COMP20003 Workshop Week 7

Sorting Algorithms + A2? + MST?

- Sorting & Properties
- Insertion Sort and Selection Sort
- Quicksort

Assignment 2:
due this Friday

MST:
next Friday in Week 8

LAB

- Get all green ticks
- Assignment 2: Q&A

Sorting Algorithms

Remember?

- Selection Sort?
- Insertion Sort?
- Quick Sort?

Main properties considered when comparing sorting algorithms:

Time Complexity

- *Input-Insensitive:* The complexity is the same for all inputs of a given size, and is described using Big-$\boldsymbol{\theta}$
- *Input-Sensitive:* The complexity depends on the specific input, and is described for best-case, average-case, and worst-case scenarios.

Space Complexity

- *In-Place*: Rearranges the input without allocating extra storage that grows with the input size.
- *Not In-Place*: Requires extra memory that grows with the input size (e.g., an auxiliary array of size O(n)).
- *Other Auxiliary Space*: The extra memory used for a sorting algorithm's internal operations, such as for a call stack in recursion. This should be considered when reporting total space usage.

***Stable*:** The algorithm maintains the relative order of elements with equal keys.

Review: Insertion Sort and Selection Sort for A[0..n-1]

	Selection Sort
Basic idea:	1. Treat the whole array as the unsorted part 2. Scan the unsorted part to find the smallest element and swap it into the first position. 3. The first element is now in its correct sorted place; treat the rest as the unsorted part. 4. Repeat step 2-3 until the unsorted part has only one element left.

Skeleton Code	`for (i=0; i<n-1; i++) {`
	`  // correct part:  A[0..i-1]`
	`  // unsorted subarray: A[i..n-1]`
	`  imin= index-of-a-minimal element of A[i..n-1]`
	`  swap( &A[i], &A[imin]);`
	`}`

Complexity:	
In-place?	
Stable? Why?	

Class Exercise: Trace the selection sort on the input array:

[5 9 4 5 1 2 7]

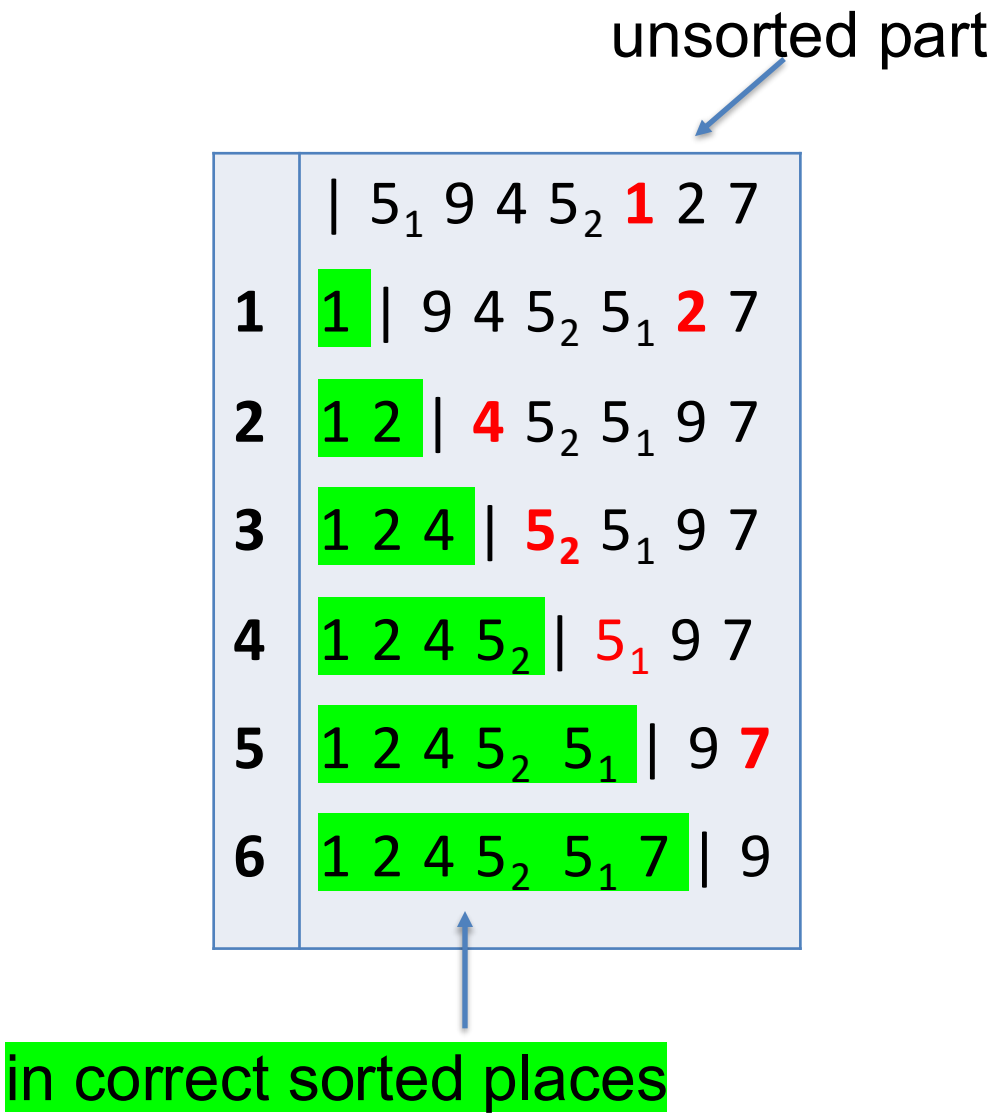
Class Exercise

Trace the selection sort on the input array:

5 9 4 5 1 2 7

How many key comparisons?
How many swaps?

Check your answer: Selection Sort



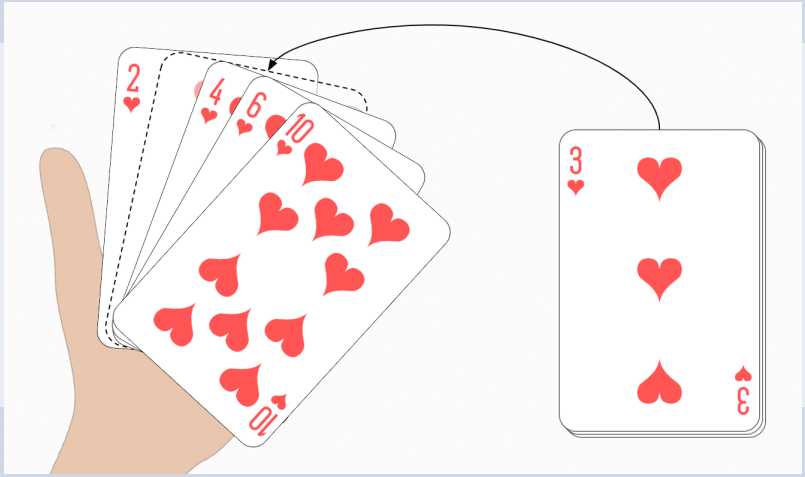
comparisons	swaps
always: (n-1) + (n-2) + ... + 1 = (n-1)n/2 = 6*7/2= 21	always: n-1 = 6

Review: Insertion Sort for A[0..n-1]

	Insertion Sort
Basic idea:	1. Treat the first element as a sorted subarray of size 1, with the rest as the unsorted part. 2. Take the next element from the unsorted part and insert it into the sorted subarray at the correct position, increasing the sorted subarray size by 1. 3. Repeat step 2 until no elements remain in the unsorted part.

Skeleton Code	for (i=1; i<n; i++) {
	// sorted subarray: A[0..i-1]
	// unsorted part: A[i..n-1]
	insert A[i] to the *sorted* A[0..i-1]
	so that to keep A[0..i] sorted
	}

Complexity	
In-place?	
Stable? Why?	



Class Exercise: Trace the selection sort on the input array:

[5 9 4 5 1 2 7]

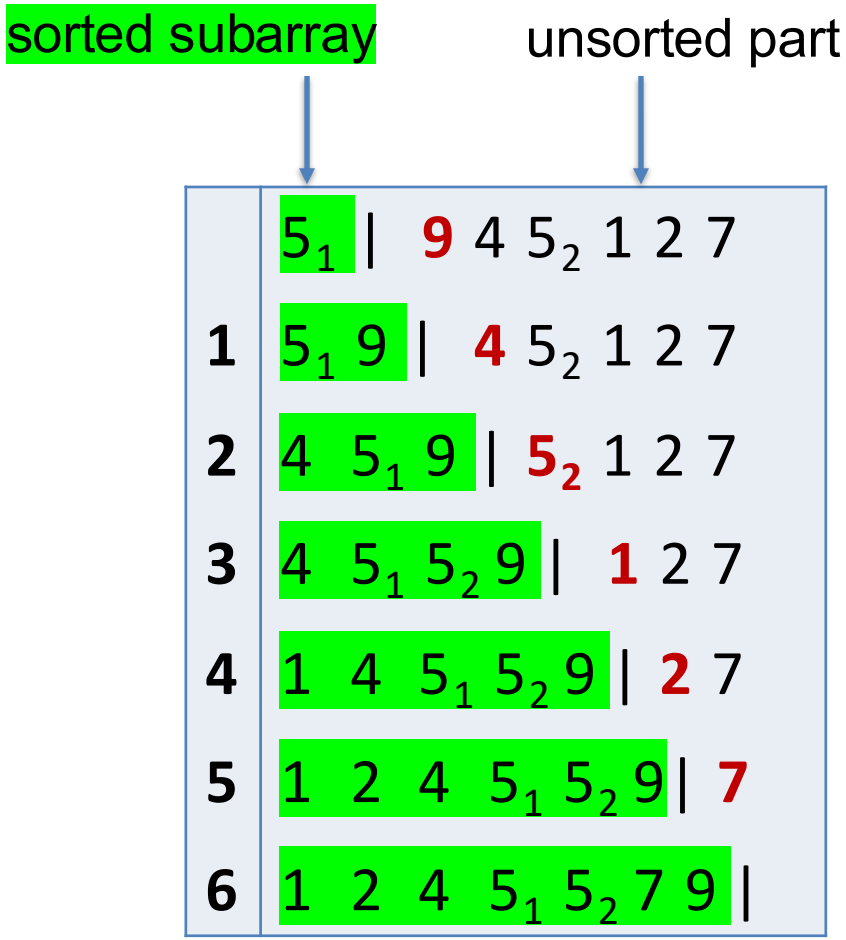
Class Exercise

Trace the insertion sort on the input array:

5 9 4 5 1 2 7

How many key comparisons?
How many swaps?

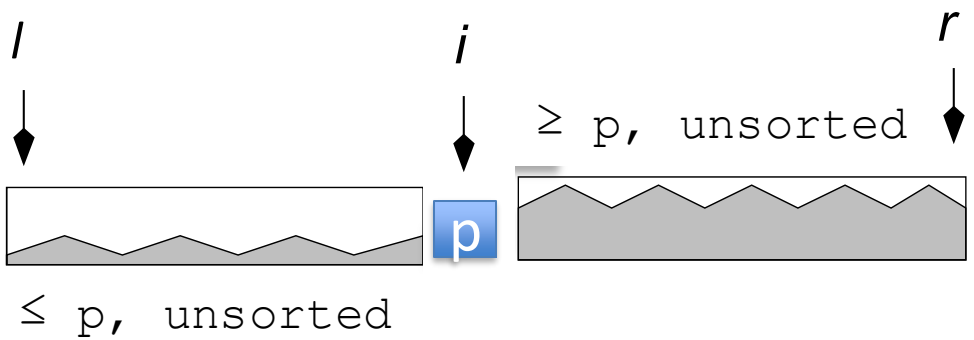
Check your answer: Insertion Sort



comparisons	swaps or shifts
1	0
2	2
2	1
4	4
5	4
2	1
total: 16	total: 12

Quicksort: recursive, usage: Quicksort(A, 0, n-1)

```
int partition(item A[],int l, int r);

void Quicksort(item A[], int l, int r) {
      if (r <= l) return;
      int i =     partition(A,l,r);
       Quicksort(A,l,i-1);
       Quicksort(A,i+1,r);
}
```



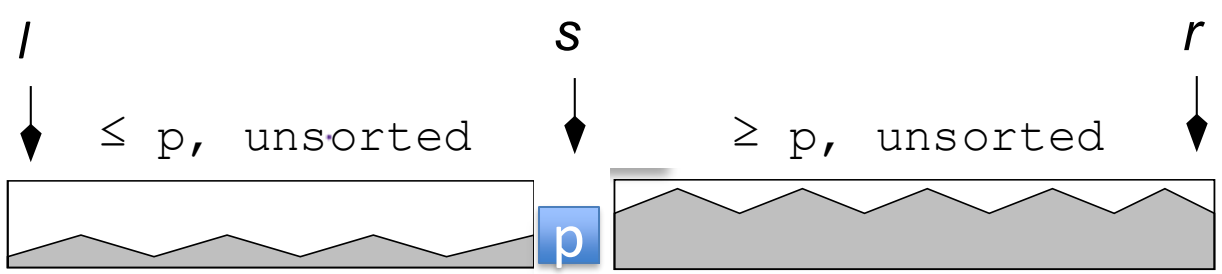
p: the ***pivot*** of this partitioning

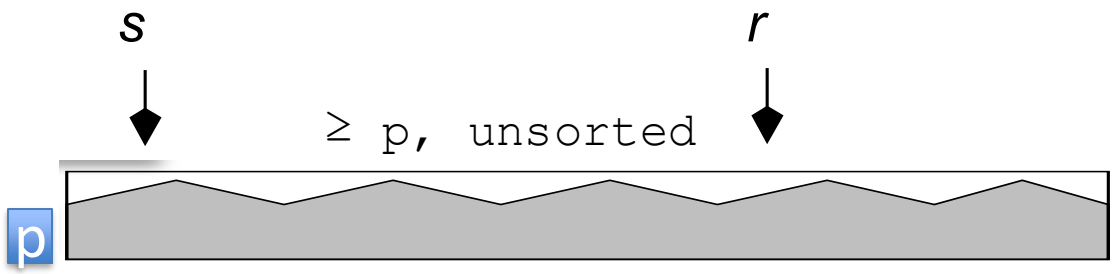
Note: a Partition can be done in-place in $\theta(n)$ time using long-distance swaps

Questions:

- Is it stable?
- Is it in-place
- What is the (additional) space complexity of Quicksort?
- What is the time complexity?

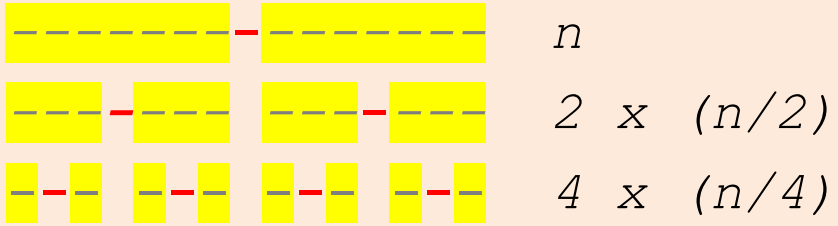
Quicksort Properties - Complexity depends on how the 2 partitions "balanced"



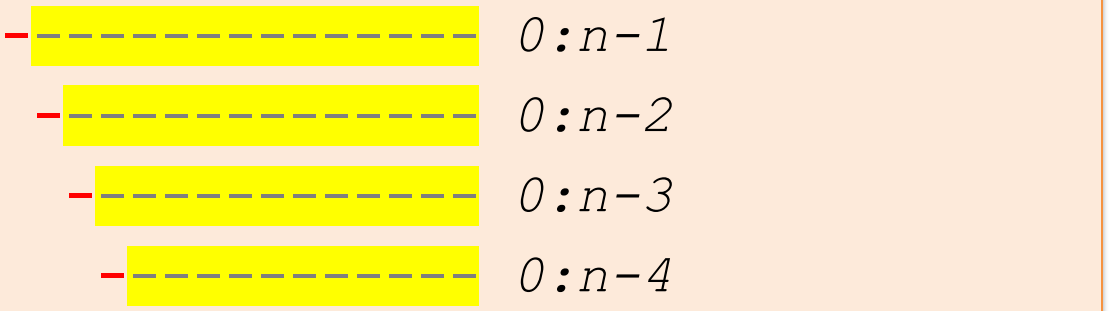


Quicksort complexity depends on the relative lengths of the left and the right parts in partitioning.

BEST: always balanced: $\theta(n \log n)$



WORST: one half always empty: $\theta(n^2)$



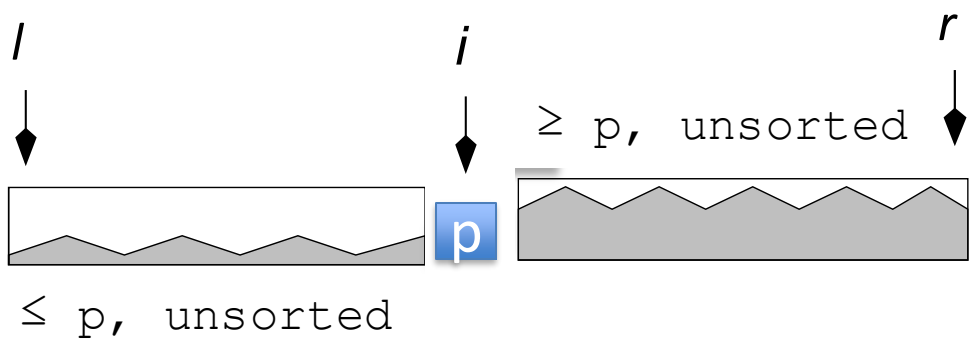
...

Time and (additional) space complexity of Quicksort?

BEST/**AVERAGE**: O(nlog n) WORST/GENERAL: $O(n^2)$

Quicksort: How to perform partitioning?

```
int partition(item A[],int l, int r);
```



Q: How-To: Partitioning?

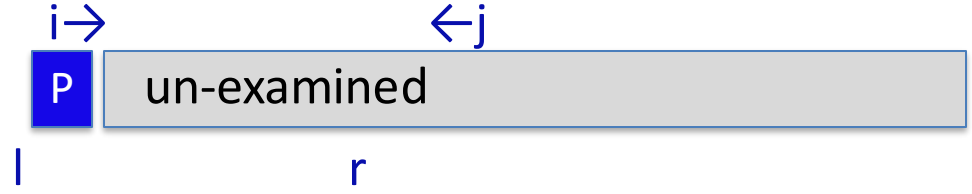
First, how to choose pivot p ?

- *Choose* p= *any element in* A*,*
- *here we always choose* p= A[l] *(the leftmost),*
- Note: if we want A[?] *to be the pivot, we just need to swap it with* A[l]

Start: Choose P= A[l] as Pivot

set i= l, j= r+1

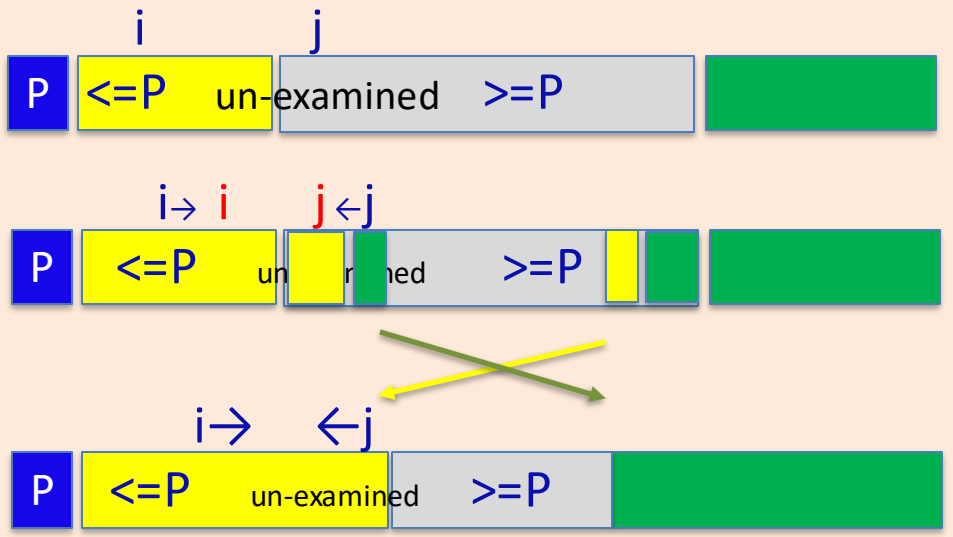
loop



start
(loop invariant)

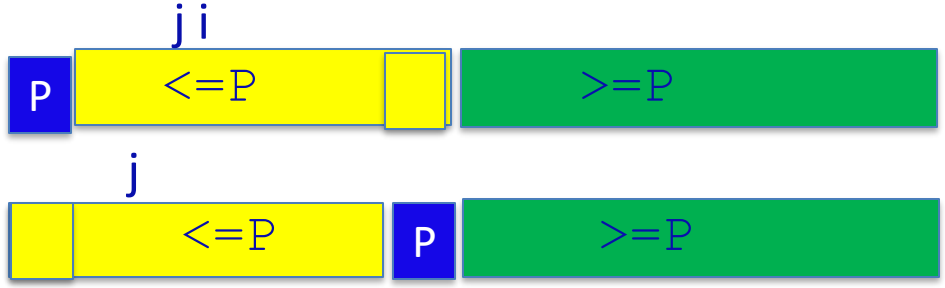
move i→, ←j

swap

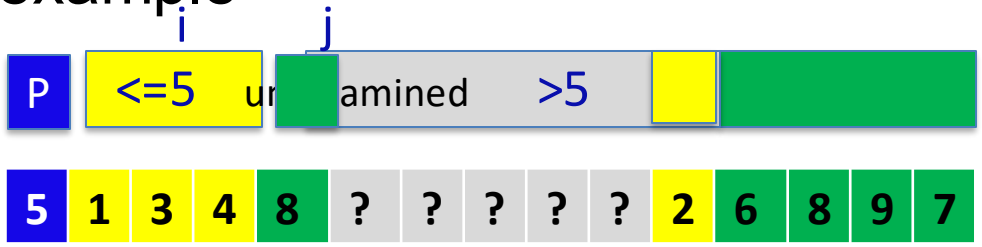


at loop exit: swap P and A[j]

End: returns index j

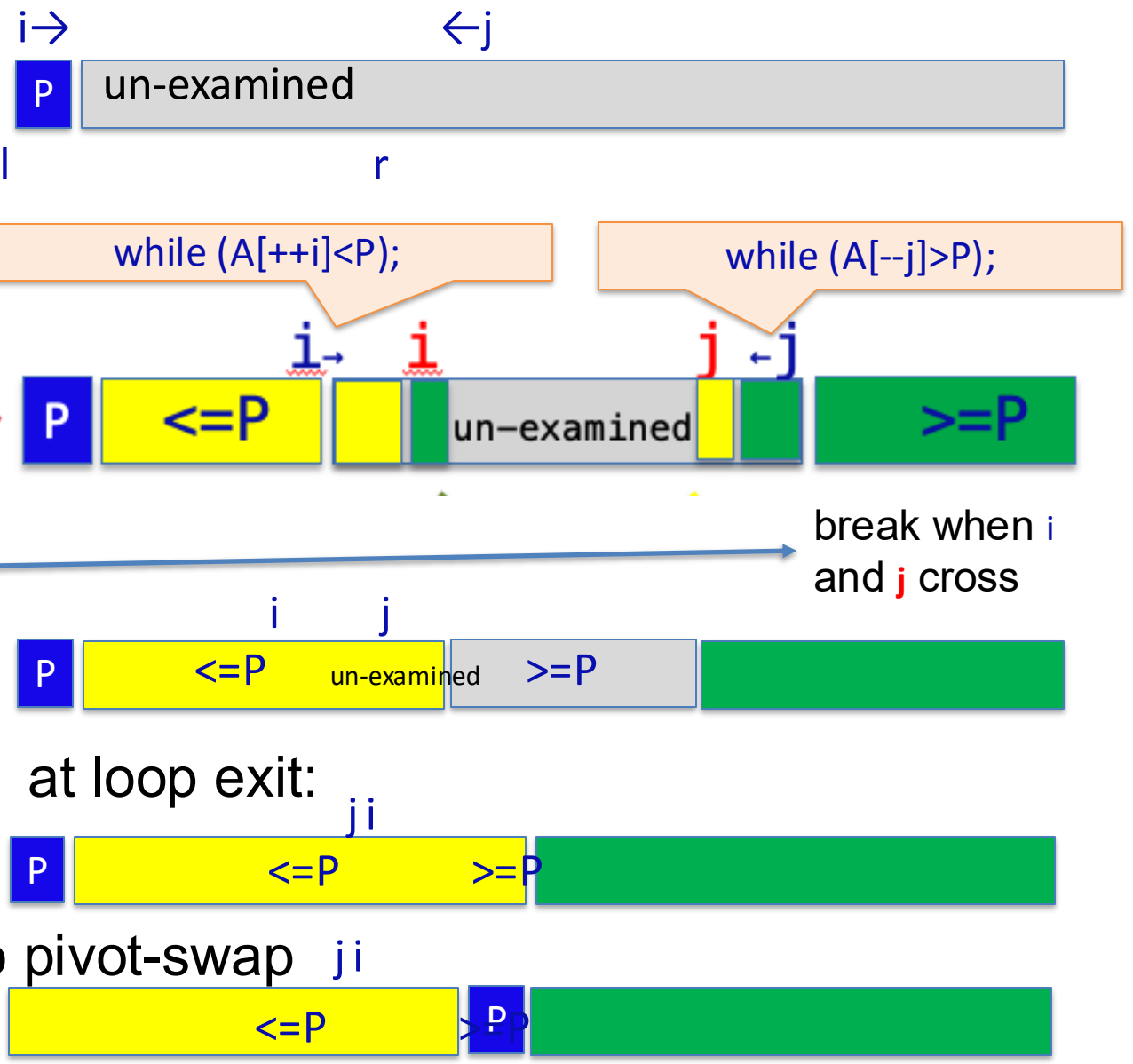


example



Partitioning (Pivot= leftmost element as pivot)

```
P = A[l];
int i = l, j = r + 1;
while (1) {
    // Move  i forward,     stop when A[i]>=P
    while (A[++i]<P);
    // Move  j  backward, stop when A[j]<=P
    while (A[--j]>v);
 if (i>=j) break;
    // swap and continue loop only if  i<j
    Swap(&A[i], &A[j]);
}
Swap(&A[l], &A[j]);
return j;
```



Example: do partitioning for

[5 1 4 2 3]

[5 9 7 6 8]

[5 9 4 5 1 2 7]

Colour Code: ≤ P == P ≥ P unknown

Ex W7.4 run qsort for [5, 1, 8, 3, 4, 6, 8, 1]; then W7.6, W7.7

Initial array: `[ 5 1 8 3 4 6 8 1]`

W7.6: solving the k-smallest problem using partitioning, and its efficiency

W7.7: implement partitioning (only)

Ex W7.4 Check Your Answer - run qsort for [5, 1, 8, 3, 4, 6, 8, 1]

Partitioning 1

swap (8,1)

pivot-swap(4,5)

[5 1 8*i* 3 4 6 8 1*j*]

5 1 1 3 4*j* 6*i* 8 8

[4 1 1 3] 5 [6 8 8]

Partitioning 2

pivot-swap(4,3)

[4 1 1 3*j* *i*]

[3 1 1] 4 []

Partitioning 3

pivot-swap(3,1)

[3 1 1*j* *i*]

[1 1] 3 []

Partitioning 4

pivot-swap(1,1)

[1 1*ij*]

[1] 1 []

Partitioning 5

pivot-swap(6,6)

[6*j* 8*i* 8]

[] 6 [8 8]

Partitioning 6

pivot-swap(8,8)

[8 8*ij*]

[8] 8 []

NOTES

- *i* marks the position of the index-from-left
- *j* marks the positiion of the index-from-right

Review (more detailed steps):

	Selection Sort	Insertion Sort	Quick Sort
Basic Steps:	1. Treat the whole array as the unsorted part 2. Scan the unsorted part to find the smallest element and swap it into the first position. 3. The first element is now in its correct sorted place; treat the rest as the unsorted part. 4. Repeat step 2-3 until the unsorted part has only one element left.	1. Treat the first element as a sorted subarray of size 1, with the rest as the unsorted part. 2. Take the next element from the unsorted part and insert it into the sorted subarray at the correct position, increasing the sorted subarray size by 1. 3. Repeat step 2 until no elements remain in the unsorted part.	1. Choose a pivot element and partition the array so that elements smaller than the pivot go to its left and larger ones go to its right. The pivot is now in its final sorted position. 2. Recursively apply the same process to the left and right subarrays until all subarrays have size 0 or 1, at which point the whole array is sorted.

Skeleton Code	for (i=0; i<n-1; i++) {
	// correct part: A[0..i-1]
	// remaining part: A[i..n-1]
	imin= index of minimal of A[i..n-1]
	swap(&A[i], &A[imin]);
	}
		for (i=1; i<n; i++) {
		// sorted subarray: A[0..i-1]
		// unsorted part: A[i..n-1]
		insert A[i] to the *sorted* A[0..i-1]
		so that to keep A[0..i] sorted
		}
			Quicksort(A[], l, r) {
			if (r <= l) return;
			i = partition(A,l,r);
			Quicksort(A,l,i-1);
			Quicksort(A,i+1,r);
			}

Complexity	$\Theta(n^2)$	$O(n^2)$ **best: $\Theta(n)$** average $\Theta(n^2)$	$O(n^2)$ **best: $\Theta(n \log n)$** **average $\Theta(n \log n)$**
In-place?	✓	✓	✓
Stable?	×	**stable ✓**	×

Sorting algorithms

	Insertion Sort	Selection Sort	Quick Sort	Merge
Basic Idea	Builds the sorted array one element at a time. Each new element is taken from the unsorted part and inserted into the correct position within the already sorted part.	Repeatedly finds the smallest element in the unsorted part and swaps it with the first unsorted position. After each step, one more element is placed in its final sorted position.	Divides the array around a pivot, placing smaller elements on the left and larger on the right, then recursively sorts the subarrays until the whole array is ordered.	Split …
Complexity	$O(n^2)$	$\theta(n^2)$	$O(n^2)$	
Best case	$\theta(n)$	$\theta(n^2)$	$\theta(n \log n)$	
Worst case	$\theta(n^2)$	$\theta(n^2)$	$\theta(n^2)$	
Average	$\theta(n^2)$	$\theta(n^2)$	$\theta(n \log n)$	
In-place?	✓	✓	✓	
Stable?	✓ **(swap adjacent)**	✗ (long-distance swap)	✗ (long-distance swap)	

2b|!2b : assignment or MST ?

Quickly get ✔ for this week:

- W7.1: for peer activity
- W7.6: solving the k-smallest problem using partitioning
- W7.7: super short coding exercise: implement partitioning
- other exercises

Do your assignment now?

or Work through the sample MST and give questions?

Notes: review, **bring the MST questions to the next workshop**

Additional Slides

Note: Recursive algorithms involves additional space for stack frames

- A recursive algorithms need extra time and memory for the recursive call.
- That additional time/space grows linearly with the recursion depth.

```
int binSearch(int A[], int n, int x) {
    int lo= 0, hi= n-1;
    while (lo <= hi) {
    int m = (lo + hi)/2;
    if (x < A[m])
        hi = m - 1;
    else if (x > A[m])
        lo = m + 1;
    else
        return m;
    }
    return NOTFOUND
}
```

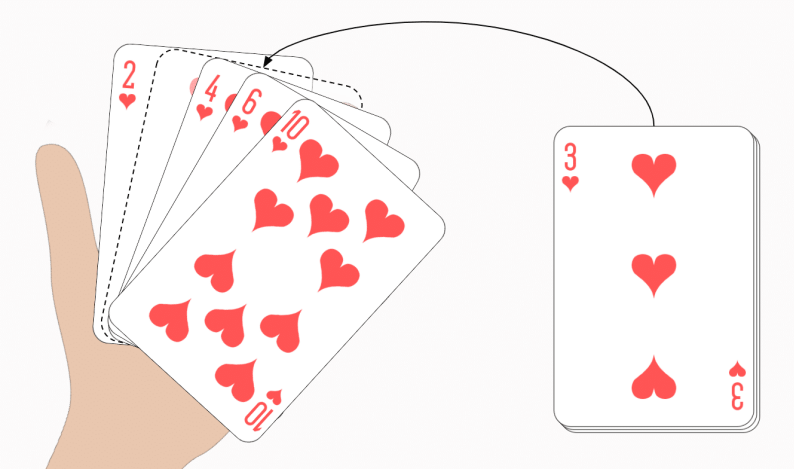
```
// recursive bin search for A[n]
// first call: recBS(A,0,n-1,x)

int recBS(int A[], int lo, int hi, int x) {
    if (lo > hi) return NOTFOUND;
    int m = (lo + hi)/2;
    if (x < A[m])
    return recBS(A, lo, m-1);
    else if (x > A[m])
        return recBS(A, m+1, hi);
    else
        return m;
}
```

```
memory use (in addition to the array):
O(1)
```

```
max recursive depth: log2n
additional memory:   O(log n)
```

Insertion Sort: In each round: Examine a single next element & Insert it to the examined area, keeping the examined area in sorted order

1. Treat the whole array as the unsorted part
2. Scan the unsorted part to find the smallest element and swap it into the first position.
3. The first element is now in its correct sorted place; treat the rest as the unsorted part.
4. Repeat step 2-3 until the unsorted part has only one element left.



Input Array		6 10 2 4 3 7 9
Round	Next Element	after inserting the next element into the **sorted subarray**
<start>		**6** 10 2 4 3 7 9
1	10	**6 10** 2 4 3 7 9
2	2	**2 6 10** 4 3 7 9
		(insert 2 by first shifting 10, then 6 to the right)
3	4	
4	3	
5	7	
6	9	