Brute Force

- **Brute force** is a straightforward approach to solve a problem based on the problem's statement and definitions of the concepts involved.
- It is considered as one of the easiest approach to apply and is useful for solving small–size instances of a problem

- Example 1: Computing a^n ($a > 0$, n a nonnegative integer) based on the definition of exponentiation

  $a^n = a * a * a * .... * a$
- The brute force algorithm requires n-1 multiplications.
  The recursive algorithm for the same problem, based on the observation that $a^n = a^{n/2} * a^{n/2}$ requires Θ(log(n)) operations.
- **Example 2:** Computing n! based on the definition n! = 1*2*3*…*n The algorithm requires Θ (n) operations

Brute-Force Search and Sort

- Sequential search in an unordered array and simple sorts – selection sort, bubble sort are brute force algorithms.
- Sequential search: the algorithm simply compares successive elements of a given list with a given search key until either a match is found or the list is exhausted without finding a match.
- The complexity of a sequential search algorithm is $\Theta(n)$ in the worst possible case and $\Theta(1)$ in the best possible case, depending on where the desired element is situated.

Brute-Force -Sort

Selection sort:

- the entire given list of *n* elements is scanned to find its smallest element and exchange it with the first element.
- Thus, the smallest element is moved to its final position in the sorted list.
- Then, the list is scanned again, starting with the second element in order to find the smallest element among the *n – 1* and exchange it with the second element.
- The second smallest element is put in its final position in the sorted list.
- After *n-1* passes, the list is sorted.

Brute-Force – Selection Sort

Algorithm SelectionSort (A[0..n-1])

for i ← 0 to n-2 do

    min ← i

    for j ← i + 1 to n-1 do

        if A[j] < A[min]

            min ← j

    swap A[i] and A[min]

- The basic operation of the selection sort is the comparison -> A[j] < A[min]. The complexity of the algorithm is $\Theta(n^2)$ and the number of key swaps is $\Theta(n)$.

Brute-Force – Bubble Sort

- Bubble sort is another application of a brute force.
- In the algorithm, adjacent elements of the list are compared and are exchanged if they are out of order.
- Algorithm BubbleSort (A[0..n-1])

```
for i ← 0 to  n-2 do
        for j ← 0 to n – 2 – i do
                if A[j+1] < A[j]
                   swap A[j] and A[j+1]
```

Brute-Force -Sort

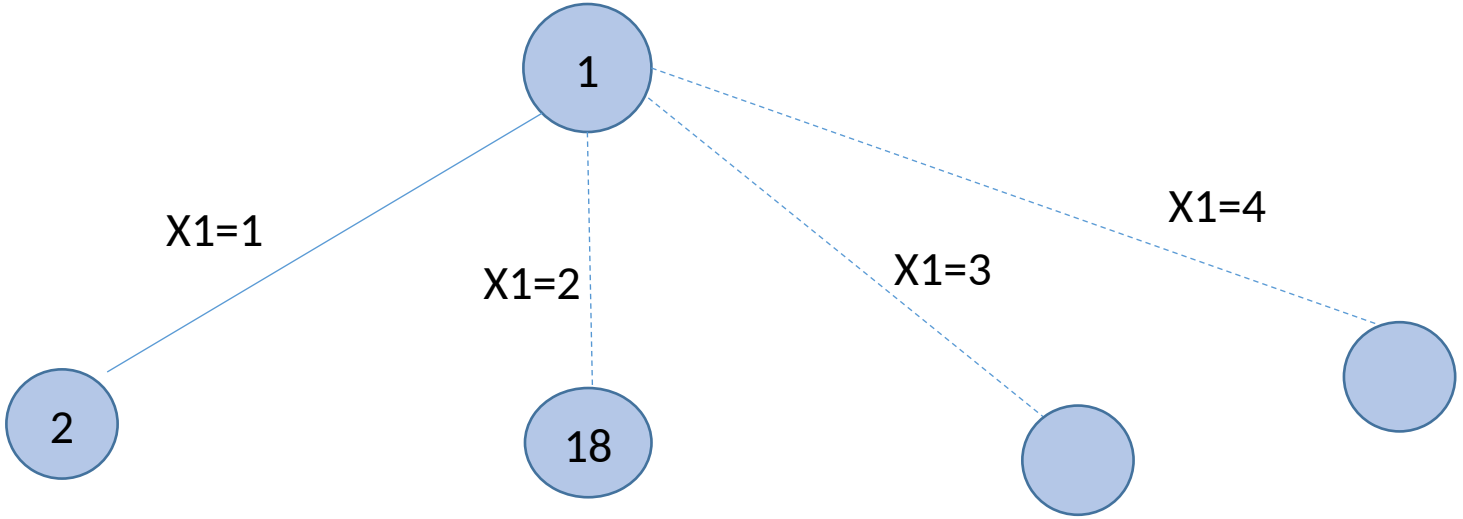
- The basic operation of the bubble sort is comparison - A[j+1] < A[j] and swapping - swap A[j] and A[j+1].
- The number of key comparisons is the same for all arrays of size n and it is $\Theta(n^2)$.
- However, the number of key swaps depends on the input and in the worst case is $\Theta(n^2)$.
- The above implementation of Bubble sort can be slightly improved if we stop the execution of the algorithm when a pass through the list makes no exchanges (i.e. indicating that the list has been sorted).
- Thus, in the best case the complexity will be $\Theta(n)$ and the worst case $\Theta(n^2)$.

Exhaustive Search

- Exhaustive search is used in problems where the solution is an object with specific properties in a set of candidate solutions.
- We have to examine all candidate solutions to find the solution if it exists.
- There are two types of problems that involve exhaustive search –
  - state-space search problems and
  - combinatorial problems.

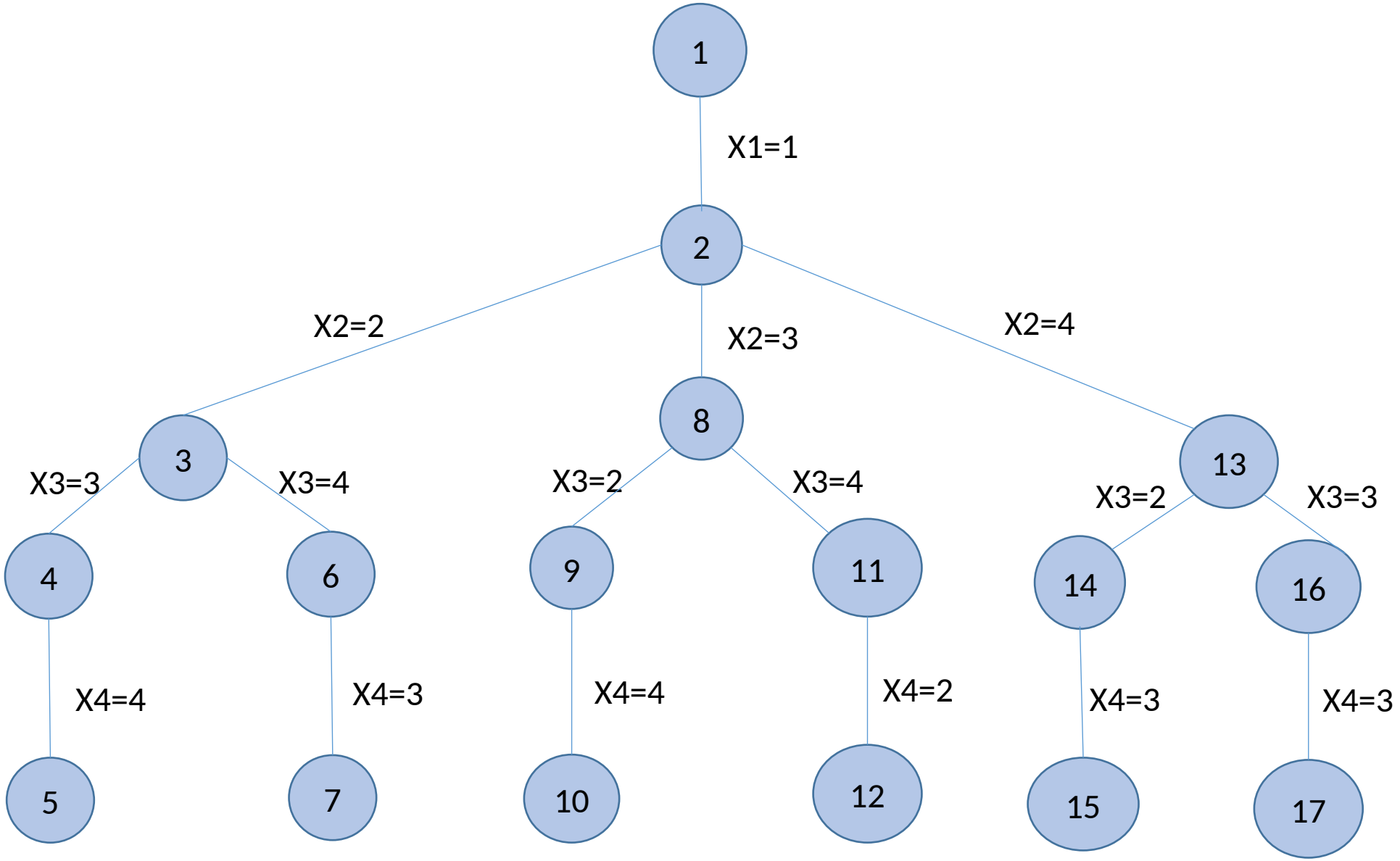
Exhaustive Search: state-space search

- In state-space search the problem is formulated as follows: Given an initial state, a goal state, and a set of operations, find a sequence of operations that transforms the initial state to the goal state. The solution process can be represented as a tree with the initial state being its root, and the goal state being a leaf in the tree, and each edge corresponds to an operation.
- At each node we attempt to apply all available operations. When an operation is applied, a new node is generated. If no operation can be applied, the node is called “dead end”. The process terminates when a node is generated that satisfies the requirements of the goal state or when no more nodes can be generated.
- The search can be performed in a breadth-first manner or in a depth-first manner. The problem-solving method is called “generate-and-test” approach because we generate a node and test to see if it is the solution.
- Exhaustive search is not feasible if the search tree grows exponentially. If this is the case, the generated nodes have to be evaluated (if possible) and so that at each step the best node is expanded. Many Artificial Intelligence problems fall into this category, e.g. the 8-puzzle problem.

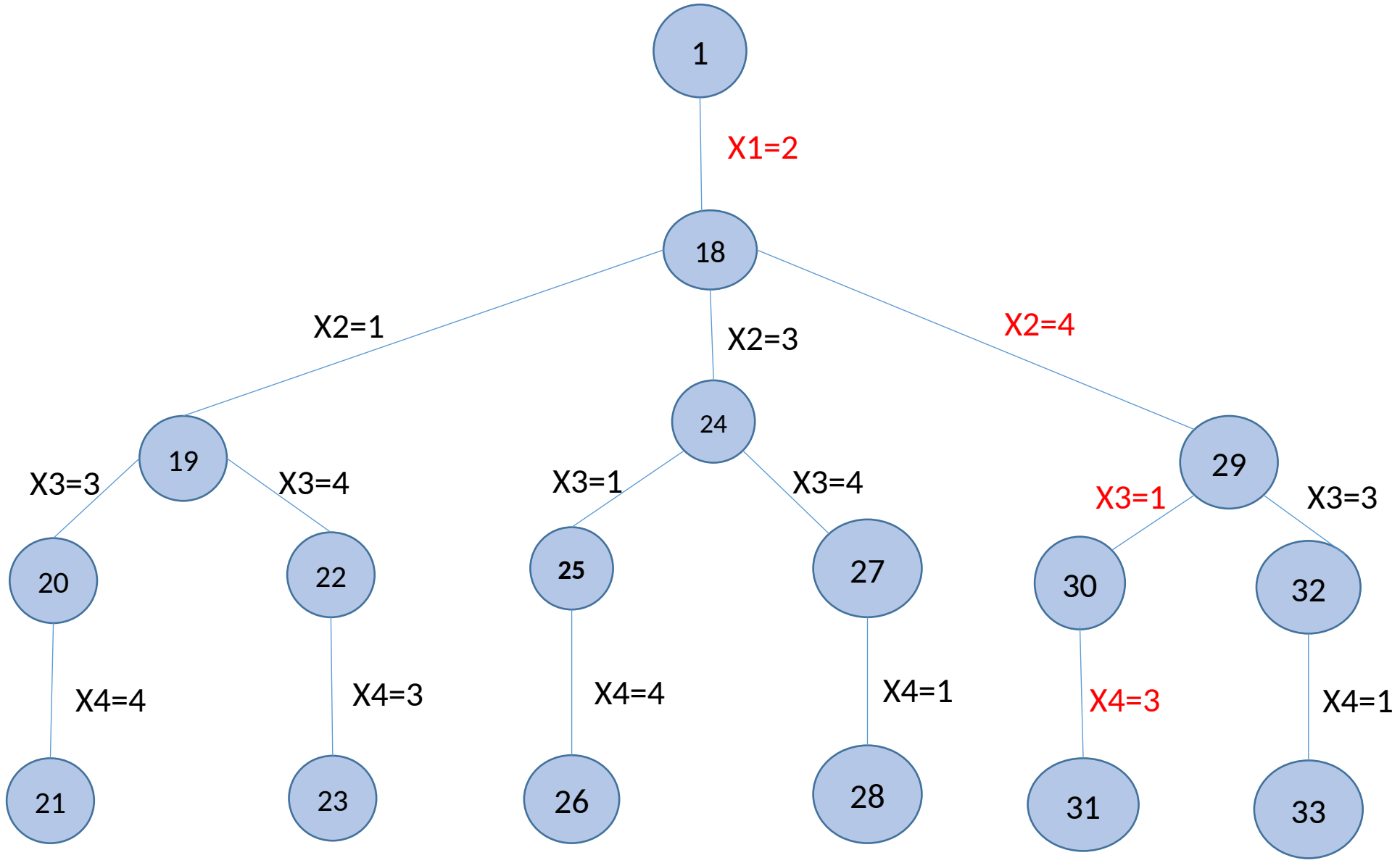


To be continued in next slide

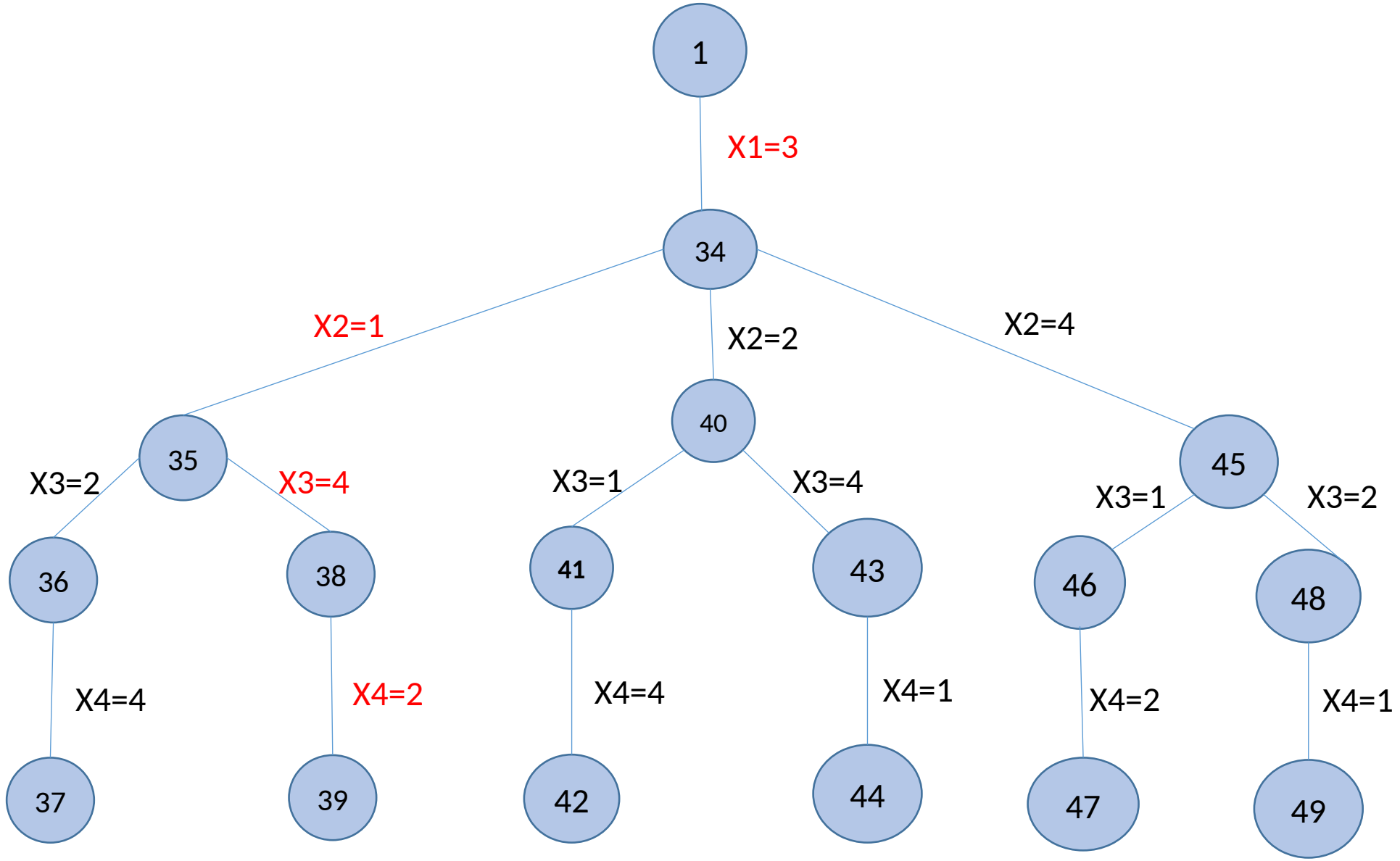
Tree organization of 4 queens solution space in DFS



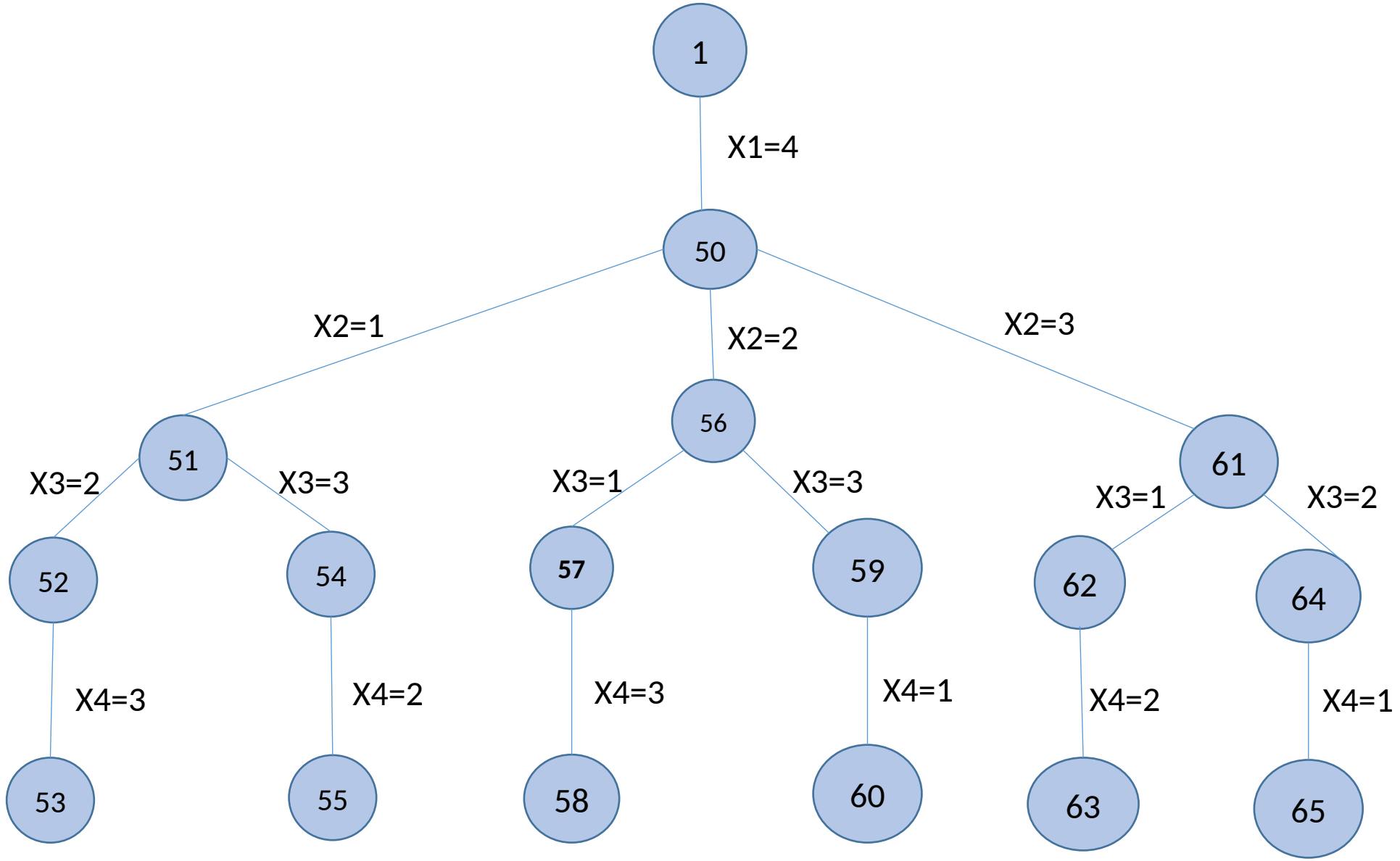
Tree organization of 4 queens solution space in DFS



Tree organization of 4 queens solution space in DFS



Tree organization of 4 queens solution space in DFS



Tree organization of 4 queens solution space in DFS

Exhaustive Search :Combinatorial problems

- In combinatorial problems the solution (if is exists) is an element of a set of combinatorial objects – permutations, combinations, or subsets. The brute-force approach consists in generating the combinatorial objects and testing each object to see if it satisfies some specified constraints. Problems like the traveling salesman, knapsack, and bin-packing can be solved by using an exhaustive search.
- The **traveling salesman problem** tries to find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city whether it started. Since each tour is a permutation of the n cities, and since we can fix the start city, the time efficiency of this algorithm is $\Theta((n-1)!)$.
- The **knapsack problem**, for given **n** items with their benefits and volume, tries to find the most valuable subset (i.e. the one with the greatest benefit) without exceeding the capacity of the knapsack. The solution is one of all subsets of the set of objects and therefore the time efficiency of this algorithm is $\Theta(2^n)$.

Exhaustive Search :Combinatorial problems

- n the assignment problem we have n persons that have to be assigned n jobs, one person per job. The problem is represented as a cost matrix, where c(i,j) is the cost of assigning job j to person i. The solution is a vector (a1, a2, ..an) such that person i is assigned job ai and the sum ∑ c(i,ai) is minimum.
- The solution is a permutation of the n jobs and therefore the time efficiency is Θ(n!).
- In combinatorial problems, The time efficiency is determined by the domain size. Once we have a candidate for a solution we can examine it in polynomial time. Thus, if we have all possible tours, we can find the shortest one in linear time. However, it take Θ((n-1)!) to generate all permutations.
- Exhaustive search algorithms run in a realistic amount of time only on very small instances of the problem at hand. In many cases, there are much better alternative algorithms. For example the 8-puzzle problem is solved using the heuristic A* algorithm. However, in some cases such as the traveling salesman problem, exhaustive search is the only known algorithm to obtain exact solution. Approximate algorithms are used to solve realistic instances of such problems.

The strengths of using a brute force approach

- It has wide applicability and is known for its simplicity.
- It yields reasonable algorithms for some important problems such as searching, string matching, and matrix multiplication.
- It yields standard algorithms for simple computational tasks such as sum and product of n numbers, and finding maximum or minimum in a list.

The weaknesses of the brute force approach

- It rarely yields efficient algorithms.
- Some brute force algorithms are unacceptably slow.
- It is neither as constructive nor creative as some other design techniques.

Sorting

- Insertion sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Best case: $\Theta(n)$
  - Worst case: $\Theta(n^2)$

- Bubble Sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Running time: $\Theta(n^2)$

Sorting

- Selection sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Running time: $\Theta(n^2)$

- Merge Sort
  - Design approach: divide and conquer
  - Sorts in place: No
  - Running time: Let's see!!

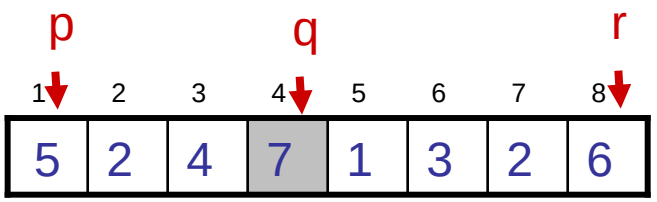
Divide-and-Conquer

- **Divide** the problem into a number of sub-problems
  - Similar sub-problems of smaller size
- **Conquer** the sub-problems
  - Solve the sub-problems recursively
  - Sub-problem size small enough ⇒ solve the problems in straightforward manner
- **Combine** the solutions of the sub-problems
  - Obtain the solution for the original problem

Merge Sort Approach

- To sort an array A[p . . r]:
- **Divide**
  - Divide the n-element sequence to be sorted into two subsequences of n/2 elements each
- **Conquer**
  - Sort the subsequences recursively using merge sort
  - When the size of the sequences is 1 there is nothing more to do
- **Combine**
  - Merge the two sorted subsequences

Merge Sort



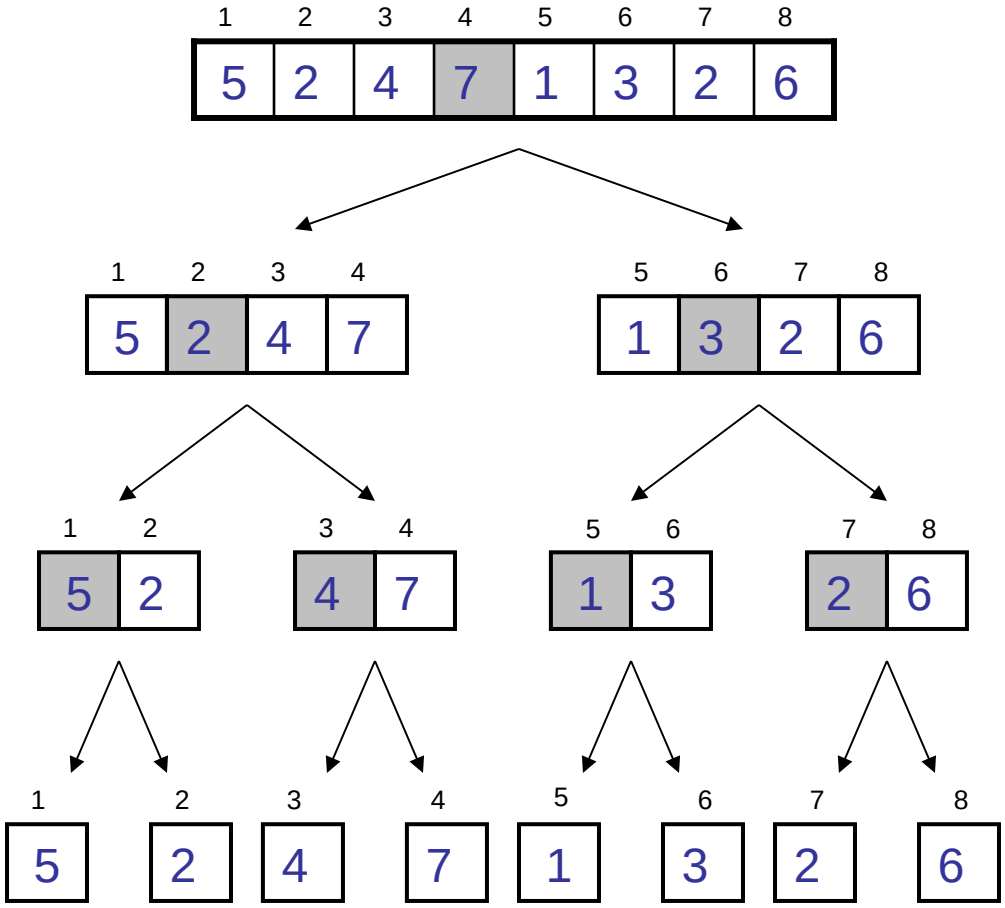
Alg.: MERGE-SORT(A, p, r)

```
if p < r                                  ▷ Check for base case
  then q ← ⌊(p + r)/2⌋                    ▷ Divide
       MERGE-SORT(A, p, q)                ▷ Conquer
       MERGE-SORT(A, q + 1, r)            ▷ Conquer
       MERGE(A, p, q, r)                  ▷ Combine
```

- Initial call: MERGE-SORT(A, 1, n)

Example – n Power of 2

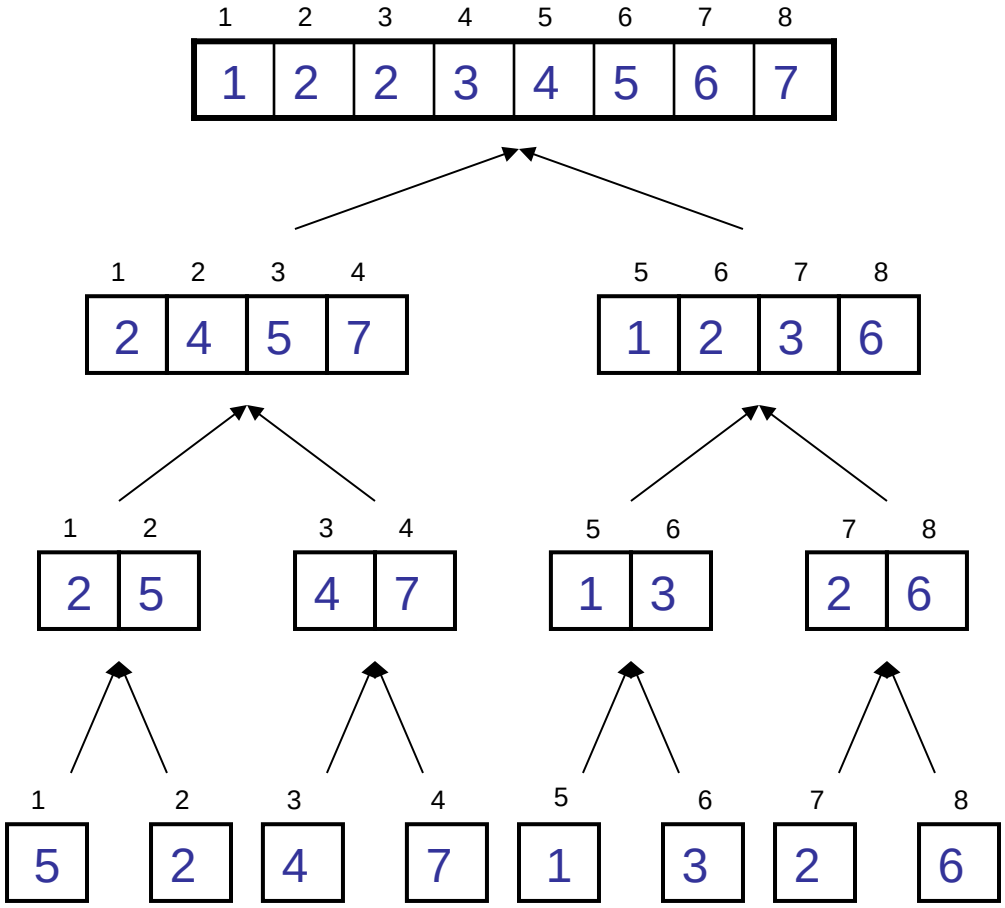
Divide



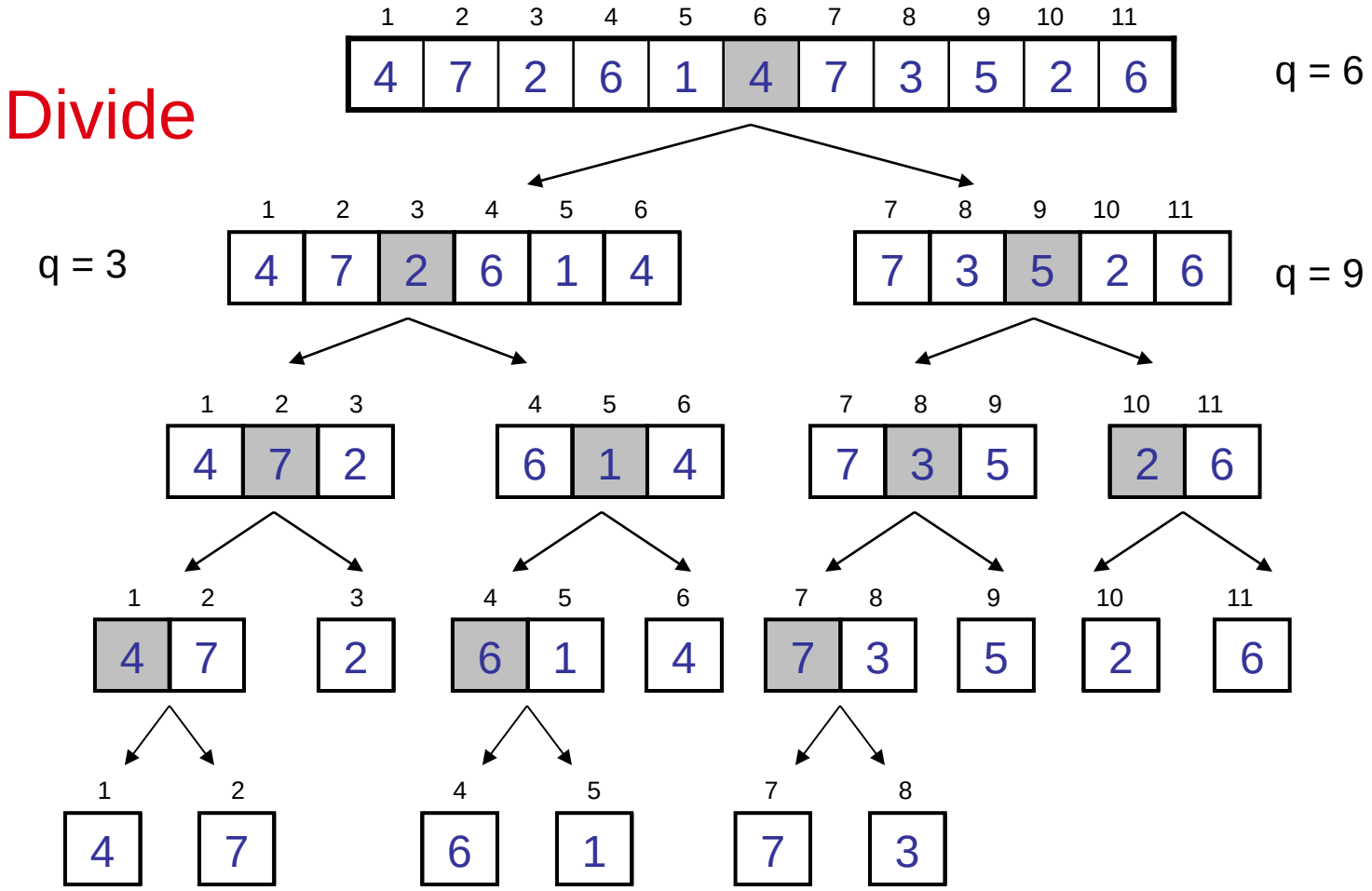
q = 4

Example – n Power of 2

Conquer and Merge

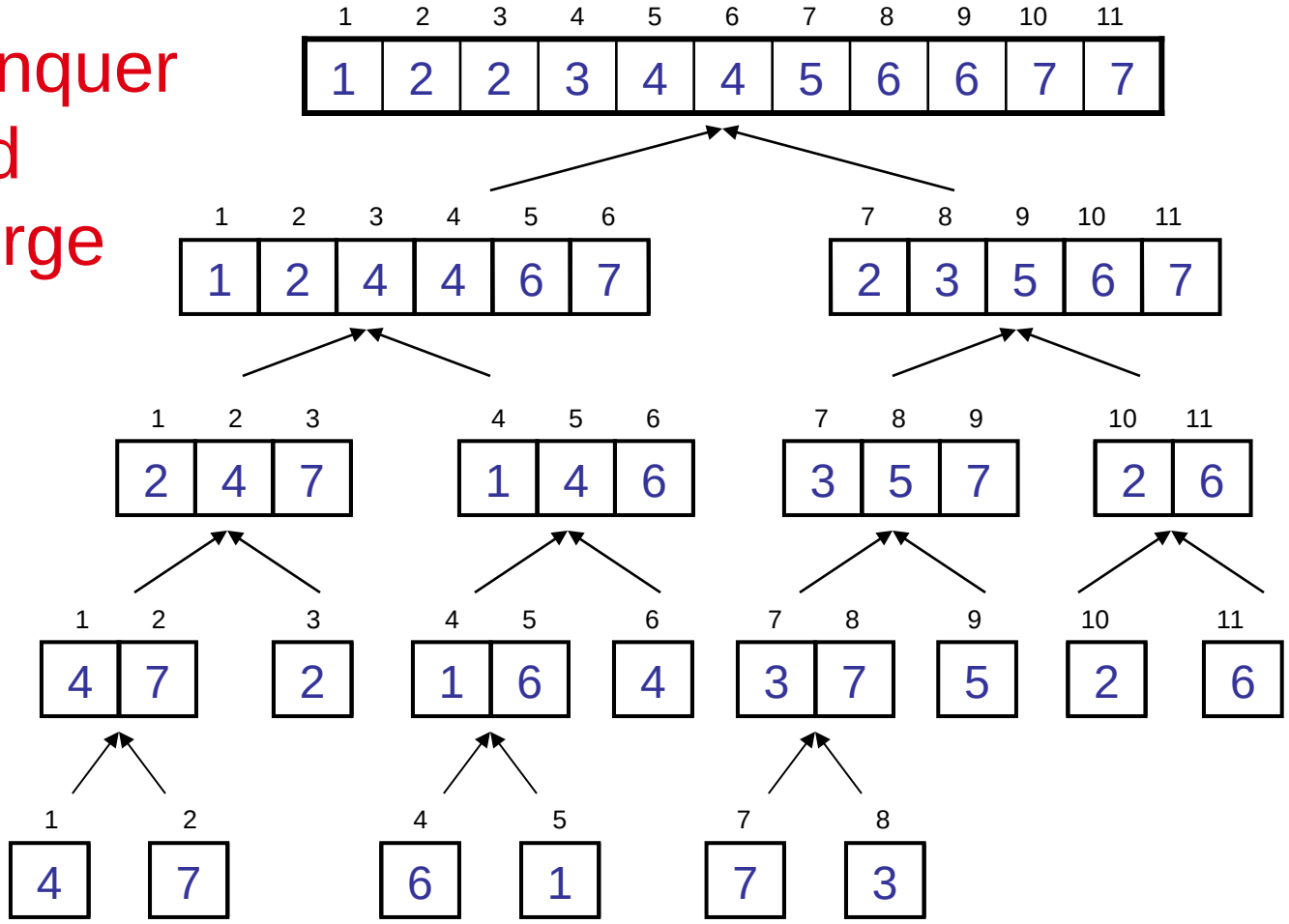


Example – n Not a Power of 2

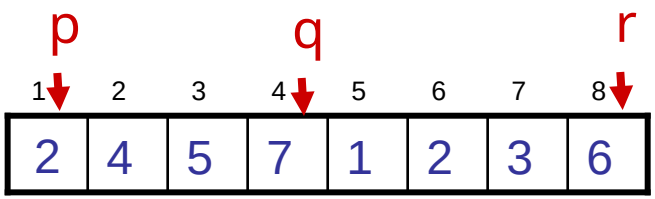


Example – n Not a Power of 2

Conquer and Merge

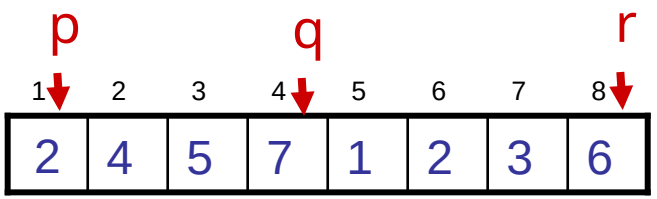


Merging

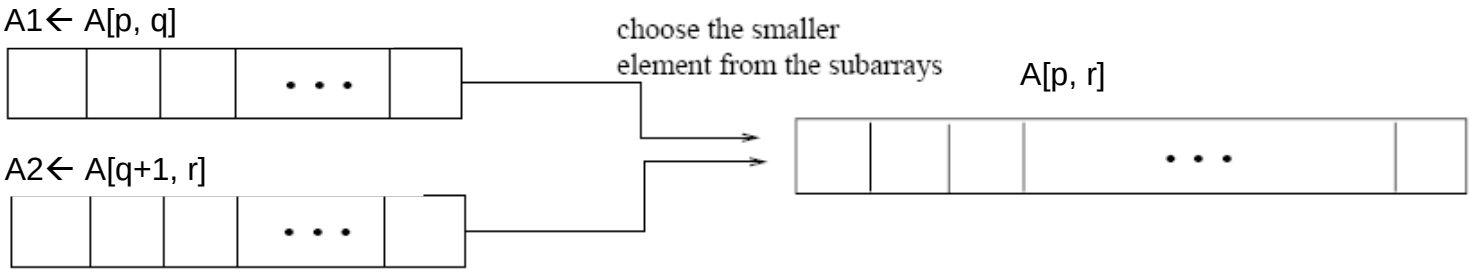


- **Input:** Array A and indices p, q, r such that $p \leq q < r$
  - Subarrays A[p . . q] and A[q + 1 . . r] are sorted
- **Output:** One single sorted subarray A[p . . r]

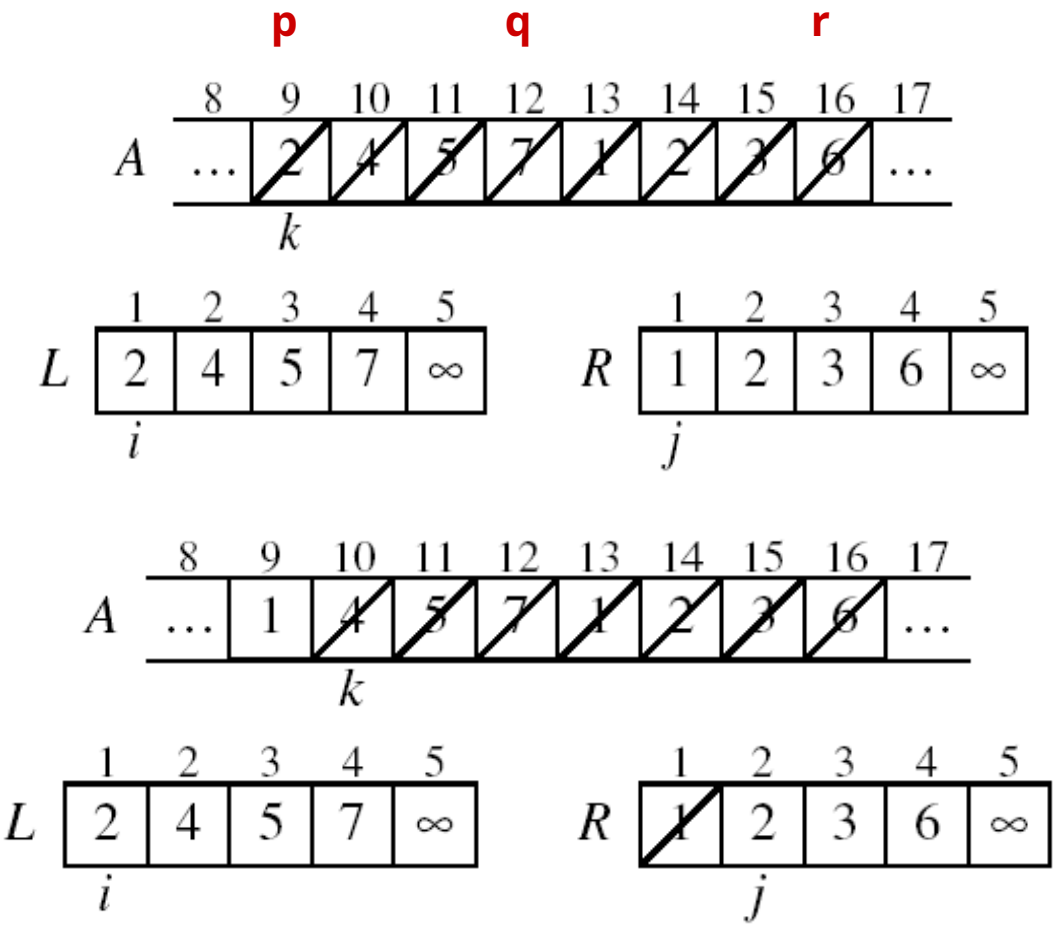
Merging



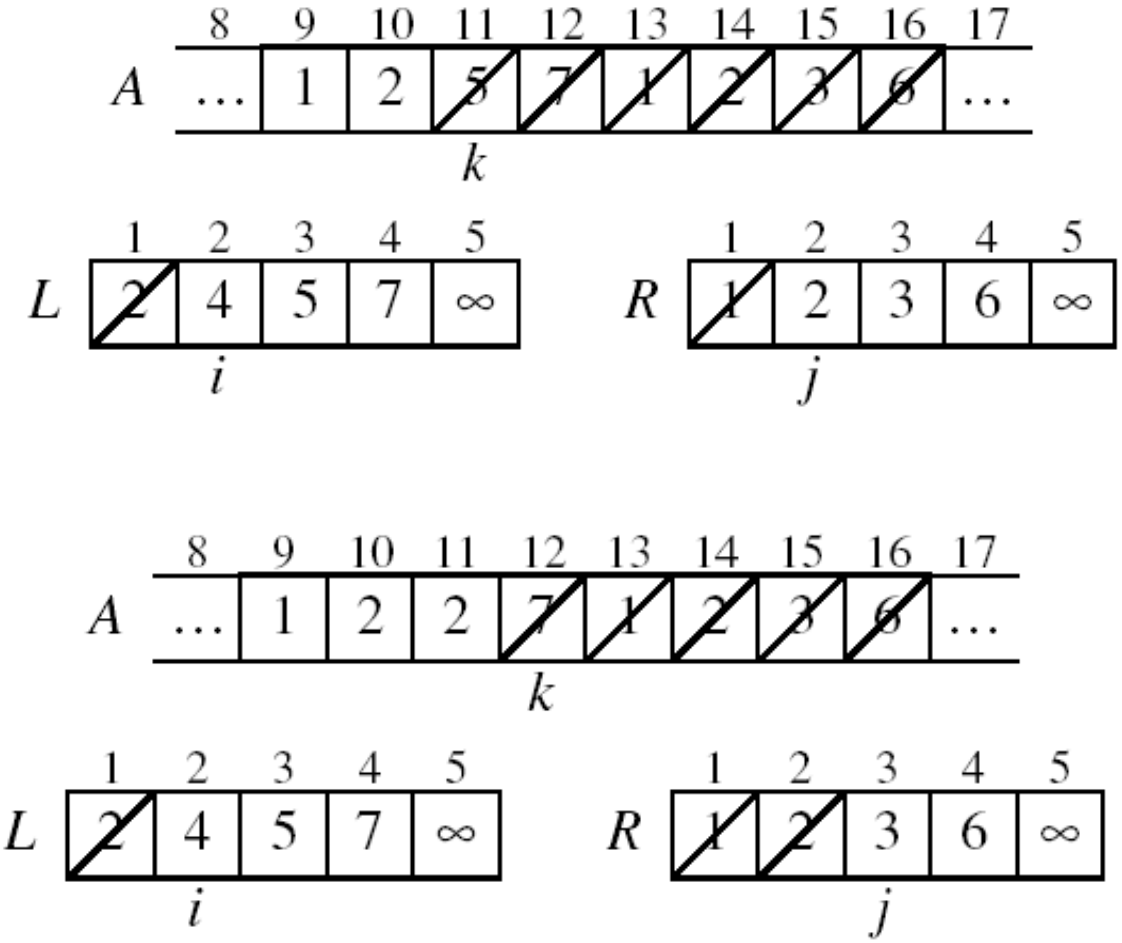
- Idea for merging:
  - Two piles of sorted cards
    - Choose the smaller of the two top cards
    - Remove it and place it in the output pile
  - Repeat the process until one pile is empty
  - Take the remaining input pile and place it face-down onto the output pile



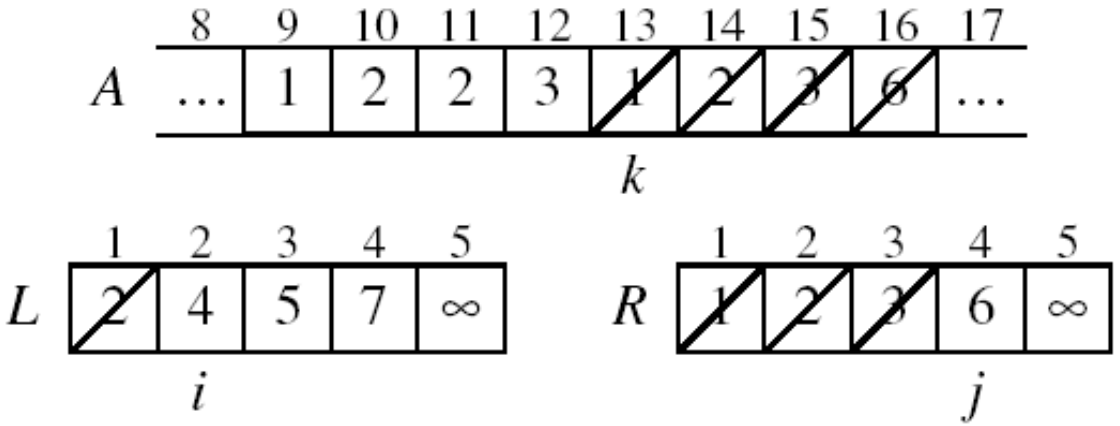
Example: MERGE(A, 9, 12, 16)

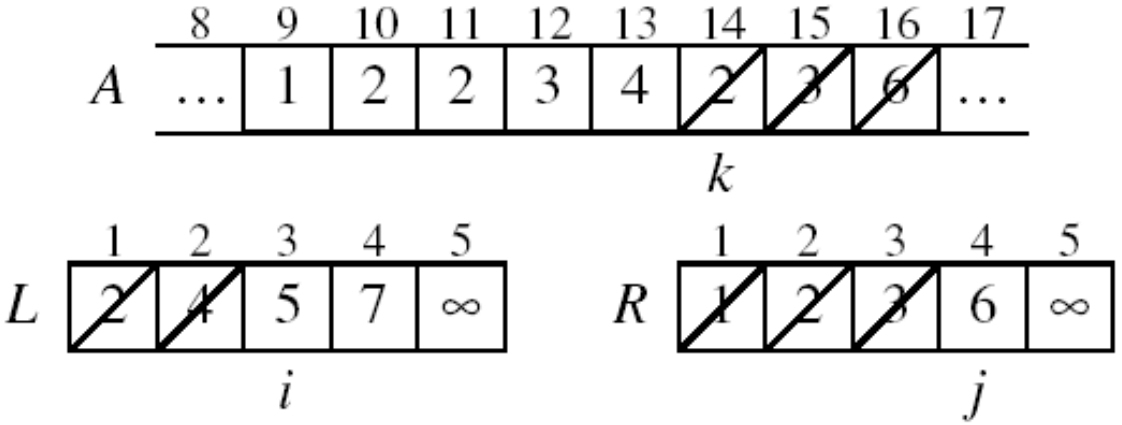


Example: MERGE(A, 9, 12, 16)

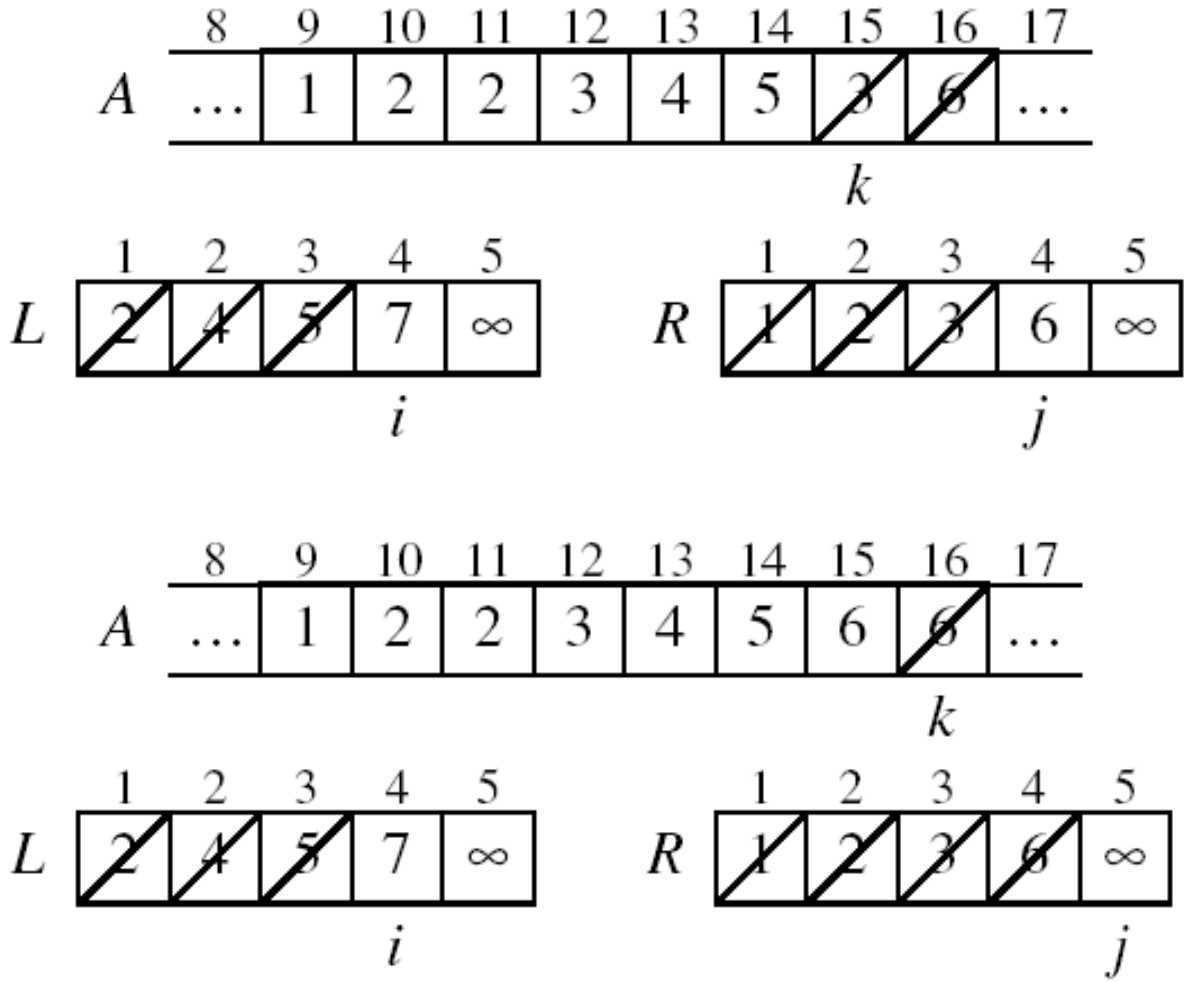


Example (cont.)

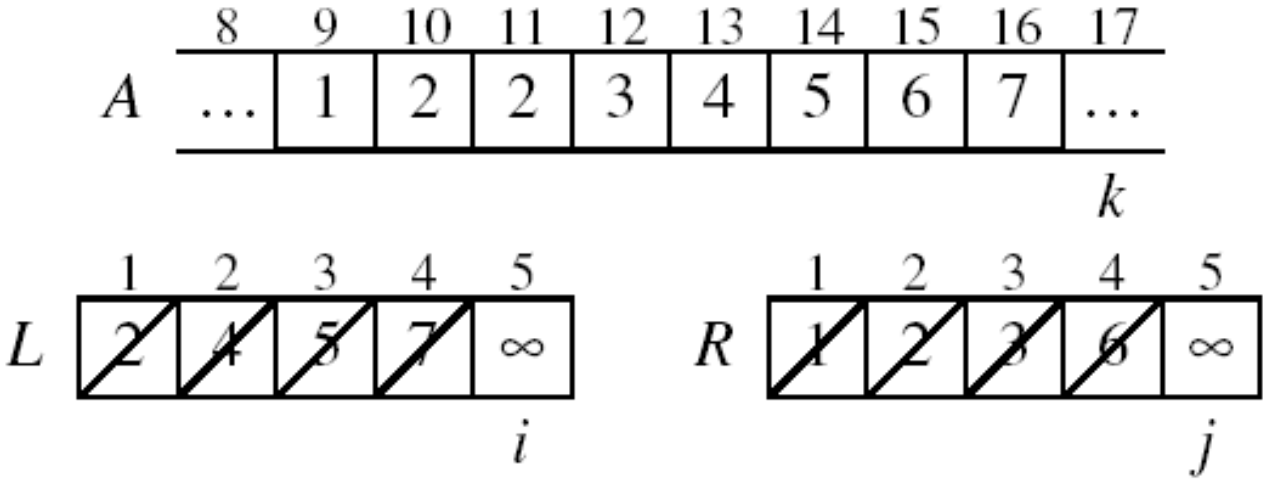




Example (cont.)

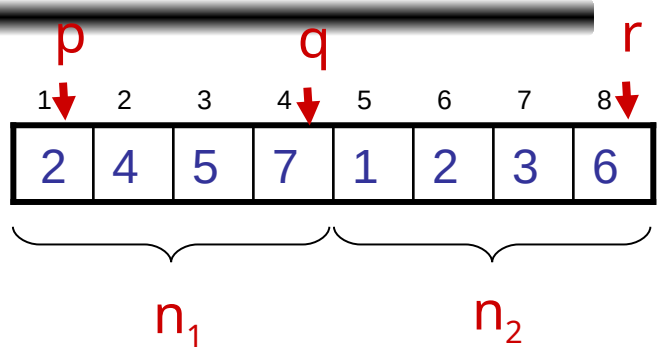


Example (cont.)



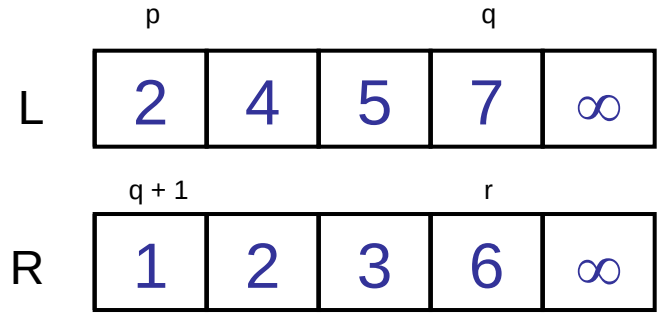
Done!

Merge - Pseudocode



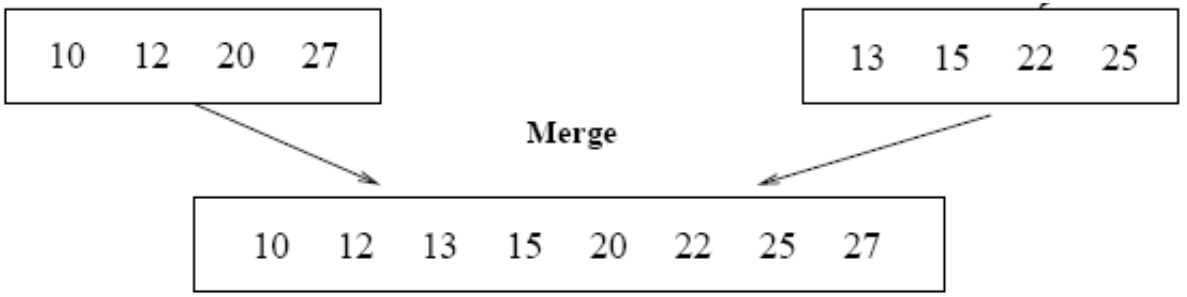
Alg.: MERGE(A, p, q, r)

1. Compute n_1 and n_2
2. Copy the first n_1 elements into L[1 . . n_1 + 1] and the next n_2 elements into R[1 . . n_2 + 1]
3. L[n_1 + 1] ← ∞; R[n_2 + 1] ← ∞
4. i ← 1; j ← 1
5. **for** k ← p **to** r
6. **do if** L[i] ≤ R[j]
7. **then** A[k] ← L[i]
8. i ← i + 1
9. **else** A[k] ← R[j]



Running Time of Merge (assume last **for** loop)

- Initialization (copying into temporary arrays):
  - $\Theta(n_1 + n_2) = \Theta(n)$
- Adding the elements to the final array:
  - n iterations, each taking constant time $\Rightarrow \Theta(n)$
- Total time for Merge:
  - $\Theta(n)$



Analyzing Divide-and Conquer Algorithms

- The recurrence is based on the three steps of the paradigm:
  - T(n) – running time on a problem of size n
  - **Divide** the problem into **a** subproblems, each of size **n/b**: takes D(n)
  - **Conquer** (solve) the subproblems aT(n/b)
  - **Combine** the solutions C(n)

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

MERGE-SORT Running Time

- **Divide:**
  - compute q as the average of p and r: $D(n) = \Theta(1)$
- **Conquer:**
  - recursively solve 2 subproblems, each of size n/2
    $\Rightarrow 2T(n/2)$
- **Combine:**
  - MERGE on an n-element subarray takes $\Theta(n)$ time
    $\Rightarrow C(n) = \Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Solve the Recurrence

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

Use Master's Theorem:

Compare n with $f(n) = cn$

Case 2: $T(n) = \Theta(n \lg n)$

Merge Sort - Discussion

- Running time insensitive of the input
- Advantages:
  - Guaranteed to run in $\Theta(n \lg n)$
- Disadvantage
  - Requires extra space $\approx N$

Sorting Challenge 1

Problem: Sort a file of huge records with tiny keys

Example application: Reorganize your MP-3 files

Which method to use?

A. merge sort, guaranteed to run in time $\sim N \lg N$
B. selection sort
C. bubble sort
D. a custom algorithm for huge records/tiny keys
E. insertion sort

Sorting Files with Huge Records and Small Keys

- Insertion sort or bubble sort?
  - NO, too many exchanges
- Selection sort?
  - YES, it takes linear time for exchanges
- Merge sort or custom method?
  - Probably not: selection sort simpler, does less swaps

Sorting Challenge 2

Problem: Sort a huge randomly-ordered file of small records

Application: Process transaction record for a phone company

Which sorting method to use?

A. Bubble sort
B. Selection sort
C. Mergesort guaranteed to run in time $\sim N \lg N$
D. Insertion sort

Sorting Huge, Randomly - Ordered Files

- Selection sort?
  - NO, always takes quadratic time
- Bubble sort?
  - NO, quadratic time for randomly-ordered keys
- Insertion sort?
  - NO, quadratic time for randomly-ordered keys
- Mergesort?
  - YES, it is designed for this problem

Sorting Challenge 3

Problem: sort a file that is already almost in order

Applications:

- Re-sort a huge database after a few changes
- Doublecheck that someone else sorted a file

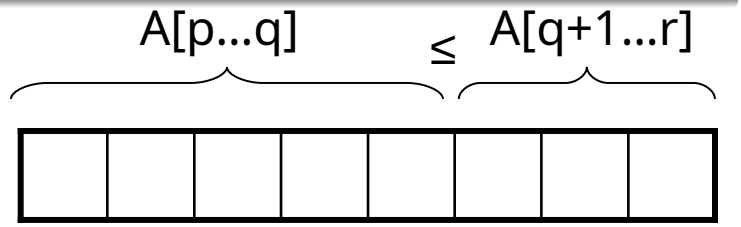
Which sorting method to use?

A. Mergesort, guaranteed to run in time $\sim N \lg N$
B. Selection sort
C. Bubble sort
D. A custom algorithm for almost in-order files
E. Insertion sort

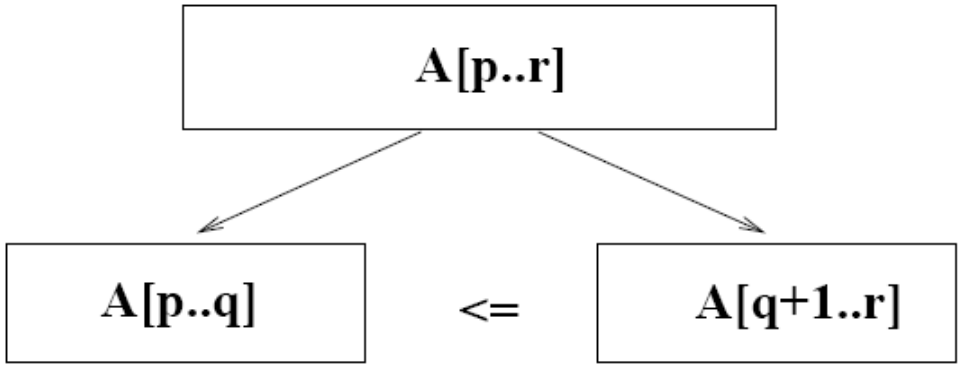
Sorting Files That are Almost in Order

- Selection sort?
  - NO, always takes quadratic time
- Bubble sort?
  - NO, bad for some definitions of “almost in order”
  - Ex: B C D E F G H I J K L M N O P Q R S T U V W X Y Z A
- Insertion sort?
  - YES, takes linear time for most definitions of “almost in order”
- Mergesort or custom method?
  - Probably not: insertion sort simpler and faster

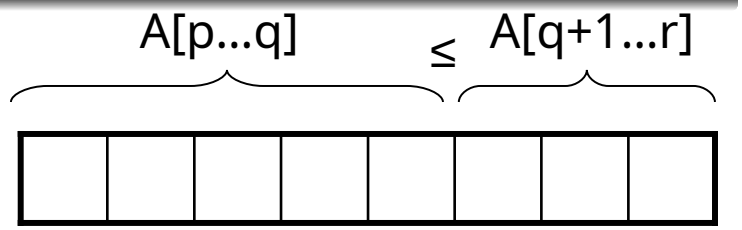
Quicksort



- Sort an array A[p...r]
- **Divide**
  - Partition the array A into 2 subarrays A[p..q] and A[q+1..r], such that each element of A[p..q] is smaller than or equal to each element in A[q+1..r]
  - Need to find index q to partition the array



Quicksort



- **Conquer**
  - Recursively sort A[p..q] and A[q+1..r] using Quicksort
- **Combine**
  - Trivial: the arrays are sorted in place
  - No additional work is required to combine them
  - The entire array is now sorted

QUICKSORT

Alg.: QUICKSORT(A, p, r)

Initially: p=1, r=n

if p < r

then q ← PARTITION(A, p, r)

QUICKSORT (A, p, q)

QUICKSORT (A, q+1, r)

Recurrence:

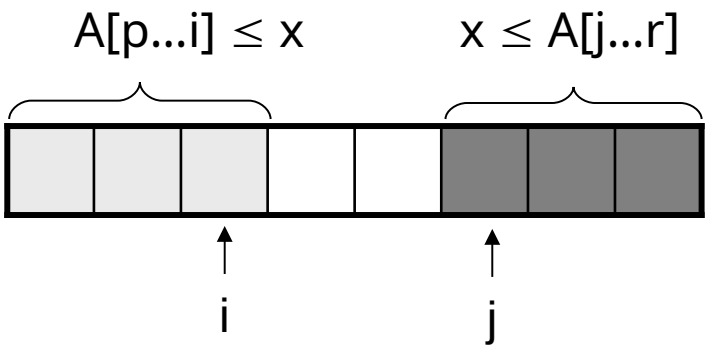
$T(n) = T(q) + T(n - q) + f(n)$ ($f(n)$ depends on PARTITION())

Partitioning the Array

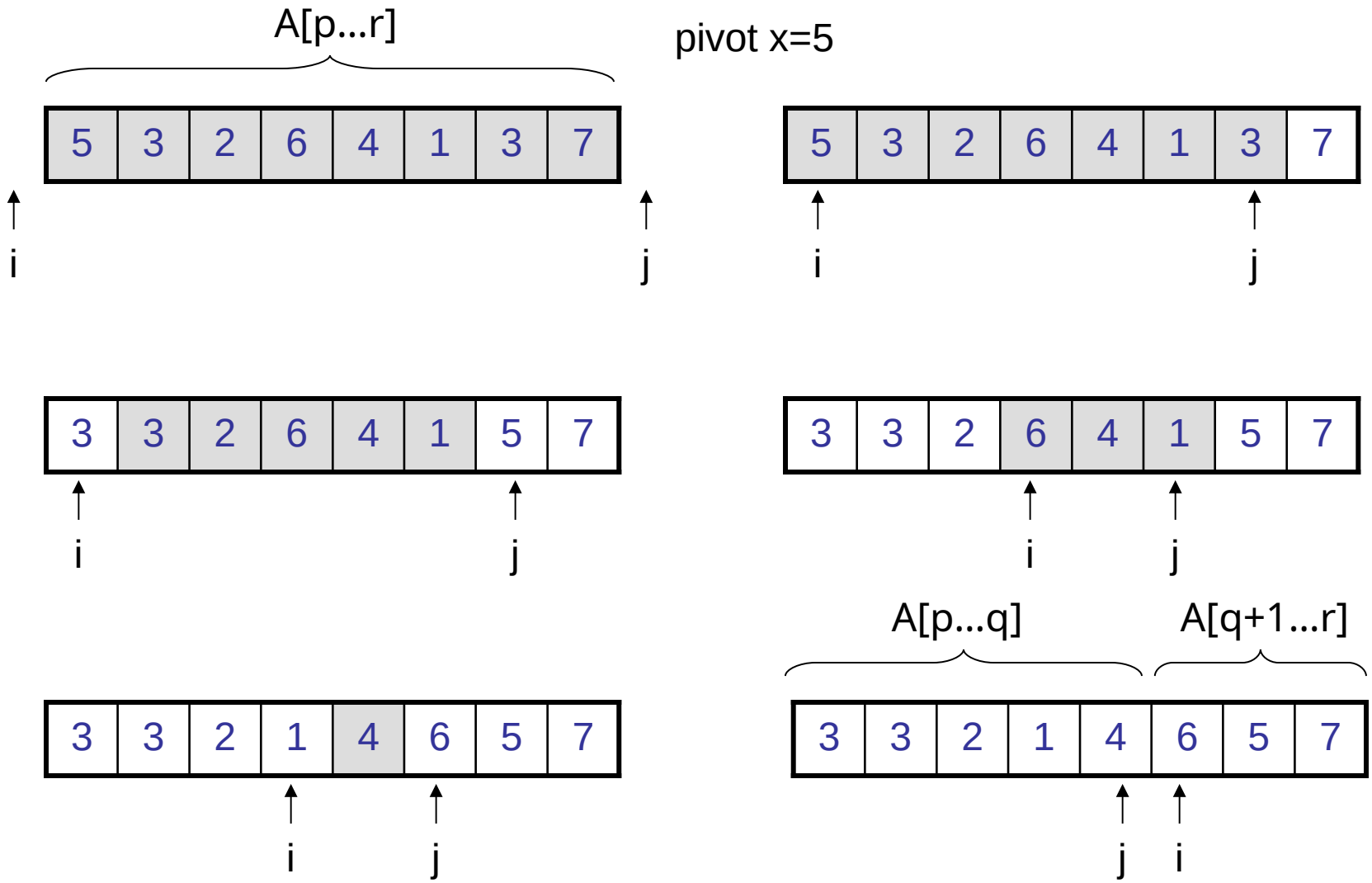
- Choosing PARTITION()
  - There are different ways to do this
  - Each has its own advantages/disadvantages
- Hoare partition (see prob. 7-1, page 159)
  - Select a pivot element **x** around which to partition
  - Grows two regions

    $A[p...i] \leq \mathbf{x}$

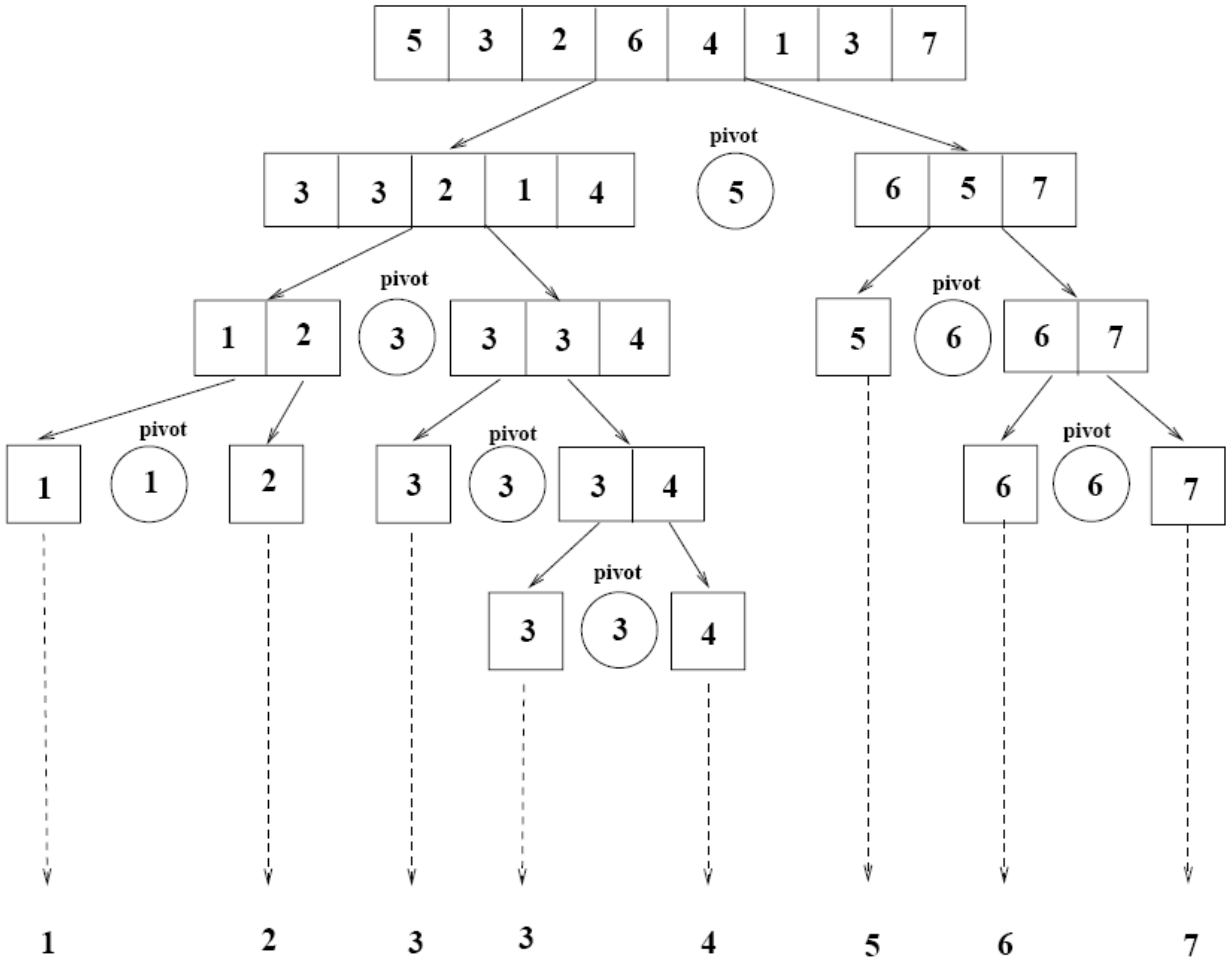
    $\mathbf{x} \leq A[j...r]$



Example


A[p...r]
pivot x=5
5 3 2 6 4 1 3 7
i
j
5 3 2 6 4 1 3 7
i
j
3 3 2 6 4 1 5 7
i
j
3 3 2 6 4 1 5 7
i
j
A[p...q]
A[q+1...r]
3 3 2 1 4 6 5 7
i
j
3 3 2 1 4 6 5 7
j
i

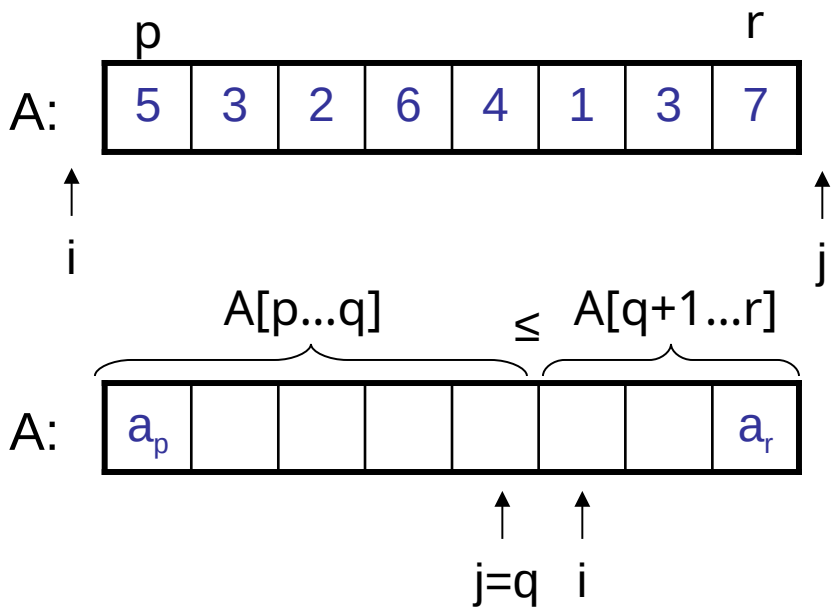
Example


5 3 2 6 4 1 3 7
pivot
5
3 3 2 1 4
6 5 7
pivot
3
1 2
3 3 4
pivot
6
5
6 7
pivot
1
1
2
3
3
3 4
pivot
6
6
7
pivot
3
3
4
1 2 3 3 4 5 6 7

Partitioning the Array

Alg. PARTITION (A, p, r)

1. $x \leftarrow A[p]$
2. $i \leftarrow p - 1$
3. $j \leftarrow r + 1$
4. **while** TRUE
5. **do repeat** $j \leftarrow j - 1$
6. **until** $A[j] \leq x$
7. **do repeat** $i \leftarrow i + 1$
8. **until** $A[i] \geq x$
9. **if** $i < j$
10. **then** exchange $A[i] \leftrightarrow A[j]$
11. **else return** j



Each element is visited once!

Running time: $\Theta(n)$
$n = r - p + 1$

Recurrence

Alg.: QUICKSORT(A, p, r)

Initially: p=1, r=n

if p < r

then q ← PARTITION(A, p, r)

QUICKSORT (A, p, q)

QUICKSORT (A, q+1, r)

Recurrence:

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

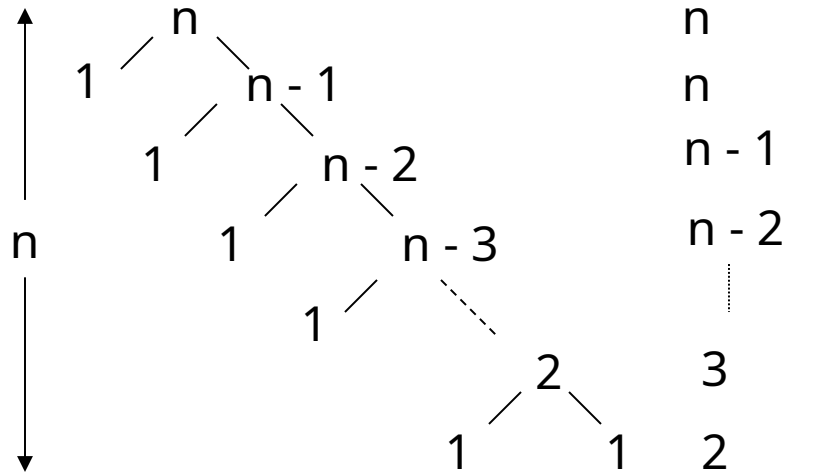
Worst Case Partitioning

- Worst-case partitioning
  - One region has one element and the other has n – 1 elements
  - Maximally unbalanced
- Recurrence: q=1

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

$T(1) = \Theta(1)$

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



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

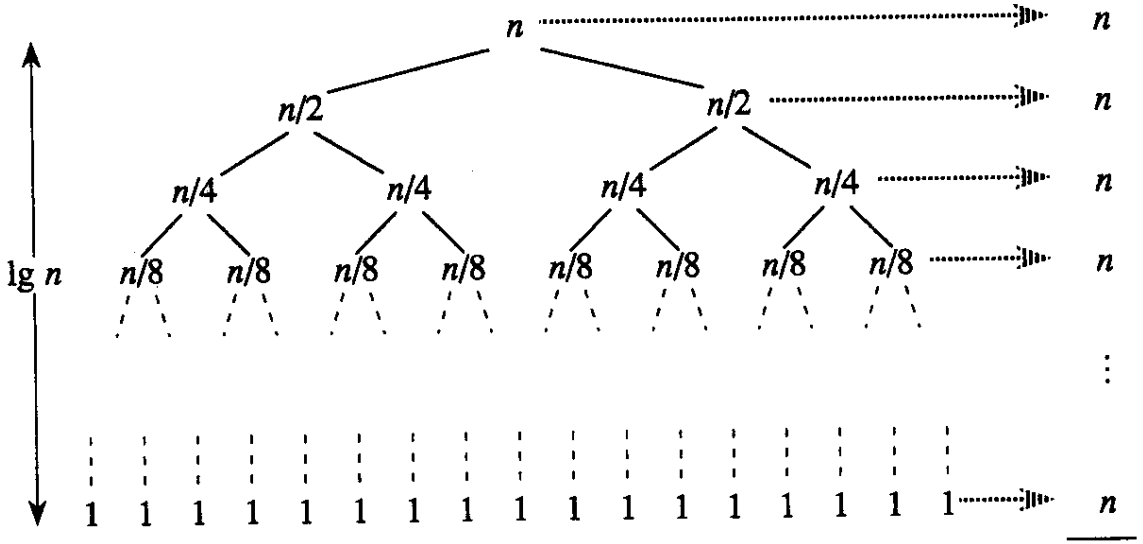
When does the worst case happen?

Best Case Partitioning

- Best-case partitioning
  - Partitioning produces two regions of size n/2
- Recurrence: q=n/2

  T(n) = 2T(n/2) + Θ(n)

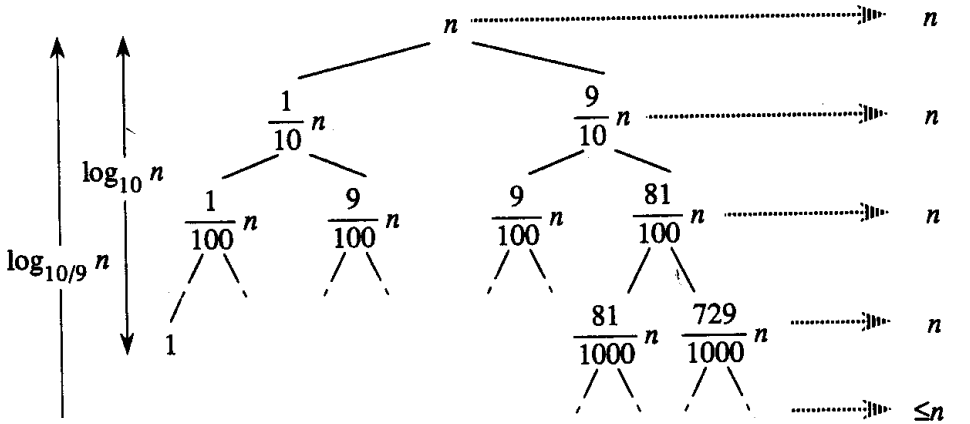
  T(n) = Θ(nlgn) (Master theorem)



Case Between Worst and Best

- 9-to-1 proportional split

$$Q(n) = Q(9n/10) + Q(n/10) + n$$



- Using the recursion tree:

longest path: $Q(n) \le n \sum_{i=0}^{\log_{10/9} n} 1 = n(\log_{10/9} n + 1) = c_2 nlgn$

shortest path: $Q(n) \ge n \sum_{i=0}^{\log_{10} n} 1 = n \log_{10} n = c_1 nlgn$

Thus, $Q(n) = \Theta(nlgn)$

How does partition affect performance?

- **Any splitting of constant proportionality** yields $\Theta(nlgn)$ time !!!

- Consider the $(1 : n-1)$ splitting:

$$\text{ratio}=1/(n-1) \text{ not a constant !!!}$$

- Consider the $(n/2 : n/2)$ splitting:

$$\text{ratio}=(n/2)/(n/2) = 1 \text{ it is a constant !!}$$

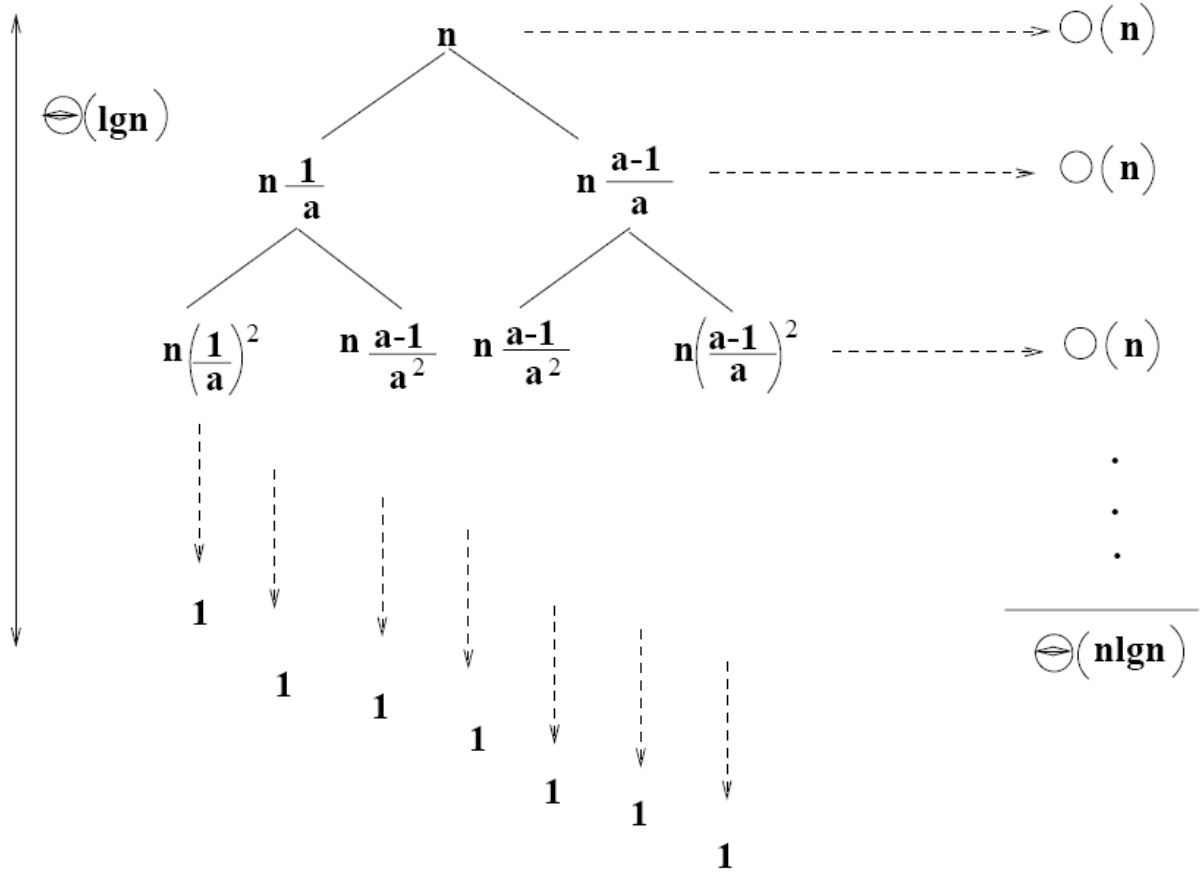
- Consider the $(9n/10 : n/10)$ splitting:

$$\text{ratio}=(9n/10)/(n/10) = 9 \text{ it is a constant !!}$$

How does partition affect performance?

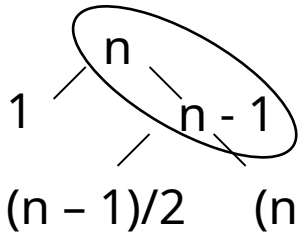
- Any $((a-1)n/a : n/a)$ splitting:

ratio=$((a-1)n/a)/(n/a) = a-1$ it is a constant !!



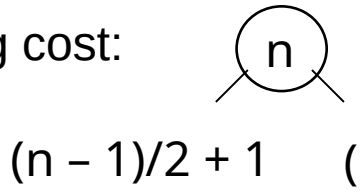
Performance of Quicksort

- Average case
  - All permutations of the input numbers are equally likely
  - On a random input array, we will have a **mix** of well balanced and unbalanced splits
  - Good and bad splits are randomly distributed across throughout the tree



combined partitioning cost:
$2n-1 = \Theta(n)$

Alternate of a good and a bad split



partitioning cost:
$n = \Theta(n)$

Nearly well balanced split

- Running time of Quicksort when levels alternate between good and bad splits is O(nlgn)

Master Method

- Many divide-and-conquer recurrence equations have the form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- The Master Theorem:

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

Master Method (Simplified)

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

$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Where $f(n)$ is $\Theta(n^c)$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

Ref Book: Sridhar
(oxford Publication)

Master Method, Example 2

- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$
- : $$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Where $f(n)$ is $\Theta(n^c)$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log n)$

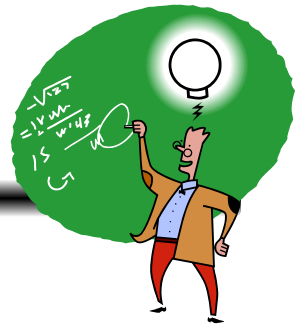
If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

- Example:

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

Solution: $\log_b a = 1$, so case 2 says T(n) is $O(n \log^2 n)$.

Master Method, Example 3



- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$
- The Master Theorem:

$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Where $f(n)$ is $\Theta(n^c)$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log n)$

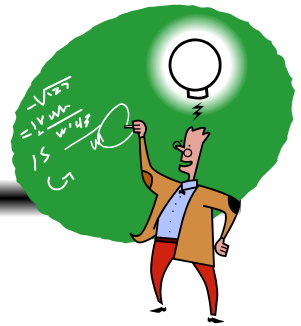
If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

- Exampl

$$T(n) = T(n/3) + n \log n$$

Solution: $\log_b a = 0$, so case 3 says T(n) is O(n log n).

Master Method, Example 4



- The form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

- The Master Theorem:

$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Where $f(n)$ is $\Theta(n^c)$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log n)$

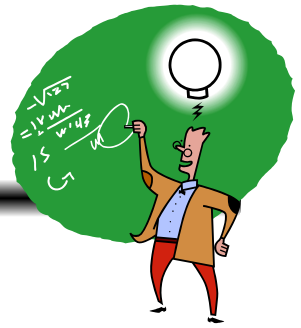
If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

- Example:

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

Solution: $\log_b a = 3$, so case 1 says T(n) is $O(n^3)$.

Master Method, Example 5

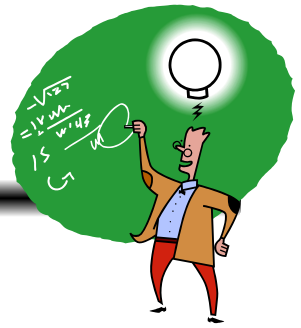


- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example:

$$T(n) = 9T(n/3) + n^3$$

Solution: $\log_b a = 2$, so case 3 says T(n) is $O(n^3)$.

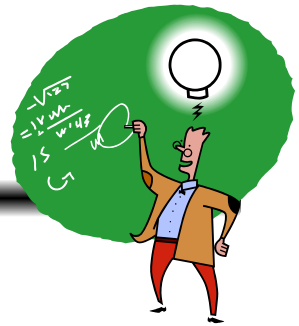
Master Method, Example 6



- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example: $$T(n) = T(n/2) + 1$$ (binary search)

Solution: $\log_b a = 0$, so case 2 says T(n) is O(log n).

Master Method, Example 7



- The form: $$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$
- The Master Theorem:
  1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
  2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
  3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.
- Example: $$T(n) = 2T(n/2) + \log n$$ (heap construction)

Solution: $\log_b a = 1$, so case 1 says T(n) is O(n).

Recurrence to Big-Θ

$$T(n) = \begin{cases} 2 & \text{if } n < 3 \\ 2T\left(\frac{n}{3}\right) + n & \text{otherwise} \end{cases}$$

- It's still really hard to tell what the Big-Θ is just by looking at it.
- But fancy mathematicians have a formula for us to use!

$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Where $f(n)$ is $\Theta(n^c)$

If $\log_b a <$ then $T(n) \in \Theta(n^c)$

If $\log_b a =$ then $T(n) \in \Theta(n^c \log n)$

If $\log_b a >$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

a=2 b=3 and c=1

$y = \log_b x$ *is equal to* $b^y = x$

$\log_3 2 \cong 0.63$

$\log_3 2 < 1$

We're in case 1

$T(n) \in \Theta(n)$

Aside Understanding the Master Theorem

$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Where $f(n)$ is $\Theta(n^c)$

If $\log_b a <$ then $T(n) \in \Theta(n^c)$

If $\log_b a =$ then $T(n) \in \Theta(n^c \log n)$

If $\log_b a >$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

- a measures how many recursive calls are triggered by each method instance
- b measures the rate of change for input
- c measures the dominating term of the non recursive work within the recursive method
- d measures the work done in the base case

- The case $\log_b a < c$
  - Recursive case does a lot of non recursive work in comparison to how quickly it divides the input size
  - Most work happens in beginning of call stack
  - Non recursive work in recursive case dominates growth, n^c term
- The case $\log_b a = c$
  - Recursive case evenly splits work between non recursive work and passing along inputs to subsequent recursive calls
  - Work is distributed across call stack
- The case $\log_b a > c$
  - Recursive case breaks inputs apart quickly and doesn't do much non recursive work
  - Most work happens near bottom of call stack

Merge Sort Recurrence to Big-Θ

$$T(n) = \begin{cases} 1 & \text{if } n \leq 3 \\ 2T\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases}$$

$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Where $f(n)$ is $\Theta(n^c)$

If $\log_b a <$ then $T(n) \in \Theta(n^c)$

If $\log_b a =$ then $T(n) \in \Theta(n^c \log n)$

If $\log_b a >$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

a=2 b=2 and c=1

$\log_2 2 = 1$

We're in case 2

$T(n) \in \Theta(n \log n)$

Proof Techniques

- Proof is a kind of demonstration to convince that the given mathematical statement is true.
- The statement which is to be proved is called **theorem**. Once a particular theorem is proved then it can be used to prove further statements.
- The theorem is also called as **Lemma**.
- The proof can be a deductive proof or inductive proof.
- The deductive proof consist of sequence of statements given with logical reasoning.
- The inductive proof is a recursive kind of proof which consists of sequence or parameterized statements that use the statement itself or the statement with lower values of its parameter.
- Various methods of proofs are-
  - Proof by contradiction
  - Proof by mathematical induction
  - Direct proofs
  - Proof by counter example
  - Proof by contraposition

Proof by Contradiction

- In this type of proof, for the statement of the form if A then B.
- We start with statement A is not true and thus by assuming false A, we try to get the conclusion of statement B.
- When it becomes impossible to reach to statement B, we contradict our self and accept that A is true.
- For example,
- Prove $P \cup Q = Q \cup P$

Prove $P \cup Q = Q \cup P$

- Proof
  - Initially we assume that $P \cup Q = Q \cup P$ is not true.
    i.e. $P \cup Q \neq Q \cup P$
  - Now consider that x is in Q, or x is in P. hence we can say x is in $P \cup Q$ (according to definition of union)
  - But this also implies that x is in $Q \cup P$ according to definition of union.
  - Hence the assumption which we made initially is false.
  - Thus $P \cup Q = Q \cup P$ is proved.

Prove by contradiction. There exist two irrational numbers x and y such that x^y is rational.

- Solution:
- An irrational number is any number that cannot be expressed as a/b where a and b are integers and value b is non zero. To prove that x^y is rational when x and y are irrational we have two choices –
- 1. x^y is rational
- 2. x^y is irrational
- Case 1 : 2 is a rational number then x = 2 and y = 2 is a irrational number, hence there exists two irrational numbers x and y such that x^y is rational

- Case 2: 2^{2} is irrational. We will consider two irrational numbers.
  - $X = 2^{2}$ and $y = 2$
  - $X^{y} = (2^{2})^{2}$
  - $= (2)^{2\,2}$
  - $= (2)^{2}$
  - $= 2$
  - Which is a rational number. Here we have x and y as irrational numbers but 2 as rational number.
  - From the two cases it it proved that if x and y are two irrational numbers then x^{y} is a rational number.

Proof by Mathematical Induction

- Inductive proofs are special proofs based on some observations.
- It is used to prove recursively defined objects. This type of proof is also called as proof by mathematical induction.
- The proof by mathematical induction can be carried out using following steps:
  - Basic: in this step, we assume the lowest possible value. This is an initial step in the proof by mathematical induction.

    For example, we can prove that the result is true for n = 0 or n = 1
  - Induction Hypothesis: in this step, we assign value of n to some other value k. that mean, we will check whether the result is true for n – k or not.
  - Inductive Step: in this step, if n = k is true then we check whether the result is true for n = k + 1 or not.
  - If we get the same result at n = k + 1 then we can state that given proof is true by principle of mathematical induction

Example : 1
Prove: 1 + 2 + 3 …. + n = n (n + 1) / 2

- Solution: initially,
- 1) Basis of induction –
- Assume, n = 1 then
- L. H. S. = n = 1
- R. H. S. = n (n + 1) / 2 = 1 (1 + 1) / 2 = 2 / 2 = 1
- 2) Induction hypothesis –
- Now we will assume n = K and will obtain the result for it. The equation then becomes,
- 1 + 2 + 3 + ….. + K = K (K + 1) / 2

- 3) Inductive step –
- Now we assume that equation is true for n = K and we will then check if it is also true for n = K +1 or not.
- Consider the equation assuming n = K + 1
- L. H. S. = $\underline{1 + 2 + 3 + \ldots + K} + K + 1$
- $= K (K + 1) / 2 + K + 1$
- $= K (K + 1) + 2 (K + 1) / 2$
- $= (K + 1) (K + 2) / 2$
- i.e. $= (K + 1) (K + 1 + 1) / 2$
- $= R. H. S.$

Example 2:
Prove : $n! >= 2^{n-1}$

- Solution: Consider,
- 1) Basis of induction -
- Let n = 1 then
- L. H. S. = 1
- R. H. S. = $2^{1-1} = 2^{0} = 1$
- Hence, $n! >= 2^{n-1}$ is proved.

- 2) Induction hypothesis-
- Let n = n +1 then
- $k! = 2^{k-1}$ where k >= 1
- Then
- $(k+1)! = (k+1)\,k!$ By definition of n!
- $= (k+1)\,2^{k-1}$
- $= 2 * 2^{k-1}$
- $= 2^{k}$
- Hence, $n! >= 2^{n-1}$ is proved.

Direct Proofs

- In direct proof, the intended proof can be proved by basic principle or axiom.
- Example - Prove that the negative of any even integer is even.
- Solution : to prove this, let n be any positive even number. Hence we can write n as
- $n = 2m$ where m can be any number
- If we multiply both side by -1, we get
- $-n = -2m$
- $-n = 2(-m)$
- Multiplying any number by 2 makes it an even number.
- Hence, -n is even.
- Thus proves that the negative of any even integer is even.

Proof by Counter-example

- In order to prove certain statements, we need to see **all possible conditions** in which that statement remains true.
- There are some situations in which the statement can not be true.
- **For example: Theorem**: there is no such pair of integers such that
- a mod b = b mod a
- **Proof**: consider a = 2 and b =3 then 2 mod 3
- Thus the given pair is true for any pair of integers but
- if a =b then naturally a mod b = b mod a
- Thus we need to change the statement slightly. We can say
- a mod b = b mod a, when a =b
- This type of proof is called **counter example**.
- Such proof is true only at some specific condition.

Proof by Contraposition

- This is a technique of proof in which A → B is true if ~ A → ~ B.
- If negative statement of given statement is true then the given statement becomes automatically true.
- Example: prove by contraposition that x + 8 is odd.
- Solution: Step 1: we assume that x is not odd
- Step 2: that means x is even. By definition of even numbers 2 * any number = even number
- x = 2 * m where m can be any number
- Step 3: we can write x + 8 as 2 * m + 8 = 2 (m + 4) = even number
- Thus x + 8 is even. That means (x + 8) is not odd.
- From step 1 and 3, we can state that if x is not odd then (x +8) is also not odd.
- Hence by contraposition theorem, we can say that x + 8 is odd if x is odd.

Design and Analysis of Algorithm

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DAA Objectives & Outcomes

Course Objectives:

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

Course Outcomes:

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

Course prerequisite

Data Structure II

C, C++, Java or other programming languages

MIT-WPU
॥ विश्वशान्तिर्ध्रुवं ध्रुवा ॥

REF BOOK: THOMAS CORMEN

Module I - Fundamentals of Algorithm

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

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

(substitution method & Master's theorem)

Why Study this Course?

REF BOOK: THOMAS CORMEN

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

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

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

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

Real Blend of Creativity and Precision

Very interesting if you can concentrate on this course

Algorithm: A brief History

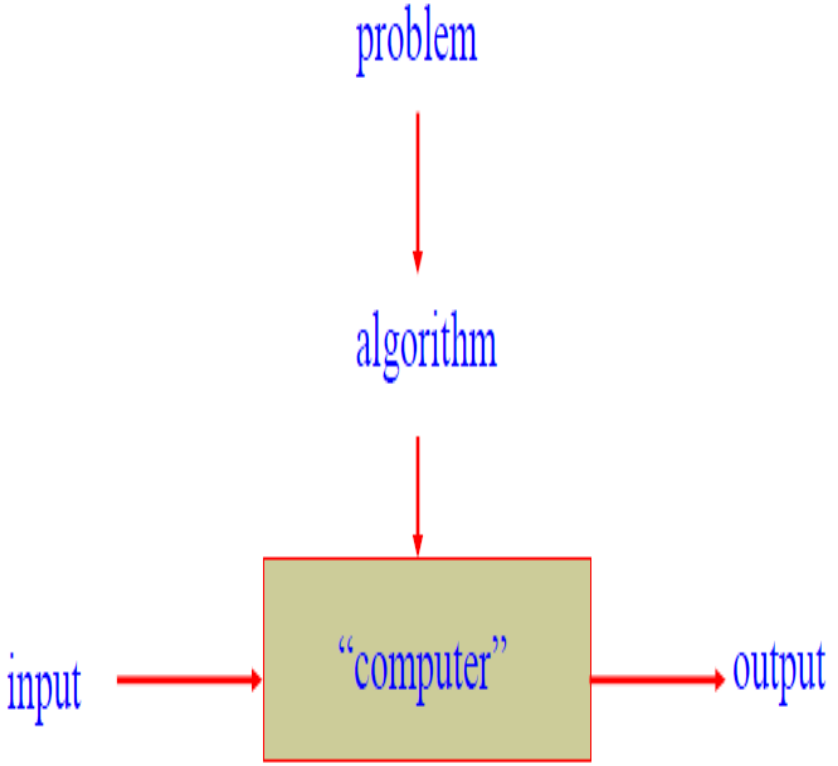
- The word algorithm comes from the name of **Persian author, Abu Ja'far Mohammed ibn Musa al Khowarizmi (c. 825 A.D.)**, who wrote a textbook on mathematics.
- The book was translated into Latin in the 12th century under the title **Algoritmi de numero Indorum.** This title means "Algoritmi on the numbers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

REF BOOK: THOMAS CORMEN

What is an Algorithm?

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

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



Algorithm & its Properties

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

Algorithm must satisfy the following properties:

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

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

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

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

Step1: factorize m.(m m1*m2*m3…)

Step2: factorize n. (n n1*n2*…)

Step 3: Identify common factors , multiply and return.

Pseudocode of Euclid's Algorithm

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

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

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

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

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

r ← n mod m

n ← m

m ← r

return m

Questions:

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

Definiteness: non-ambiguity

Effectiveness: effectively computable.

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

Example of Euclid's Algorithm

Simple Algorithm

m = 36, n = 48

m = 2 * 2 * 3 * 3

n = 2 * 2 * 2 * 2 * 3

Common factors

2, 2, 3

GCD = 12

9 divisions

Euclid Algorithm

36 divides 48

r = 48 mod 36

n = 36

m = 12

Return 12

2 divisions

- Another example:
  m = 434 and n = 966

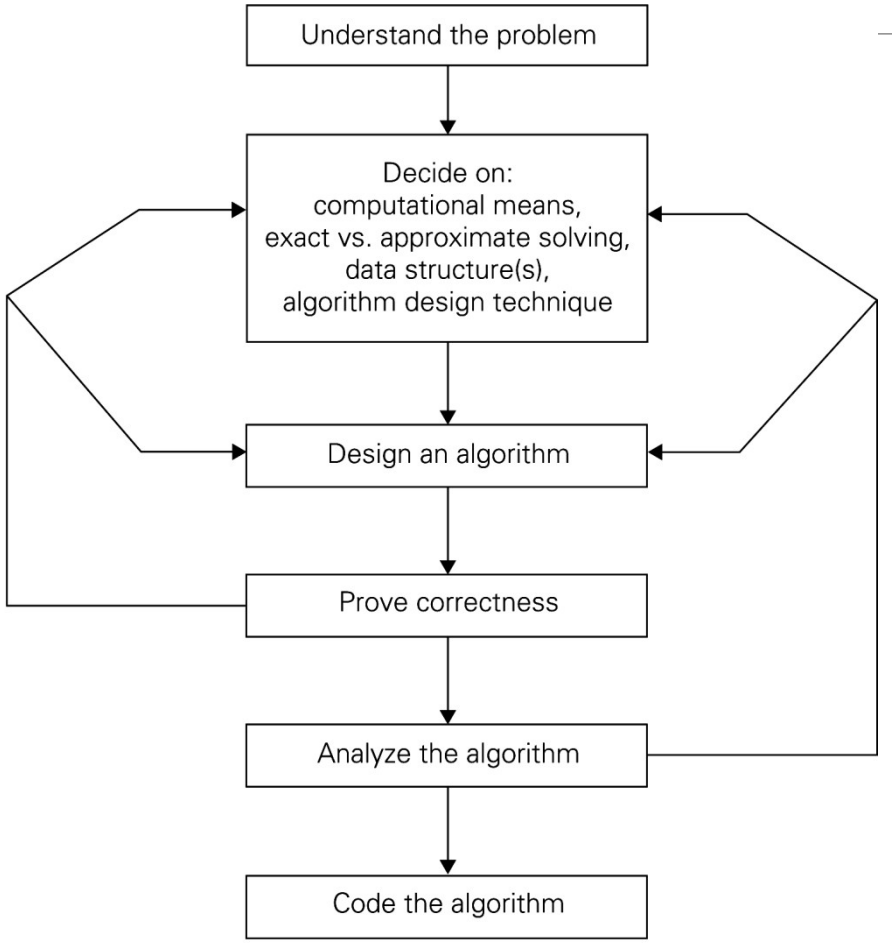
REF BOOK: THOMAS CORMEN

Fundamentals of Algorithmic Problem Solving

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

REF BOOK: THOMAS CORMEN

Algorithm Design and Analysis Process



Algorithms as a Technology

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

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

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

Memory is cheap but not free

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

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

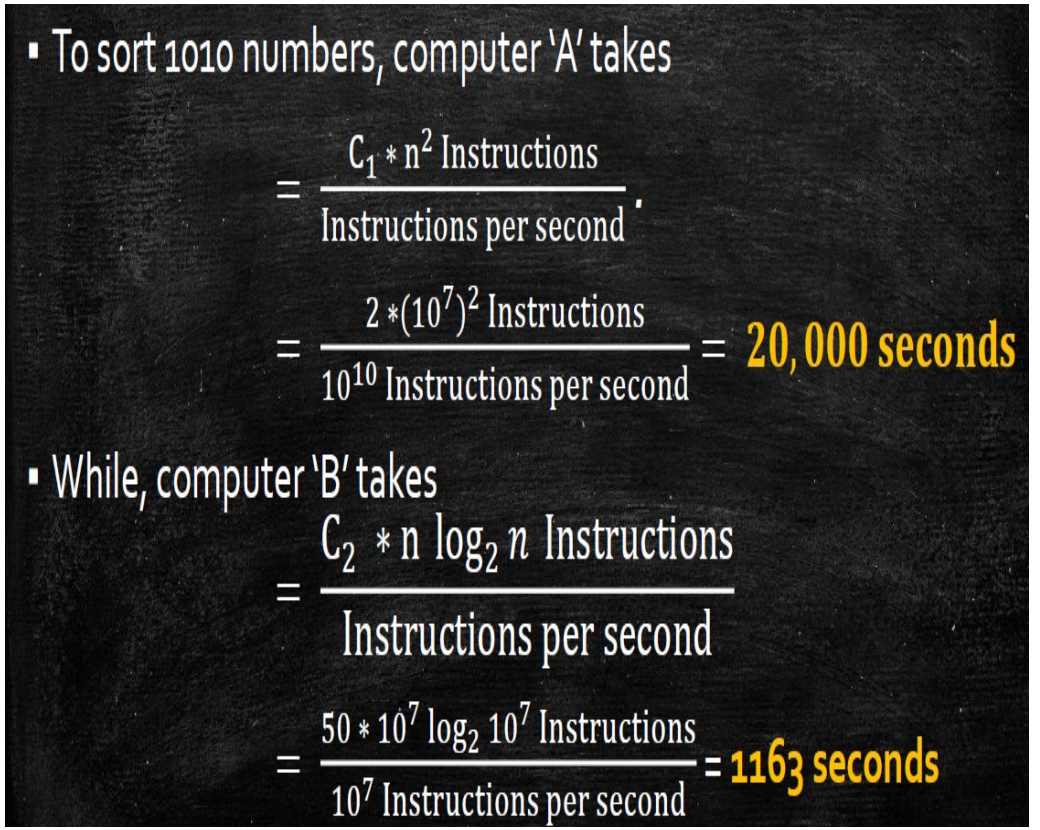
REF BOOK: THOMAS CORMEN

Algorithm Efficiency

Time and space efficiency are the goal

Algorithms often differ dramatically in their efficiency

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



Efficiency

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

Methods of Specifying an Algorithm

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

Pseudocode Conventions

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

Pseudocode Conventions

- Looping statement can be employed as follows :

(while, for, repeat until)

While Loop:

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

For Loop:

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

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

i=1,3,5,7,9

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

A conditional statement has the following forms:

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

We also employ the following case statement:

case

{

: ⟨condition 1⟩: ⟨statement 1⟩

.

: ⟨condition n⟩: ⟨statement n⟩

: **else**: ⟨statement n + 1⟩

}

Pseudocode Conventions

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

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

Algorithm Name (⟨parameter list⟩)

{

// body

}

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

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

REF BOOK: THOMAS CORMEN

Analysis of Algorithms

Two main issues related to algorithms

- How to design algorithms
- How to analyze algorithm efficiency

Analysis of Algorithms

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

Common Rates of Growth

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

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

REF BOOK: NPTEL

Why Analyse?

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

What to analyse?

Core Issue: Cannot control what we cannot measure

Time

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

Space

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

Examples : Sum of Natural Numbers

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

Time T(n) = n (additions)

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

Tabular Method

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

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

Machine Model: Random Access Machine (RAM) Computing Model

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

REF BOOK: THOMAS CORMEN

Asymptotic Analysis

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

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

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

REF BOOK: THOMAS CORMEN

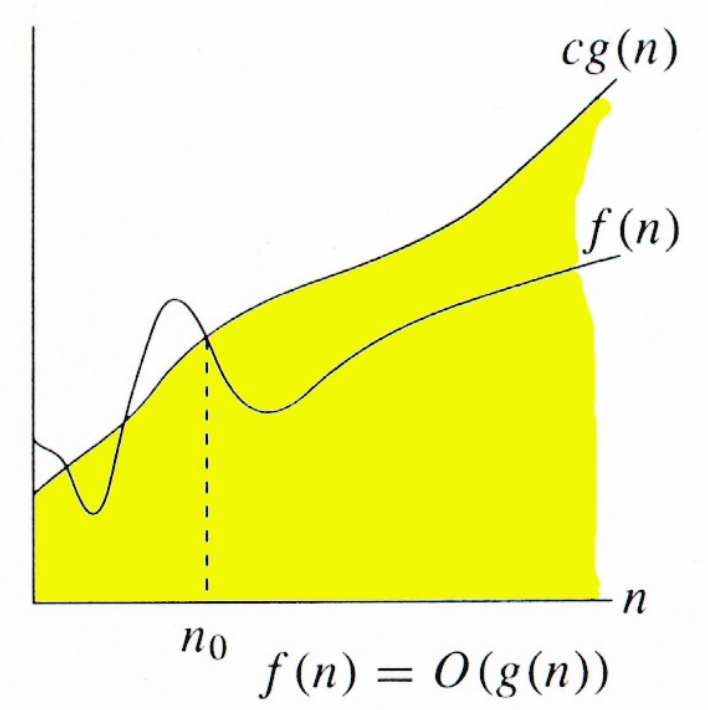
Asymptotic Upper Bound

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

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

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

Example :



Big Oh - Example

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

$g(n) = n^2$ ……. c = 2

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

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

Big Oh - Example

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

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

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

Big Oh - Example

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

Proof.

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

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

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

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

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

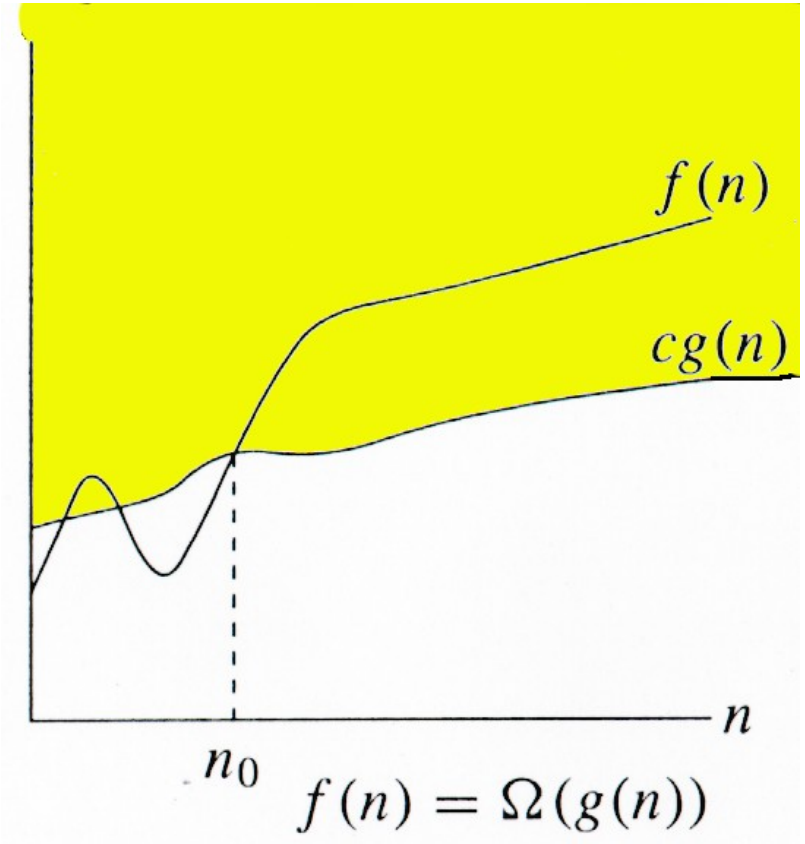
ASYMPTOTICS NOTATIONS
Ω-notation

REF BOOK: THOMAS CORMEN

Asymptotic lower bound

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

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



Big Omega - Example

Example 1 :

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

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

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

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

Example 1 :

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

Proof.

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

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

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

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

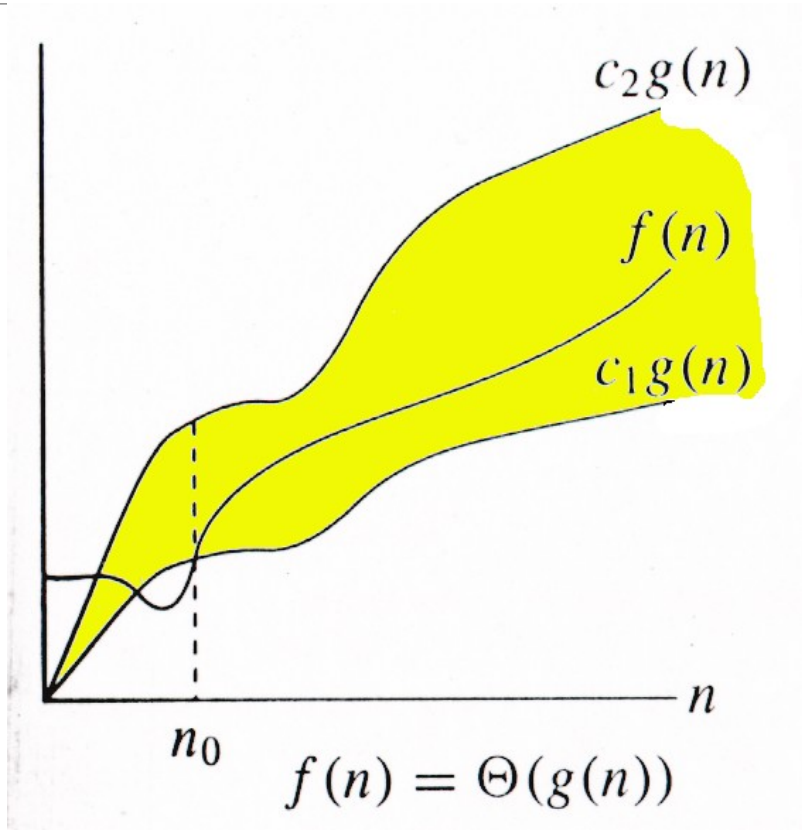
ASYMPTOTICS NOTATIONS
Θ-notation

REF BOOK: THOMAS CORMEN

Asymptotic tight bound

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

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



Theta Example

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

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

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

Theta means the bound is the tightest possible.

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

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

o notation (Little oh)

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

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

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

ω notation (Little omega)

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

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

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

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

REF BOOK: THOMAS CORMEN

Asymptotic Notations

It is a way to compare “sizes” of functions

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

Θ-notation ----------Equal to (“=“)

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

$$O \approx \leq$$

$$\Omega \approx \geq$$

$$\Theta \approx =$$

$$o \approx <$$

$$\omega \approx >$$

REF BOOK: THOMAS CORMEN/SAHNI

Examples On Asymptotic Notations

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

Show that

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

Algorithm Design Techniques/Strategies

REF BOOK: THOMAS CORMEN

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

Divide & Conquer

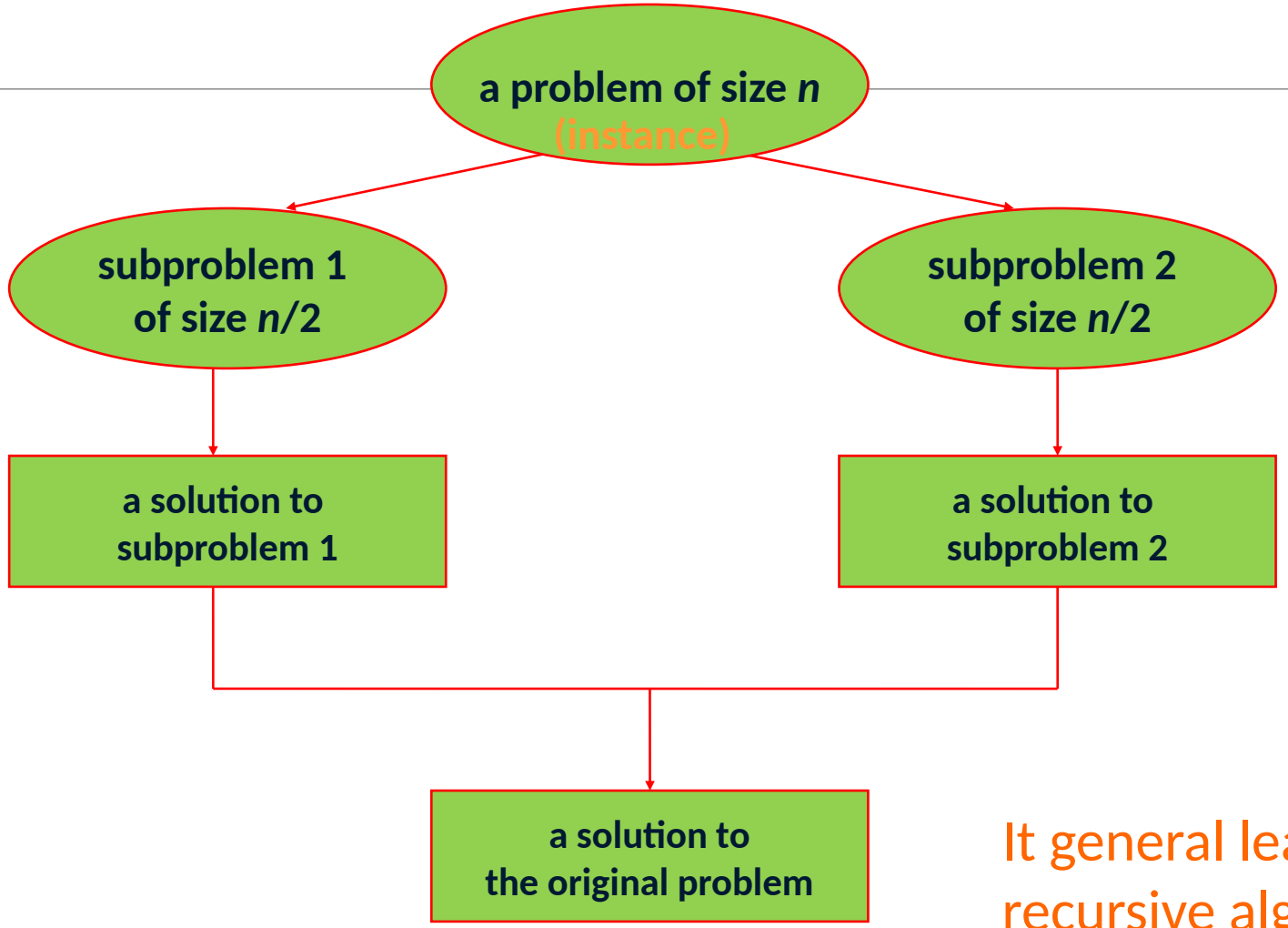
Control Abstraction

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

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

Divide-and-Conquer Technique (cont.)

REF BOOK: INTERNET



It general leads to a recursive algorithm!

Time complexity of the general algorithm

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

Special techniques are required to analyze the space and time required

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

Time complexity (recurrence relation):

where D(n) : time for splitting

C(n) : time for conquer

c : a constant

REF BOOK: THOMAS CORMEN

Methods for Solving recurrences

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

REF BOOK: INTERNET

Math You need to Review

properties of logarithms:

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

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

$\log_b xa = a\log_b x$

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

- **properties of exponentials**:

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

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

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

$b = a^{\log_a b}$

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

REF BOOK: THOMAS CORMEN

Substitution Method

- Sum of n Natural Numbers

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

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

Solve Recurrence in Long hand:

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

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

- Factorial (Recursive)

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

- Time $T(n)$ (multiplication)

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

$$T(n) = n$$

Quick Sort

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

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

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

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

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

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

Binary Search

EXAMPLE 2: BINARY SEARCH

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

$T(1)=1$

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

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

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

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

$= \log n$

T(n)= O(logn)

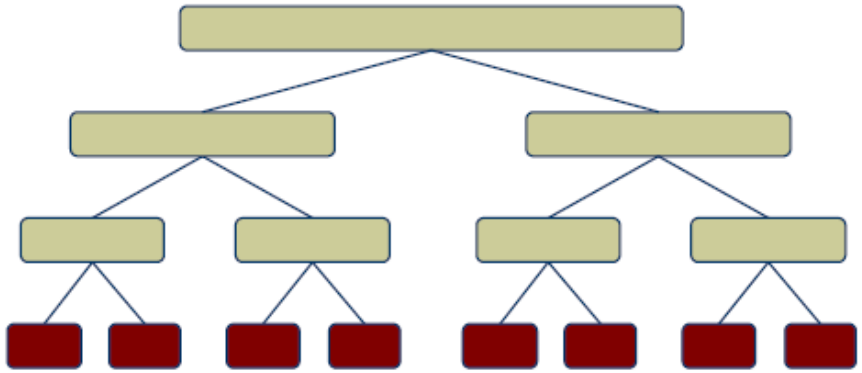
REF BOOK: THOMAS CORMEN

The Recursion Tree

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

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

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



Total time = $bn + bn \log n$

(last level plus all previous levels)

REF BOOK: THOMAS CORMEN

Master Theorem

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

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

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

There are 3 cases:

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

Example of Master Method

REF BOOK: THOMAS CORMEN

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

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

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

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

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

REF BOOK: SRIDHAR (OXFORD PUBLICATION)

Master Method (Simplified)

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

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

$$T(1) = c$$

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

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

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

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

REF BOOK: THOMAS CORMEN/SAHNI

Divide-and-Conquer Examples

Sorting : Mergesort and Quicksort

Binary tree traversals

Binary search

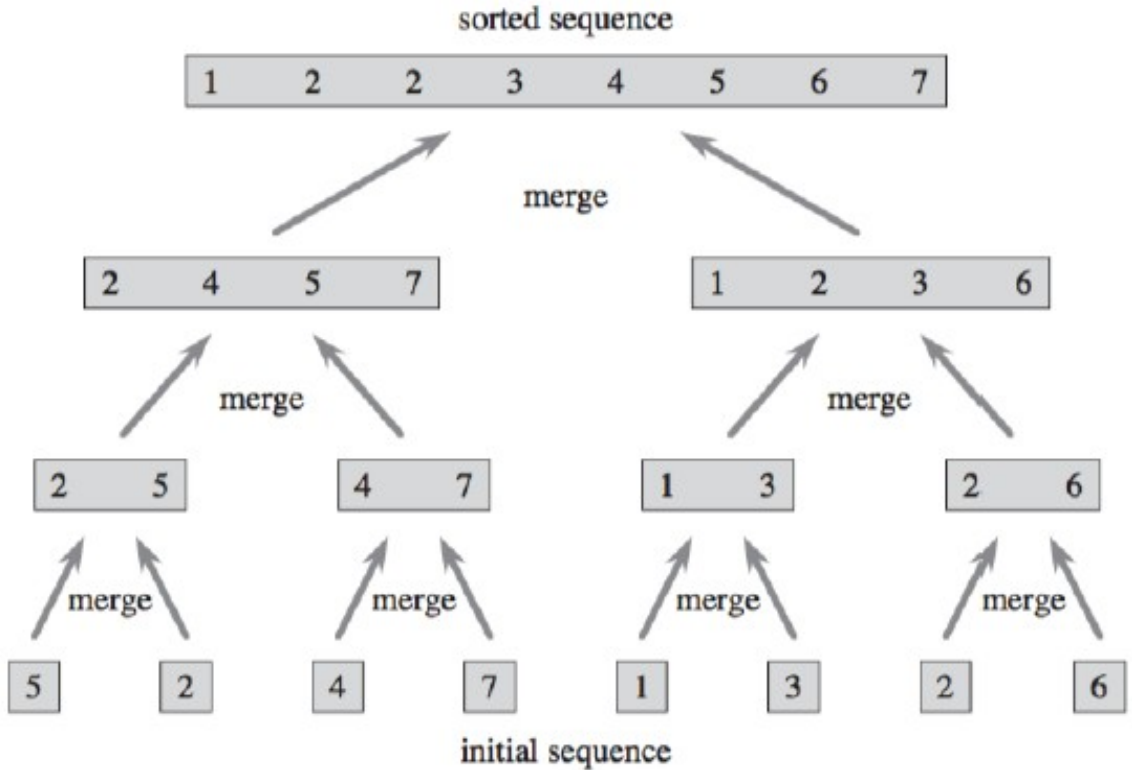
Multiplication of large integers

Matrix multiplication: Strassen’s algorithm

Closest-pair and convex-hull algorithms

Mergesort

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



REF BOOK: THOMAS CORMEN

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

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

OUTPUT: an ordered sequence of *n* numbers

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

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

Procedure Merge

REF BOOK: THOMAS CORMEN

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

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

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

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

Analysis of Mergesort

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

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

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

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

Total:

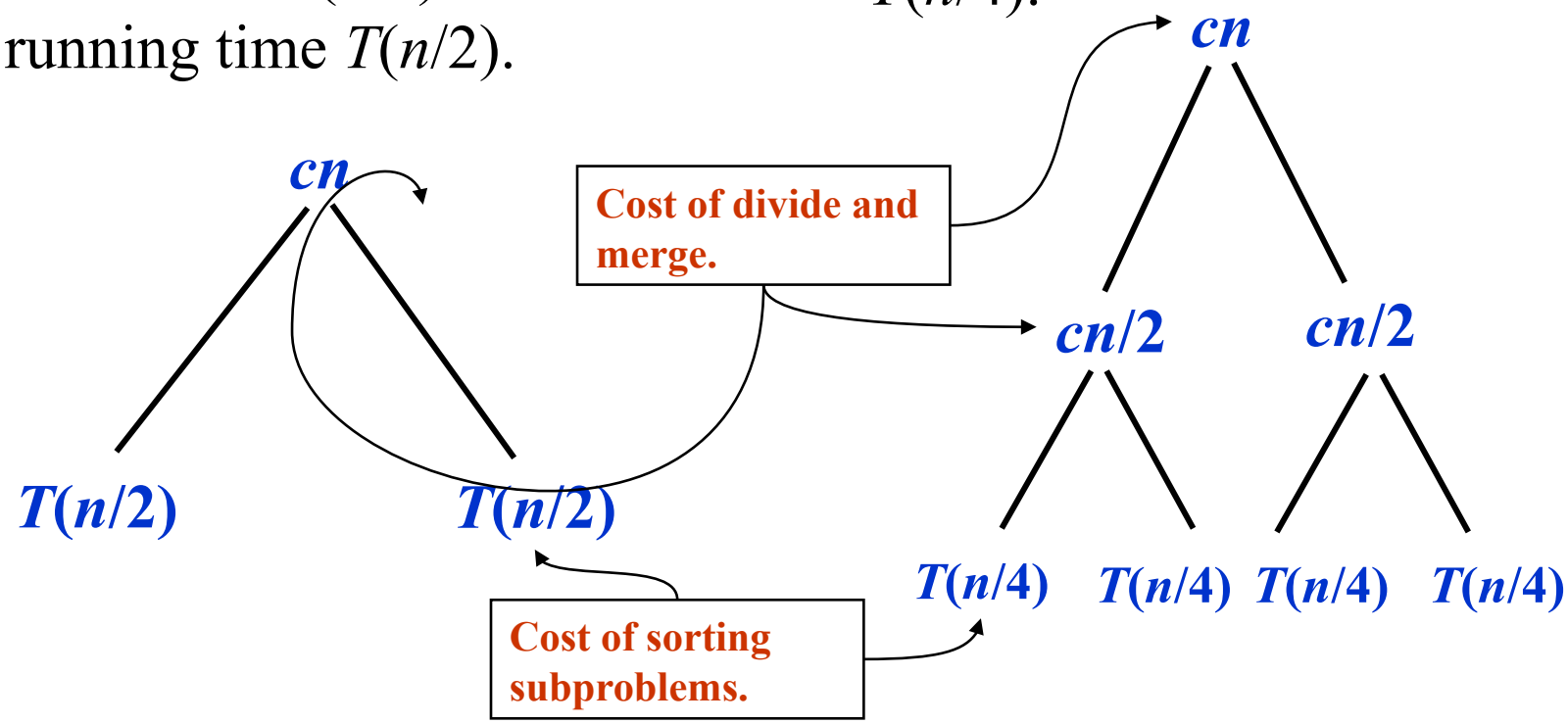
$$T(n) = \Theta(1) \quad \text{if } n = 1$$
$$T(n) = 2T(n/2) + \Theta(n) \quad \text{if } n > 1$$

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

Recursion Tree for Merge Sort

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

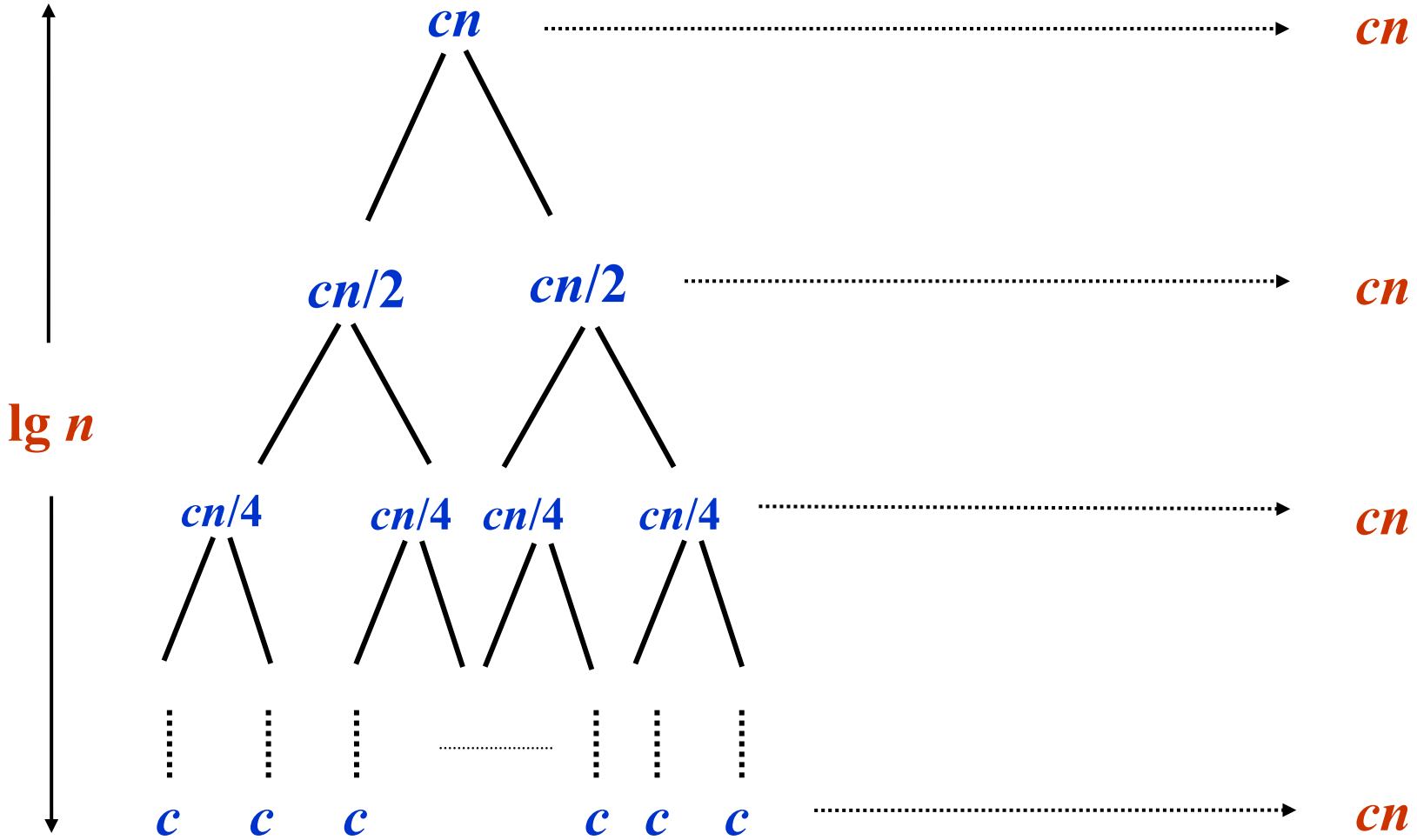
Each of the size $n/2$ problems has a cost of $cn/2$ plus two subproblems, each costing $T(n/4)$.



Recursion Tree for Merge Sort

Ref Book: Thomas Cormen

Continue expanding until the problem size reduces to 1.

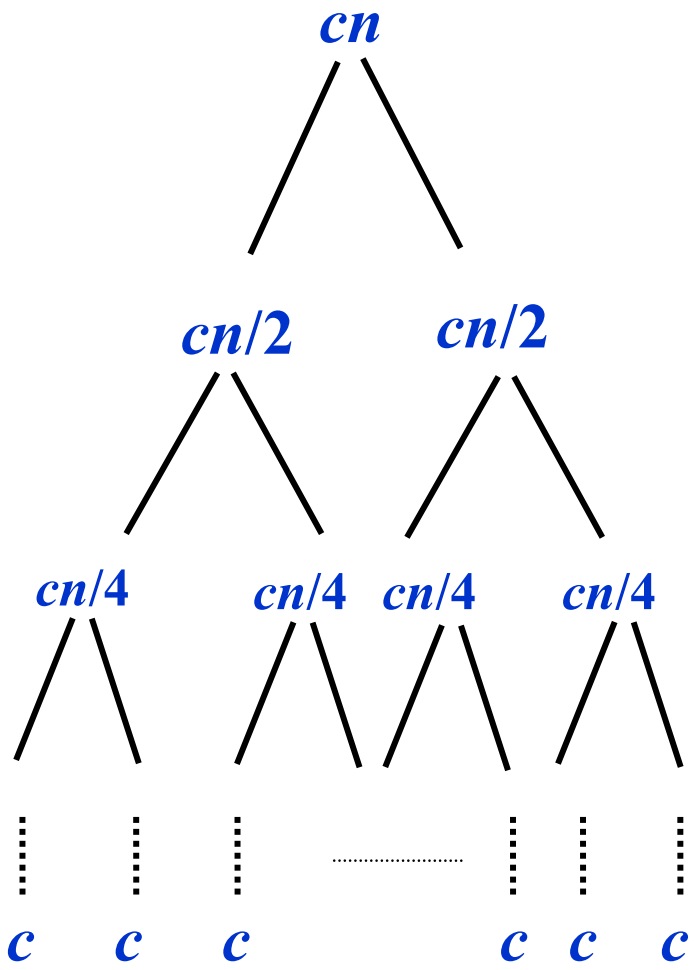


Total : $cn\lg n + cn$

Recursion Tree for Merge Sort

Ref Book: Thomas Cormen

Continue expanding until the problem size reduces to 1.



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

Ref Book: Thomas Cormen

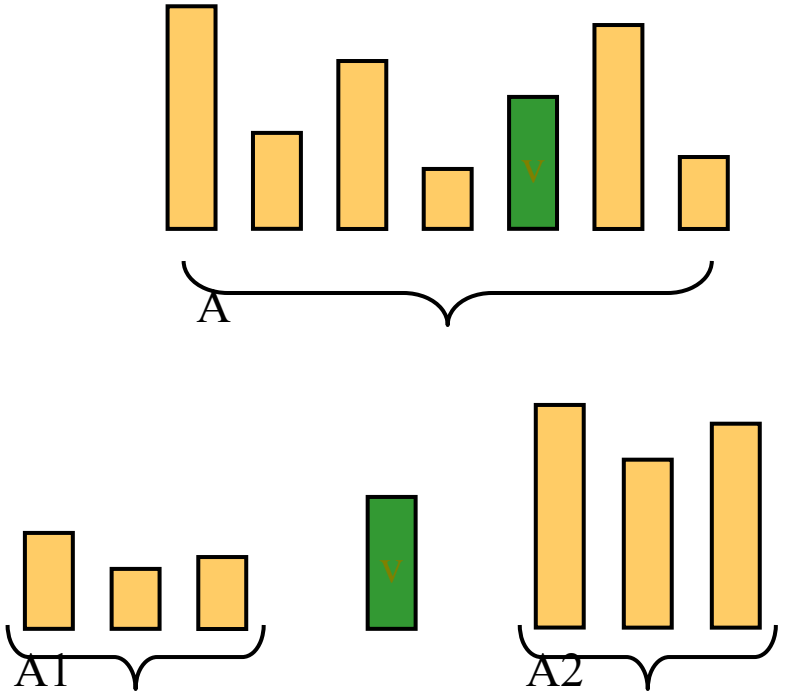
Quicksort

Ref Book: Thomas Cormen

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

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

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



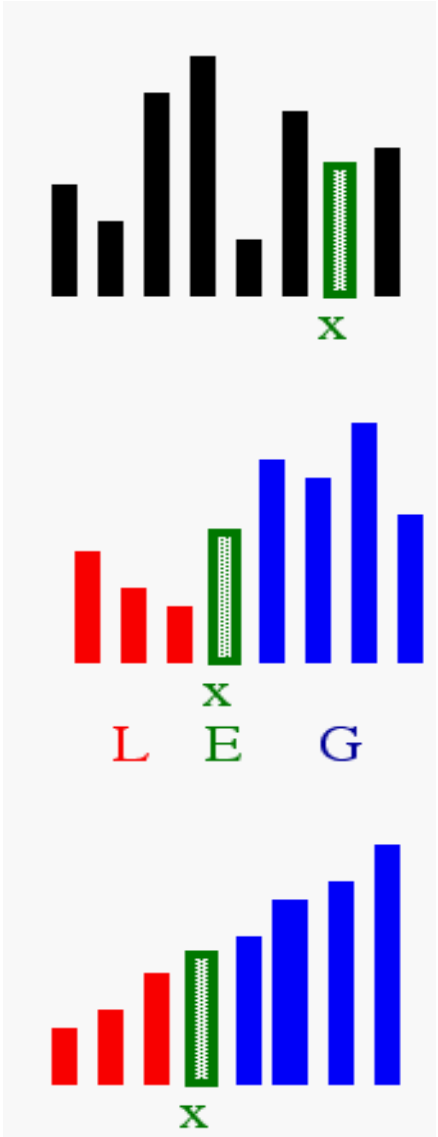
Idea of Quick Sort

Ref Book: Internet

1) **Select**: pick an element

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

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



Ref Book: Thomas Cormen

Quicksort Pseudo-code

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

OUTPUT: an ordered sequence of *n* numbers

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

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

Procedure Partitioning the array

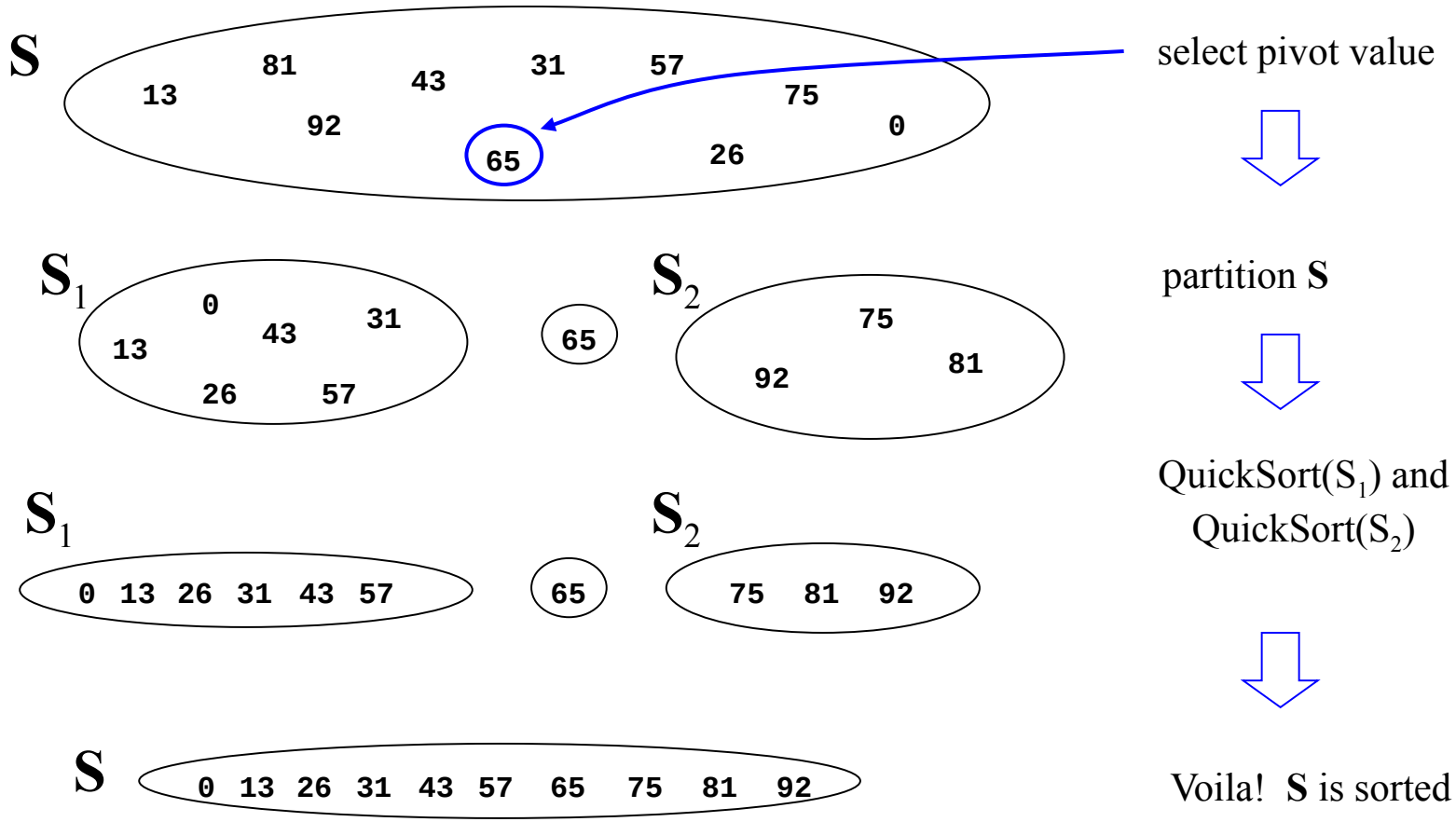
Ref Book: Thomas Cormen

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

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

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

The steps of QuickSort



Quicksort Analysis

Assumptions:

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

Running time

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

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

- i: number of elements in S1

Worst-Case Analysis

worst case Partition?

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

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

Best-case Analysis

best case Partitioning?

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

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

Average-Case Analysis

Intution for Average Case

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

Summary Analysis of Quicksort

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

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

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

Improvements:

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

REF BOOK: THOMAS CORMEN

Large Integer multiplication

References

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