Lecture 1

(Analysis of Algorithm)

Today's Agenda

- Objective of the course & Course outline.
- Origin of word: *Algorithm*
- Teaching Procedure, Material/Resources, Grading.
- Algorithm?
- Data Structure? Algorithmics?
- How do we Analyze?
- Aim of Analysis of Algorithm.
- Hard Problems.
- Examples of some Multiplication Algorithms

Objective of the course

- Understanding the foundations of algorithms and use of Data Structures in the development of application-oriented algorithms.
- Inculcate skills to understand mathematical notations in algorithms and their simple mathematical proofs.
- Develop expertise needed for analyzing the algorithms.
- Gain familiarity with a number of classical problems that occur frequently in real-world applications.

Origin of word: *Algorithm*

- The word *Algorithm* comes from the name of the muslim author *Abu Ja'far Mohammad ibn Musa-al-Khowarizmi*. He was born in the eighth century at Khwarizm (Kheva), a town south of river Oxus in present Uzbekistan. Uzbekistan, a Muslim country for over a thousand years, was taken over by the Russians in 1873.
- Much of al-Khwarizmi's work was written in a book titled *al Kitab al-mukhatasar fi hisab al-jabrwa'l-muqabalah* (The Compendious Book on Calculation by Completion and Balancing). It is from the titles of these writings and his name that the words *algebra* and *algorithm* are derived. As a result of his work, al-Khwarizmi is regarded as the most outstanding mathematician of his time

Teaching Procedure

- Lectures
- Discussions
- Assignments (Important)
- Sudden Quizzes
- Mid Term
- Final Exam

Material / Resources

- Text Book
  - "Introductions to Algorithms", 2nd Edition
    by
    Thomas H. Cormen
    Charles E. Leiserson
    Ronald L. Rivest
    Clifford Stein

- For other books, view course outline
- WWW
- Any other good book on Algorithm Analysis.

Grading

- Assignments..... 05 %
- Quizzes/class participation..... 10 %
- Project + Presentation ..... 05 %
- Mid Exam ..... 30 %
- Final Exam ..... 50 %

What is Algorithm?

- Informally, an algorithm is any well-defined computational procedure that takes some value or set of values as input and produce some value or set of values as output.

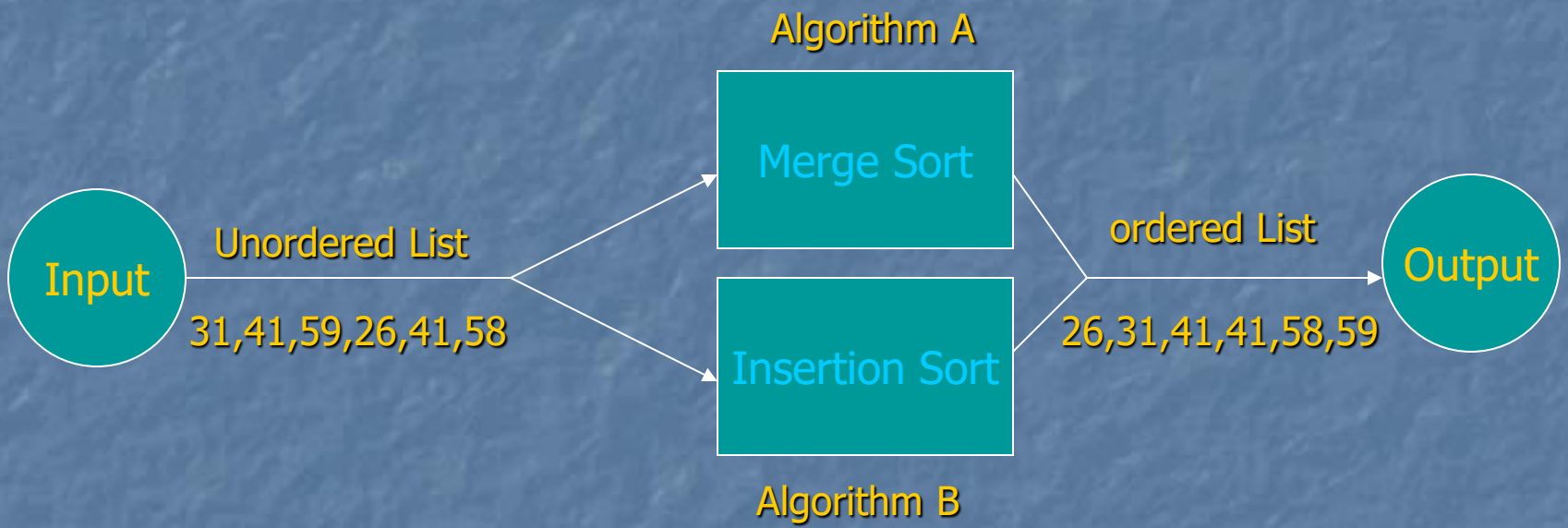
or

- We can also view an algorithm as a tool for solving a well specified computational problem.

or

- We can say algorithm is a sequence of operations to solve problems correctly.

Example: *Algorithms* to sort numbers in ascending order



In above figure the whole set of input numbers are known as instance of the sorting problem.

Data Structure

- A Data Structure is a systematic way of organizing and accessing data with a specific relationship between the elements, in order to facilitate access and modifications.

- No single data structure works well for all purposes, so it is important to be familiar with pros and cons of several Data Structures.

Algorithmics

- It is the science that lets designers study and evaluate the effect of algorithms based on various factors so that the best algorithm is selected to meet a particular task in given circumstances.

- It is also the science that tells how to design a new algorithm for a particular job.

How do we Analyze?

- Every Algorithm has a parameter N or n that effects its running time.
- For example, for sorting different numbers the parameter N is the number of input numbers to be sorted.

- So for analyzing algorithms our starting point is to have n or N

N or n ⟶ shows size of the Input.

Aim of Analysis of Algorithm

- Primary Concern:
  - Time (i.e. less number of time taken by Algo)
  - Space (i.e. less memory space to be taken)

- Secondary issues:
  - Size of instances to be handled
  - Type of language to be used for programming
  - Type of machine for implementation

Hard Problems

- Most of the contents of this course are about to address/discuss algorithms and their efficiency. Our usual measure of efficiency is speed.
- There are some problems, however, for which no efficient solution is known.
- We will study few of these kind of problems later in the course, which are known as NP-Complete problems.

PARAMETERS FOR SELECTION OF AN ALGORITHM

- Priority of Task
- Type of Available Computing Equipment
- Nature of Problem
- Speed of Execution
- Storage Requirement
- Programming Effort

A good choice can save both money and time, and can successfully solve the problem.

MULTIPLICATION
(981 x 1234)

981	981
1234	1234
3924	981
2943	1962
1962	2943
981	3924
1210554	1210554
American	English

MULTIPLICATION (981 x 1234)

(*a la russe*)

981	1234	1234
490	2468	
245	4936	4936
122	9872	
61	19744	19744
30	39488	
15	78976	78976
7	157952	157952
3	315904	315904
1	631808	631808
		1210554

MULTIPLICATION (981 x 1234)
(*DIVIDE & CONQUER*)

	Multiply		Shift	Result
i)	09	12	4	108 . . . .
ii)	09	34	2	306 . .
iii)	81	12	2	972 . .
iv)	81	34	0	2754
				1210554

MULTIPLICATION (9 x 12)
(*DIVIDE & CONQUER*)

	Multiply		Shift	Result
i)	0	1	2	0 . .
ii)	0	2	1	0 .
iii)	9	1	1	9 .
iv)	9	2	0	18
				108

Assignment No 1

- Implement Multiplication algorithm using *Divide and Conquer* approach to multiply any two integer numbers.
- Use any language or visual language (tool) of your choice. Due Coming Monday
- Copying assignment is strictly prohibited. If found, will lead to cancellation of assignment.

Thank You ...

Lecture 2

Growth of Functions - Asymptotic Notations

Growth of Functions

- We can sometimes determine the exact running time of the algorithm, however, the *extra precision is not usually* worth the effort of computing it.

- For large inputs, the *multiplicative constants* and *lower order terms* of an exact running time are dominated by the effects of the *input size itself*.

Things to consider when Analyzing Algorithms

1. Ignore constants. For example,
   $f(n) = 25n^2$
   or
   $f(n) = 25n^2 + 2000$
   $f(n) = O(n^2)$

2. Ignore small terms
   $f(n) = 25n^3 + 30n^2 + 10n$
   $f(n) = O(n^3)$

3. Application Independent
   i.e. not dependent on any tool, platform or application.

Asymptotic Performance

- In this course, we care most about *asymptotic performance. i.e.*
  - ***How does the algorithm behave as the problem size gets very large?***
    - Running time
    - Memory/storage requirements
    - Bandwidth/power requirements/logic gates/etc.

- When we look at the *input sizes large enough* to make only the order of growth of the running time relevant, we are studying the *asymptotic efficiency* of the algorithm.
- That is we are concerned with how the running time of the algorithm increases with the size of the input *in the limit*, as the size of the input increases without any bound.
- We will study standard methods for simplifying the asymptotic analysis of the algorithm. We will begin with by defining several types of "asymptotic notations" like Theta (Θ) Notation, Big-Oh (*O*) Notation and Omega (Ω) Notation.

Theta Notation (Θ)

- For a given function g(n), we denote by Θ(g(n)) the set of functions

Θ(g(n)) = { f(n) : there exist positive constants c1,c2 and n0 such that

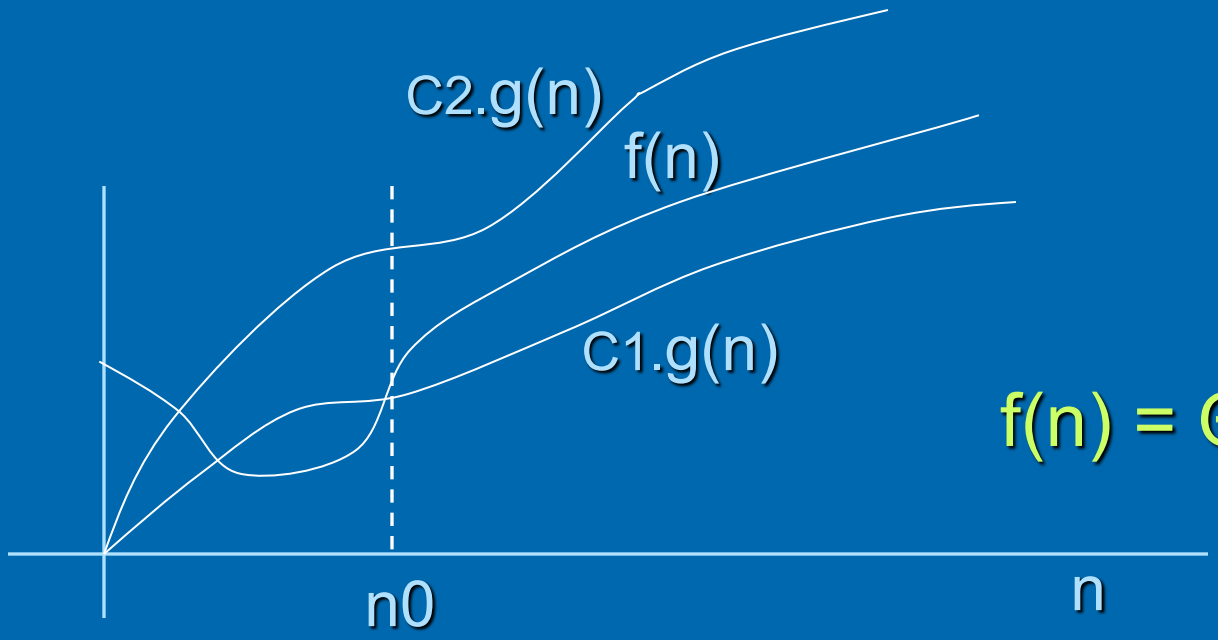
$0 \leq c1.g(n) \leq f(n) \leq c2.g(n)$ for all $n \geq n0$ }[1]

[1] colon in set means “such that”

- A function f(n) belongs to the set Θ(g(n)) if there exist positive constants c1 and c2 such that it can be *sandwiched* between c1g(n) and c2g(n) for sufficiently large n.

- Since Θ(g(n)) is a set, we could write f(n) Є Θ(g(n)) to indicate that f(n) is a member of Θ(g(n)). Instead we will usually write f(n) = Θ(g(n)).

Є means “belongs to”



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

- In the above figure, for all values of n to the right of n0, the value of f(n) lies at or above C1.g(n) and at or below C2.g(n).
- In other words, for all n ≥ n0, the function f(n) is equal to g(n) to *within constant factor*.
- We say that g(n) is ***asymptotic tight bound*** for f(n).

- Lets us justify this intuition by using the formal definition to show that $f(n) = n^2/2-3n = \Theta(n^2)$.
- To do so we must find the constants c1,c2, and n0 such that

$$c1.n^2 \leq n^2/2-3n \leq c2.n^2$$

for all $n \geq n0$. dividing by n^2

$$c1 \leq 1/2-3/n \leq c2$$

The right hand side inequality can be made to hold for any value of $n \geq 1$ by choosing $c2 \geq ½$.

Likewise , the left hand side inequality can be made to hold for any value of $n \geq 7$ by choosing $c1 \leq 1/14$. ***Thus***, by choosing c1=1/14 , c2=1/2 and n0 = 7, we can verify that $n^2/2-3n = \Theta(n^2)$.

- In above example, certainly other choices for the constants exist, but the important thing is that some choice exists.

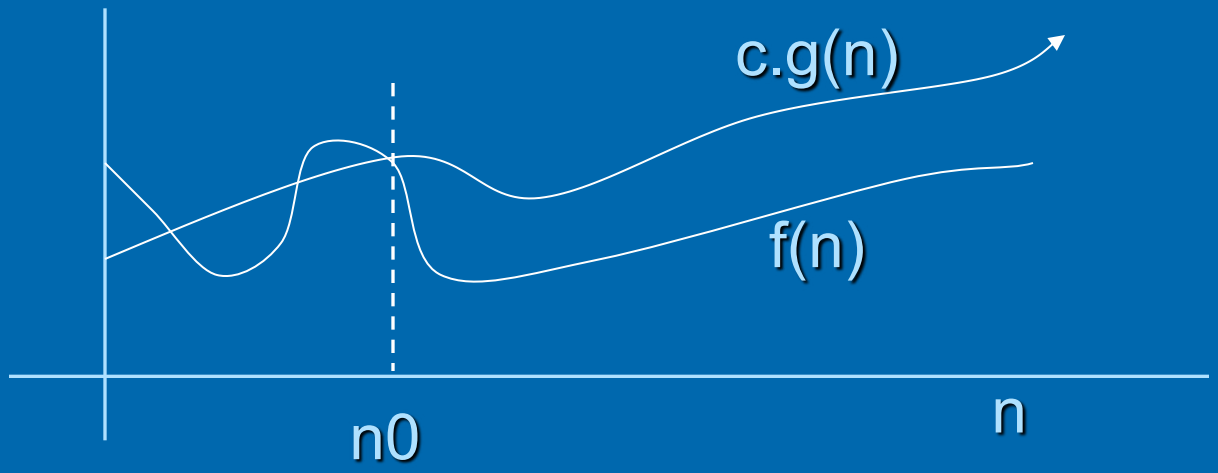
- Since any constant is a degree-0 polynomial, we can express any constant function as $\Theta(n^0)$ or $\Theta(1)$. We shall always use the notation $\Theta(1)$ to mean either constant or constant function with respective to some variable.

Big-Oh Notation (*O*)

- When we have only an ***asymptotic upper bound***, we use *O* – notation. For a given function g(n), we denote by *O*(g(n)) the set of functions

O(g(n)) = { f(n) : there exist positive constants c, and n0 such that

$0 \leq f(n) \leq c.g(n)$ for all $n \geq n0$ }[1]



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

- We write $f(n) = O(g(n))$ to indicate that $f(n)$ is a member of the set $O(g(n))$.

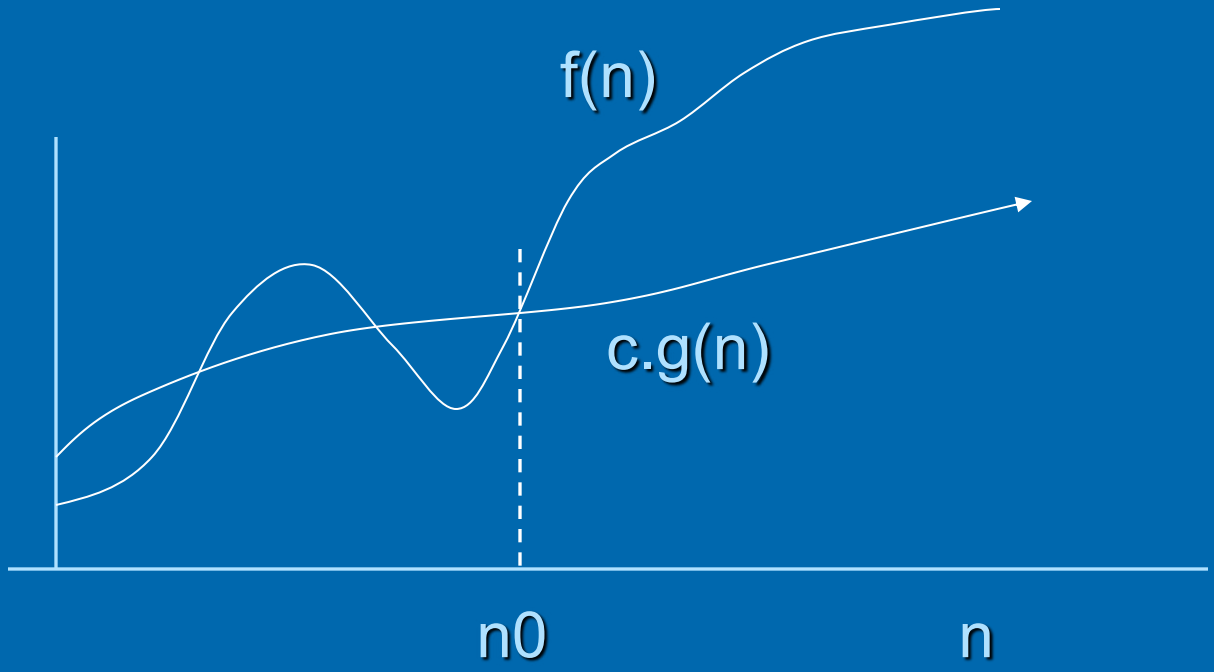
- Note that $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$, since Θ notation is stronger notation than O notation. So for quadratic function $an^2+bn + c$, where $a>0$, is in $\Theta(n^2)$ also shows that any quadratic function is in $O(n^2)$.

Omega Notation (Ω)

- Just as *O*-notation provides an upper bound on a function, Ω notation provides an ***asymptotic lower bound***. For a given function g(n), we denote by Ω(g(n)) the set of functions

Ω(g(n)) = { f(n) : there exist positive constants c, and n0 such that

$0 \leq c.g(n) \leq f(n)$ for all $n \geq n0$ }[1]



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

Example:

- Objective is to find the maximum of an unordered set having N elements.
  - Input : An unordered list.
  - Output : Maximum of input set.
- Algorithm

```
1. int Max = 0 //assume S[N] is filled with +ve numbers
2. For (int I = 0;I<N;I++)
3.  {
4.      If (S[I] > Max)
5.          Max = S[I]
6.  }
7. cout<< Max
```

Simple Analysis of previous Algorithm

Instruction	No of times Executed	cost
1	1	1
2	N	N
3	-	-
4	N	N
5	N	N
6	-	-
7	1	1

So f(n) = 1 + N + N + N + 1 = 2 + 3N = 3N+2

so f(n) = *O* (N) or *O* (n)

NOTE : we always look for worst case of each instruction to be executed while finding Big-Oh (*O*). (Assume all instructions take *unit time in above example)*.

- The worst case running time of an algorithm is an upper bound on the running time for an input.
- Knowing it, gives us guarantee that algorithm will never take any longer.
- We need not make some educated guess about the running time and hope that it never gets much worse.

Types of Functions (Bounding Functions)

1. Constant Function: $O(1)$
   For example, addition of two numbers will take same for Worst case, Best case and Average case.
2. Logarithmic Function: $O(\log(n))$
3. Linear Function : $O(n)$
4. $O(n.\log(n))$
5. Quadratic Function : $O(n^2)$
6. Cubic Function : $O(n^3)$
7. Polynomial Function : $O(n^k)$
8. Exponential Function : $O(2^n)$

   etc…

Some facts about summation

- If c is a constant

$$\sum_{i=1}^{n} c\,a_i = c\sum_{i=1}^{n} a_i$$

And

$$\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$$

Mathematical Series

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

$$\sum_{i=1}^{n} i^2 = 1 + 4 + 9 + \ldots + n^2 = \frac{2n^3 + 3n^2 + n}{6} = \Theta(n^3)$$

$$\sum_{i=0}^{n} x^i = 1 + x + x^2 + \ldots + x^n = \frac{x^{(n+1)} - 1}{x - 1}$$

- If $0 < x < 1$ then this is $\Theta(1)$, and if $x > 1$, then this is $\Theta(x^n)$.

- **Harmonic series For n ≥ 0**

$$
\begin{aligned}
H_n &= \sum_{i=1}^{n} \frac{1}{i} \\
&= 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \approx \ln n \\
&= \Theta(\ln n)
\end{aligned}
$$

Analysis: A Harder Example

- Let us consider a harder example.

```
NESTED-LOOPS()
1   for i ← 1 to n
2   do
3       for j ← 1 to 2i
4       do k = j ...
5           while (k ≥ 0)
6           do k = k − 1 ...
```

- How do we analyze the running time of an algorithm that has complex nested loop? The answer is we write out the loops as summations and then solve the summations. To convert loops into summations, we work from inside-out.

- Consider the *inner most while* loop.

```
NESTED-LOOPS()
1   for i ← 1 to n
2   do for j ← 1 to 2i
3       do k = j
4           while (k ≥ 0)
5           do k = k − 1
```

- It is executed for k = j, j - 1, j - 2, . . . , 0. Time spent inside the while loop is constant. Let I() be the time spent in the while loop. Thus

$$I(j) = \sum_{k=0}^{j} 1 = j + 1$$

- Consider the *middle for* loop.

```
NESTED-LOOPS()
1  for i ← 1 to n
2  do for j ← 1 to 2i
3      do k = j
4          while (k ≥ 0)
5          do k = k − 1
```

- Its running time is determined by i. Let M() be the time spent in the for loop:

$$
\begin{aligned}
M(i) &= \sum_{j=1}^{2i} I(j) \\
&= \sum_{j=1}^{2i} (j+1) \\
&= \sum_{j=1}^{2i} j + \sum_{j=1}^{2i} 1 \\
&= \frac{2i(2i+1)}{2} + 2i \\
&= 2i^2 + 3i
\end{aligned}
$$

- Finally the *outer-most for* loop.

```
NESTED-LOOPS()
1  for i ← 1 to n
2  do for j ← 1 to 2i
3       do k = j
4            while (k ≥ 0)
5            do k = k − 1
```

- Let T() be running time of the entire algorithm

$$
\begin{aligned}
T(n) &= \sum_{i=1}^{n} M(i) \\
&= \sum_{i=1}^{n} (2i^2 + 3i) \\
&= \sum_{i=1}^{n} 2i^2 + \sum_{i=1}^{n} 3i \\
&= 2\frac{2n^3 + 3n^2 + n}{6} + 3\frac{n(n+1)}{2} \\
&= \frac{4n^3 + 15n^2 + 11n}{6} \\
&= \Theta(n^3)
\end{aligned}
$$

Theta Notation (Θ) Example

- Let $f(n) = 8n^2 + 2n – 3$. Let's show why f(n) is not in some other asymptotic class. First, let's show that $f(n) \neq \Theta(n)$.
- If this were true, we would have had to satisfy both the upper and lower bounds. The lower bound is satisfied because $f(n) = 8n^2 + 2n - 3$ does grow at least as fast asymptotically as n.
- But the upper bound is false. Upper bounds requires that there exist positive constants c_2 and n_0 such that $f(n) \leq c_2 n$ for all $n \geq n_0$.

- Informally we know that $f(n) = 8n^2 + 2n - 3$ will eventually exceed $c_2 n$ no matter how large we make c_2.
- To see this, suppose we assume that constants c_2 and n0 did exist such that $8n^2 + 2n - 3 \leq c_2 n$ for all $n \geq n_0$ since this is true for all sufficiently large n then it must be true in the limit as n tends to infinity. If we divide both sides by n, we have

$$\lim_{n \to \infty} \left(8n + 2 - \frac{3}{n} \right) \leq c_2.$$

- It is easy to see that in the limit, the left side tends to ∞. So, no matter how large c_2 is, the statement is violated. Thus $f(n) \neq \Theta(n)$.

- Let's show that $f(n) \neq \Theta(n^3)$. The idea would be to show that the lower bound $f(n) \geq c_1 n^3$ for all $n \geq n_0$ is violated. (c_1 and n_0 are positive constants). Informally we know this to be true because any cubic function will overtake a quadratic.
- If we divide both sides by n^3:

$$\lim_{n \to \infty} \left(\frac{8}{n} + \frac{2}{n^2} - \frac{3}{n^3} \right) \geq c_1$$

- The left side tends to 0. The only way to satisfy this is to set c1 = 0. But by hypothesis, c1 is positive. This means that $f(n) \neq \Theta(n^3)$.

References: Mathematical Series*

$$\sum_{k=1}^{n} k = \frac{1}{2} n (n+1)$$

$$\sum_{k=1}^{n} k^2 = \frac{1}{6} n (n+1) (2n+1)$$

$$\sum_{k=1}^{n} k^3 = \frac{1}{4} n^2 (n+1)^2$$

$$\sum_{k=1}^{n} k^4 = \frac{1}{30} n (n+1) (2n+1) \left(3n^2 + 3n - 1\right)$$

$$\sum_{k=1}^{n} k^5 = \frac{1}{12} n^2 (n+1)^2 \left(2n^2 + 2n - 1\right)$$

* http://mathworld.wolfram.com/PowerSum.html

References: Mathematical Series

$$\sum_{k=1}^{n} k^6 = \frac{1}{42} n(n+1)(2n+1)\left(3n^4 + 6n^3 - 3n + 1\right)$$

$$\sum_{k=1}^{n} k^7 = \frac{1}{24} n^2 (n+1)^2 \left(3n^4 + 6n^3 - n^2 - 4n + 2\right)$$

$$\sum_{k=1}^{n} k^8 = \frac{1}{90} n(n+1)(2n+1)\left(5n^6 + 15n^5 + 5n^4 - 15n^3 - n^2 + 9n - 3\right)$$

$$\sum_{k=1}^{n} k^9 = \frac{1}{20} n^2 (n+1)^2 \left(n^2 + n - 1\right)\left(2n^4 + 4n^3 - n^2 - 3n + 3\right)$$

$$\sum_{k=1}^{n} k^{10} = \frac{1}{66} n(n+1)(2n+1)\left(n^2 + n - 1\right)\left(3n^6 + 9n^5 + 2n^4 - 11n^3 + 3n^2 + 10n - 5\right)$$

Lecture 3

Asymptotic Notations – Insertion Sort

o-notation

- The asymptotic upper bound provided by big O-notation may or may not be asymptotically tight. The bound $2n^2=O(n^2)$ is asymptotically tight, but the bound $2n=O(n^2)$ is not.
- We use o-notation to denote an upper bound that *is not asymptotically tight*.
- We formally define o(g(n)) as the set

  $o(g(n))=\{f(n)$: for any positive constant $c>0$, there exists a constant $n_0>0$ such that $0 \leq f(n)<cg(n)$ for all $n \geq n_0\}$

  For example, $2n=o(n^2)$, but $2n^2 \neq o(n^2)$.

- The definition of *O*-notation and o-notation are similar. The main difference is that in *f(n)=O(g(n))*, the bound $0 \leq f(n) \leq cg(n)$ holds for some constant c>0, but in f(n)=o(g(n)), the bound $0 \leq f(n) < cg(n)$ holds for all constants *c>0.*
- Intuitively, in the o-notation, the function *f(n)* becomes insignificant relative to *g(n)* as n approaches infinity; that is,

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

ω-notation

- We use ω-notation to denote a lower bound that is not asymptotically tight. One way to define it is by $f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$.
- Formally, however we define $\omega(g(n))$ as the set $\omega(g(n))=\{f(n)$: for any positive constant $c>0$, there exists a constant $n_0>0$ such that $0 \leq cg(n)<f(n)$ for all $n \geq n_0\}$
- For example, $n^2/2=\omega(n)$, but $n^2/2 \neq \omega(n^2)$. The relation $f(n)= \omega(g(n))$ implies that

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

if the limit exists that is, $f(n)$ becomes arbitrarily large relative to $g(n)$ as n approaches infinity.

- Some readers may find it strange that we should write, for example, $n = O(n^2)$.
- In literature, O – notation is sometimes used informally to describe ***asymptotic tight bound***, i.e. what we have done using Θ–notation.

- However, in this course, when we write

  $f(n) = O(g(n))$, we are merely (purely) claiming that some constant multiple of $g(n)$ is an ***asymptotic upper bound*** on $f(n)$, with no claim about how tight an upper bound it is.

- Worst case
  - Provides an upper bound on running time
  - An absolute guarantee
- Average case
  - Provides the expected running time
  - Very useful, but treat with care: what is "average"?
    - Random (equally likely) inputs
    - Real-life inputs
- Best case
  - Provides lower bound on running time

Input Size

- In general, time taken by an algorithm grows with the size of the input, so it is traditional to describe the running time of the program as a function of ***size of its input***.
- **How we characterize input (for e.g.)**:
  - Sorting: number of input items
  - Multiplication: total number of bits
  - If the input to the algorithm is Graph, the ***input size*** can be described by the : number of nodes & edges
  - Etc

Running Time

- The running time of an algorithm on a particular input is the number of primitive steps or operations executed.
- For the next coming lectures we will assume a constant amount of time is required to execute each line of our pseudo code.
- One line may take different amount of time than another line, but we shall assume that each execution of the i^{th} line takes time c_i, where c_i is the constant.

Example: Insertion Sort (pseudo code)

```
Insertion_Sort(A, n)
{
  for j = 2 to n
  {
      key = A[j]
      i = j - 1;
      while (i > 0) and (A[i] > key)
      {
           A[i+1] = A[i]
           i = i - 1
      }

      A[i+1] = key
  }
}
```

30	10	40	20
1	2	3	4

An Example: insertion Sort

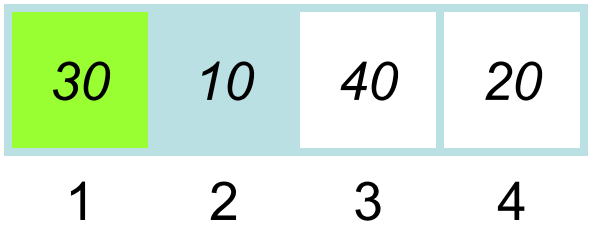
30	10	40	20
1	2	3	4

$j = \varnothing \quad i = \varnothing \quad key = \varnothing$

$A[i] = \varnothing \qquad A[i+1] = \varnothing$

```
jnsertjonSort(A, n) {
    for j = 2 to n {
        key = A[j]
        i = j - 1;
        whjle (i > 0) and (A[i] > key) {
            A[i+1] = A[i]
            i = i - 1
        }
        A[i+1] = key
    }
}
```

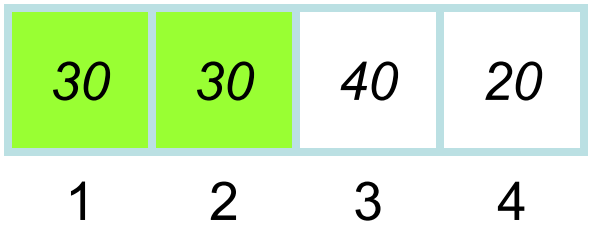
An Example: insertion Sort



j = 2	i = 1	key = 10
A[i] = 30		A[i+1] = 10

```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       while (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

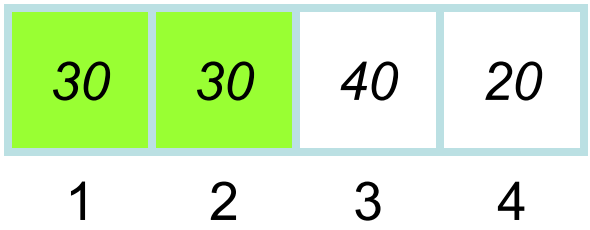
An Example: insertion Sort



j = 2	i = 1	key = 10
A[i] = 30		A[i+1] = 30

```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       while (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

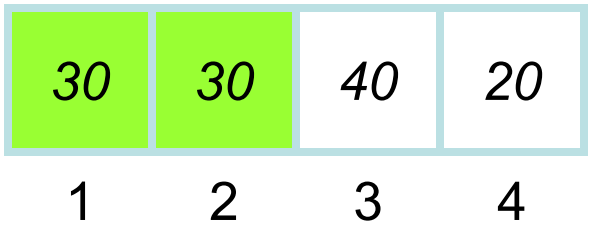
An Example: insertion Sort



j = 2	i = 1	key = 10
A[i] = 30		A[i+1] = 30

```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       while (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

An Example: insertion Sort

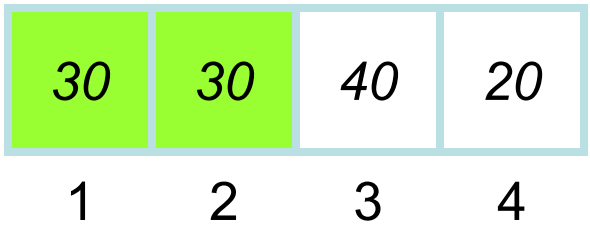


$j = 2 \quad i = 0 \quad key = 10$

$A[i] = \varnothing \qquad A[i+1] = 30$

```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       while (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

An Example: insertion Sort

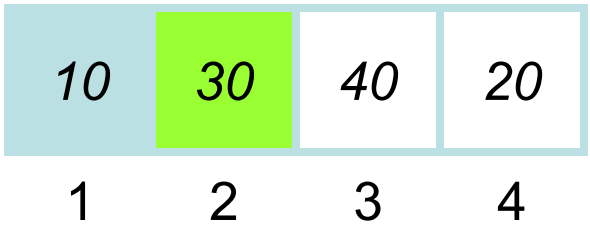


$j = 2 \quad i = 0 \quad key = 10$

$A[i] = \varnothing \qquad A[i+1] = 30$

```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       while (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

An Example: insertion Sort

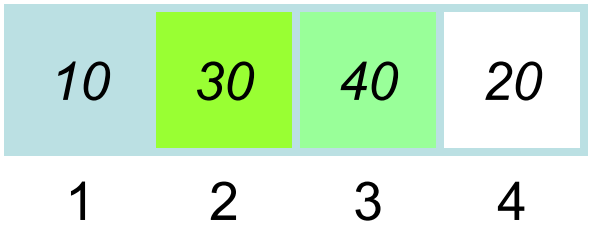


$j = 2 \quad i = 0 \quad key = 10$

$A[i] = \varnothing \qquad A[i+1] = 10$

```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       while (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

An Example: insertion Sort

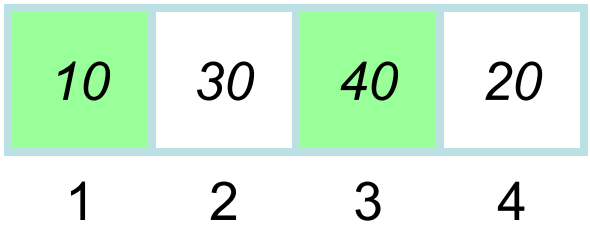


$j = 3 \quad i = 0 \quad key = 10$

$A[i] = \varnothing \qquad A[i+1] = 10$

```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       while (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

An Example: insertion Sort

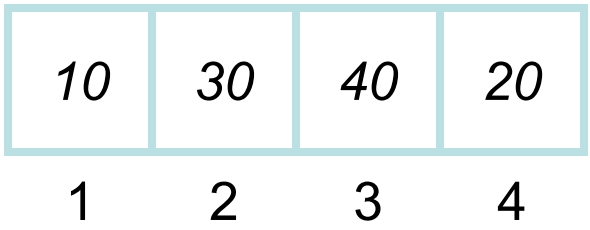


$j = 3 \quad i = 0 \quad key = 40$

$A[i] = \varnothing \qquad A[i+1] = 10$

```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       while (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

An Example: insertion Sort



$j = 3 \quad i = 0 \quad key = 40$

$A[i] = \varnothing \qquad A[i+1] = 10$

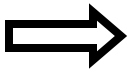
```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       while (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

An Example: insertion Sort

10	30	40	20
1	2	3	4

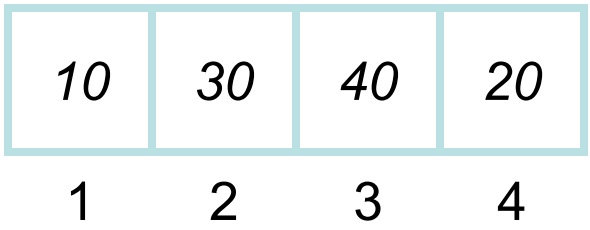
$j = 3 \quad i = 2 \quad key = 40$

$A[i] = 30 \quad A[i+1] = 40$



```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       while (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

An Example: insertion Sort

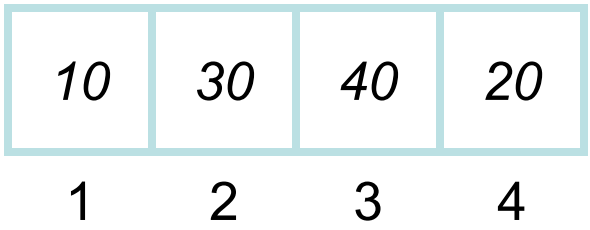


$j = 3 \quad i = 2 \quad key = 40$

$A[i] = 30 \quad A[i+1] = 40$

```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       while (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

An Example: insertion Sort

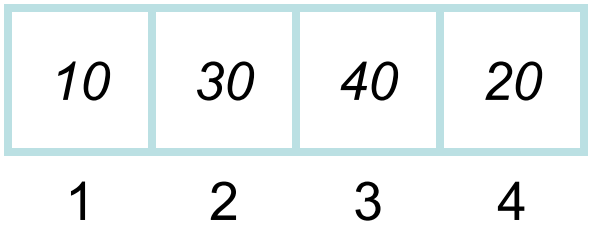


$j = 3 \quad i = 2 \quad key = 40$

$A[i] = 30 \quad A[i+1] = 40$

```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       while (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

An Example: insertion Sort

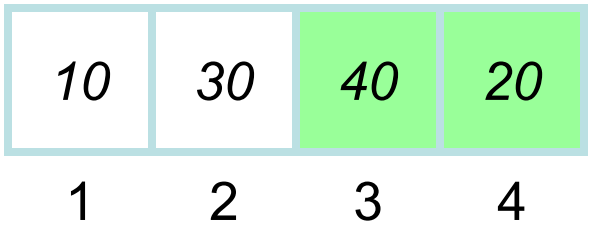


$j = 4 \quad i = 2 \quad key = 40$

$A[i] = 30 \quad A[i+1] = 40$

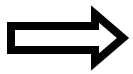
```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       while (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

An Example: insertion Sort



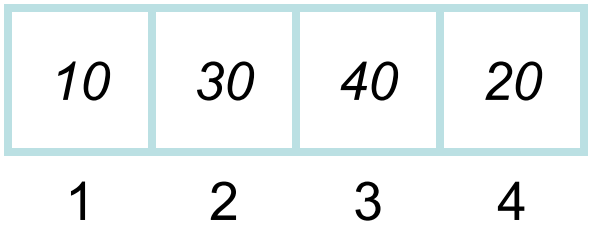
$j = 4 \quad i = 2 \quad key = 20$

$A[i] = 30 \quad A[i+1] = 40$



```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       while (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

An Example: insertion Sort



$j = 4 \quad i = 2 \quad key = 20$

$A[i] = 30 \quad A[i+1] = 40$

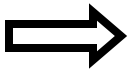
```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       while (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

An Example: insertion Sort

10	30	40	20
1	2	3	4

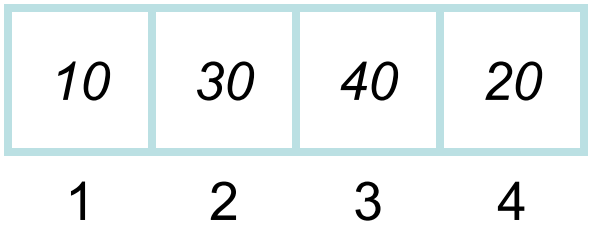
$j = 4 \quad i = 3 \quad key = 20$

$A[i] = 40 \quad A[i+1] = 20$



```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       whjle (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

An Example: insertion Sort

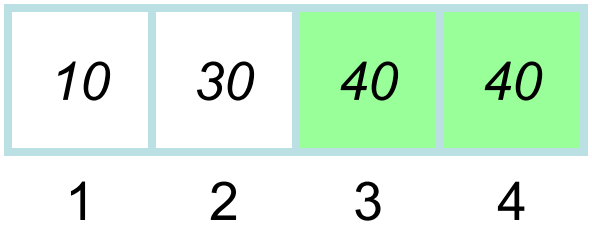


$j = 4 \quad i = 3 \quad key = 20$

$A[i] = 40 \quad A[i+1] = 20$

```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       whjle (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

An Example: insertion Sort

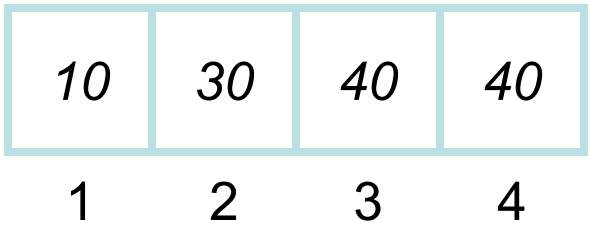


$j = 4 \quad i = 3 \quad key = 20$

$A[i] = 40 \qquad A[i+1] = 40$

```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       whjle (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

An Example: insertion Sort

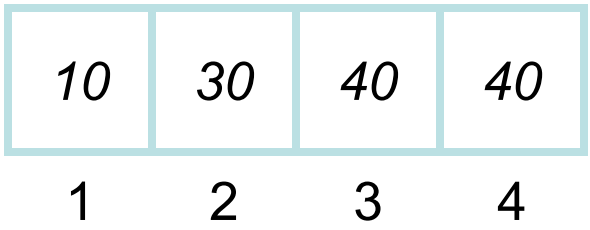


$j = 4 \quad i = 3 \quad key = 20$

$A[i] = 40 \qquad A[i+1] = 40$

```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       whjle (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

An Example: insertion Sort



$j = 4 \quad i = 3 \quad key = 20$

$A[i] = 40 \qquad A[i+1] = 40$

```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       whjle (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

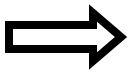
An Example: insertion Sort

10	30	40	40
1	2	3	4

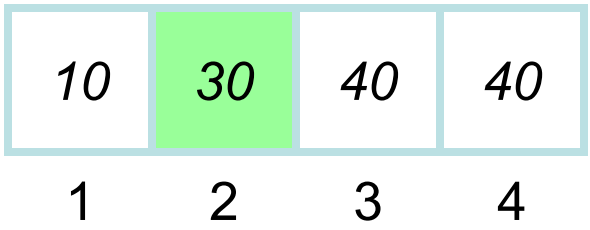
$j = 4 \quad i = 2 \quad key = 20$

$A[i] = 30 \quad A[i+1] = 40$

```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       whjle (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```



An Example: insertion Sort

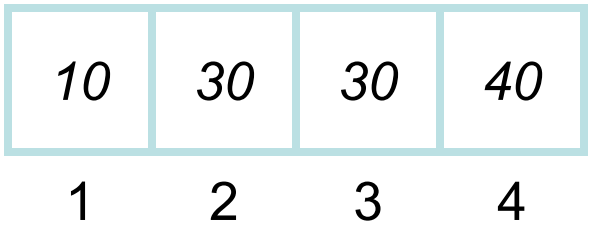


$j = 4 \quad i = 2 \quad key = 20$

$A[i] = 30 \quad A[i+1] = 40$

```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       whjle (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

An Example: insertion Sort

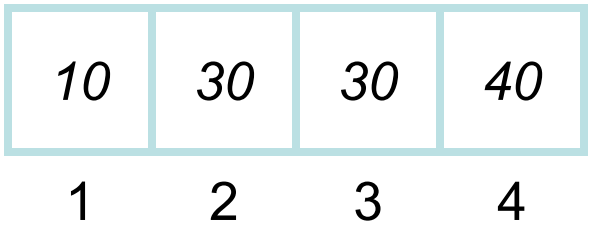


$j = 4 \quad i = 2 \quad key = 20$

$A[i] = 30 \qquad A[i+1] = 30$

```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       whjle (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

An Example: insertion Sort



$j = 4 \quad i = 2 \quad key = 20$

$A[i] = 30 \qquad A[i+1] = 30$

```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       whjle (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

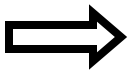
An Example: insertion Sort

10	30	30	40
1	2	3	4

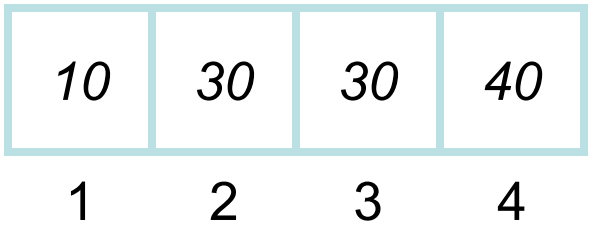
$j = 4 \quad i = 1 \quad key = 20$

$A[i] = 10 \quad A[i+1] = 30$

```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       whjle (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```



An Example: insertion Sort

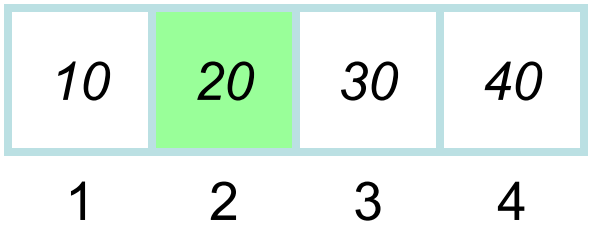


$j = 4 \quad i = 1 \quad key = 20$

$A[i] = 10 \quad A[i+1] = 30$

```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       whjle (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

An Example: insertion Sort

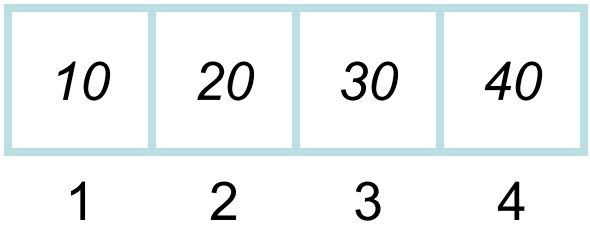


$j = 4 \quad i = 1 \quad key = 20$

$A[i] = 10 \quad A[i+1] = 20$

```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       whjle (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

An Example: insertion Sort



$j = 4 \quad i = 1 \quad key = 20$

$A[i] = 10 \quad A[i+1] = 20$

```
jnsertjonSort(A, n) {
   for j = 2 to n {
       key = A[j]
       i = j - 1;
       whjle (i > 0) and (A[i] > key) {
             A[i+1] = A[i]
             i = i - 1
       }
       A[i+1] = key
   }
}
```

Done!

- We start by presenting the insertion sort with the time “cost” of each statement and the number of times each statement is executed.

- For each value of j = 2,3,4…… n, where n = length of array (input), we say t_j *be the number of times the while loop test in above program is executed for that value of* j.

Insertion Sort

Statement	cost	times
`InsertionSort(A, n) {`		
`  for j = 2 to n {`	c_1	n
`  key = A[j]`	c_2	(n-1)
`  i = j - 1;`	c_4	(n-1)
`  while (i > 0) and (A[i] > key){`	c_5	A
`  A[i+1] = A[i]`	c_6	B
`  i = i - 1`	c_7	C
`  }`	0	
`  A[i+1] = key`	c_8	(n-1)
`  }`	0	
`}`		

Where

$$A = \sum_{j=2}^{n} tj$$

$$B = \sum_{j=2}^{n} tj - 1$$

$$C = \sum_{j=2}^{n} tj - 1$$

tj means t_j

- So The running time of the algorithm is the sum of each statement executed. To compute T(n), the total running time of INSERTION SORT, we sum the products of the cost and times columns, obtaining

T(n) = c1n + c2(n-1) + c4(n-1) +

$$c5\sum_{j=2}^{n} tj + c6\sum_{j=2}^{n} (tj-1) + c7\sum_{j=2}^{n} (tj-1)$$

+ c8(n-1)

- The best case occurs if array is already sorted. Thus tj = 1 for all j = 2,3,4….n, and the best case running time would be

$$T(n) = c1n + c2(n-1) + c4(n-1) + c5(n-1) + c8(n-1)$$
$$= (c1+c2+c4+c5+c8)n - (c2+c4+c5+c8)$$

- This running time can be expressed as an + b for constants a & b; depends on the statement costs: thus it is a ***linear function*** of n.

- If array is in reverse sorted order, then worst case running time would be resulting.
- We have the mathematical formula:

$$\sum_{j=1}^{n} j = n\frac{n+1}{2}$$

- So we will have

$$\sum_{j=2}^{n} j = n\frac{n+1}{2} - 1$$

- Similarly we have

$$\sum_{j=2}^{n} j-1 \quad = \quad n\frac{n-1}{2}$$

- We find that worst case running time of Insertion sort is:

T(n) = c1n + c2(n-1) + c4(n-1) + $c5(n\frac{n+1}{2}-1)$ +

$$c6(n\frac{n-1}{2}) \quad + \quad c7(n\frac{n-1}{2}) \qquad + c8(n-1)$$

The worst case running time can be expressed as $an^2 + bn + c$; thus a ***quadratic function*** of n.

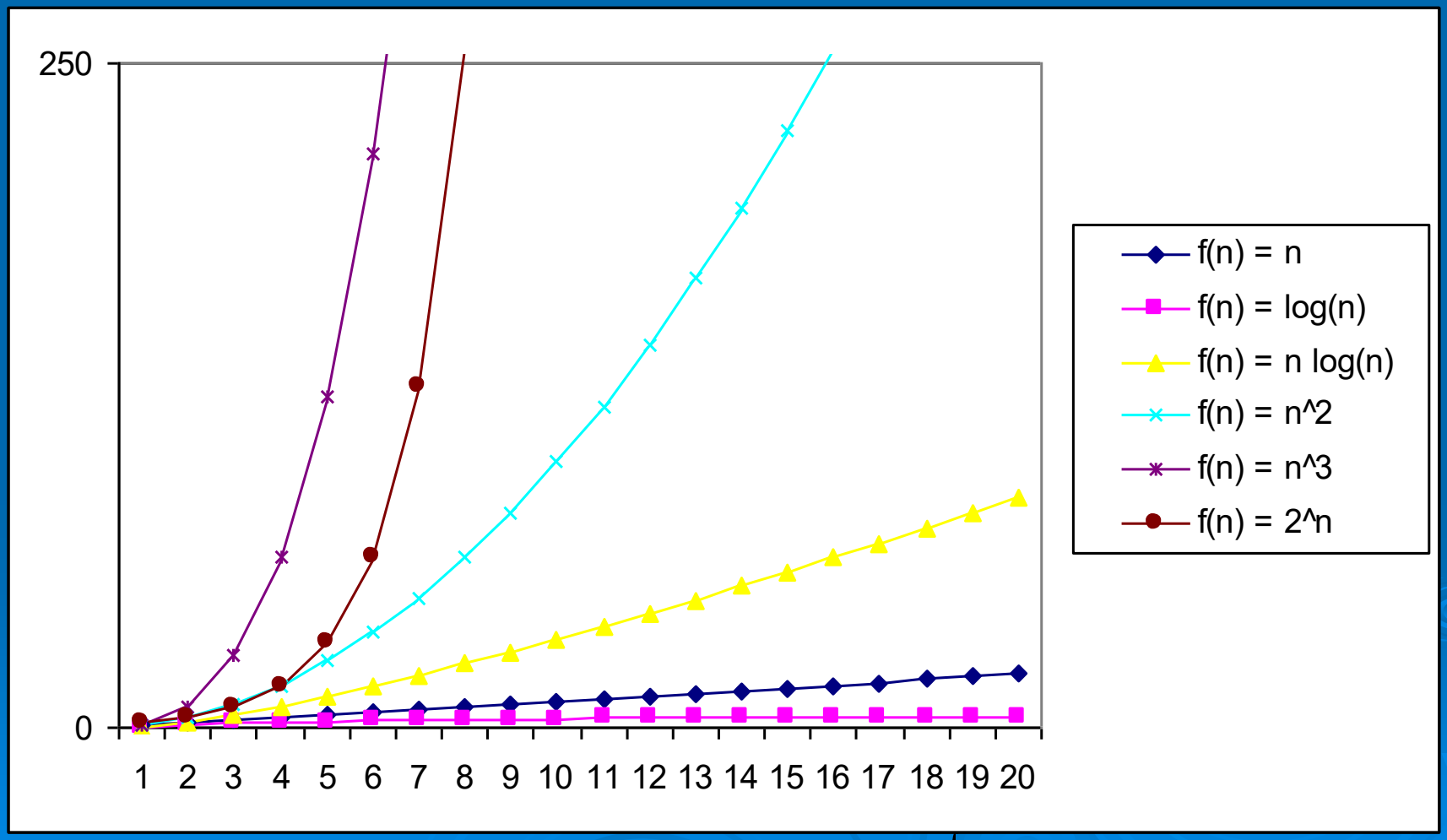
- We say Insertion Sort's run time is $O(n^2)$
  - Properly we should say run time is *in* $O(n^2)$

- Questions
  - Is InsertionSort $O(n)$?
  - Is InsertionSort $O(n^3)$?
  - Is InsertionSort $\Omega(n)$?

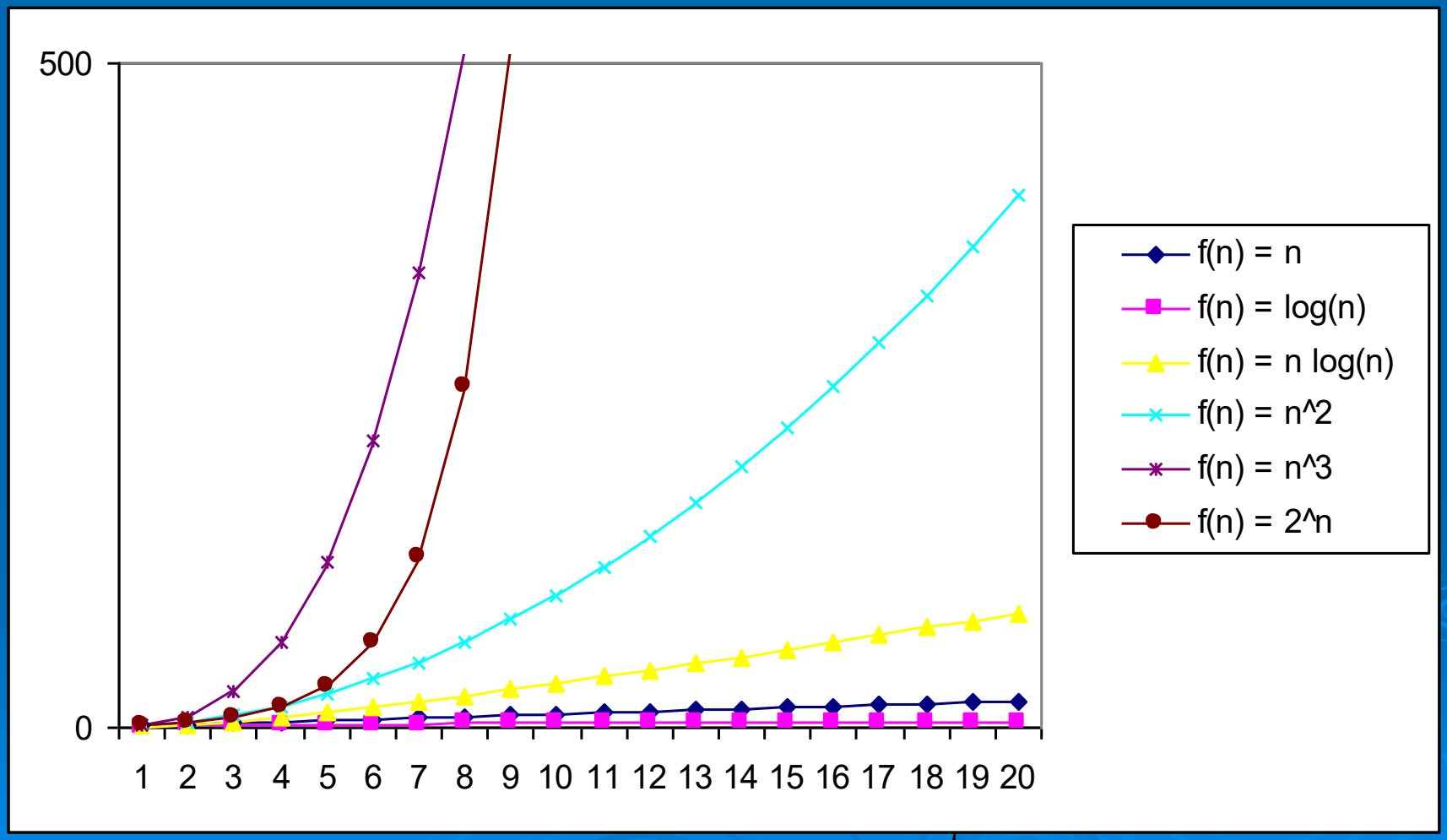
- We have studied insertion sort. The best case running time of this algorithm is $\Omega(n)$ which implies that the running time of insertion sort is $\Omega(n)$.
- So it means running time of insertion sort falls *between* $\Omega(n)$ and $O(n^2)$.
- It does not means that **running time** of insertion sort is $\Omega(n^2)$.
- It is not contradictory, however to say that **worst case running time** of insertion sort is $\Omega(n^2)$, since there is an input which causes the algorithm to take $\Omega(n^2)$ time.

- When we say that the **running time** (no modifier) of an algorithm is Ω(g(n)), we mean that no matter what particular input of size n is chosen , the running time on that input is at least constant times g(n) for *sufficiently large* n.

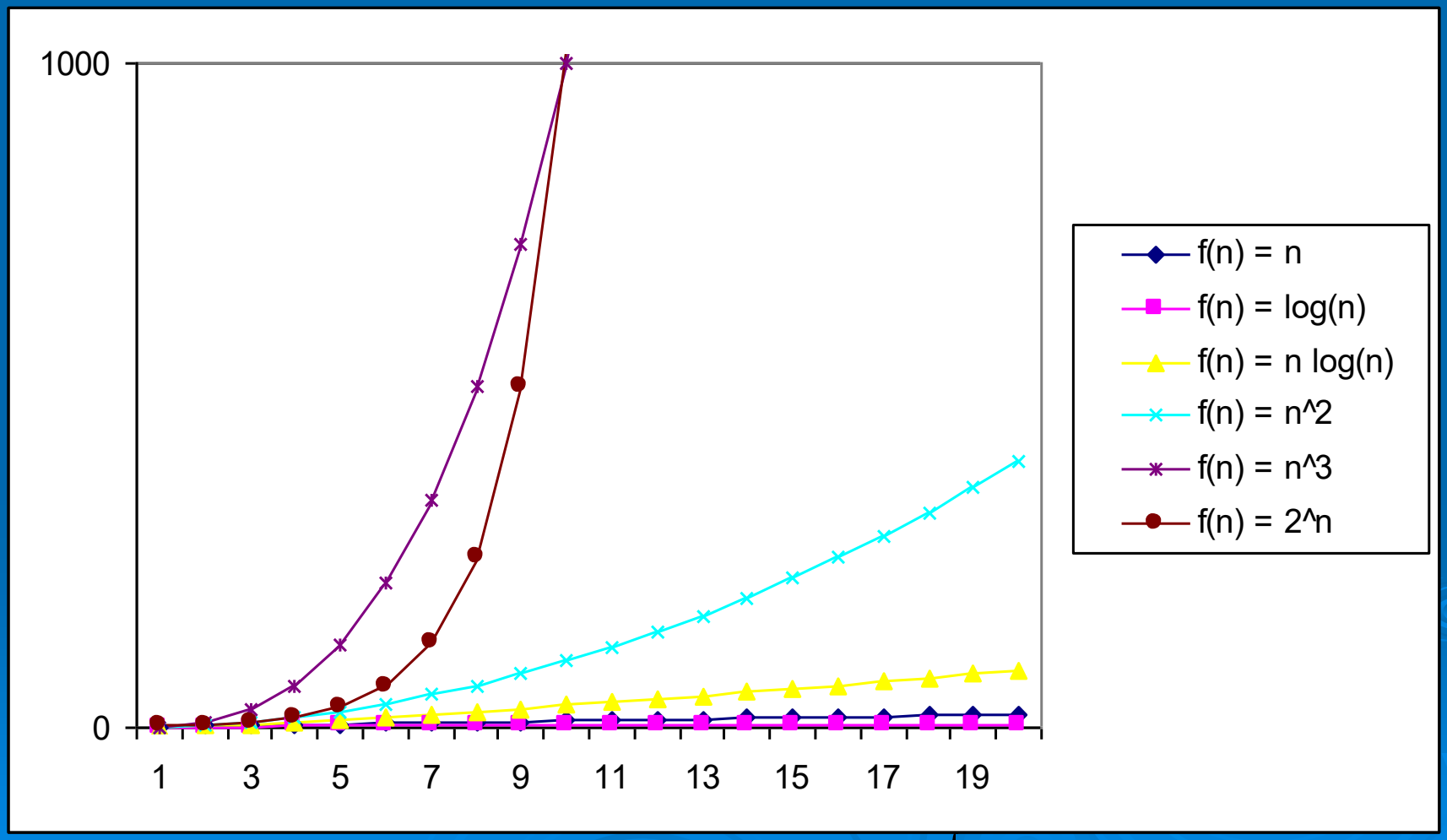
Practical Complexity



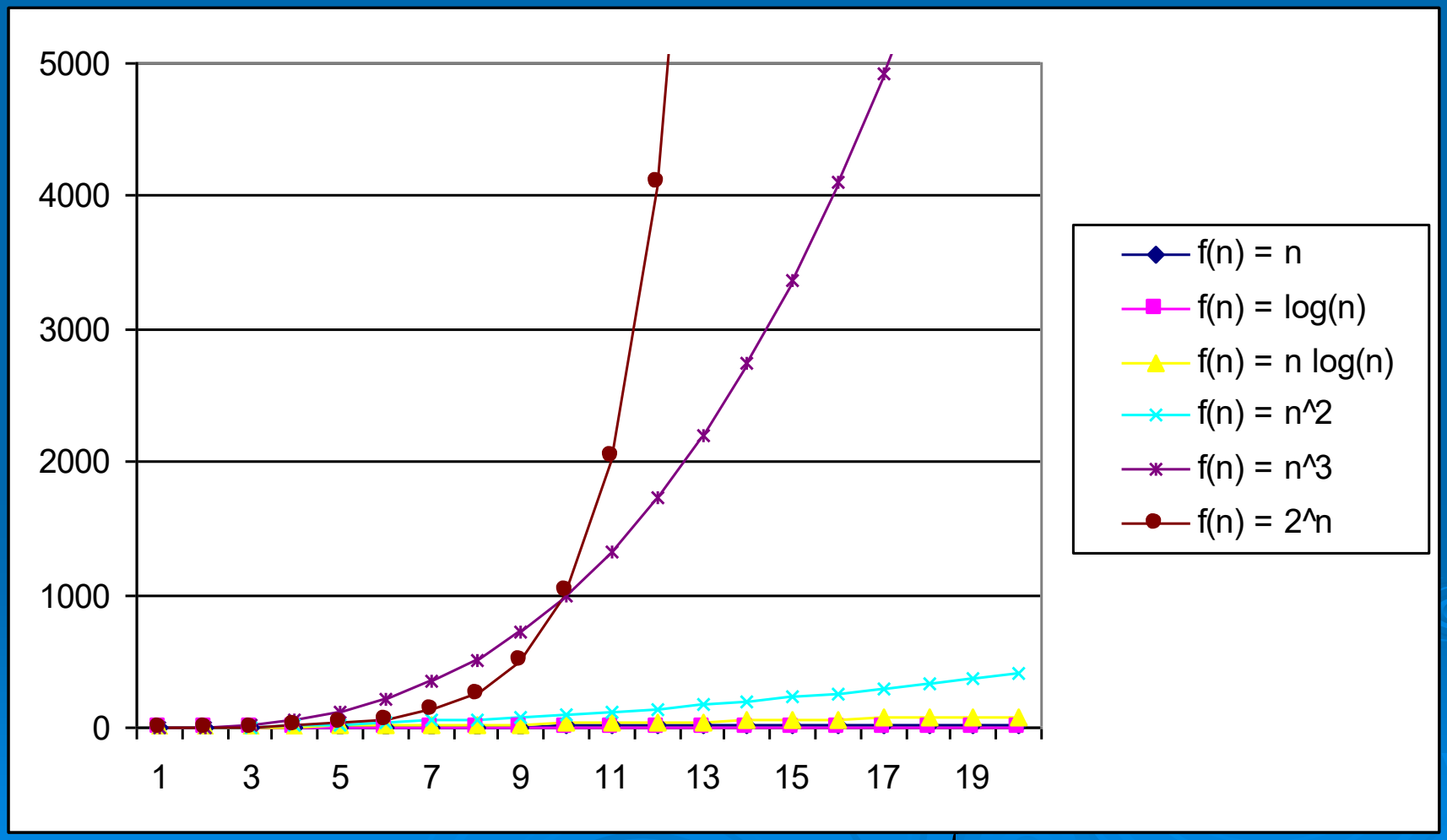
Practical Complexity



Practical Complexity



Practical Complexity



Useful Properties

- **Transitivity**

  $f(n) = \Theta(g(n)) \ \& \ g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
  $f(n) = O(g(n)) \ \& \ g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
  $f(n) = \Omega(g(n)) \ \& \ g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$

- **Reflexivity**

  $f(n) = \Theta(f(n))$
  $f(n) = O(f(n))$
  $f(n) = \Omega(f(n))$

- **Symmetry**

  $f(n) = \Theta(g(n))$ *if and only if* $g(n) = \Theta(f(n))$

Floors and ceilings

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$$

$$\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$$

$$\lceil \lceil n/a \rceil / b \rceil = \lceil n/ab \rceil$$

$$\lfloor \lfloor n/a \rfloor / b \rfloor = \lfloor n/ab \rfloor$$

$$\lceil a/b \rceil \leq (a + (b-1))/b$$

$$\lfloor a/b \rfloor \geq (a - (b-1))/b$$

$\lfloor x \rfloor$

Stands for “floor of x”
(greatest integer less than or equal to x)

$\lceil x \rceil$

Stands for “ceiling of x”
(least integer greater than or equal to x)

Lecture 4

- Divide & Conquer Approach
- Recurrences

Designing Algorithms

- There are many ways to design algorithms.
- In insertion sort we used an incremental approach: i.e. having sorted the sub list of 1…… j-1 , we insert a single element x, at its proper place which yields again the sorted sub list of 1……. j elements.
- In the following section, we are going to study another approach of designing algorithms, known as Divide and Conquer approach.

Divide and Conquer Approach

- Divide-and-conquer is a technique for designing algorithms that consists of breaking the problem into several sub-problems that are similar to the original problem but smaller in size,
- Solve the sub-problem recursively (successively and independently), **and**
- Then combine these solutions to create a solution to the original problem.

The divide and conquer paradigm involves three steps at each level of recursion.

- ***Divide*** the problem into number of sub problems
- ***Conquer*** the sub problems by solving them recursively. If the sub problem sizes are small enough then solve them in straight forward manner.
- ***Combine*** the solutions to the sub problems into the solution for the original problem.

Merge Sort

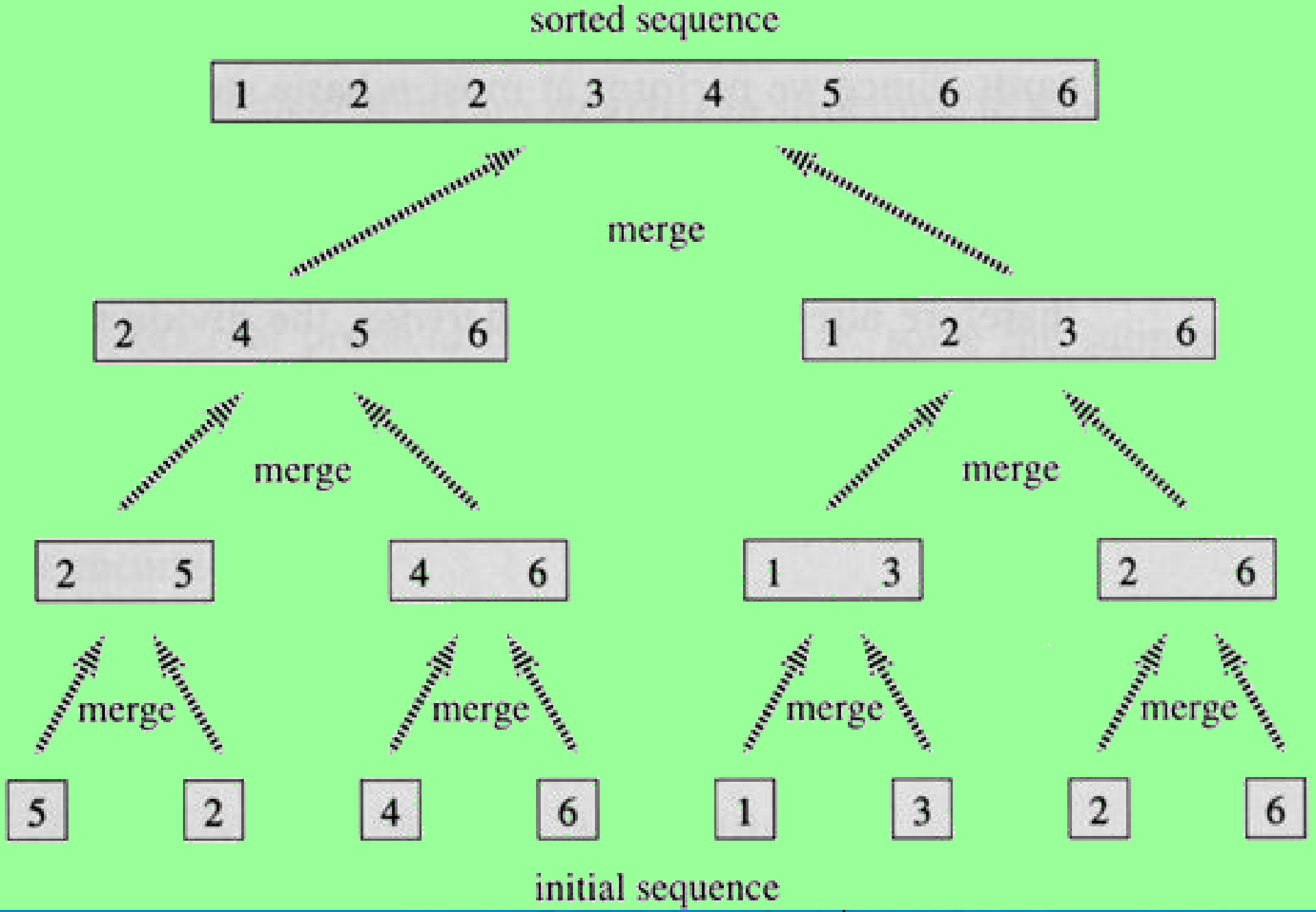
The **Merge sort** algorithm closely follows the divide and conquer paradigm. It works as follows.

- ***Divide*** : Divide the n - element sequence to be sorted into sub sequences each of size n / 2 elements.
- ***Conquer*** : Sort the two sub sequences recursively using merge sort.
- ***Combine*** : Merge the two sorted sub sequences to produce another sorted list.

- The key operation of the merge sort algorithm is the merging of two sub sequences in the "combine" step.

- To perform merging, we use Merge(A, left, mid, right); function, where A is an array of elements and left, right and mid representing leftmost, rightmost and center indices of an array respectively.

- The above function assumes that the sub array A[left…..mid] and A[mid+1 …… right] are in sorted order where n = right - left + 1


sorted sequence
1 2 2 3 4 5 6 6
merge
2 4 5 6
1 2 3 6
merge
merge
2 5
4 6
1 3
2 6
merge
merge
merge
merge
5
2
4
6
1
3
2
6
initial sequence

Analyzing Divide and Conquer Algorithms

- When an algorithm contains a recursive call to itself, its running time can often be described by a ***recurrence equation*** or ***recurrence***, which describes the overall running time on a problem of size n in terms of running time on smaller inputs or ***Recurrence*** is an equation or an inequality that describes the function in terms of its value on smaller inputs.
- We can use mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm.

- A recurrence for the running time of a Divide-and-Conquer algorithm is based on three steps of basic paradigm.
- Let T(n) is the running time of problem of size n. if the problem size is small enough , say n <= c for some constant c, the solution takes constant time, which we write as Θ(1).
- Suppose our division of the sub problem yields **a** sub problems, each of size **1/b** of the size of original. If we take **D(n)** time to divide the problem into sub problems and **C(n)** time to combine the solutions to sub problems into the solution to the original problem, we get the following recurrence.

$$T(n) = \begin{cases} \Theta(1) & if n \le c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & otherwise \end{cases}$$

- We will see later how to solve common recurrences of this form.

Analysis of Merge Sort

- Although the merge sort algorithm works fine when the number of elements are not even, for our simplicity we assume that the original problem size is a power of 2. later we will see that this assumption does not effect the order of growth of the solution to recurrence.

- Merge sort on just one element takes constant time. When $n > 1$, we break down the running time as follows.

- ***Divide*** : The Divide step just computes the middle of the sub array which takes constant time. Thus $D(n) = \Theta(1)$

- ***Conquer*** : We recursively solve two problems, each of size $n/2$ which contributes

  $T(n/2) + T(n/2) = 2T(n/2)$ to the running time

- ***Combine*** : Merge procedure on n-elements takes $\Theta(n)$ time, ***how ????***.

  So $C(n) = \Theta(n)$

Analysis of Merge Sort

Statement	Effort :- T(n)

```
MergeSort(A, left, right)
{
    if (left < right)                          Θ(1)
    {
        mid = floor((left + right) / 2);       Θ(1)
        MergeSort(A, left, mid);               T(n/2)
        MergeSort(A, mid+1, right);            T(n/2)
        Merge(A, left, mid, right);            Θ(n)
    }
}
```

Analysis of Merge Sort

```
MERGE( array A, int p, int q int r)
 1   int B[p..r]; int i ← k ← p; int j ← q + 1
 2   while (i ≤ q) and (j ≤ r)
 3   do if (A[i] ≤ A[j])
 4         then B[k++] ← A[i++]
 5         else B[k++] ← A[j++]
 6   while (i ≤ q)
 7   do B[k++] ← A[i++]
 8   while (j ≤ r)
 9   do B[k++] ← A[j++]
10   for i ← p to r
11   do A[i] ← B[i]
```

- We have D(n) + C(n) = Θ(1) + Θ(n). This sum is a linear function of n, i.e. Θ(n) . The worst case running time T(n) of merge sort:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(1) + \Theta(n) & n > 1 \end{cases}$$

OR

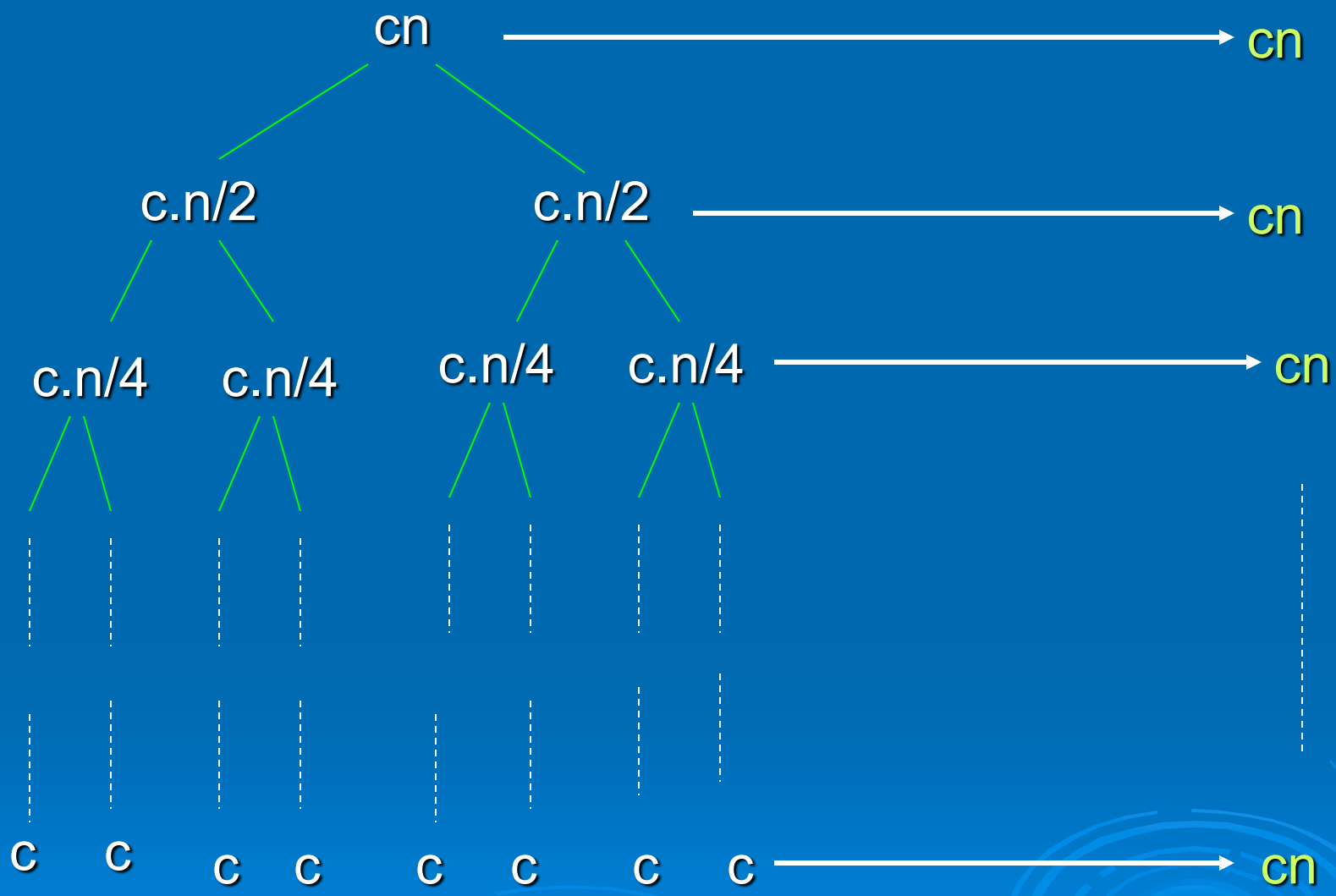
$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & n > 1 \end{cases}$$

- We shall see in “Master Theorem” (to be discussed later) that T(n) for Merge Sort is **Θ(n.lgn)** [lgn means $\log_2 n$].
- Even we can predict and prove this time without using master theorem. Lets rewrite the above recurrence.

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & n > 1 \end{cases}$$

- Where c represents the time required to solve the problem of size 1.

Construction of recurrence tree for the recurrence T(n) = 2T(n/2) + cn



- Now we add the costs of each level:

  Top level : cn

  Next level from top: c.n/2 + c.n/2 = cn

  Next level : c.n/4 + c.n/4 + c.n/4 + c.n/4 = cn

- In general each level i (starting from 0) has 2^i nodes and each node contributing the cost $c(n/2^i)$.
- So that the total cost the ith level has the total cost $2^i c(n/2^i) = cn$
- At the bottom level, there are n nodes each contributing a cost of c, for total cost of cn.

- The total number of levels of above recursion tree are lgn + 1.

- So to compute total cost represented by the recurrence, we simply add up all costs of all levels. There are lgn + 1 levels and each is costing cn, for total cost of

$$cn(\lg n + 1) = cn\lg n + cn$$

ignoring lower order terms and the constant c we get the desired result of $\Theta(n.\lg n)$.

- The worst case running time T(n) of merge sort could be described by the recurrence:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & n > 1 \end{cases}$$

Whose solution was claimed to be T(n) = Θ (n.lg n)

- We will study three methods of solving recurrences I.e to obtain the asymptotic *O* or Θ bounds on the solution.

Recurrence Examples

$$s(n) = \begin{cases} 0 & n = 0 \\ c + s(n-1) & n > 0 \end{cases}$$

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + c & n > 1 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

Methods for Solving Recurrences

1. Substitution method
2. Recursion Tree (iteration) method
3. Master method

- The recurrence describing the worst case running time of Merge Sort is really :

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \Theta(n) & n > 1 \end{cases}$$

the recurrence that arise from the running time of an algorithm generally have T(n) = Θ (1), for sufficiently small n. consequently we generally omit the statements of the boundary conditions of recurrences and assume that T(n) is constant for small n.

- When we state and solve recurrences, we often omit floors, ceilings and boundary conditions. We move ahead and later see whether or not they matter. They usually don't but it is important to know when they do so.

Substitution Method

- The substitution method entails two steps
  - Guess the form of the solution
  - Use mathematical induction to find the constants and show that the solution works.
- This method is powerful but is obviously applied only in cases when it is easy to guess the form of the answer.
- The substitution method can be used to establish either upper or lower bounds on the recurrence.

- As an example, lets determine an upper bound on the recurrence

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

- We ***guess*** that the solution is **$T(n) = O(n.\lg n)$**. It means that we have to prove that **$T(n) \leq cn.\lg n$** for an appropriate choice of the constant **$c > 0$**.

- Similarly from above we can say that

  $T(n/2) \leq c.n/2.\lg n/2$.

- We have

$$T(n) = 2T(n/2) + n$$
$$\leq 2.c.n/2.\ \lg n/2 + n$$
$$= c.n.\ \lg n/2 + n$$
$$= c.n\lg n - c.n\lg_2 2 + n$$
$$= c.n\lg n - c.n + n$$

ignoring small terms from above

$$\leq c.n\lg n$$

Where the last step holds as long as $c \geq 1$.

So from above it is proved that **$T(n) = O(n.\ \lg n\ )$**.

Making a good guess

- Unfortunately there is no general way to have a good guess. Guessing a solution needs experience and occasionally creativity.
- We can use recursion trees for generating good guess, which we saw while analyzing Merge Sort.
- Another way to to make good guess is prove for loose upper and lower bounds on the recurrence and the reduce the range of uncertainty. We can gradually lower the upper bound and raise the lower bound until we converge on the correct, asymptotically correct solution

Avoiding Pitfalls

- We can misinterpret the asymptotic notation if we say that our guess for **T(n) = 2T(n/2) + n** is **O(n)**. I.e. **T(n) ≤ cn** and then arguing that

$$T(n) = 2T(n/2) + n$$
$$\leq 2.c.n/2 + n$$
$$= cn + n \text{ or } n.(c + 1)$$
$$= O(n) \leftarrow \textit{Wrong !!!}$$

The error is that we haven't proved the exact form of our hypothesis I.e. **T(n) ≤ cn**

Recursion Tree Method

- As we found that it is sometime difficult to have a good guess in case of substitution method.
- Drawing out the recursion tree as we did in the analysis of Merge Sort is the straightforward way to have a good guess.
- In this method each node represents the cost of each sub problem somewhere in the set of recursive function invocations. We sum all the costs of sub problems at all levels of recursion to find the logical guess for our solution.

- This method is more useful for describing the running time of divide and conquer problems.

- This method is used to generate a very good guess which can be later verified by the substitution method. However you can use a recursion tree method as a direct proof of a solution to the recurrence.

Master's Theorem

- The Master Method provides the bound for the recurrence of the form

  $$T(n) = aT(n/b) + f(n)$$

  where $a >= 1$, $b > 1$ and $f(n)$ is a given function.

- The above recurrence describes the running time of an algorithm that divides the problem of size n into a sub problems, each of size n/b, where a and b are positive constants.

- Then a sub problems are solved recursively, each in time T(n/b).

- The cost of dividing problem and combining the results of the sub problems is described by the function f(n) i.e.

  $f(n) = D(n) + C(n)$

- This method requires the memorization of three cases as follows

The Master Theorem

- if T(n) = aT(n/b) + f(n) then

$$T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right) & f(n) = O\left(n^{\log_b a - \varepsilon}\right) \\ \Theta\left(n^{\log_b a} \log n\right) & f(n) = \Theta\left(n^{\log_b a}\right) \\ \Theta\left(f(n)\right) & f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right) \text{AND} \\ & af(n/b) \le cf(n) \text{ for large } n \end{cases} \quad \begin{array}{l} \varepsilon > 0 \\ c < 1 \end{array}$$

- You can go through Book (second edition) page no 73 for clear understanding of previous three cases.

Master Method -- Examples

- Let T(n) = 9T(n/3) + n
  - a=9, b=3, f(n) = n
  - Now $n^{\log_b a} = n^{\log_3 9} = n^2 = \Theta(n^2)$
  - So we need to subtract power from 2, so that it seems to be like f(n) which is = n
  - So let ε=1 and
  - f(n) = $O(n^{\log_3 9 - \varepsilon})$, where ε=1, case 1 applies: i.e.

$$T(n) = \Theta\left(n^{\log_b a}\right) \text{when } f(n) = O\left(n^{\log_b a - \varepsilon}\right)$$

  - f(n) = $O(n^{2-1})$
  - Thus the solution is $T(n) = \Theta(n^2)$

Example 2:

- Let $T(n) = T(2n/3) + 1$
  - a=1, b=3/2, f(n) = 1
  - Now $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$
  - So we can apply case 2:
  - i.e if $f(n) = \Theta(n^{\log_{3/2} 1}) = \Theta(1)$,
  - Thus the solution is $T(n) = \Theta(\log_2 n)$

Example 3

- Let T(n) = 3T(n/4) + n.log n
  - a=3, b=4, f(n) = n.log n
  - Now $n^{\log_b a} = n^{\log_4 3} = n^{0.8}$
  - So we need to add some factor with power (0.8), so that it seems to be like f(n) which is = n.log n , which corresponds to third case of our theorem.
  - So let ε=0.2 and we have $n^{0.8+0.2} = n^1$ (near to f(n))
  - Now we have to fulfill the second part of this case i.e.
  - af(n/b) ≤ c.f(n) for c < 1
  - 3f(n/4) ≤ c. n. log n
  - 3. n/4. log (n/4) ≤ c. n. log n let c = ¾ < 1
  - 3/4. n. log (n/4) ≤ 3/4. n. log n
  - Above inequality is proved so we have T(n) = Θ(n.logn)

Assignment # 2

- Use the Master method to give tight asymptotic bounds for the following recurrences

  $T(n) = 4T(n/2) + n$

  $T(n) = 4T(n/2) + n^2$

  $T(n) = 4T(n/2) + n^3$

  $T(n) = 2T(n/4) + n^{1/2}$

Due Next Week, same day, same time

Lecture # 5

- Elementary Data Structures
- Hashing

Divide-and-Conquer Algorithm

- The simplest application of divide-and-conquer algorithm is Binary Search algorithm.
- Before going to binary search we analyze linear search as it takes $\Omega(n)$ time in *worst case* and $O(1)$ in the *best case*.
- The *pre-requisite* for binary search algorithm is that the list must be sorted in some order.

```
void b_search :: search(int m,int n)
{
  low=m;
  high=n;
  temp = mid;
  mid=(low+high)/2;
  if( (temp == mid) && (a[mid] < num) )
              mid++;
  if( (temp == mid) && (a[mid] > num) )
              mid--;
  if(a[mid]==num)
   {
      test=1;
      return;     }
  if(a[mid]>num)
     high=mid;
  if(a[mid]<num)
     low=mid;
  if(low>=high)
     return;
  search(low,high);
}
```

Subject to improve …

- In above algorithm (code) each pass to this b_search (int m, int n) function reduces the array to be searched by at least one half the size of the sub list still to be searched.

- The last pass occurs when size of the sub list reaches one.

- Thus the total number of recursive function calls is 1 plus the number k passes required to produce a list of size 1.

- Since the size of the sub list after k passes is at most $n/2^k$. We must have

$$n/2^k < 2$$

i.e.

$n < 2 * 2^k$

$n < 2^{k+1}$

Taking $\log_2$ on both sides

$\log_2 n < \log_2 2^{k+1}$

$\log_2 n < K+1 * \log_2 2$

$\log_2 n < K+1 * 1$

$\log_2 n < K+1$

so

- So the required number of passes, therefore is the smallest integer that satisfies this inequality.

- So the Time Complexity of binary search is $O(\log_2 n)$.

- How the two discussed divide - and - conquer algorithms differ – Merge sort and Binary Search **???**
- What would be the recurrence relation for binary search algorithm **???**

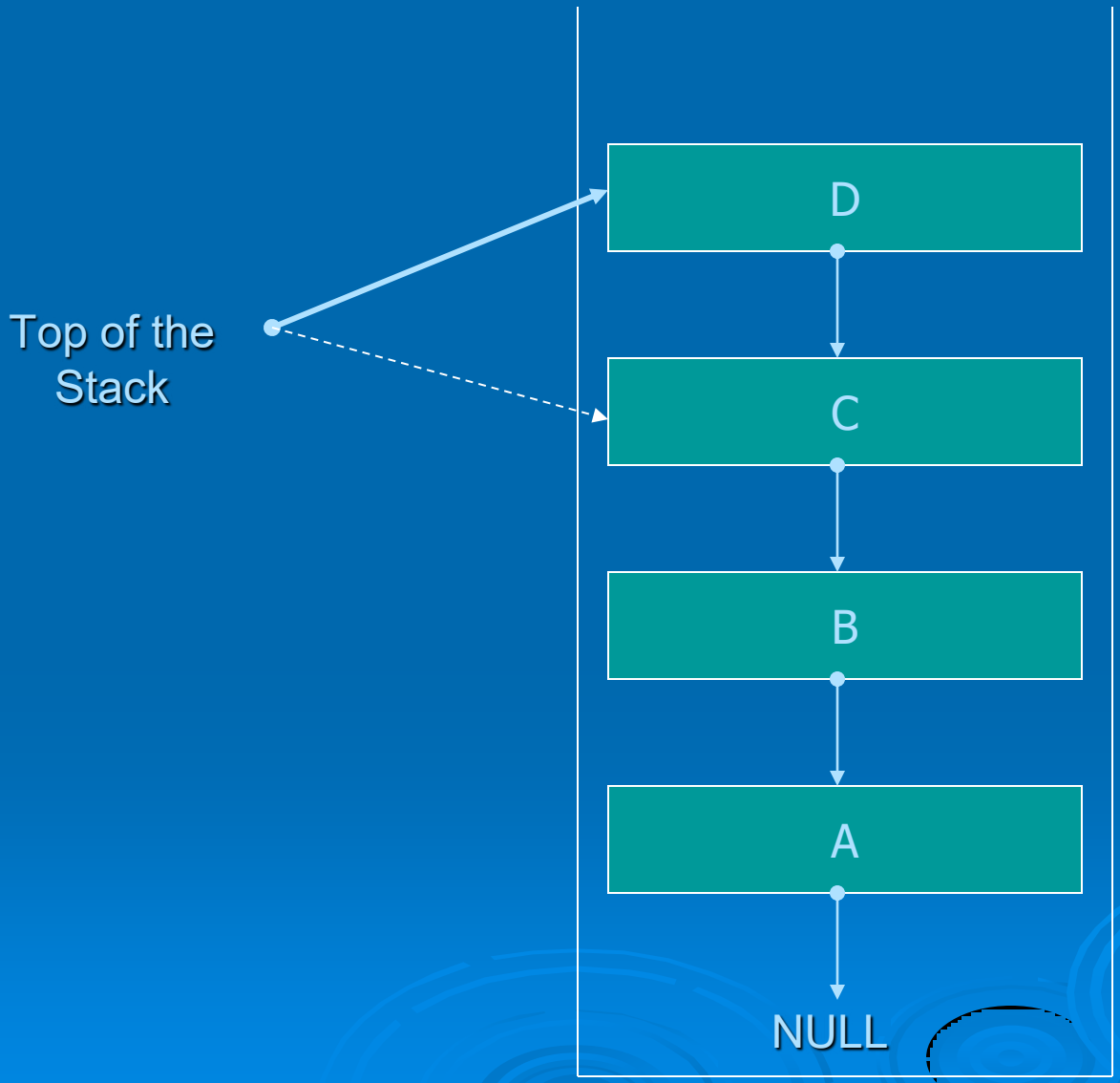
$$T(n) = \begin{cases} \Theta(1) & n \leq 2 \\ T\left(\frac{n}{2}\right) + \Theta(1) & n > 2 \end{cases}$$

Elementary Data Structures

- A stack is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end only, called **top** of the stack. i.e. deletion/insertion can only be done from **top** of the stack.

- The insert operation :- **Push**
- The delete operation :- **Pop**

- Figure showing stack…..



- Stack is also referred as Last-in First-out i.e LIFO.
- empty means stack underflows
- If top of the stack exceeds n, the stack overflows.

- **Time Complexity:**
  - Push :

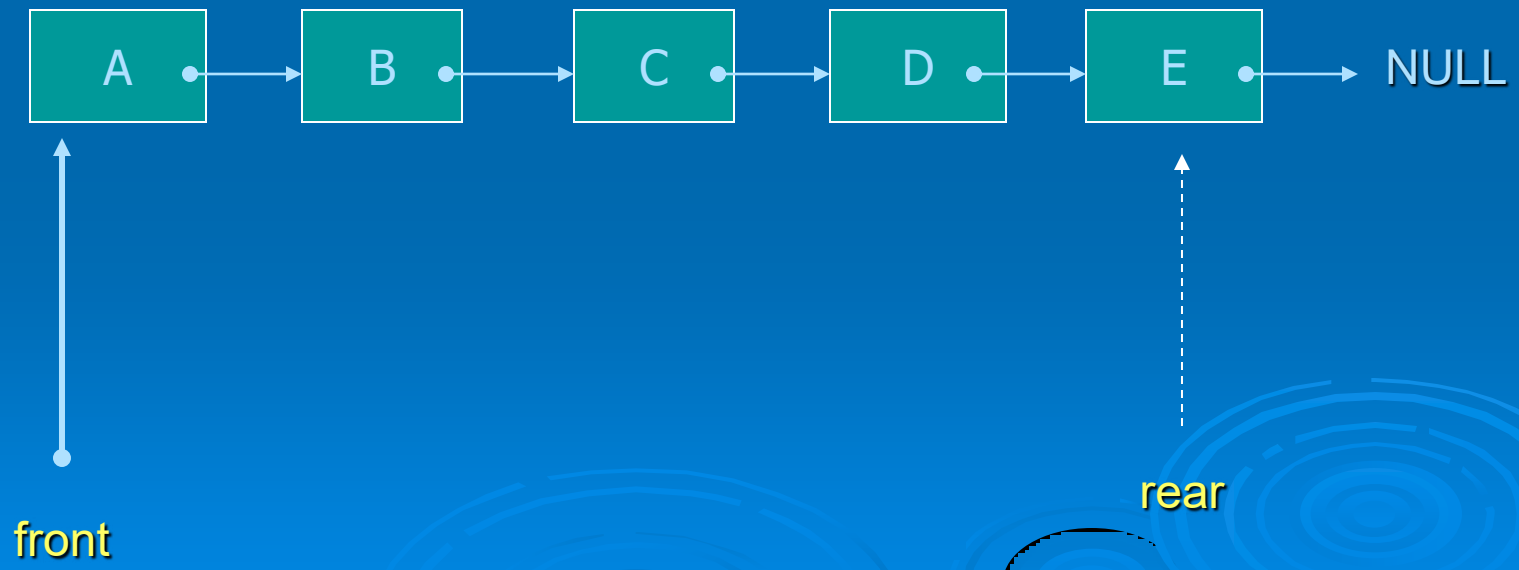
    $O(1)$ as only one operation is required.
  - Pop :

    $O(1)$ as only one operation is required.
  - Is_Empty :

    $O(1)$ as only need to check the top of Stack.

- In a queue, the element deleted from the list is the one which inserted first.

- Queue is also referred as First-in First-out i.e FIFO.



- If an empty queue is deleted, we say queue underflows and
- If rear of the queue exceeds n, the queue overflows.
- Insertion to queue is called Enqueue and deletion form queue is called Dequeue.

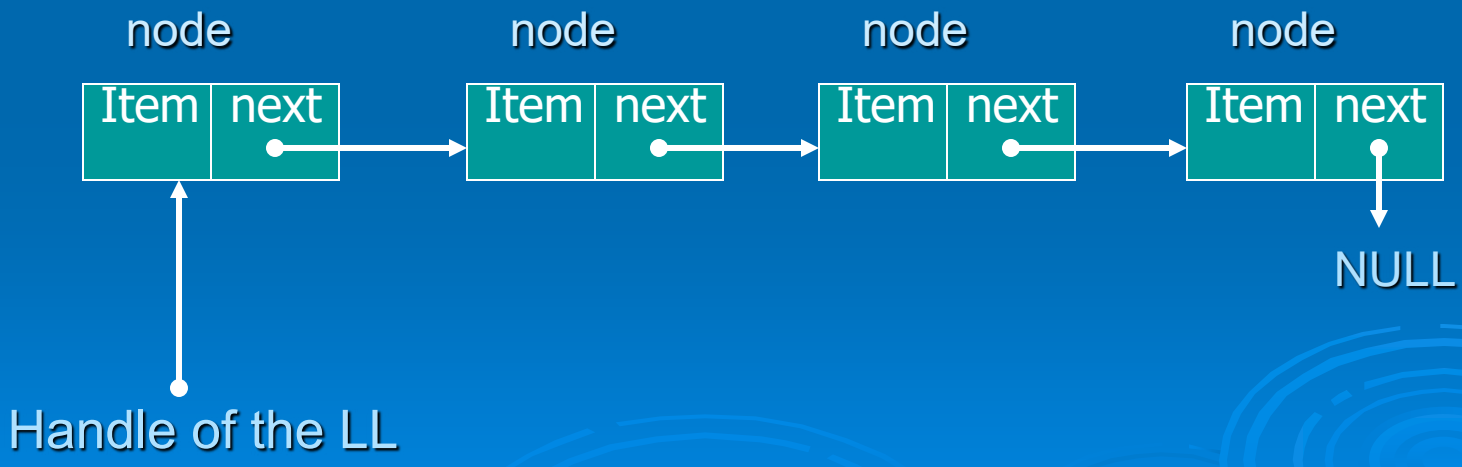
- **Time Complexity:**
  - Enqueue :

    O (1) as only one operation is required.
  - Dequeue :

    O (1) as only one operation is required.

- Link List is a Data Structure in which elements are explicitly ordered, that is each item contains within itself the address of next item.

- Comparison with array Data Structure



Analysis & Operations of Link List

- Adding (Inserting) to a Link List (LL) :

  Time Complexity : $O(1)$ or $\Theta(n)$

- Searching a Link List :

  Time complexity : $\Theta(n)$

- Deletion from Link List :

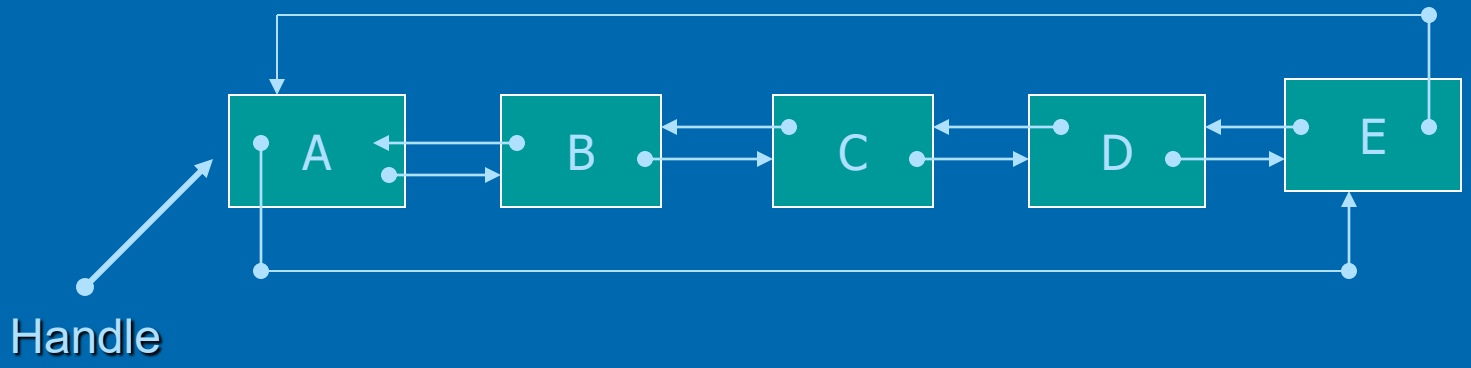
  From the start of link list.

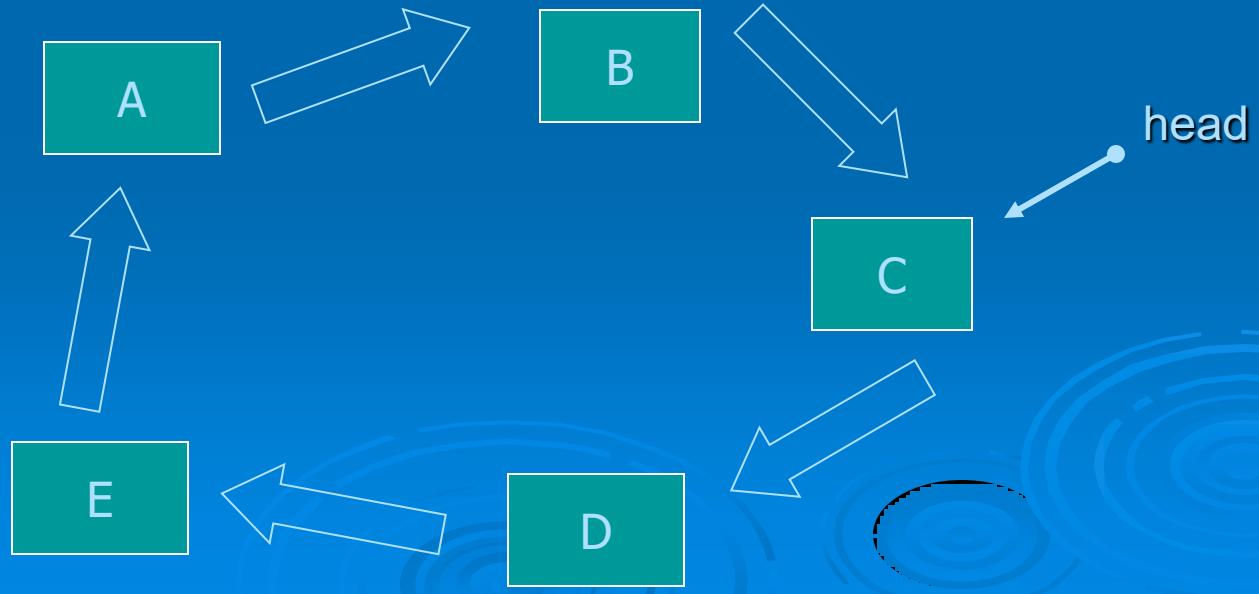
  Time Complexity : $O(1)$

  Deletion of particular node (block) with given key

  Time Complexity : $\Theta(n)$

Analysis of Ring / Doubly Ring .....





- Can *binary search* kind of algorithm be applied on *doubly linked list* to reduce the time of an algorithm.
  - If yes how …
  - If no why …

Hash Table

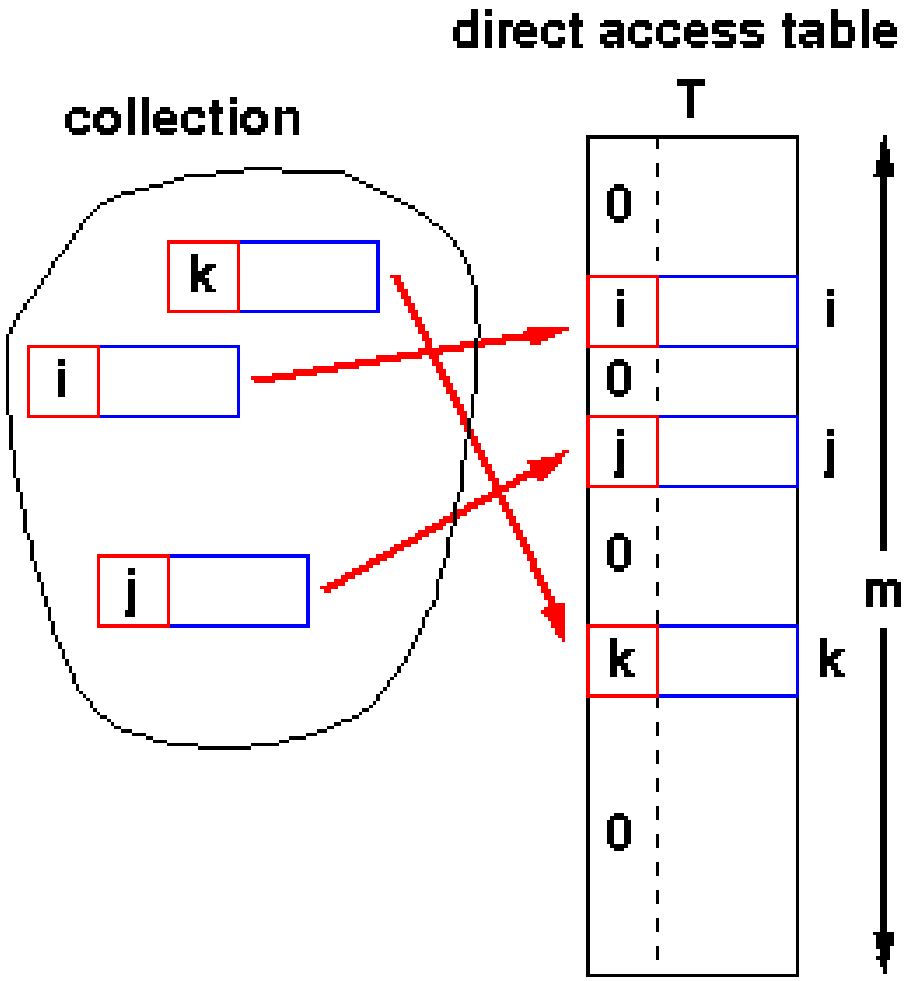
- Given a table *T* and a record *x*, (with key and satellite data), we need to support:
  - Insert (*T*, *x*)
  - Delete (*T*, *x*)
  - Search(*T*, *x*)
- We want these to be fast, but don't care about sorting the records

Direct-address table

- If the keys are drawn from the reasoning small universe $U = \{0, 1, \ldots, m-1\}$ of keys, a solution is to use a Table T[0, . m-1], indexed by keys.

- To represent the dynamic set, we use an array, or direct-address table, denoted by T[0 . . m-1], in which each slot corresponds to a key in the universe.

- Following figure illustrates the approach.



- Each key in the universe U {Collection} corresponds to an index in the table T[0 . . m-1]

- Using this approach, all three basic operations

  Insert (*T*, *x*),
  Delete (*T*, *x*),
  Search(*T*, *x*)

  } (dictionary operations)

  take $\theta(1)$ in the worst case.

Hash Tables

- When the size of the universe is much larger the same approach (direct address table) could still work in principle, but the size of the table would make it impractical. A solution is to *map the keys onto a small range*, using a function called a hash function. The resulting data structure is called *hash table*.
- With direct addressing, an element with key k is stored in slot k. With hashing, this same element is stored in slot h(k); that is we use a hash function h to compute the slot from the key. Hash function maps the universe *U* of keys into the slot of a hash table T[0 . . .m-1] i.e. h: $U \rightarrow \{0, 1, \ldots, m-1\}$

- Typical, hash functions generate "random looking" values.
- For example, the following function usually works well

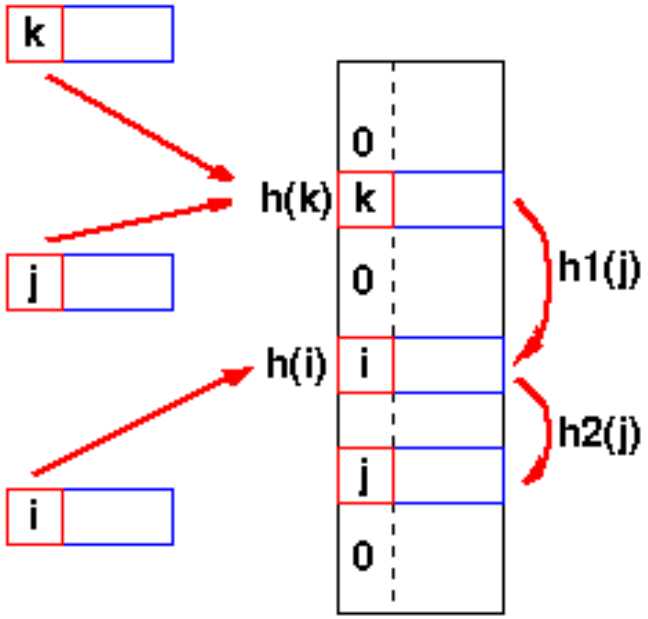
  $h(k) = k \bmod m$

  (where m is a prime number)

- The point of the hash function is to reduce the range of array indices that need to be handled.

Collision

- As keys are inserted in the table, it is possible that two keys may hash to the same table slot.
  If the hash function distributes the elements uniformly over the table, the number of collisions cannot be too large on the average, but it is very likely that there will be at least one collision, even for a lightly loaded table .

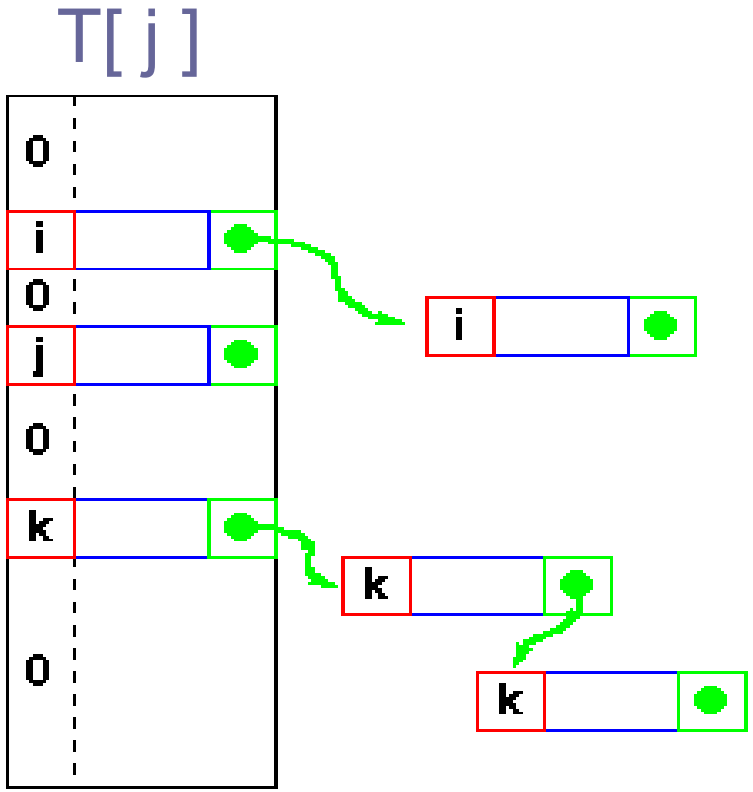


- A hash function h map the keys k and j to the same slot, so they collide .

- There are two basic methods for handling collisions in a hash table:
  - Chaining and
  - Open addressing.

Collision Resolution by **Chaining**

- When there is a collision (keys hash to the same slot), the incoming keys is stored in an overflow area and the corresponding record is appeared at the end of the linked list



- Each slot T[j] contains a linked list of all the keys whose hash value is j. For example, h(k1) = h(kn) and h(k5) = h(k2) = h(k7).
- For Example:
- keys: 5, 28, 19, 15, 20, 33, 12, 17, 10
  slots: 9
  hash function = h(k) = k mod 9

  h(5) = 5 mod 9 = 5
  h(28) = 28 mod 9 = 1
  h(19) = 19 mod 9 = 1
  h(15) = 15 mod 9 = 6
  h(20) = 20 mod 9 = 2
  h(33) = 33 mod 9 = 6
  h(12) = 12mod 9 = 3
  h(17) = 17 mod 9 = 8
  h(10) = 10 mod 9 = 1

- The worst case running time for insertion is O(1), if node is added at the head of list.
- In the worst case behavior of chain-hashing, all n keys hash to the same slot, creating a list of length n. The worst-case time for search is thus θ(n) plus the *time to compute the hash function*.
- Deletion of an element x has the same worst-case time as for search operation if the lists are singly linked.

Assignment # 3

- You have the two sorted lists of N/2 elements each. Find the median N numbers when two lists are combined. Suggest an *efficient algorithm* for solving the problem and give its analysis (Recurrence Relation / Asymptotic notation etc...)

- For example

  List 1 :- 80 85 86 87 100

  List 2 :- 2 60 69 70 86

  Median is :- 80 and 85

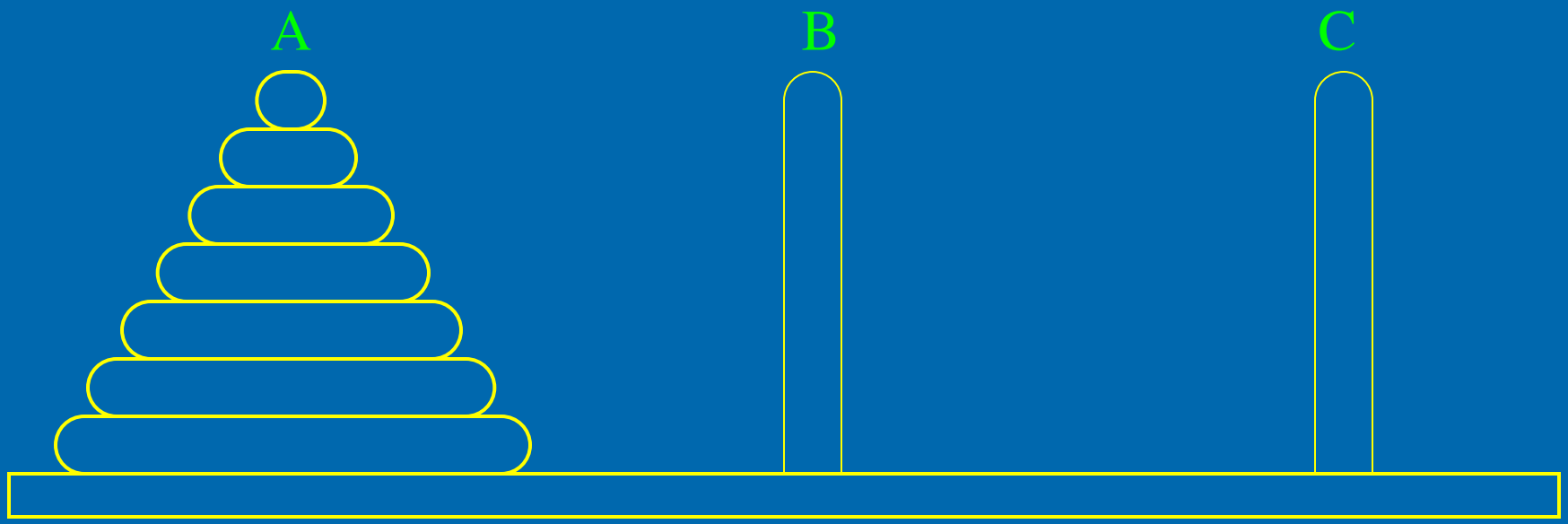
  Due: Next Week……..

Lecture # 6

- Towers of Hanoi
- Sorting Algorithms

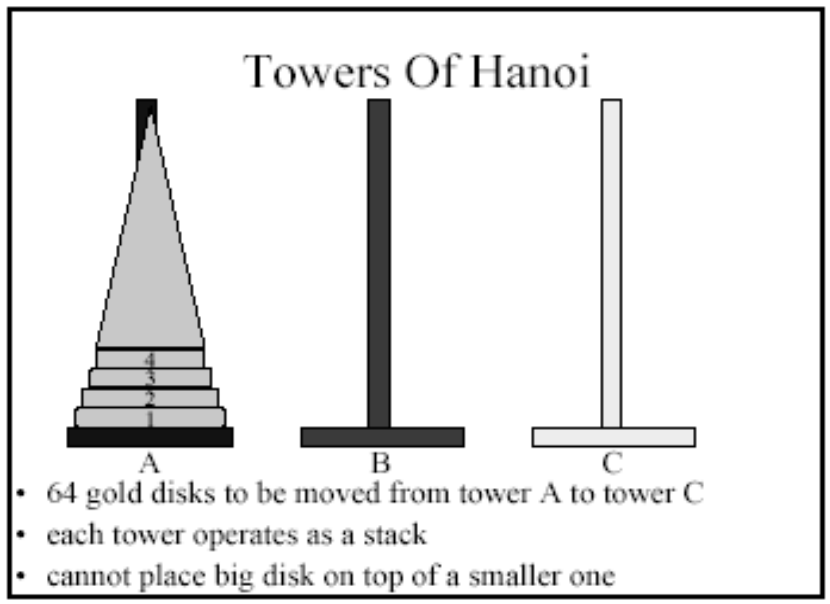
Towers of Hanoi

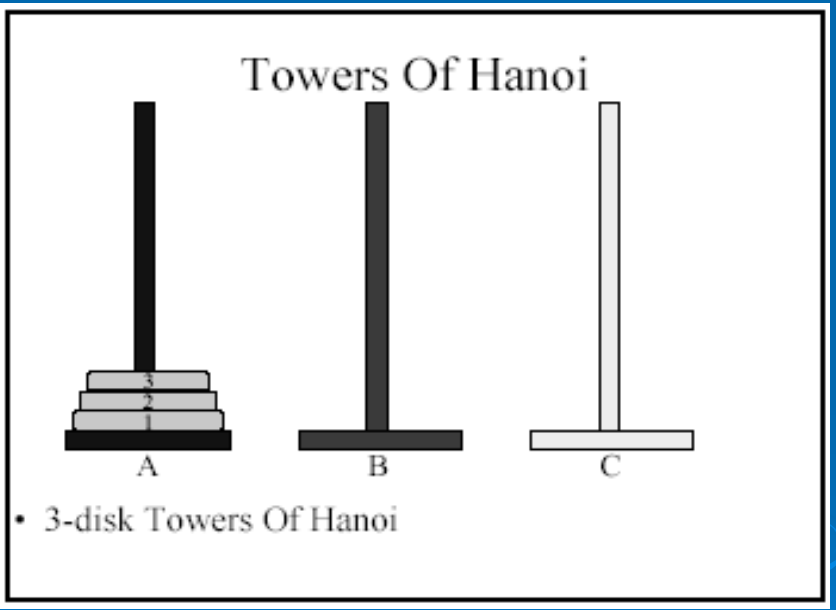
- It provides analysis of recursive algorithm
- If rods = 3 and
  rings = 64 of different sizes
- All rings are arranged in decreasing size on rod 1 (the largest at the bottom and the smallest at the top)
- Transfer all the rings to the 3rd rod
  - Move one ring in one move
  - No ring is even placed on top of another smaller one on any rod
- Time required to finish this work for 64 rings is 500,000 million years if one ring is moved in one second.
  2^{64} = 18,446,744,073,709,551,616 Moves



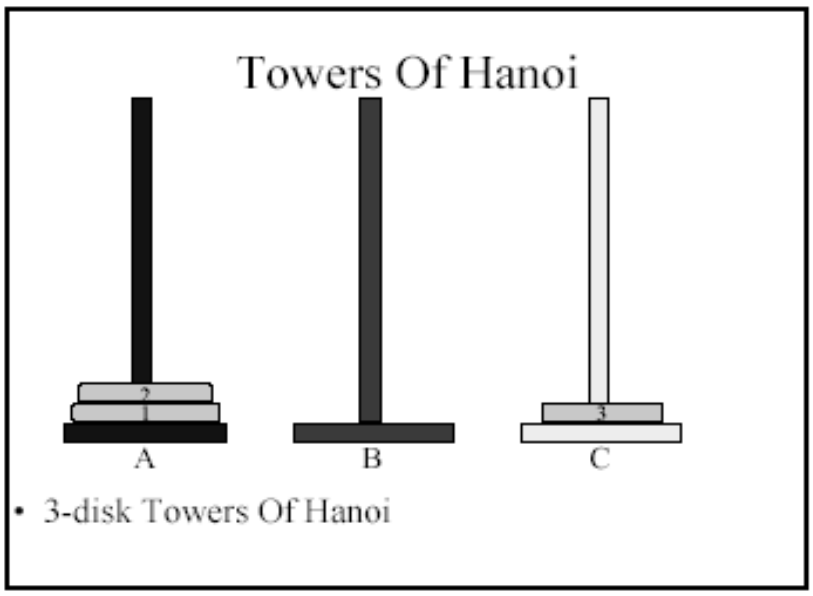
Initial setup of Towers of Hanoi with n = 7

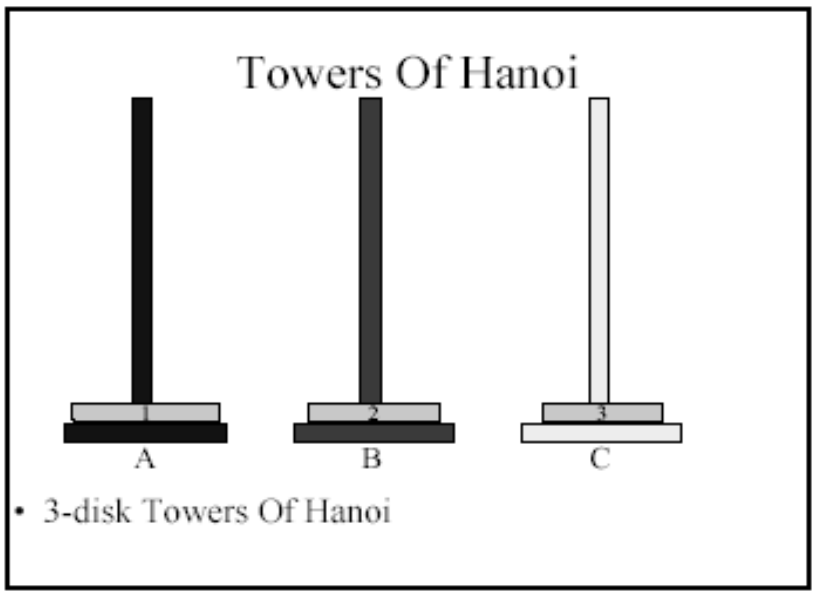
Towers of Hanoi



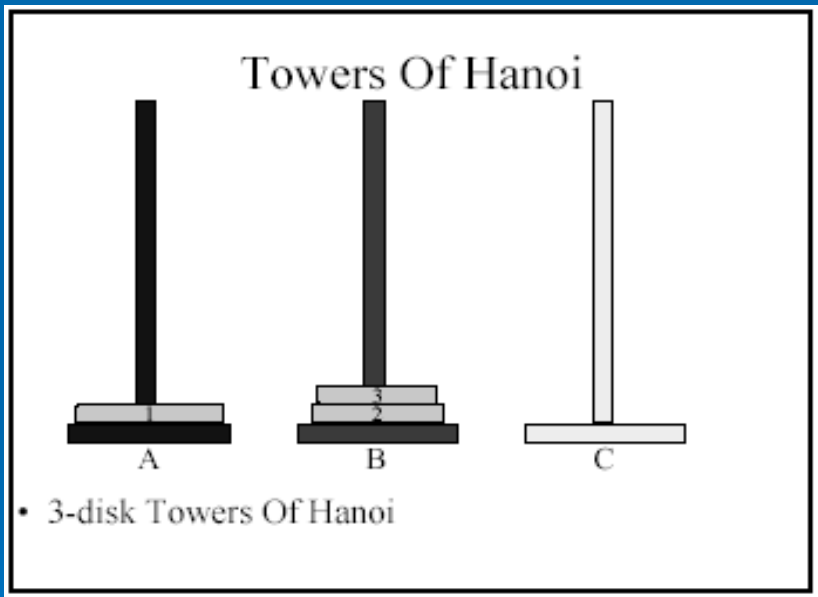


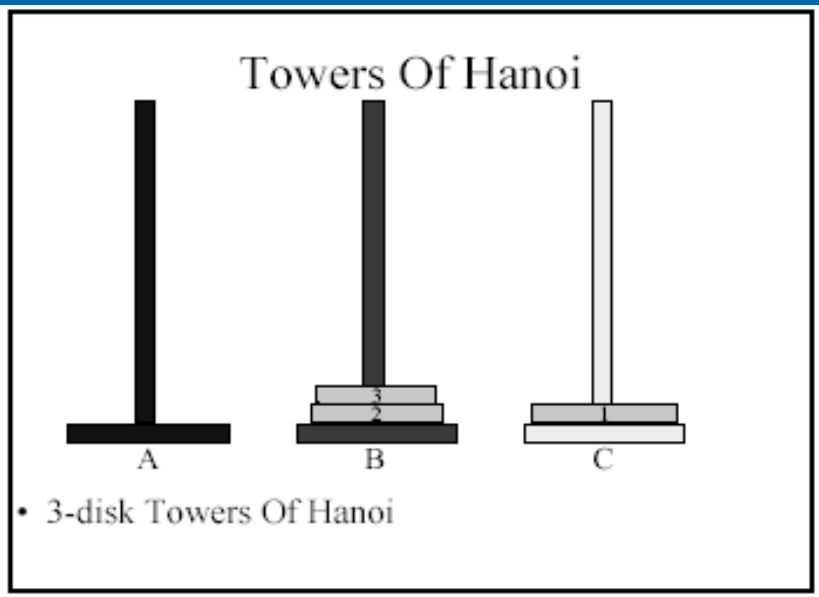
Towers of Hanoi



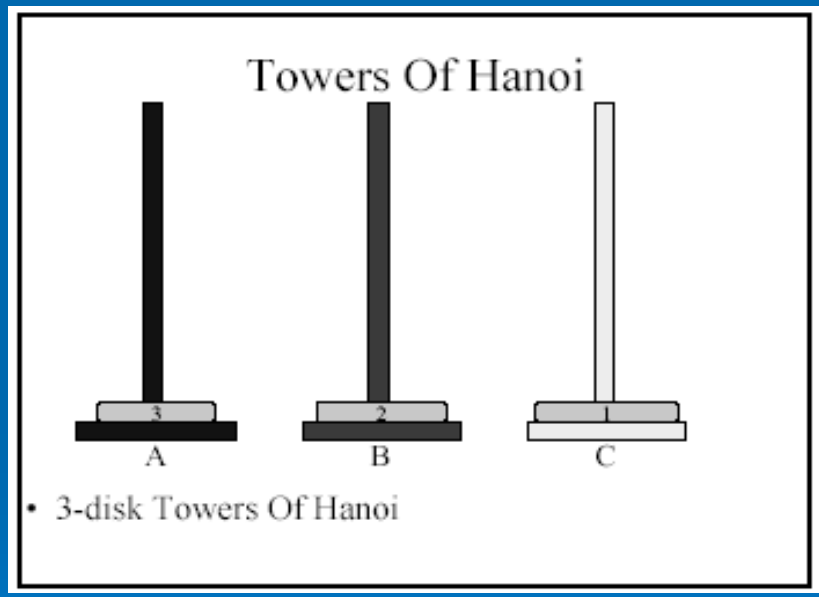


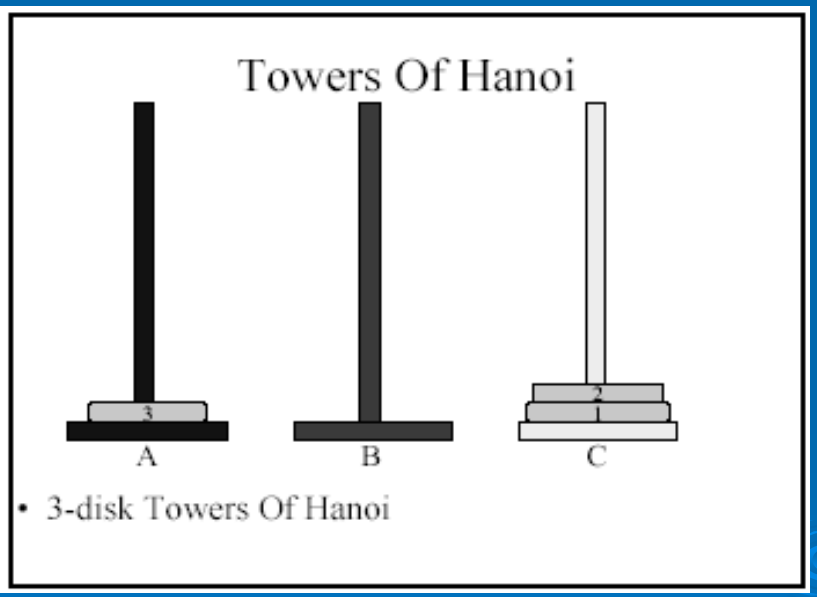
Towers of Hanoi



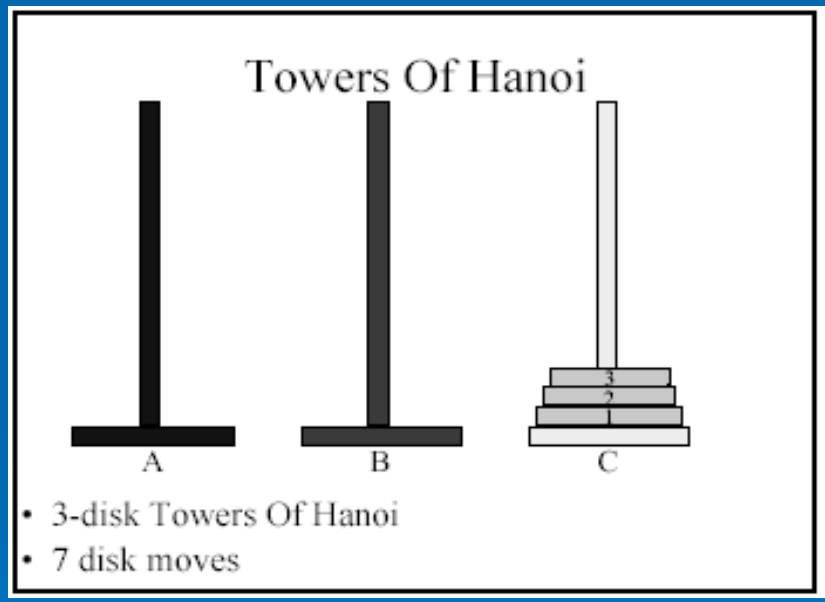


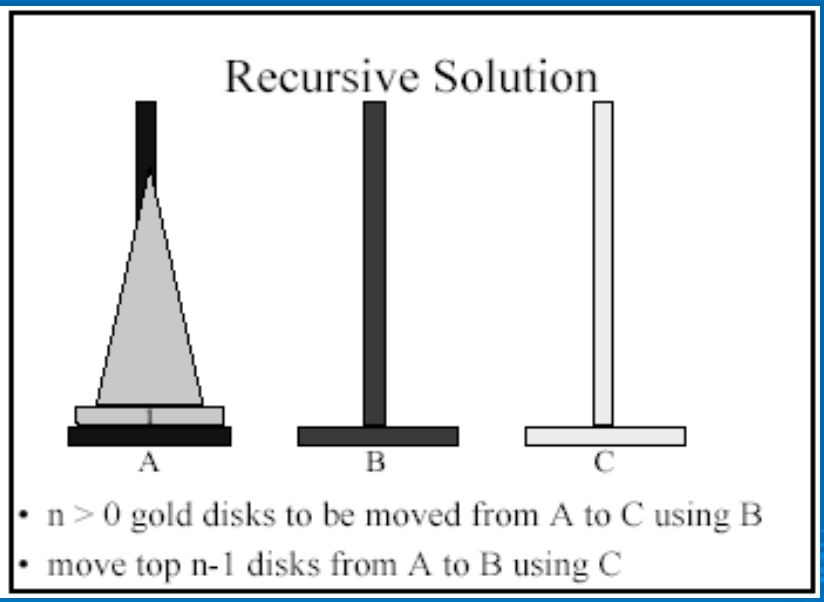
Towers of Hanoi



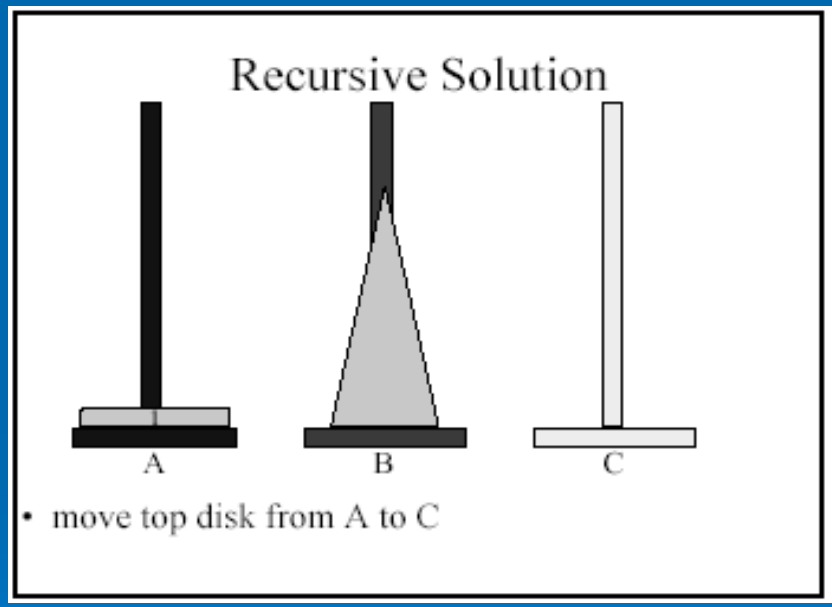


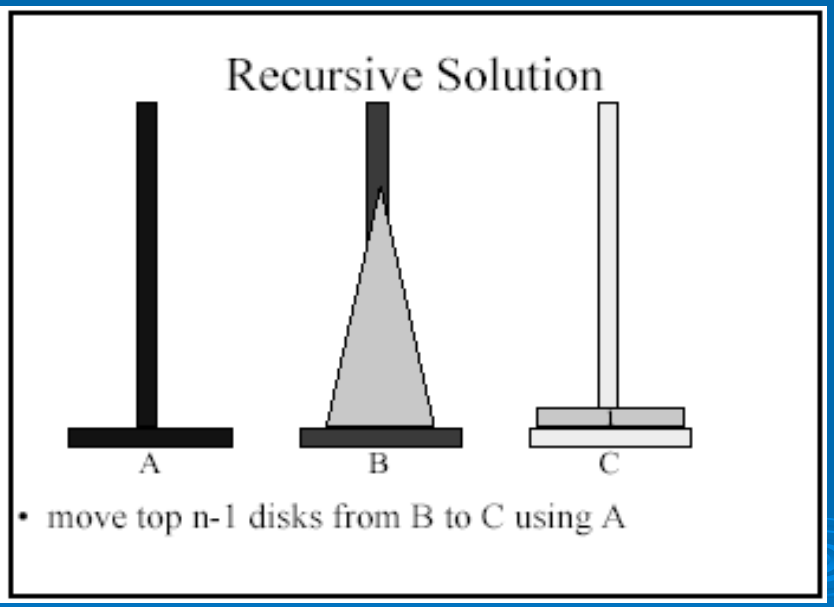
Towers of Hanoi





Towers of Hanoi





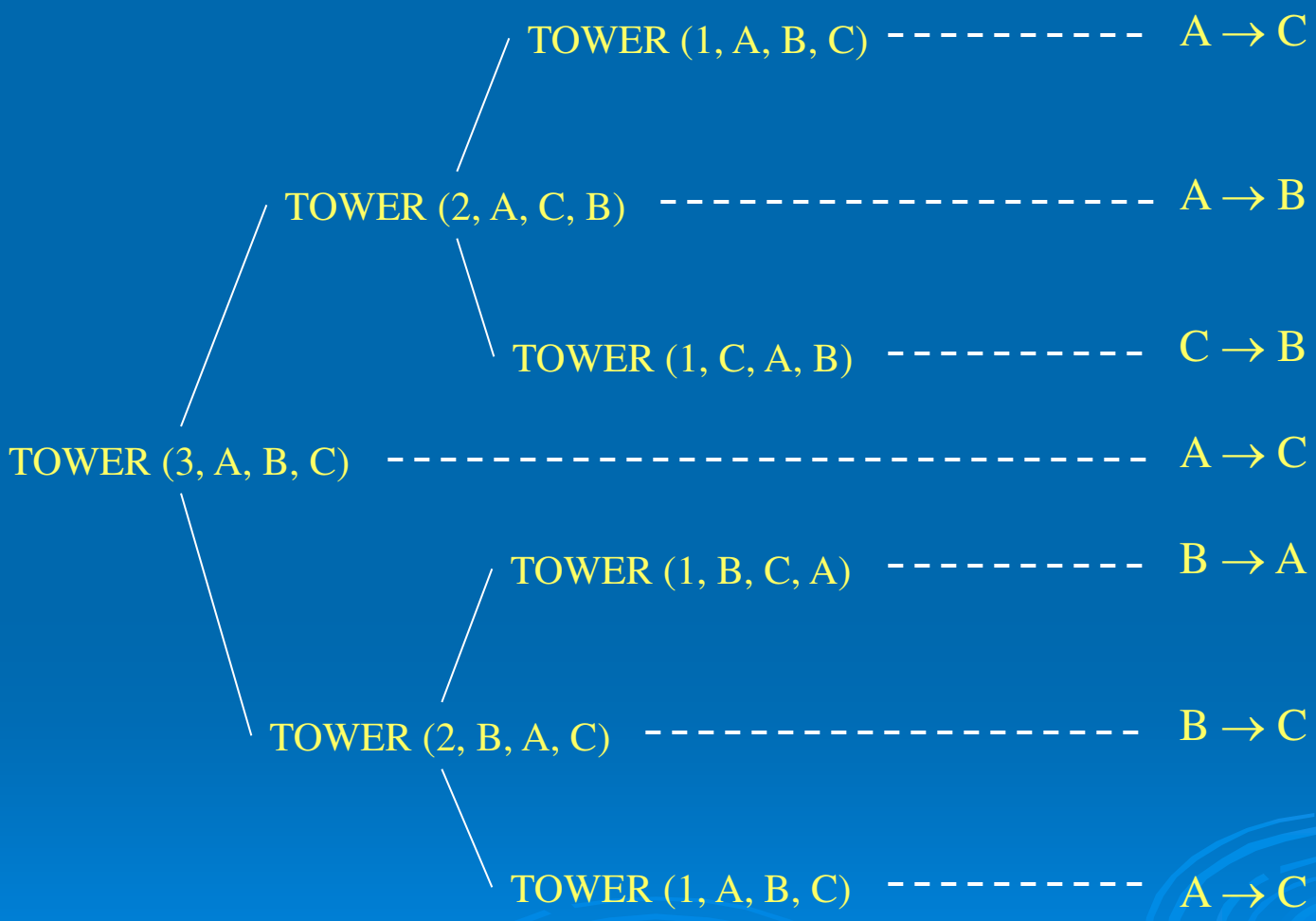
```
TOWER (N, BEG, AUX, END)
If N=1 then BEG "→" END
If N>1 then
        Tower (N-1, BEG, END, AUX)
        write BEG "→" END
        Tower (N-1, AUX, BEG, END)
```

- Let T(N) denote the number of times it is executed on a call of N.

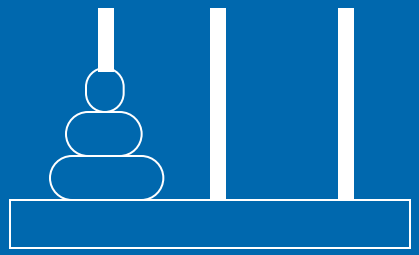
$$T(N) = \begin{cases} 0 & N = 0 \\ 2T(N-1) + \Theta(1) & N > 0 \end{cases}$$

- Number of moves are: $2^N - 1$
- Time to solve the problem with N rings = $\theta(2^N)$

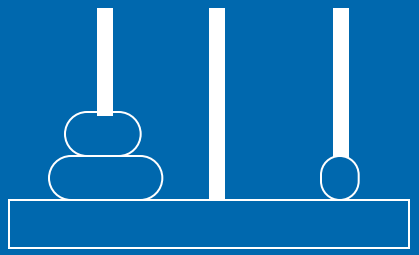
$n = 3$


TOWER (3, A, B, C)
TOWER (2, A, C, B)
TOWER (1, A, B, C)
A → C
TOWER (1, C, A, B)
C → B
A → B
A → C
TOWER (2, B, A, C)
TOWER (1, B, C, A)
B → A
B → C
TOWER (1, A, B, C)
A → C

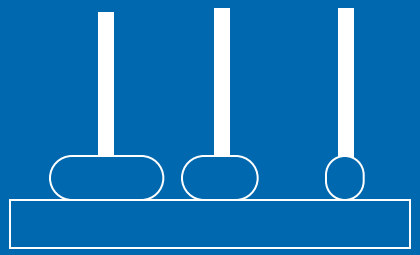
$n = 3$



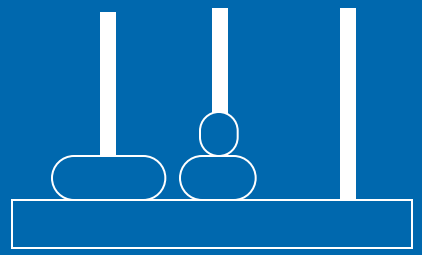
(a) Initial



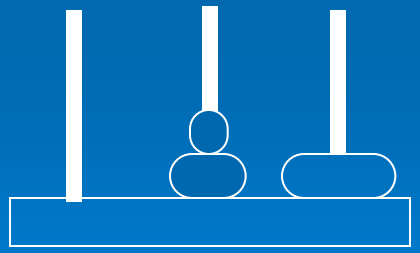
(1) A → C



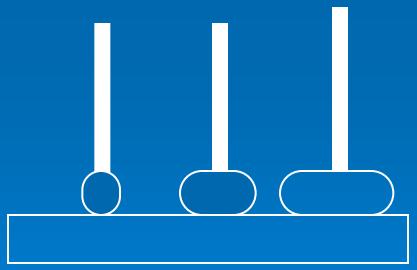
(2) A → B



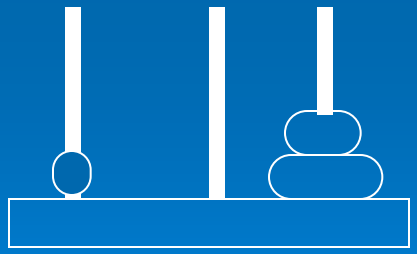
(3) C → B



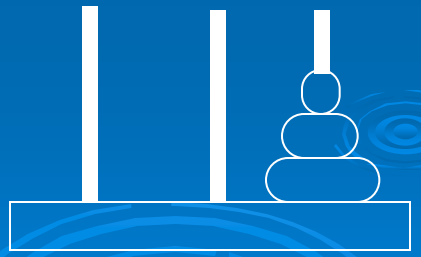
(4) A → C



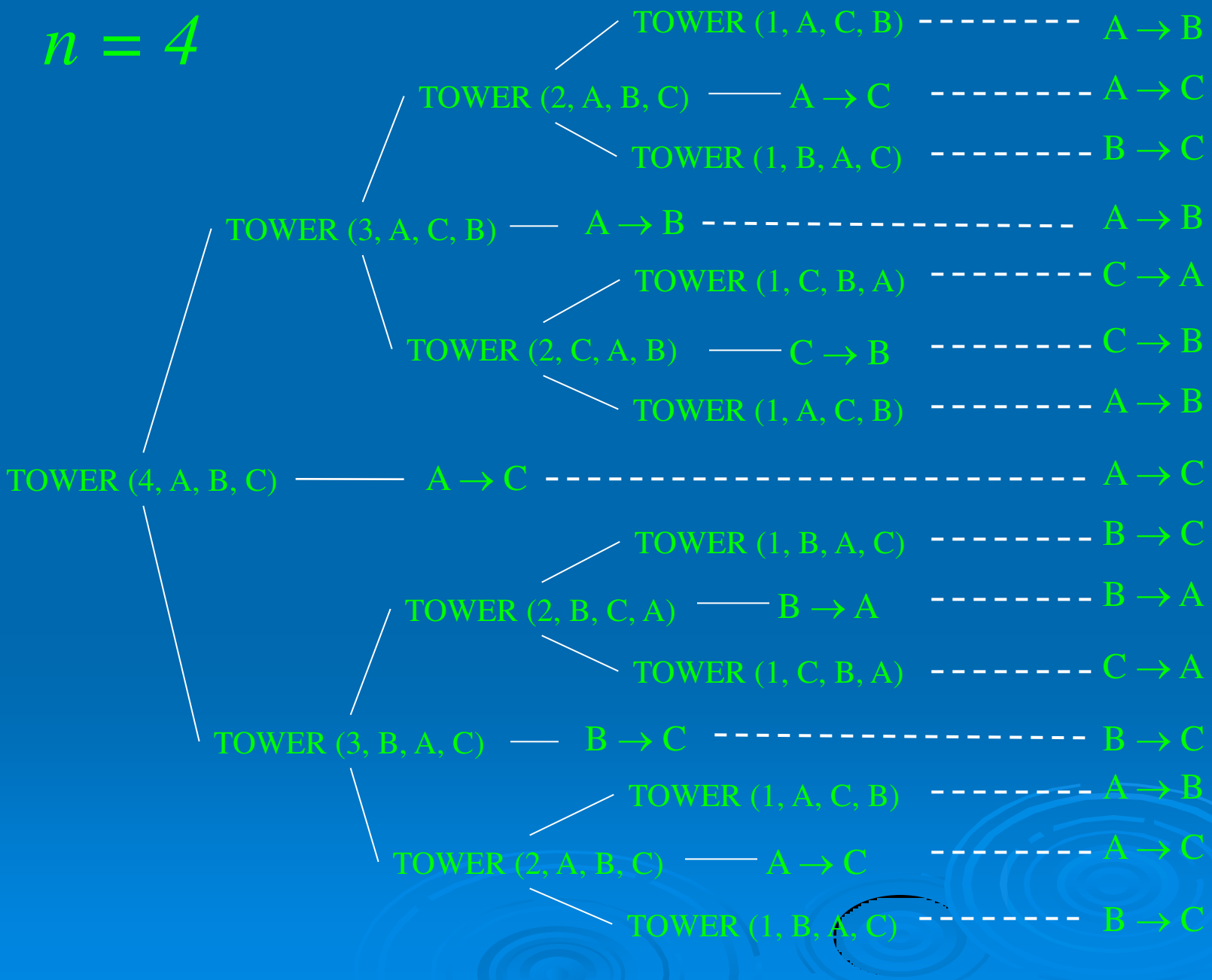
(5) B → A



(6) B → C



(3) A → C


n = 4
TOWER (4, A, B, C)
A → C
A → C
TOWER (3, A, C, B)
A → B
A → B
TOWER (2, A, B, C)
A → C
A → C
TOWER (1, A, C, B)
A → B
TOWER (1, B, A, C)
B → C
TOWER (2, C, A, B)
C → B
C → B
TOWER (1, C, B, A)
C → A
TOWER (1, A, C, B)
A → B
TOWER (3, B, A, C)
B → C
B → C
TOWER (2, B, C, A)
B → A
B → A
TOWER (1, B, A, C)
B → C
TOWER (1, C, B, A)
C → A
TOWER (2, A, B, C)
A → C
A → C
TOWER (1, A, C, B)
A → B
TOWER (1, B, A, C)
B → C

Non Credit Assignment

- Implement the problem of *Towers of Hanoi* in any language of your choice.
- Also view that increasing N how much your system becomes slow down.

Selection Sort

- The basic idea of Selection Sort is to make a number of passes through the list or part of the list and, on each pass, select one element to be correctly positioned.
- For example, on each pass through a sub list, the smallest element in the sub list might be found and then moved to its proper location.

- As an illustration, suppose that following list has to be sorted into ascending order :

67 33 21 84 49 50 75

- We scan the list to locate the smallest element and find it in position 3:
- Then we interchange this element with the first element and thus properly position the smallest element at the beginning of the list.

21 33 67 84 49 50 75

- We now scan the sub list consisting of the elements from position 2 on to find the next smallest element

  21 33 67 84 49 50 75

  and exchange it with the second element (itself in this case) and then properly position the next-to-smallest element in position 2

- We continue in this fashion, locating the smallest element in the sub list of elements from position 3 and interchanging it with the third element, then properly positioning the smallest element in the sub list of elements from position 4 on, and so on until we eventually do this for the sub list consisting of the last two elements as follows.

21 33 49 84 67 50 75

21 33 49 50 67 84 75

21 33 49 50 67 84 75

21 33 49 50 67 75 84

- Positioning the smallest element in this last sub list obviously also positions the last element correctly and thus completes the sort.

Analysis of Selection Sort

- Looking to the above algorithm of selection sort we observe that on the first pass through the list, the first item is compared with each of n-1 elements that follow it.
- On the second pass, second element is compared with the n-2 elements following it and so on. So a total of

  (n-1) + (n-2) + (n-3) + ….. + 2 + 1 = n(n-1)/2

  comparisons are required for any list.

and

$(n-1) + (n-2) + (n-3) + \ldots\ldots + 2 + 1 = n(n-1)/2$

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

$= n^2/2 - n/2$

$= 0.5n^2 - 0.5n$

So ignoring smaller terms and coefficients we conclude that the worst case computing time for selection sort is $O(n^2)$.

Exchange Sort (Bubble Sort)

- Exchange sort systematically interchange pairs of elements that are out of order until eventually no such pairs remain and the list is therefore sorted.

- To illustrate bubble sort, consider the following list

67 33 21 84 49 50 75

- On the first pass, we compare the first two elements, 67 and 33 and interchange them because they are out of order.

67 33 21 84 49 50 75

33 67 21 84 49 50 75

- Now we compare the second and third elements, 67 and 21 and interchange them

33 21 67 84 49 50 75

- Next we compare 67 and 84 but do not interchange them because they are already in the correct order.

  33 21 67 84 49 50 75

- Next 84 and 49 are compared and interchanged.

  33 21 67 49 84 50 75

- Then 84 and 50 are compared and interchanged.

  33 21 67 49 50 84 75

- Finally 84 and 75 are compared and interchanged.

  33 21 67 49 50 75 84

So the first pass through the list is now complete.

- We are guaranteed that on this pass, the largest element in the list will go down to the end of the list, since it will obviously be moved and pass all smaller elements.
- But notice that some of smaller items have "Bubbled Up" toward their proper positions nearer the front of the list.
- So we scan the list again but this time we ignore the last item because it is already in its proper location.

33 21 67 49 50 75 **84**

So in each pass the last element will be placed in its proper location in the sub list.

- The *worst case* of bubble sort occurs when the list elements are in reverse order because in this case only one item (the largest) item is placed correctly on each pass through the list.

- On the first pass through the list n-1 comparisons and interchanges are made and only the largest element is correctly positioned.

- On the next pass, sub list consisting of the first n-1 elements are scanned; there are n-2 comparisons and interchanges; and the next largest element sink to position n-1.

- Finally, process continues ***until*** the sub list consisting of the first two elements is scanned and on this pass there is one comparison & interchange.
- Thus a total of

  $(n-1) + (n-2) + (n-3) + \ldots\ldots + 1 = n(n-1)/2$

  $= n^2/2 - n/2$

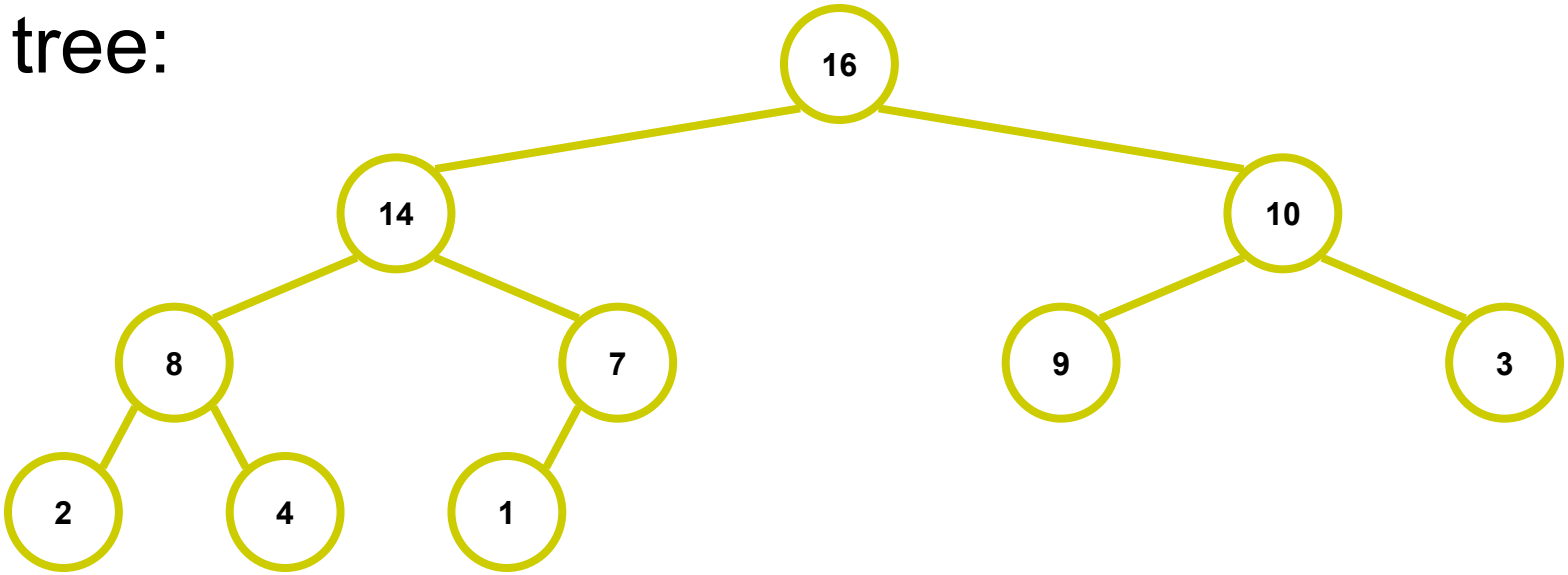
  Comparisons and interchanges are required. It follows that worst case computing time for Bubble Sort is $\boldsymbol{O(n^2)}$

Lecture # 7

- Binary Heaps
- Heap Sort

Heaps

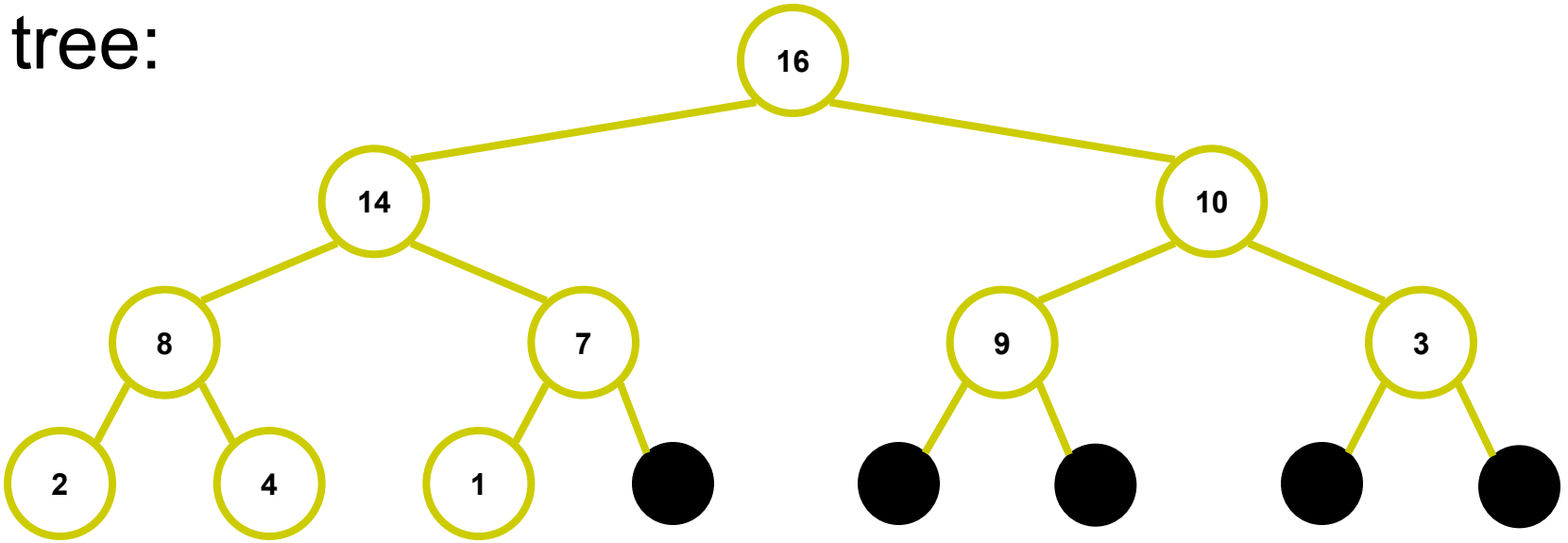
- A *heap* can be seen as a complete binary tree:



  - *What makes a binary tree complete?*
  - *Is the example above complete?*

Heaps

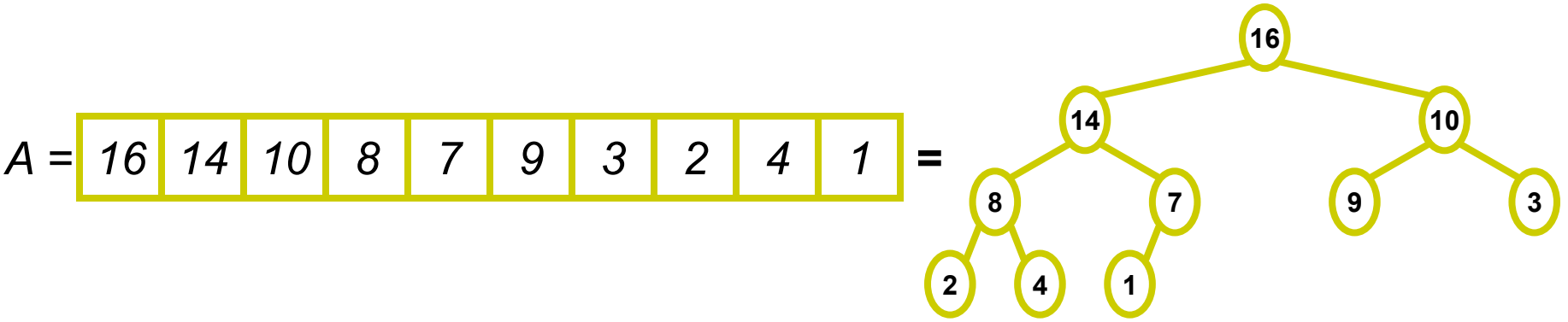
- A *heap* can be seen as a complete binary tree:



  - Some books calls them "***nearly complete***" binary trees; can think of unfilled slots as null pointers

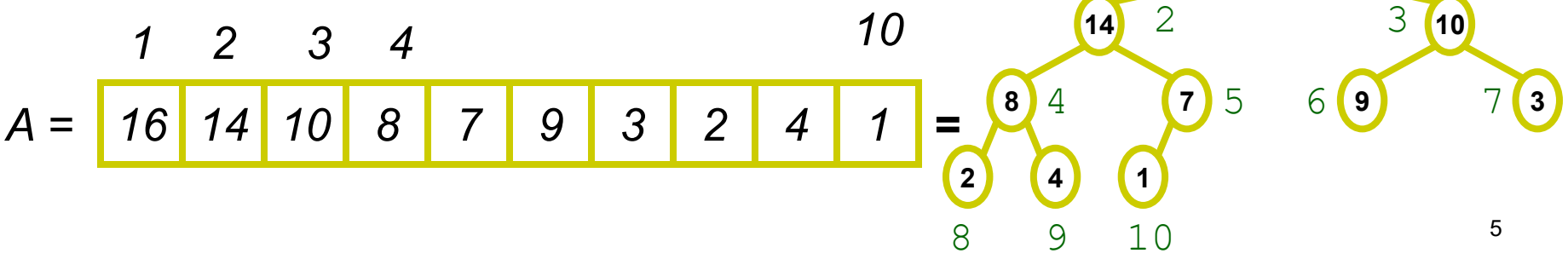
Heaps

- In practice, heaps are usually implemented as arrays:



Heaps

- To represent a complete binary tree as an array:
  - The root node is A[1]
  - Node *i* is A[i]
  - The parent of node *i* is A[*i*/2] (note: integer divide)
  - The left child of node *i* is A[2*i*]
  - The right child of node *i* is A[2*i* + 1]



Referencing Heap Elements

- So…

  Parent(i) { return $\lfloor i/2 \rfloor$; }
  Left(i) { return 2*i; }
  right(i) { return 2*i + 1; }

The *max-Heap* Property

- max-*heap property says*:

  A[*Parent*(*i*)] ≥ A[*i*] for all nodes *i* > 1

  - In other words, the value of a node is at most the value of its parent
  - Where is the largest element in a heap stored?
- Definitions:
  - The *height* of a node in the tree = the number of edges on the longest downward path to a leaf
  - The height of a tree = the height of its root

The *min-Heap* Property

- min-*heap property says*:

  A[*Parent*(i)] $\leq$ A[i] for all nodes *i* > 1

  – Where is the smallest element in a heap stored?

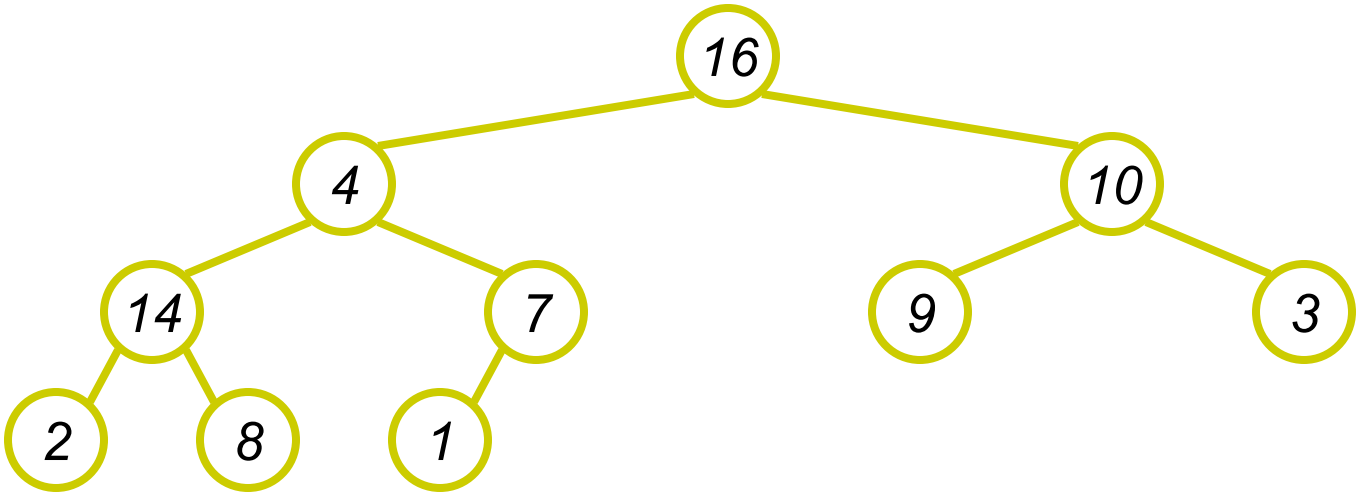
Heap Height

- What is the height of an n-element heap? Why?
- Basic heap operations take at most time proportional to the height of the heap

Heap Operations: Max-Heapify(A,i)

- Max-Heapify(): maintain the heap property
  - Its input is an Array A and an index i of array
  - Left and right sub trees of node i are assumed to be max-heaps.
  - Problem: The sub tree rooted at i may violate the heap property (*How?*)
  - Action: let the value of the parent node "float down" so sub tree at i satisfies the heap property

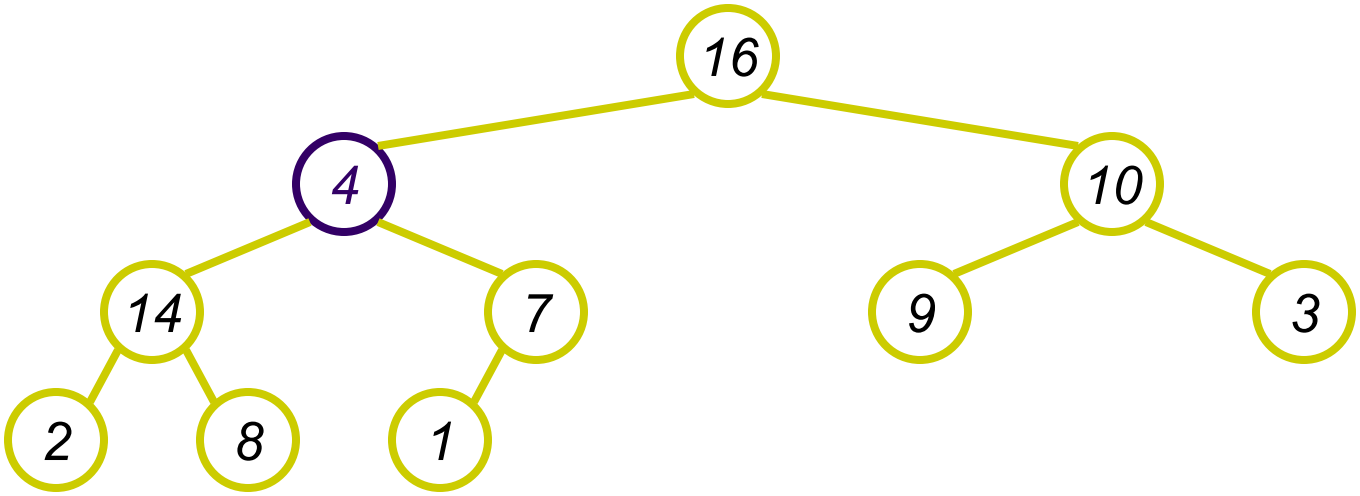
Max-Heapify( ) Example



$A =$

16	4	10	14	7	9	3	2	8	1

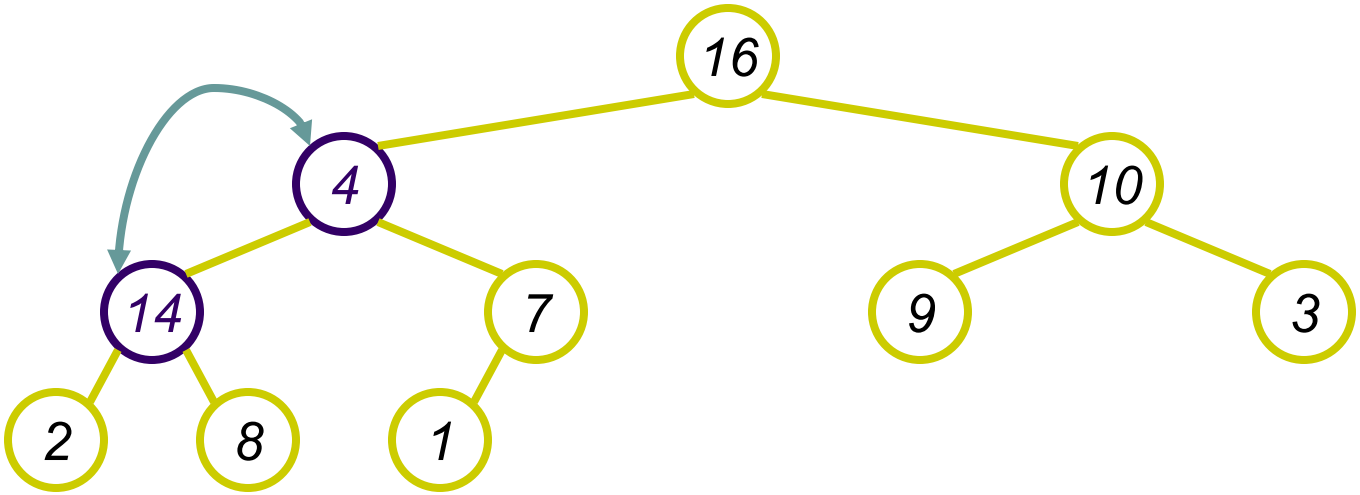
Heapify() Example

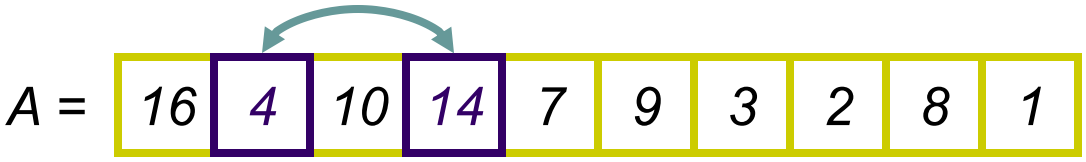


A =

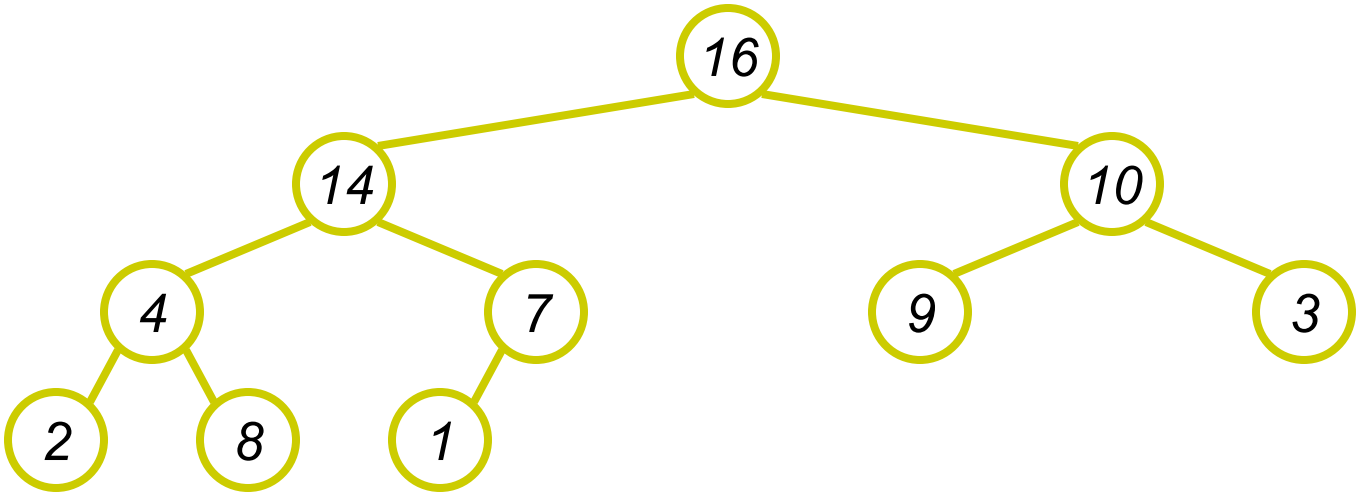
16	4	10	14	7	9	3	2	8	1

Heapify() Example





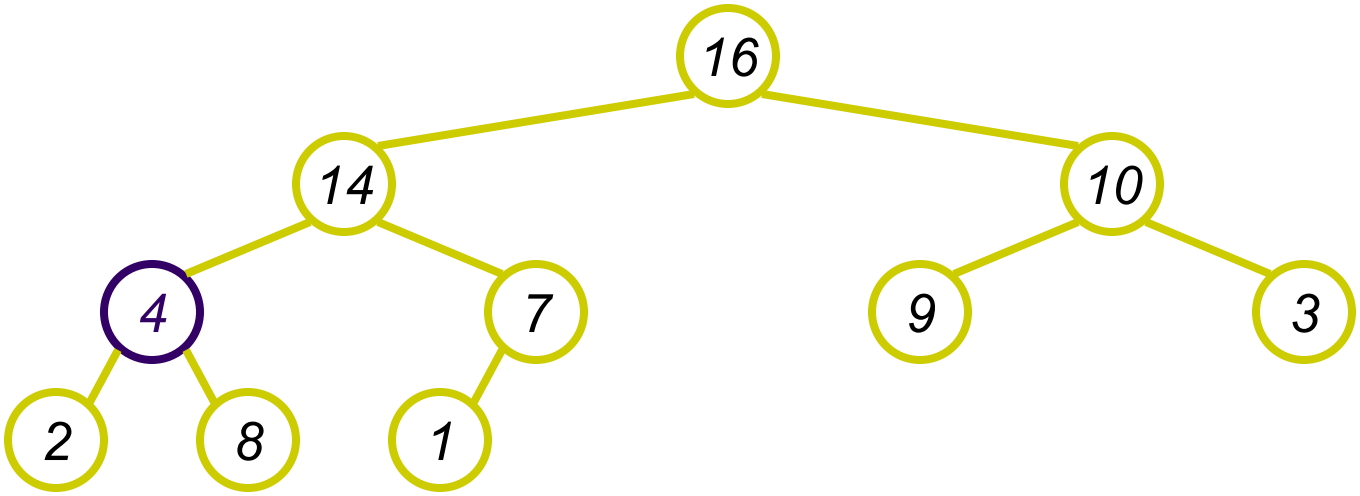
Heapify() Example



$A =$

16	14	10	4	7	9	3	2	8	1

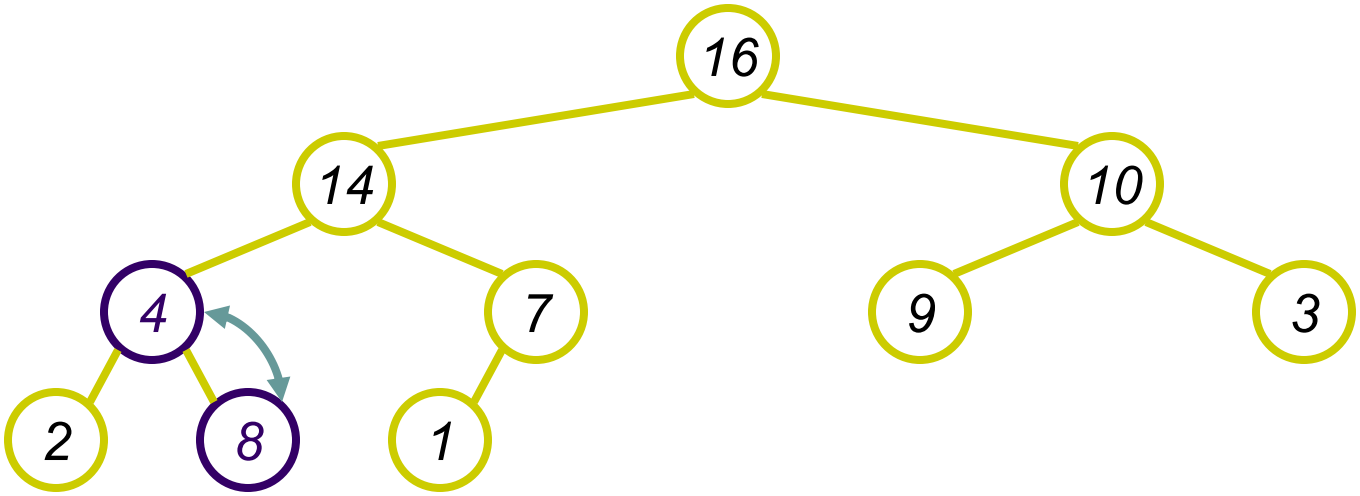
Heapify() Example

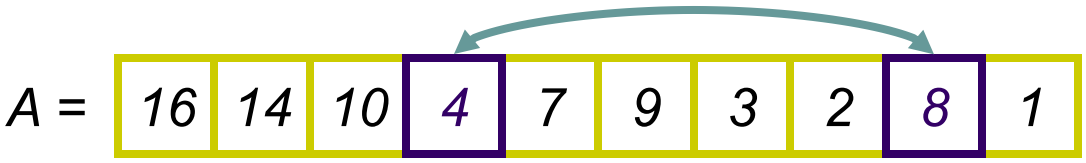


$A =$

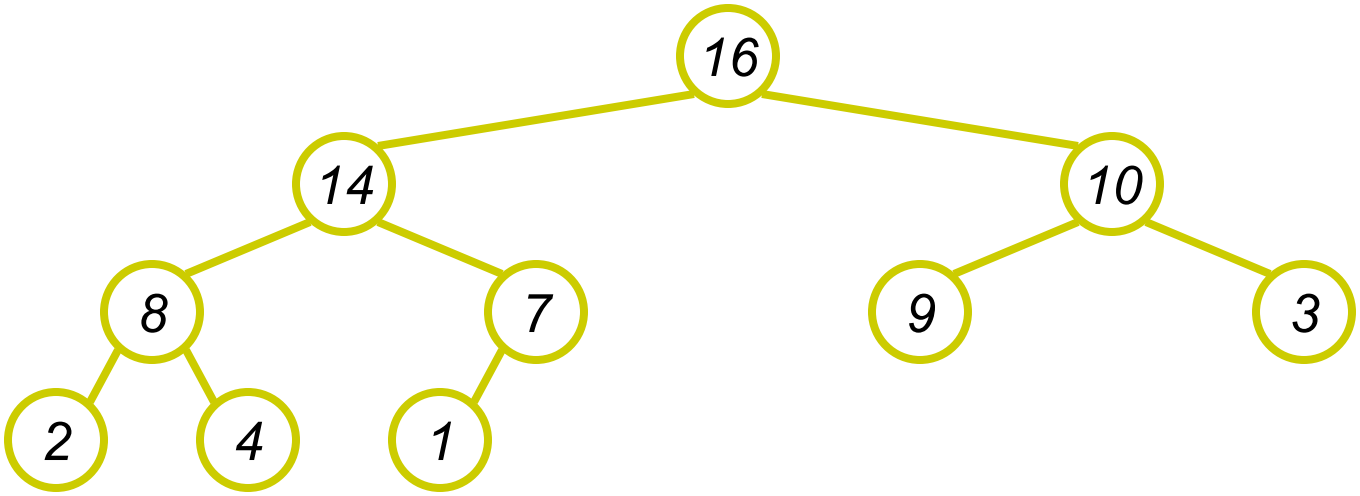
16	14	10	4	7	9	3	2	8	1

Heapify() Example





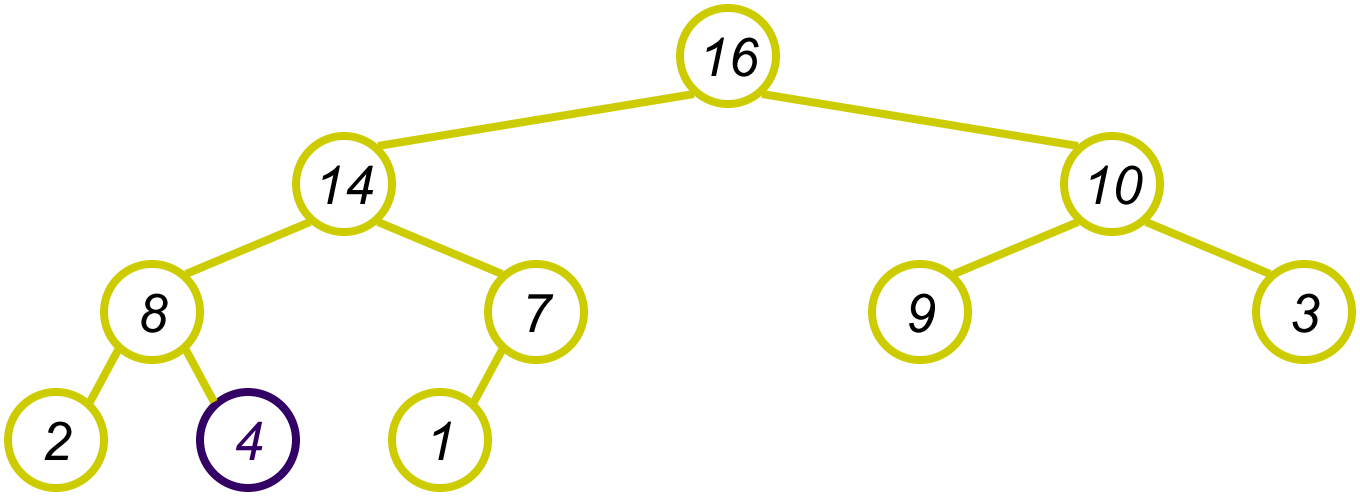
Heapify() Example



$A =$

16	14	10	8	7	9	3	2	4	1

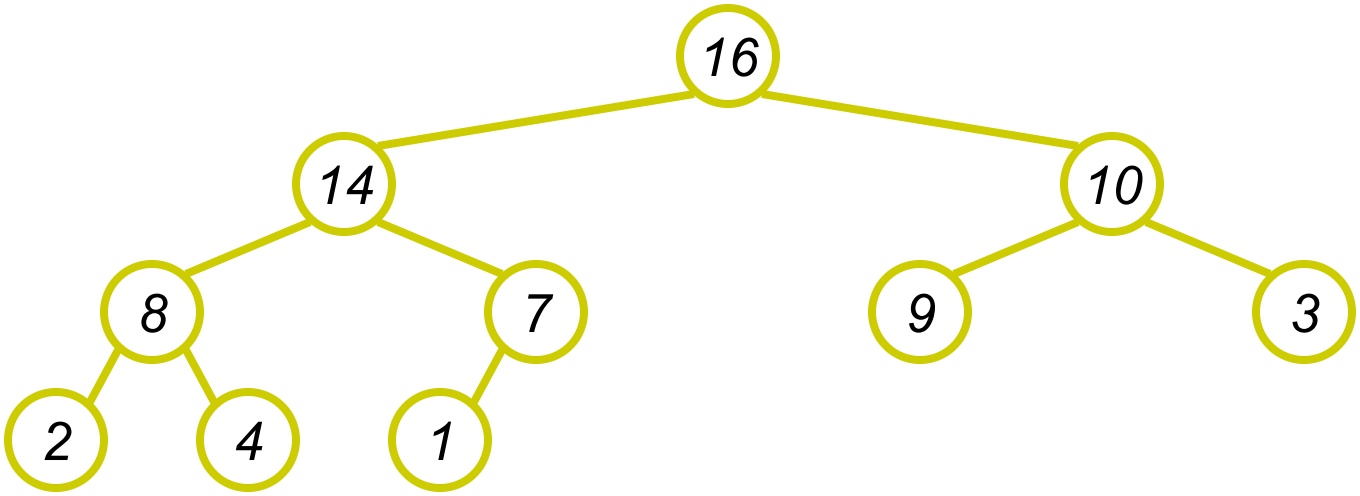
Heapify() Example



$A =$

16	14	10	8	7	9	3	2	4	1

Heapify() Example



$A =$

16	14	10	8	7	9	3	2	4	1

Analyzing Heapify( )

- The number of nodes in a complete binary tree of height h is

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

- So h in terms of n is:

$$h = (\log(n + 1)) - 1 \approx \log n \in \Theta(\log n)$$

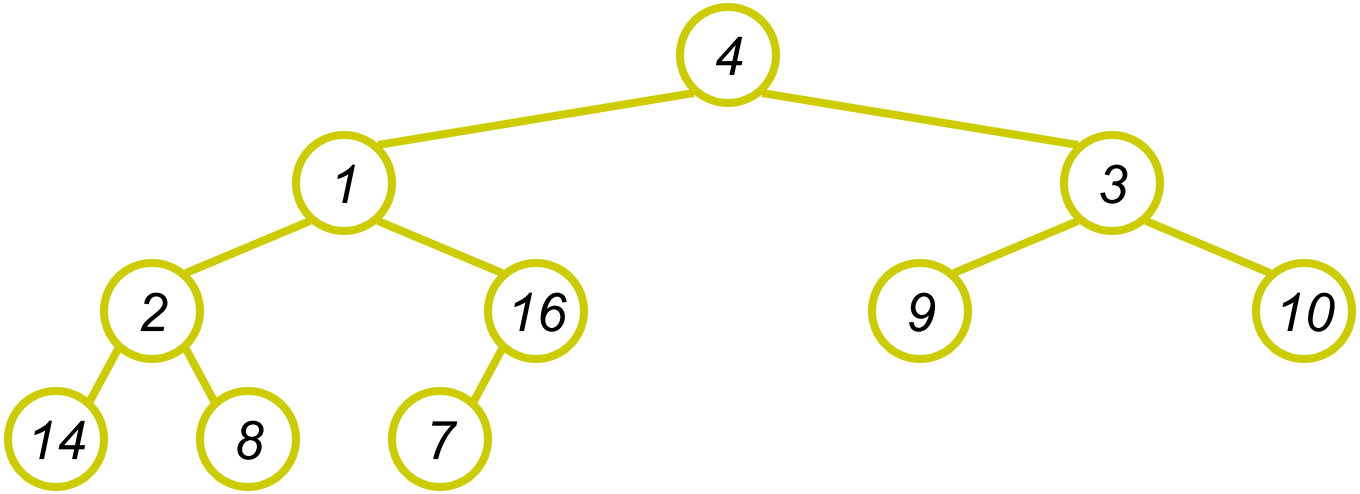
- Running time of Max-Heapify( ) on a node of height h will be O(lg *n*) or O(h).

Heap Operations: BuildHeap()

- We can build a heap in a bottom-up manner by running **`Heapify()`** on successive subarrays
  - Fact: for array of length *n*, all elements in range A[$\lfloor n/2 \rfloor + 1$ .. n] are heaps (*Why?*)
  - Ans: Because sub array A[$\lfloor n/2 \rfloor + 1$ .. n] are all leaves.
  - So:
    - Walk backwards through the array from n/2 to 1, calling **`Heapify()`** on each node.
    - Order of processing guarantees that the children of node *i* are heaps when *i* is processed

BuildHeap() Example

- Work through example
  A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}



BuildHeap( )

- For array of length *n*, all elements in range A[$\lfloor n/2 \rfloor$ + 1 .. n] are leaves of the tree.

- The procedure BuildHeap( ) goes through the remaining each and every node of the tree and apply Heapify( ) on every node so that the whole tree is in the form of heap, having all heap properties.

Analyzing BuildHeap( )

- Each call to Max-Heapify() costs O(lg *n*) or O(h) time, and
- There would be total of O(*n*) such calls
  (specifically, $\lfloor n/2 \rfloor$) for complete set of nodes in a heap.

- Thus the running time is O(n.lg *n*). This upper bound is correct but not *asymptotically tight*.

Analyzing BuildHeap(): Tight bound

- The time required by max-heapify() when called on single node of height h is O(h).

- Fact: an *n*-element heap has height $\lfloor \lg n \rfloor$ and at most $\lceil n/2^{h+1} \rceil$ nodes of any height h.

  (leaves are at height 0)

- We have the formula:

$$\sum_{k=0}^{\infty} k.x^k = x / (1-x)^2 \text{ for } |x| < 1$$

Put $x = ½ < 1$

$$\sum_{k=0}^{\infty} k/2^k = 2$$

- So we can express the total cost of BuildHeap() as follows.

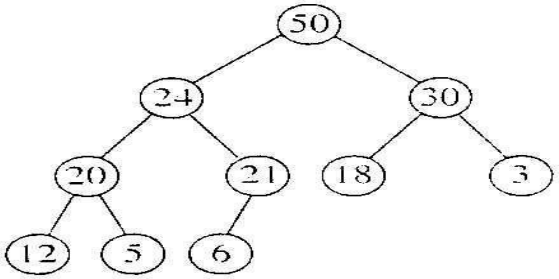
$$\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil . O(h) = O\left(n . \sum_{h=0}^{\lfloor \lg n \rfloor} \lceil h/2^{h} \rceil \right)$$

As $\sum_{h=0}^{\infty} h/2^{h} = 2$

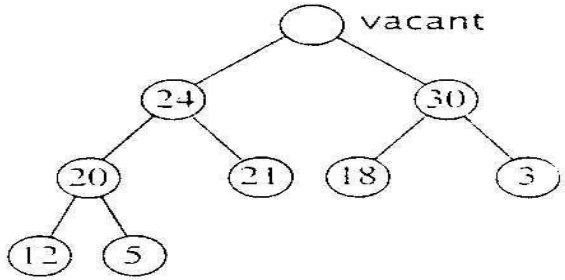
- Thus running time of BuildHeap() can be O(n).

Heapsort

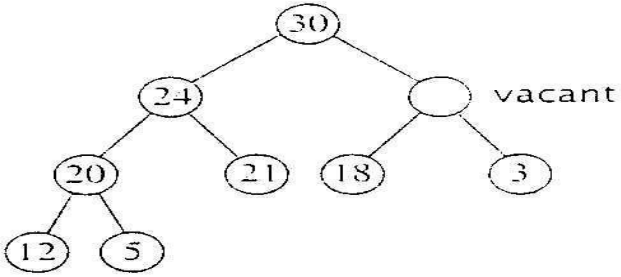
- Given `BuildHeap()`, an in-place sorting algorithm is easily constructed:
  - Maximum element is at A[1]
  - Discard by swapping with element at A[n]
    - Decrement heap_size[A]
    - A[n] now contains correct value
  - Restore heap property at A[1] by calling `Heapify()`
  - Repeat, always swapping A[1] for A[heap_size(A)]



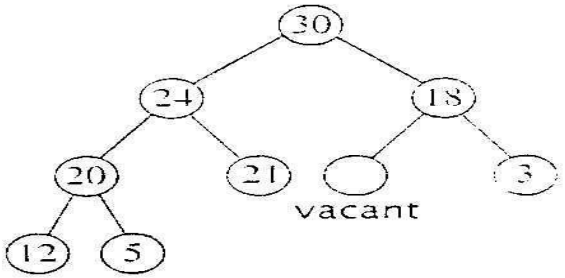
(a) The heap



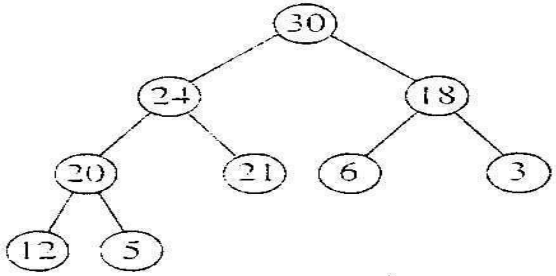
(b) The key at the root has been removed; the rightmost leaf at the bottom level has been removed. $K = 6$ must be reinserted.



(c) The larger child of vacant, 30, is greater than K, so it moves up and vacant moves down.

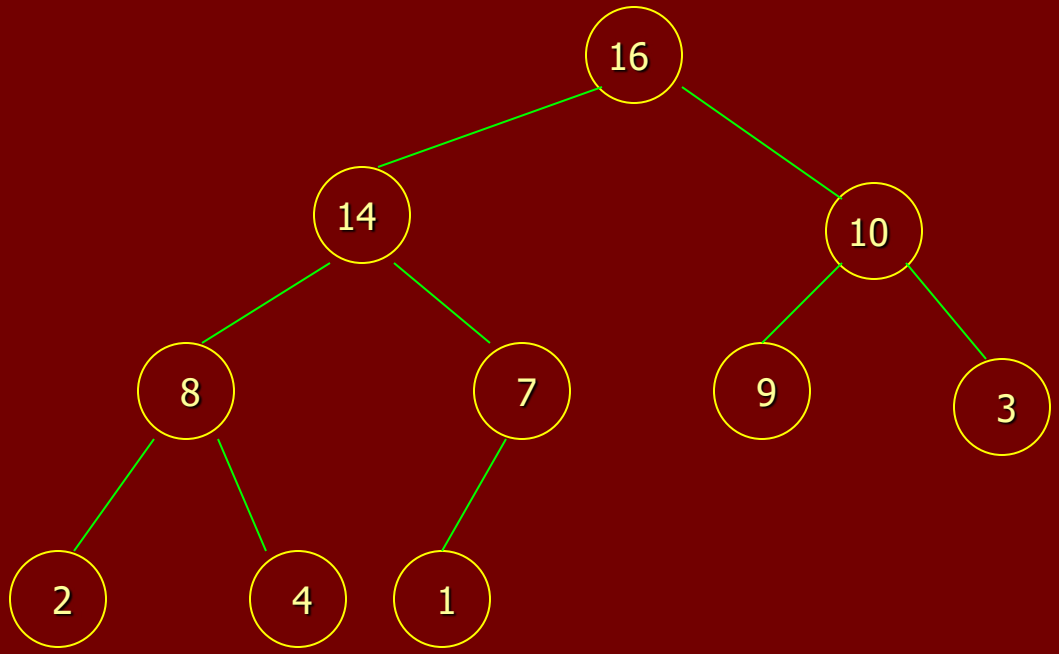


(d) The larger child of vacant, 18, is greater than K, so it moves up and vacant moves down.



(e) Finally, since vacant is a leaf, $K = 6$ is inserted.

- View book (Cormen) page number 137 figure 6.4 for better understanding of heap sort mechanism for following Tree.



Analyzing Heapsort

- The call to `BuildHeap()` takes $O(n)$ time
- Each of the $n - 1$ calls to `Heapify()` takes $O(\lg n)$ time

- Thus the total time taken by `HeapSort()`
  $= O(n) + (n - 1)\, O(\lg n)$
  $= O(n) + O(n \lg n)$
  $= O(n \lg n)$