Design and Analysis of Algorithms (BCS-28)

[B Tech IIIrd Year, Vth Sem, Session: 2020-21]

Prof. Rakesh Kumar
Department of Computer Science and Engineering
MMM University of Technology Gorakhpur-273010
Email: rkcs@mmmut.ac.in, rkiitr@gmail.com

DESIGN & ANALYSIS OF ALGORITHMS (BCS-28)

Course Category	Department Core (DC)
Pre-requisite Subject	NIL
Contact Hours/Week	Lecture: 3, Tutorial: 1, Practical: 2
Number of Credits	5
Course Assessment Methods	Continuous assessment through tutorials, attendance, home assignments, quizzes, practical work, record, viva voce and Three Minor tests and One Major Theory & Practical Examination
Course Outcomes	The students are expected to be able to demonstrate the following knowledge, skills and attitudes after completing this course.
1. Define the basic concepts of algorithms and analyze the performance of algorithms. 2. Discuss various algorithm design techniques for developing algorithms. 3. Discuss various searching, sorting and graph traversal algorithms. 4. Understand NP completeness and identify different NP complete problems. 5. Discuss various advanced topics on algorithm	

DESIGN & ANALYSIS OF ALGORITHMS (BCS-28)

UNIT-I	
Introduction: Algorithms, Analyzing Algorithms, Complexity of Algorithms, Growth of Functions, Performance Measurements, Sorting and Order Statistics - Shell Sort, Quick Sort, Merge Sort, Heap Sort, Comparison of Sorting Algorithms, Sorting in Linear Time. Divide and Conquer with Examples such as Sorting, Matrix Multiplication, Convex Hull and Searching.	9
UNIT-II	
Greedy Methods with Examples such as Optimal Reliability Allocation, Knapsack, Minimum Spanning Trees – Prim‟s and Kruskal‟s Algorithms, Single Source Shortest Paths - Dijkstra‟s and Bellman Ford Algorithms. Dynamic Programming with Examples such as Multistage Graphs, Knapsack, All Pair Shortest Paths -Warshal‟s and Floyd‟s Algorithms, Resource Allocation Problem.	9

DESIGN & ANALYSIS OF ALGORITHMS (BCS-28)

UNIT-III	
Backtracking, Branch and Bound with Examples such as Travelling Salesman Problem, Graph Coloring, N-Queen Problem, Hamiltonian Cycles and Sum of Subsets Advanced Data Structures: Red-Black Trees, B – Trees, Binomial Heaps, Fibonacci Heaps.	
UNIT-IV	
Selected Topics: String Matching, Text Processing- Justification of Text, Theory of NP-Completeness, Approximation Algorithms and Randomized Algorithms, Algebraic Computation, Fast Fourier Transform.	

DESIGN & ANALYSIS OF ALGORITHMS (BCS-28)

EXPERIMENTS

1. To analyze time complexity of Insertion sort.
2. To analyze time complexity of Quick sort.
3. To analyze time complexity of Merge sort.
4. To Implement Largest Common Subsequence.
5. To Implement Matrix Chain Multiplication.
6. To Implement Strassen‟s matrix multiplication Algorithm, Merge sort and Quick sort.
7. To implement Knapsack Problem.
8. To implement Activity Selection Problem.
9. To implement Dijkstra‟s Algorithm.
10. To implement Warshall‟s Algorithm.
11. To implement Bellman Ford‟s Algorithm.
12. To implement Naïve String Matching Algorithm.
13. To implement Rabin Karp String Matching Algorithm
14. To implement Prim‟s Algorithm.
15. To implement Kruskal‟s Algorithm.

DESIGN & ANALYSIS OF ALGORITHMS (BCS-28)

Textbooks

1. Thomas H. Coreman, Charles E. Leiserson and Ronald L. Rivest, Introduction to Algorithms, PHI.
2. RCT Lee, SS Tseng, RC Chang and YT Tsai, “Introduction to the Design and Analysis of Algorithms”, McGraw Hill, 2005.
3. Ellis Horowitz and Sartaj Sahni, Fundamentals of Computer Algorithms, Computer Science Press, Maryland, 1978
4. Berman, Paul,” Algorithms”, Cengage Learning.
5. Aho, Hopcraft, Ullman, “The Design and Analysis of Computer Algorithms” Pearson Education, 2008.

Reference books

1. Berlion, P. Izard, P., Algorithms-The Construction, Proof and Analysis of Programs, 1986. Johan Wiley & Sons.
2. Bentley, J.L., Writing Efficient Programs, PHI
3. Ellis Horowitz, Sartaj Sahni, and Sanguthevar Rajasekaran, Computer Algorithms, W. H. Freeman, NY, 1998
4. Goodman, S.E. & Hedetnien, introduction to Design and Analysis of Algorithm1997, MGH.
5. Knuth, D.E , Fundamentals of Algorithms: The Art of Computer Programming Vol,1985

UNIT – I

What is an algorithm?

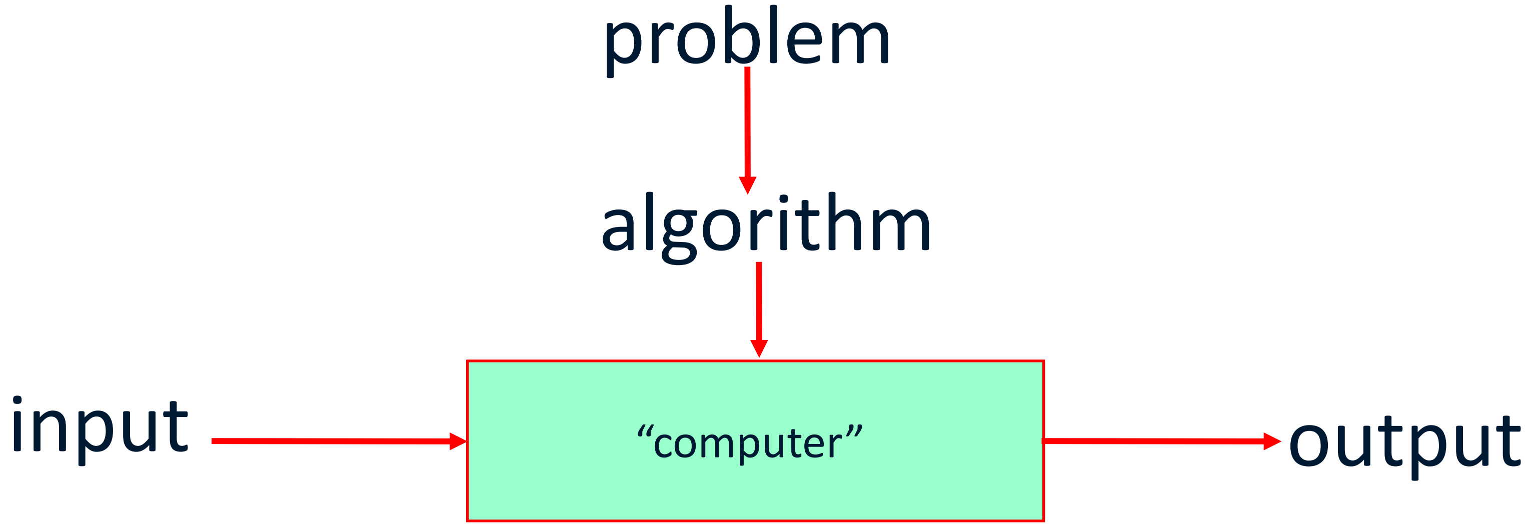
- An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values as output.

OR

- An algorithm is a list of steps (sequence of unambiguous instructions) for solving a problem that transforms the input into the output.

OR

- A finite set of instructions that specifies a sequence of operation to be carried out in order to solve a specific problem is called an algorithm



Example: gcd*(m, n)*

Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.

The greatest common divisor of two nonnegative, not-both-zero integers ***m*** and ***n***, denoted gcd***(m, n)***, is defined as the largest integer that divides both ***m*** and ***n*** evenly.

In modern terms, ***Euclid's algorithm*** is based on applying repeatedly the equality

$$\gcd(m, n) = \gcd(n, m \bmod n),$$

where ***m*** mod ***n*** is the remainder of the division of ***m*** by ***n***, until ***m*** mod ***n*** is equal to 0.

Since gcd***(m,*** 0***)*** = ***m***, the last value of ***m*** is also the greatest common divisor of the initial ***m*** and ***n***.

I. **Euclid's algorithm** for computing gcd*(m, n)*

For example, gcd(60, 24) can be computed as follows:

gcd(60, 24) = gcd(24, 12) = gcd(12, 0) = 12

Here is a more structured description of this algorithm:

Euclid's algorithm for computing gcd*(m, n)*

Step 1 If ***n*** = 0, return the value of ***m*** as the answer and stop; otherwise, proceed to Step 2.

Step 2 Divide ***m*** by ***n*** and assign the value of the remainder to ***r***.

Step 3 Assign the value of ***n*** to ***m*** and the value of ***r*** to ***n***. Go to Step 1.

Euclid's algorithm for computing gcd(*m, n*) (contd…)

Alternatively, we can express the same algorithm in pseudocode:

ALGORITHM *Euclid*(***m, n***)

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

//Input: Two nonnegative, not-both-zero integers ***m*** and ***n***
//Output: Greatest common divisor of ***m*** and ***n***

```
while $n$ = 0 do
$r \leftarrow m$ mod $n$
$m \leftarrow n$
$n \leftarrow r$ return $m$
```

II. Middle-school procedure for computing gcd*(m, n)*

Step 1 Find the prime factors of ***m***.
Step 2 Find the prime factors of ***n***.

Step 3 Identify all the common factors in the two prime expansions found in Step 1 and Step 2.
(If p is a common factor occurring p_m and p_n times in m and n, respectively, it should be repeated $\min\{p_m, p_n\}$ times.)

Step 4 Compute the product of all the common factors and return it as the greatest common divisor of the numbers given.

Thus, for the numbers 60 and 24, we get
$60 = 2 \cdot 2 \cdot 3 \cdot 5$

$24 = 2 \cdot 2 \cdot 2 \cdot 3$
$\gcd(60, 24) = 2 \cdot 2 \cdot 3 = 12.$

Characteristics of an Algorithm

- Input
- Output
- Definiteness
- Finiteness
- Effectiveness
- Correctness
- Simplicity
- Unambiguous
- Feasibility
- Portable
- Independent

Apriori and posteriori Analysis

- **Apriori analysis** means, analysis is performed prior to running it on a specific system. This analysis is a stage where a function is defined using some theoretical model.

  Hence, we determine the time and space complexity of an algorithm by just looking at the algorithm rather than running it on a particular system with a different memory, processor, and compiler.

- **Posteriori analysis** of an algorithm means we perform analysis of an algorithm only after running it on a system. It directly depends on the system and changes from system to system.
- In an industry, we cannot perform Aposteriori analysis as the software is generally made for an anonymous user, which runs it on a system different from those present in the industry.
- In Apriori, it is the reason that we use asymptotic notations to determine time and space complexity as they change from computer to computer; however, asymptotically they are the same.

Algorithm Expectation

- **Correctness:**
  - Correct: Algorithms must produce correct result.
  - Produce an incorrect answer: Even if it fails to give correct results all the time still there is a control on how often it gives wrong result. Eg. Rabin-Miller Primality Test
  - Approximation algorithm: Exact solution is not found, but near optimal solution can be found out. (Applied to optimization problem.)
- **Less resource usage:**
  - Algorithms should use less resources (time and space).
- **Resource usage:**

Here, the time is considered to be the primary measure of efficiency .

We are also concerned with how much the respective algorithm involves the computer memory. But mostly time is the resource that is dealt with.

The actual running time depends on a variety of backgrounds: like the speed of the Computer, the language in which the algorithm is implemented, the compiler/interpreter, skill of the programmers etc.

So, mainly the resource usage can be divided into: 1.Memory (space) 2.Time

Analysis of an algorithm

Performance

- Two areas are important for performance:

1. space efficiency - the memory required, also called, space complexity

2. time efficiency - the time required, also called time complexity

- **Space efficiency:**

There are some circumstances where the space/memory used must be analyzed. For example, for large quantities of data or for embedded systems programming.

- Components of space/memory use:

1. instruction space	Affected by: the compiler, compiler options, target computer (cpu)
2. data space	Affected by: the data size/dynamically allocated memory, static program variables,
3. run-time stack space	Affected by: the compiler, run-time function calls and recursion, local variables, parameters

Space efficiency

3. run-time stack space

Affected by: the compiler, run-time function calls and recursion, local variables, parameters

The space requirement has fixed/static/compile time and a variable/dynamic/runtime components. Fixed components are the machine language instructions and static variables. Variable components are the runtime stack usage and dynamically allocated memory usage.

One circumstance to be aware of is, does the approach require the data to be duplicated in memory (as does merge sort). If so we have *2N* memory use.

Time efficiency

The actual running time depends on many factors:

- The speed of the computer: cpu (not just clock speed), I/O, etc.
- The compiler, compiler options .
- The quantity of data - ex. search a long list or short.
- The actual data - ex. in the sequential search if the name is first or last.

Time Efficiency - Approaches

When analyzing for time complexity we can take two approaches:

1. Order of magnitude/asymptotic categorization - This uses coarse categories and gives a general idea of performance. If algorithms fall into the same category, if data size is small, or if performance is critical, then the next approach can be looked at more closely.

2. Estimation of running time.

Algorithm as a Technology

Algorithms are just like a technology. We all use latest and greatest processors but we need to run implementations of good algorithms on that computer in order to properly take benefits of our money that we spent to have the latest processor.

Example:

- A faster computer (computer A) running a sorting algorithm whose running time on n values grows like n^2 against a slower computer (computer B) running a sorting algorithm whose running time grows like n lg n.
- They each must sort an array of 10 million numbers.

Algorithm as a Technology (Contd..)

- Computer A executes 10 billion instructions per second (faster than any single sequential computer at the time of this writing)
- Computer B executes only 10 million instructions per second, so that computer A is 1000 times faster than computer B in raw computing power.
- To make the difference even more dramatic, suppose that the world's craftiest programmer codes in machine language for computer A, and the resulting code requires $2n^2$ instructions to sort n numbers.
- Suppose further that just an average programmer writes for computer B, using a high level language with an inefficient compiler, with the resulting code taking 50n lg n instructions.

Algorithm as a Technology (Contd..)

Computer A	Computer B
Running time grows like n^2.	Grows in n log n.
10 billion instructions per sec.	10million instruction per sec
$2n^2$ instruction.	50 n log n instruction
Time taken= 2* (107)2/1010=20,000	(50*107*log 107)/107≈1163
It is more than 5.5hrs	it is under 20 mins.

- So choosing a good algorithm (algorithm with slower rate of growth) as used by computer B affects a lot.

Faster Algorithm vs. Faster CPU

- A faster algorithm running on a slower machine will always win for large enough instances
- Suppose algorithm S1 sorts n keys in $2n^2$ instructions
- Suppose computer C1 executes 1 billion instruc/sec
  - When n = 1 million, takes 2000 sec
- Suppose algorithm S2 sorts n keys in $50n\log_2 n$ instructions
- Suppose computer C2 executes 10 million instruc/sec
  - When n = 1 million, takes 100 sec

Algorithm Design Techniques

The following is a list of several popular design approaches:

1. Divide and Conquer Approach: It is a top-down approach. The algorithms which follow the divide & conquer techniques involve three steps:

•Divide the original problem into a set of subproblems.

•Solve every subproblem individually, recursively.

•Combine the solution of the subproblems (top level) into a solution of the whole original problem.

2. Greedy Technique: Greedy method is used to solve the optimization problem. An optimization problem is one in which we are given a set of input values, which are required either to be maximized or minimized (known as objective), i.e. some constraints or conditions.

•Greedy Algorithm always makes the choice (greedy criteria) looks best at the moment, to optimize a given objective.

•The greedy algorithm doesn't always guarantee the optimal solution however it generally produces a solution that is very close in value to the optimal.

Algorithm Design Techniques

3. Dynamic Programming: Dynamic Programming is a bottom-up approach we solve all possible small problems and then combine them to obtain solutions for bigger problems.
This is particularly helpful when the number of copying subproblems is exponentially large. Dynamic Programming is frequently related to **Optimization Problems**.

4. Branch and Bound: In Branch & Bound algorithm a given subproblem, which cannot be bounded, has to be divided into at least two new restricted subproblems. Branch and Bound algorithm are methods for global optimization in non-convex problems. Branch and Bound algorithms can be slow, however in the worst case they require effort that grows exponentially with problem size, but in some cases we are lucky, and the method coverage with much less effort.

Algorithm Design Techniques

5. Randomized Algorithms: A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased random bits, and it is then allowed to use these random bits to influence its computation.

6. Backtracking Algorithm: Backtracking Algorithm tries each possibility until they find the right one. It is a depth-first search of the set of possible solution. During the search, if an alternative doesn't work, then backtrack to the choice point, the place which presented different alternatives, and tries the next alternative.

7. Randomized Algorithm: A randomized algorithm uses a random number at least once during the computation make a decision.

Algorithm Design and Analysis Process

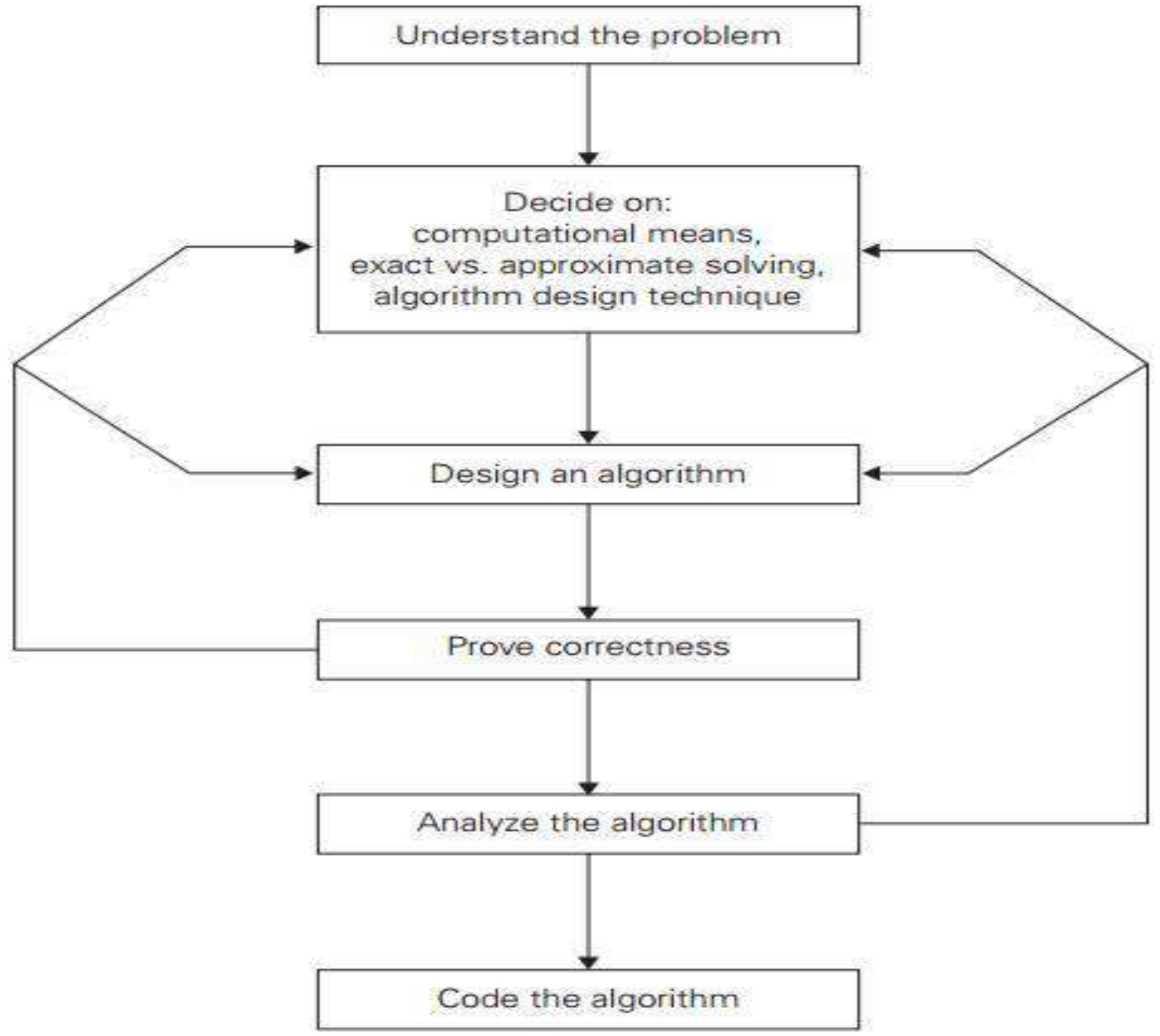


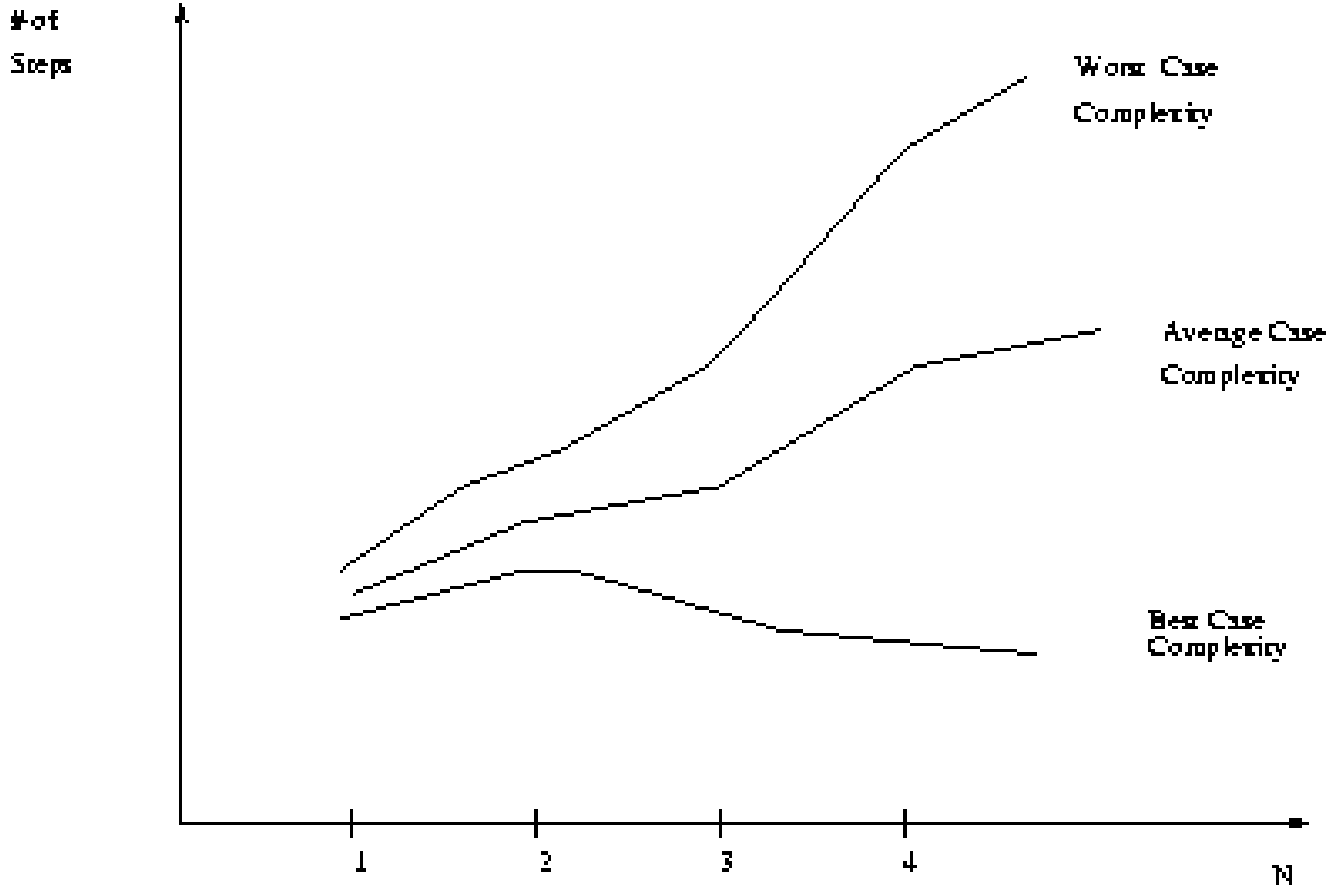
FIGURE 1.2 Algorithm design and analysis process.

Algorithm Analysis

Generally, we perform the following types of analysis −

- Worst-case − The maximum number of steps taken on any instance of size a.
- Best-case − The minimum number of steps taken on any instance of size a.
- Average case − An average number of steps taken on any instance of size a.

Performance measures: worst case, average case and Best case



Running time

- The running time depends on the input: an already sorted sequence is easier to sort.
- Major Simplifying Convention: Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
    - ➢ $T_A(n)$ = time of A on length n inputs
- Generally, we seek upper bounds on the running time, to have a guarantee of performance.

Kinds of analyses

Worst-case: (usually)

- $T(n)$ = maximum time of algorithm on any input of size n.

Average-case: (sometimes)

- $T(n)$ = expected time of algorithm over all inputs of size n.
- Need assumption of statistical distribution of inputs.

Best-case: (NEVER)

- Cheat with a slow algorithm that works fast on *some* input.

Methodology of Analysis

- **Asymptotic Analysis**
  - The asymptotic behavior of a function ***f(n)*** refers to the growth of ***f(n)*** as **n** gets large.
  - We typically ignore small values of **n**, since we are usually interested in estimating how slow the program will be on large inputs.
  - A good rule of thumb is that the slower the asymptotic growth rate, the better the algorithm. Though it’s not always true.
  - For example, a linear algorithm $f(n)=d*n+k$ is always asymptotically better than a quadratic one, $f(n)=c.\ n^2+q$.

Methodology of Analysis

- **Solving Recurrence Equations**
- A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs. Recurrences are generally used in divide-and-conquer paradigm.
- Let us consider ***T(n)*** to be the running time on a problem of size **n**.
- If the problem size is small enough, say ***n < c*** where **c** is a constant, the straightforward solution takes constant time, which is written as **θ(1)**. If the division of the problem yields a number of sub-problems with size *nb*
- A recurrence relation can be solved using the following methods −
  - **Substitution Method** − In this method, we guess a bound and using mathematical induction we prove that our assumption was correct.
  - **Recursion Tree Method** − In this method, a recurrence tree is formed where each node represents the cost.
  - **Master's Theorem** − This is another important technique to find the complexity of a recurrence relation.

Growth of Function (Asymptotic Notations)

- The complexity of an algorithm describes the efficiency of the algorithm in terms of the amount of the memory required to process the data and the processing time.
- Complexity of an algorithm is analyzed in two perspectives: **Time** and **Space**.
- Execution time of an algorithm depends on the instruction set, processor speed, disk I/O speed, etc. Hence, we estimate the efficiency of an algorithm asymptotically.
- Time function of an algorithm is represented by **T(n)**, where **n** is the input size.
- Different types of asymptotic notations are used to represent the complexity of an algorithm. Following asymptotic notations are used to calculate the running time complexity of an algorithm.
- **O** − Big Oh
- **Ω** − Big omega
- **θ** − Big theta
- **o** − Little Oh
- **ω** − Little omega

Growth of Function (Asymptotic Notations)contd..

- **O: Asymptotic Upper Bound**
- ‘O’ (Big Oh) is the most commonly used notation. A function ***f(n)*** can be represented is the order of ***g(n)*** that is ***O(g(n))***, if there exists a value of positive integer **n** as $\mathbf{n_0}$ and a positive constant **c** such that −

  $f(n) \leqslant c.g(n)$ for $n > n0$ in all case
- Hence, function ***g(n)*** is an upper bound for function ***f(n)***, as ***g(n)*** grows faster than ***f(n)***.

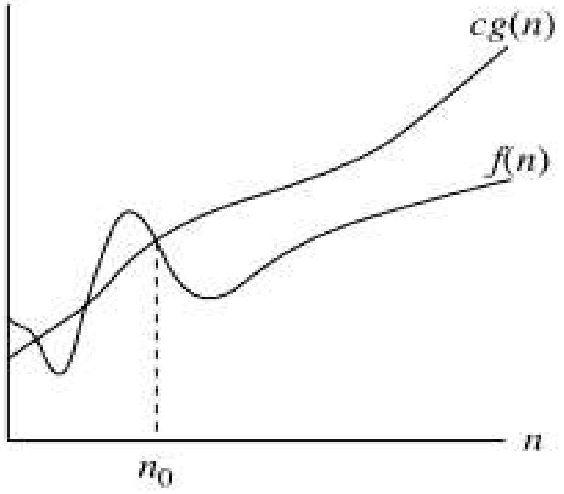
Example

- Let us consider a given function, $f(n)=4.n^3+10.n^2+5.n+1$
- Considering $g(n)=n^3$
- $f(n) \leqslant 5.g(n)$ for all the values of $n>2$
- Hence, the complexity of ***f(n)*** can be represented as $O(g(n))$ i.e. $O(n^3)$

Growth of Function (Asymptotic Notations)Contd…

***O*-notation**

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



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

Big-O Examples

Example 1

$5n + 7 <= 5n + n$ for $n >= 7$

$<= 6n$

So $c = 6$ and $n_0 = 7$, thus $5n + 7$ is $O(n)$.

Example 2

$3n^2 + 4n <= 3n^2 + n^2$ for $n >= 4$

$<= 4n^2$

So $c = 4$ and $n_0 = 4$, thus $3n^2 + 4n$ is $O(n^2)$.

It is usually assumed the bound is tight. For example, $3n + 4$ function is bounded by n^2 for $n > 4$ and so is $O(n^2)$. This is not a tight bound however, $3n + 4$ is $O(n)$.

Growth of Function (Asymptotic Notations)Contd…

- **θ: Asymptotic Tight Bound**
- We say that $f(n)=\theta(g(n))$ when there exist constants $\mathbf{c_1}$ and $\mathbf{c_2}$ that $c1.g(n)\leqslant f(n)\leqslant c2.g(n)$for all sufficiently large value of **n**. Here **n** is a positive integer.
- This means function **g** is a tight bound for function **f**.

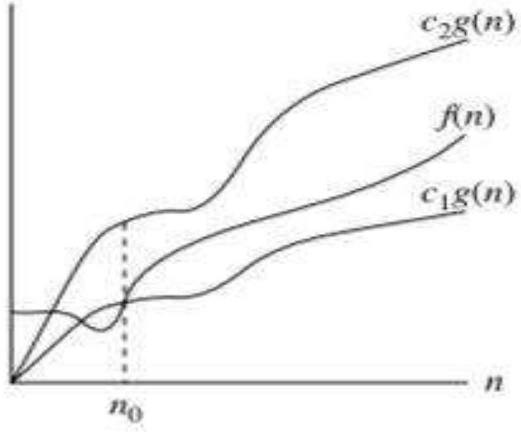
Example

- Let us consider a given function, $f(n)=4.n^3+10.n^2+5.n+1$
- Considering $g(n)=n^3$, $4.g(n)\leqslant f(n)\leqslant 5.g(n)$for all the large values of **n**.
- Hence, the complexity of ***f(n)*** can be represented as $\theta(g(n))$, i.e. $\theta(n^3)$.

Growth of Function (Asymptotic Notations)Contd…

Θ-notation

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



$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Growth of Function (Asymptotic Notations)Contd…

- Ω: Asymptotic Lower Bound
- We say that f(n)=Ω(g(n))

f(n)=Ω(g(n)) when there exists constant **c** that $f(n) \geqslant c.g(n)$,

$f(n) \geqslant c.g(n)$ for all sufficiently large value of **n**. Here **n** is a positive integer. It means function ***g*** is a lower bound for function ***f***; after a certain value of ***n, f*** will never go below ***g***.

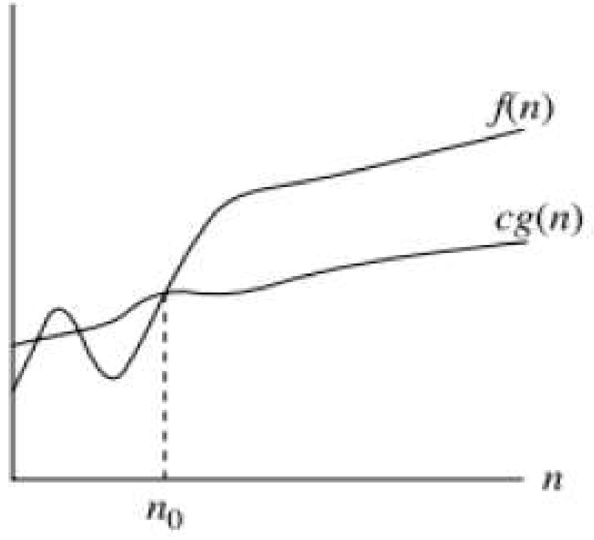
Example

- Let us consider a given function, $f(n)=4.n^3+10.n^2+5.n+1$
- Considering $g(n)=n^3$, $f(n) \geqslant 4.g(n)$ for all the values of $n>0$.
- Hence, the complexity of ***f(n)*** can be represented as Ω(g(n)), i.e. $\Omega(n^3)$

Growth of Function (Asymptotic Notations)Contd…

Ω-notation

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



$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

Common Asymptotic Notations

Function	Big-O	Name
1	$O(1)$	constant
$\log n$	$O(\log n)$	logarithmic
n	$O(n)$	linear
$n \log n$	$O(n \log n)$	n log n
n^2	$O(n^2)$	quadratic
n^3	$O(n^3)$	cubic
2^n	$O(2^n)$	exponential
$n!$	$O(n!)$	factorial

Rate of growth of function

Logarithmic	**Linear**	**Linear logarithmic**	**Quadratic**	**Polynomial**	**Exponential**
Log_2n	N	$nlog_2n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296
3.322	10	33.22	10^2	10^3	$> 10^3$
6.644	10^2	664.4	10^4	10^6	$> >10^{25}$
9.966	10^3	9966.0	10^6	10	$> > 10^{250}$

Comparison of Functions

- $f(n) = O(g(n))$ and
  $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$

  Transitivity
- $f(n) = \Omega(g(n))$ and
  $g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$
- $f(n) = \Theta(g(n))$ and
  $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$

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

  Reflexivity

Comparison of Functions

- $f(n) = \Theta(g(n)) \Longleftrightarrow g(n) = \Theta(f(n))$ *Symmetry*

- $f(n) = O(g(n)) \Longleftrightarrow g(n) = \Omega(f(n))$ *Transpose*
- $f(n) = o(g(n)) \Longleftrightarrow g(n) = \omega(f(n))$ *Symmetry*

- $f(n) = O(g(n))$ and
  $f(n) = \Omega(g(n)) \Rightarrow f(n) = \Theta(g(n))$ *Theorem 3.1*

Asymptotic Analysis and Limits

If $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = 0$, then $f(n) = o(g(n))$.

If $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = c$, for some constant $c > 0$, then $f(n) = \Theta(g(n))$.

If $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = 0$, then $a^{f(n)} = o(a^{g(n)})$, for any $a > 1$.

$f(n) = o(g(n)) \Longrightarrow a^{f(n)} = o(a^{g(n)})$, for any $a > 1$.

$f(n) = \Theta(g(n)) \not\Longrightarrow a^{f(n)} = \Theta(a^{g(n)})$

Comparison of Functions

- $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n)) \Rightarrow$
  $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$

- $f(n) = O(g(n)) \Rightarrow f(n) + g(n) = O(g(n))$

Standard Notation and Common Functions

- ***Monotonicity***

A function $f(n)$ is ***monotonically increasing*** if $m \le n$ implies $f(m) \le f(n)$.

A function $f(n)$ is ***monotonically decreasing*** if $m \le n$ implies $f(m) \ge f(n)$.

A function $f(n)$ is ***strictly increasing***
if $m < n$ implies $f(m) < f(n)$.

A function $f(n)$ is ***strictly decreasing***
if $m < n$ implies $f(m) > f(n)$.

Standard Notation and Common Functions

- ***Floors and ceilings***

  For any real number x, the greatest integer less than or equal to x is denoted by $\lfloor x \rfloor$.

  For any real number x, the least integer greater than or equal to x is denoted by $\lceil x \rceil$.

  For all real numbers x,
  $x-1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x+1$.

  Both functions are monotonically increasing.

Recurrences and Running Time

- An equation or inequality that describes a function in terms of its value on smaller inputs.

$$T(n) = T(n-1) + n$$

- Recurrences arise when an algorithm contains recursive calls to itself
- What is the actual running time of the algorithm?
- Need to solve the recurrence
    - Find an explicit formula of the expression
    - Bound the recurrence by an expression that involves n

Example Recurrences

- $T(n) = T(n-1) + n$ $\Theta(n^2)$
  - Recursive algorithm that loops through the input to eliminate one item
- $T(n) = T(n/2) + c$ $\Theta(\lg n)$
  - Recursive algorithm that halves the input in one step
- $T(n) = T(n/2) + n$ $\Theta(n)$
  - Recursive algorithm that halves the input but must examine every item in the input
- $T(n) = 2T(n/2) + 1$ $\Theta(n)$
  - Recursive algorithm that splits the input into 2 halves and does a constant amount of other work

Analysis of BINARY-SEARCH

Alg.: BINARY-SEARCH (A, lo, hi, x)

```
if (lo > hi)                                   ← constant time: $c_1$
    return FALSE
mid ← $\lfloor (lo+hi)/2 \rfloor$              ← constant time: $c_2$
if x = A[mid]                                  ← constant time: $c_3$
    return TRUE
if ( x < A[mid] )
    BINARY-SEARCH (A, lo, mid-1, x)            ← same problem of size n/2
if ( x > A[mid] )
    BINARY-SEARCH (A, mid+1, hi, x)            ← same problem of size n/2
```

- $T(n) = c + T(n/2)$
  - $T(n)$ – running time for an array of size n

Methods for Solving Recurrences

- Iteration method
- Substitution method
- Recursion tree method
- Master method

The Iteration Method

- Convert the recurrence into a summation and try to bound it using known series
    - Iterate the recurrence until the initial condition is reached.
    - Use back-substitution to express the recurrence in terms of *n* and the initial (boundary) condition.

The Iteration Method

$\mathbf{T(n) = c + T(n/2)}$

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

$T(n/4) = c + T(n/8)$

$$\begin{aligned} T(n) &= c + T(n/2) \\ &= c + c + T(n/4) \\ &= c + c + c + T(n/8) \end{aligned}$$

Assume $n = 2^k$

$$\begin{aligned} T(n) &= \underbrace{c + c + \ldots + c}_{k \text{ times}} + T(1) \\ &= c\lg n + T(1) \\ &= \Theta(\lg n) \end{aligned}$$

Iteration Method – Example

$$\mathbf{T(n) = n + 2T(n/2)}$$ Assume: $n = 2^k$

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

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

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

$= n + n + 4(n/4 + 2T(n/8))$

$= n + n + n + 8T(n/8)$

… $= in + 2^iT(n/2^i)$

$= kn + 2^kT(1)$

$= n\lg n + nT(1) = \Theta(n\lg n)$

The substitution method

1. Guess a solution
2. Use induction to prove that the solution works

Substitution method

- Guess a solution
  - $T(n) = O(g(n))$
  - Induction goal: apply the definition of the asymptotic notation
    - $T(n) \le d\, g(n)$, for some $d > 0$ and $n \ge n_0$ (strong induction)
  - Induction hypothesis: $T(k) \le d\, g(k)$ for all $k < n$
- Prove the induction goal
  - Use the **induction hypothesis** to find some values of the constants d and n_0 for which the **induction goal** holds

Example: Binary Search

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

- Guess: $T(n) = O(\lg n)$
  - Induction goal: $T(n) \le d \lg n$, for some d and $n \ge n_0$
  - Induction hypothesis: $T(n/2) \le d \lg(n/2)$
- Proof of induction goal:

$$T(n) = T(n/2) + c \le d \lg(n/2) + c$$

$$= d \lg n - d + c \le d \lg n$$

if: $-d + c \le 0$, $d \ge c$

Example

$$T(n) = T(n-1) + n$$

- Guess: $T(n) = O(n^2)$
  - Induction goal: $T(n) \le c\ n^2$, for some c and $n \ge n_0$
  - Induction hypothesis: $T(n-1) \le c(n-1)^2$ for all $k < n$
- Proof of induction goal:

$T(n) = T(n-1) + n \le c\ (n-1)^2 + n$

$= cn^2 - (2cn - c - n) \le cn^2$

if: $2cn - c - n \ge 0 \Leftrightarrow c \ge n/(2n-1) \Leftrightarrow c \ge 1/(2 - 1/n)$

  - For $n \ge 1 \Rightarrow 2 - 1/n \ge 1 \Rightarrow$ any $c \ge 1$ will work

The recursion-tree method

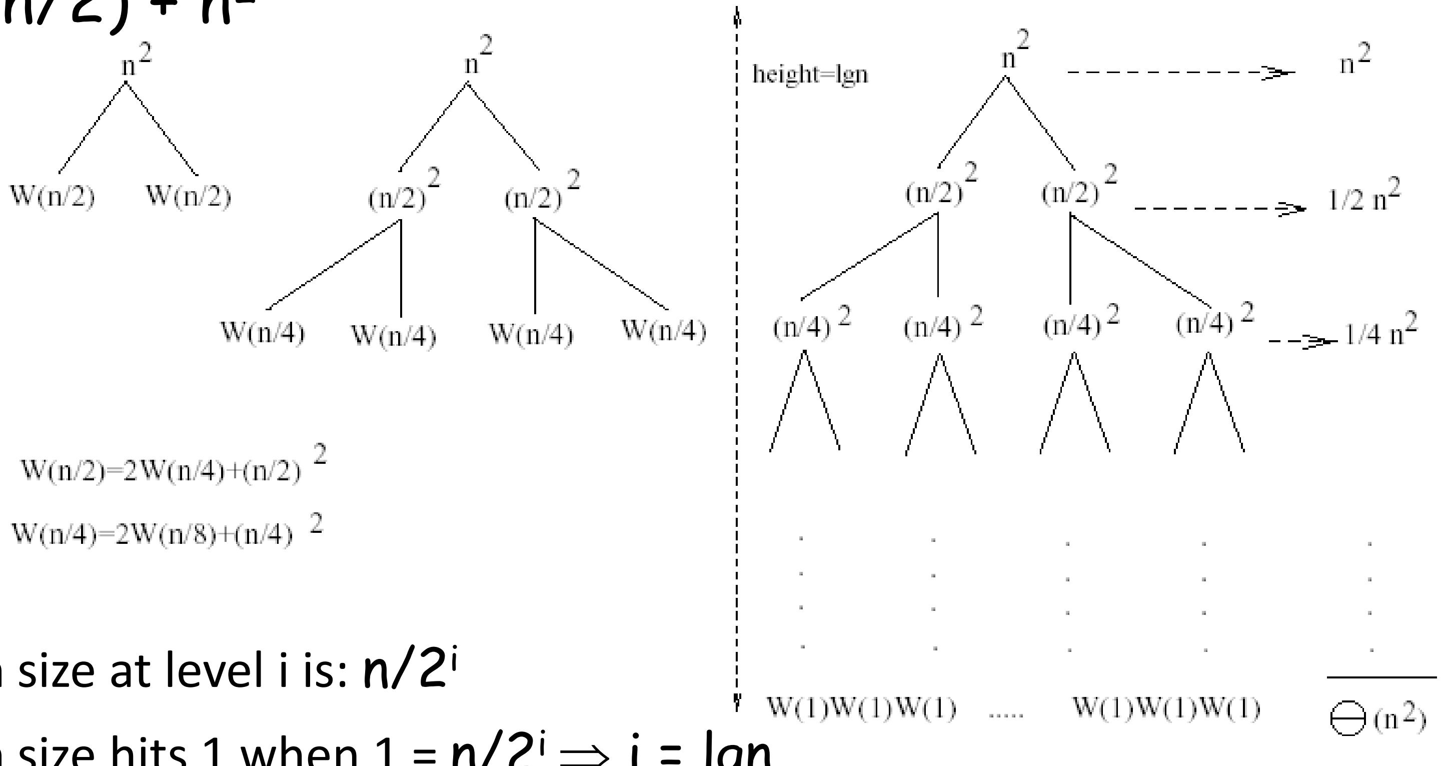
Convert the recurrence into a tree:

- Each node represents the cost incurred at various levels of recursion
- Sum up the costs of all levels

Used to “guess” a solution for the recurrence

Example 1

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

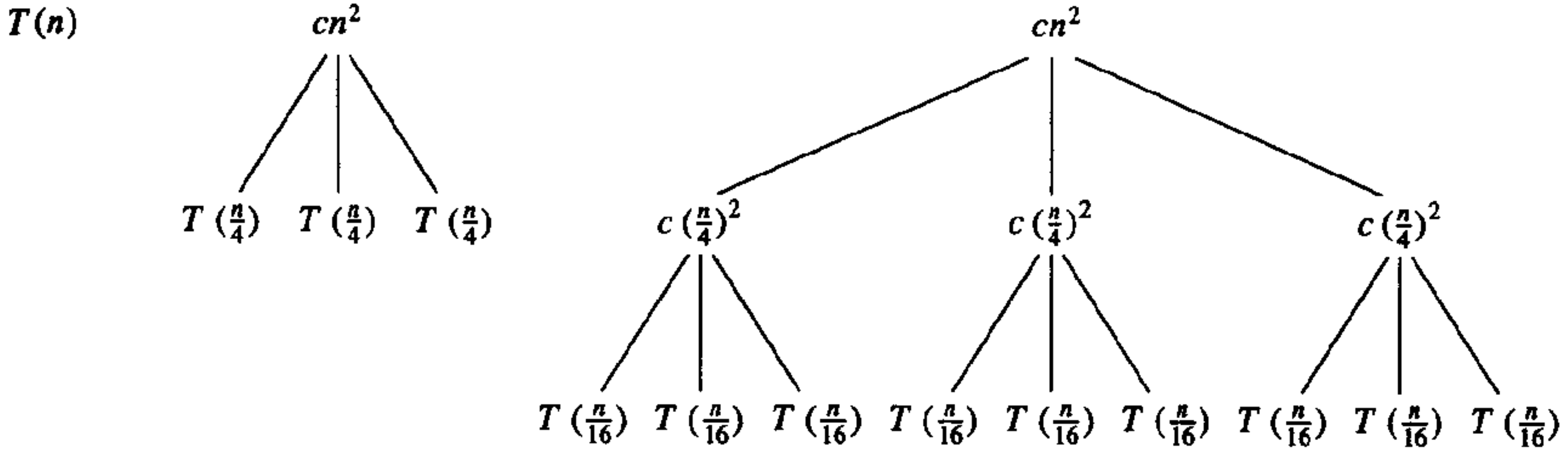


- Subproblem size at level i is: $n/2^i$
- Subproblem size hits 1 when $1 = n/2^i \Rightarrow i = \lg n$
- Cost of the problem at level i = $(n/2^i)^2$ No. of nodes at level $i = 2^i$
- Total cost: $W(n) = \sum_{i=0}^{\lg n-1} \frac{n^2}{2^i} + 2^{\lg n} W(1) = n^2 \sum_{i=0}^{\lg n-1} \left(\frac{1}{2}\right)^i + n \le n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i + O(n) = n^2 \frac{1}{1-\frac{1}{2}} + O(n) = 2n^2$

$\Rightarrow W(n) = O(n^2)$

Example 2

E.g.: $T(n) = 3T(n/4) + cn^2$



$T(n)$
cn^2
$T(\frac{n}{4})$ $T(\frac{n}{4})$ $T(\frac{n}{4})$
cn^2
$c(\frac{n}{4})^2$ $c(\frac{n}{4})^2$ $c(\frac{n}{4})^2$
$T(\frac{n}{16})$ $T(\frac{n}{16})$ $T(\frac{n}{16})$ $T(\frac{n}{16})$ $T(\frac{n}{16})$ $T(\frac{n}{16})$ $T(\frac{n}{16})$ $T(\frac{n}{16})$ $T(\frac{n}{16})$

- Subproblem size at level i is: $n/4^i$
- Subproblem size hits 1 when $1 = n/4^i \Rightarrow i = \log_4 n$
- Cost of a node at level i = $c(n/4^i)^2$
- Number of nodes at level i = $3^i \Rightarrow$ last level has $3^{\log_4 n} = n^{\log_4 3}$ nodes
- Total cost:

$$T(n) = \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right) \le \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right) = \frac{1}{1-\frac{3}{16}} cn^2 + \Theta\left(n^{\log_4 3}\right) = O(n^2)$$

$\Rightarrow T(n) = O(n^2)$

Master's method

- Solving recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where, $a \geq 1$, $b > 1$, and $f(n) > 0$

Idea: compare $f(n)$ with $n^{\log_b a}$

- $f(n)$ is asymptotically smaller or larger than $n^{\log_b a}$ by a polynomial factor n^{ε}
- $f(n)$ is asymptotically equal with $n^{\log_b a}$

Master's method

- For solving recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where, $a \geq 1$, $b > 1$, and $f(n) > 0$

Case 1: if $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then: $T(n) = \Theta(n^{\log_b a})$

Case 2: if $f(n) = \Theta(n^{\log_b a})$, then: $T(n) = \Theta(n^{\log_b a} \lg n)$

Case 3: if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, and if

$af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large n, then:

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

regularity condition

Example

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

$a = 2$, $b = 2$, $\log_2 2 = 1$

Compare $n^{\log_2 2}$ with $f(n) = n$

$\Rightarrow f(n) = \Theta(n) \Rightarrow$ Case 2

$\Rightarrow T(n) = \Theta(n\lg n)$

Example

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

$a = 2$, $b = 2$, $\log_2 2 = 1$

Compare n with $f(n) = n^2$

$\Rightarrow f(n) = \Omega(n^{1+\varepsilon})$ Case 3 $\Rightarrow$ verify regularity cond.

$a\ f(n/b) \leq c\ f(n)$

$\Leftrightarrow 2\ n^2/4 \leq c\ n^2 \Rightarrow c = ½$ is a solution $(c<1)$

$\Rightarrow T(n) = \Theta(n^2)$

Example

$$T(n) = 3T(n/4) + n\lg n$$

$a = 3$, $b = 4$, $\log_4 3 = 0.793$

Compare $n^{0.793}$ with $f(n) = n\lg n$

$f(n) = \Omega(n^{\log_4 3+\varepsilon})$ Case 3

Check regularity condition:

$$3*(n/4)\lg(n/4) \leq (3/4)n\lg n = c * f(n),\ c=3/4$$

$\Rightarrow T(n) = \Theta(n\lg n)$

The problem of sorting

Input: sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.

Output: permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

Example:

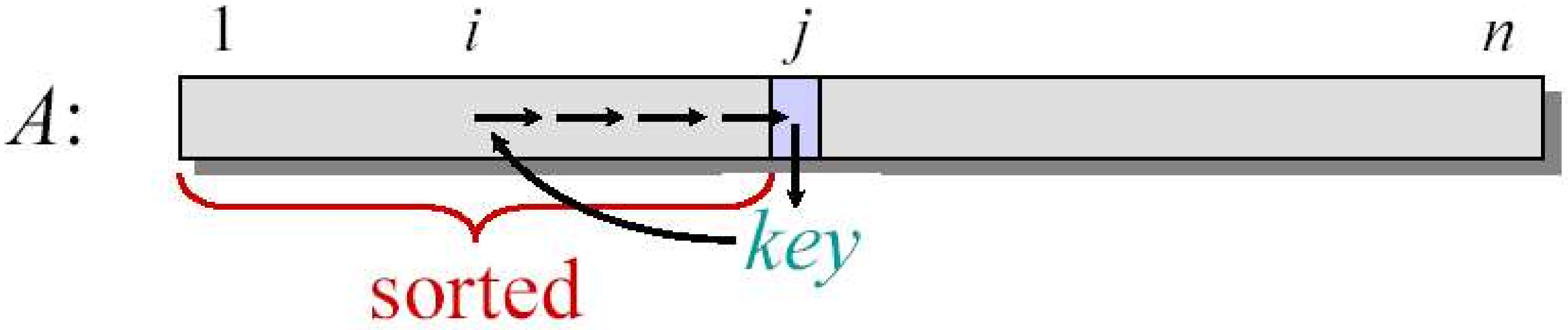
Input: 8 2 4 9 3 6

Output: 2 3 4 6 8 9

Insertion Sort

“pseudocode”

```
INSERTION-SORT $(A, n)$        $\triangleright A[1 \,..\, n]$
    for $j \leftarrow 2$ to $n$
        do $key \leftarrow A[j]$
           $i \leftarrow j - 1$
           while $i > 0$ and $A[i] > key$
               do $A[i+1] \leftarrow A[i]$
                  $i \leftarrow i - 1$
           $A[i+1] = key$
```



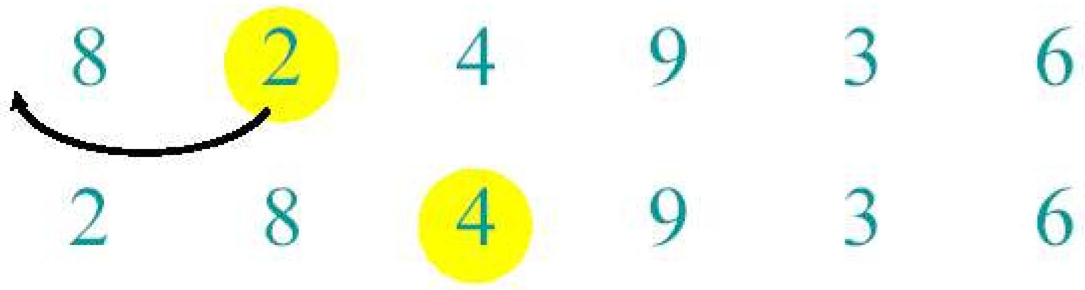
Example of Insertion Sort

8 2 4 9 3 6

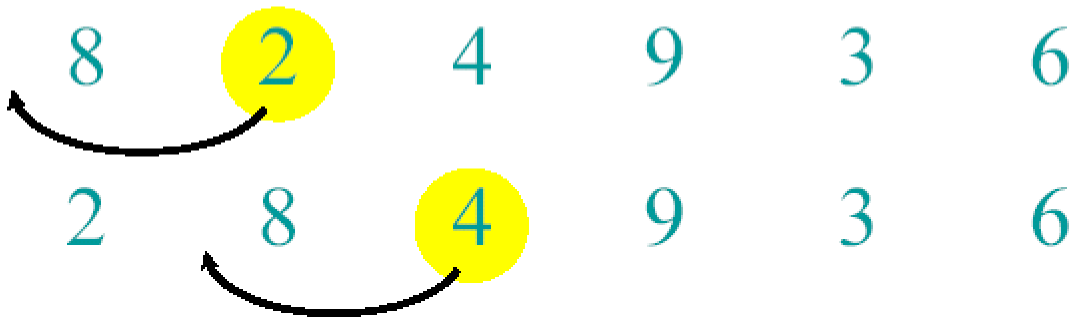
Example of Insertion Sort

$$\begin{tikzcd}
8 & 2 \arrow[l, bend left] & 4 & 9 & 3 & 6
\end{tikzcd}$$

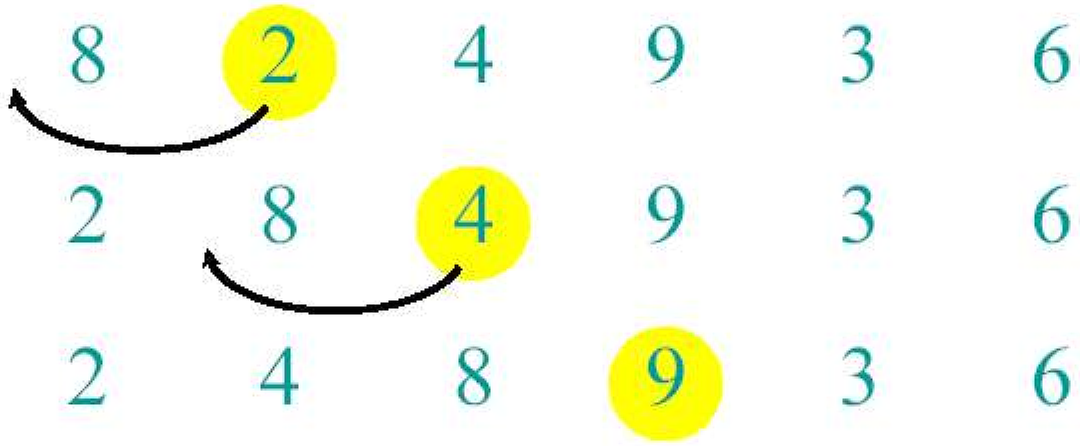
Example of Insertion Sort



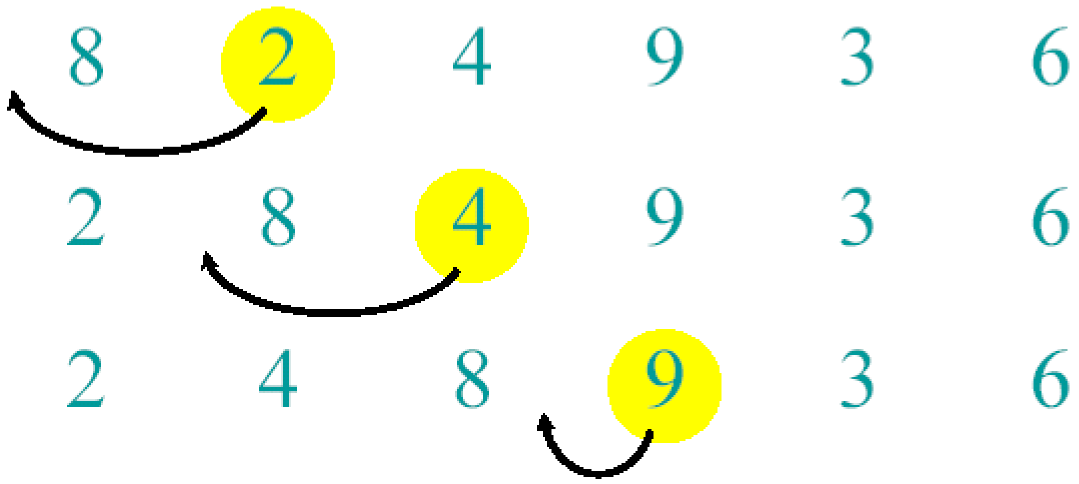
Example of Insertion Sort



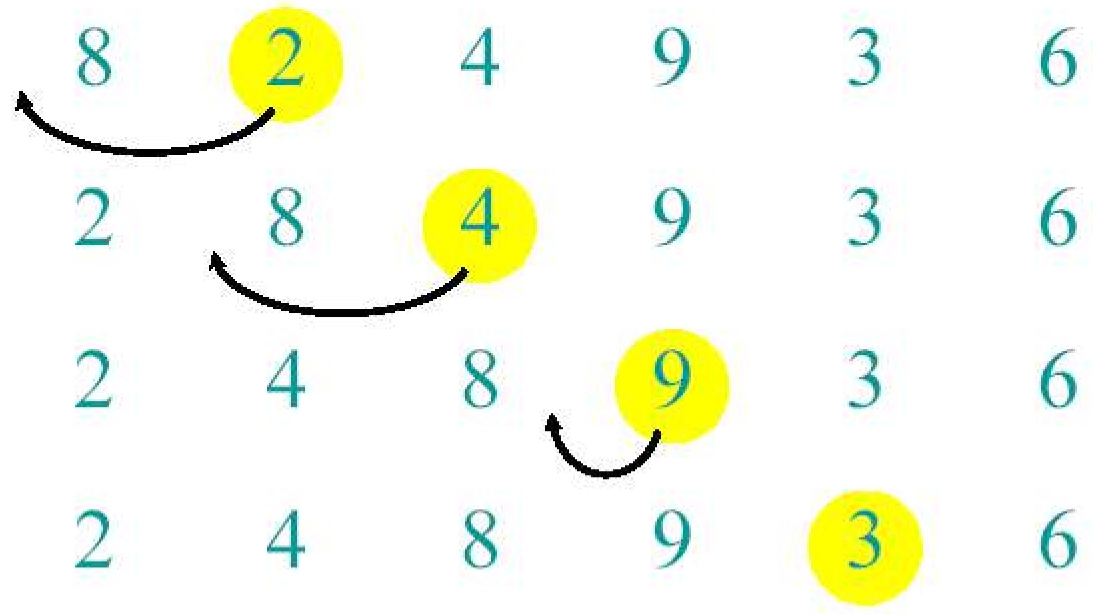
Example of Insertion Sort



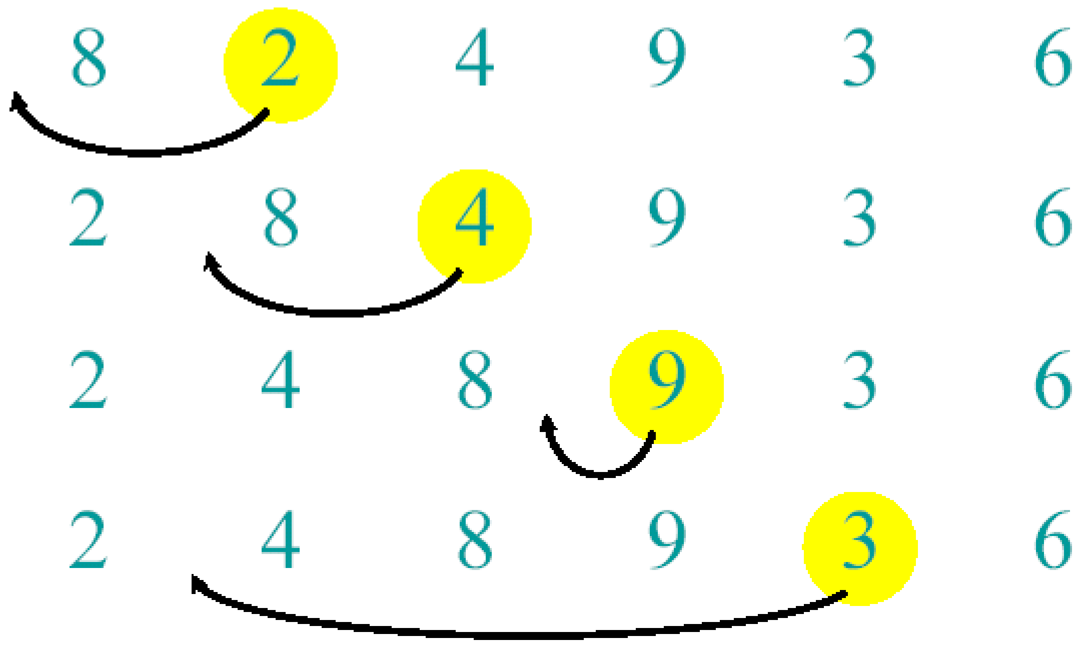
Example of Insertion Sort



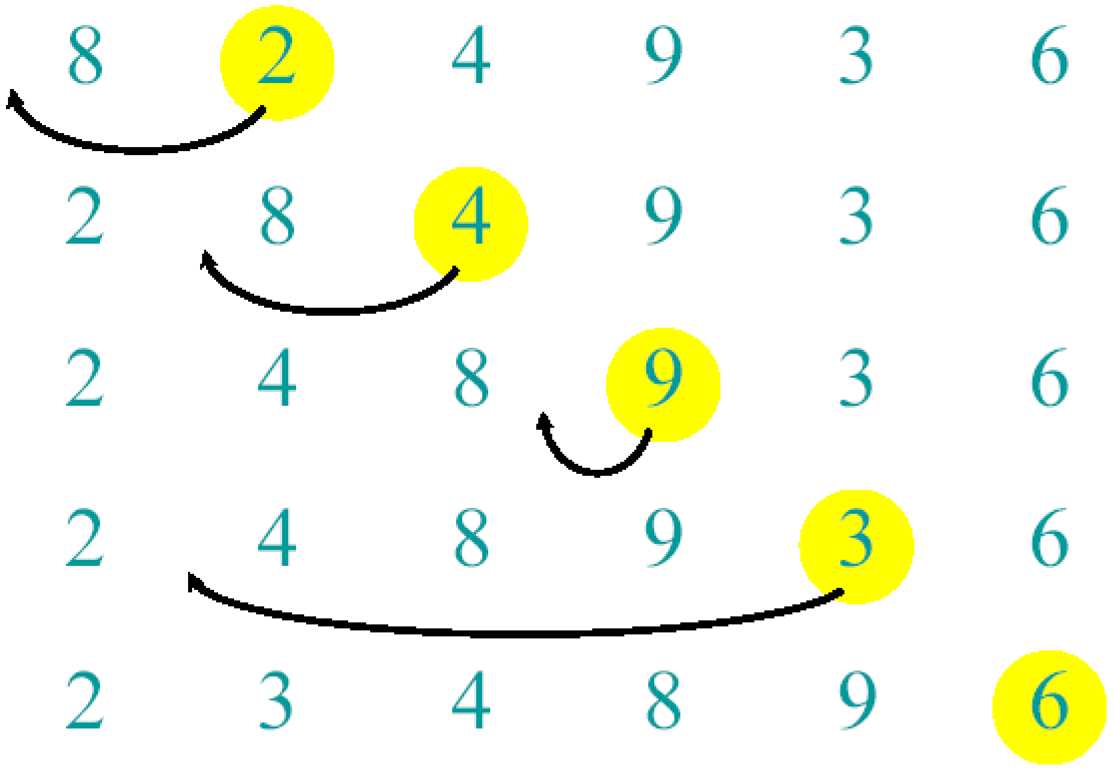
Example of Insertion Sort



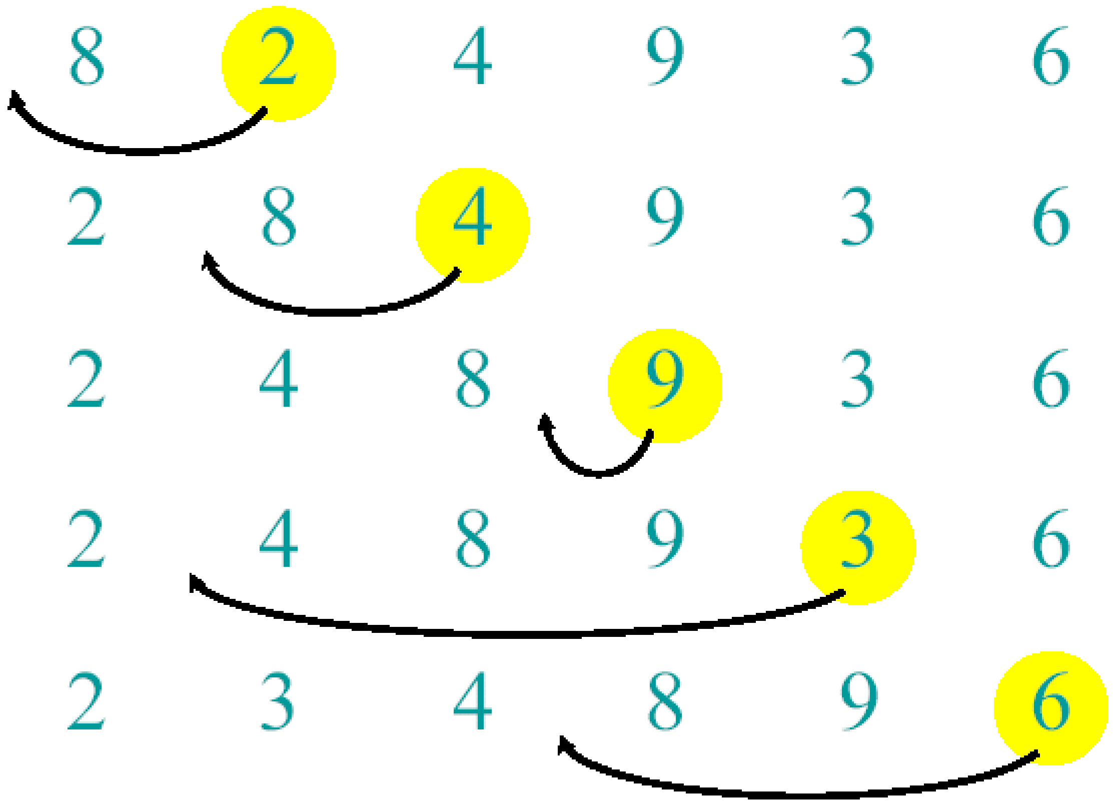
Example of Insertion Sort



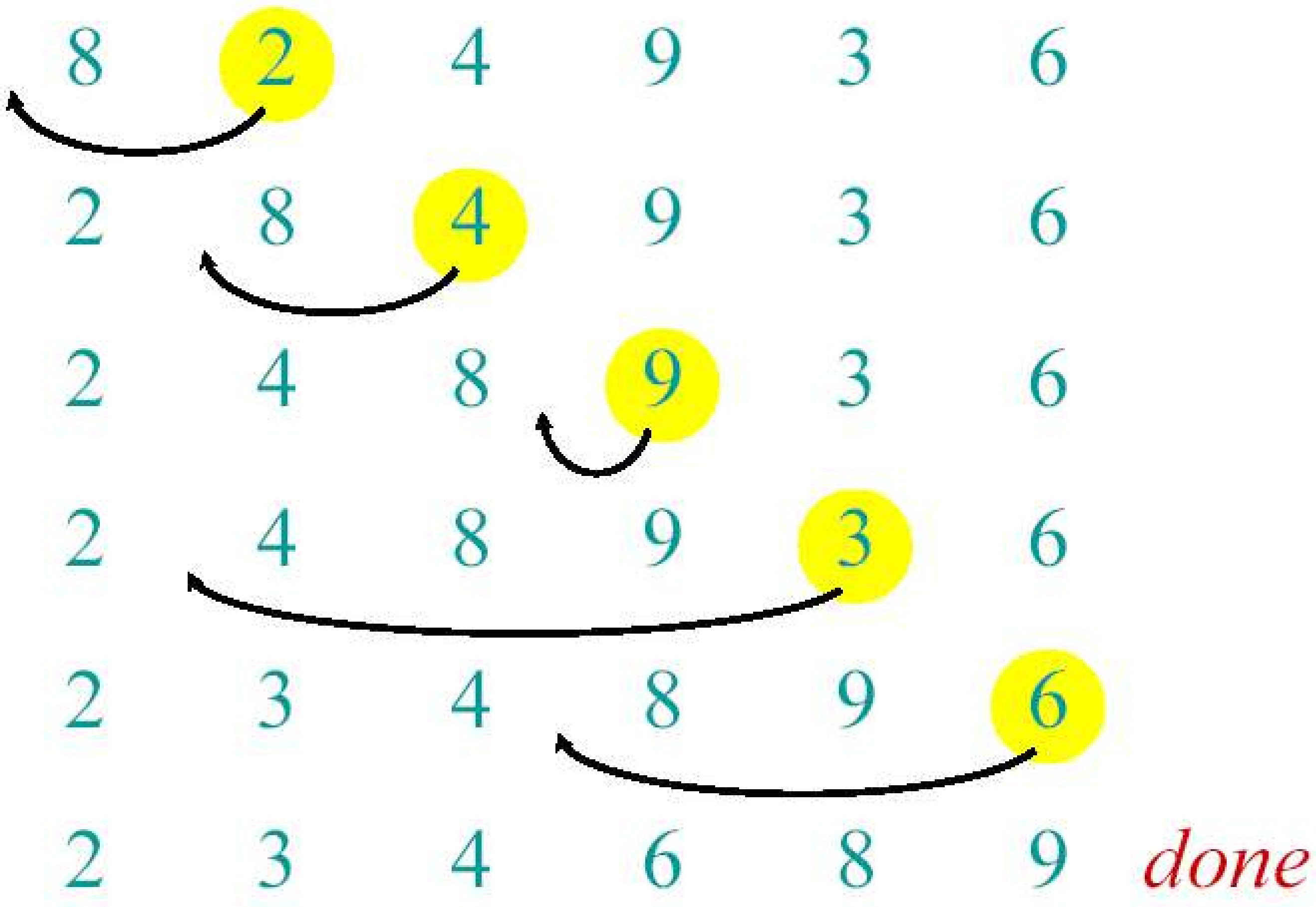
Example of Insertion Sort



Example of Insertion Sort



Example of Insertion Sort



Machine-independent time: An example

A *pseudocode* for insertion sort (INSERTION SORT).

```
INSERTION-SORT(A)
1  for $j \leftarrow 2$ to $length[A]$
2        do $key \leftarrow A[j]$
3        $\nabla$ Insert A[j] into the sortted sequence A[1,..., j-1].
4        $i \leftarrow j - 1$
5        while $i > 0$ and $A[i] > key$
6              do $A[i+1] \leftarrow A[i]$
7                 $i \leftarrow i - 1$
8     $A[i+1] \leftarrow key$
```

Analysis of INSERTION-SORT(contd.)

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

Analysis of INSERTION-SORT (contd.)

The total running time is

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

Analysis of INSERTION-SORT(contd.)

The best case: The array is already sorted.
(t_j =1 for j=2,3, ...,n)

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$

$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8).$$

Analysis of INSERTION-SORT(contd.)

- The worst case: The array is reverse sorted

  (t_j =j for j=2,3, ...,n).

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

$$T(n) = c_1 n + c_2(n-1) + c_5(n(n+1)/2-1)$$

$$+ c_6(n(n-1)/2) + c_7(n(n-1)/2) + c_8(n-1)$$

$$= (c_5/2 + c_6/2 + c_7/2)n^2 + (c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8)n$$

$$T(n) = an^2 + bn + c$$

Shell Sort

- Invented by Donald Shell in 1959.
- 1st algorithm to break the quadratic time barrier but few years later, a sub quadratic time bound was proven
- Shell sort works by comparing elements that are distant rather than adjacent elements in an array.
- Shellsort uses a sequence h_1, h_2, …, h_t called the ***increment sequence***. Any increment sequence is fine as long as $h_1 = 1$ and some other choices are better than others.
- Shell sort makes multiple passes through a list and sorts a number of equally sized sets using the insertion sort.
- Shell sort improves on the efficiency of insertion sort by *quickly* shifting values to their destination.

Shell sort

- Shell sort is also known as ***diminishing increment sort***.
- The distance between comparisons decreases as the sorting algorithm runs until the last phase in which adjacent elements are compared
- After each phase and some increment h_k, for every ***i***, we have $a[\,i\,] \leq a[\,i + h_k\,]$ all elements spaced h_k apart are sorted.
- The file is said to be h_k – sorted.

Empirical Analysis of Shell sort

Advantage:

- Advantage of Shellsort is that its only efficient for medium size lists. For bigger lists, the algorithm is not the best choice. Fastest of all O(N^2) sorting algorithms.
- 5 times faster than the bubble sort and a little over twice as fast as the insertion sort, its closest competitor.

Empirical Analysis of Shell sort

Disadvantage:

- Disadvantage of Shellsort is that it is a complex algorithm and its not nearly as efficient as the merge, heap, and quick sorts.
- The shell sort is still significantly slower than the merge, heap, and quick sorts, but its relatively simple algorithm makes it a good choice for sorting lists of less than 5000 items unless speed important. It's also an excellent choice for repetitive sorting of smaller lists.

Shell sort Best Case

- Best Case: The best case in the shell sort is when the array is already sorted in the right order. The number of comparisons is less.

Shell sort Worst Case

- The running time of Shellsort depends on the choice of increment sequence.
- The problem with Shell’s increments is that pairs of increments are not necessarily relatively prime and smaller increments can have little effect.

Shell sort Example

Sort: 18   32   12   5   38   33   16   2

8 Numbers to be sorted, Shell's increment will be floor(n/2)

*** floor(8/2) ➔ floor(4) = 4**

increment 4:   **1**   **2**   **3**   **4**   (visualize underlining)

18   32   12   5   38   33   16   2

Step 1) Only look at 18 and 38 and sort in order ;
18 and 38 stays at its current position because they are in order.

Step 2) Only look at **32** and **33** and sort in order ;
32 and **33** stays at its current position because they are in order.

Shell sort Example

Sort: 18   32   12   5   38   33   16   2

8 Numbers to be sorted, Shell's increment will be floor(n/2)

*** floor(8/2) ➔ floor(4) = 4**

increment 4:   **1**   **2**   **3**   **4**   (visualize underlining)

18   32   12   5   38   33   16   2

Step **3**) Only look at **12** and **16** and sort in order ;
12 and **16** stays at its current position because they are in order.

Step **4**) Only look at **5** and **2** and sort in order ;
2 and **5** need to be switched to be in order.

Shell sort Example (contd..)

Sort: 18 32 12 5 38 33 16 2

*** floor(2/2) ➔ floor(1) = 1**

increment 1: 1

12	**2**	**16**	**5**	**18**	**32**	**38**	**33**
2	**5**	**12**	**16**	**18**	**32**	**33**	**38**

The last increment or phase of Shellsort is basically an Insertion Sort algorithm.

Divide and Conquer Problem

- In **divide and conquer approach**, a problem is divided into smaller problems, then the smaller problems are solved independently, and finally the solutions of smaller problems are combined into a solution for the large problem.

- Generally, divide-and-conquer algorithms have three parts −
  - **Divide the problem** into a number of sub-problems that are smaller instances of the same problem.
  - **Conquer the sub-problems** by solving them recursively. If they are small enough, solve the sub-problems as base cases.
  - **Combine the solutions** to the sub-problems into the solution for the original problem.
- **Application of Divide and Conquer Approach**

  Finding the maximum and minimum of a sequence of numbers

  Strassen’s matrix multiplication

  Merge sort

  Binary search

  Quick Sort

Quick Sort

- Fastest known sorting algorithm in practice
- Average case: O(N log N) (we don’t prove it)
- Worst case: $O(N^2)$
    - But, the worst case seldom happens.
- Another divide-and-conquer recursive algorithm, like merge sort

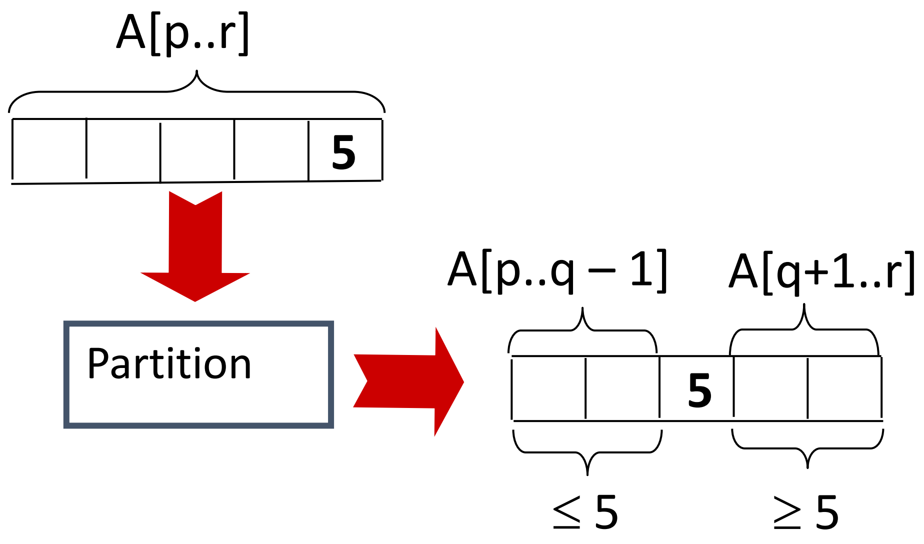
Quick Sort

- Follows the **divide-and-conquer** paradigm.
- ***Divide*: **Partition (separate) the array $A[p..r]$ into two (possibly empty) sub arrays $A[p..q–1]$ and $A[q+1..r]$.
    - Each element in $A[p..q–1] < A[q]$.
    - $A[q] <$ each element in $A[q+1..r]$.
    - Index q is computed as part of the partitioning procedure.
- ***Conquer*:** Sort the two sub arrays by recursive calls to quick sort.
- ***Combine*:** The sub arrays are sorted in place – no work is needed to combine them.
- How do the divide and combine steps of quick sort compare with those of merge sort?

Pseudocode

```
Quicksort(A, p, r)
    if p < r then
        q := Partition(A, p, r);
        Quicksort(A, p, q – 1);
        Quicksort(A, q + 1, r)
```

```
Partition(A, p, r)
    x, i := A[r], p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
            A[i] ↔ A[j]
    A[i + 1] ↔ A[r];
    return i + 1
```



Example

```
                      p                        r
initially:          2  5  8  3  9  4  1  7  10  6      note: pivot (x) = 6
                  i  j

next iteration:         2  5  8  3  9  4  1  7  10  6
                    i  j

next iteration:         2  5  8  3  9  4  1  7  10  6
                       i   j

next iteration:         2  5  8  3  9  4  1  7  10  6
                       i      j

next iteration:         2  5  3  8  9  4  1  7  10  6
                          i      j
```

```
Partition(A, p, r)
    x, i := A[r], p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
        A[i] ↔ A[j]
    A[i + 1] ↔ A[r];
    return i + 1
```

Example (contd...)

```
next iteration:   2  5  3  8  9  4  1  7  10 6
                  i     j
next iteration:   2  5  3  8  9  4  1  7  10 6
                   i       j
next iteration:   2  5  3  4  9  8  1  7  10 6
                    i       j
next iteration:   2  5  3  4  1  8  9  7  10 6
                        i       j
next iteration:   2  5  3  4  1  8  9  7  10 6
                        i          j
next iteration:   2  5  3  4  1  8  9  7  10 6
                        i             j
after final swap: 2  5  3  4  1  6  9  7  10 8
                           i              j
```

```
Partition(A, p, r)
    x, i := A[r], p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
        A[i] ↔ A[j]
    A[i + 1] ↔ A[r];
    return i + 1
```

Partitioning

- Select the last element A[*r*] in the subarray *A*[*p*..*r*] as the *pivot* – the element around which to partition.
- As the procedure executes, the array is partitioned into four (possibly empty) regions.
  1. *A*[*p*..*i*] — All entries in this region are < ***pivot***.
  2. *A*[*i*+1..*j* – 1] — All entries in this region are > ***pivot***.
  3. *A*[*r*] = *pivot*.
  4. *A*[*j*..*r* – 1] — Not known how they compare to *pivot*.
- The above hold before each iteration of the *for* loop, and constitute a *loop invariant*. (4 is not part of the loopi.)

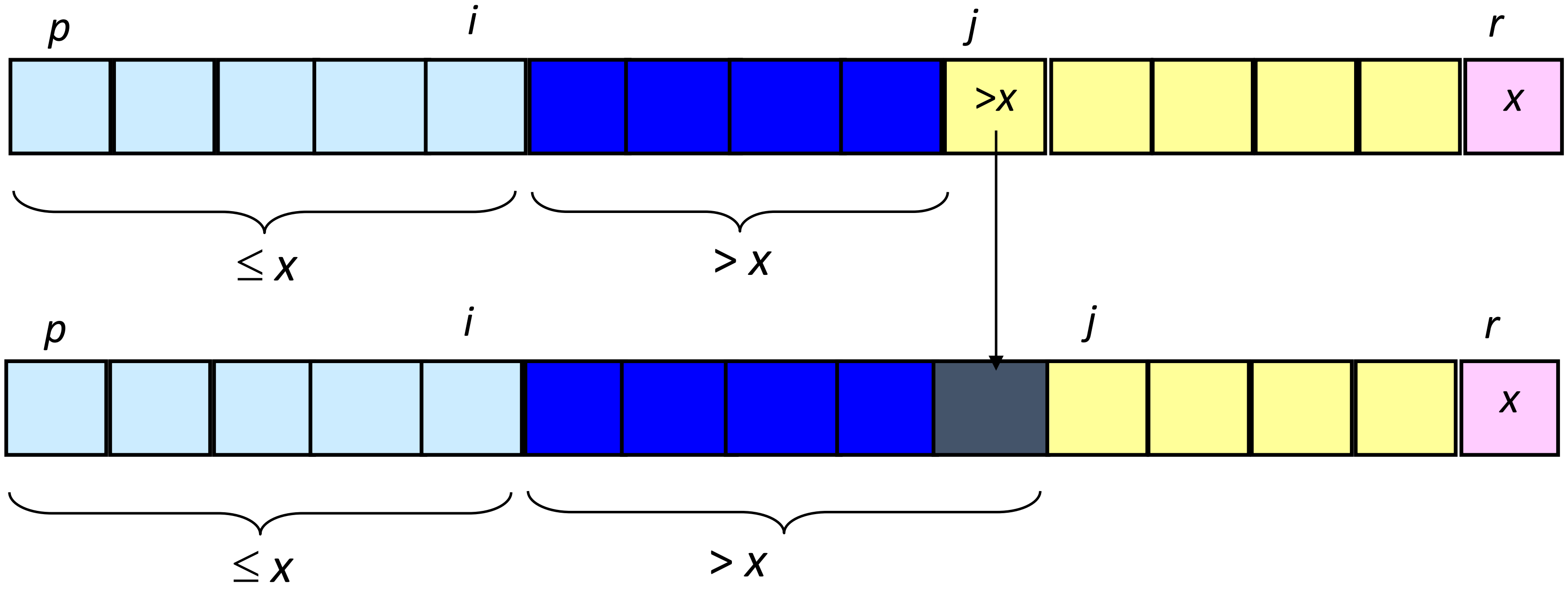
Correctness of Partition

- Use loop invariant.
- **Initialization:**
  - Before first iteration
    - $A[p..i]$ and $A[i+1..j-1]$ are empty – Conds. 1 and 2 are satisfied (trivially).
    - r is the index of the *pivot*
      - Cond. 3 is satisfied.
- **Maintenance:**
  - **Case 1:** $A[j] > x$
    - Increment j only.
    - Loop Invariant is maintained.

```
Partition(A, p, r)
    x, i := A[r], p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
        A[i] ↔ A[j]
    A[i + 1] ↔ A[r];
    return i + 1
```

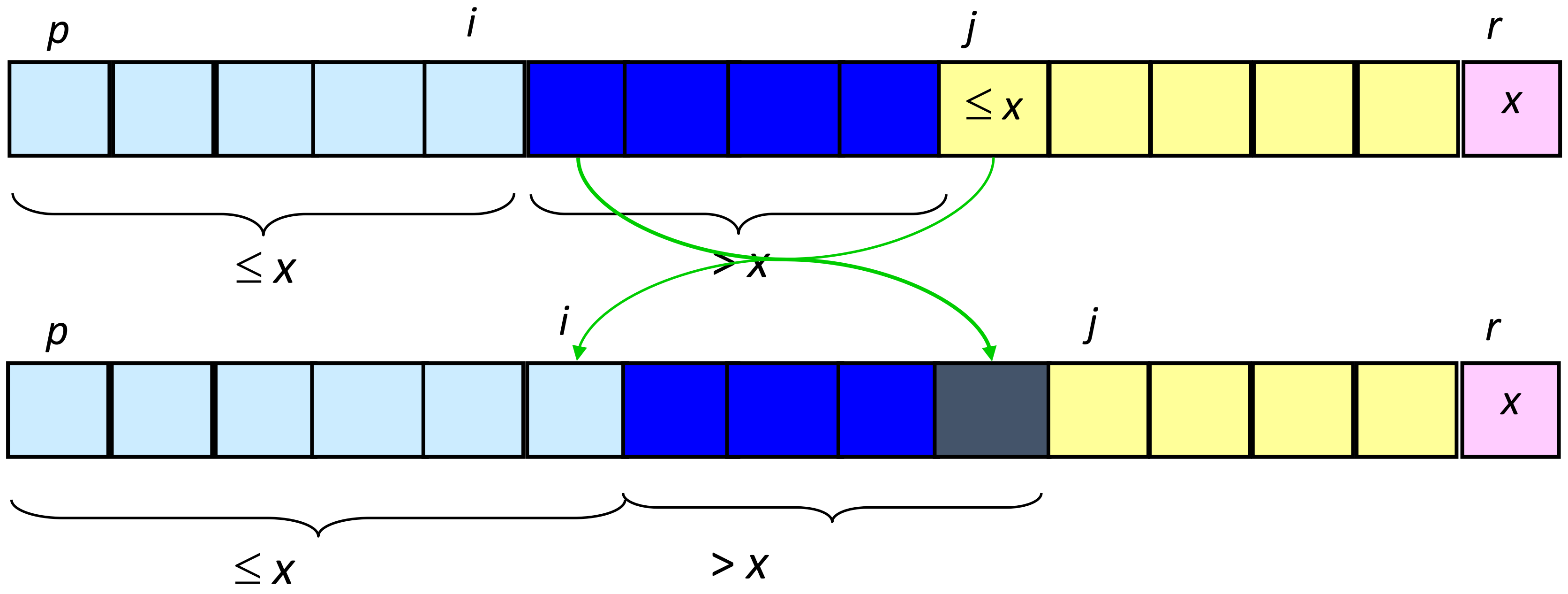
Correctness of Partition

Case 1:



Correctness of Partition

- **Case 2:** $A[j] \leq x$
  - Increment i
  - Swap $A[i]$ and $A[j]$
    - Condition 1 is maintained.
- Increment j
  - Condition 2 is maintained.
- $A[r]$ is unaltered.
  - Condition 3 is maintained.



Correctness of Partition

- **Termination:**
  - When the loop terminates, $j = r$, so all elements in A are partitioned into one of the three cases:
    - $A[p..i] \leq$ ***pivot***
    - $A[i+1..j-1] >$ ***pivot***
    - $A[r] =$ ***pivot***
- The last two lines swap $A[i+1]$ and $A[r]$.
  - *Pivot* moves from the end of the array to between the two subarrays.
  - Thus, procedure *partition* correctly performs the divide step.

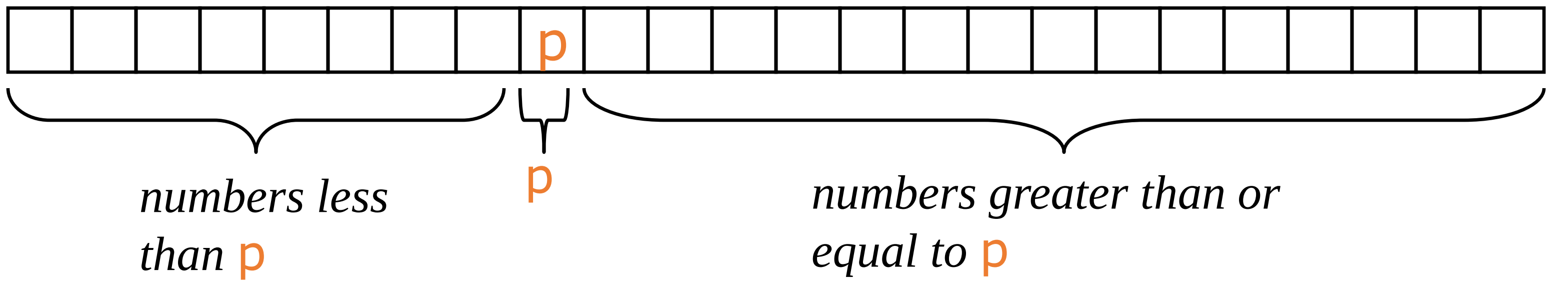
Complexity of Partition

- PartitionTime(n) is given by the number of iterations in the *for* loop.
- $\Theta(n)$: $n = r - p + 1$.

```
Partition(A, p, r)
    x, i := A[r], p – 1;
    for j := p to r – 1 do
        if A[j] $\leq$ x then
            i := i + 1;
        A[i] $\leftrightarrow$ A[j]
    A[i + 1] $\leftrightarrow$ A[r];
    return i + 1
```

Partitioning in Quick sort

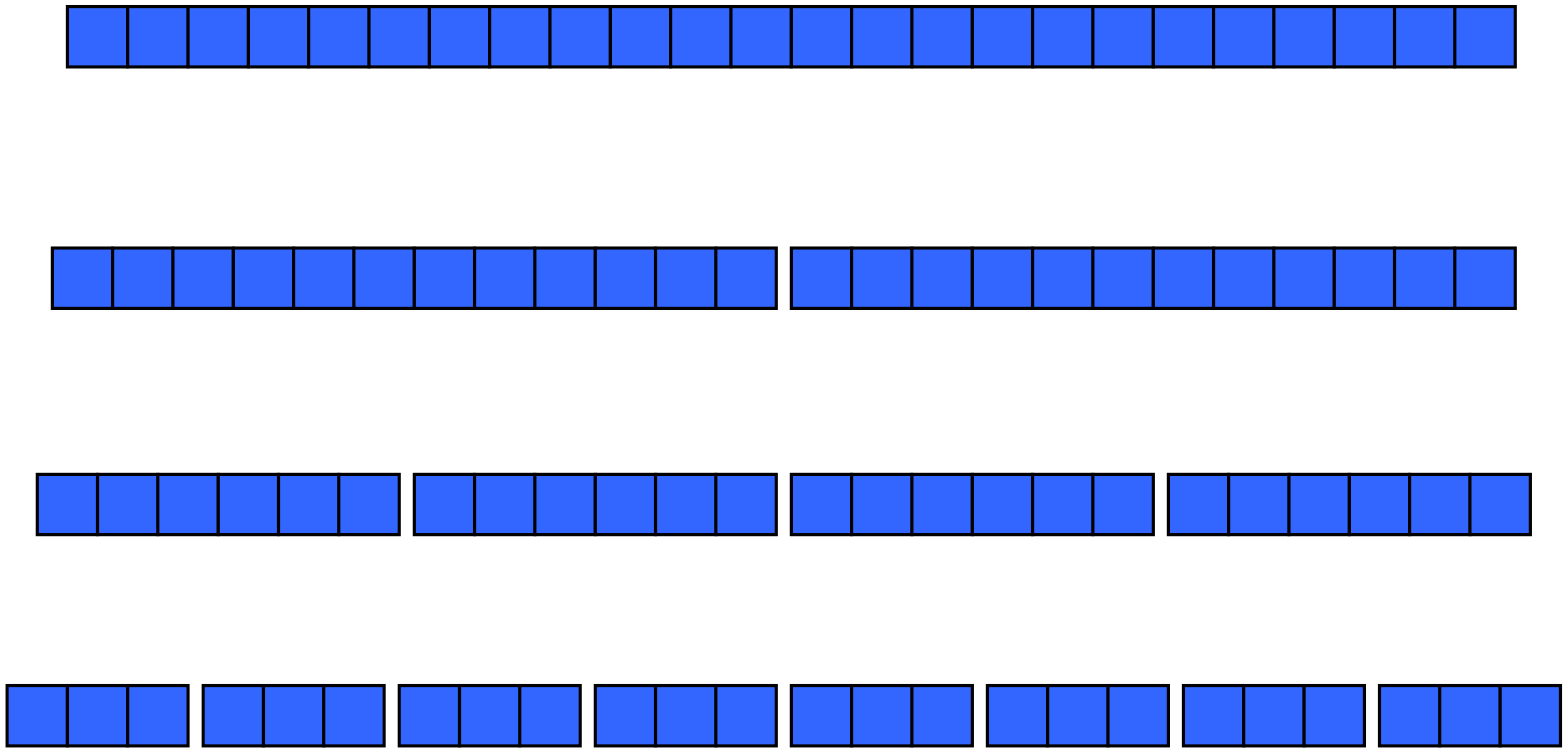
- A key step in the Quick sort algorithm is partitioning the array
  - We choose some (any) number p in the array to use as a pivot
  - We partition the array into three parts:



Analysis of quick sort—best case

- Suppose each partition operation divides the array almost exactly in half
- Then the depth of the recursion in $\log_2 n$
    - Because that's how many times we can halve n
- We note that
    - Each partition is linear over its subarray
    - All the partitions at one level cover the array

Partitioning at various levels



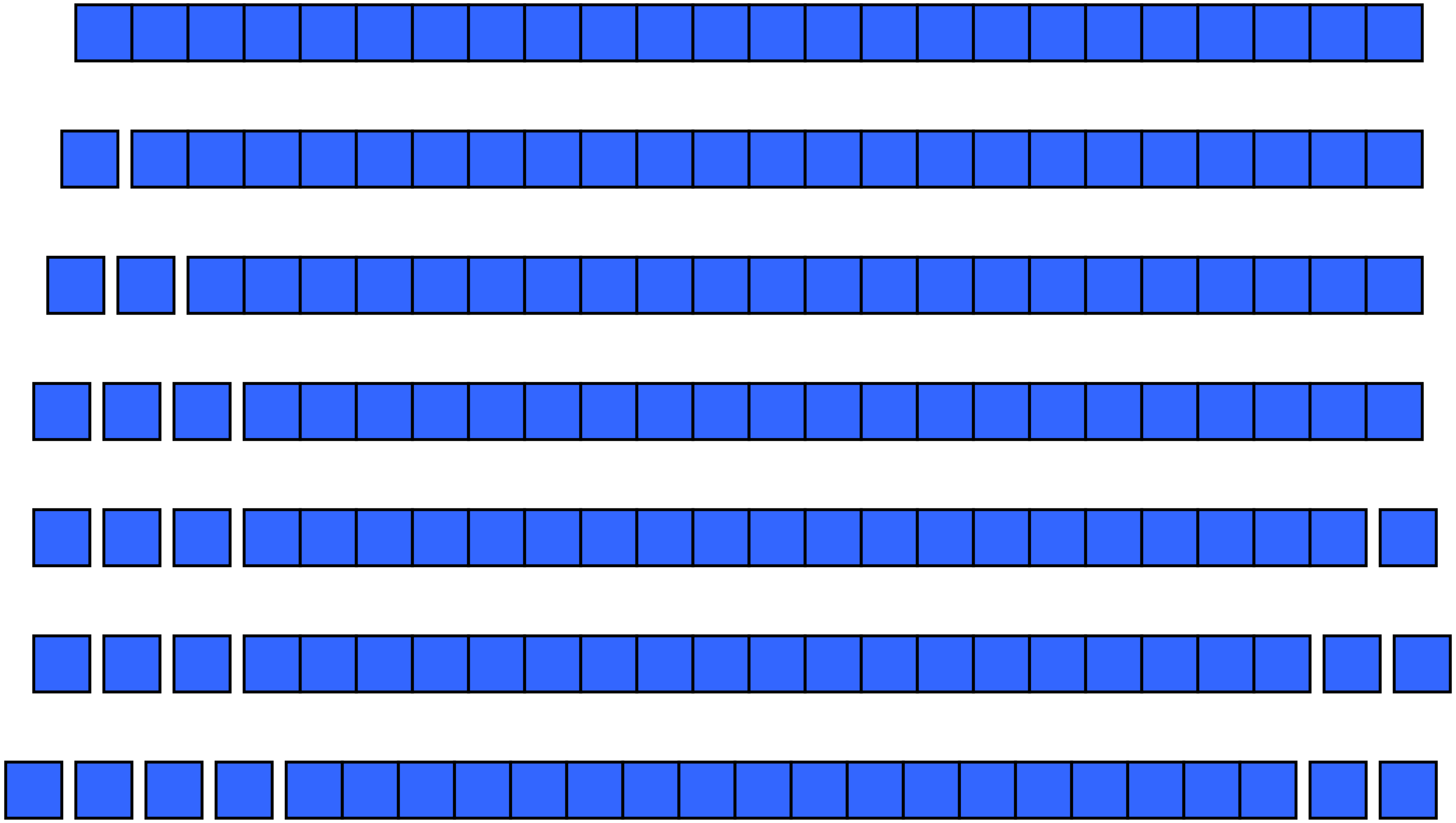
Best Case Analysis

- We cut the array size in half each time
- So the depth of the recursion in $\log_2 n$
- At each level of the recursion, all the partitions at that level do work that is linear in n
- $O(\log_2 n) * O(n) = O(n \log_2 n)$
- Hence in the best case, quicksort has time complexity $O(n \log_2 n)$
- What about the worst case?

Worst case

- In the worst case, partitioning always divides the size n array into these three parts:
  - A length one part, containing the pivot itself
  - A length zero part, and
  - A length n-1 part, containing everything else
- We don’t recur on the zero-length part
- Recurring on the length n-1 part requires (in the worst case) recurring to depth n-1

Worst case partitioning



Worst case for quick sort

- In the worst case, recursion may be n levels deep (for an array of size n)
- But the partitioning work done at each level is still n
- $O(n) * O(n) = O(n^2)$
- So worst case for Quicksort is $O(n^2)$
- When does this happen?
    - There are many arrangements that *could* make this happen
    - Here are two common cases:
        - When the array is already sorted
        - When the array is *inversely* sorted (sorted in the opposite order)

Typical case for quick sort

- If the array is sorted to begin with, Quick sort is terrible: $O(n^2)$
- It is possible to construct other bad cases
- However, Quick sort is *usually* $O(n \log_2 n)$
- The constants are so good that Quick sort is generally the faster algorithm.
- Most real-world sorting is done by Quick sort

Picking a better pivot

- Before, we picked the *first* element of the sub array to use as a pivot
  - If the array is already sorted, this results in $O(n^2)$ behavior
  - It's no better if we pick the *last* element
- We could do an *optimal* quick sort (guaranteed $O(n \log n)$) if we always picked a pivot value that exactly cuts the array in half
  - Such a value is called a **median**: half of the values in the array are larger, half are smaller
  - The easiest way to find the median is to *sort* the array and pick the value in the middle (!)

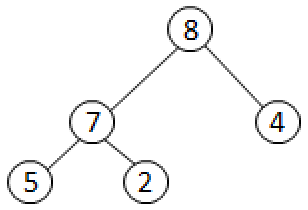
Quicksort for Small Arrays

- For very small arrays (N<= 20), quicksort does not perform as well as insertion sort
- A good cutoff range is N=10
- Switching to insertion sort for small arrays can save about 15% in the running time

Heap Sort

- Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case running time. Heap sort involves building a **Heap** data structure from the given array and then utilizing the Heap to sort the array.
- Def: A **heap** is a <u>nearly complete</u> binary tree with the following two properties:
    - **Structural property:** all levels are full, except possibly the last one, which is filled from left to right
    - **Order (heap) property:** for any node x

$$\text{Parent}(x) \geq x$$



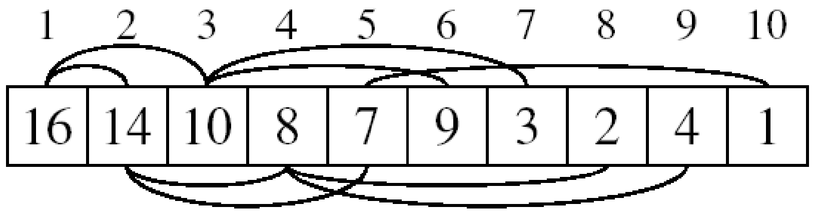
Heap

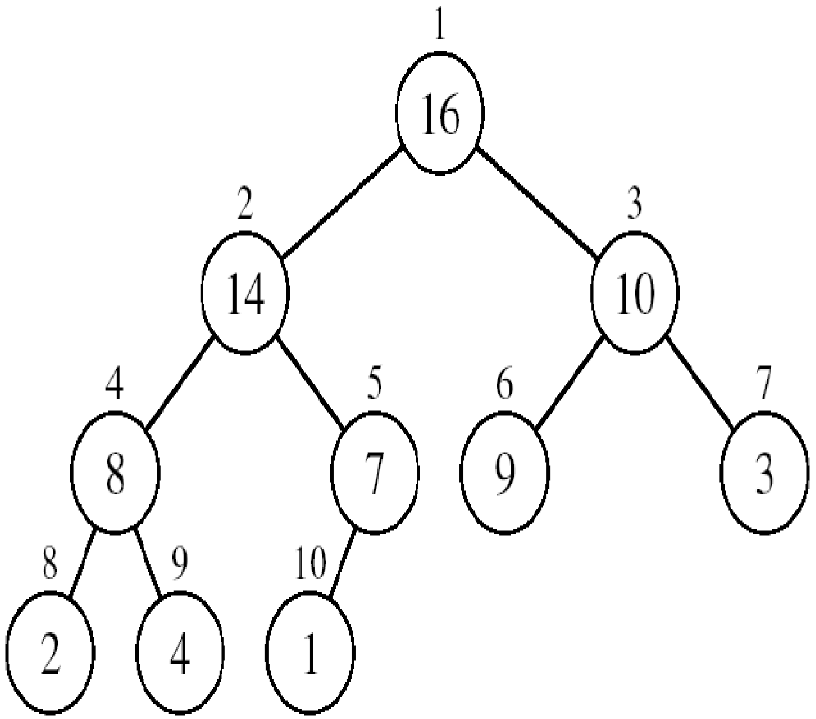
From the heap property, it follows that:
"The root is the maximum element of the heap!"

A heap is a binary tree that is filled in order

Array Representation of Heaps

- A heap can be stored as an array *A*.
    - Root of tree is A[1]
    - Left child of A[i] = A[2i]
    - Right child of A[i] = A[2i + 1]
    - Parent of A[i] = A[⌊i/2⌋]
    - Heapsize[A] ≤ length[A]
- The elements in the subarray A[(⌊n/2⌋+1) .. n] are leaves





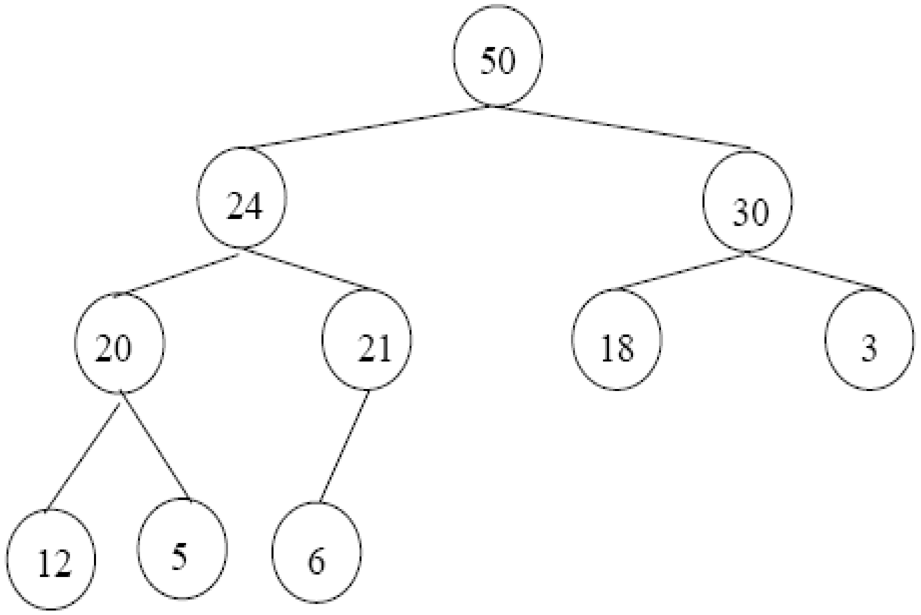
Heap Types

- **Max-heaps** (largest element at root), have the *max-heap property:*
    - for all nodes i, excluding the root:
    - A[PARENT(i)] ≥ A[i]

- **Min-heaps** (smallest element at root), have the *min-heap property:*
    - for all nodes i, excluding the root:
    - A[PARENT(i)] ≤ A[i]

Adding/Deleting Nodes

- New nodes are always inserted at the bottom level (left to right)
- Nodes are removed from the bottom level (right to left)

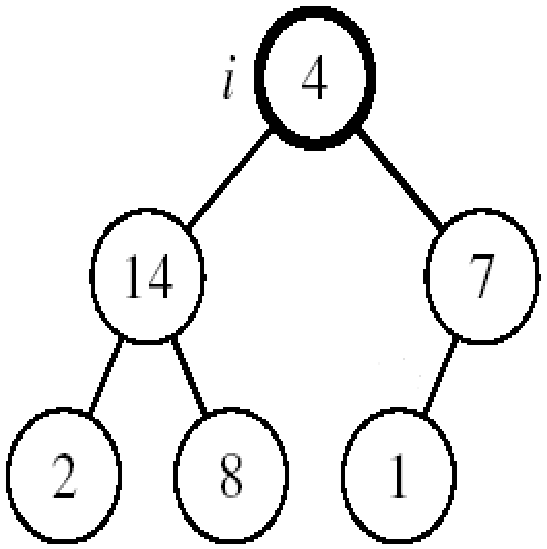


Operations on Heaps

- Maintain/Restore the max-heap property
    - MAX-HEAPIFY
- Create a max-heap from an unordered array
    - BUILD-MAX-HEAP
- Sort an array in place
    - HEAPSORT
- Priority queues

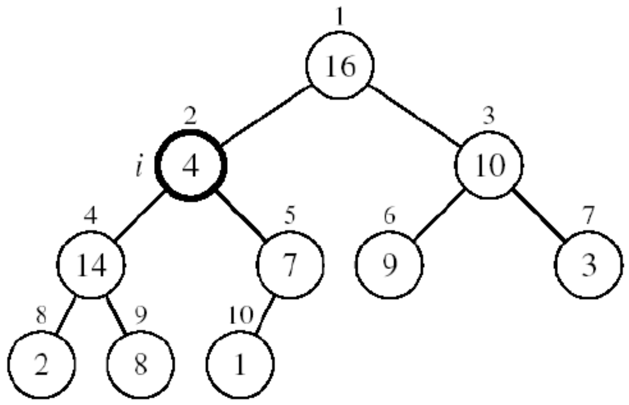
Maintaining the Heap Property

- Suppose a node is smaller than a child
  - Left and Right subtrees of i are max-heaps
- To eliminate the violation:
  - Exchange with larger child
  - Move down the tree
  - Continue until node is not smaller than children



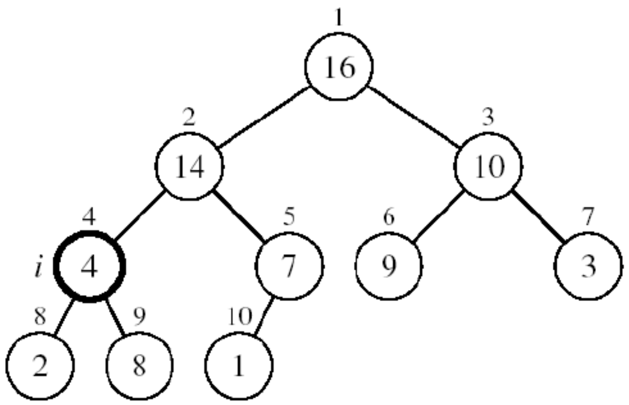
Example

MAX-HEAPIFY(A, 2, 10)



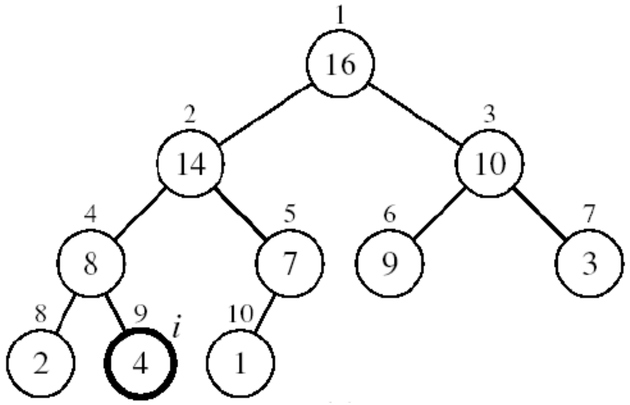
A[2] violates the heap property

A[2] ↔ A[4]



A[4] violates the heap property

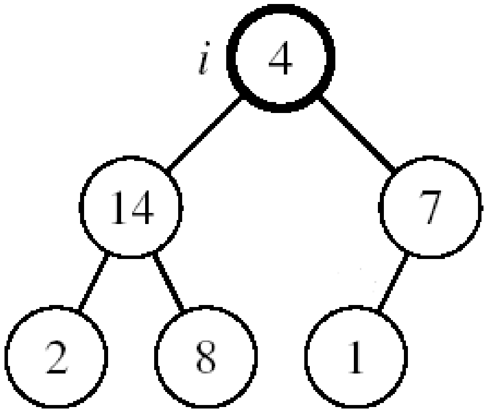
A[4] ↔ A[9]



Heap property restored

Maintaining the Heap Property

- Assumptions:
  - Left and Right subtrees of i are max-heaps
  - A[i] may be smaller than its children

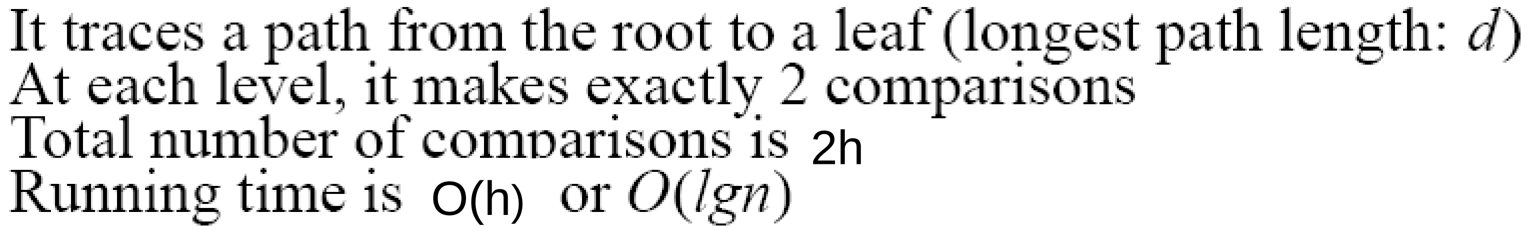


Alg: MAX-HEAPIFY(A, i, n)

```
1.  l ← LEFT(i)
2.  r ← RIGHT(i)
3.  if l ≤ n and A[l] > A[i]
4.     then largest ←l
5.     else largest ←i
6.  if r ≤ n and A[r] > A[largest]
7.     then largest ←r
8.  if largest ≠ i
9.     then exchange A[i] ↔ A[largest]
10.            MAX-HEAPIFY(A, largest, n)
```

MAX-HEAPIFY Running Time

- Intuitively:


It traces a path from the root to a leaf (longest path length: d)
At each level, it makes exactly 2 comparisons
Total number of comparisons is 2h
Running time is O(h) or $O(lgn)$

- Running time of MAX-HEAPIFY is O(lg n)

- Can be written in terms of the height of the heap, as being O(h)

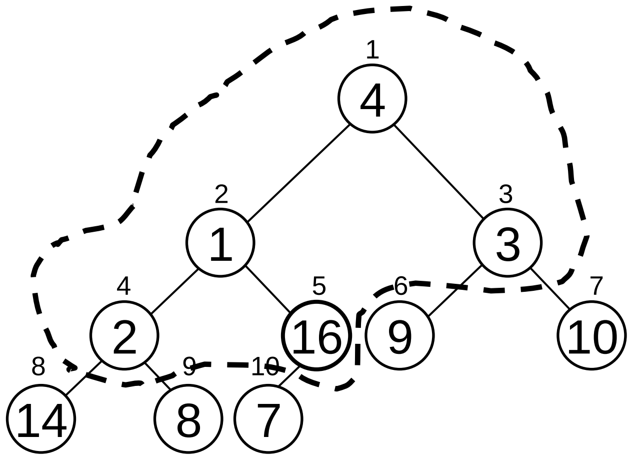
  - Since the height of the heap is ⌊lg n⌋

Building a Heap

- Convert an array A[1 … n] into a max-heap (n = length[A])
- The elements in the subarray A[(⌊n/2⌋+1) .. n] are leaves
- Apply MAX-HEAPIFY on elements between 1 and ⌊n/2⌋

Alg: BUILD-MAX-HEAP(A)

```
1.    n = length[A]
2.    for i ← ⌊n/2⌋ downto 1
3.         do MAX-HEAPIFY(A, i, n)
```



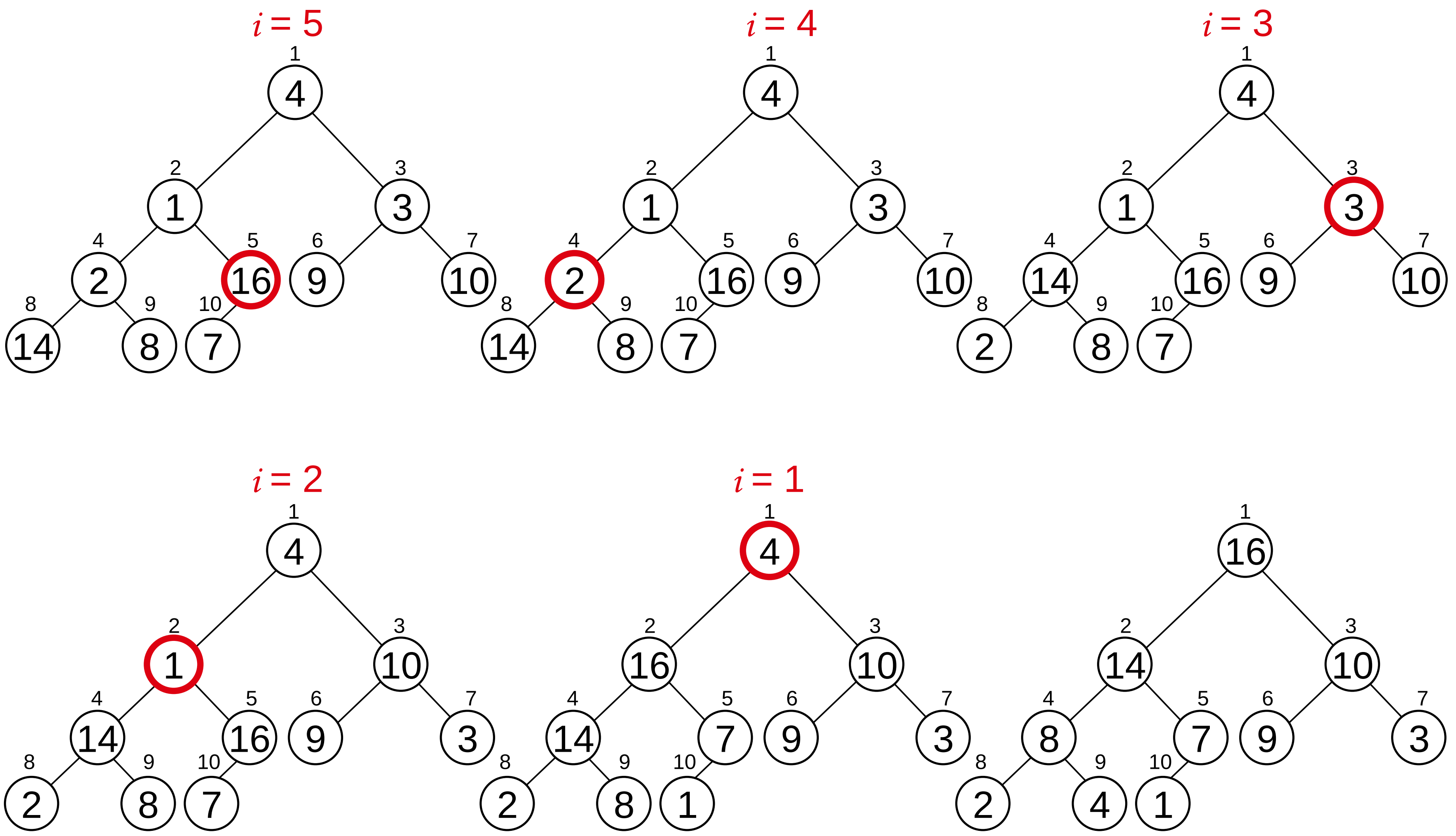
A:

4	1	3	2	16	9	10	14	8	7

Example:

A

4	1	3	2	16	9	10	14	8	7



Running Time of BUILD MAX HEAP

```
Alg: BUILD-MAX-HEAP(A)
1.  n = length[A]
2.  for i ← ⌊n/2⌋ downto 1
3.      do MAX-HEAPIFY(A, i, n)
```

$O(\lg n)$ $\left.\right\}$ $O(n)$

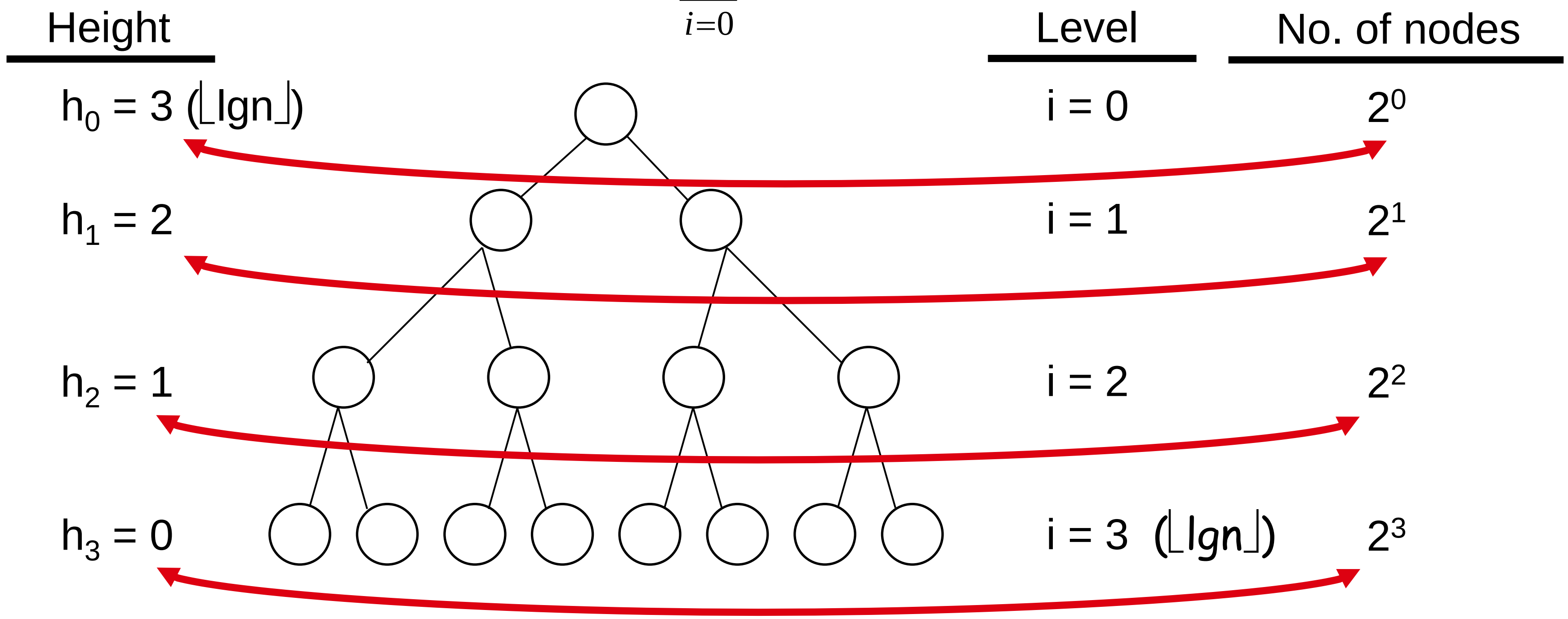
$\Rightarrow$ Running time: $O(n \lg n)$

- This is not an asymptotically tight upper bound

Running Time of BUILD MAX HEAP

- HEAPIFY takes O(h) ⇒ the cost of HEAPIFY on a node i is proportional to the height of the node i in the tree

$$\Rightarrow T(n) = \sum_{i=0}^{h} n_i h_i$$



$h_i = h - i$ height of the heap rooted at level i

$n_i = 2^i$ number of nodes at level i

$$= \sum_{i=0}^{h} 2^i (h - i)$$

Running Time of BUILD MAX HEAP

$$T(n) = \sum_{i=0}^{h} n_i h_i$$
Cost of HEAPIFY at level i $*$ number of nodes at that level

$$= \sum_{i=0}^{h} 2^i (h - i)$$
Replace the values of n_i and h_i computed before

$$= \sum_{i=0}^{h} \frac{h-i}{2^{h-i}} 2^h$$
Multiply by 2^h both at the nominator and denominator and write 2^i as $\frac{1}{2^{-i}}$

$$= 2^h \sum_{k=0}^{h} \frac{k}{2^k}$$
Change variables: k = h - i

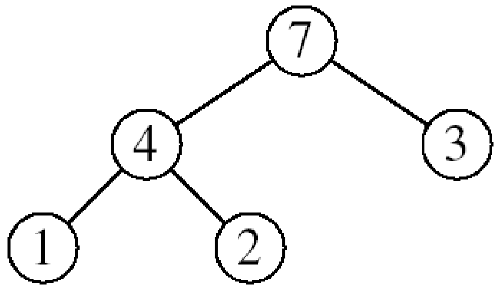
$$\leq n \sum_{k=0}^{\infty} \frac{k}{2^k}$$
The sum above is smaller than the sum of all elements to ∞ and h = lgn

$$= O(n)$$
The sum above is smaller than 2

Running time of BUILD-MAX-HEAP: T(n) = O(n)

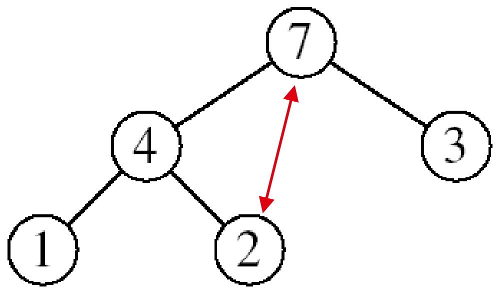
Heapsort

- Goal:
    - Sort an array using heap representations
- Idea:
    - Build a **max-heap** from the array
    - Swap the root (the maximum element) with the last element in the array
    - “Discard” this last node by decreasing the heap size
    - Call MAX-HEAPIFY on the new root
    - Repeat this process until only one node remains

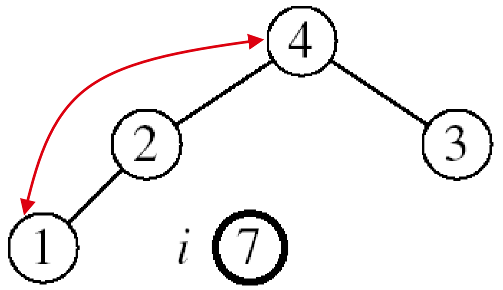


Example:

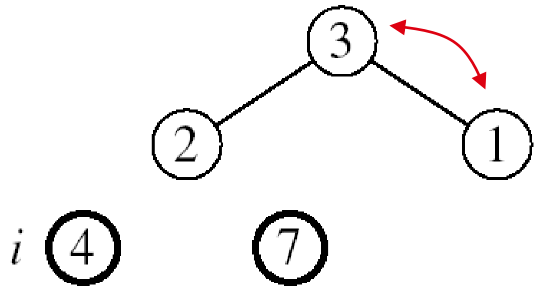
A=[7, 4, 3, 1, 2]



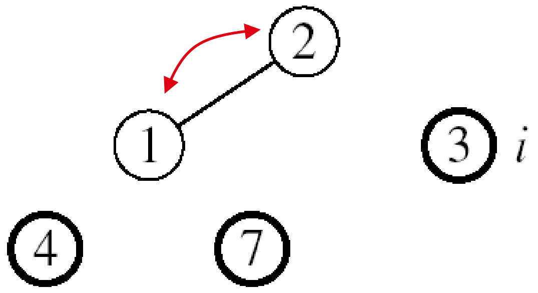
MAX-HEAPIFY(A, 1, 4)



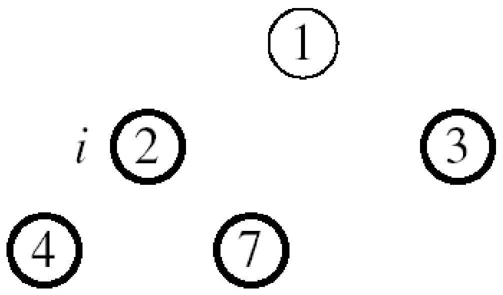
MAX-HEAPIFY(A, 1, 3)

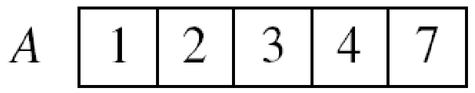


MAX-HEAPIFY(A, 1, 2)



MAX-HEAPIFY(A, 1, 1)





A	1	2	3	4	7

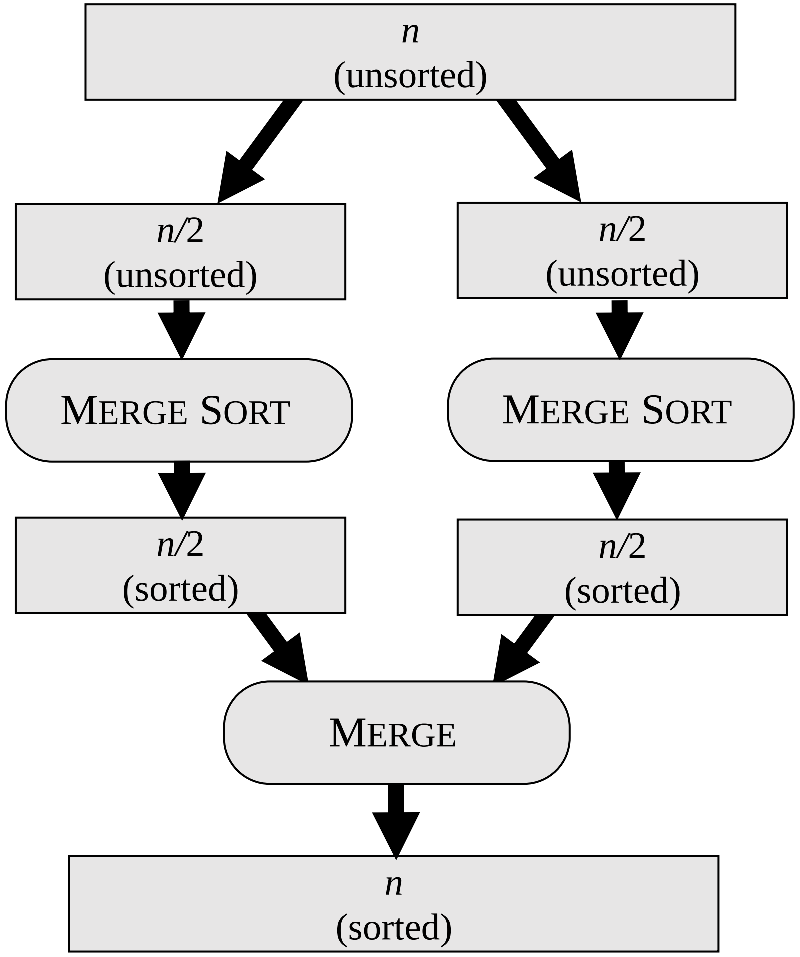
Algorithm: HEAPSORT*(A)*

```
1.  BUILD-MAX-HEAP(A)                                O(n)
2.  for i ← length[A] downto 2                     ⎫
3.      do exchange A[1] ↔ A[i]                    ⎬ n-1 times
4.          MAX-HEAPIFY(A, 1, i - 1)       O(lgn)  ⎭
```

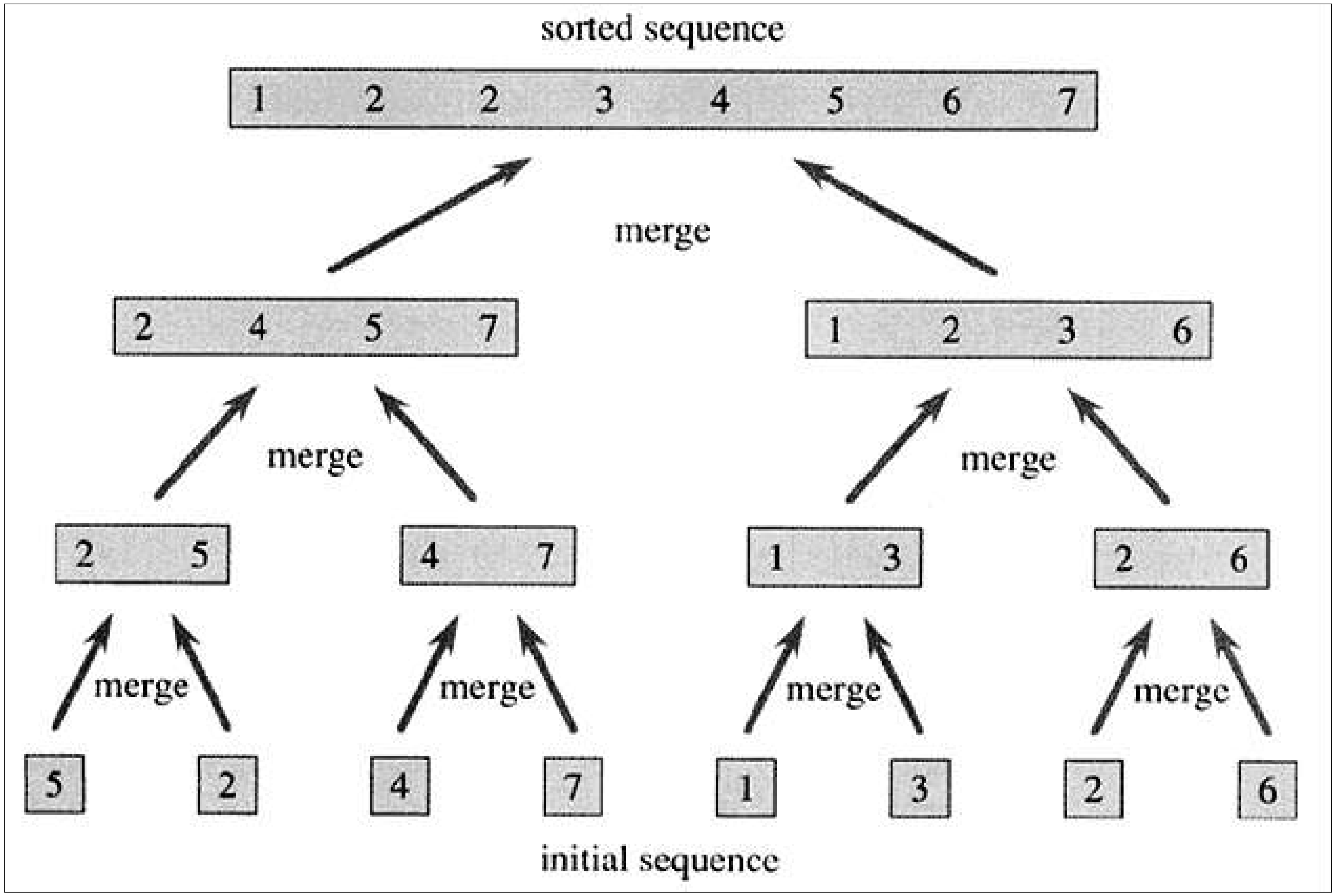
- Running time: O(nlgn) --- Can be shown to be Θ(nlgn)

Merge Sort

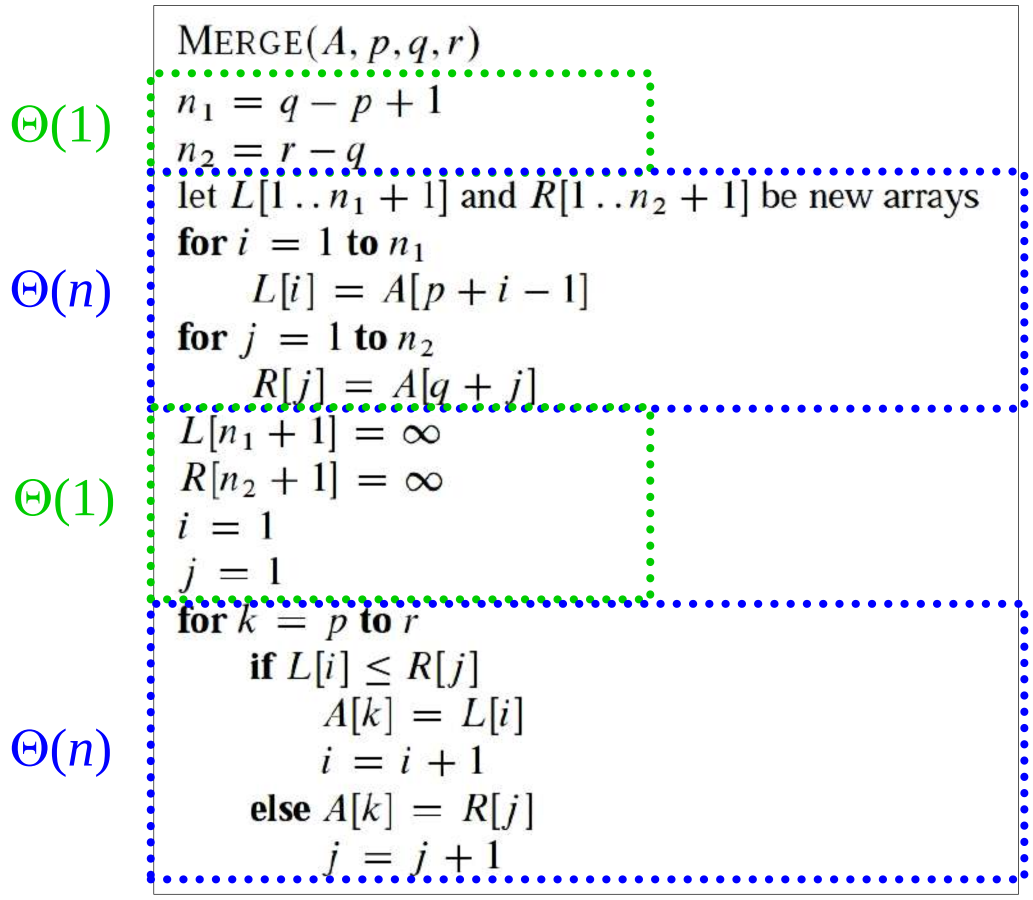
- ***Divide stage***: Split the *n*-element sequence into two subsequences of *n*/2 elements each
- ***Conquer stage***: Recursively sort the two subsequences
- ***Combine stage***: Merge the two sorted subsequences into one sorted sequence (the solution)



Merging Sorted Sequences



Merging Sorted Sequences



```
MERGE(A, p, q, r)
$\Theta(1)$
$n_1 = q - p + 1$
$n_2 = r - q$
$\Theta(n)$
let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays
for $i = 1$ to $n_1$
    $L[i] = A[p + i - 1]$
for $j = 1$ to $n_2$
    $R[j] = A[q + j]$
$\Theta(1)$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
$\Theta(n)$
for $k = p$ to $r$
    if $L[i] \le R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    else $A[k] = R[j]$
        $j = j + 1$
```

- Combines the sorted subarrays $A[p..q]$ and $A[q+1..r]$ into one sorted array $A[p..r]$
- Makes use of two working arrays L and R which initially hold copies of the two subarrays
- Makes use of sentinel value (∞) as last element to simplify logic

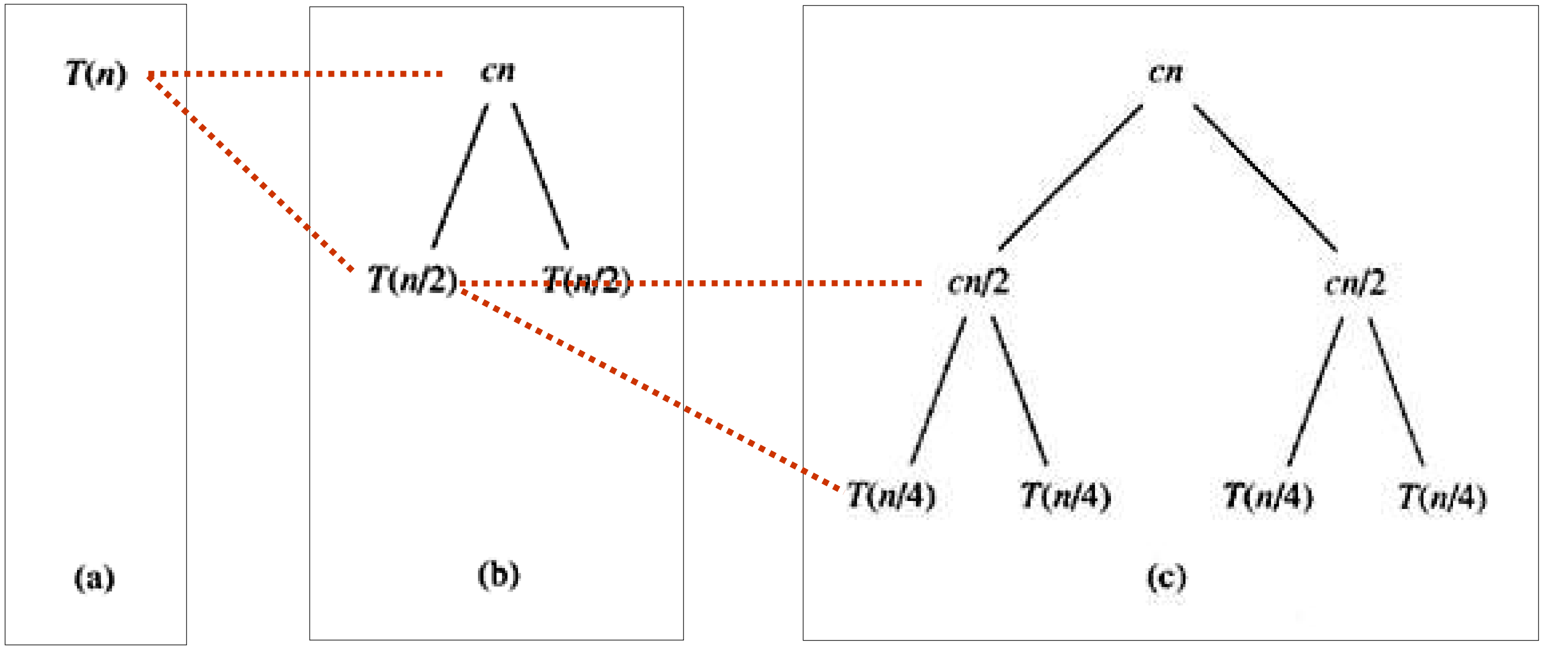
Merge Sort Algorithm

```
         MERGE-SORT(A, p, r)
         if $p < r$                               // check for base case
$\Theta(1)$        $q = \lfloor (p + r)/2 \rfloor$              // divide
$T(n/2)$       MERGE-SORT(A, p, q)              // conquer
$T(n/2)$       MERGE-SORT(A, q + 1, r)          // conquer
$\Theta(n)$        MERGE(A, p, q, r)                // combine
```

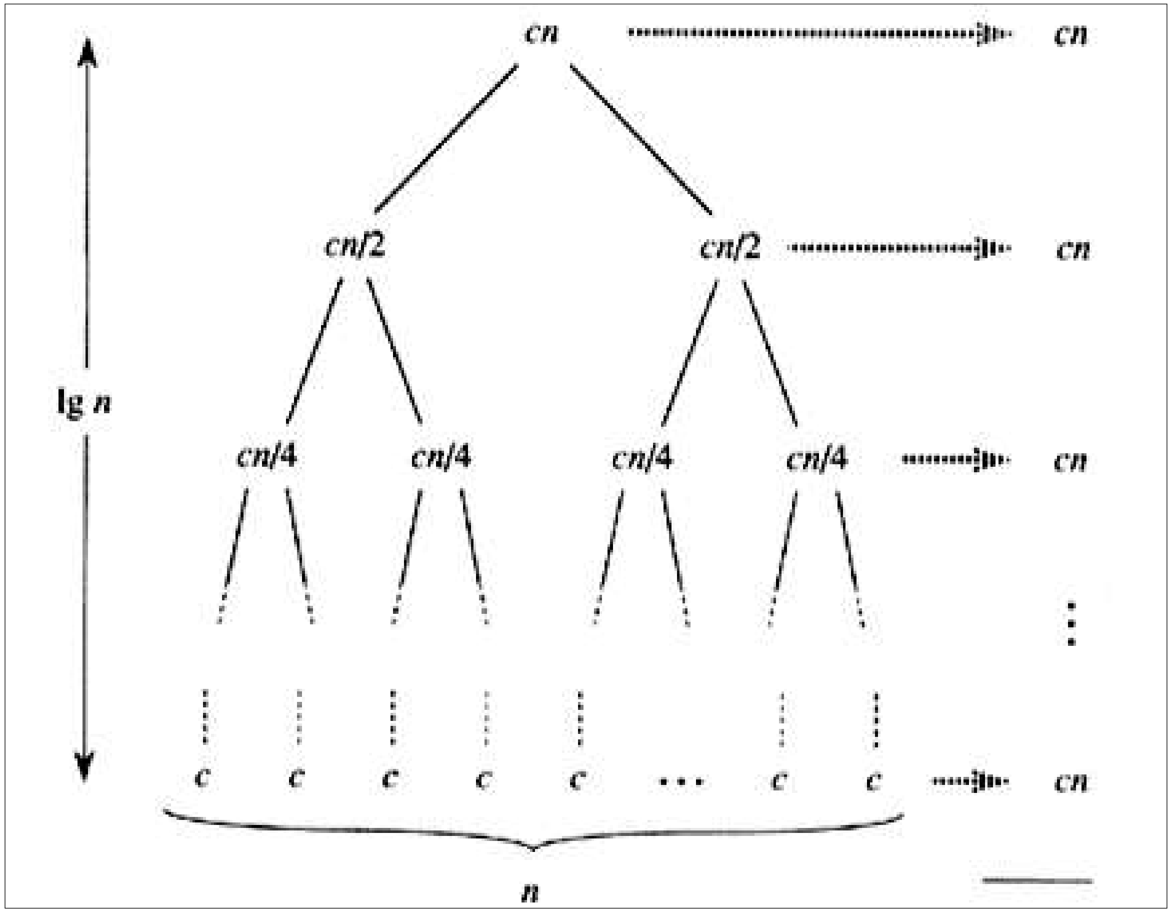
$$T(n) = 2T(n/2) + \Theta(n)$$

Analysis of Merge Sort

Analysis of recursive calls …



Analysis of Merge Sort



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

$T(n)$ is $\Theta(n \lg n)$

Merge Sort

Analysis

Let us consider, the running time of Merge-Sort as ***T(n)***. Hence,

$$T(n) = \begin{cases} c & if\, n \leqslant 1 \\ 2\,x\,T(\frac{n}{2}) + d\,x\,n & otherwise \end{cases}$$ where *c* and *d* are constants

Therefore, using this recurrence relation,

$$T(n) = 2^i T(\frac{n}{2^i}) + i.d.n$$

As, $i = log\,n,\ T(n) = 2^{log\,n} T(\frac{n}{2^{log\,n}}) + log\,n.d.n$

$$= c.n + d.n.log\,n$$

Therefore, $T(n) = O(n\,log\,n)$

Analysis of Different Sorting Algorithms

Comparison based sorting:

- **Bubble sort and Insertion sort**
  Average and worst case time complexity: n^2
  Best case time complexity: n when array is already sorted.
  Worst case: when the array is reverse sorted.
- **Selection sort**
  Best, average and worst case time complexity: n^2 which is independent of distribution of data.
- **Merge sort**
  Best, average and worst case time complexity: nlogn which is independent of distribution of data.

Analysis of Different sorting Algorithms Contd…

- **Heap sort**
  Best, average and worst case time complexity: nlogn which is independent of distribution of data.
- **Quick sort**
  It is a divide and conquer approach with recurrence relation: $T(n) = T(k) + T(n-k-1) + cn$
- Worst case: when the array is sorted or reverse sorted, the partition algorithm divides the array in two subarrays with 0 and n-1 elements. Therefore,
- $T(n) = T(0) + T(n-1) + cn$ Solving this we get, $T(n) = O(n^2)$
- Best case and Average case: On an average, the partition algorithm divides the array in two subarrays with equal size. Therefore,
- $T(n) = 2T(n/2) + cn$ Solving this we get, $T(n) = O(n\log n)$

Summary of time complexity of comparison-based Sorting Techniques

Here we will see some sorting methods. We will see few of them. Some sorting techniques are comparison based sort, some are non-comparison based sorting technique. Comparison Based Soring techniques are bubble sort, selection sort, insertion sort, Merge sort, quicksort, heap sort etc. These techniques are considered as comparison based sort because in these techniques the values are compared, and placed into sorted position in different phases. Here we will see time complexity of these techniques.

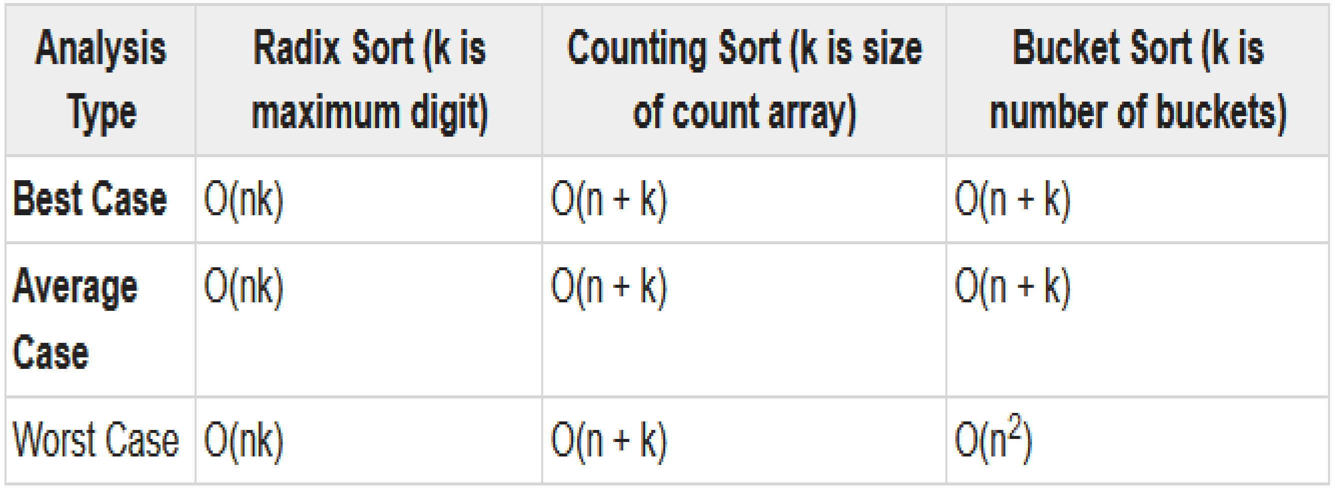
Analysis Type	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort	Quick Sort	Heap Sort
Best Case	$O(n^2)$	$O(n^2)$	O(n)	O(log n)	O(log n)	O(logn)
Average Case	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(log n)	O(log n)	O(log n)
Worst Case	$O(n^{2)}$	$O(n^2)$	$O(n^2)$	O(log n)	$O(n^2)$	O(log n)

Analysis of Different Sorting Algorithms :Non Comparison Based

Non-comparison based sorting :
In non-comparison based sorting, elements of array are not compared with each other to find the sorted array.

- **Radix sort**
  Best, average and worst case time complexity: nk where k is the maximum number of digits in elements of array.
- **Count sort**
  Best, average and worst case time complexity: n+k where k is the size of count array.
- **Bucket sort**
  Best and average time complexity: n+k where k is the number of buckets.
  Worst case time complexity: n^2 if all elements belong to same bucket.

Summary of time complexity of non-comparison based Sorting Techniques



Analysis Type	Radix Sort (k is maximum digit)	Counting Sort (k is size of count array)	Bucket Sort (k is number of buckets)
Best Case	O(nk)	O(n + k)	O(n + k)
Average Case	O(nk)	O(n + k)	O(n + k)
Worst Case	O(nk)	O(n + k)	$O(n^2)$

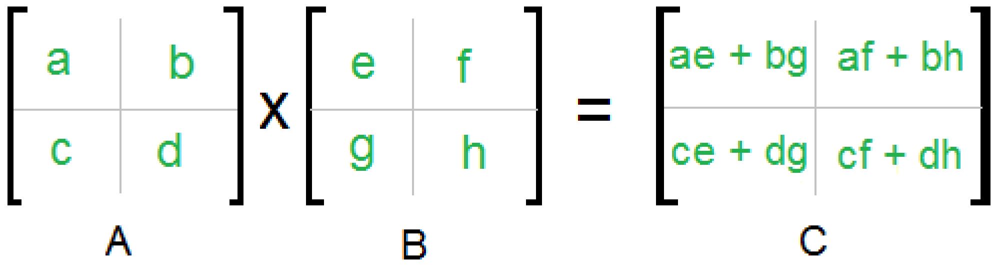
Matrix Multiplication Problem

Divide and Conquer

Following is simple Divide and Conquer method to multiply two square matrices.

1) Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.

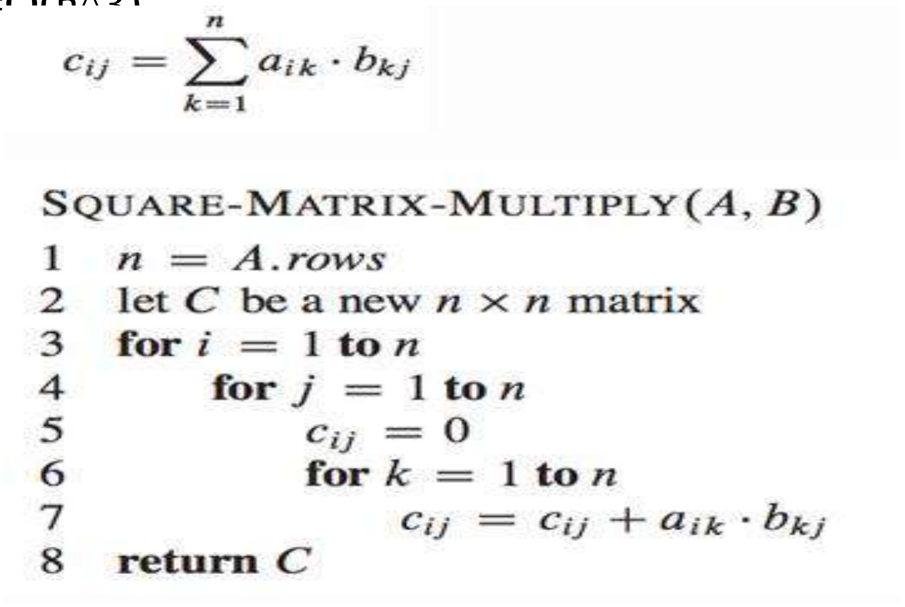
2) Calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh.



A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

Matrix Multiplication Problem

- **Naive matrix multiplication**
- Let us start with two square matrices A and B which are both of size n by n. In the product C = A X B we define the entry cij, the entry in the ith row and the jth column of A, as being the dot product of the ith row of A with the jth column of B. Taking the dot product of vectors just means that you take the products of the individual components and then add up the results.
- Complexity =O(n^3)



$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

```
SQUARE-MATRIX-MULTIPLY(A, B)
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        $c_{ij} = 0$
        for k = 1 to n
            $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
return C
```

Matrix Multiplication Problem

- In the above method, we do 8 multiplications for matrices of size N/2 x N/2 and 4 additions. Addition of two matrices takes $O(N^2)$ time. So the time complexity can be written
- $T(N) = 8T(N/2) + O(N^2)$
- From Master's Theorem, time complexity of above method is $O(N^3)$ which is unfortunately same as the above naive method.
- ***Simple Divide and Conquer also leads to $O(N^3)$, can there be a better way?*** In the above divide and conquer method, the main component for high time complexity is 8 recursive calls. The idea of **Strassen's method** is to reduce the number of recursive calls to 7. Strassen's method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size N/2 x N/2 as shown in the above diagram, but in Strassen's method, the four sub-matrices of result are calculated using following formulae.

Matrix Multiplication Problem

p1 = a(f - h)

p2 = (a + b)h

p3 = (c + d)e

p4 = d(g - e)

p5 = (a + d)(e + h)

p6 = (b - d)(g + h)

p7 = (a - c)(e + f)

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\underbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}_{A} \times \underbrace{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}_{B} = \underbrace{\begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}}_{C}$$

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

Time Complexity of Strassen’s Method

- Addition and Subtraction of two matrices takes $O(N^2)$ time. So time complexity can be written as

  $$T(N) = 7T(N/2) + O(N^2)$$

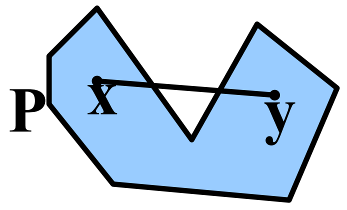
- From Master's Theorem, time complexity of above method is $O(N^{Log7})$ which is approximately $O(N^{2.8074})$

- Generally Strassen’s Method is not preferred for practical applications for following reasons.
  1) The constants used in Strassen’s method are high and for a typical application Naive method works better.
  2) For Sparse matrices, there are better methods especially designed for them.
  3) The submatrices in recursion take extra space.

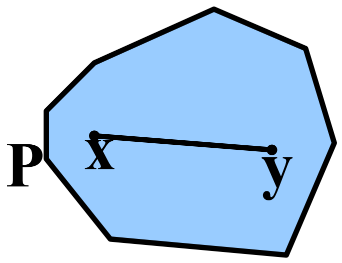
Convex Hull

Convex vs. Concave

- A polygon P is convex if for every pair of points x and y in P, the line xy is also in P; otherwise, it is called concave.



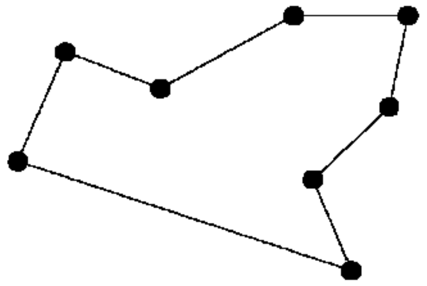
concave

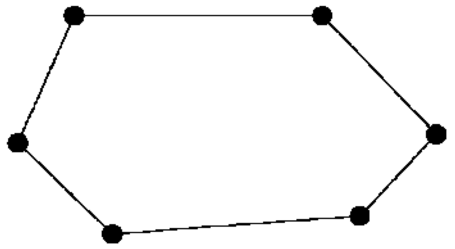


convex

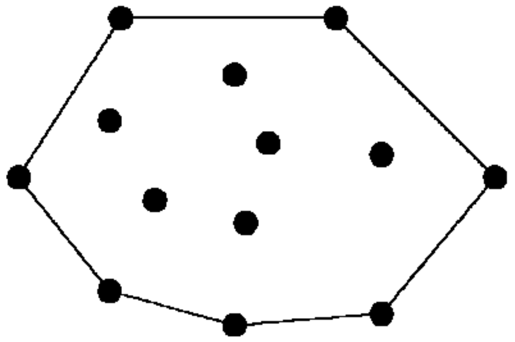
The convex hull problem

concave polygon: convex polygon:





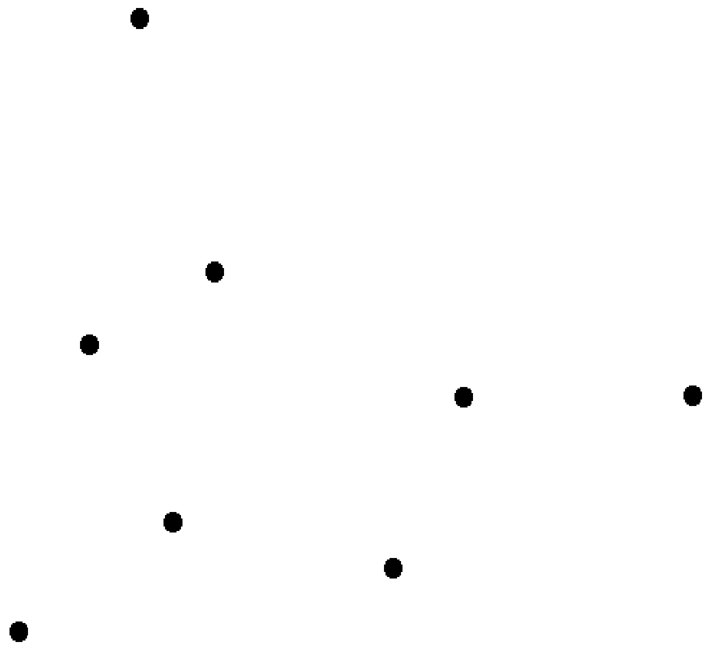
- The convex hull of a set of planar points is the smallest convex polygon containing all of the points.



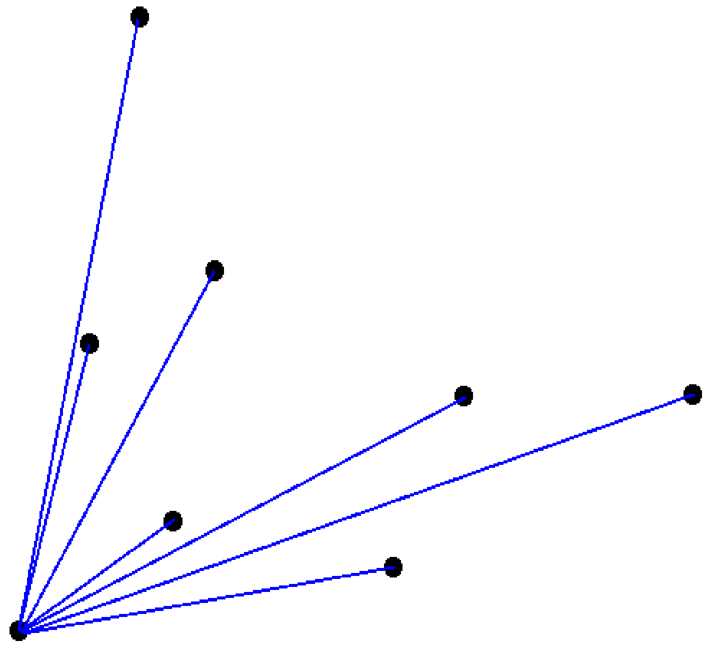
Graham’s Scan

- **Graham's scan** is a method of finding the convex hull of a finite set of points in the plane with time complexity O(n log n).
- It is named after Ronald **Graham**, who published the original algorithm in 1972.
- The algorithm finds all vertices of the convex hull ordered along its boundary.
- Start at point guaranteed to be on the hull. (the point with the minimum y value)
- Sort remaining points by polar angles of vertices relative to the first point.
- Go through sorted points, keeping vertices of points that have left turns and dropping points that have right turns.

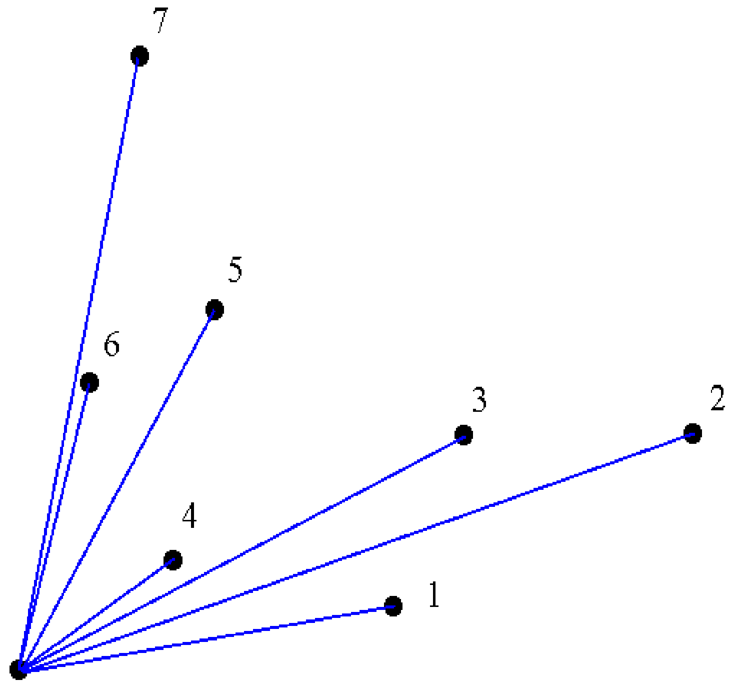
Graham’s Scan



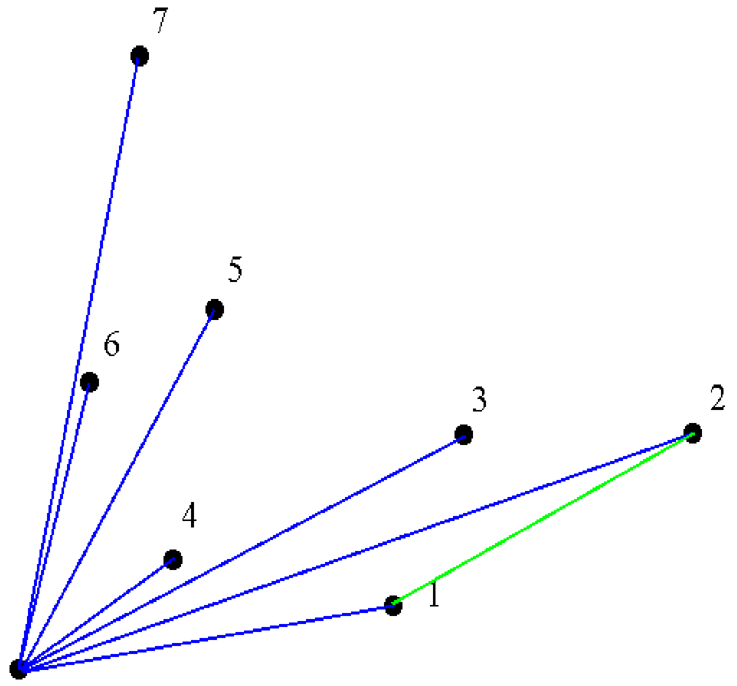
Graham’s Scan



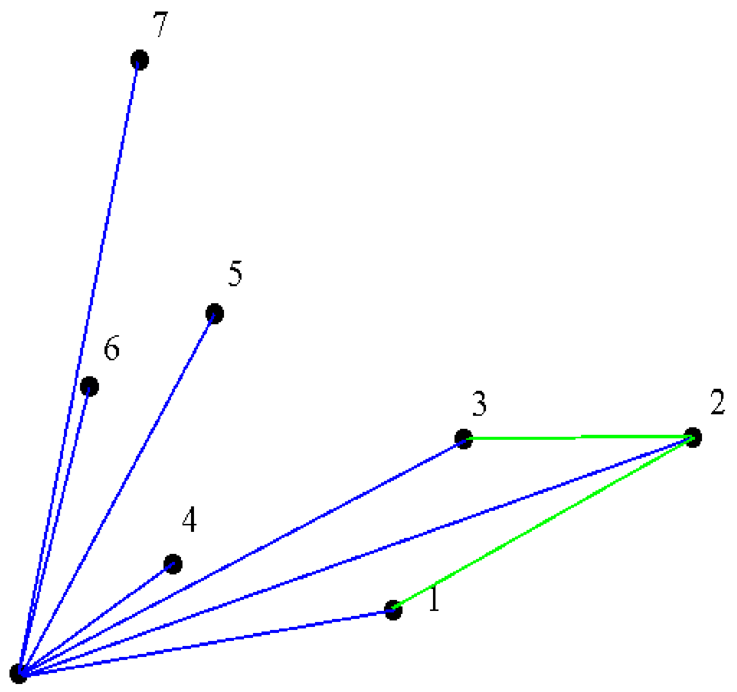
Graham's Scan



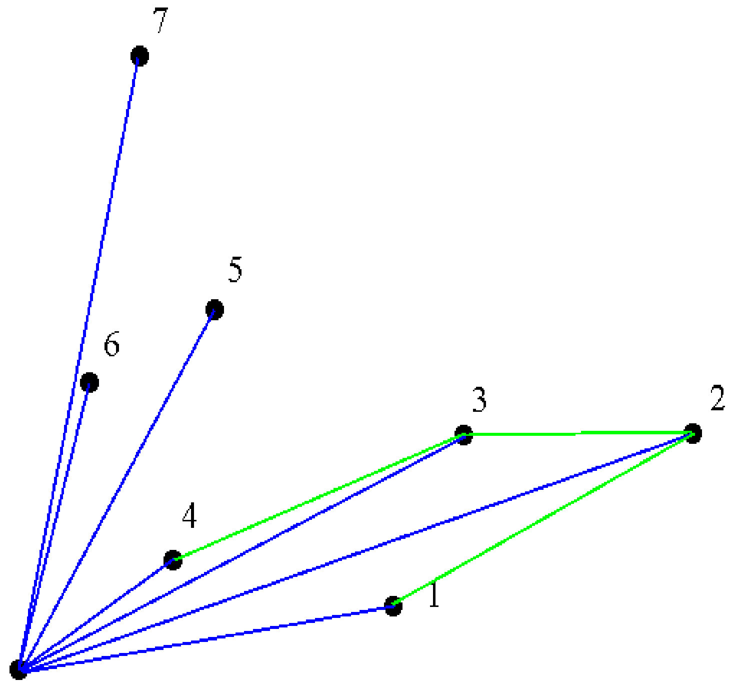
Graham's Scan



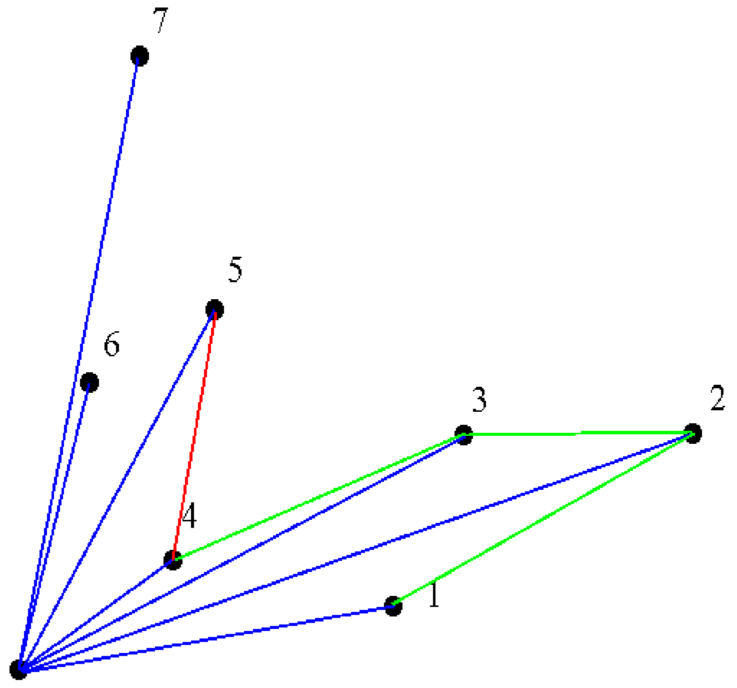
Graham’s Scan



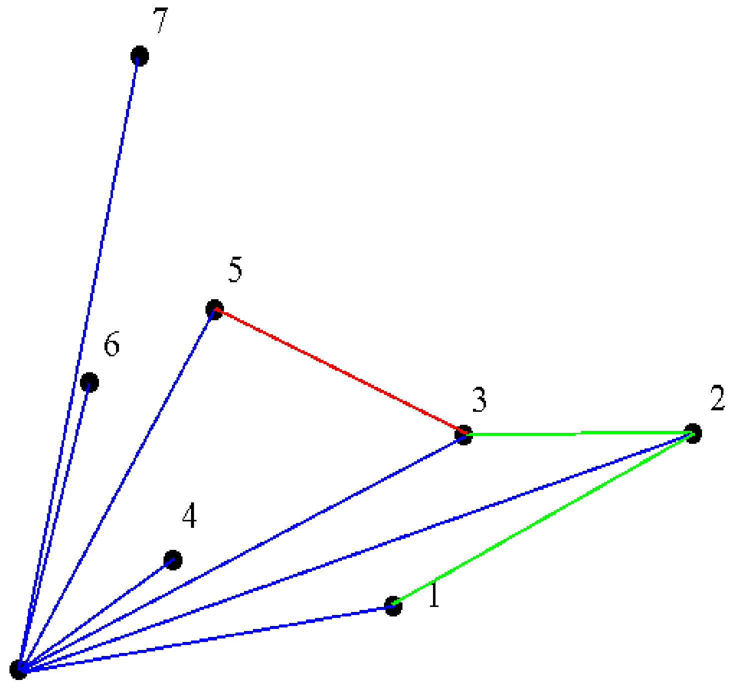
Graham’s Scan



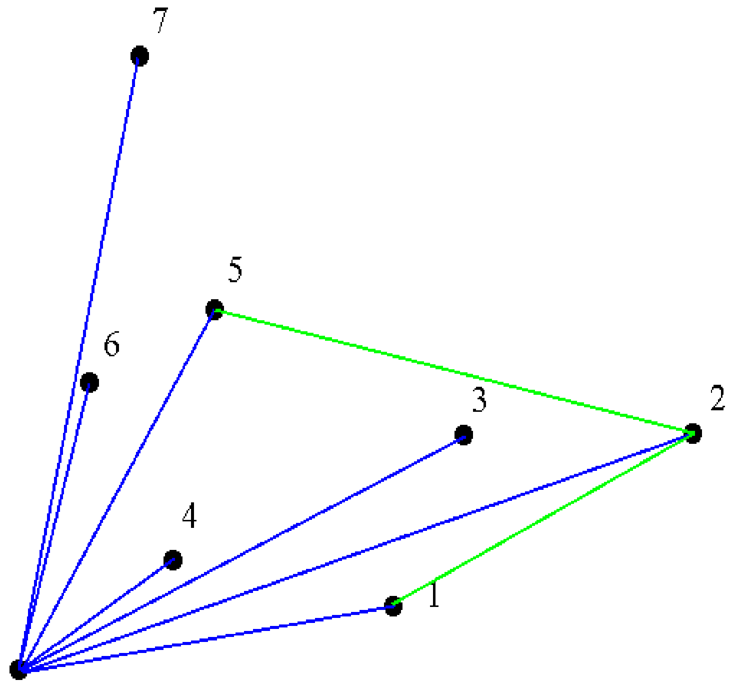
Graham's Scan



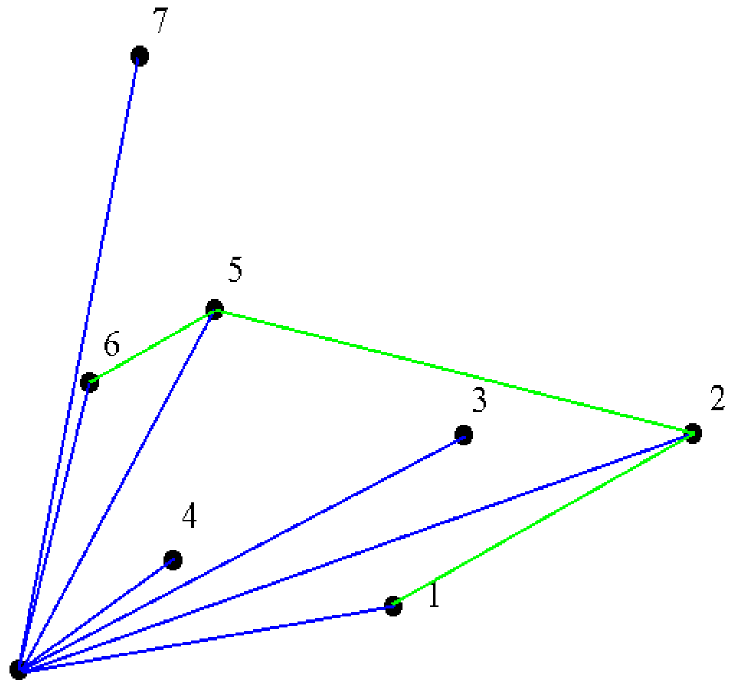
Graham’s Scan



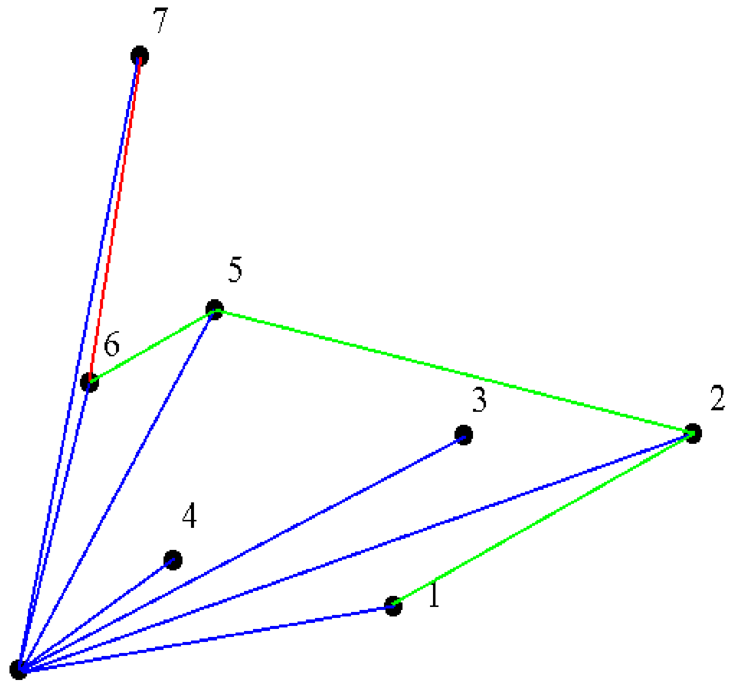
Graham's Scan



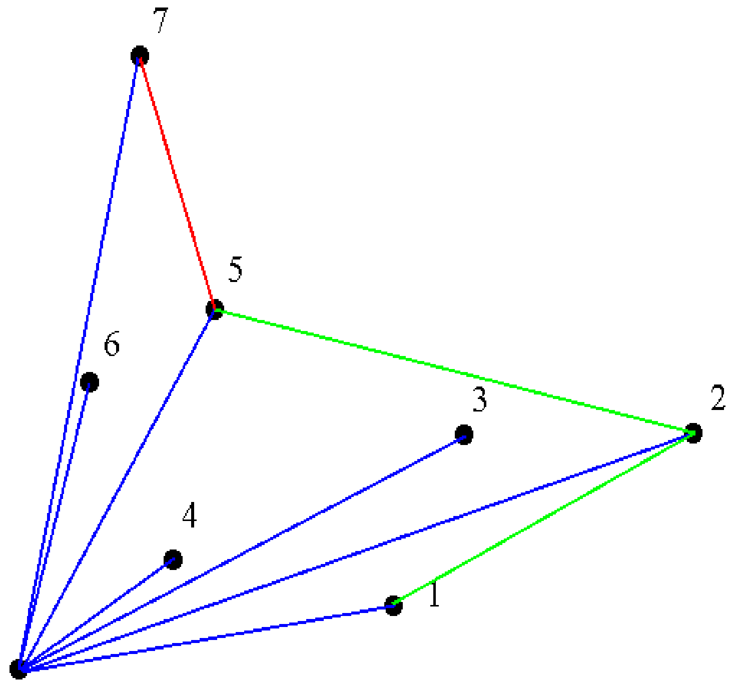
Graham’s Scan



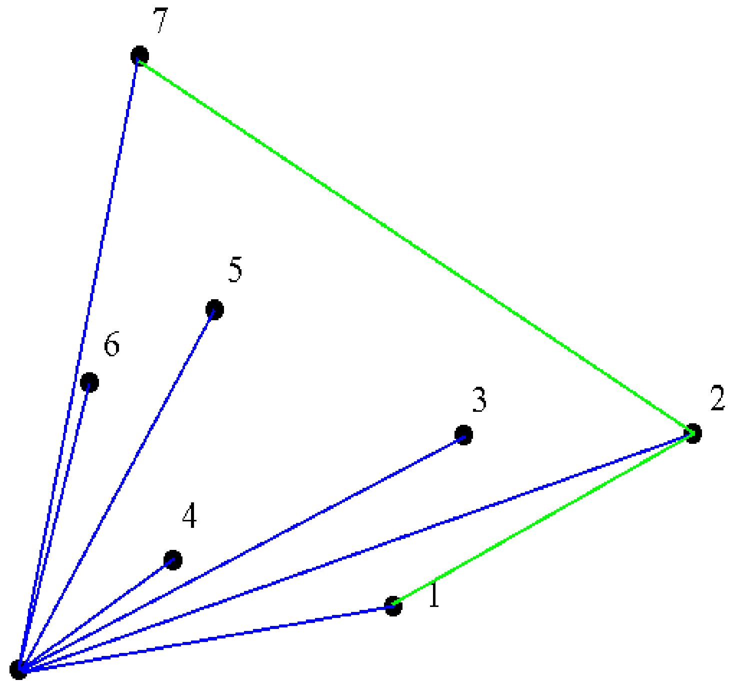
Graham’s Scan



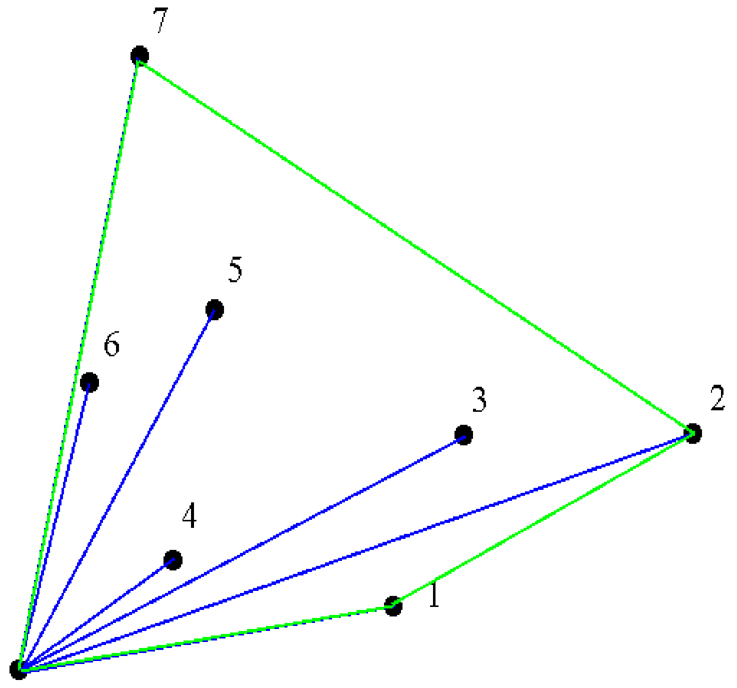
Graham's Scan



Graham’s Scan



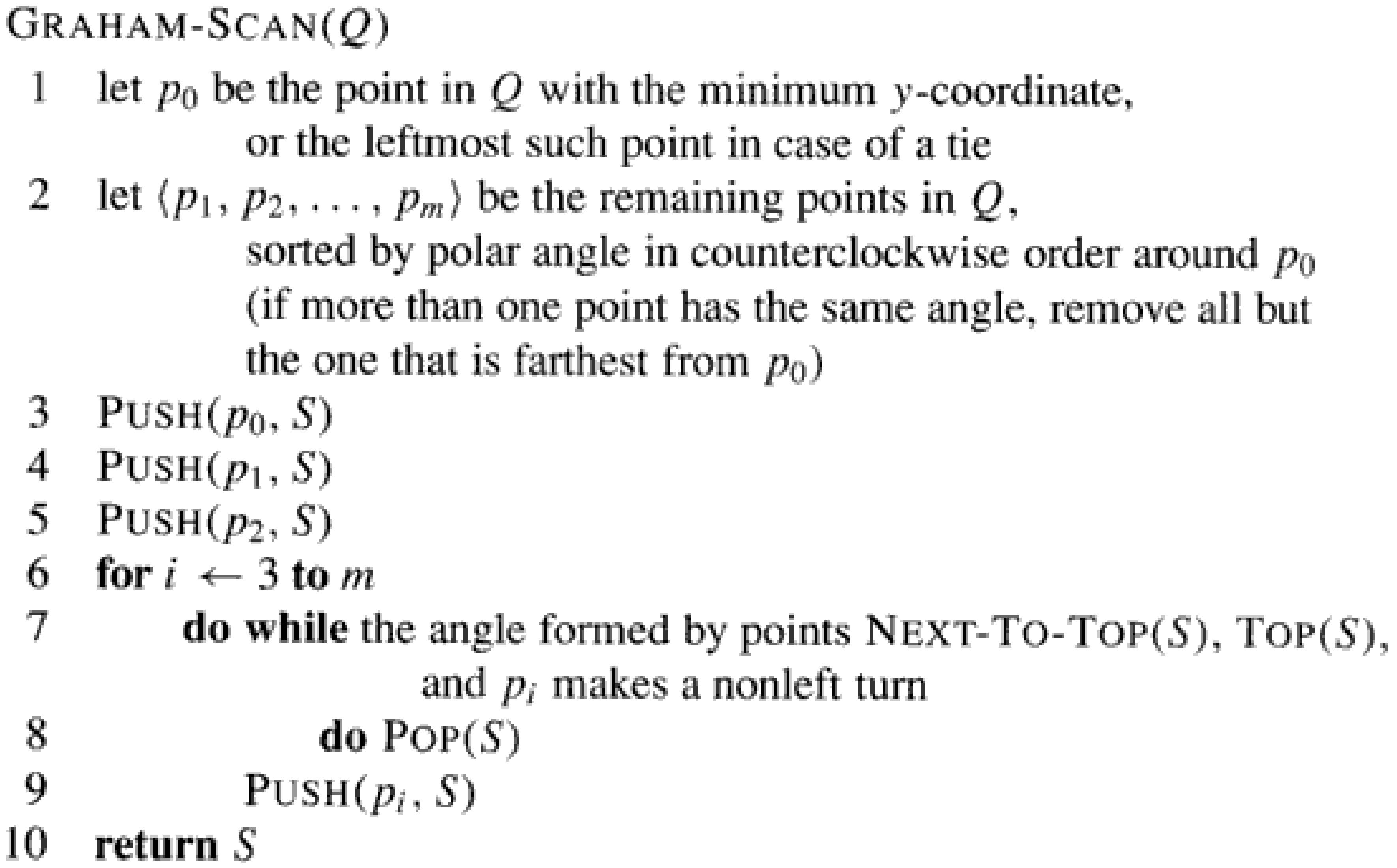
Graham’s Scan



Graham’s Runtime

- Graham’s scan is O(n log n) due to initial sort of angles.

A more detailed algorithm

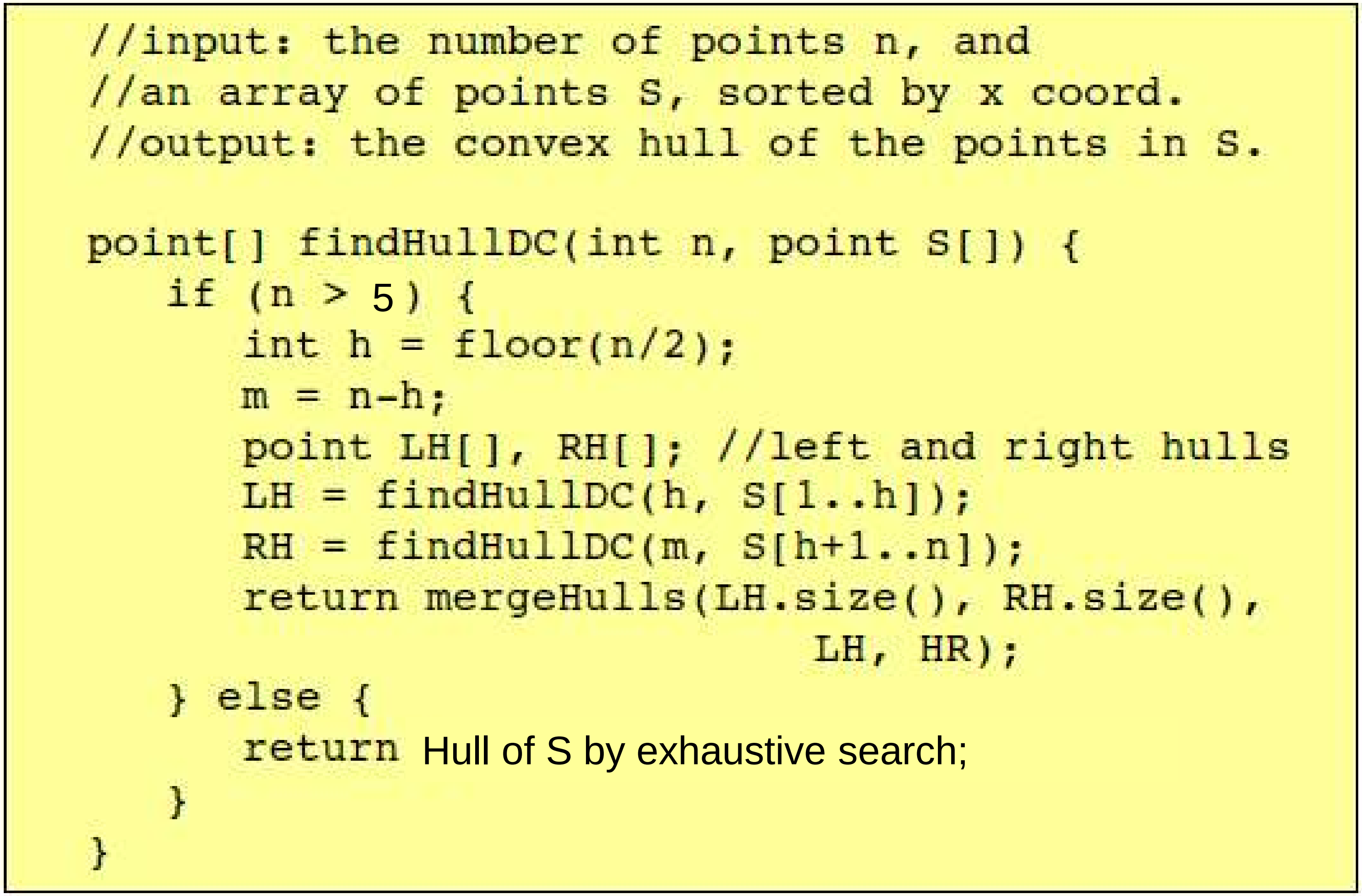


```
GRAHAM-SCAN($Q$)
 1  let $p_0$ be the point in $Q$ with the minimum $y$-coordinate,
            or the leftmost such point in case of a tie
 2  let $\langle p_1, p_2, \ldots, p_m \rangle$ be the remaining points in $Q$,
            sorted by polar angle in counterclockwise order around $p_0$
            (if more than one point has the same angle, remove all but
            the one that is farthest from $p_0$)
 3  PUSH($p_0$, $S$)
 4  PUSH($p_1$, $S$)
 5  PUSH($p_2$, $S$)
 6  for $i \leftarrow 3$ to $m$
 7       do while the angle formed by points NEXT-TO-TOP($S$), TOP($S$),
                    and $p_i$ makes a nonleft turn
 8              do POP($S$)
 9          PUSH($p_i$, $S$)
10  return $S$
```

Convex Hull by Divide-and-Conquer

- First, sort all points by their x coordinate.
  - (O(n log n) time)
- Then divide and conquer:
  - Find the convex hull of the left half of points.
  - Find the convex hull of the right half of points.
  - Merge the two hulls into one.

Convex Hull Pseudocode



```
//input: the number of points n, and
//an array of points S, sorted by x coord.
//output: the convex hull of the points in S.

point[] findHullDC(int n, point S[]) {
    if (n > 5) {
        int h = floor(n/2);
        m = n-h;
        point LH[], RH[]; //left and right hulls
        LH = findHullDC(h, S[1..h]);
        RH = findHullDC(m, S[h+1..n]);
        return mergeHulls(LH.size(), RH.size(),
                                      LH, HR);
    } else {
        return Hull of S by exhaustive search;
    }
}
```

Convex Hull: Runtime

• Preprocessing: sort the points by x-coordinate	$O(n \log n)$ just once
• Divide the set of points into two sets **A** and **B**:	$O(1)$
• **A** contains the left $\lfloor n/2 \rfloor$ points,	
• **B** contains the right $\lceil n/2 \rceil$ points	
• Recursively compute the convex hull of **A**	$T(n/2)$
• Recursively compute the convex hull of **B**	$T(n/2)$
• Merge the two convex hulls	$O(n)$

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

$$T(n) = O(n \log n)$$

Thank You