i + 3\sum^{n}_{j=i}1]$

$= \sum^{n}_{i=1}[3(\sum^{n}_{j=1}j - \sum^{i-1}_{j=1}j) - 3i(n-i+1) + 3(n-i+1)]$

$= \sum^{n}_{i=1}[3[(n(n+1))/2 - (i(i-1))/2)] - 3i(n-i+1) + 3(n-i+1)]$

$\approx n^3/10$ additions, that is, $O(n^3)$

Growth-Rate Functions – Recursive Algorithms

```
void hanoi(int n, char source, char dest, char spare) {      Cost
  if (n > 0) {                                                c1
    hanoi(n-1, source, spare, dest);                          c2
     cout << "Move top disk from pole " << source             c3
          << " to pole " << dest << endl;
    hanoi(n-1, spare, dest, source);                          c4
} }
```

- The time-complexity function T(n) of a recursive algorithm is defined in terms of itself, and this is known as **recurrence equation** for T(n).
- To find the growth-rate function for that recursive algorithm, we have to solve that recurrence relation.

Growth-Rate Functions – Hanoi Towers

- What is the cost of `hanoi(n,'A','B','C')`?

when n=0

$T(0) = c1$

when n>0

$T(n) = c1 + c2 + T(n-1) + c3 + c4 + T(n-1)$

$= 2*T(n-1) + (c1+c2+c3+c4)$

$= \mathbf{2*T(n-1) + c}$ ← recurrence equation for the growth-rate function of hanoi-towers algorithm

- We have to solve this recurrence equation to find the growth-rate function of hanoi-towers algorithm

Recurrences

There are four methods for solving recurrences, which is for obtaining asymptotic Θ and O bounds on the solution.

1. The *iteration method* converts the recurrence into a summation and then relies on techniques for bounding summations to solve the recurrence.
2. In the *substitution method*, we guess a bound and then use mathematical induction to prove our guess correct.
3. The *master method* provides bounds for recurrences of the form

   $T(n) = aT(n/b) + f(n)$ where $a \geq 1$, $b \geq 2$, and $f(n)$ is a given function.
4. Using the *characteristic equation*.

Recurrences (cont.)

2 **Solving recurrences by iteration method:** converts the recurrence into a summation and then relies on techniques for bounding summations to solve the recurrence.

Example: The recurrence arises for a recursive program that makes at least n number of disk moves from one peg to another peg and then move them to the third one.

$t_n = 2t_{n-1} + 1, \qquad n \geq 2,$ subject to $t_1 = 1.$

$t_n = 2t_{n-1} + 1 = 2^2 t_{n-2} + 2 + 1 = \ldots = 2^{n-1}t_1 + 2^{n-2} + \ldots + 2 + 1$
$= 2^n - 1. \quad \rightarrow t_n \in O(2^n).$

- This is the minumum number of moves required to transfer n disks from one peg to another. Indeed, before this move, we must first move all other disks to the third disk and also, after the move of the largest disk, we must move all other disks to the top of the largest disk.
- Therefore, the minimum number f(n) of moves satisfies the following inequality: $f_1 = 1$ and $f_n \geq 2f_{n-1} + 1$, $n \geq 2$.

  Thus, $f_n \geq t_n = 2^n - 1 \rightarrow f_n \in O(2^n)$

For n = 64, it takes $584 * 10^6$ years if every 1000 moves takes 1 seconds.

Recurrences (cont.)

2 **Example:** The recurrence arises for a recursive program that loops through the input to eliminate one item:

$t_n = t_{n-1} + n \qquad n \geq 2,$ subject to $t_1 = 1.$

To solve such a recurrence, we "telescope" it by applying it to itself, as follows:

$t_n = t_{n-1} + n$

$t_n = t_{n-2} + (n-1) + n$

$t_n = t_{n-3} + (n-2) + (n-1) + n$

……

$t_n = t_1 + 2 + \ldots + (n-2) + (n-1) + n$

$t_n = 1 + 2 + \ldots + (n-2) + (n-1) + n$

$= n(n+1)/2$

Hence, $t_n \in \Theta(n^2/2)$

Recurrences (cont.)

Example: The recurrence arises for a recursive program that has to make a linear pass through the input, before, during, or after it is split into two halves:

$t_n = 2t_{n/2} + n \qquad n \geq 2,$ subject to $t_1 = 0.$

To solve such a recurrence, we assume that $n = 2^k$ ($k = \lg n$) and then "telescope" it by applying it to itself, as follows:

$t_n = 2(2t_{n/4} + n/2) + n$

$= 4t_{n/4} + 2n/2 + n = 4t_{n/4} + n\,(1+ 1)$

$= 4(2t_{n/8} + n/4) + 2n/2 + n$

$= 8t_{n/8} + 4n/4 + 2n/2+ n = 8t_{n/8} + n\,(1+ 1 + 1)$

….

$= nt_{n/n} + n(1+\ldots+1)$

$= 2^k t_1 + n.k$

$t_n = 0+ n(\lg n)$

Hence, $t_n \in \Theta(n \lg n)$

Recurrences (cont.)

2 Solving recurrences by substitution (guess) method: Guesses a bound and then use mathematical induction to prove the guess correct.

Example: The recurrence arises for a recursive program that halves the input in one step:

$t_n = t_{n/2} + c$ $\quad n \geq 2$, subject to $t_1 = 1$.

$t_2 = 1 + c$

$t_4 = 1 + 2c$, $t_8 = 1 + 3c$, ….

$t_k = 1 + kc$, where $n = 2^k$, $t_n = 1 + c.\lg n$, Therefore, $t_n \in \Theta(\lg n)$

Proof by induction:

Base case: n=1, $t_1 = 1$ is given

Induction case: **n=k,** $t_k = 1 + kc$**,** where $n = 2^k$, for $t_k = t_{k/2} + c$, is true

Proof: n=k+1, $t_{k+1} = 1 + (k+1)c = 1 + kc + c$???

From the given recurrence

($t_{k+1} = T(2^{k+1}) = T(2^{k+1}/2) + c = T(2^k) + c) = t_k + c = 1 + kc + c$ (by the induction step).

Therefore, $t_n = 1 + c.\lg n$, for $n = 2^k$

Recurrences (cont.)

- **Example:** The recurrence arises for a recursive program that halves the input in one step, applies to 3 subproblems and loops through each input:

$t_n = 3t_{n/2} + cn$ $\quad n \geq 2$, subject to $t_1 = 1$.

$t_2 = 3 + 2c$

$t_4 = 9 + 10c$

$t_8 = 27 + 38c$

$t_{16} = 81 + 130c$

….

$t_2k = 3^k + 2^k c\,[(3/2)^k - 1] / [(3/2) - 1]$, where $n = 2^k$ and $k = \lg n$

$t_n = 3^{\lg n} + cn(3^{\lg n}/n - 1) / 1/2$

$t_n = n^{1.59} + 2cn\,(n^{0.59} - 1)$, where $3^{\lg n} = n^{\lg 3} = 3^{1.59}$

$= n^{1.59}\,(1 + 2c) - 2cn$

$t_n \in \Theta(n^{1.59})$

Recurrences (cont.)

3 Solving recurrences by master method:

The master method depends on the following theorem:

Master theorem: Let $a \geq 1$ and $b \geq 2$ be constants and let T(n) be a function, and let T(n) be defined on the nonnegative integers by the recurrence

$T(n) = aT(n/b) + cn^i$ $(n > n_0)$, when n/ n_0 is a power of b.

Show that the exact order of T(n) is given by

$$T(n) \in \begin{cases} \Theta(n^i \log_b n) & \textbf{case 1: } a = b^i, i = \log_b a \\ \Theta(n^{\log b^a}) & \textbf{case 2: } a > b^i, i < \log_b a \\ \Theta(n^i) & \textbf{case 3: } a < b^i, i > \log_b a \end{cases}$$

Example: The recurrence arises for a recursive program that halves the input, but perhaps must examine every item in the input:

$t_n = t_{n/2} + n$ for $n \geq 2$, subject to $t_1 = 0$.

For this recurrence, $a = 1$, $b = 2$, $i = 1$, and thus case 3: $a < b^i$ applies and

$t_n \in \Theta(n)$

Example: The recurrence arises from matrix multiplication:

$t_n = 8t_{n/2} + cn^2$ $n \geq 2$, subject to $t_1 = 1$.

For this recurrence, $a = 8$, $b = 2$, $i = 2$, and thus case 3: $a > b^i$ applies and

$t_n \in \Theta(n^{\log 2^8}) = \Theta(n^3)$

Recurrences (cont.)

Master Theorem *(Another Version (text book):* let $a \geq 1$ and $b>1$ be constants, let f(n) be a function, and T(n) be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n), \; a \geq 1 \text{ and } b > 1.$$

- ✓ *The function divides the problem of size n into a subproblems, each of size n/b. The subproblems are solved recursively each in time T(n/b). The cost of dividing the problem and combining the results of the subproblems is described by the f(n). We interpret n/b to mean either $\lceil n/b \rceil$ or $\lfloor n/b \rfloor$, which does not affect the asymptotic behaviour of the recurrence.*

T(n) can be bounded asymptotically as follows:

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

- ✓ Intuitively, the solution to the recurrence is determined by the larger of the two functions, f(n) and $n^{\log_b a}$. In case 1, not only must f(n) be smaller than $n^{\log_b a}$, it must be polynomially smaller. That is, f(n) must be asymptotically smaller than $n^{\log_b a}$ by a factor of n^{ε} for some constant $\varepsilon > 0$. Similar technicality is valid for case 3, in addition satisfy the "regularity" condition that $af(n/b) \leq cf(n)$.

Recurrences (cont.)

The three cases do not cover all the possibilities for f(n).

Example: The recurrence arises for a recursive program that divides the input into three and solves 9 of sub probles, and perhaps must examine every item in the input:

$t_n = 9t_{n/3} + n$, for $n \geq 3$, subject to $t_1 = 1$.

Here a = 9, b = 3, f(n) = n and $n^{\log}b^a = n^{\log}3^9 = \Theta(n^2)$.

Since $f(n) = O(n^{\log}3^{9-\varepsilon})$, where $\varepsilon = 1$, we can apply case 1 of the master theorem and conclude that the solution is $t_n \in \Theta(n^2)$.

Example: A recurrence arises from a search algorithm.

$t_n = t_{2n/3} + 1$, $n \geq 3$, subject to $t_2 = 1$.

Here a = 1, b = 3/2, thus $n^{\log}b^a = n^{\log}3/2^1 = \Theta(n^0) = 1$, and f(n) = 1.

Therefore, case 2 applies, since $f(n)=\Theta(n^{\log}b^a)=\Theta(1)$, and the solution to the recurrence is $t_n = \Theta(\lg n)$.

Recurrences (cont.)

Example: The recurrence arises from a recursive algorithm.

$t_n = 3t_{n/4} + n\lg n, \quad n \geq 4,$ subject to $t_1 = 1$.

For this recurrence, $a = 3$, $b = 4$, and thus $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$, $f(n) = n\lg n$.

Since $f(n) = \Omega(n^{\log_4 3+\varepsilon})$, where $\varepsilon \approx 0.2$, case 3 applies if we can show that the regularity condition holds for $f(n)$. For sufficiently large n, $af(n/b) = 3(n/4)\lg(n/4) \leq (3/4)n\lg n = cf(n)$ for $c = ¾ < 0$.

Consequently, by case 3, the solution to the recurrence is, $t_n \in \Theta(n\lg n)$.

Example: The master method does not apply the following recurrence.

$t_n = 2t_{n/2} + n\lg n$. In this $a = 2$, $b = 2$, $f(n) = n\lg n$, and $n^{\log_b a} = n$.

It seems that case 3 should apply, since $f(n) = n\lg n$ is asymptotically larger than $n^{\log_b a} = n$ but not polynomially larger. The ratio $f(n)/n^{\log_b a} = (n\lg n)/n = \lg n$ is asymptotically less than n^{ε} for any positive constant ε. Consequently, the recurrence falls into the gap between case 2 and case 3. Therefore, the master method cannot be used to solve this recurrence.

Recurrences (cont.)

4 Solving Recurrences Using the Characteristic Equation

- **Homogeneous Recurrences:** First we introduce homogeneous linear recurrences with constant coefficients, that is, recurrence of the form
- $a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = 0$ (1)
- where

i. the t_i are the values we are looking for. The recurrence is linear since it does not contain terms of the form $t_i t_{i+j}$, t_i^2, and so on;

ii. the coefficients a_i are constants; and

iii. the recurrence is homogeneous because the linear combination of the t_i is equal to zero.

- **Example**: Consider the familiar recurrence for the Fibonacci sequence.
- $f_n = f_{n-1} + f_{n-2}$ which is $f_n - f_{n-1} - f_{n-2} = 0$,
- Therefore, Fibonacci sequence corresponds to a homogeneous linear recurrence with constants coefficients $k = 2$, $a_0 = 1$ and $a_1 = a_2 = -1$.
- It is interesting to note that any linear combination of solutions is itself a solution.

Recurrences (cont.)

- Trying to solve a few easy examples of recurrences of the form (1) by intelligent guesswork (or after a while intuition) may suggest we look for a solution of the form
- $t_n = x^n$
- where x is a constant as yet unknown. If we try this solution in (1), we obtain

  $a_0x^n + a_1x^{n-1} + \ldots + a_kx^{n-k} = 0$
- This equation is satisfied if x = 0, a trivial solution of no interest. Otherwise, the equation is satisfied if and only if
- $a_0x^k + a_1x^{k-1} + \ldots + a_k = 0.$
- This equation of degree k in x is called the *characteristic equation* of the recurrence of (1) and

  $P(x) = a_0x^k + a_1x^{k-1} + \ldots + a_k$
- is called its *characteristic polynomial.*
- Recall that the fundamental theorem of algebra states that any polynomial p(x) of degree k has exactly k roots (not necessarily distinct), which means that it can be factorized as a product of k monomials
- $P(x) = \Pi_{1 \le i \le k} (x - r_i)$
- where the r_i may be complex numbers. Moreover, these r_i are the only solutions of the equation p(x).

Recurrences (cont.)

Suppose that the k roots $r_1, r_2, \ldots, r_k$ of this characteristic equation are *all distinct*. Since $p(r_i) = 0$, it follows that $x = r_i$ is a solution to the characteristic equation and therefore r_i^n is a solution to the recurrence. Since any linear combination of solutions is also a solution, we conclude that the following satisfies the recurrence (1) for any choice of constants $c_1, c_2, \ldots, c_k$:

$$t_n = \sum c_i r_i^n$$

- of terms r_i^n is a solution of the recurrence (1), where the k constants $c_1, c_2, \ldots, c_k$ are determined by the initial conditions. We need exactly k initial conditions to determine the values of these k constants. The remarkable fact, which we do not prove here, is that (1) has only solutions of this form.
- **Example**: Consider the recurrence

  $t_n - 3t_{n-1} - 4t_{n-2} = 0 \quad n \geq 2$, subject to $t_0 = 0, t_1 = 1$.
- The characteristic equation of the recurrence is

  $x^2 - 3x - 4 = 0$, whose roots are -1 and 4.

  The general solution therefore has the form $t_n = c_1(-1)^n + c_2 4^n$.
- The initial conditions give

  $c_1 + c_2 = 0 \quad n = 0$,

  $-c_1 + 4c_2 = 1 \quad n = 1$, that is, $c_1 = -1/5, c_2 = 1/5$
- We finally obtain $t_n = 1/5\,[4^n - (-1)^n]$, that is, $t_n \in \Theta(4^n)$.

Recurrences (cont.)

Example: (Fibonacci) Consider the recurrence

$t_n - t_{n-1} - t_{n-2} = 0 \qquad n \geq 2$, subject to $t_0 = 0, t_1 = 1$.

The characteristic polynomial is

$P(x) = x^2 - x - 1 = 0$

Whose roots are $(1+\sqrt{5})/2$ and $(1-\sqrt{5})/2$.

The general solution is therefore of the form

$t_n = c_1((1+\sqrt{5})/2)^n + c_2((1-\sqrt{5})/2)^n$.

The initial conditions give

$c_1 + c_2 = 0 \quad n = 0,$

$((1+\sqrt{5})/2)c_1 + ((1-\sqrt{5})/2)c_2 = 1 \quad n = 1,$

Solving these equations, we obtain $c_1 = 1/\sqrt{5}$, $c_2 = -1/\sqrt{5}$.

We finally obtain

$t_n = 1/\sqrt{5}\,[((1+\sqrt{5})/2)^n - ((1-\sqrt{5})/2)^n]$,

which is de Moivre's famous formula for the Fibonacci sequence.

Therefore, $t_n \in \Theta((1+\sqrt{5})/2)^n$.

Recurrences (cont.)

Now suppose that the roots of the characteristic equation are *not all distinct*. Then

$t_n = nr^n$ is also a solution of (1). That is,

$a_0 nr^n + a_1(n-1)r^{n-1} + \ldots + a_k(n-k)r^{n-k} = 0,$

More generally, if m is the multiplicity of the root r, then

$t_n = r^n$, $t_n = nr^n$, $t_n = n^2r^n$, …, $t_n = n^{m-1}r^n$ are all possible solutions of (1).

The general solution is a linear combination of these terms and of the terms contributed by the other roots of the characteristic equation. Once again there are k constants to be determined by the initial conditions.

Example: Solve the following recurrence:

$t_n = 5t_{n-1} - 8t_{n-2} + 4t_{n-3}$ $\quad n \geq 3,$ subject to $t_0 = 0$, $t_1 = 1$, and $t_2 = 2$.

The characteristic equation is

$x^3 - 5x^2 + 8x - 4 = 0,$ or $(x-1)(x-2)^2 = 0.$

The roots are 1 and 2 (of multiplicity 2). The general solution is therefore

$t_n = c_1 1^n + c_2 2^n + c_3 n 2^n.$ The initial conditions give

$$c_1 + c_2 = 0 \quad n = 0,$$
$$c_1 + 2c_2 + 2c_3 = 1 \quad n = 1,$$
$$c_1 + 4c_2 + 8c_3 = 2 \quad n = 2,$$

From which we find $c_1 = -2$, $c_2 = 2$, $c_3 = -1/2$. Therefore, $t_n = 2^{n+1} - n2^{n-1} - 2$.

$t_n \in \Theta(n2^n)$.

Recurrences (cont.)

Inhomogeneous Recurrences: We now consider recurrences of a slightly more general form.

$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b^n p(n)$ (2)

The left hand side is the same as (1), but on the right hand side we have $b^n p(n)$, where b is a constant; an

p(n) is a polynomial in n of degree d.

Example: Solve the following recurrence: $t_n - 2t_{n-1} = 3^n$.

In this case b =3 and p(n) = 1, a polynomial of degree d=0. A little manipulation allows us to reduce this example to the form (1). To see this, we first multiply the recurrence by 3, obtaining

$3t_n - 6t_{n-1} = 3^{n+1}$.

If we replace n by n+1 in the original recurrence, we get

$t_{n+1} - 2t_n = 3^{n+1}$.

Finally, subtracting these two equations, we have

$t_{n+1} - 5t_n + 6t_{n-1} = 0$,

which can be solved by the above method. The characteristic equation is

$x^2 - 5x + 6 = 0$. That is, $(x-2)(x-3) = 0$.

Intuitively we can see that the factor (x-2) corresponds to the left-hand side of the original recurrence, whereas the factor (x-3) has appeared as a result of our manipulation to get rid of the right-hand side.

Recurrences (cont.)

Generalizing this approach, we can show that to solve (2) it is sufficient to take the following characteristic equation

$(a_0x^k + a_1x^{k-1} + \ldots + a_k)(x-b)^{d+1} = 0.$

Once this equation is obtained, proceed as in the homogeneous case.

Example: The number of movements of a ring required in the Towers of Hanoi problem is given by the following recurrence relation:

$t_n = 2t_{n-1} + 1,\ n \geq 1,$ subject to $t_0 = 0.$

The recurrence can be written

$t_n - 2t_{n-1} = 1,$ which is of the form (2) with b = 1, p(n) = 1, a polynomial of degree 0. The characteristic equation is therefore

$(x-2)(x-1) = 0.$

The roots of this equation are 1 and 2, so the general solution of the recurrence is

$t_n = c_1 1^n + c_2 2^n.$

We need two initial conditions. We know that $t_0 = 0$; to find a second initial condition we use the recurrence itself to calculate $t_1 = 2t_0 + 1 = 1.$

We finally have $c_1 + c_2 = 0 \quad n = 0$

$c_1 + 2c_2 = 1 \quad n = 1$

From which we obtain the solution, $t_n = 2^n - 1$, where $c_1 = -1$, $c_2 = 1$.

Then we can conclude that $t_n \in \Theta(2^n)$.

Recurrences (cont.)

A further generalization of the same type of argument allows us finally to solve recurrences of the form

$a_0t_n + a_1t_{n-1} + \ldots + a_kt_{n-k} = b_1^n p_1(n) + b_2^n p_2(n) + \ldots$ (3)

where the b_i are distinct constants and $p_i(n)$ are polynomials in n respectively of degree d_i. It suffices to write the characteristic equation

$(a_0x^k + a_1x^{k-1} + \ldots + a_k)(x - b_1)^{d1+1}(x - b_2)^{d2+1} \ldots = 0,$

Example: Solve the following recurrence:

$t_n = 2t_{n-1} + n + 2^n \qquad n \geq 1,$ subject to $t_0 = 0.$

$t_n - 2t_{n-1} = n + 2^n,$ which is of the form (3) with $b_1=1$, $p_1(n) = n$, $b_2=2$, $p_2(n) = 1$. The characteristic equation is $(x-2)(x-1)^2(x-2) = 0$, Roots: 1, 2, 2.

The general solution of the recurrence is therefore of the form

$$t_n = c_1 1^n + c_2 n 1^n + c_3 2^n + c_4 n 2^n$$

Using the recurrence, we can calculate $t_1 = 3$ ($t_1 = 0+1+2^1 = 3$), $t_2 = 12$, $t_3 = 35$.

$c_1 + c_3 = 0 \quad n = 0,$

$c_1 + c_2 + 2c_3 + 2c_4 = 3 \quad n = 1$

$c_1 + 2c_2 + 4c_3 + 8c_4 = 12 \quad n = 2$

$c_1 + 3c_2 + 8c_3 + 24c_4 = 35 \quad n = 3,$ arriving at $t_n = -2 - n + 2^{n+1} + n2^n$

Therefore, $t_n \in \Theta(n2^n)$.

Recurrences (cont.)

Change of Variable:

- We write T(n) for the term of a general recurrence, and t_k for the term of a new recurrence obtained by a change of variable.

Example: Solve the recurrence if n is a power of 2 and if

$$T(n) = 4T(n/2) + n \qquad n > 1.$$

Replace n by 2^k (so that $k = \lg n$) to obtain $T(2^k) = 4T(2^{k-1}) + 2^k$. This can be written as

$$t_k = 4t_{k-1} + 2^k \qquad \text{if } t_k = T(2^k) = T(n).$$

We know how to solve this new recurrence: the characteristic equation is

$$(x-4)(x-2) = 0$$

and hence $t_k = c_1 4^k + c_2 2^k$.

Putting n back instead of k, we find

$$T(n) = c_1 n^2 + c_2 n.$$

T(n) is therefore in $O(n^2 \mid n \text{ is a power of } 2)$.

Recurrences (cont.)

Example: Solve the recurrence if n is a power of 2 and if

$T(n) = 2T(n/2) + n\lg n \qquad n > 1.$

Replace n by 2^k (so that $k = \lg n$) to obtain $T(2^k) = 2T(2^{k-1}) + k2^k$.

This can be written as $t_k = 2t_{k-1} + k2^k$ if $t_k = T(2^k) = T(n)$.

We know how to solve this new recurrence: the characteristic equation is

$(x-2)^3 = 0$, (since $t_k - 2t_{k-1} = k2^k$, where $b = 2$, $p(k) = k$, and $d = 1$),

and hence $t_k = c_1 2^k + c_2 k2^k + c_3 k^2 2^k$. Putting n back instead of k, we find

$T(n) = c_1 n + c_2 n\lg n + c_3 n\lg^2 n$.

Hence $T(n) \in O(n\lg^2 n \mid n \text{ is a power of } 2)$.

Example: Solve the recurrence if n is a power of 2 and if

$T(n) = 3T(n/2) + cn$ (c is constant, $n = 2^k > 1$.)

Replace n by 2^k (so that $k = \lg n$) to obtain $T(2^k) = 3T(2^{k-1}) + c2^k$.

This can be written as $t_k = 3t_{k-1} + c2^k$ if $t_k = T(2^k) = T(n)$.

The characteristic equation is $(x-3)(x-2) = 0$,

(since $t_k - 3t_{k-1} = c2^k$, where $b = 2$, $p(k) = c$, and $d = 0$), and hence

$t_k = c_1 3^k + c_2 2^k$ If we put n back, instead of k, we find

$T(n) = c_1 3^{\lg n} + c_2 n$.

Since $a^{\lg b} = b^{\lg a}$, therefore, $T(n) = c_1 n^{\lg 3} + c_2 n$. Hence, $T(n) \in O(n^{\lg 3})$.