Introduction to Algorithm Design

Lecture Notes 4

ROAD MAP

- **Divide And Conquer**
  - **Binary Search**
  - **Maximum Subsequence Problem**
  - Merge Sort
  - Quick Sort
  - Multiplication of Large Integers
  - Strassen's Matrix Multiplication
  - Closest Pair of Points
  - Convex Hull

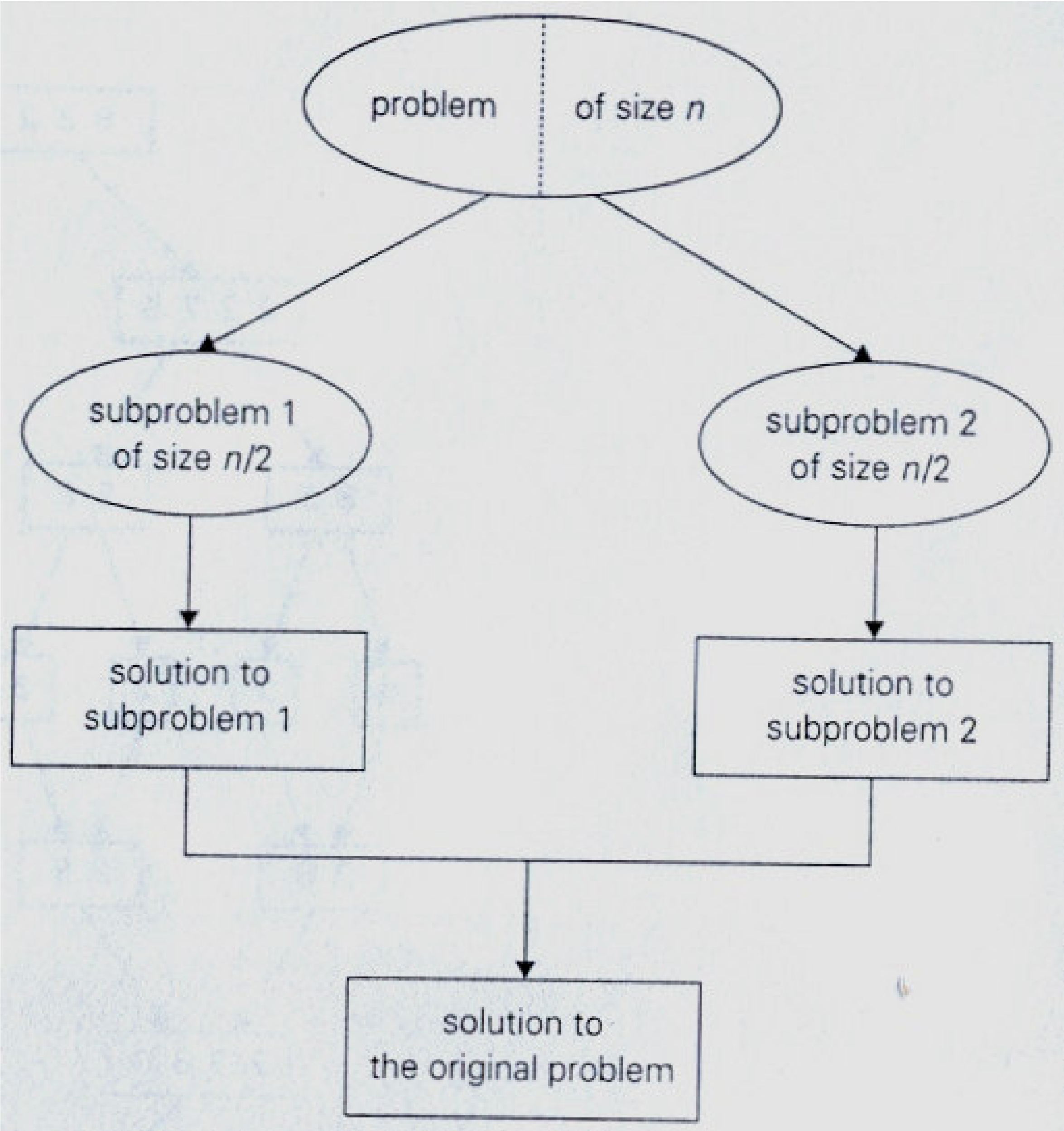
Divide And Conquer

A well known general algorithm design technique

Approach*:*

- A problem's instance is divided into several smaller instances of the same problem
  - ideally of about the same size
- The smaller instances are solved
  - typically recursively
- The solutions obtained for the smaller instances are combined to get a solution to the original problem

Divide And Conquer



Divide And Conquer

- **Algorithm :**

```
D&C (P)
if small (P) then return S(P)
else
{
    divide P into P1, P2, …, Pk    k≥1
    apply D&C to Pi
    return combine ( D&C (P1), …, D&C (Pk) )
}
```

Divide And Conquer

- **Analysis :**

$$T(P) = T(P_1) + T(P_2) + \ldots + T(P_k) + \underbrace{f(n)}_{to \quad divide \& \quad combine}$$

$$T(n) = T(n_1) + T(n_2) + \ldots + T(n_k) + f(n)$$

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

A simple Example

Problem:

- Compute the sum of n numbers

Approach:

- Divide the problem into two subproblems

- What about the analysis?
  - Is it more efficient than brute force approach?

Binary Search

Binary search is a remarkably efficient algorithm for searching in a ***sorted*** array

An array A[0 .. n-1] and a search key *K* is given

Approach :

1. Comparing a search key *K* with the array's middle element A[m]

2. If they match, the algorithm stops

3. Otherwise same operation is repeated recursively for the first half of the array if K<A[m] and for the second half if K>A[m]

Binary Search

ALGORITHM *BinarySearch*$(A[0..n-1], K)$

//Implements nonrecursive binary search
//Input: An array $A[0..n-1]$ sorted in ascending order and
//   a search key K
//Output: An index of the array's element that is equal to K
//   or -1 if there is no such element
$l \leftarrow 0$; $r \leftarrow n - 1$
while $l \leq r$ **do**
    $m \leftarrow \lfloor (l + r)/2 \rfloor$
    if $K = A[m]$ **return** m
    else if $K < A[m]$ $r \leftarrow m - 1$
    else $l \leftarrow m + 1$
return -1

Binary Search

Example :

Apply binary search to searching for K =70 in the array

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	l						m						r
iteration 2								l		m			r
iteration 3								l,m	r				

Binary Search

- **Analysis :**

What is the basic operation ?

Binary Search

- Basic operation is the ***comparison*** of the search key and an element of the array
- How many comparisons are made?
  - Depends on n
  - Also depends on the problem instance
- Requires best, worse and average case analysis

Binary Search

- Best case: When the search key is in the middle of the array.

$$C_b(n) = \Theta(1)$$

Binary Search

- Worst case: When the search key is not found in the array

$$C_w(n) = C_w(\lfloor n/2 \rfloor) + 1 \text{ for } n > 1, \; C_w(1) = 1.$$

By assuming that n = 2^k and solving the recurrence by backward substitution, we get

$$C_w(2^k) = k + 1 = \log_2 n + 1.$$

In general

$$C_w(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n+1) \rceil.$$

Binary Search

It is possible to verify the general formula by substituting it into the recurrence equation:

- left-hand side of recurrence equation for $n = 2i$ is

$$C_w(n) = \lfloor \log_2 n \rfloor + 1 = \lfloor \log_2 2i \rfloor + 1 = \lfloor \log_2 2 + \log_2 i \rfloor + 1$$
$$= (1 + \lfloor \log_2 i \rfloor) + 1 = \lfloor \log_2 i \rfloor + 2.$$

- right-hand side of recurrence equation for $n = 2i$ is

$$C_w(\lfloor n/2 \rfloor) + 1 = C_w(\lfloor 2i/2 \rfloor) + 1 = C_w(i) + 1$$
$$= (\lfloor \log_2 i \rfloor + 1) + 1 = \lfloor \log_2 i \rfloor + 2$$

Since both expressions are same, verification is achieved.

Binary Search

- What can you say about the average case ?

Binary Search

- An analysis shows that the average number of key comparisons made by binary search is only slightly smaller than that in the worst case

$$C_{avg}(n) \approx \log n$$

Binary Search

Discussion :

- advantages
  - be able to process large amounts of data
  - has a low cost per run

- disadvantages:
  - needs to be able to look at the whole array.
  - if there is too many data items then it requires a lot of memory.

Maximum Subsequence Sum Problem

- **Problem Definition :**

  Given possibly negative integers a[1], …, a[n] find the maximum value of the

$$\sum_{k=i}^{j} a_k \qquad 1 \le i \le j \le n$$

Example :

$$-4 \quad \overbrace{\underbrace{10 \quad 12}_{22} \quad -5 \quad -7 \quad 8 \quad 3 \quad 1}^{22}$$

Maximum Subsequence Sum Problem

- **Divide and Conquer Algorithm**

```
1. divide the sequence into two halves
2. find MAXSUM (L)
3. find MAXSUM (R)
4. MAXSUM = max ( MAXSUM (L), MAXSUM (R) )
5. find MAXSUM′ (L)
6. find MAXSUM′ (R)
7. MAXSUM = max ( MAXSUM (L), MAXSUM (R)
                  MAXSUM′ (L) + MAXSUM′ (R) )
```

Maximum Subsequence Sum Problem

- **Analysis :**

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

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

ROAD MAP

- **Divide And Conquer**
  - Binary Search
  - Maximum Subsequence Problem
  - **Merge Sort**
  - **Quick Sort**
  - Multiplication of Large Integers
  - Strassen’s Matrix Multiplication
  - Closest Pair of Points
  - Convex Hull

Mergesort

- Mergesort is a perfect example of a successfull application of divide & conquer technique
- Solves the sorting problem

- Given array **`A[0..n-1]`**

Approach :

```
1. divide array into two halves
   A[0..n/2-1] and A[n/2..n-1]
2. sort each halve recursively
3. merge two smaller sorted arrays into a
single sorted one
```

Mergesort

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

```
// sorts array A[0..n-1] by recursive mergesort
// input  : An array A[0..n-1] of orderable elements
// output : Array A[0..n-1] sorted in nondecreasing
order

If n>1
    copy A[0..(n/2)-1] to B[0..(n/2)-1]
    copy A[n/2..n-1]   to C[0..(n/2)-1]
    Mergesort (B[0..(n/2)-1])
    Mergesort (C[0..(n/2)-1])
    Merge (B, C, A)
```

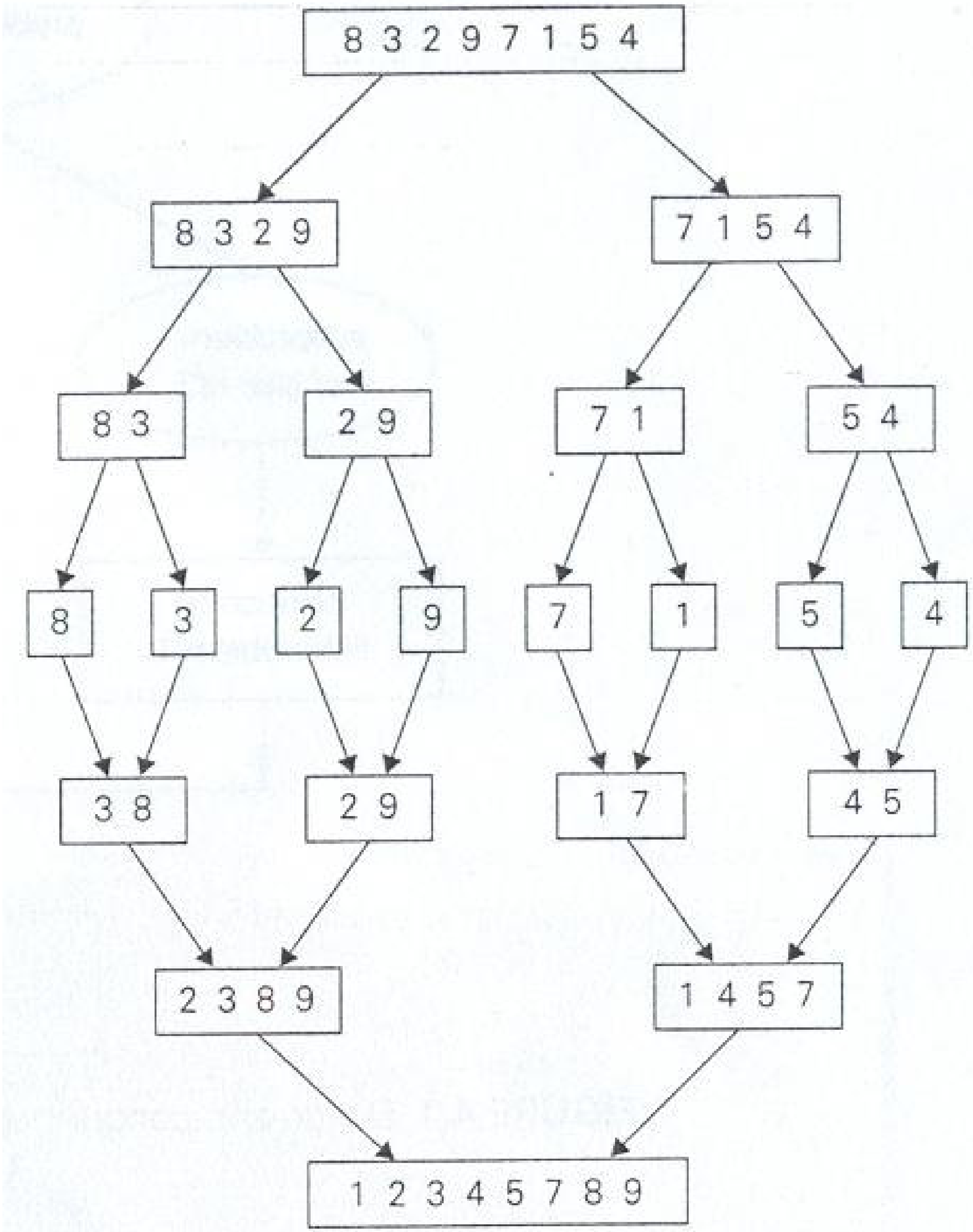
Mergesort

- **ALGORTHM** `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 the 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 Example



Mergesort

Analysis :

Count the number of comparisons

$$C(n) = 2C(n/2) + C_{merge}(n) \qquad \text{for} \qquad n > 1,$$

$$C(1) = 0$$

What about the merge operation?

- Worst case: when the smaller comes from alternating array

$$C_{merge}(n) = n - 1$$

Mergesort

Analysis :

$$C_w(n) = 2C_w(n/2) + n - 1 \qquad \text{for} \qquad n > 1,$$

$$C_w(1) = 0$$

By backward substitution

$$C_w(n) = n\log_2 n - n + 1 = O(n\log n)$$

Or we can use Master Theorem if asymptotic solution is sufficient

Mergesort

- **Discussion :**
  - Perfect example of a successfull application of divide & conquer technique
  - Optimal with respect to number of comparisons
  - Disadvantages
    - Extra space used in Merge
      - How big it is?
      - How to reduce?
    - Recursive calls – stack space
      - use insertion sort for small # of elements
      - iterative

Quicksort

- Quicksort is an important sorting algorithm based on Divide & Conquer strategy
- It sorts a given array **`A[0..n-1]`**

Quicksort

Given an array **A[0..n-1]**

Approach :

1. **Divide input's elements according to their values**
   - **Rearrange elements of a given array A[0..n-1] to achieve a *partition***

$$\underbrace{A[0]\dots A[s-1]}_{\text{all are } \leq A[s]} \qquad A[s] \qquad \underbrace{A[s+1]\dots A[n-1]}_{\text{all are } \geq A[s]}$$

   - **After a partition has been achieved, A[s] will be in its final position in sorted array**

3. **Continue sorting two subarrays of elements preceding**

Divide And Conquer

- **Algorithm :**

```
D&C (P)
if small (P) then return S(P)
else
{
    divide P into P1, P2, …, Pk    k≥1
    apply D&C to Pi
    return combine ( D&C (P1), …, D&C (Pk) )
}
```

Quicksort

ALGORITHM `Quicksort (A[l..r])`

```
// Sorts a subarray by quicksort
// Input  : A subarray A[l..r] of A[0..n-1],
defined by its left and right indces l and r
// Output : The 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])
```

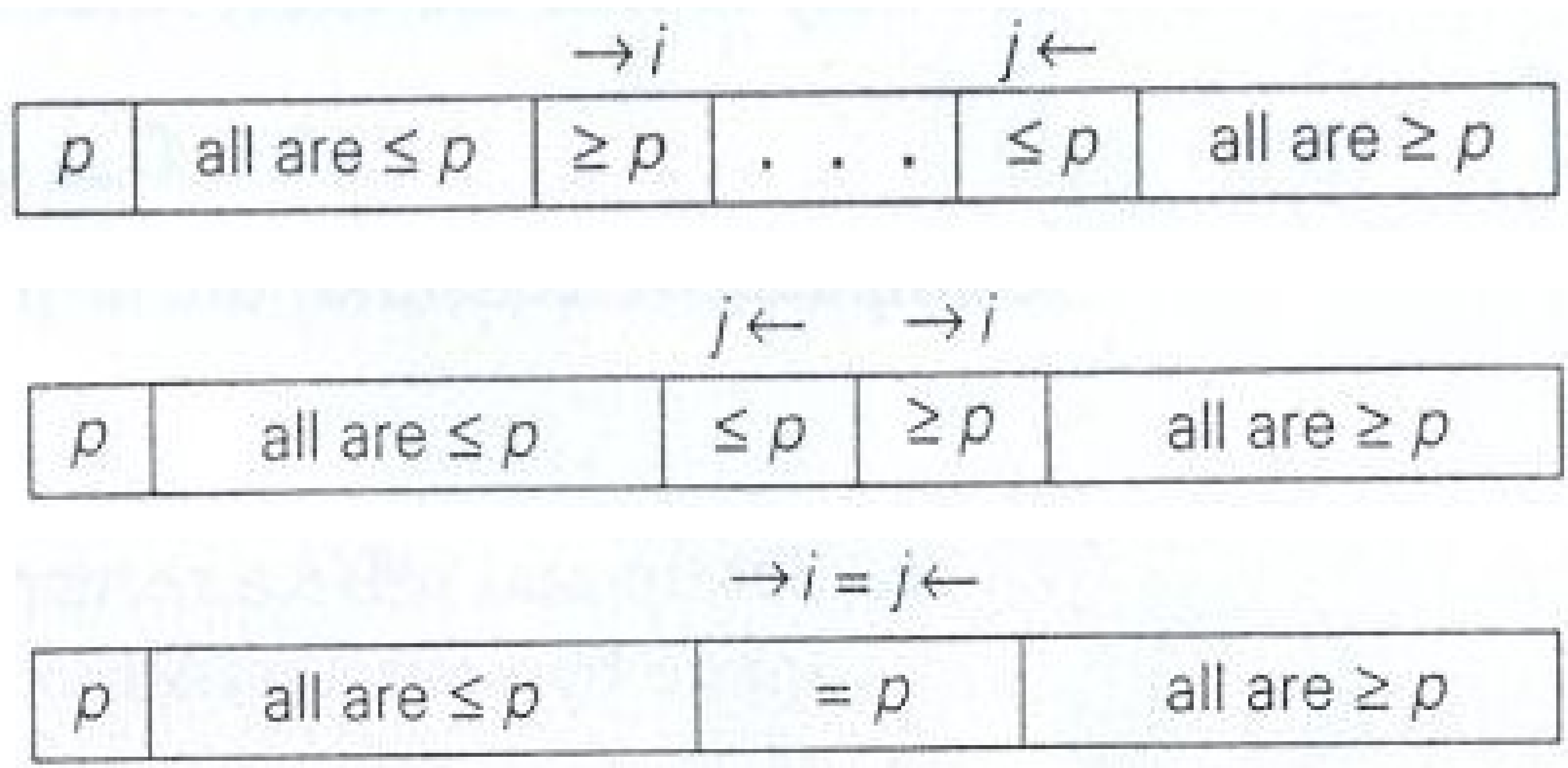
Quicksort

- How to achieve a partition of A[0..n-1] ?
  - Select an element with respect to whose value we are going to divide subarray
    - this element is called ***pivot***

- There are several strategies to select a pivot.
  - Simplest strategy: Pivot is the first element; *p=A[1]*

Quicksort

- Partitioning :



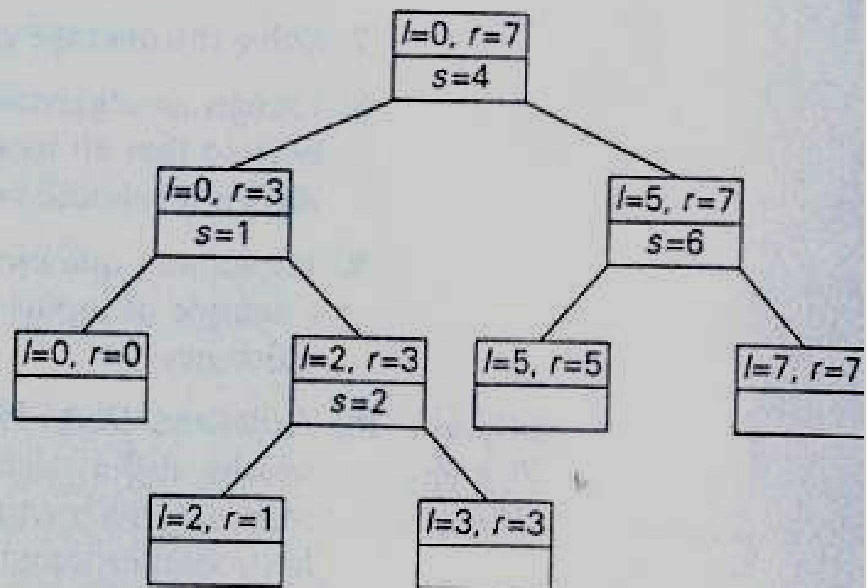
Quicksort

ALGORITHM Partition (A[l..r])

```
// Partitions a subarray by using its first element
as a pivot
// Input  : A subarray A[l..r] of A[0..n-1], defined
by its left and right indices l and r (l < r)
// Output : A partition of A[l..r], with the split
position returned as this function's value

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
```

0	1	2	3	4	5	6	7
5	$\overset{i}{3}$	1	9	8	2	4	$\overset{j}{7}$
5	3	1	$\overset{i}{9}$	8	2	$\overset{j}{4}$	7
5	3	1	$\overset{i}{4}$	8	2	$\overset{j}{9}$	7
5	3	1	4	$\overset{i}{8}$	$\overset{j}{2}$	9	7
5	3	1	4	$\overset{i}{2}$	$\overset{j}{8}$	9	7
5	3	1	4	$\overset{j}{2}$	$\overset{i}{8}$	9	7
2	3	1	4	**5**	8	9	7
2	$\overset{i}{3}$	1	$\overset{j}{4}$				
2	$\overset{i}{3}$	$\overset{j}{1}$	4				
2	$\overset{i}{1}$	$\overset{j}{3}$	4				
2	$\overset{j}{1}$	$\overset{i}{3}$	4				
1	**2**	3	4				
1							
		3	$\overset{ij}{4}$				
		$\overset{j}{\mathbf{3}}$	$\overset{i}{4}$				
			4				
					8	$\overset{i}{9}$	$\overset{j}{7}$
					8	$\overset{i}{7}$	$\overset{j}{9}$
					8	$\overset{j}{7}$	$\overset{i}{9}$
					7	**8**	9
					7		
							9



Quick Sort

- **Analysis :**

```
n:# of elements
T(partition) = O(n) → n+1
```

- **Best case**

  If all the splits happen in the middle of the corresponding subarrays

$$T(n) = 2T(n/2) + n \qquad for \qquad n > 1$$

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

Quick Sort

- **Analysis :**

  - Worst-case
    - All splits will be skewed to the extreme
      - One of the two subarrays will be empty while the size of the other will be just one less than the size of a subarray being partitioned
    - If A[0 .. n-1] is a strictly increasing array and we use A[0] as the pivot
      - Left to right scan will stop on A[1]
      - Right to left scan will go all the way to reach A[0]

$$T(n) = (n+1) + n + \ldots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \theta(n^2)$$

Quick Sort

Analysis :

- Average Case: Each element has an equal probability of being the pivot

$$P = 1/n$$

Quick Sort

Analysis :

- Average Case: Each element has an equal probability of being the pivot

P = 1/n

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

Quick Sort

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

$$nT(n) = \sum_{k=1}^{n} T(k-1) + T(n-k) + n(n+1)$$

$$nT(n) = 2\big(T(0) + T(1) + \ldots + T(n-1)\big) + n(n+1)$$

$$- \quad (n-1)T(n-1) = 2\big(T(0) + \ldots + T(n-2)\big) + n(n-1)$$

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1}$$

$$\frac{T(n)}{n+1} = \frac{T(n-2)}{n-1} + \frac{2}{n+1} + \frac{2}{n}$$

$$\frac{T(n)}{n+1} = \frac{T(n-3)}{n-2} + \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1}$$

$$\vdots$$

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

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

$$\frac{T(n)}{n+1} \le 2\log(n+1)$$

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

Quick Sort

<u>**Discussion :**</u>

- Quicksort is a very efficient algorithm on average
- Its performance depends on the *pivot* selection
  - The farther we get from the median for the pivot value the more lopsided the partitions become and the greater the depth of the recursion needs to be

Binary Tree and Its Properties

- Recall recursive definition of Binary Trees
- Many problems about BTs can be solved by applying divide-and-conquer technique
  - The height of the tree
  - The number of nodes in the tree
  - Traversals
    - Preorder
    - Postorder
    - Inorder