CMP418: Algorithm and Complexity Analysis (3 units)

Lecture 5: Divide and Conquer

MR. M. YUSUF

Outline

- Mergesort
- Master Theorem
- Quicksort
- Hoare Partition
- Binary Tree Traversals Related Properties
  - Binary Tree Traversal

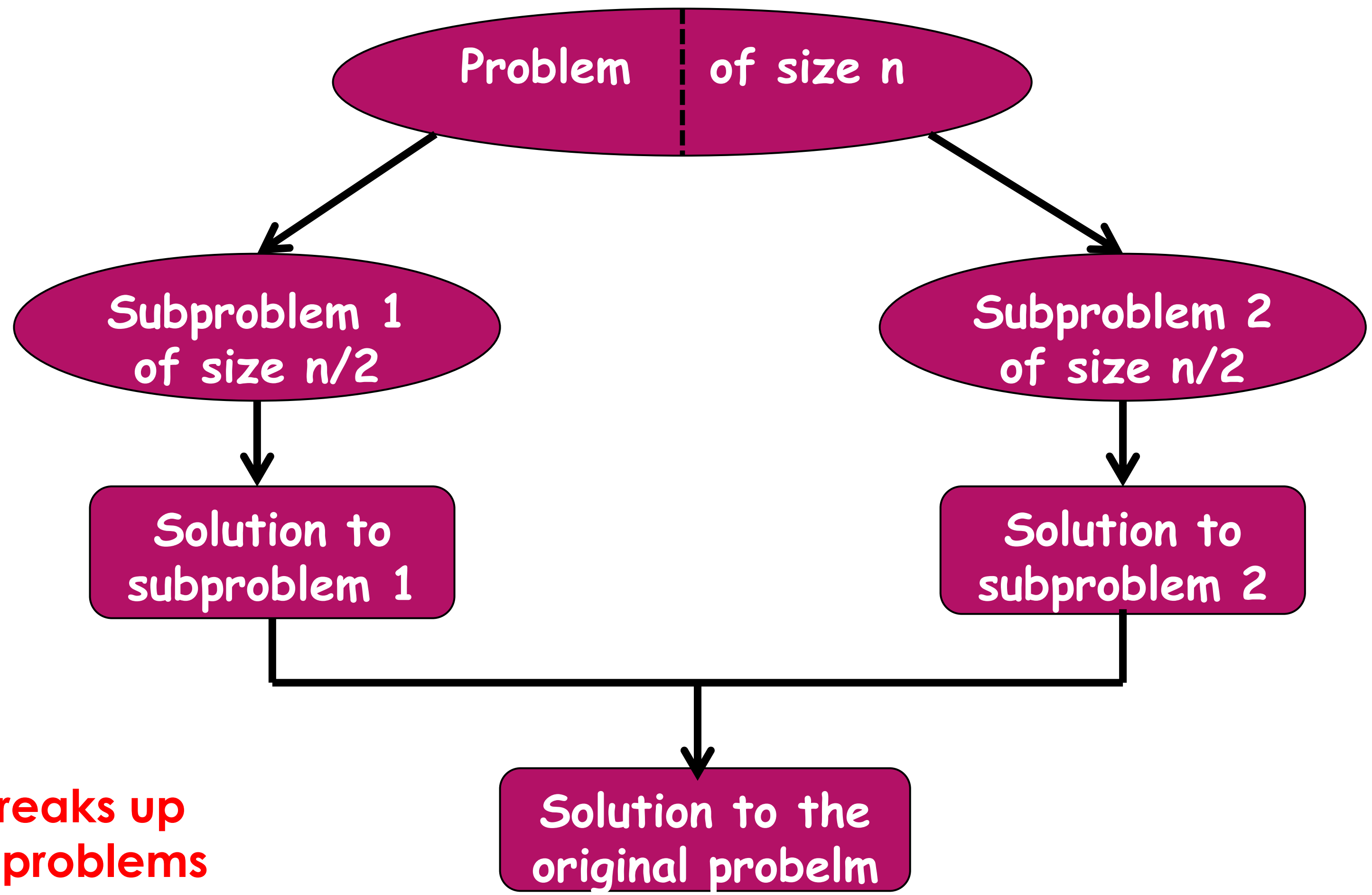
What is Divide and Conquer (DnC)?

- DnC is probably the best-known general algorithm design technique.
- Thus, not every DnC algorithm is necessarily more efficient than even a brute-force solution.
- The time spent on executing the DnC plan turns out to be significantly smaller than solving a problem by a different method.
- DnC yields some of the most important and efficient algorithms in computer science.
- DnC ideally suited for parallel computations, in which each subproblem can be solved simultaneously by its own processor.

How Divide and Conquer Techniques Works

1. A problem is divided into several subproblems of the same type, ideally of about equal size.
2. The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).
3. If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

A Typical Case of Divide-and-conquer technique



Don't assume always breaks up into 2, could be > 2 subproblems

Case Study to Add n Numbers

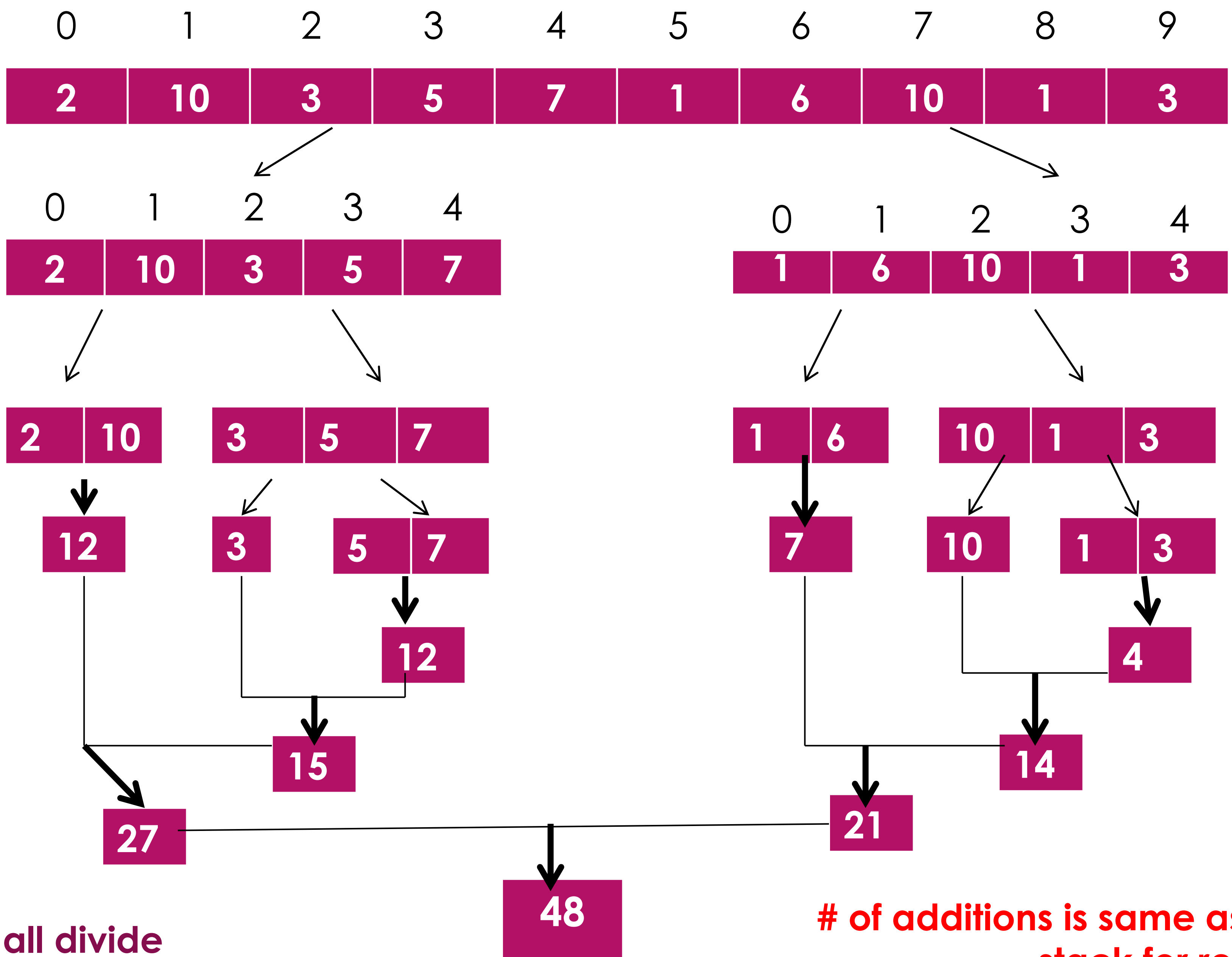
$$a_0 + a_1 + \ldots\ldots + a_{n-1}$$

$$a_0 + \ldots\ldots + a_{\lfloor n/2 \rfloor - 1} \qquad\qquad a_{\lfloor n/2 \rfloor} + \ldots\ldots + a_{n-1}$$

Is it more efficient than brute force ?

Let's see with an example

M. Yusuf



Bad! not all divide and conquer works!!

# of additions is same as in brute force, needs stack for recursion…

Using Divide-and-Conquer Recurrence Relation

- Usually in DnC a problem instance of size n is divided into two instances of size n/2
- More generally, an instance of size ***n*** can be divided into ***b*** instances of size $\frac{n}{b}$, with ***a*** of them needing to be solved
- Assuming that n is a power of b ($n = b^m$), we get
  - $T(n) = aT(\frac{n}{b}) + f(n)$

    This the general DnC recurrence Relation
  - Here, f(n) accounts for the time spent in dividing an instance of size n into subproblems of size $\frac{n}{b}$ **and** combining their solution
  - For adding n numbers, a = b = 2 and f(n) = 1

Outline

- Mergesort
- Master Theorem
- Quicksort
- Hoare Partition
- Binary Tree Traversals Related Properties
  - Binary Tree Traversal

Master Theorem

$T(n) = aT(\frac{n}{b})+f(n)$, $a \geq 1$, $b > 1$

If $f(n) \in \Theta(n^d)$ where $d \geq 0$ then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

For adding n numbers with divide and conquer technique, the number of additions A(n) is:

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

Here, $a = ?$, $b = ?$, $d = ?$ $a = 2$, $b = 2$, $d = 0$

Which of the 3 cases holds ? $a = 2 > b^d = 2^0$, case 3

So, $A(n) \in \Theta(n^{\log_2 2})$
Or, $A(n) \in \Theta(n)$

M. Yusuf

Master Theorem: Example

$T(n) = aT(n/b)+f(n)$, $a \ge 1$, $b > 1$
If $f(n) \in \Theta(n^d)$ where $d \ge 0$, then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \lg n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

$T(n) = 2T(n/2)+6n-1$?

$T(n) = 3\,T(n/2) + n$
$a=3 > b^d=2^1$

$a = 3$, $b = 2$, $f(n) \in \Theta(n^1)$, so $d = 1$
Case 3: $T(n) \in \Theta(\, n^{\log_2 3}\,) = \Theta(\, n^{1.5850}\,)$

$T(n) = 3\,T(n/2) + n^2$
$a=3 < b^d=2^2$

$a = 3$, $b = 2$, $f(n) \in \Theta(n^2)$, so $d = 2$
Case 1: $T(n) \in \Theta(\, n^2\,)$

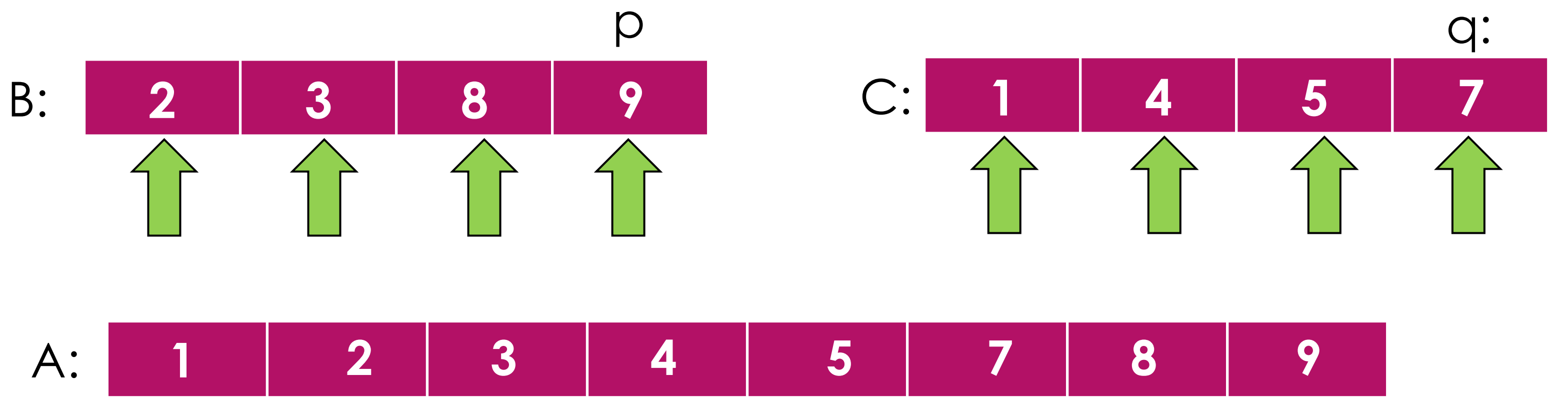
$T(n) = 4\,T(n/2) + n^2$
$a=4 = b^d=2^2$

$a = 4$, $b = 2$, $f(n) \in \Theta(n^2)$, so $d = 2$
Case 2: $T(n) \in \Theta(\, n^2 \lg n\,)$

$T(n) = 0.5\,T(n/2) + 1/n$ ⟵ Master th^m doesn't apply, $a<1$, $d<0$
$T(n) = 2\,T(n/2) + n/\lg n$ ⟵ Master th^m doesn't apply f(n) not polynomial
$T(n) = 64\,T(n/8) - n^2 \lg n$ f(n) is not positive, doesn't apply
$T(n) = 2^n\,T(n/8) + n$ a is not constant, doesn't apply

First Snippet of Mergesort Algorithm

ALGORITHM Mergesort(A[0..n-1])
//sorts array A[0..n-1] by recursive mergesort
//Input: A[0..n-1] to be sorted
//Output: Sorted A[0..n-1]
if n > 1
    copy A[0..$\lfloor n/2 \rfloor$-1] to B[0.. $\lfloor n/2 \rfloor$-1]
    copy A[$\lfloor n/2 \rfloor$..n-1] to C[0..$\lceil n/2 \rceil$-1]
    Mergesort(B[0..$\lfloor n/2 \rfloor$-1])
    Mergesort(C[0..$\lceil n/2 \rceil$-1])
    Merge(B, C, A)



Second Snippet of Mergesort Algorithm

```
ALGORITHM Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0..p-1] and C[0..q-1] both sorted
//Output: Sorted array A[0..p+q-1] of elements of B and C
i <- 0; j <- 0; k <- 0;
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] <- B[i];
        i <- i+1
    else
        A[k] <- C[j];
        j <- j+1
    k <- k+1
if i = p
    copy C[j..q-1] to A[k..p+q-1]
else
    copy B[i..p-1] to A[k..p+q-1]
```

Mergesort Algorithm Comparison…

ALGORITHM Mergesort(A[0..n-1])

//sorts array A[0..n-1] by recursive mergesort

//Input: A[0..n-1] to be sorted

//Output: Sorted A[0..n-1]

if n > a

  copy A[0..$\lfloor n/2 \rfloor$-1] to B[0.. $\lfloor n/2 \rfloor$-1]

  copy A[$\lfloor n/2 \rfloor$..n-1] to C[0..$\lceil n/2 \rceil$-1]

  Mergesort(B[0..$\lfloor n/2 \rfloor$-1])

  Mergesort(C[0..$\lceil n/2 \rceil$-1])

  Merge(B, C, A)

ALGORITHM Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])

//Merges two sorted arrays into one sorted array
//Input: Arrays B[0..p-1] and C[0..q-1] both sorted
//Output: Sorted array A[0..p+q-1] of elements of B
//and C

i <- 0; j <- 0; k <- 0;

while i < p **and** j < q do

  if B[i] ≤ C[j]

    A[k] <- B[i]; i <- i+1

  else

    A[k] <- C[j]; j <- j+1

  k <- k+1

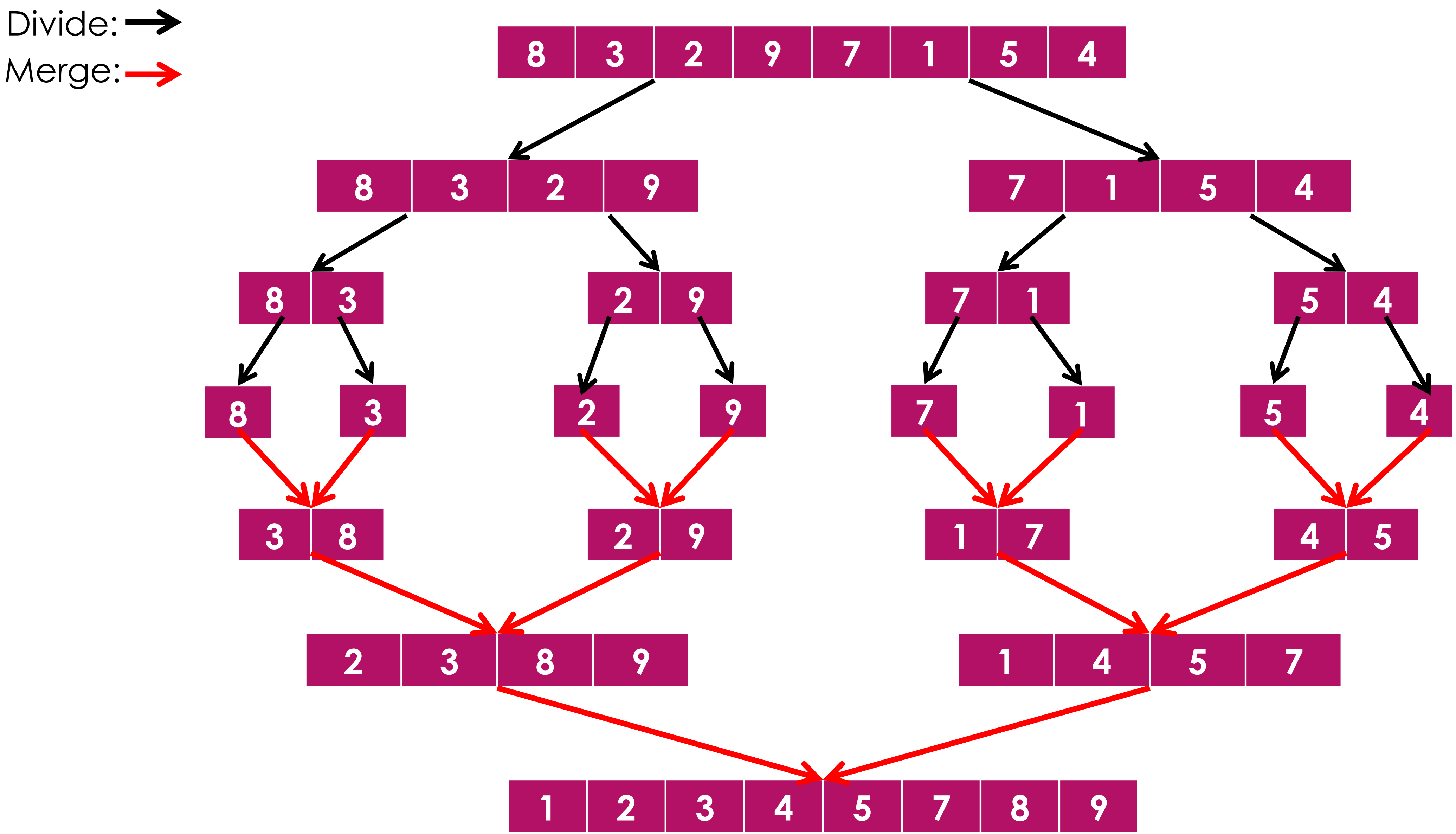
if i = p

  copy C[j..q-1] to A[k..p+q-1]

else

  copy B[i..p-1] to A[k..p+q-1]

Merge Sort Algorithm: Example



Summary of Mergesort Algorithm

- Worst-case of Mergesort is Θ(nlogn)
- Average-case is also Θ(nlogn)
- It is stable but quicksort and heapsort are not
- Possible improvements
  - Implement bottom-up. Merge pairs of elements, merge the sorted pairs, so on… (does not require recursion-stack anymore)
  - Could divide into more than two parts, particularly useful for sorting large files that cannot be loaded into main memory at once: this version is called "multiway mergesort"
- Not in-place, needs linear amount of extra memory
  - Though we could make it in-place, adds a bit more "complexity" to the algorithm

Exercise

Attempt question 1, 2 and 6 of exercise 5.1 on page 174

Outline

- Mergesort
- Master Theorem
- **Quicksort**
- Hoare Partition
- Binary Tree Traversals Related Properties
  - Binary Tree Traversal

Quicksort Algorithm

- A divide and conquer based sorting algorithm, discovered by **C. A. R. Hoare (British)** in **1960** while trying to sort words for a machine translation project from Russian to English
- Instead of "Merge" in Mergesort, Quicksort **uses the idea of partitioning** which we already have seen with "Lomuto Partition"
- In Mergesort all work is in combining the partial solutions.
- In Quicksort all work is in dividing the problem, Combining does not require much work!

How to Quicksort

$$A[0]\ldots A[n-1] \rightarrow \underbrace{A[0]\ldots A[s-1]}_{\text{all are } \le A[s]}\ A[s]\ \underbrace{A[s+1]\ldots A[n-1]}_{\text{all are } \ge A[s]}$$

A[s] is in it's final position

Now continue working with these two parts

Quicksort Algorithm…

- As a partition algorithm we could use “Lomuto Partition”
- But we shall use the more sophisticated “Hoare Partition” instead

ALGORITHM Quicksort(A[l..r])

//Sorts a subarray by quicksort

//Input: Subarray of A[0..n-1] defined by its

//left and right indices l and r

//Output: Subarray A[l..r] sorted in nondecreasing

//order

if l < r

    s <- **Partition**(A[l..r]) // s is a split position

    Quicksort(A[l..s-1])

    Quicksort(A[s+1]..r)

We can replace the partition part with any partitioning algorithm like Lomuto Partition or Hoare Partition

Outline

- Mergesort
- Master Theorem
- Quicksort
- **Hoare Partition**
- Binary Tree Traversals Related Properties
  - Binary Tree Traversal

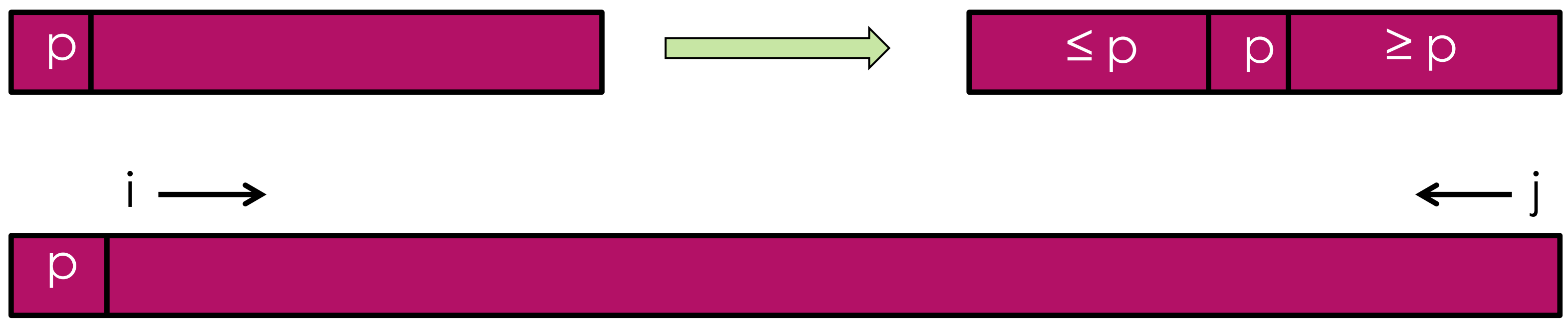
Quicksort Algorithm: Hoare Partitioning

- When using "Hoare Partition"
- We start by selecting a "pivot"
- There are various strategies to select the pivot,
- we shall use the simplest:
- we shall select pivot, p =A[l], the first element of A[l..r]

ALGORITHM HoarePartition(A[l..r])
//Output: the split position
p <- A[l]
i <- l; j <- r+1
repeat
    repeat i <- i+1 **until** A[i] ≥ p
    repeat j <- j-1 **until** A[j] ≤ p
    swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j]) // undo last swap when i ≥ j
swap(A[l], A[j])
return j

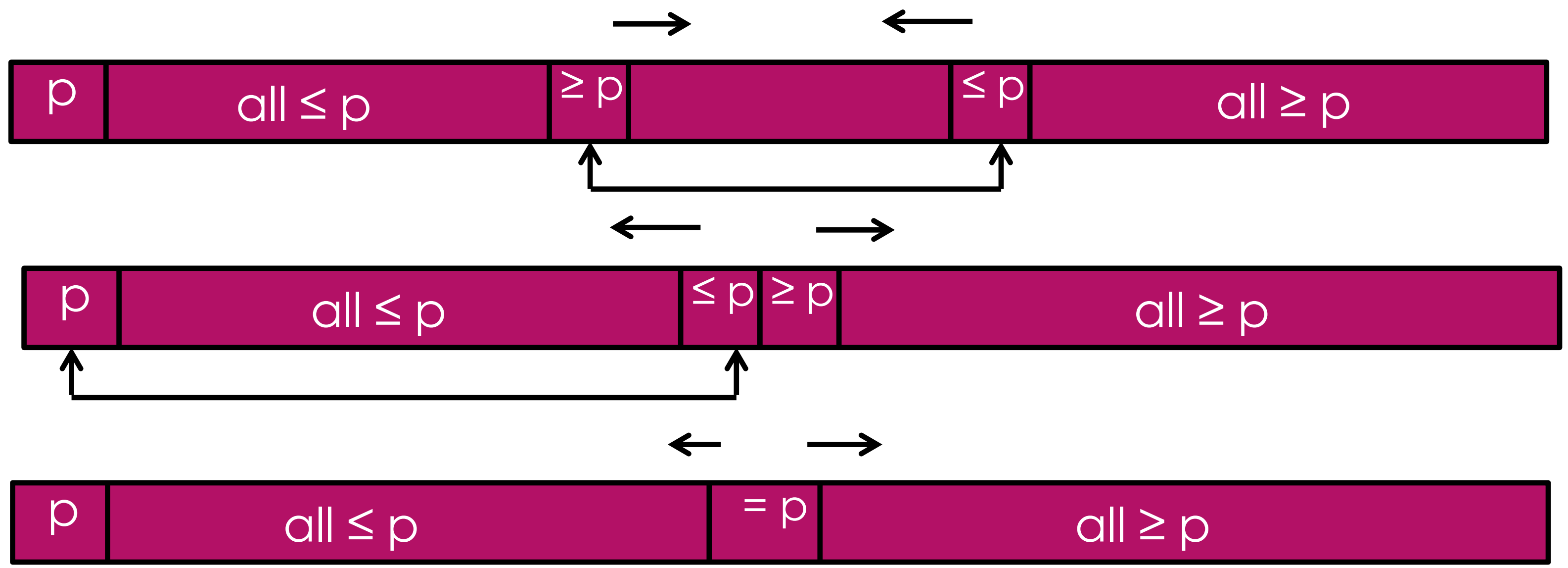
M. Yusuf

How to Sort with Quicksort Algorithm



Step 1: If A[i] < p, we continue incrementing I and move towards RHS, stop when A[i] ≥ p

Step 2: If A[j] > p, we continue decrementing j and move towards the LHS, stop when A[j] ≤ p



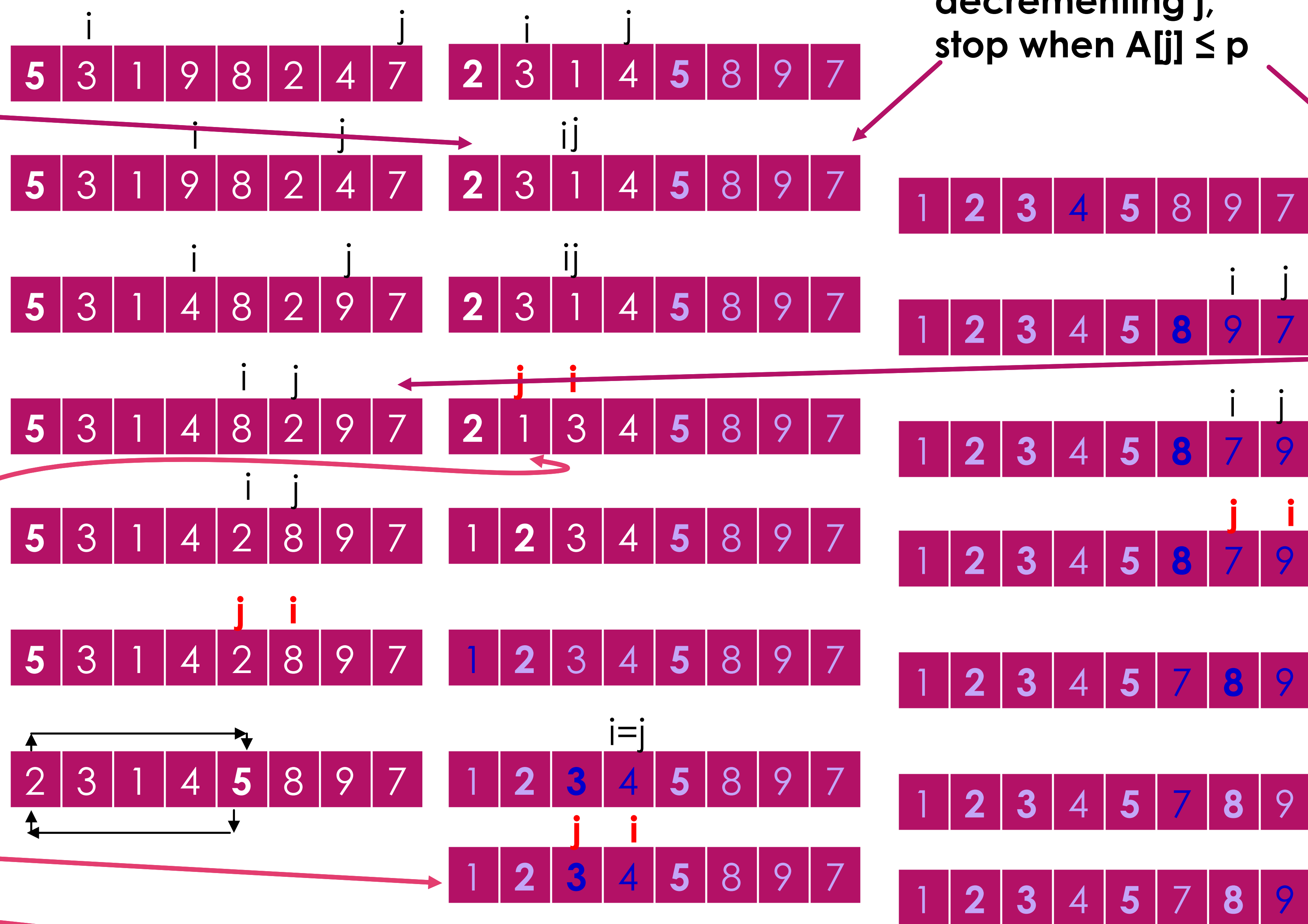
Step 3: If A[i] crosses A[j], New p = point of cross

Step 4: GOTO step 1

Quicksort Example

If A[i] < p, we continue incrementing i, stop when A[i] ≥ p

If A[j] > p, we continue decrementing j, stop when A[j] ≤ p



When ever i and j crosses each other, we swap pivot element with element at j

Quicksort Algorithm…

ALGORITHM Quicksort(A[l..r])

if l < r

    s <- HoarePartition (A[l..r])

    Quicksort(A[l..s-1])

    Quicksort(A[s+1]..r)

ALGORITHM HoarePartition(A[l..r])

//Output: the split position

p <- A[l]

i <- l; j <- r+1

repeat

    repeat i <- i+1 **until** A[i] ≥ p

    repeat j <- j-1 **until** A[j] ≤ p

    swap(A[i], A[j])

until i ≥ j

swap(A[i], A[j]) // undo last swap when i ≥ j

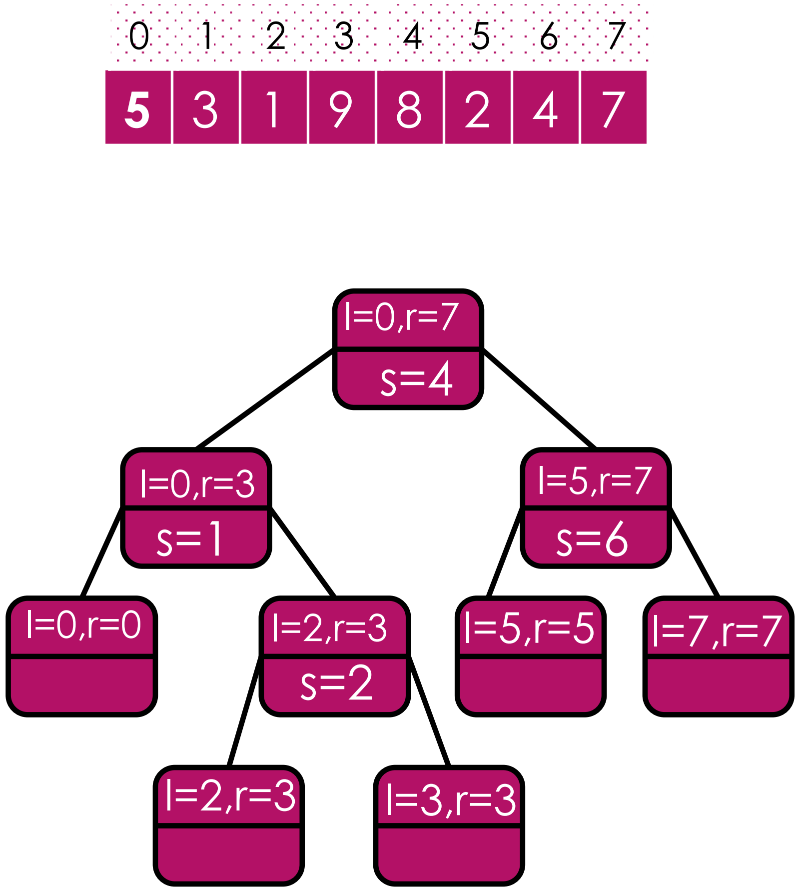
swap(A[l], A[j])

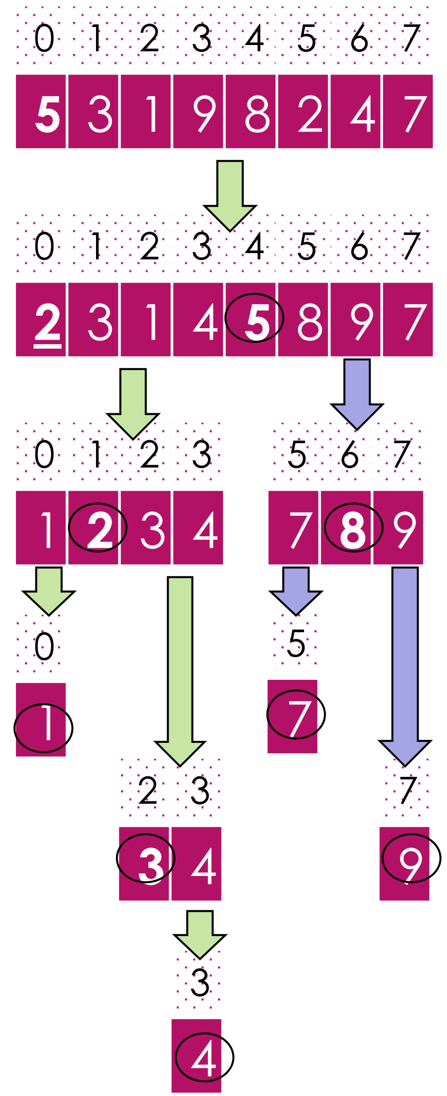
return j

M. Yusuf

Quicksort Operation

(a)





(b)

If all splits happen in the middle, it is the best-case!

(a) Array's transformations with pivots shown in bold.

(b) **Tree of recursive calls** to *Quicksort with input values* ***l*** and **r** of subarray bounds and split position **s** of a partition obtained.

Solving Quicksort with Master Theorem

T(n) = aT(n/b)+f(n), a ≥ 1, b > 1
If f(n) ϵ n^d with d ≥ 0, then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \lg n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

ALGORITHM Quicksort(A[l..r])
if l < r
    s <- Partition(A[l..r])
    Quicksort(A[l..s-1])
    Quicksort(A[s+1]..r)

$C_{worst}(n)$ = (n+1) + (n-1+1) + … + (2+1) = (n+1) + … + 3
= (n+1) + … + 3 + 2 + 1 – (2 + 1) = $\sum_1^{n+1} i$ - 3
= $\frac{(n+1)(n+2)}{2}$ - 3 ϵ Θ(n^2) !

So, Quicksort's fate depends on its average-case!

How can we Improve the Performance of Quicksort Algorithm

- Recall that for Quicksort, $C_{best}(n) \approx n\lg n$
- Quicksort is usually ***faster*** than Mergesort or Heapsort on randomly ordered arrays of nontrivial sizes
- Some possible improvements
    - **Randomized quicksort**: selects a random element as pivot
    - **Median-of-three**: selects median of left-most, middle, and right-most elements as pivot
    - **Switching to insertion sort** on very small subarrays, or not sorting small subarrays at all and finish the algorithm with insertion sort applied to the entire nearly sorted array
    - **Modify partitioning**: three-way partition
    - **These improvements can speed up by 20% to 30%**
- **Weaknesses - Not Stable**

Exercise

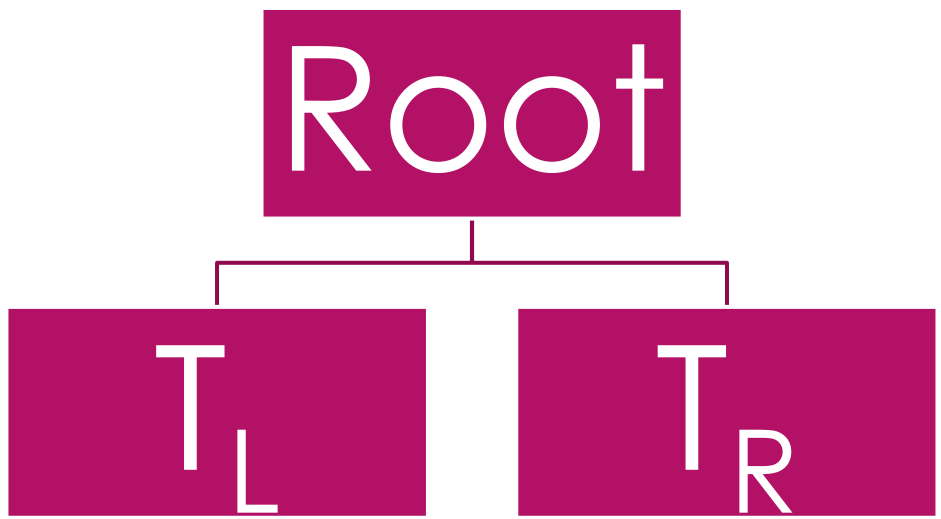
1. Attempt question 1, of exercise 5.2 on page 181
2. Given that $T(n) = 2T(\frac{n}{2}) + 1$, $T(1) = 1$, Derive the complexity class of the algorithm
3. Use master theorem to compute the following
   a) $T(n) = 9T(\frac{n}{2}) + 1$,
   b) $T(n) = 3T(\frac{n}{9}) + n^3$,
   c) $T(n) = 4T(\frac{n}{2}) + n^2$,

Outline

- Mergesort
- Master Theorem
- Quicksort
- Hoare Partition
- **Binary Tree Traversals Related Properties**
  - Binary Tree Traversal

Binary Tree Traversals and Related Properties

- We discuss how the divide-and-conquer technique can be applied to binary trees.
- A ***binary tree T*** *is defined as a finite set of nodes that is either empty* or consists of a ***root*** and ***two disjoint*** binary trees T_L and T_R called, respectively, the left and right subtree of the root.



Binary Search Tree

ALGORITHM *Height(T)*

//Compute recursively the height of a binary tree

//Input: A binary tree T

//Output: The height of tree T

If T = 0

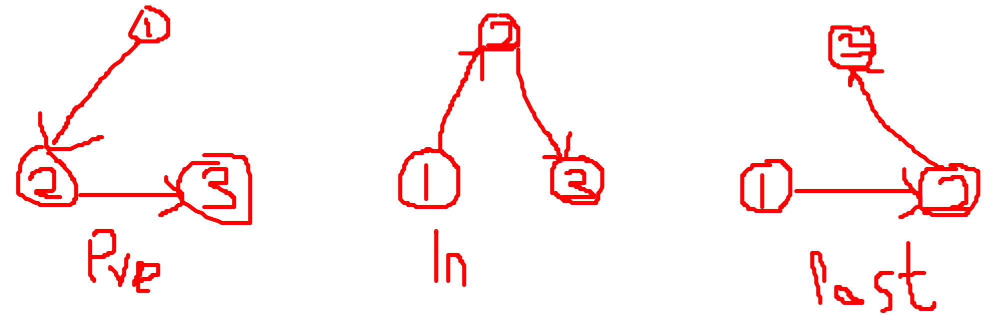
    return -1

else

    return max{height(T_{left}), Height(T_{right})} + 1

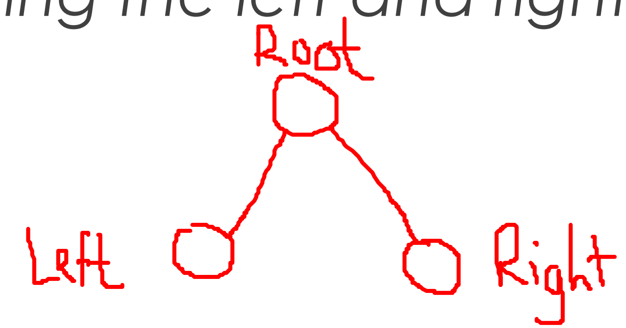
M. Yusuf

Divide and Conquer: Binary Search Tree

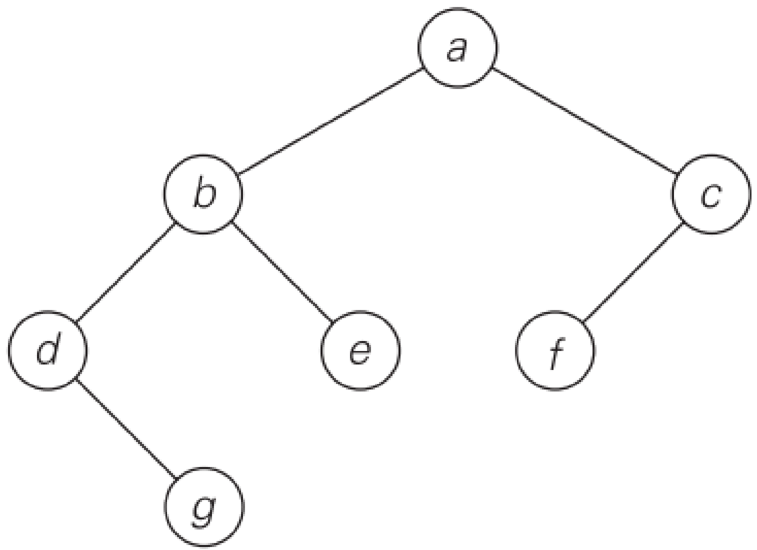


- The most important divide-and-conquer algorithms for binary trees are the three classic traversals namely;

1. the ***preorder*** *traversal, the root is visited before the left and right subtrees* are visited
2. the ***inorder*** *traversal, the root is visited after visiting its left subtree but* before visiting the right subtree.
3. the ***postorder*** *traversal, the root is visited after visiting the left and right* subtrees.



Binary Tree Traversal: Example



preorder: *a, b, d, g, e, c, f*
inorder: *d, g, b, e, a, f, c*
postorder: *g, d, e, b, f, c, a*

Exercise

1. Attempt question 1, of exercise 5.2 on page 181
2. Given that T(n) = T($\frac{n}{2}$) + 1, T(1) = 0, Derive the complexity class of the algorithm

Thank You