COMP20003 Workshop Week 7

Sorting Algorithms + MST

Sorting & Properties: Insertion Sort and Selection Sort
Quicksort

MST = ?
Ass2 Q&A

MST in Friday Week 8
Questions: today or in the Week 8 Workshop

Note: Recursive algorithms involves additional time & space

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

Examples:

- Recursive function for n! has depth n
  - extra θ(n) time for function calls, and
  - extra θ(n) space for all the stack frames
- Recursive Binary Search (in sorted arrays) has depth $\log_2(n)$ in the worst case:
  - extra O(log n) time
  - extra O(log n) space

Sorting Algorithms

Remember well:

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

Properties of sorting algorithms:

- *in-place: not using additional arrays to store data/keys (**not counting** additional cost for recursion)*
- *stable: maintaining the relative order of (data with) equal keys*

Review: Sorting algorithms for A[0..n-1]

	Insertion Sort	Selection Sort
Basic idea	Start: • examined area: A[0..0] (sorted) • remaining area: A[1..n-1] Examine the next element and insert it to the examined area so that the latter is sorted.	Start: • completed area: A[0..-1] (empty) • remaining area: A[0..n-1] Traverse the remaining area to find the smallest, then swap it to the right position and join it to the completed area.

Skeleton Code	`for (i=1; i<n; i++) {`
	insert A[i] to the *sorted* A[0..i-1]
	so that to keep A[0..i] sorted
	`}`
		`for (i=0; i<n-1; i++) {`
		`imin=` index-of-a-minimal element of A[i..n-1]
		`swap( &A[i], &A[imin]);`
		`}`

Input-sensitive?		
In-place?		
Stable?		

Class Exercise: Manually run the 2 algorithms on the input array:

[5 9 4 5 1 2 7]

Check your answer: Insertion Sort Selection Sort

Insertion Sort

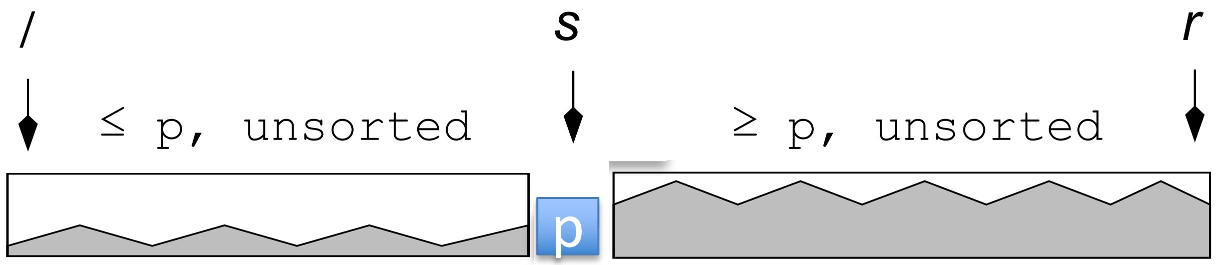
	5_1 | **9** 4 5_2 1 2 7
1	5_1 9 | **4** 5_2 1 2 7
2	4 5_1 9 | $\mathbf{5_2}$ 1 2 7
3	4 5_1 5_2 9 | **1** 2 7
4	1 4 5_1 5_2 9 | **2** 7
5	1 2 4 5_1 5_2 9 | **7**
6	1 2 4 5_1 5_2 7 9 |

Selection Sort

	| 5_1 9 4 5_2 **1** 2 7
1	1 | 9 4 5_2 5_1 **2** 7
2	1 2 | **4** 5_2 5_1 9 7
3	1 2 4 | $\mathbf{5_2}$ 5_1 9 7
4	1 2 4 5_2 | 5_1 9 7
5	1 2 4 5_2 5_1 | 9 **7**
6	1 2 4 5_2 5_1 7 | 9

Quicksort idea (recursive, usage: `Quicksort(A[0..n-1])`)

function QUICKSORT($A[l..r]$)
  if $l < r$ **then**
    $s \leftarrow$ PARTITION($A[l..r]$)
    QUICKSORT($A[l..s-1]$)
    QUICKSORT($A[s+1..r]$)



p = A[s] is called the ***pivot*** of this partitioning

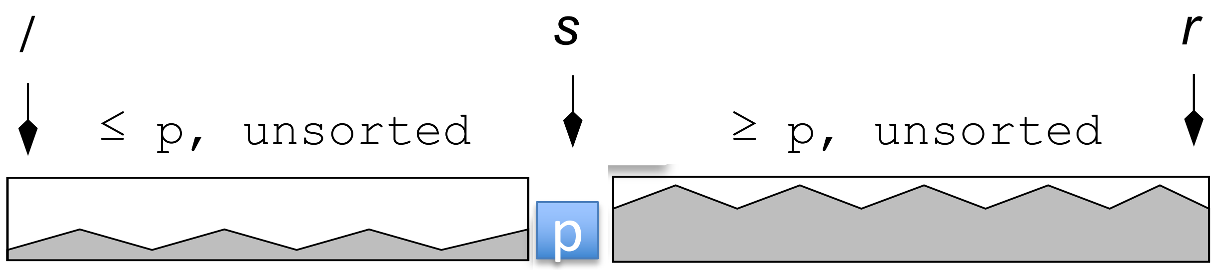
Note: a **PARTITION** of n elements has the complexity of $\theta(n)$

Questions:

- What is the (additional) space complexity of **QUICKSORT**?
- What is the time complexity?
- Is it input-sensitive?
- Is it in-place?
- Is it stable?

Quicksort Properties - Check your answers

function QUICKSORT($A[l..r]$)
    if $l < r$ **then**
        $s \leftarrow$ PARTITION($A[l..r]$)
        QUICKSORT($A[l..s-1]$)



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

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

```
--------------------    n
-------  --------     2 x (n/2)
---  ---  ---  ---     4 x (n/4)
-  -  -  -  -  -  -  -
```

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

```
-----------------  0:n
 ----------------  0:n-1
  ---------------  0:n-2
   --------------  0:n-3
        ...
```

***AVERAGE:* $\theta(n \log n)$**

What is the (additional) space complexity of QUICKSORT?
BEST/AVERAGE O(log n)   WORST O(n)

- Is it input-sensitive? Y
- Is it in-place? Y
- Is it stable? N – but we need to check with Partitioning

Quicksort (recursive, usage: `Quicksort(A[0..n-1])`)

function $\textsc{Quicksort}(A[l..r])$
  if $l < r$ **then**
    $s \leftarrow \textsc{Partition}(A[l..r])$
    $\textsc{Quicksort}(A[l..s-1])$
    $\textsc{Quicksort}(A[s+1..r])$

p = A[s] is called the *pivot* of the partitioning

Q: How to do the Partitioning?

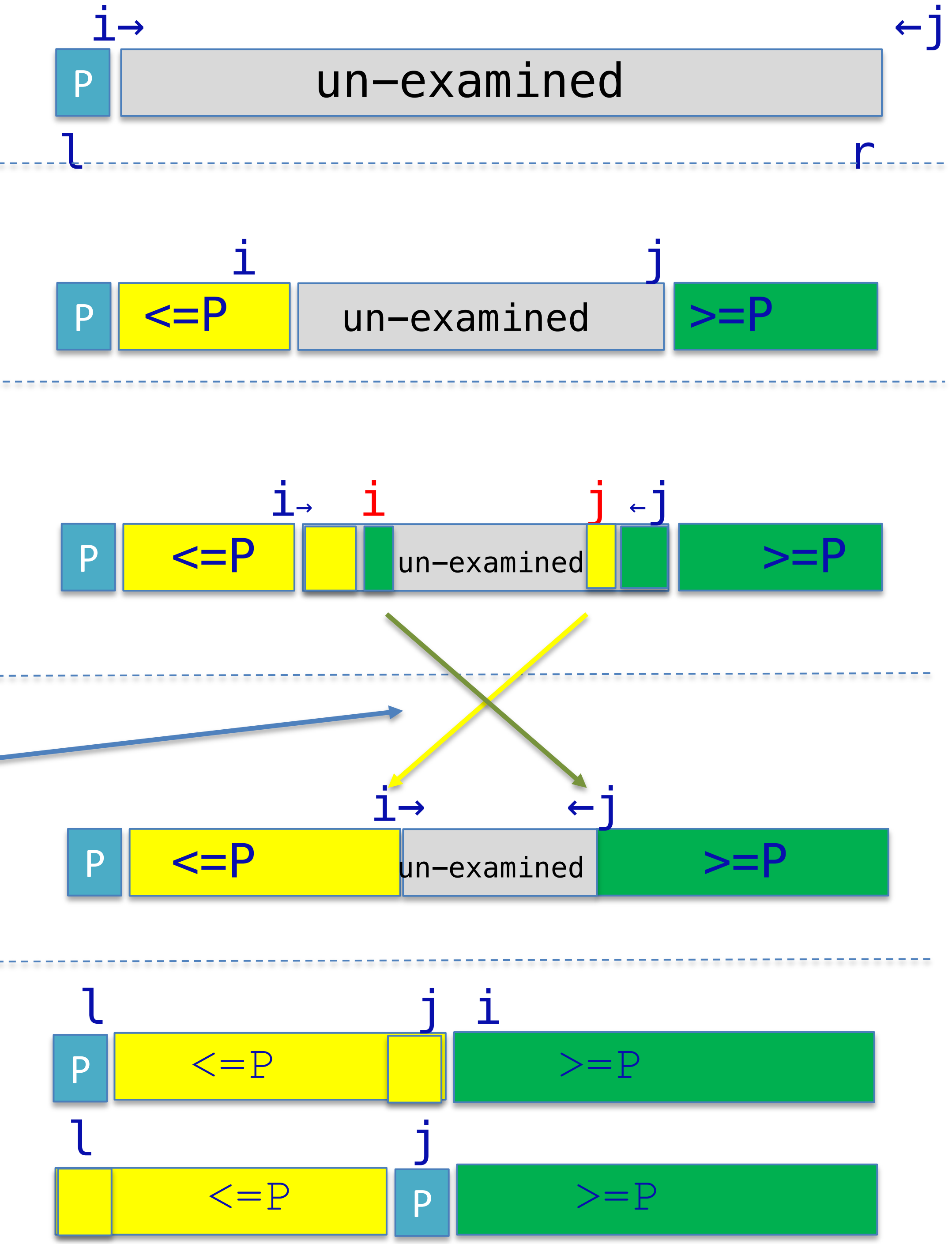
First, how to choose pivot p ?

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

```
int HoarePartition(item A[], int l, int r) {
int i = l, j = r + 1; // i from left, j from right
P = A[l];
// loop invariant:
l                  i                                j          r
A[l+1..i]≤P A[i+1..j-1] un-examined A[j..r]≥P

while (1) {
   # move i forward until A[i]>=P
   while (A[++i]<P);
   # move j backward until A[j]<=P
   while (A[--j]>v);
   if (i>=j) break;  // exit loop if i and j crossed
   # extend yellow and green area
   #   at the same time by swapping
   Swap(&A[i], &A[j]);
}

# at loop's exit: i and j crossed
Swap(&A[l], &A[j]);
return j;
}
```



Partitioning (using the leftmost element as pivot) – do together

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

Example: do partitioning for [5 9 4 5 1 2 7]

start 5 9 4 5 1 2 7

Partitioning: Check your answer

start	5	9	4	5	1	2	7	(represents i, reprsesents j)
move:	5	9	4	5	1	2	7	
swap:	5	2	4	5	1	9	7	
move:	5	2	4	5	1	9	7	
swap:	5	2	4	1	5	9	7	
move:	5	2	4	1	5	9	7	(i and j crossed, stop!)

final_swap: [1 2 4] 5 [5 9 7] return j;

Your Task: Finish the quicksort algorithm

Initial array : [(5) 9 4 5 1 2 7] (5) is pivot

after 1st partition: [1 2 4] 5 [5 9 7]

→

quicksort: Check your answer

```
Initial array:      [ (5)  9  4  5  1  2  7 ]
after 1st partition:   [1  2  4]  5  [5  9  7]
                 [(1)  2  4]  5   [(5)  9  7]
                    1  [2  4]  5   5   [9  7]
               1   [(2)  4]  5   5   [(9)  7]
                 1    2  [4]  5   5   [7]  9
                      1  2  4  5  7  9
```

Discussion:

- What's the complexity of partitioning on n elements? $\theta(n)$
- What if I want to use some other element as pivot, say A[k]?
  - → just swap A[k] and A[l], then continue the algorithm as it is
- What is a good choice for k (ie. index of pivot A[k])?
  - → The Lord of Randomness has a great power!

Sorting algorithms

	Insertion Sort	Selection Sort	Quick Sort	Merge
Basic Idea	Remove the next element and insert it to the examined area so that the latter is sorted.	Traverse the remaining area to find the smallest, then swap it to the right position and exclude it from the remaining area	Choose a pivot, partition array into a *lesser* and a *greater* (than pivot) halves. Do recursively with each half.	Split …
Complexity	$O(n^2)$	$\theta(n^2)$	$O(n^2)$	
Best case	$\theta(n)$		$O(n \log n)$	
Worst case	$\theta(n^2)$		$\theta(n^2)$	
Average	$O(n^2)$	$O(n^2)$	$O(n \log n)$	
In-place?	✓	✓	✓	
Stable?	✓	✗	✗	

2b|!2b : assignment or MST ?

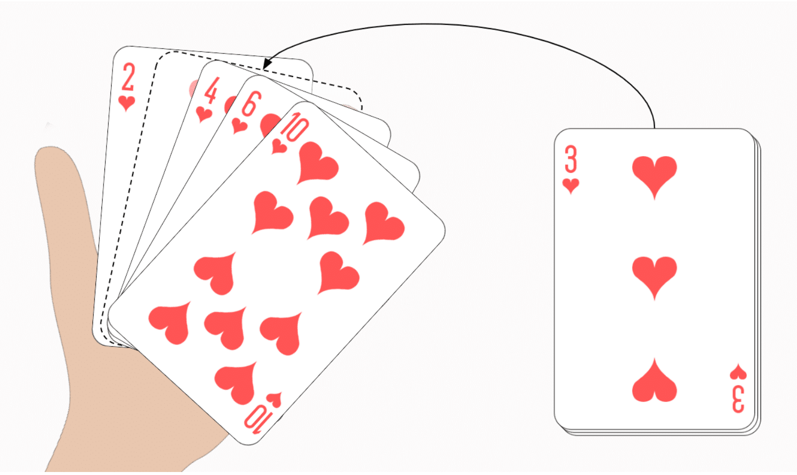
Do your assignment now?

or Work through the sample MST and give questions?

[you can bring the MST questions to the next workshop]

Additional Slides

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



Input Array		6 10 2 4 3_1 7 3_2
Round	Next Element	Array, after inserting the next element into the **examined part**
<start>	6	**6** 10 2 4 3_1 7 3_2
1	10	**6 10** 2 4 3_1 7 3_2
2	2	**2 6 10** 4 3_1 7 3_2
3	4	(insert 2 by first shifting 10, then 10 to the right)
4	3_1	
5	7	
6	3_2	