Announcement

The deadline of Homework #3 has been extended to 11:59pm, Feb 25.

Announcement

Homework #4 has been posted in Blackboard and course website.

Due: Thursday, March 3 before class starts.

Announcement

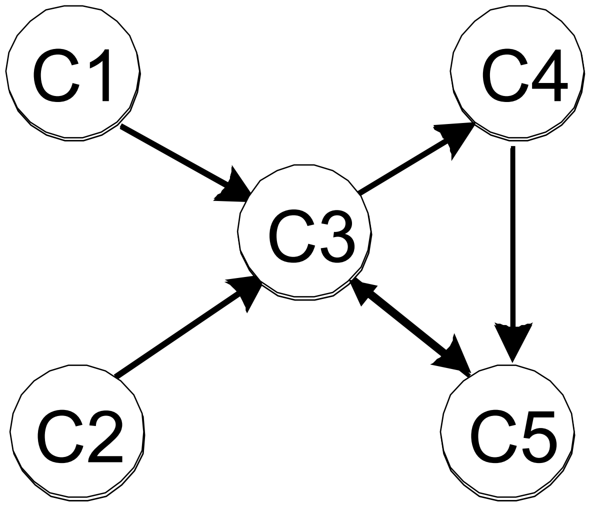
We will have an in-class quiz (Quiz #2) on Tuesday, March 1. It is open-book and open-notes.

The question will ask you to perform quicksort on a given list of numbers.

Topological Sorting

Problem: find an order of vertices such that for every edge in the graph, the starting vertex is listed before the ending vertex

Example:



Five courses has the prerequisite relation shown in the left. Find the right order to take all of them sequentially

Note: problem is solvable iff graph is DAG

Topological Sorting Algorithms

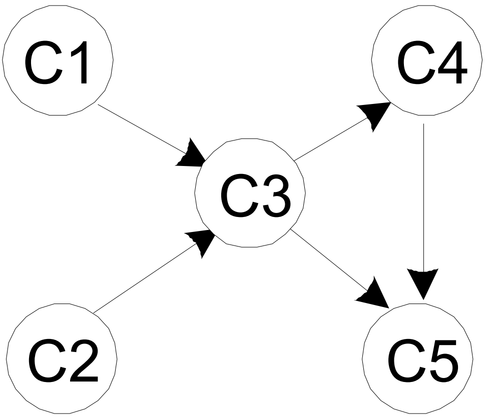
DFS-based algorithm:

- DFS traversal: note the order with which the vertices are popped off stack (dead end)
- Reverse order solves topological sorting
- Back edges encountered?→ NOT a DAG!

Source removal algorithm

- Repeatedly identify and remove a *source* vertex, i.e., a vertex that has no incoming edges

An Example: DFS-based Topological Sorting



(a)

$C5_1$

$C4_2$

$C3_3$

$C1_4$ $C2_5$

(b)

The popping-off order:

C5, C4, C3, C1, C2

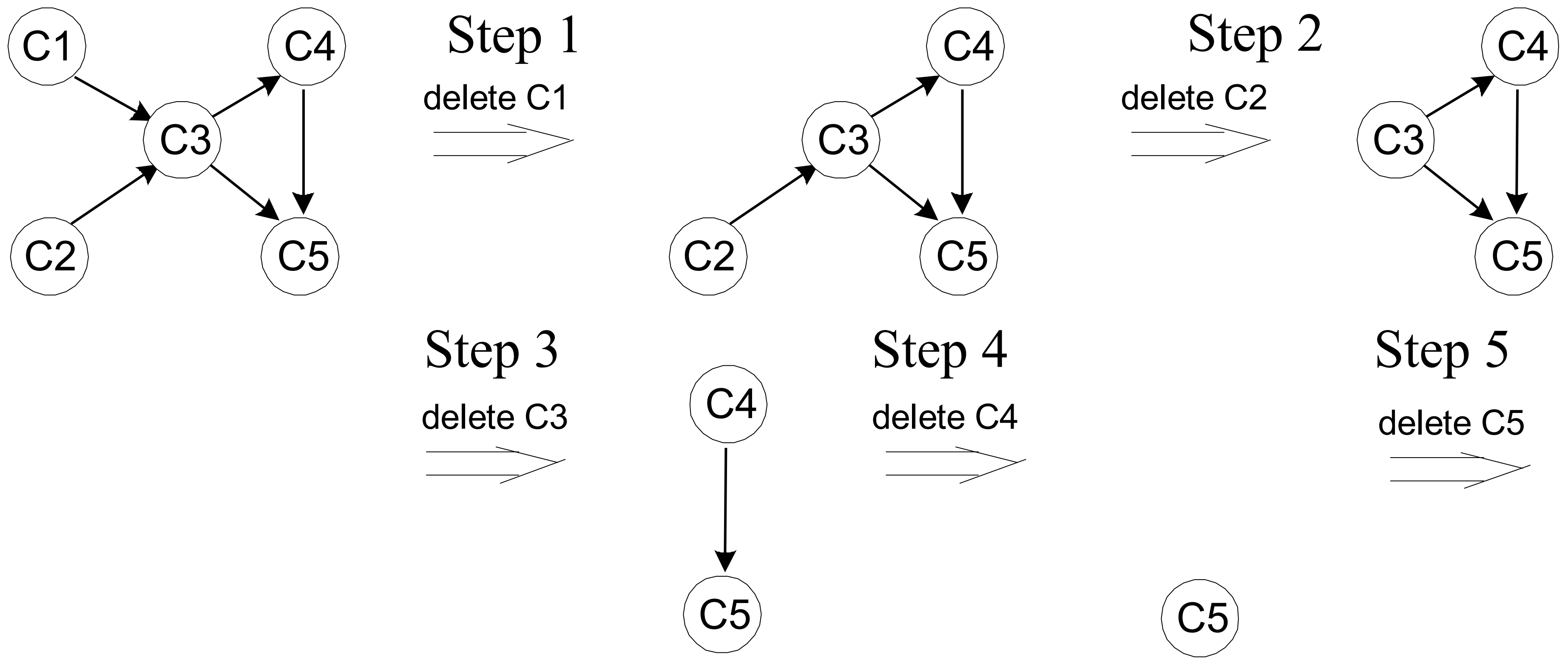
The topologically sorted list:

C2 C1 C3 C4 C5

(c)

Θ(V+E) using adjacency linked lists

An Example: Source removal



C1 C2 C3 C4 C5

Θ(V+E) using adjacency linked lists

How to implement it?

Comparison

DFS based algorithm and the source removal algorithm may produce different valid topological order lists.

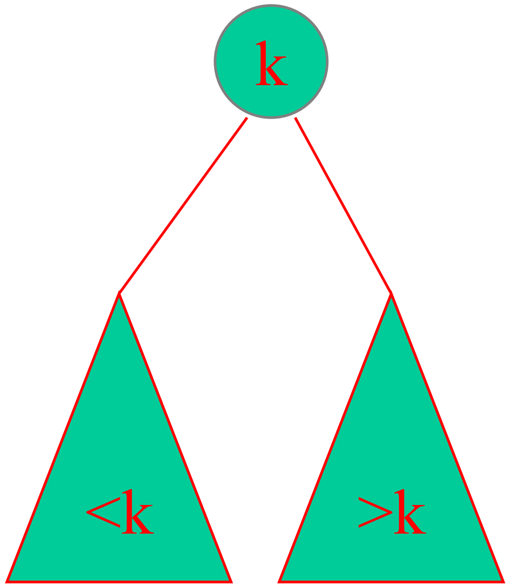
$$\begin{tikzcd}
C2 \arrow[rr, bend right] & C1 \arrow[r] & C3 \arrow[r] \arrow[rr, bend right] & C4 \arrow[r] & C5
\end{tikzcd}$$

$$\begin{tikzcd}
C1 \arrow[rr, bend right] & C2 \arrow[r] & C3 \arrow[r] \arrow[rr, bend right] & C4 \arrow[r] & C5
\end{tikzcd}$$

Variable-Size-Decrease: Binary Search Trees

- Every element in the left subtree is smaller than the root
- Every element in the right subtree is larger than the root
- Search a key in a binary search tree is reduced to search in a subtree in each iteration.
- The height of the subtree changes each time

$\Rightarrow$ variable-size-decrease



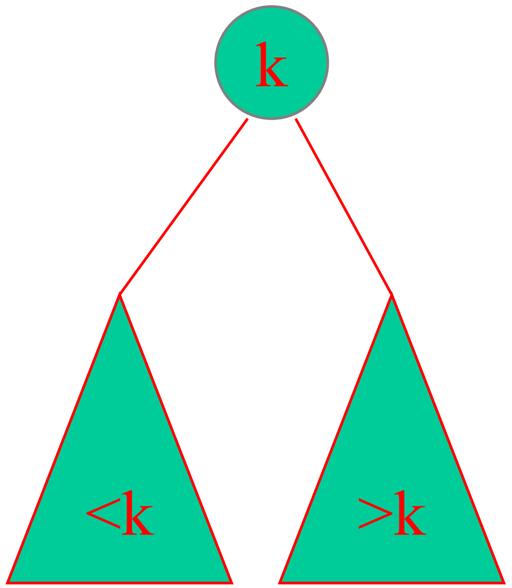
Search a Key in a Binary Search Tree

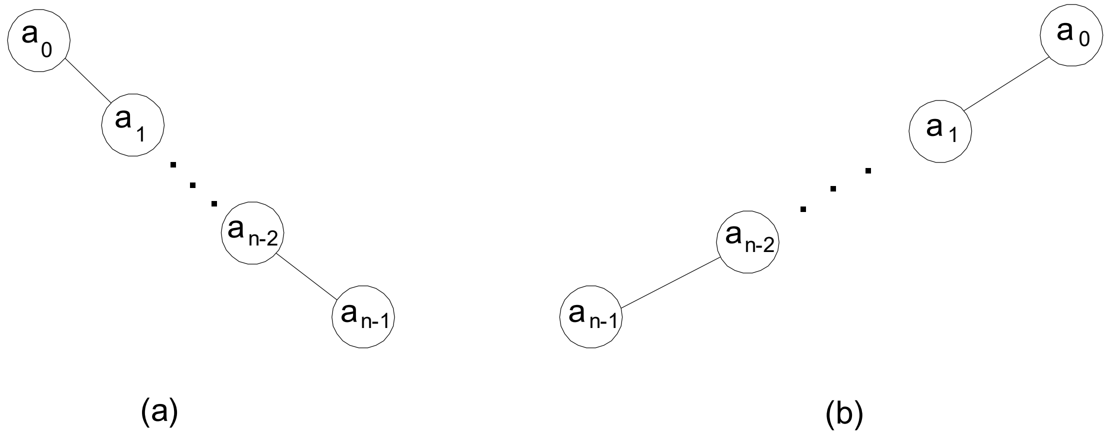
Basic operation: key comparison

# of comparisons in the worst case: h+1

$$\log|V| \le h \le |V| - 1$$

Worst case: the tree degrades to a singly linked list Θ(|V|)
Average case: Θ(log|V|)





Searching and insertion in binary search trees

Searching – straightforward

Insertion – search for key, insert at leaf where search terminated

Example 1: 5, 10, 3, 1, 7, 12, 9

Example 2: 4, 5, 7, 2, 1, 3, 6

Reading Assignments

Chapter 5.3, 5.4 and 5.5

Now, Chapter 5 -- Divide and Conquer

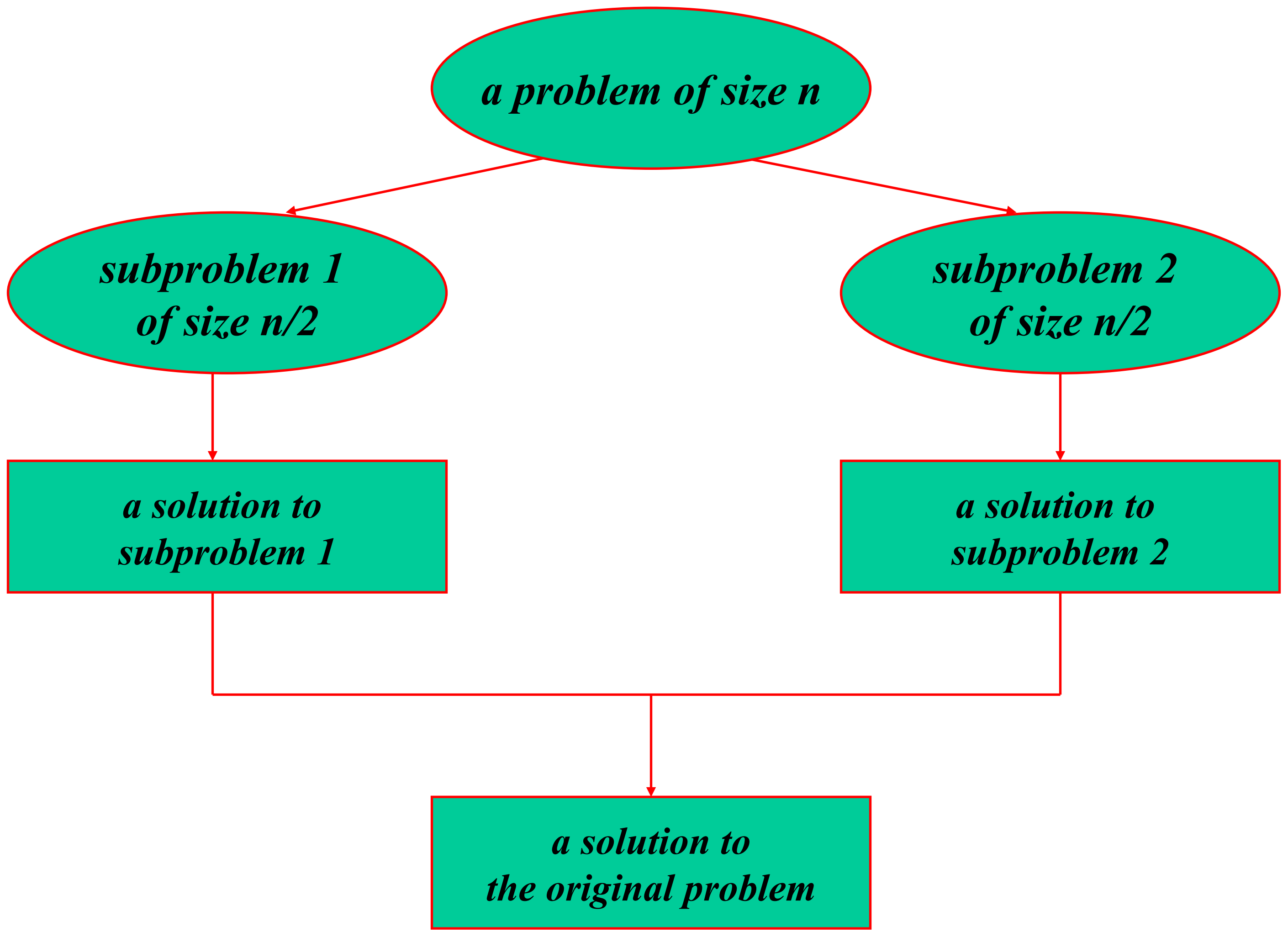
The most well-known algorithm design strategy:

Divide instance of problem into two or more smaller instances of the same problem, ideally of about the same size

Solve smaller instances recursively

Obtain solution to original (larger) instance by combining these solutions obtained for the smaller instances

Divide-and-conquer technique


a problem of size n
subproblem 1 of size n/2
subproblem 2 of size n/2
a solution to subproblem 1
a solution to subproblem 2
a solution to the original problem

An Example

Compute the sum of *n* numbers a_0, a_1, …, a_{n-1}.

Question: How to design a divide-and-conquer algorithm to solve this problem and what is its complexity?

Use divide-and-conquer strategy:

$$\boxed{a_0 + \ldots + a_{n-1} = (a_0 + \ldots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + a_{n-1})}$$

What is the recurrence and the complexity of this recursive algorithm?

Does it improve the efficiency of the brute-force algorithm?

General Divide and Conquer Recurrence

$$C(n) = 2C\left(\frac{n}{2}\right) + 1, for\ n > 1 \qquad C(1) = 0$$

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^k)$

$a < b^k$ $\quad T(n) \in \Theta(n^k)$

$a = b^k$ $\quad T(n) \in \Theta(n^k \log n)$

$\boxed{a > b^k \quad T(n) \in \Theta(n^{\log_b a})}$

a=2, b=2, k=0

$a>b^k$, C(n) belongs to Θ (n)

Mergesort

Algorithm:

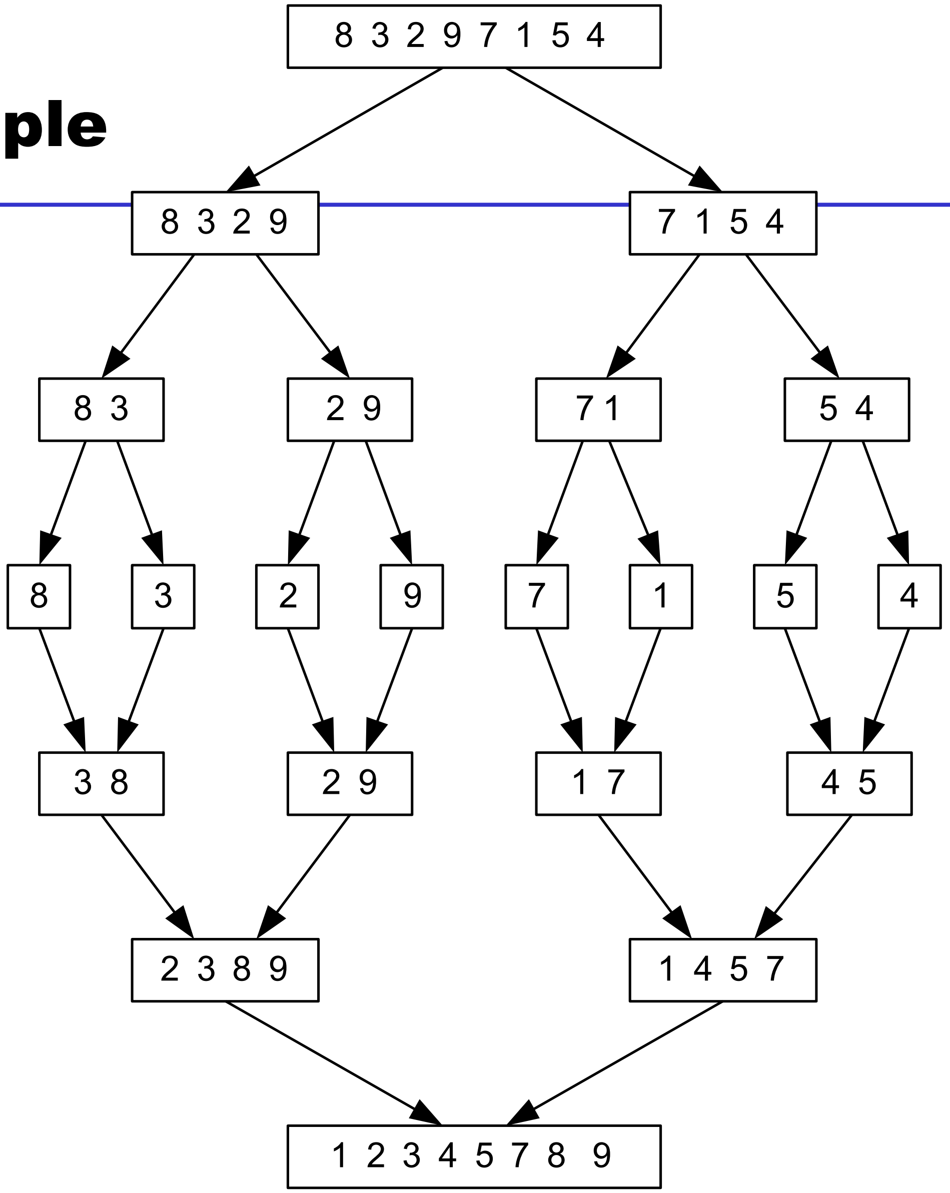
Split array A[1..*n*] in two and make copies of each half in arrays B[1.. $\lfloor n/2 \rfloor$] and C[1.. $\lceil n/2 \rceil$]

Sort arrays B and C

Merge sorted arrays B and C into array A as follows:

- Repeat the following until no elements remain in one of the arrays:
  - compare the first elements in the remaining unprocessed portions of the arrays
  - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
- Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Mergesort Example



Algorithm in Pseudocode

```
ALGORITHM $MergeSort(A[0..n-1])$
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 $C[0..\lceil n/2 \rceil - 1]$
    $MergeSort\ (B[0..\lfloor n/2 \rfloor - 1])$
    $MergeSort\ (C[0..\lceil n/2 \rceil - 1])$
    $Merge(B, C, A)$
```

Merge Algorithm in Pseudocode

```
**ALGORITHM** $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$
$i \leftarrow 0; j \leftarrow 0; k \leftarrow 0$
**while** $i < p$ **and** $j < q$ **do**
    **if** $B[i] \le C[j]$
        $A[k] \leftarrow B[i]; i \leftarrow i+1$
    **else** $A[k] \leftarrow \mathrm{C}[j]; j \leftarrow j+1$
    $k \leftarrow k+1$
**if** $i = p$
    **copy** $C[j..q-1]$ **to** $A[k..p+q-1]$
**else**
    **copy** $B[i..p-1]$ **to** $A[k..p+q-1]$
```

Efficiency

Recurrence

$$C(n)=2C(n/2)+C_{merge}(n) \text{ for } n>1, \quad C(1)=0$$

Basic operation is a comparison and we have

$$C_{merge}(n)=n-1 \text{ ------ worst case}$$

General Divide and Conquer Recurrence

$$C(n) = 2C\left(\frac{n}{2}\right) + n - 1, for\ n > 1 \qquad C(1) = 0$$

$T(n) = aT(n/b) + f(n)$ $\qquad$ **where $f(n) \in \Theta(n^k)$**

$a < b^k$	**$T(n) \in \Theta(n^k)$**
$a = b^k$	**$T(n) \in \Theta(n^k \log n)$**
$a > b^k$	**$T(n) \in \Theta(n^{\log_b a})$**

$a=2$, $b=2$, $k=1$
$a=b^k$, $C(n)$ belongs to $\Theta(n\log n)$

Efficiency

Recurrence

$$C(n)=2C(n/2)+C_{merge}(n) \text{ for } n>1, \quad C(1)=0$$

Basic operation is a comparison and we have

$$C_{merge}(n)=n-1 \text{ ------ worst case}$$

Using the Master Theorem, the complexity of mergesort algorithm is

$$\Theta(n \log n)$$

It is more efficient than SelectionSort, BubbleSort and InsertionSort, where the time complexity is $\Theta(n^2)$

Quicksort

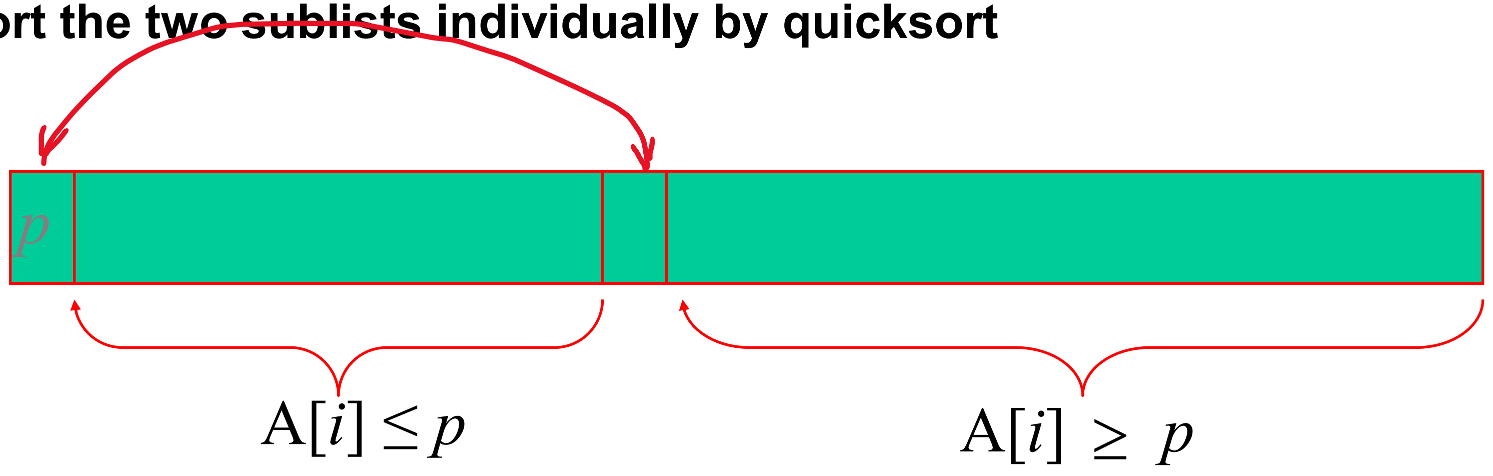
Select a *pivot* (partitioning element)

Rearrange the list so that all the elements in the positions before the pivot are smaller than or equal to the pivot and those after the pivot are larger than or equal to the pivot

Exchange the pivot with the last element in the first (i.e., ≤) sublist– the pivot is now in its final position

Partition into two sublists.

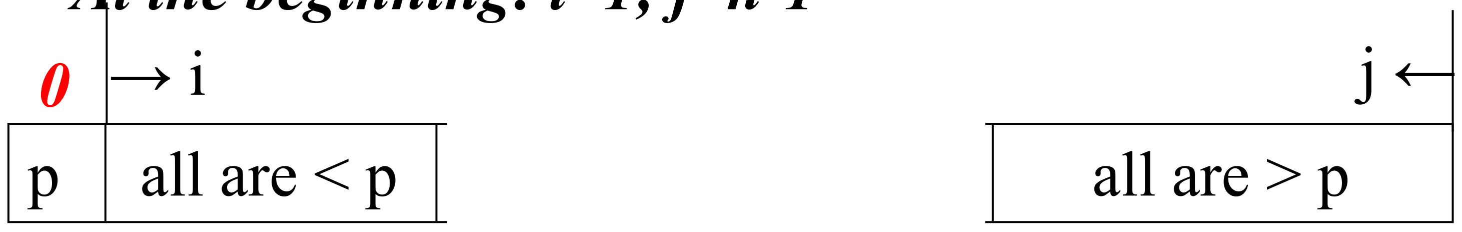
Sort the two sublists individually by quicksort



Illustrations

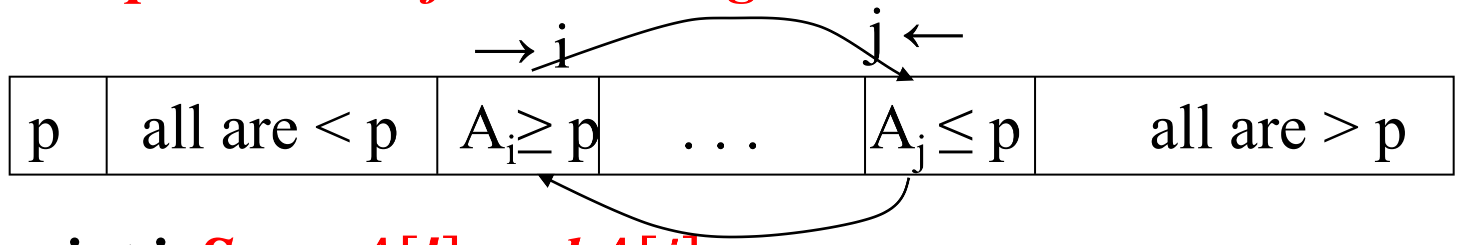
Search from left to right and right to left simultaneously

At the beginning: i=1; j=n-1



Stop searching while the conditions violate the requirements

Case 1: stop earlier before meeting with each other

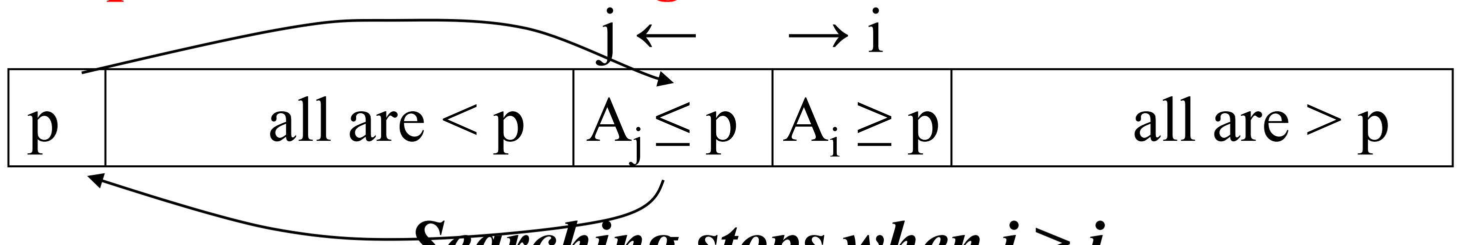


i < j, Swap A[i] and A[j]

After swapping, keep searching

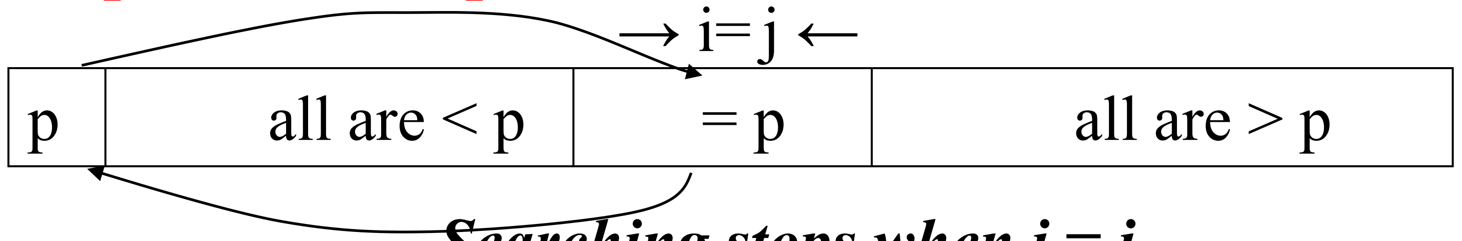
Illustrations

Case 2: stop when two searching directions cross



Searching stops when i > j

Case 3: stop at the same position



Searching stops when i = j

For both two cases, the pivot position = j

Swap $A[p]$ and $A[j]$

QuickSort Algorithm

```
**ALGORITHM** $QuickSort(A[l..r])$
**if** $l < r$
    $s \leftarrow Partition(A[l..r])$ // $s$ **is a split position**
    $QuickSort\ A[l..s-1]$
    $QuickSort\ A[s+1..r]$
```

The Partition Algorithm

```
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]    ← The leftmost element in the subarray is chosen as the pivot
i ← l;  j ← r + 1
repeat
    repeat i ← i + 1 until A[i] ≥ p  or i = r
    repeat j ← j − 1 until A[j] ≤ p  or j = l
    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
```

Quicksort Example

5 3 1 9 8 2 4 7

Initialization:
i=1 and j=7

0	1	2	3	4	5	6	7
	$i \longrightarrow$					$\longleftarrow$	j
5	3	1	9	8	2	4	7

From left to right,
compare:
5 and 3,
5 and 1,
5 and 9

From right to left,
compare:
5 and 7,
5 and 4

Quicksort Example

5 3 1 9 8 2 4 7

First stop

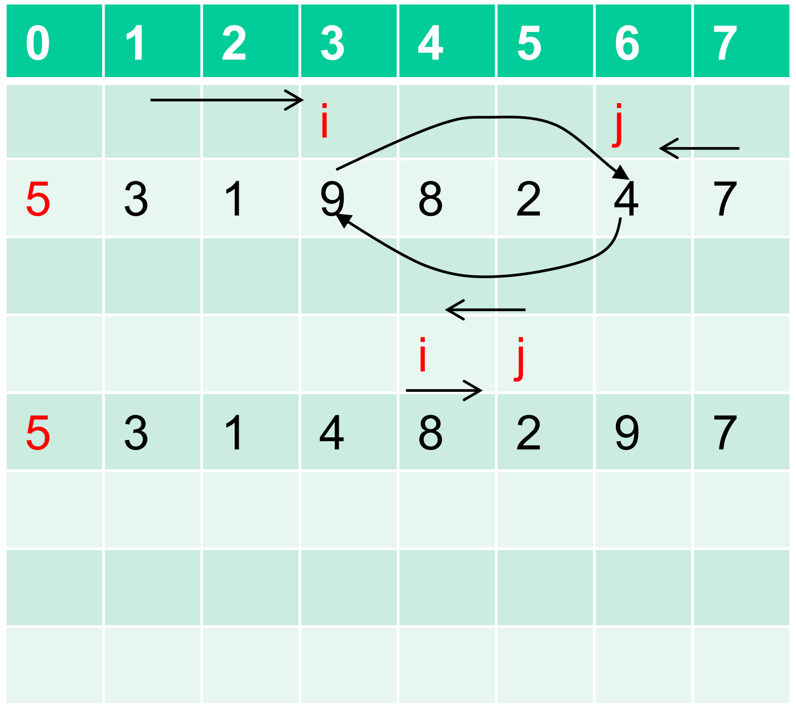
0	1	2	3	4	5	6	7
	$\longrightarrow$	$\longrightarrow$	i			j	$\longleftarrow$
5	3	1	9	8	2	4	7

Quicksort Example

5 3 1 9 8 2 4 7

Swap 4 and 9

Keep working:
From left to right, compare:
5 and 8



0	1	2	3	4	5	6	7
			i			j	
5	3	1	9	8	2	4	7
				i	j		
5	3	1	4	8	2	9	7

From right to left, compare:
5 and 2

Quicksort Example

5 3 1 9 8 2 4 7

Second stop --
Swap 8 and 2

0	1	2	3	4	5	6	7
			→	i	j	←	
5	3	1	9	8	2	4	7
			←	j	i →		
5	3	1	4	2	8	9	7

Keep working:
From left to right,
compare:
5 and 8

From right to left,
compare:
5 and 2

Quicksort Example

5 3 1 9 8 2 4 7

0	1	2	3	4	5	6	7
				j	i		
5	3	1	4	2	8	9	7

l=0, r=7
S=4

i >= j ⇨ ***Pivot position s=4***

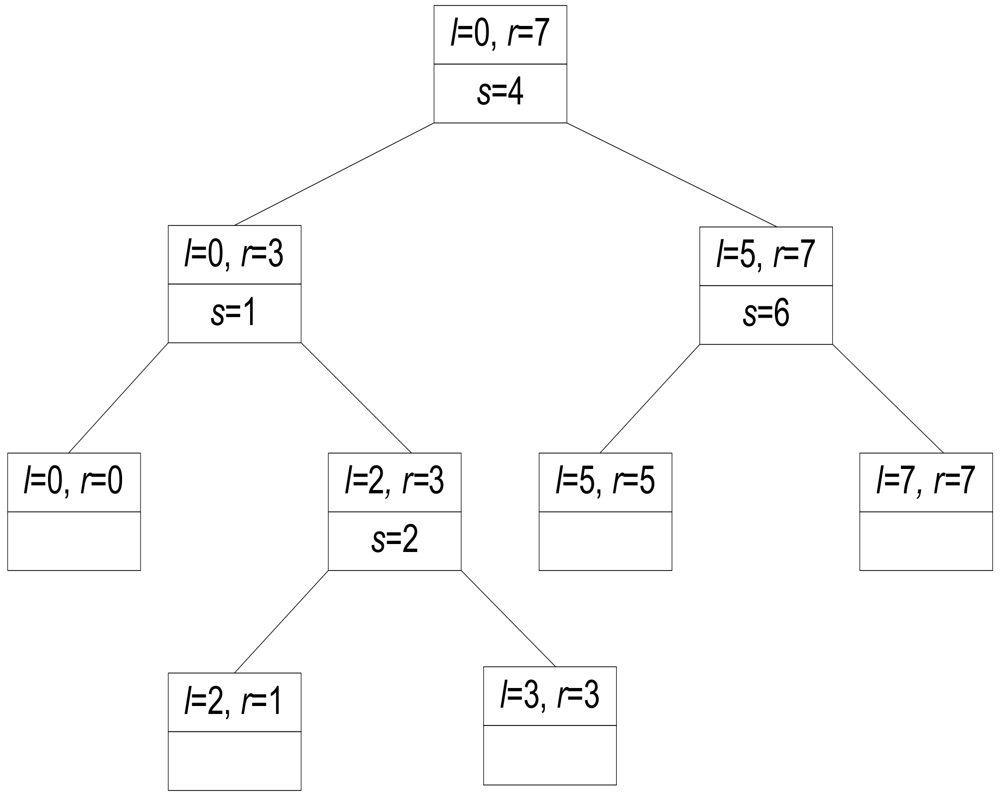
Quicksort Example

5 3 1 9 8 2 4 7

0	1	2	3	4	5	6	7
	→	→	i			j	←
5	3	1	9	8	2	4	7
		→	→	i	j	←	
5	3	1	4	8	2	9	7
				j	i		
5	3	1	4	2	8	9	7
2	3	1	4	5	8	9	7

Pivot position s=4

Perform quicksort on these two new arrays separately



l=0, r=7
s=4
l=0, r=3
s=1
l=5, r=7
s=6
l=0, r=0
l=2, r=3
s=2
l=5, r=5
l=7, r=7
l=2, r=1
l=3, r=3