Design and Analysis of Algorithm

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DAA Objectives & Outcomes

Course Objectives:

- Study the performance analysis of algorithms
- Select algorithmic strategies to solve given problem.
- Explore solution space to solve the problems.
- Provide the knowledge about complexity theory

Course Outcomes:

1. Analyze the algorithm complexity using asymptotic notations and describe the divide-and-conquer paradigm and recite algorithms that employ this paradigm.
2. Describe greedy and dynamic programming algorithmic strategies and analysis algorithms that employ this paradigm.
3. Illustrate the solution space using backtracking and branch and bound algorithmic techniques.
4. Describe the concept of complexity theory.

Course prerequisite

Data Structure II

C, C++, Java or other programming languages

MIT-WPU
॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥

REF BOOK: THOMAS CORMEN

Module I - Fundamentals of Algorithm

The Role of Algorithms in Computing Algorithmic specifications, Analyzing algorithm, asymptotic notations, order of growth

Algorithm design strategy, Divide and conquer - Merge Sort, Quick sort, Large Integer multiplication, solving recurrences

(substitution method & Master's theorem)

Why Study this Course?

REF BOOK: THOMAS CORMEN

Donald E. Knuth stated “Computer Science is the study of algorithms”

- Cornerstone of computer science. Programs will not exist without algorithms.

Algorithms are needed (most of which are novel) to solve the many problems listed here

- Computational Primitives –CG –Geometric Algorithms
- Communication Network –Shortest path Algorithms
- Genome Structure in Bioinformatics –Dynamic Programming
- Search engines –Page Rank Algorithm by Google
- Challenging (i.e. Good for brain !!!)

Real Blend of Creativity and Precision

Very interesting if you can concentrate on this course

Algorithm: A brief History

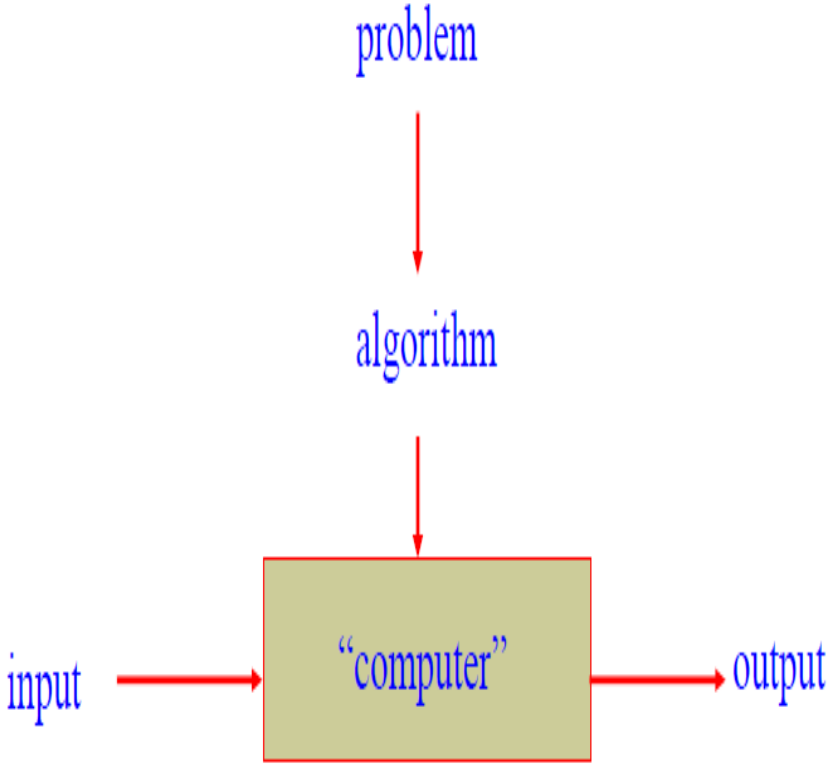
- The word algorithm comes from the name of **Persian author, Abu Ja'far Mohammed ibn Musa al Khowarizmi (c. 825 A.D.)**, who wrote a textbook on mathematics.
- The book was translated into Latin in the 12th century under the title **Algoritmi de numero Indorum.** This title means "Algoritmi on the numbers of the Indians", where "Algoritmi" was the translator's Latinization of Al-Khwarizmi's name.
- Many centuries later, decimal system was adopted in Europe, and the *procedures in Al Khwarizmi's book were named after him as "Algorithms".*

REF BOOK: THOMAS CORMEN

What is an Algorithm?

An algorithm is a sequence of unambiguous instructions for solving a computational problem i.e., for obtaining a required output for any legitimate input in a finite amount of time.

It is any well defined computational procedure that takes some value or set of values as **input** and produces some value, or set of values as **output**



Algorithm & its Properties

More precisely, An *algorithm* is a finite set of instructions that accomplishes a particular task.

Algorithm must satisfy the following properties:

- Input
  - Valid inputs must be clearly specified.
- Output
  - can be proved to produce the correct output given a valid input.
- Finiteness
  - terminates after a finite number of steps
- Definiteness
  - Each instruction must be clear and unambiguously specified
- Effectiveness
  - Every instruction must be sufficiently simple and basic.

Examples of Algorithms – Computing the Greatest Common Divisor of Two Integers

Problem: Find gcd (*m*,*n*), the greatest common divisor of two nonnegative, not both zero integers *m* and *n*

//First try –**School level algorithm:**

Step1: factorize m.(m m1*m2*m3…)

Step2: factorize n. (n n1*n2*…)

Step 3: Identify common factors , multiply and return.

Pseudocode of Euclid's Algorithm

Algorithm : ***Euclid (m, n)***

//Computes gcd(m, n) by Euclid's algorithm

//Input: Two non negative, not-both-zero integers m and n

//Output: Greatest common divisor of m and n **gcd(m, n) = gcd(n, m mod n)**

While (m does not divide n) **do**

r ← n mod m

n ← m

m ← r

return m

Questions:

Finiteness: how do we know that Euclid's algorithm actually comes to a stop?

Definiteness: non-ambiguity

Effectiveness: effectively computable.

Which algorithm is faster, the Euclid's or previous one?

Example of Euclid's Algorithm

Simple Algorithm

m = 36, n = 48

m = 2 * 2 * 3 * 3

n = 2 * 2 * 2 * 2 * 3

Common factors

2, 2, 3

GCD = 12

9 divisions

Euclid Algorithm

36 divides 48

r = 48 mod 36

n = 36

m = 12

Return 12

2 divisions

- Another example:
  m = 434 and n = 966

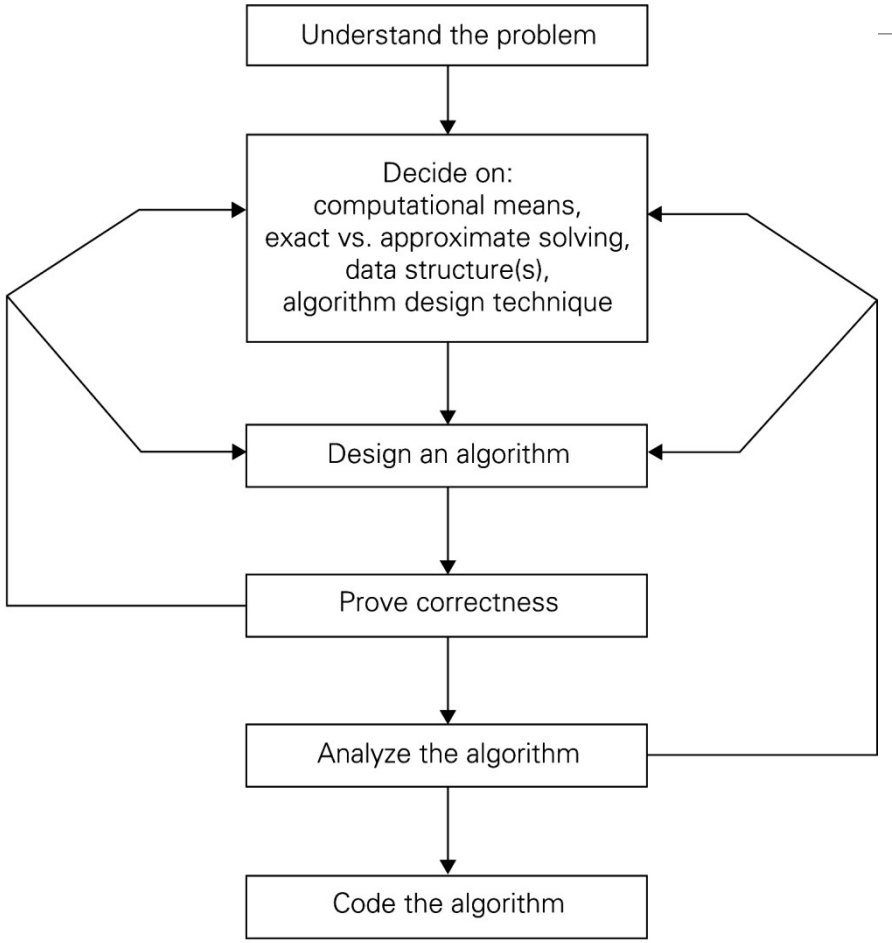
REF BOOK: THOMAS CORMEN

Fundamentals of Algorithmic Problem Solving

- Understanding the problem
  - Asking questions, do a few examples by hand, think about special cases, etc.
- Deciding on
  - Exact vs. approximate problem solving
- Appropriate data structure
- Design an algorithm
- Proving correctness
- Analyzing an algorithm
  - Time efficiency : how fast the algorithm runs
  - Space efficiency: how much extra memory the algorithm needs.
- Coding an algorithm

REF BOOK: THOMAS CORMEN

Algorithm Design and Analysis Process



Algorithms as a Technology

Even if computers were infinitely fast and memory was plentiful and free

- Study of algorithms still important – still need to establish algorithm correctness
- Since time and space resources are infinitely fast, any correct algorithm for solving a problem would do.

Real-world computers may be fast but not infinitely fast.

Memory is cheap but not free

Hence Computing time is bounded resource and so need space in memory.

We should use resources wisely by efficient algorithm in terms of space and time

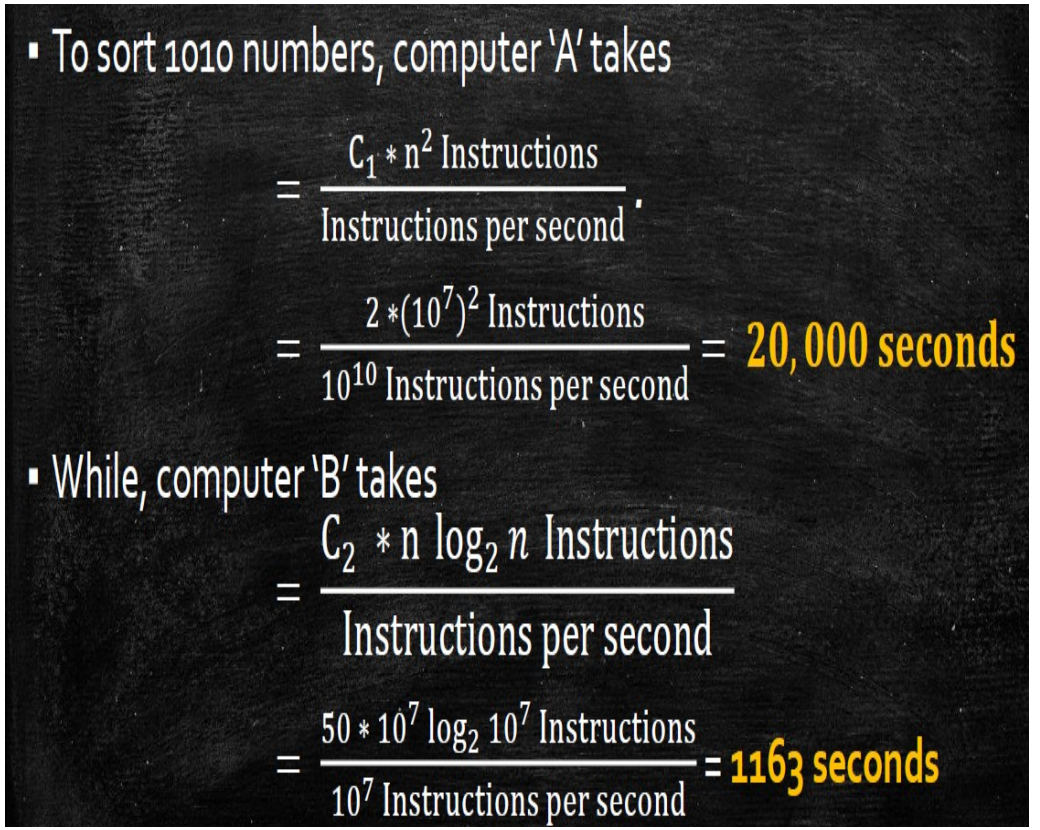
REF BOOK: THOMAS CORMEN

Algorithm Efficiency

Time and space efficiency are the goal

Algorithms often differ dramatically in their efficiency

- Example: Two sorting algorithms
  - **INSERTION-SORT** – time efficiency is $c_1 n^2$
  - **MERGE-SORT** – time efficiency is $c_1 n \log n$
- For which problem instances would one algorithm be preferable to the other?
- For example,
- A faster computer 'A' (10^{10} instructions/sec) running insertion sort against a slower computer 'B' (10^7 instructions/sec) running merge sort. Suppose that c1=2 , c2=50 and n= 10^7.



Efficiency

Problem Size	Machine A Insertion-Sort	Machine B Merge- Sort
n	$2n^2/10^9$	$50n\log n/10^7$
10,000	0.20	0.66
50,000	5.00	3.90
100,000	20.00	8.30
500,000	500.00	47.33
1,000,000	2,000.00	99.66
5,000,000	50,000.00	556.34
10,000,000	200,000.00	1,162.67
50,000,000	5,000,000.00	6,393.86

Methods of Specifying an Algorithm

- Natural language
  - Ambiguous
    - Example : “Mike ate the sandwich on a bed.”
- Flowchart
  - Graphic representations called flowcharts , only if the algorithm is small and simple.
- Pseudocode
  - A mixture of a natural language and programming language-like structures
  - Precise and concise.
- Pseudocode in this course
  - omits declarations of variables
  - use indentation to show the scope of such statements as for, if, and while.
  - use for assignment

Pseudocode Conventions

- Comments begin with // and continue until the end of line.
- Blocks are indicated with matching braces :{ and }. A compound statement (i.e., a collection of simple statements) can be represented as as a block. The body of a procedure also forms a block. Statements are delimited by;
- Assignment of values to variables is done using the assignment statement
- ⟨variable⟩:=⟨expression⟩;
- There are two Boolean values ***true*** and ***false***. In order to produce these values,
  - Logical operators : ***and***, ***or***, and ***not***
  - Relational operators<, ≤, =, ≠, ≥, and >
- Elements of multidimensional arrays are accessed using [and]. Ex A[i,j]

Pseudocode Conventions

- Looping statement can be employed as follows :

(while, for, repeat until)

While Loop:

```
While < condition > do
{
<statement-1>
.
.
.
<statement-n>
}
```

For Loop:

```
For variable: = value-1 to value-2 step
        step- value do
                {
                <statement-1>
                .
                .
                .
                <statement-n>
                }
```

```
For i:=1 to 10 step 2 do
{
}

i=1,3,5,7,9

For i:=1 to 10 do
{
}
i=1,2,3,....10.
```

A conditional statement has the following forms:

- **If** ⟨condition⟩ **then** ⟨statement⟩
- **If** ⟨condition⟩ **then** ⟨statement 1⟩ **else** ⟨statement 2⟩

We also employ the following case statement:

case

{

: ⟨condition 1⟩: ⟨statement 1⟩

.

: ⟨condition n⟩: ⟨statement n⟩

: **else**: ⟨statement n + 1⟩

}

Pseudocode Conventions

Input and output are done using the instructions ***read*** and ***write***. No format is used to specify the size of input or output quantities.

An algorithm consists of a heading and a body. The heading takes the form

Algorithm Name (⟨parameter list⟩)

{

// body

}

Example : Algorithm to find and return the maximum of n given numbers:

```
Algorithm Max (A, n)
// A is an array of size n.
{
Result :=A[1];
For i :=2 to n do
If A[i] > Result then Result :=A[i];
Return Result;
}
```

REF BOOK: THOMAS CORMEN

Analysis of Algorithms

Two main issues related to algorithms

- How to design algorithms
- How to analyze algorithm efficiency

Analysis of Algorithms

- How good is the algorithm?
  - time efficiency
  - space efficiency
- Does there exist a better algorithm?
  - lower bounds
  - optimality

Common Rates of Growth

Let n be the size of input to an algorithm, and k some constant. The following are common rates of growth.

- Constant: $\Theta(k)$, for example $\Theta(1)$
- Linear: $\Theta(n)$
- Logarithmic: $\Theta(\log_k n)$
- Linear : n $\Theta(n)$ or n log n: $\Theta(n \log_k n)$
- Quadratic: $\Theta(n^2)$
- Polynomial: $\Theta(n^k)$
- Exponential: $\Theta(k^n)$

REF BOOK: NPTEL

Why Analyse?

- Practical reasons:
  - Resources are scarce
  - Greed to do more with less
  - Avoid performance bugs
- Core Issues:
- Predict performance
  - How much time does binary search take?
- Compare algorithms
  - How quick is Quicksort?
- Provide guarantees
  - Size not with standing, Red-Black tree inserts in O(log n)
- Understand theoretical basis
  - Sorting by comparison cannot do better than (n log n)

What to analyse?

Core Issue: Cannot control what we cannot measure

Time

- The ***time complexity, T(n),*** taken by a program P is the sum of the running times for each statement executed

Space

The ***space complexity*** of a program is the amount of memory that it needs to run to completion

Examples : Sum of Natural Numbers

```
// Sum of Natural Numbers
Algorithm sum (a, n)
{
s := 0 ;
For i := 1 to n do
s := s + a[i];
return s;
}
```

Time T(n) = n (additions)

Space S(n) = 2 (n, s)

Tabular Method

Iterative function to sum a list of numbers (steps/execution)

Statement	s/e	Frequency	Total steps
Algorithm sum (a, n)	0	-	0
{	0	-	0
s := 0 ;	1	1	1
For i := 1 to n do	1	n+1	n+1
s := s + a[i];	1	n	n
return s;	1	1	1
}			
Total T(n)			2n+3

Machine Model: Random Access Machine (RAM) Computing Model

- Input data & size
- Operations
- Intermediate Stages
- Output data & size

REF BOOK: THOMAS CORMEN

Asymptotic Analysis

Core Idea: Cannot compare actual times; hence compare Growth or how time increases with input

- O notation (“Big Oh”)
- Ω notation (Omega)
- Θ notation (Theta)
- o notation (Little oh)
- ω notation (Little omega)

ASYMPTOTICS NOTATIONS
O-notation **(Big Oh)**

REF BOOK: THOMAS CORMEN

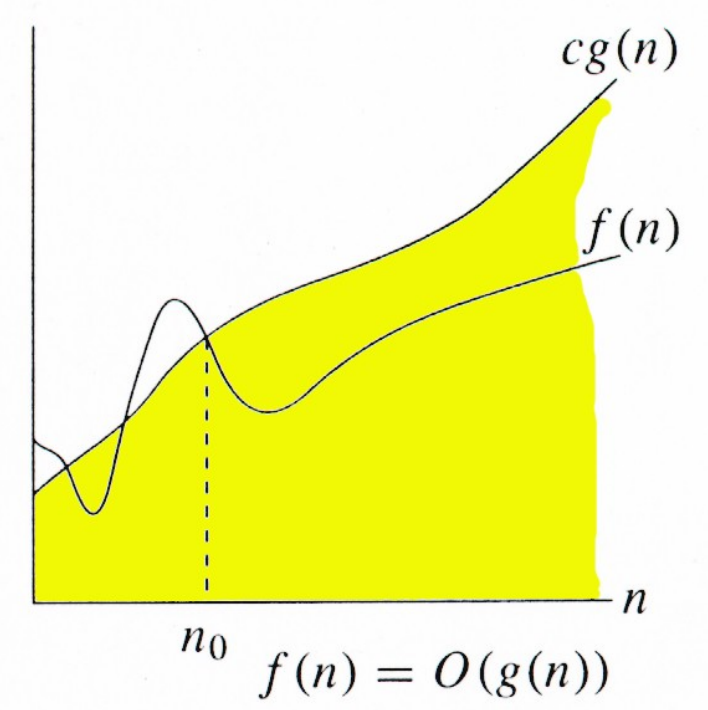
Asymptotic Upper Bound

For a given function g(n), we denote O(g(n)) as the set of functions:

$O(g(n)) = \{ f(n) |$ there exists positive constants c and n0 such that $0 \leq f(n) \leq c\, g(n)$ for all $n \geq n0 \}$

It is used to represent the worst case growth of an algorithm in time or a data structure in space when they are dependent on n, where n is big enough

Example :



Big Oh - Example

$f(n) = n^2 + 5n = O(n2)$

$g(n) = n^2$ ……. c = 2

n	$n^2 + 5n$	$2n^2$
1	5	2
2	14	8
5	50	50

$f(n) <= c\ g(n)$ for all $n> = n_0$ where $c=2$ & $n_0 =5$

Big Oh - Example

Let $f(N) = 2N^2$. Then

- $f(N) = O(N^4)$ (loose bound)
- $f(N) = O(N^3)$ (loose bound)
- $f(N) = O(N^2)$ (It is the best answer and the bound is asymptotically tight.)

$O(N^2)$: reads “order N-squared” or “Big-Oh N-squared”.

Big Oh - Example

Prove that if $T(n) = 15n^3 + n^2 + 4$, $T(n) = O(n^3)$.

Proof.

Let $c = 20$ and $n_0 = 1$.

Must show that $0 \leq f(n)$ and $f(n) \leq cg(n)$.

$0 \leq 15n^3 + n^2 + 4$ for all $n \geq n_0 = 1$.

$f(n) = 15n^3 + n^2 + 4 \leq 20n^3$ ($20g(n) = cg(n)$)

As per defination of Big O, hence $T(n) = \mathbf{O(n^3)}$

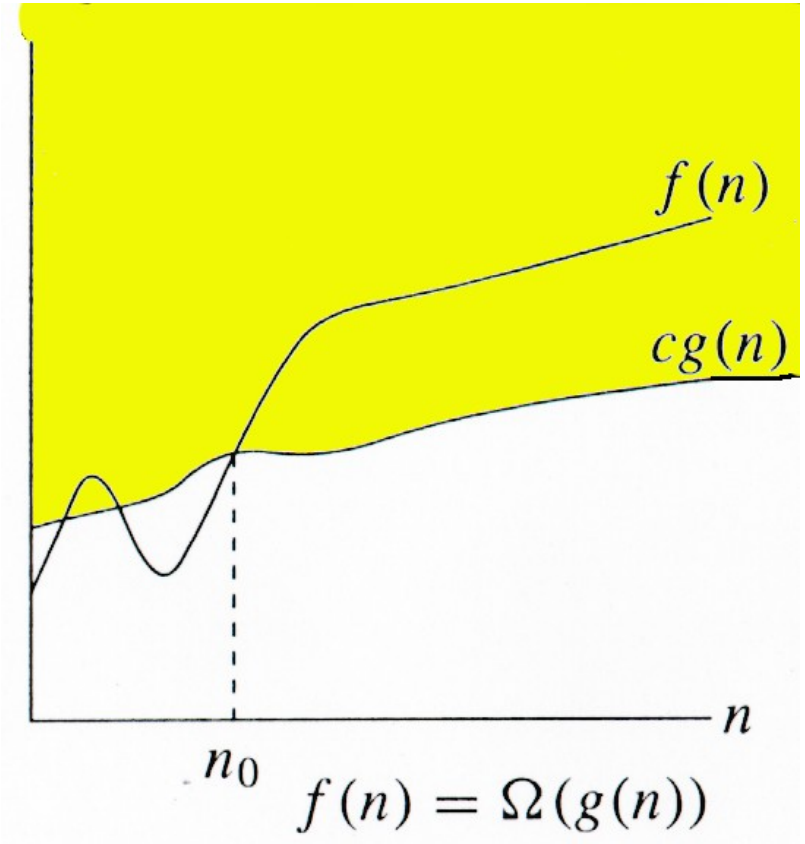
ASYMPTOTICS NOTATIONS
Ω-notation

REF BOOK: THOMAS CORMEN

Asymptotic lower bound

Ω (g(n)) represents a set of functions such that:

Ω(g(n)) = {f(n): there exist positive
constants c and n0 such that
$0 \leq c\ g(n) \leq f(n)$ for all $n \geq n0$}



Big Omega - Example

Example 1 :

$f(n) = n^2 + 5n$

$g(n) = n^2$ ..... $c = 1$

n	$n^2 + 5n$	$c * n^2$
1	5	1
2	14	4
5	50	25

$f(n) >= c\ g(n)$ for all $n> = n_0$ where $c=1$ & $n_0 =1$

Example 1 :

Prove that if $T(n) = 15n^3 + n^2 + 4$, $T(n) = \Omega (n^3)$.

Proof.

Let $c = 15$ and $n_0 = 1$.

Must show that $0 \leq cg(n)$ and $cg(n) \leq f(n)$.

$0 \leq 15n^3$ for all $n \geq n_0 = 1$.

$cg(n) = 15n^3 \leq 15n^3 + n^2 + 4 = f(n)$

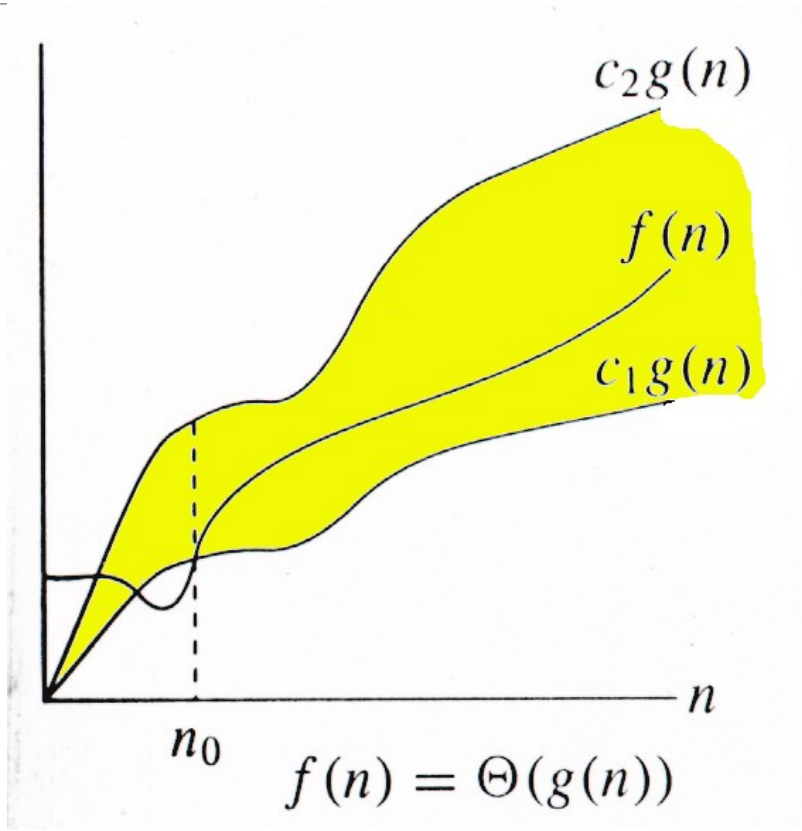
ASYMPTOTICS NOTATIONS
Θ-notation

REF BOOK: THOMAS CORMEN

Asymptotic tight bound

Θ (g(n)) represents a set of functions such that:

Θ (g(n)) = {f(n): there exist positive
constants c1, c2, and n0such
that $0 \leq c1g(n) \leq f(n) \leq c2g(n)$
for all $n \geq n0$ }



Theta Example

$f(N) = \Theta(g(N))$ iff $f(N) = O(g(N))$ and $f(N) = \Omega(g(N))$

It can be read as "f(N) has order exactly g(N)".

The growth rate of f(N) equals the growth rate of g(N). The growth rate of f(N) is the same as the growth rate of g(N) for large N.

Theta means the bound is the tightest possible.

If T(N) is a polynomial of degree k, $T(N) = \Theta(N^k)$.

For logarithmic functions, $T(\log_m N) = \Theta(\log N)$.

o notation (Little oh)

The function $f(n) = o(g(n))$ iff $\lim_{n->\infty} \frac{f(n)}{g(n)} = 0$

$\boldsymbol{o(g(n))} = \{f(n)$: $\forall$ **$c > 0$**, $\forall$ **$n_0 > 0$** such that
$\forall n \geq n_0$, we have $0 \leq f(n) < cg(n)\}$.

Example: The function $3n + 2 = o(n^2)$ as $\lim_{n->\infty} \frac{3n+2}{n^2} = 0$

ω notation (Little omega)

The function $f(n) = \omega(g(n))$ iff

$$\lim_{n->\infty} \frac{g(n)}{f(n)} = 0$$

$\omega(g(n))$ = {$f(n)$: $\forall$ $c > 0$, $\forall$ $n_0 > 0$ such that

$\forall$ $n \geq n_0$, we have $0 \leq cg(n) < f(n)$}.

REF BOOK: THOMAS CORMEN

Asymptotic Notations

It is a way to compare “sizes” of functions

O-notation ----------Less than equal to (“≤”)

Θ-notation ----------Equal to (“=“)

Ω-notation ----------Greater than equal to (“≥”)

$$O \approx \leq$$

$$\Omega \approx \geq$$

$$\Theta \approx =$$

$$o \approx <$$

$$\omega \approx >$$

REF BOOK: THOMAS CORMEN/SAHNI

Examples On Asymptotic Notations

Explain How is f(x) = 4n^2 – 5n + 3 is O(n^2)

Show that

1) $30n+8$ is $O(n)$
2) $100n + 5 \neq \Theta(n^2)$
3) $5n^2 = \Theta(n)$
4) $100n + 5 = O(n^2)$
5) $n^2/2 - n/2 = \Theta(n^2)$

Algorithm Design Techniques/Strategies

REF BOOK: THOMAS CORMEN

- Brute force
- Divide and conquer
- Space and time tradeoffs
- Greedy approach
- Dynamic programming
- Backtracking
- Branch and bound

Divide & Conquer

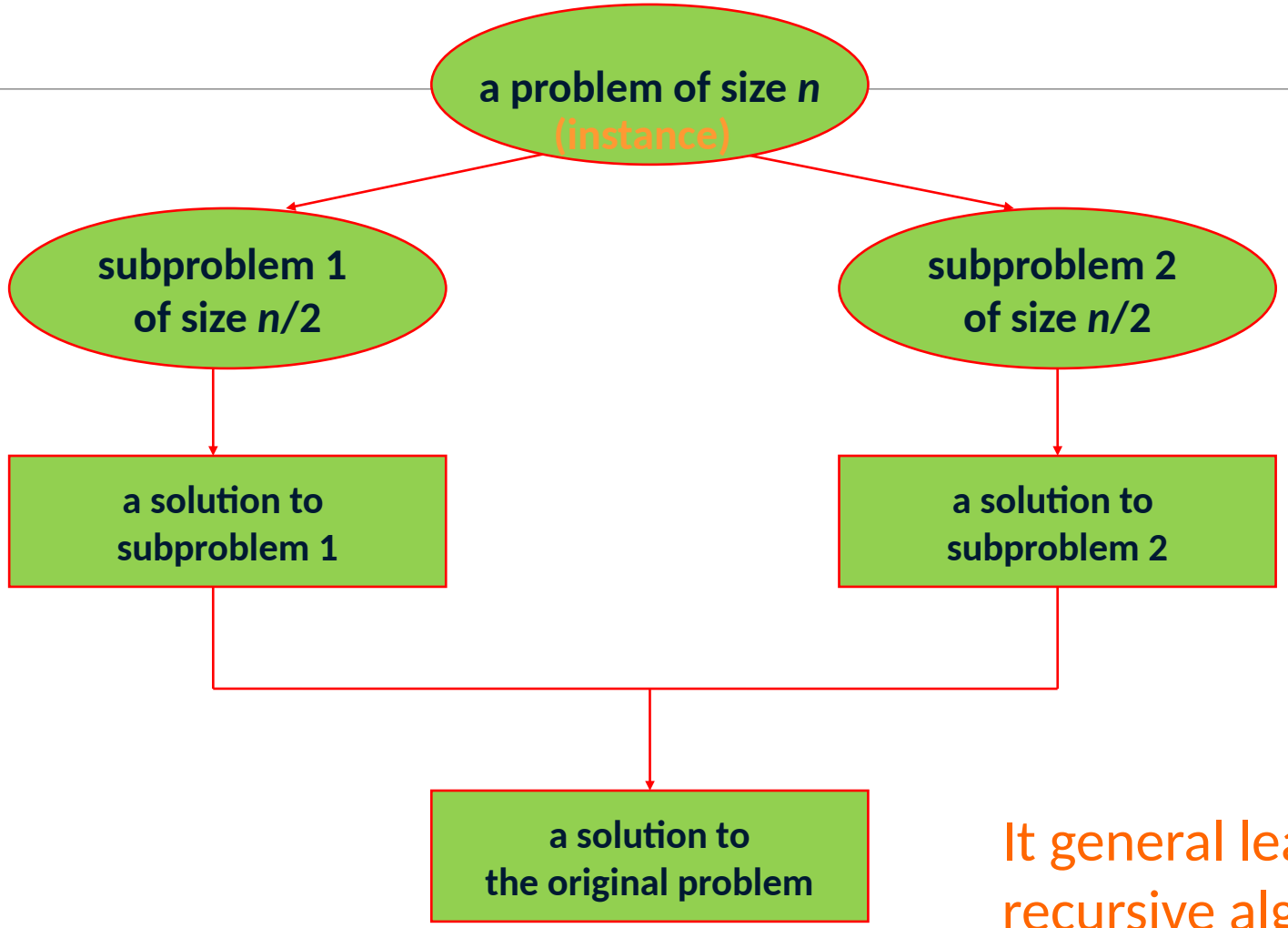
Control Abstraction

```
DANDC (P)
{
if SMALL (P) then return S (p);
else
{
divide p into smaller instances p1, p2, …. Pk, k >= 1;
apply DANDC to each of these sub problems;
return (COMBINE (DANDC (p1) , DANDC (p2),…., DANDC (pk));
}
}
```

Divide the problem into smaller sub problems
Conquer the sub problems by solving them recursively.
Combine the solutions to the sub problems into the solution of the original problem.

Divide-and-Conquer Technique (cont.)

REF BOOK: INTERNET



It general leads to a recursive algorithm!

Time complexity of the general algorithm

A Recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs

Special techniques are required to analyze the space and time required

$$T(n)=\begin{cases} aT(n/b)+D(n)+C(n) & , n \geq c \\ O(1) & , n < c \end{cases}$$

Time complexity (recurrence relation):

where D(n) : time for splitting

C(n) : time for conquer

c : a constant

REF BOOK: THOMAS CORMEN

Methods for Solving recurrences

- Substitution Method
  - We guess a bound and then use mathematical induction to prove our guess correct.
- Recursion Tree
  - Convert recurrence into tree
- Master Method

REF BOOK: INTERNET

Math You need to Review

properties of logarithms:

$\log_b(xy) = \log_b x + \log_b y$

$\log_b (x/y) = \log_b x - \log_b y$

$\log_b xa = a\log_b x$

$\log_b a = \log_x a/\log_x b$

- **properties of exponentials**:

$a^{(b+c)} = a^b a^c$

$a^{bc} = (a^b)^c$

$a^b / a^c = a^{(b-c)}$

$b = a^{\log_a b}$

$b^c = a^{c*\log_a b}$

REF BOOK: THOMAS CORMEN

Substitution Method

- Sum of n Natural Numbers

```
int sum(int n) {
    int s = 0;
    for(; n > 0; --n)
        s = s + n;
    return s;
}
```

$$\begin{aligned} T(n) &= T(n-1)+1, & n>1 \\ &= 1, & n=1 \end{aligned}$$

Solve Recurrence in Long hand:

$$\begin{aligned} T(n) &= T(n-1)+1 \\ &= T(n-2)+(1+1) \\ &= \cdots \\ &= T(1)+(n-1) \\ &= 1+(n-1) \\ &= n \end{aligned}$$

- Time $T(n)$ (additions)
- Space $S(n) = 2$

- Factorial (Recursive)

```
int fact(int n) {
    if (0 != n) return n*fact(n-1);
    return 1;
}
```

- Time $T(n)$ (multiplication)

$$\begin{aligned} T(n) &= T(n-1)+1, & n>0 \\ &= 0, & n=0 \end{aligned}$$

$$T(n) = n$$

Quick Sort

$$T(n) = 2T(n/2) + O(n)$$

$$= 2(2(n/2^2) + (n/2)) + n$$

$$= 2^2\, T(n/2^2) + n + n$$

$$= 2^2\, (T(n/2^3) + (n/2^2)) + n + n$$

$$= 2^3\, T(n/2^3) + \underline{n + n + n}$$

$$= \boldsymbol{n \log n}$$

Binary Search

EXAMPLE 2: BINARY SEARCH

$T(n)=O(1) + T(n/2)$

$T(1)=1$

Above is another example of recurrence relation and the way to solve it is by Substitution.

$T(n)=T(n/2) +1$

$= T(n/2^2)+1+1$

$= T(n/2^3)+1+1+1$

$= \log n$

T(n)= O(logn)

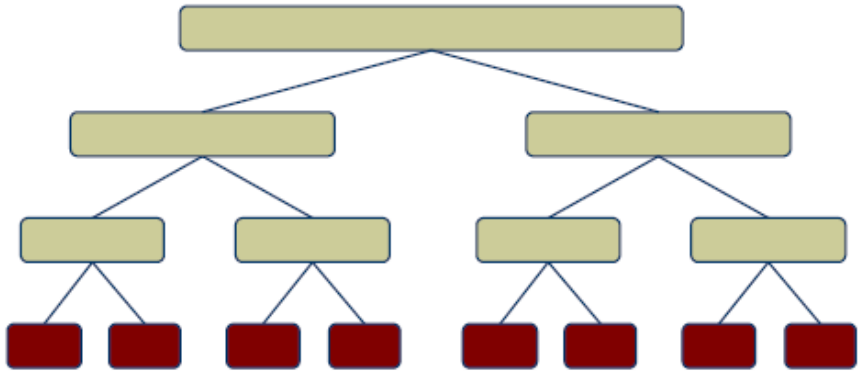
REF BOOK: THOMAS CORMEN

The Recursion Tree

Draw the recursion tree for the recurrence relation and look for a pattern:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

depth	T's	size	time
0	1	n	bn
1	2	$n/2$	bn
i	2^i	$n/2^i$	bn
...	...	...	...



Total time = $bn + bn \log n$

(last level plus all previous levels)

REF BOOK: THOMAS CORMEN

Master Theorem

Master method provides a "cookbook" method for solving recurrences of the following form

$$T(n) = a\,T(n/b) + f(n)$$

where a ≥ 1, b > 1. If f(n) is asymptotically positive function. T(n) has following asymptotic bounds:

There are 3 cases:

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 $f(n) = \Theta(n^{\log_b a} \log^k n)$ with[1] $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ with $\epsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$.
   Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

Example of Master Method

REF BOOK: THOMAS CORMEN

To use the master theorem, we simply plug the numbers into the formula

Example 1: $T(n) = 9T(n/3)+n$. Here $a = 9$, $b = 3$, $f(n) = n$, and $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = O(n^{\log_3 9-\epsilon})$ for $\epsilon = 1$, case 1 of the master theorem applies, and the solution is $T(n) = \Theta(n^2)$.

Example 2: $T(n) = T(2n/3)+1$. Here $a = 1$, $b = 3/2$, $f(n) = 1$, and $n^{\log_b a} = n^0 = 1$. Since $f(n) = \Theta(n^{\log_b a})$, case 2 of the master theorem applies, so the solution is $T(n) = \Theta(\log n)$.

Example 3: $T(n) = 3T(n/4) + n\log n$. Here $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. For $\epsilon = 0.2$, we have $f(n) = \Omega(n^{\log_4 3+\epsilon})$. So case 3 applies if we can show that $af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large n. This would mean $3\frac{n}{4}\log\frac{n}{4} \leq cn\log n$. Setting $c = 3/4$ would cause this condition to be satisfied.

Example 4: $T(n) = 2T(n/2)+n\log n$. Here the master method does not apply. $n^{\log_b a} = n$, and $f(n) = n\log n$. Case 3 does not apply because even though $n\log n$ is asymptotically larger than n, it is not polynomially larger. That is, the ratio $f(n)/n^{\log_b a} = \log n$ is asymptotically less than n^ϵ for all positive constants ϵ.

REF BOOK: SRIDHAR (OXFORD PUBLICATION)

Master Method (Simplified)

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

$$T(n) = a\,T(n/b) + f(n)$$

$$T(1) = c$$

where a ≥ 1, b ≥ 2, c>0. If f(n) is Θ(n^d) where d ≥ 0 then solution to recurrence relation is given as

Case 1: $T(n) \in \Theta(n^d)$ if $a < b^d$

Case 2: $T(n) \in \Theta(n^d \log n)$ if $a = b^d$

Case 3: $T(n) \in \Theta(n^{\log_b a})$ if $a > b^d$

REF BOOK: THOMAS CORMEN/SAHNI

Divide-and-Conquer Examples

Sorting : Mergesort and Quicksort

Binary tree traversals

Binary search

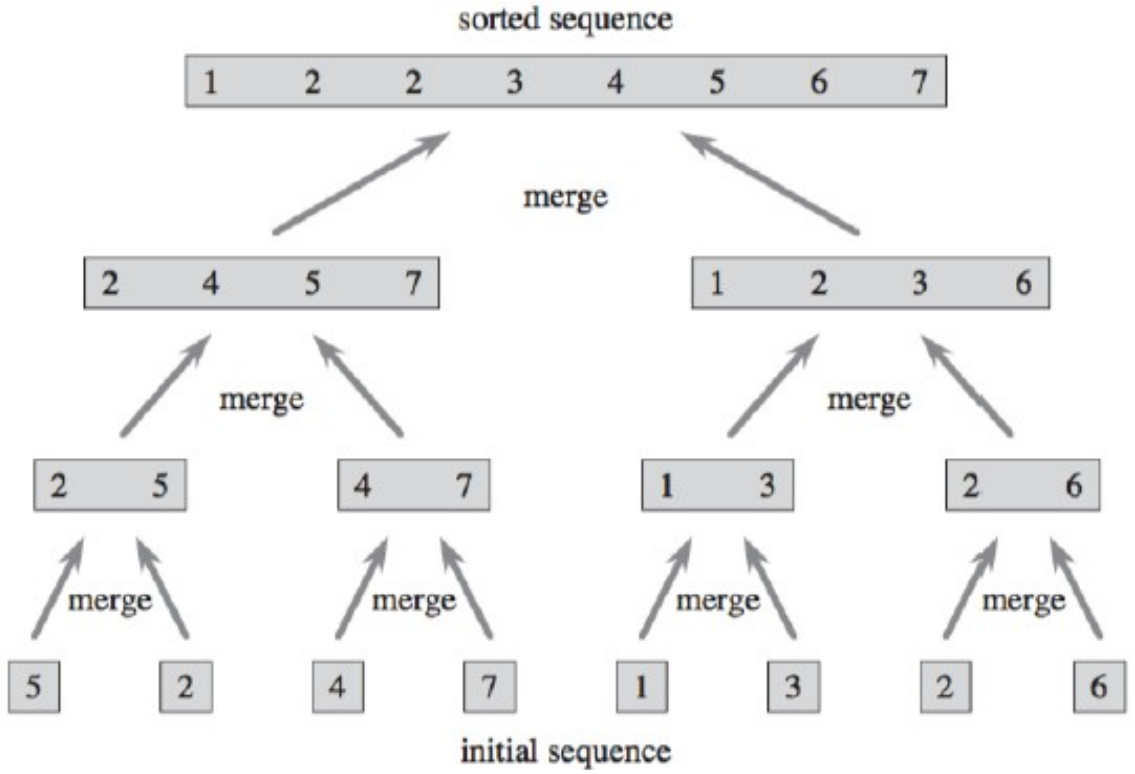
Multiplication of large integers

Matrix multiplication: Strassen’s algorithm

Closest-pair and convex-hull algorithms

Mergesort

Merge sort is a divide and conquer algorithm for sorting arrays. To sort an array, first you split it into two arrays of roughly equal size. Then sort each of those arrays using merge sort, and merge the two sorted arrays



REF BOOK: THOMAS CORMEN

Pseudocode of Merge-Sort (A, p, r)

INPUT: a sequence of *n* numbers stored in array A

OUTPUT: an ordered sequence of *n* numbers

```
MergeSort (A, p, r)   // sort A[p..r] by divide & conquer
1   if p < r
2       then q ← ⌊(p+r)/2⌋
3            MergeSort (A, p, q)
4            MergeSort (A, q+1, r)
5            Merge (A, p, q, r) // merges A[p..q] with A[q+1..r]
```

Initial Call: MergeSort(A, 1, *n*)

Procedure Merge

REF BOOK: THOMAS CORMEN

```
Merge(A, p, q, r)
1  n1 ← q − p + 1
2  n2 ← r − q
3      for i ← 1 to n1
4          do L[i] ← A[p + i − 1]
5      for j ← 1 to n2
6          do R[j] ← A[q + j]
7      L[n1+1] ← ∞
8      R[n2+1] ← ∞
9      i ← 1
10     j ← 1
11     for k ← p to r
12         do if L[i] ≤ R[j]
13            then A[k] ← L[i]
14                 i ← i + 1
15            else A[k] ← R[j]
16                 j ← j + 1
```

Input: Array containing sorted subarrays $A[p..q]$ and $A[q+1..r]$.

Output: Merged sorted subarray in $A[p..r]$.

Sentinels, to avoid having to check if either subarray is fully copied at each step.

Analysis of Mergesort

Running time $T(n)$ of Merge Sort:

Divide: computing the middle takes $\Theta(1)$

Conquer: solving 2 sub problems takes $2T(n/2)$

Combine: merging n elements takes $\Theta(n)$

Total:

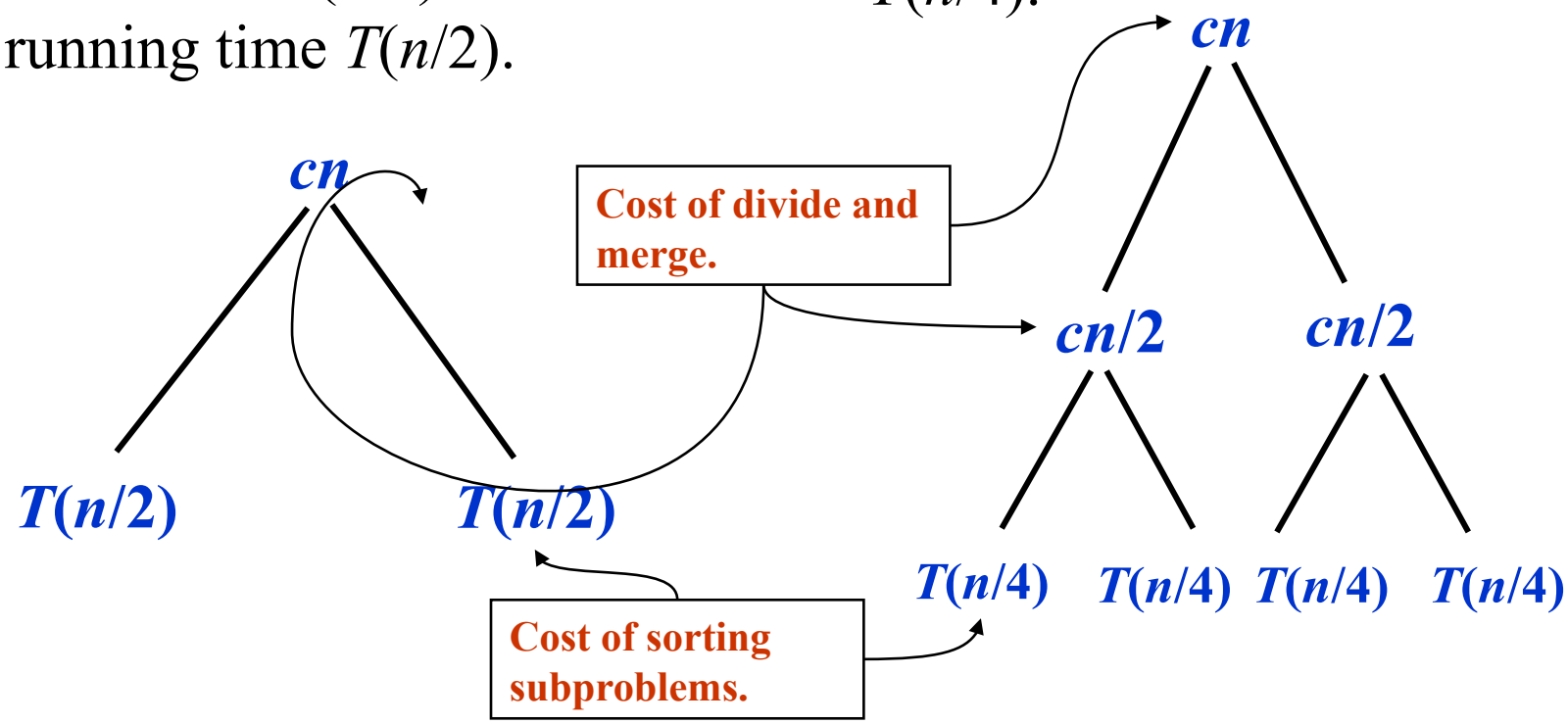
$$T(n) = \Theta(1) \quad \text{if } n = 1$$
$$T(n) = 2T(n/2) + \Theta(n) \quad \text{if } n > 1$$

- $T(n) = \Theta(n \lg n)$
- Space requirement: $\Theta(n)$ (not in-place)

Recursion Tree for Merge Sort

For the original problem, we have a cost of cn, plus two subproblems each of size $(n/2)$ and running time $T(n/2)$.

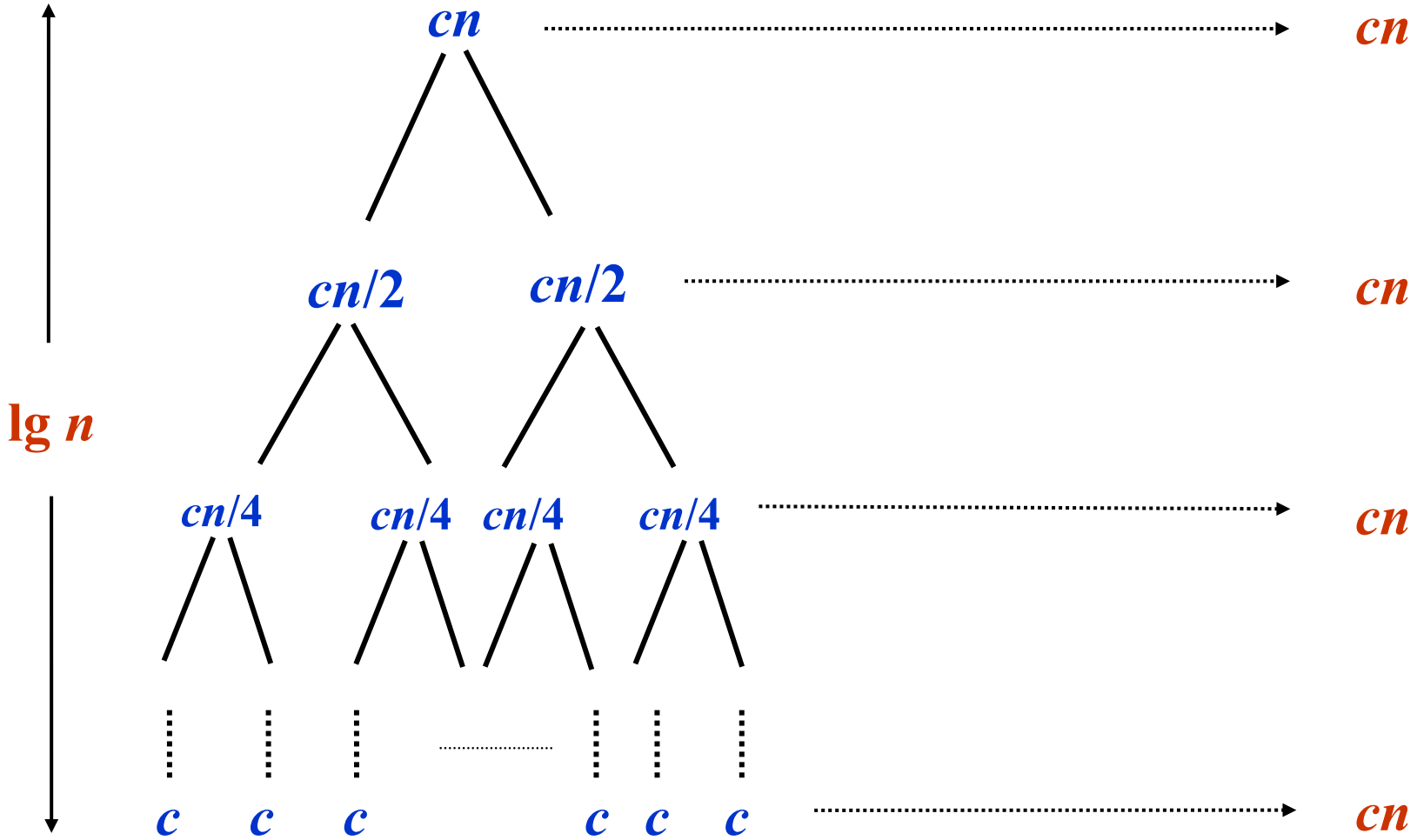
Each of the size $n/2$ problems has a cost of $cn/2$ plus two subproblems, each costing $T(n/4)$.



Recursion Tree for Merge Sort

Ref Book: Thomas Cormen

Continue expanding until the problem size reduces to 1.

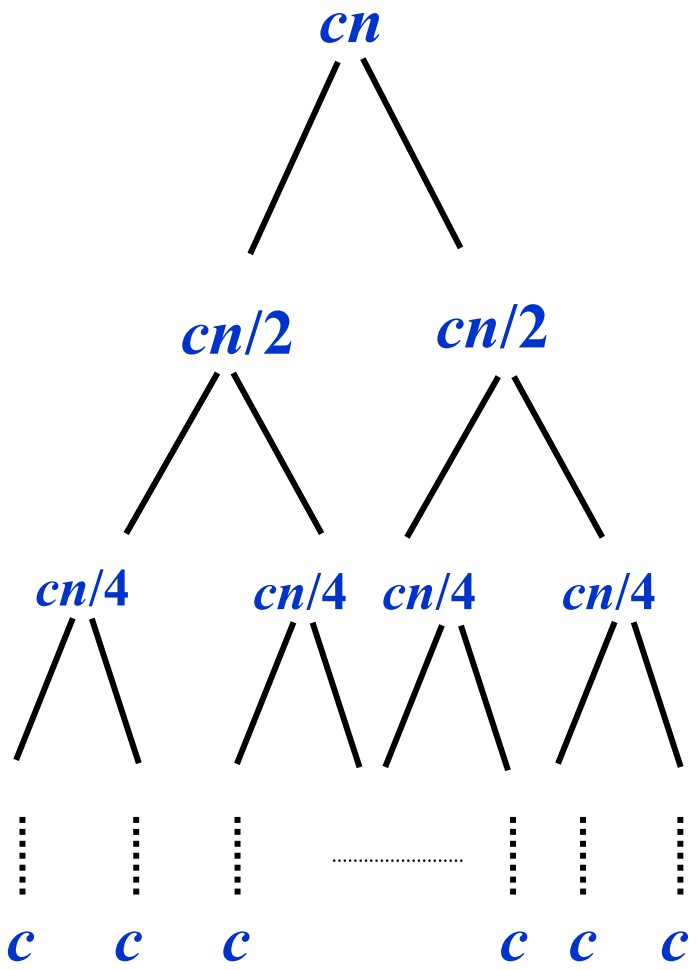


Total : $cn\lg n + cn$

Recursion Tree for Merge Sort

Ref Book: Thomas Cormen

Continue expanding until the problem size reduces to 1.



- Each level has total cost ***cn***.
- Each time we go down one level, the number of subproblems doubles, but the cost per subproblem halves *cost per level remains the same*.
- There are lg n + 1 levels, height is lg n. (Assuming n is a power of 2.)
  - Can be proved by induction.
- Total cost = sum of costs at each level = (lg n + 1)cn = cnlg$n$ + cn = (*n* lg*n*).

Ref Book: Thomas Cormen

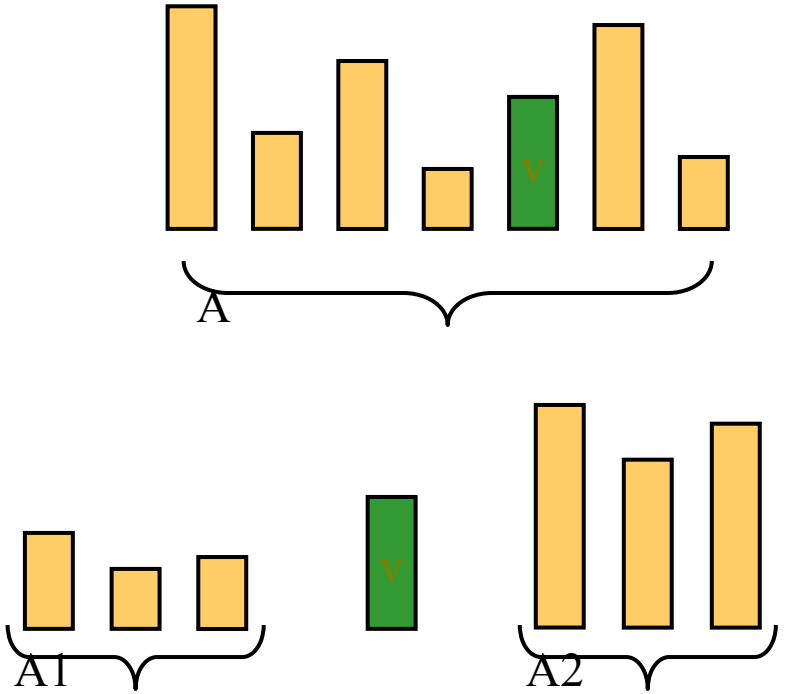
Quicksort

Ref Book: Thomas Cormen

- Divide :
  - Pick any element (***pivot***) v in array A[p…r]
  - Partition array A into two groups

    A1[p - - -q-1] , A2 [q+1----r] Compute the index q

  A1 element < A[q] < A2 element
- Conquer step: recursively sort A1 and A2
- Combine step: the sorted A1 (by the time returned from recursion), followed by A[q], followed by the sorted A2 (i.e., nothing extra needs to be done)



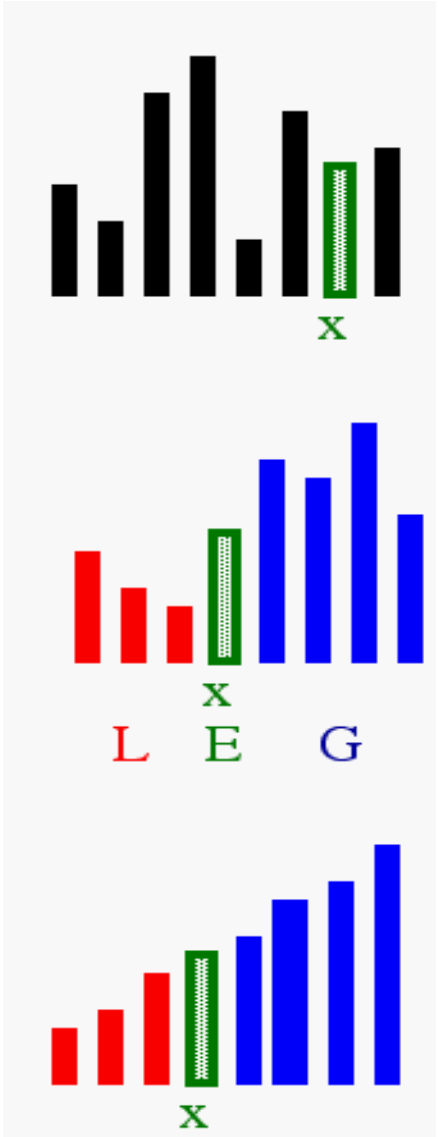
Idea of Quick Sort

Ref Book: Internet

1) **Select**: pick an element

2) **Divide**: rearrange elements so that x goes to its final position E

3) **Recurse and Conquer**: recursively sort



Ref Book: Thomas Cormen

Quicksort Pseudo-code

INPUT: a sequence of *n* numbers stored in array A

OUTPUT: an ordered sequence of *n* numbers

```
QuickSort (A, p, r)   // sort A[p..r] by divide & conquer
1    if p < r
2        then q = Partition (A, p, r)
3             QuickSort (A, p, q-1)
4             QuickSort (A, q+1, r)
```

Initial Call: QuickSort(*A*, 1, *n*)

Procedure Partitioning the array

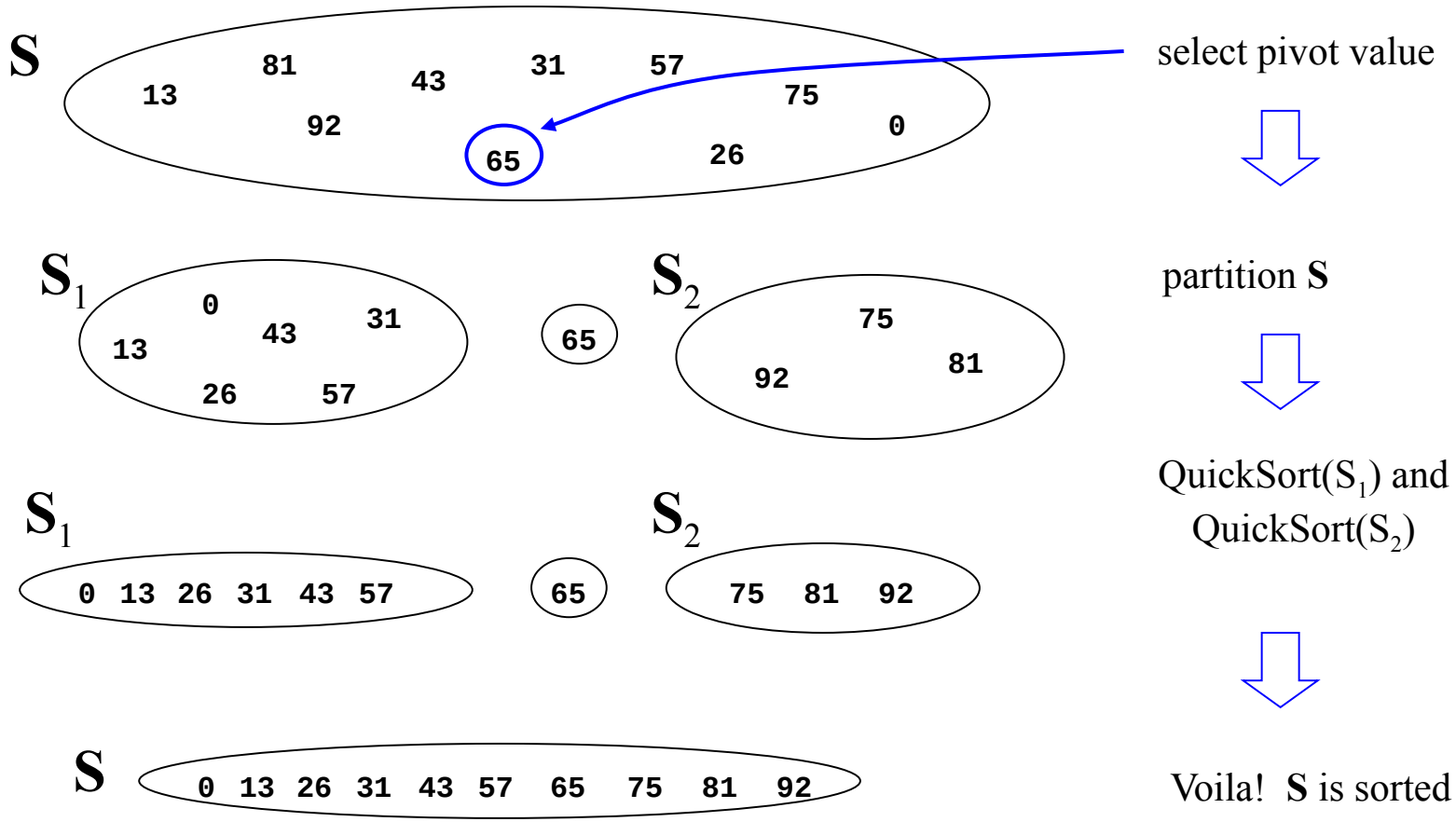
Ref Book: Thomas Cormen

```
Partition(A, p, r)
1  x = A[r]
2  i = p - 1
3      for j  p to r-1
4          if A[j]  x
5              i= i +1
6          exchange A[i] with A[j]
7      exchange A[i] with A[j]
8      Return i + 1
```

Input: Array containing sorted subarrays $A[p..q]$ and $A[q+1..r]$.

Output: sorted subarray in $A[p..r]$.

The steps of QuickSort



Quicksort Analysis

Assumptions:

- A random pivot (no median-of-three partitioning)
- No cutoff for small arrays

Running time

- pivot selection: constant time, i.e. O(1)
- partitioning: linear time, i.e. O(N)
- running time of the two recursive calls

T(N)=T(i)+T(N-i-1)+cN where c is a constant

- i: number of elements in S1

Worst-Case Analysis

worst case Partition?

- The pivot is the smallest element, all the time
- Partition is always unbalanced

$$
\begin{aligned}
T(N) &= T(N-1) + cN \\
T(N-1) &= T(N-2) + c(N-1) \\
T(N-2) &= T(N-3) + c(N-2) \\
&\vdots \\
T(2) &= T(1) + c(2) \\
T(N) &= T(1) + c\sum_{i=2}^{N} i = O(N^2)
\end{aligned}
$$

Best-case Analysis

best case Partitioning?

- Partition is perfectly balanced.
- Pivot is always in the middle (median of the array)

$$
\begin{aligned}
T(N) &= 2T(N/2) + cN \\
\frac{T(N)}{N} &= \frac{T(N/2)}{N/2} + c \\
\frac{T(N/2)}{N/2} &= \frac{T(N/4)}{N/4} + c \\
\frac{T(N/4)}{N/4} &= \frac{T(N/8)}{N/8} + c \\
&\vdots \\
\frac{T(2)}{2} &= \frac{T(1)}{1} + c \\
\frac{T(N)}{N} &= \frac{T(1)}{1} + c \log N \\
T(N) &= cN \log N + N = O(N \log N)
\end{aligned}
$$

Average-Case Analysis

Intution for Average Case

On average, the running time is O(N log N)

Summary Analysis of Quicksort

Best case: split in the middle — $\Theta(n \log n)$

Worst case: sorted array! — $\Theta(n^2)$

Average case: random arrays — $\Theta(n \log n)$

Improvements:

- better pivot selection: median of three partitioning
- switch to insertion sort on small subfiles
- elimination of recursion

REF BOOK: THOMAS CORMEN

Large Integer multiplication

References

1. Thomas H Cormen and Charles E.L Leiserson, "Introduction to Algorithm" PHI Third Edition
2. Horowitz and Sahani, "Fundamentals of Computer Algorithms", 2ND Edition. University Press, ISBN: 978 81 7371 6126, 81 7371 61262.