

# SORTING ALGORITHMS

CS210/SE202  
Data Structures and Algorithms

## Sorting

- Fundamental application for computers
- one of the most well-studied operations in computer science
- Sorting
  - in main memory (small number of items)
  - external sorting - disk or tape

CS210/SE202  
Data Structures and Algorithms

## Sorting Techniques

Some of the sorting techniques used to arrange data are as follows :

- Insertion
- Selection
- Bubble
- Heap
- Mergesort
- Quicksort

CS210/SE202  
Data Structures and Algorithms

## Insertion Sort



The general idea of the insertion sort method is that for each element, find the slot where it belongs.

CS210/SE202  
Data Structures and Algorithms

## Example

- The element in position `Array[0]` is certainly sorted.
- Thus, move on to insert the second character, `D`, into the appropriate location to maintain the alphabetical order.

		SORTED		UNSORTED			
Array		G	D	Z	F	B	E
Index		0	1	2	3	4	5

		SORTED		UNSORTED			
Array		D	G	Z	F	B	E
Index		0	1	2	3	4	5

CS210/SE202  
Data Structures and Algorithms

## How does it work?

- Each element `Array[j]` is taken one at a time from  $j = 0$  to  $n-1$ .
- Before insertion of `Array[j]`, the subarray from `Array[0]` to `Array[j-1]` is sorted, and the remainder of the array is not.
- After insertion, `Array[0...j]` is correctly ordered, while the subarray with elements `Array[j+1].....Array[n-1]` is unsorted.

CS210/SE202  
Data Structures and Algorithms

### Insertion Sort technique.

	SORTED			UNSORTED		
Array	D	G	Z	F	B	E
Index	0	1	2	3	4	5

	SORTED			UNSORTED		
Array	D	F	G	Z	B	E
Index	0	1	2	3	4	5

CS210/SE202  
Data Structures and Algorithms

### Insertion Sort technique.

	SORTED				UNSORTED	
Array	B	D	F	G	Z	E
Index	0	1	2	3	4	5

	SORTED					
Array	B	D	E	F	G	Z
Index	0	1	2	3	4	5

CS210/SE202  
Data Structures and Algorithms

### Insertion Sort Algorithm

```

for i = 1 to n-1
    temp = a[i]
    loc = i
    while(loc>0 && (a[loc-1]> temp))
        a[loc] = a[loc-1]
        loc = loc - 1
    a[loc] = temp
    
```

CS210/SE202  
Data Structures and Algorithms

### Insertion sort

- The initial state is that the first element, considered by itself, is sorted
- The final state is that all elements, considered as a group, are sorted.
- Basic action is to arrange that elements in positions 0 through 'i'. In each stage 'i' increases by 1. The outer loop controls this.

CS210/SE202  
Data Structures and Algorithms

### Insertion sort

- When the body of the outer for loop is entered, we know that elements at positions 0 through 'i' are sorted and we need to extend this to positions 0 to n-1.
- At each step the element indexed by 'i' needs to be added to the sorted part of the array. This is done by placing it in a temporary variable and sliding all elements larger than it one position to the right.
- Then the temporary element is copied into the leftmost relocated element. The counter 'loc' indicates this position.

CS210/SE202  
Data Structures and Algorithms

### Complexity

- **Best situation:** the data is already sorted. The inner loop is never executed, and the outer loop is executed  $n - 1$  times for total complexity of  $O(n)$ .

CS210/SE202  
Data Structures and Algorithms

## Complexity

- **Worst situation:** data in reverse order. The inner loop is executed the maximum number of times. Thus the complexity of the insertion sort in this worst possible case is quadratic or  $O(n^2)$ .

CS210/SE202  
Data Structures and Algorithms

## Selection Sort



The general idea of the selection sort is that for each slot, find the element that belongs there.

CS210/SE202  
Data Structures and Algorithms

## Selection Sort Algorithm

```
for i = 0 to n-2
  temp = a[i]
  loc = i
  for j = i+1 to n-1
    if a[j] < a[loc]
      loc = j
  a[i] = a[loc]
  a[loc] = temp
```

CS210/SE202  
Data Structures and Algorithms

- The inner loop in the above algorithm finds the location of the next smallest element in the array (the location of the next smallest element in the array).
- The outer loop moves along the array as the elements are sorted.
- What is the complexity of selection sort?

CS210/SE202  
Data Structures and Algorithms

## MergeSort

- *Merging* involves the combination of two or more ordered files into a single ordered file.
  - **Example:** Merge the two files:
    - 503, 703, 765    ➡    087, 503, 512, 677, 703, 765
    - 087, 512, 677
  - This is solved by comparing the two smallest items, output the smallest, and then repeat the process.
- Be careful when one of the two files become exhausted!

CS210/SE202  
Data Structures and Algorithms

## Exercise:

- Write a function which merges two sorted arrays of length  $n$  ( $n > 0$ ) and  $m$  ( $m > 0$ ) respectively.
- The function will have the following form:  
/\*\*Merges the sorted subarrays A[p...q] and A[q+1...r] to form a single sorted array A[p ... r] \*/  
MERGE (A,p,q,r)

This Algorithm should be of complexity  $O(n)$

CS210/SE202  
Data Structures and Algorithms

### Merge - example

A: 

1	13	24	26	2	15	27	38
---	----	----	----	---	----	----	----

p ↑ q ↑ start ↑ r ↑

Compare 1 and 2,  
1 is added to C,  
then compare 13 and 2.

start = q+1;

1	13	24	26
---	----	----	----

2	15	27	38
---	----	----	----

p ↑ q+1 ↑

1							
---	--	--	--	--	--	--	--

pos ↑

CS210/SE202  
Data Structures and Algorithms

### Merge - example

A: 

1	13	24	26	2	15	27	38
---	----	----	----	---	----	----	----

p ↑ q ↑ r ↑

C: 

1	2	13	15	24	26	27	38
---	---	----	----	----	----	----	----

p ↑ r ↑

CS210/SE202  
Data Structures and Algorithms

### Merging Two Sequences (cont.)

• Some pictures:

a)

$S_1$ 

24	45	63	85
----	----	----	----

$S_2$ 

17	31	50	96
----	----	----	----

$S$

b)

$S_1$ 

24	45	63	85
----	----	----	----

$S_2$ 

31	50	96
----	----	----

$S$ 

17
----

CS210/SE202  
Data Structures and Algorithms

### Merging Two Sequences (cont.)

c)

$S_1$ 

45	63	85
----	----	----

$S_2$ 

31	50	96
----	----	----

$S$ 

17	24
----	----

d)

$S_1$ 

45	63	85
----	----	----

$S_2$ 

50	96
----	----

$S$ 

17	24	31
----	----	----

CS210/SE202  
Data Structures and Algorithms

### Merging Two Sequences (cont.)

e)

$S_1$ 

63	85
----	----

$S_2$ 

50	96
----	----

$S$ 

17	24	31	45
----	----	----	----

f)

$S_1$ 

63	85
----	----

$S_2$ 

96
----

$S$ 

17	24	31	45	50
----	----	----	----	----

CS210/SE202  
Data Structures and Algorithms

### Merging Two Sequences (cont.)

g)

$S_1$ 

85
----

$S_2$ 

96
----

$S$ 

17	24	31	45	50	63
----	----	----	----	----	----

h)

$S_1$

$S_2$ 

96
----

$S$ 

17	24	31	45	50	63	85
----	----	----	----	----	----	----

CS210/SE202  
Data Structures and Algorithms

## Merging Two Sequences (cont.)



CS210/SE202  
Data Structures and Algorithms

## Divide and Conquer

- **Divide:** To solve a problem, it is subdivided into sub problems each of which is similar to the given problem.
- **Conquer:** Each of the sub problems is solved independently
- **Combine:** The solutions to the sub problems are combined in order to obtain the solution to the original problem

CS210/SE202  
Data Structures and Algorithms

## Divide and Conquer

Divide and conquer algorithms are often implemented using recursion. However not all recursive functions are divide and conquer algorithm. Generally, the sub problems solved by divide and conquer algorithm are non overlapping

Let's look at a classical example....

CS210/SE202  
Data Structures and Algorithms

## Towers of Hanoi 1

- Given 3 pegs A,B,C
- Peg A has  $n$  disks, starting with the largest one on the bottom and successively smaller ones on top.
- The object of the puzzle is to move the disks one at a time from peg to peg, never placing a larger one on top of a smaller one, eventually ending with all the disks on peg B

CS210/SE202  
Data Structures and Algorithms

## Towers of Hanoi 1

**Solution1:**

Imagine the pegs arranged in a triangle.

On odd numbered moves, move the smallest disk one peg clockwise.

On even numbered moves make the only legal move not involving the smallest disk.

CS210/SE202  
Data Structures and Algorithms

## Towers of Hanoi 2

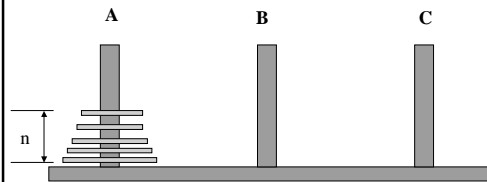
**Solution2:**

The divide and conquer technique.

The problem of moving the  $n$  smallest disks from A to B can be thought of as consisting of 2 sub problems of size  $n-1$

CS210/SE202  
Data Structures and Algorithms

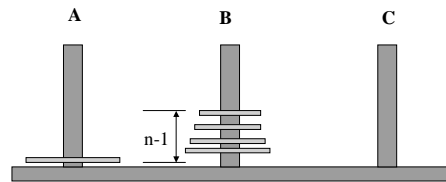
## Towers of Hanoi 2



CS210/SE202  
Data Structures and Algorithms

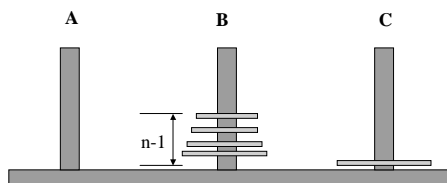
## Towers of Hanoi 2

Sub Problem 1: Get  $n-1$  disks from A to B



CS210/SE202  
Data Structures and Algorithms

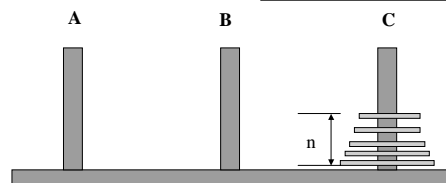
## Towers of Hanoi 2



CS210/SE202  
Data Structures and Algorithms

## Towers of Hanoi 2

Sub Problem 2: Get  $n-1$  disks from B to C



CS210/SE202  
Data Structures and Algorithms

## Towers of Hanoi 2

- First move the  $n-1$  smallest disks from A to C exposing the  $n$ th smallest disk on peg A.
- Move that disk from A to B
- Then move the  $n-1$  smallest disks from C to B
- The movement of the  $n-1$  smallest disks is accomplished by a recursive application of the procedure

CS210/SE202  
Data Structures and Algorithms

## Back to MergeSort...

- In a Divide and conquer algorithm
  - two half-sized problems are solved recursively
- MergeSort is such an algorithm

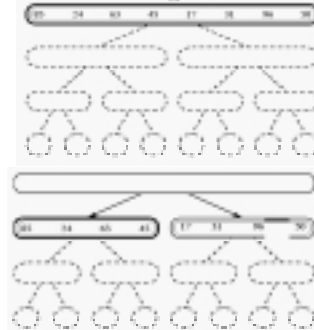
CS210/SE202  
Data Structures and Algorithms

## Merge Sort

- Using this Divide and Conquer Strategy:
- **Divide:** Divide the  $n$  element sequence to be sorted into two sub sequences of  $n/2$  elements each
- **Conquer:** Sort the two sub sequences recursively using merge sort
- **Combine:** Merge the two sorted sub sequences to produce a sorted answer
- **Note:** The recursion bottoms out when the length of the sequence to be sorted is 1.

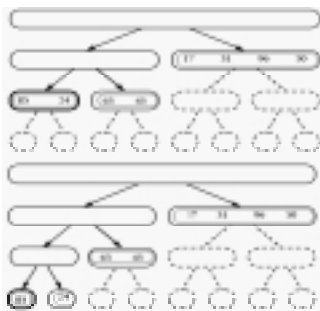
CS210/SE202  
Data Structures and Algorithms

## Merge-Sort



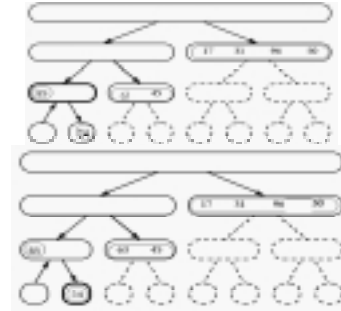
CS210/SE202  
Data Structures and Algorithms

## Merge-Sort(cont.)



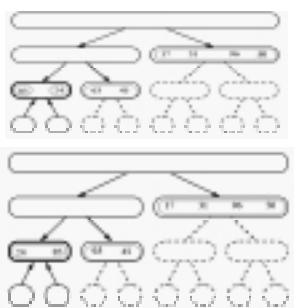
CS210/SE202  
Data Structures and Algorithms

## Merge-Sort (cont.)



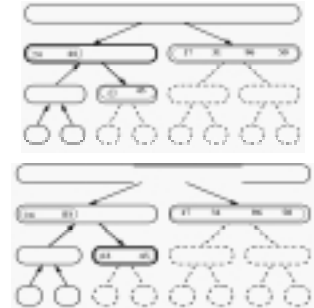
CS210/SE202  
Data Structures and Algorithms

## Merge-Sort (cont.)



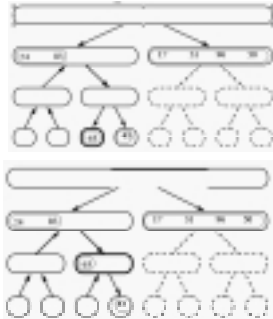
CS210/SE202  
Data Structures and Algorithms

## Merge-Sort (cont.)



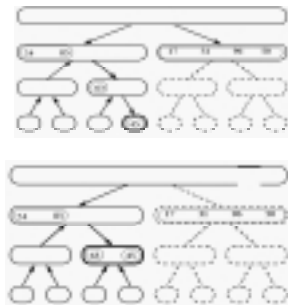
CS210/SE202  
Data Structures and Algorithms

### Merge-Sort (cont.)



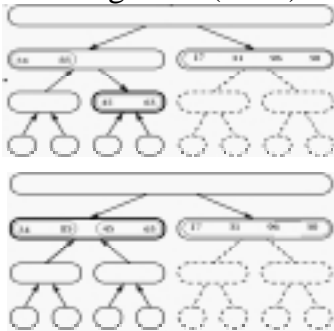
CS210/SE202  
Data Structures and Algorithms

### Merge-Sort (cont.)



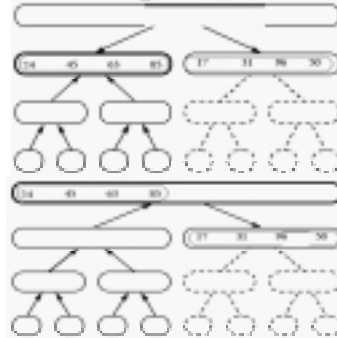
CS210/SE202  
Data Structures and Algorithms

### Merge-Sort(cont.)



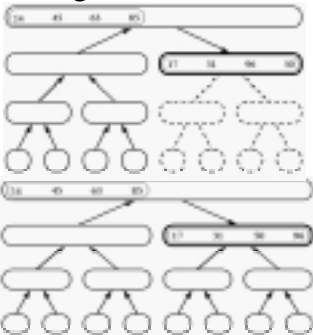
CS210/SE202  
Data Structures and Algorithms

### Merge-Sort (cont.)



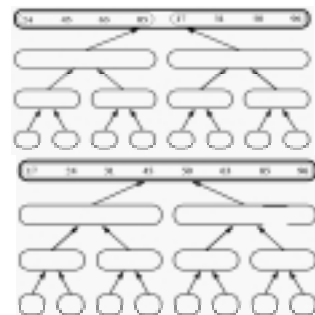
CS210/SE202  
Data Structures and Algorithms

### Merge-Sort (cont.)



CS210/SE202  
Data Structures and Algorithms

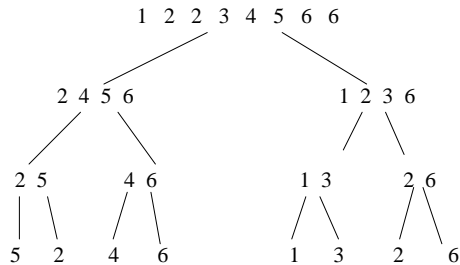
### Merge-Sort (cont.)



CS210/SE202  
Data Structures and Algorithms



## Merge sort 5,2,4,6,1,3,2,6



CS210/SE202  
Data Structures and Algorithms

## Mergesort

- MERGESORT(A,p,r) sorts the elements in the subarray A[p..r]

If  $p \geq r$ , the subarray has at most one element and is therefore already sorted.

Otherwise the divide step calculates an index  $q$  that partitions A[p..r] into two subarrays containing  $\lceil n/2 \rceil$  elements and A[q+1 ...r] containing  $\lfloor n/2 \rfloor$  elements.

- MERGESORT(A,0,n-1) sorts the array A of length  $n$

CS210/SE202  
Data Structures and Algorithms

### MERGESORT(A,p,r)

```

{
  if the subarray has more than one element then
    Divide array into two subarrays
    Call MERGESORT on subarray 1
    Call MERGESORT on subarray 2
    Merge these SORTED subarrays
}
  
```

When an algorithm contains a recursive call to itself its running time can be described by a recurrence equation or recurrence.

CS210/SE202  
Data Structures and Algorithms

### MERGESORT(A,p,r)

```

{
  if ( p < r ) then
    q = ⌊ (p+r) / 2 ⌋
    MERGESORT(A,p,q)
    MERGESORT(A,q+1,r)
    MERGE(A,p,q,r)
}
  
```

When an algorithm contains a recursive call to itself its running time can be described by a recurrence equation or recurrence

CS210/SE202  
Data Structures and Algorithms

## Merging Two Sequences

• Pseudo-code for merging two sorted sequences into a unique sorted sequence

**Algorithm merge** (S1, S2, S):

**Input:** Sequences S1 and S2 (on whose elements a total order relation is defined) sorted in nondecreasing order, and an empty sequence S.

**Output:** Sequence S containing the union of the elements from S1 and S2 sorted in nondecreasing order; sequence S1 and S2 become empty at the end of the execution

```

while S1 is not empty and S2 is not empty do
  if S1.first().element() < S2.first().element() then
    {move the first element of S1 at the end of S}
    S.insertLast(S1.remove(S1.first()))
  else
    {move the first element of S2 at the end of S}
    S.insertLast(S2.remove(S2.first()))
while S1 is not empty do
  S.insertLast(S1.remove(S1.first()))
while S2 is not empty do
  S.insertLast(S2.remove(S2.first()))
  
```

## Quicksort

- The Quicksort algorithm was developed by C.A.R. Hoare. It has the best average behaviour in terms of complexity:

Average case:  $O(n \log_2 n)$

Worst case:  $O(n^2)$

CS210/SE202  
Data Structures and Algorithms

## Quicksort

Given a list of elements,

- take a partitioning element
- and create a (sub)list
  - such that all elements to the left of the partitioning element are less than it,
  - and all elements to the right of it are greater than it.
- Now repeat this partitioning effort on each of these two sublist

CS210/SE202  
Data Structures and Algorithms

## Quicksort

- And so on in a recursive manner until all the sublists are empty, at which point the (total) list is sorted
- Partitioning can be effected simultaneously, scanning left to right and right to left, interchanging elements in the wrong parts of the list
- The partitioning element is then placed between the resultant sublists (which are then partitioned in the same manner)

CS210/SE202  
Data Structures and Algorithms

## Implementation of Quicksort

```

If anything to be partitioned THEN{
  choose a pivot
  REPEAT{
    scan from left to right until we find an element
    > pivot: i points to it

    scan from right to left until we find an element
    < pivot: loc points to it

    IF i < loc THEN
      exchange pivot and loth element
  }UNTIL i > loc
}
    
```

CS210/SE202  
Data Structures and Algorithms

## Implementation of Quicksort()

exchange pivot and loc element

partition from 1<sup>st</sup> to loc-1<sup>th</sup> elements  
(\* i.e. quicksort 1st to loc<sup>th</sup> \*)

partition from loc<sup>th</sup> to r<sup>th</sup> elements  
(i.e. quicksort loc<sup>th</sup> to r<sup>th</sup> \*)

CS210/SE202  
Data Structures and Algorithms

## Example

G	B	S	E	V	M	P	H	D	C	?
0	1	2	3	4	5	6	7	8	9	10

**i = 0**  
**first = 0**  
**loc = 10**  
**last = 9**  
**pivot = G**

CS210/SE202  
Data Structures and Algorithms

## Example

G	B	C	E	V	M	P	H	D	S	?
0	1	2	3	4	5	6	7	8	9	10

**i = 2**  
**first = 0**  
**loc = 9**  
**last = 9**

CS210/SE202  
Data Structures and Algorithms

### Example

*i* stops at 4 ( $a[4]$  or  $V \geq G$ ), and *loc* at 8 ( $a[8]$  or  $D \leq G$ ).  
These values are then swapped.

G	B	C	E	D	M	P	H	V	S	?
0	1	2	3	4	5	6	7	8	9	10

***i* = 4**  
**first = 0**  
**loc = 8**  
**last = 9**

CS210/SE202  
Data Structures and Algorithms

### Example

*i* and *loc* continue to move until  $i > loc$   
This happens when *i* points to M and *loc* points to D

G	B	C	E	D	M	P	H	V	S	?
0	1	2	3	4	5	6	7	8	9	10

Now exchanging  $a[loc]$  and pivot partitions *a*.

***i* = 5**  
**first = 0**  
**loc = 4**  
**last = 9**

CS210/SE202  
Data Structures and Algorithms

### Example

Algorithm for *partition(a,first,last,loc)*

***i* = first**  
**loc = last + 1**  
**pivot =  $a[first]$**

CS210/SE202  
Data Structures and Algorithms

### Recursive Quicksort Algorithm

if  $first < last$  then  
    partition(*a*,first,last,loc)  
    quicksort(*a*,first,loc-1)  
    quicksort(*a*,loc+1,last)

The pivot can be any of the array  
elements, such as the leftmost one.

e.g.

Pivot =  $a[first]$

CS210/SE202  
Data Structures and Algorithms

### Recursive Quicksort Algorithm

- Two indices, *i* and *loc*, need to be initialised to indicate the outside bounds: *first* and *last+1*, respectively of the elements to be divided into two collections.

CS210/SE202  
Data Structures and Algorithms

### Recursive Quicksort Algorithm

- i* = first**  
**loc = last + 1**
- Although *last+1* may be out of bounds for array *a*, there is no error because  $a[last+1]$  is never accessed, because *loc* is decremented first before referencing  $a[loc]$ .
- In the partition procedure, we wish to have the elements less than or equal to the pivot toward the left and those greater than or equal toward the right.

CS210/SE202  
Data Structures and Algorithms

## Quicksort

Given a list of elements,

- take a partitioning element
- and create a (sub)list
  - such that all elements to the left of the partitioning element are less than it,
  - and all elements to the right of it are greater than it.
- Now repeat this partitioning effort on each of these two sublist

CS210/SE202  
Data Structures and Algorithms

## Quicksort

- And so on in a recursive manner until all the sublists are empty, at which point the (total) list is sorted
- Partitioning can be effected simultaneously, scanning left to right and right to left, interchanging elements in the wrong parts of the list
- The partitioning element is then placed between the resultant sublists (which are then partitioned in the same manner)

CS210/SE202  
Data Structures and Algorithms

## The Partition

```
do{
    do {
        i = i+1;
    } while (a[i] < pivot && (i < last));
    do {
        loc = loc-1;
    } while (a[loc] > pivot);
    if (i < loc){
        swap (a(i), a(loc))
    }
} while (i < loc);
swap (a(first), a(loc))
}
```

## Recursive Quicksort Algorithm

```
if first < last then
    partition(a,first,last,loc)
    quicksort(a,first,loc-1)
    quicksort(a,loc+1,last)
```

The pivot can be any of the array elements, such as the leftmost one.

**Pivot = a[first]**

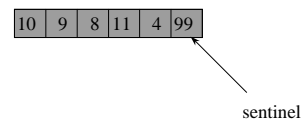
CS210/SE202  
Data Structures and Algorithms

## Divide and Conquer

- Quicksort is a divide and Conquer Algorithm
- A divide-and-conquer algorithm is one that divides the problem into smaller problems upon which it performs the same process.

CS210/SE202  
Data Structures and Algorithms

## Quicksort



CS210/SE202  
Data Structures and Algorithms

10	9	8	11	4	99
----	---	---	----	---	----

**QS(A, , )**

**L:**

**R:**

**i:**

**j:**

**pivot:**

$\uparrow$   
 $\uparrow$   
**i**   **j**

```

do{
    do {
        i = i+1;
    } while (a[i] < pivot && (i < last));
    do {
        loc = loc-1;
    } while (a[loc] > pivot);
    if (i < loc){
        swap (a[i], a[loc])
    }
} while (i < loc);

swap (a(first), a(loc))
}
        
```

# Quicksort

10	9	8	11	4	99
----	---	---	----	---	----

↑  
**i**

↑  
**j**

**QS(A,1 ,6 )**

**L:     1**

**R:     6**

**i:     1**

**j:     6**

**pivot: 10**

```
do{  
    do {  
        i = i+1;  
    } while (a[i] < pivot && (i < last));  
    do {  
        loc = loc-1;  
    } while (a[loc] > pivot);  
    if (i < loc){  
        swap (a(i), a(loc))  
    }  
} while (i < loc);  
  
swap (a(first), a(loc))  
}
```

## Quicksort

10	9	8	11	4	99
----	---	---	----	---	----

↑    ↑

**i    j**

**QS(A,1,6)**

**L:    1**

**R:    6**

**i:    1 2 3 4**

**j:    6 5**

**pivot: 10**

```

do{
    do {
        i = i+1;
    } while (a[i] < pivot && (i < last));
    do {
        loc = loc-1;
    } while (a[loc] > pivot);
    if (i < loc){
        swap (a(i), a(loc))
    }
} while (i < loc);

swap (a(first), a(loc))
        
```

## Quicksort

10	9	8	4	11	99
----	---	---	---	----	----

**i    j**

↑    ↑

**swap**

**QS(A,1 ,6 )**

**L:        1**

**R:        6**

**i:        1 2 3 4**

**j:        6 5**

**pivot: 10**

```
do{
    do {
        i = i+1;
    } while (a[i] < pivot && (i < last));
    do {
        loc = loc-1;
    } while (a[loc] > pivot);
    if (i < loc){
        swap (a(i), a(loc))
    }
} while (i < loc);

swap (a(first), a(loc))
}
```

## Quicksort

10	9	8	4	11	99
----	---	---	---	----	----

↑    ↑

**j   i**

**QS(A,1 ,6 )**

**L:     1**

**R:     6**

**i:     1 2 3 4 5**

**j:     6 5 4**

**pivot: 10**

```

do{
    do {
        i = i+1;
    } while (a[i] < pivot && (i < last));
    do {
        loc = loc-1;
    } while (a[loc] > pivot);
    if (i < loc){
        swap (a(i), a(loc))
    }
} while (i < loc);

swap (a(first), a(loc))
}
        
```

# Quicksort

4	9	8	10	11	99
---	---	---	----	----	----

↑    ↑

**j   i**

**QS(A,1 ,6 )**

**L:     1**

**R:     6**

**i:     1 2 3 4 5**

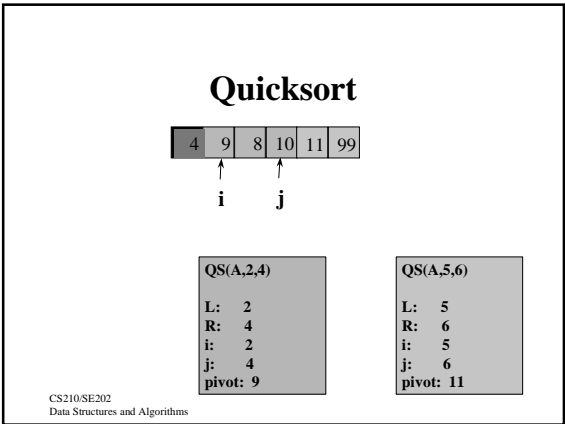
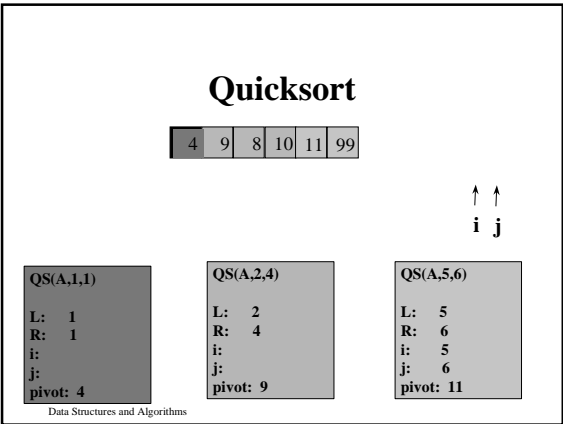
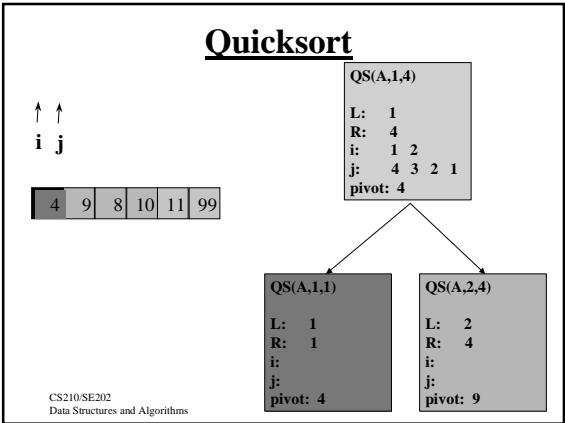
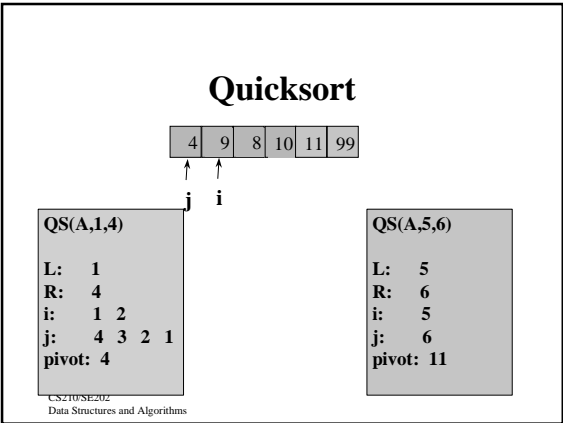
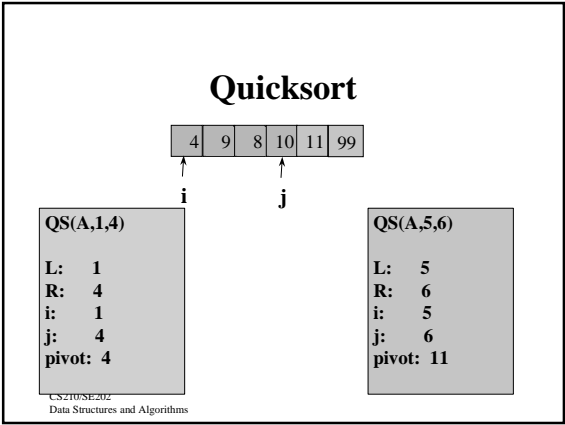
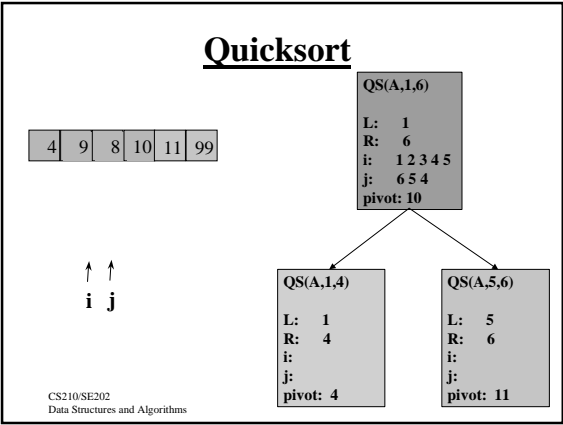
**j:     6 5 4**

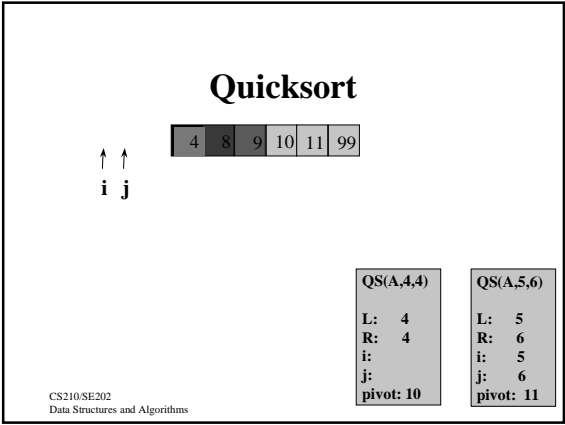
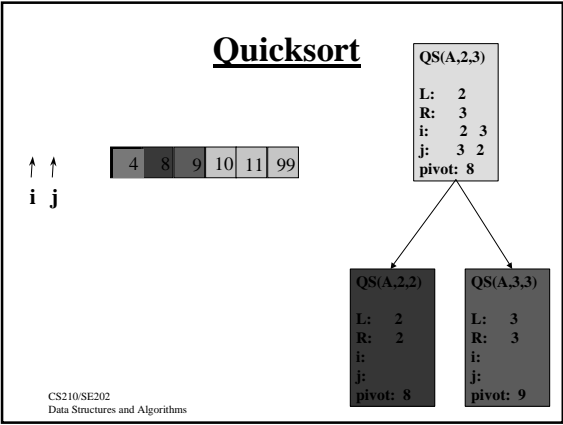
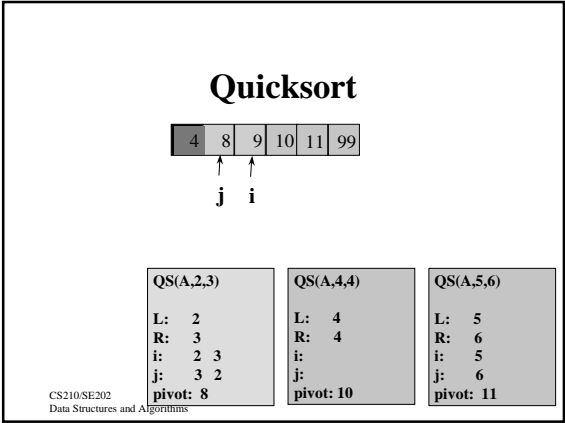
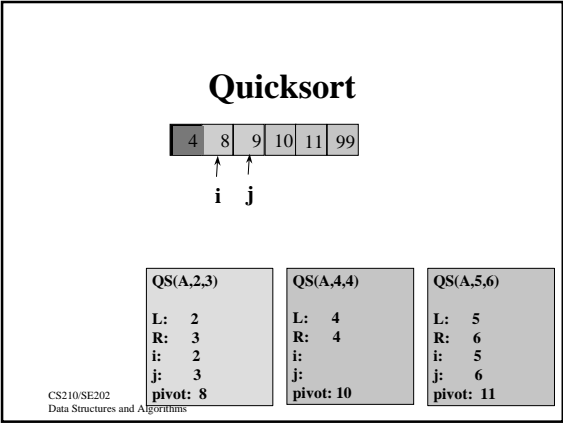
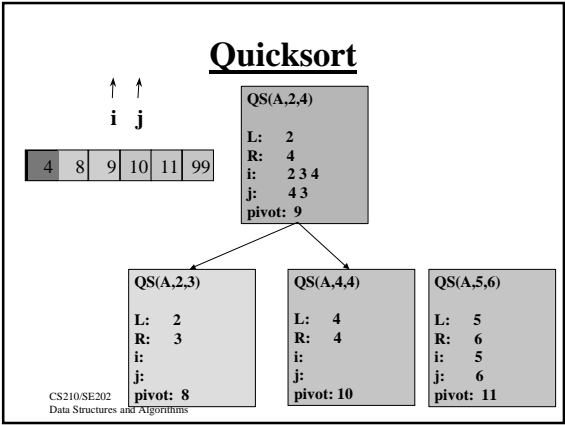
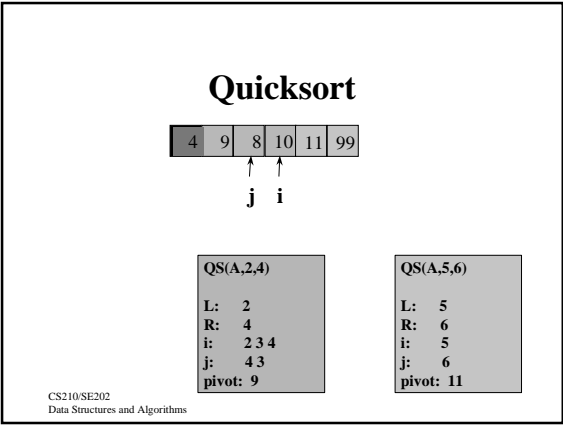
**pivot: 10**

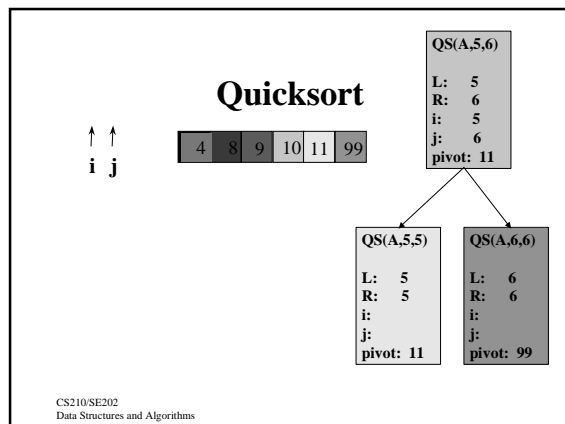
```

do{
    do {
        i = i+1;
    } while (a[i] < pivot && (i < last));
    do {
        loc = loc-1;
    } while (a[loc] > pivot);
    if (i < loc){
        swap (a(i), a(loc))
    }
} while (i < loc);

swap (a(first), a(loc))
}
                
```







## Quicksort

- Fastest known sorting algorithm
- Relies on recursion and relatively simple to understand

CS210/SE202  
Data Structures and Algorithms

## Quicksort Algorithm

### Quicksort(S)

- If the number of elements in **S** is 0 or 1, then return
- Pick *any* element **v** in **S**. This is called the *pivot*.
- *Partition S-{v}* (the remaining elements in **S**) into disjoint groups:
  - $L = \{x \in S - \{v\} \mid x \leq v\}$  and  $R = \{x \in S - \{v\} \mid x > v\}$
- Return the result of **Quicksort(L)** followed by **v** followed by **Quicksort(R)**.

CS210/SE202  
Data Structures and Algorithms

## Quicksort

- The algorithm allows any element to be used as the *pivot*.
- The *pivot* divides the array elements into two groups
  - elements that are smaller than the *pivot*
  - elements that are larger than the *pivot*
- Some choices for the *pivot* are better than others, so it is good to make an educated choice for the *pivot*.

CS210/SE202  
Data Structures and Algorithms

## Quicksort

- In the *partition* step every element in **S**, except the *pivot*, is placed in either **L** (the left part of the array) or in **R** (the right part of the array).
  - Elements that are smaller than the *pivot* go to **L**
  - Elements that are greater than the *pivot* go to **R**.
  - Note: We need to take account of duplicate elements.
- The reason **Quicksort** is faster than **Mergesort** is that the partitioning step can be performed significantly faster than the merging step.
  - the partitioning step can be performed without using an extra array.

CS210/SE202  
Data Structures and Algorithms

## Picking the Pivot

- The wrong way
  - A popular, uninformed choice is to use the first element as pivot.
  - If input is presorted or in reverse order, this is a poor choice as the pivot will be an extreme element and the behaviour will continue recursively.
  - Never use first element as the pivot.
- A safe choice
  - a reasonable choice is the middle element

CS210/SE202  
Data Structures and Algorithms



## Median-of-three Partitioning

- This is an attempt to pick a better than average *pivot*.
- We take a sample of three numbers and find their median, the first, middle and last elements.
  - E.g. input: 8, 1, 4, 9, 6, 3, 5, 2, 7, 0
  - leftmost is 8, rightmost is 0, middle is 6
  - Here 6 is the *pivot*.

CS210/SE202  
Data Structures and Algorithms

## A partitioning strategy

- We look at a simple strategy for partitioning and then at the advantages of median-of-three partitioning
- First get the pivot out of the way by swapping it with the last element.

8	1	4	9	0	3	5	2	7	6
---	---	---	---	---	---	---	---	---	---

- Move all small elements to the left part of the array and all large elements to the right part
  - small and large are relative to the *pivot*.

CS210/SE202  
Data Structures and Algorithms

## Partitioning strategy

- Search from left to right looking for large elements - we use a counter *i*, initialized to position **Low** (leftmost element)
- Search from right to left looking for small elements - we use a counter *j*, initialized to position **High-1**.
- In the example, elements which are not known to be correctly placed are dark. Correctly placed cells are white

CS210/SE202  
Data Structures and Algorithms

- *i* initially stops at 8 and *j* at 2

8	1	4	9	0	3	5	2		6
<i>i</i>							<i>j</i>		

- By skipping 7, we know that it is not smaller than the *pivot* and so is correctly placed.
- We now swap the large element 8 on the left of the array and the small element 2 on the right to correctly place them.

	1	4	9	0	3	5			6
--	---	---	---	---	---	---	--	--	---

CS210/SE202  
Data Structures and Algorithms

- AS the algorithm continues *i* stops at 9 and *j* stops at 5. elements that *i* and *j* skip are guaranteed to be correctly placed.

	1	4	9	0	3	5			6
			<i>i</i>			<i>j</i>			

- elements 9 and 5 are then swapped.

				0	3				6
--	--	--	--	---	---	--	--	--	---

CS210/SE202  
Data Structures and Algorithms

- scan continues with *i* stopping at 9 and *j* stopping at 3. However *i* and *j* have crossed positions so a swap would be useless. We can see that 3 is already correctly placed.

					9				6
				<i>j</i>	<i>i</i>				

- All but two items are correctly placed, so swap element in position *i* with last cell (the *pivot*).

					6				
--	--	--	--	--	---	--	--	--	--

CS210/SE202  
Data Structures and Algorithms

## Keys equal to the pivot

- We require that both i and j stop when they encounter an element equal to the pivot.
- This avoids a worst-case run-time for the algorithm which is the same as using the first element as the pivot.

	1	4	9	6	3	5	2	7	
--	---	---	---	---	---	---	---	---	--

CS210/SE202  
Data Structures and Algorithms

## Median-of-three partitioning

- With median-of-three partitioning we do a simple optimisation that saves comparisons and simplifies the code.
- The easiest way to find the median of the first, middle and last elements is to sort them.

8	1	4	9	6	3	5	2	7	0
---	---	---	---	---	---	---	---	---	---

the original array

	1	4	9	6	3	5	2	7	
--	---	---	---	---	---	---	---	---	--

result of sorting (first, middle, last)

CS210/SE202  
Data Structures and Algorithms

## Median-of-three partitioning

- the element in the first position is guaranteed to be smaller(or equal to) than the pivot
- the element in the last position is guaranteed to be greater(or equal to) than the pivot. So,
  - we should not swap the pivot with the last element. Instead swap the pivot with the next to last element.
  - We can start i at Low+1 and j at High-2.
  - We are guaranteed that when i searches for a large element it will stop as it at worst will encounter the pivot
  - We are guaranteed that when j searches for a small element it will stop as it at worst will encounter the first element.

CS210/SE202  
Data Structures and Algorithms

## Quicksort Algorithm

```
void Quicksort( int A[ ], int Low, int High)
{
    // sort Low, Middle, High
    int Middle = ( Low + High ) / 2;
    if ( A[ Middle ] < A[ Low ] )
        Swap( A[ Low ], A[ Middle ] );
    if ( A[ High ] < A[ Low ] )
        Swap( A[ Low ], A[ High ] );
    if ( A[ High ] < A[ Middle ] )
        Swap( A[ Middle ], A[ High ] );
}
```

CS210/SE202  
Data Structures and Algorithms

```
//Place pivot at Position High-1
int Pivot = A[ Middle ];
Swap( A[Middle], A[ High-1 ] );
```

```
//Begin Partitioning
int i, j;
for( i = Low, i = High-1; ; )
{
    while ( A[ ++i ] < Pivot );
    while ( Pivot < A[ --j ] );
    if ( i < j )
        Swap ( A[ i ], A[ j] );
    else
        break;
}
```

CS210/SE202  
Data Structures and Algorithms

```
Swap( A[ i ], A[ High-1 ] ); //Restore Pivot
```

```
Quicksort( A, Low, i-1); // Sort small elements
Quicksort( A, i+1, High); // Sort large elements
}
```

```
void Quicksort(int A[ ], int N)
{
    Quicksort(A,0,N-1);
}
```

CS210/SE202  
Data Structures and Algorithms