# COMP20003 Workshop Week 7

| | |
|---|---|
| **1** | Sorting, Insertion Sort and Selection Sort revisited |
| **2** | Quicksort + Q 6.1 |
| **3** | Review for the Test / sample test |
| **4** | Implementing Hash Table (P6.1) |
| | MST:<br>Week 8 |

# Sorting Algorithms

Any problem with:

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

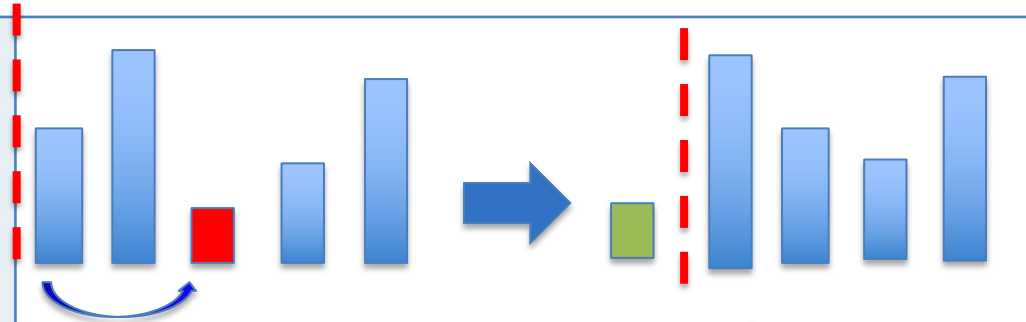
Properties of sorting algorithms:

- *in-place*
- *stable*

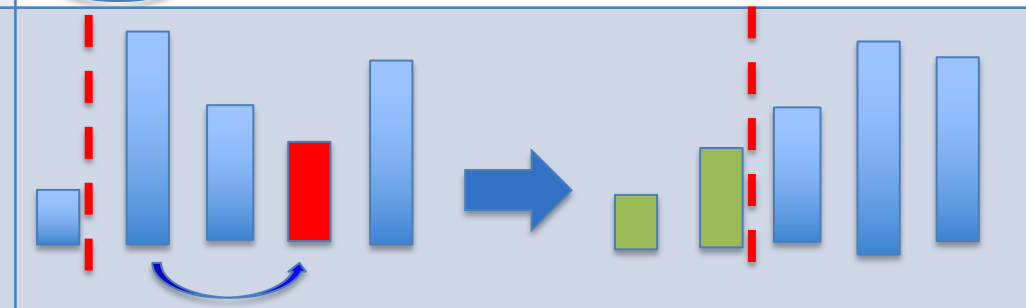
***Note:*** *In this workshop, suppose we need to sort an array in non-decreasing order.*

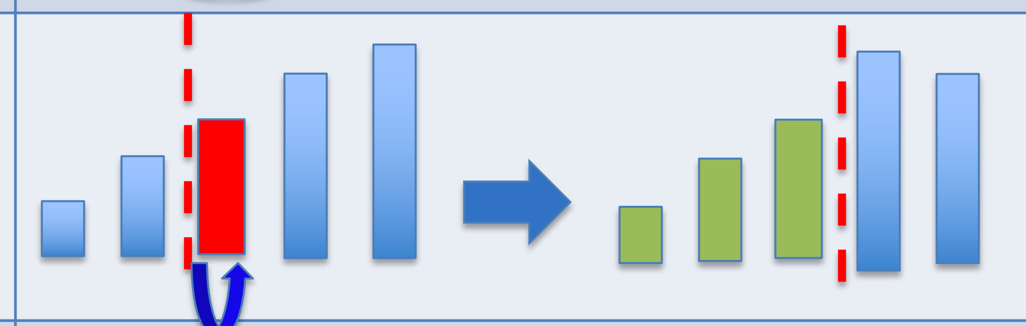
# Selection Sort: `n=5, (by selecting the smallest)`

Round 1: consider `A[0..4]`
- *determine position of the smallest*
- *swap with the first, ie.* `A[0]`



Round 2: consider `A[1..4]`
- *determine position of the smallest*
- *swap with the first, ie.* `A[1]`





Round 4: consider `A[3..4]`
- *determine position of the smallest*
- *swap with the first, ie.* `A[3]`

# Selection Sort

| | |
|---|---|
| - | A N A L Y S I S |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |

i. Run the algorithm on the input array: [A N A L Y S I S]
ii. What is the time complexity of the algorithm?
iii. Is the sorting algorithm stable?
iv. Is the algorithm in-place?

# Selection Sort

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **-** | | **A** | N | A | L | Y | S | I | S |
| **1** | A | | **N** | **A** | L | Y | S | I | S |
| **2** | A | A | | **N** | L | Y | S | **I** | S |
| **3** | A | A | I | | **L** | Y | S | N | S |
| **4** | A | A | I | L | | **Y** | S | **N** | S |
| **5** | A | A | I | L | N | | **S** | Y | S |
| **6** | A | A | I | L | N | S | | **Y** | **S** |
| **7** | A | A | I | L | N | S | S | | Y |

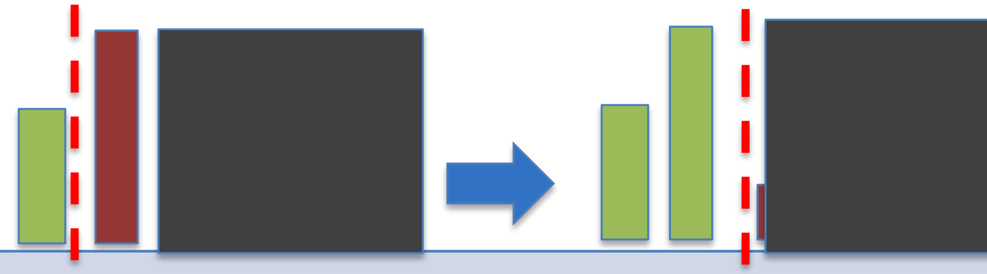
i. Run the algorithm on the input array: `[A N A L Y S I S]`

ii. What is the time complexity of the algorithm?

iii. Is the sorting algorithm stable?

iv. Is the algorithm in-place?

# Insertion Sort: understanding (`n=5`)

| | |
|---|---|
| Round 1: consider `A[1]`<br>• `A[0..0]` *is sorted*<br>• *insert* `A[1]` *to the left so that* `A[0..1]` *is sorted* |  |
| Round 2: consider `A[2]`<br>• `A[0..1]` *is sorted*<br>• *insert* `A[2]` *to the left so that* `A[0..2]` *is sorted* |  |
| |  |
| Round i: consider `A[4]`<br>• `A[0..4]` *is sorted*<br>• *insert* `A[i]` *to the left so that* `A[0..4]` *is sorted* |  |

# Insertion Sort

| | A N A L Y S I S |
|---|---|
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |

i. Run the algorithm on the input array: [A N A L Y S I S]

ii. What is the time complexity of the algorithm?

iii. Is the sorting algorithm stable?

iv. Does the algorithm sort in-place?

# Insertion Sort

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | A | | N | A | L | Y | S | I | S |
| **1** | A | N | | A | L | Y | S | I | S |
| **2** | A | A | N | | L | Y | S | I | S |
| **3** | A | A | L | N | | Y | S | I | S |
| **4** | A | A | L | N | Y | | S | I | S |
| **5** | A | A | L | N | S | Y | | I | S |
| **6** | A | A | I | L | N | S | Y | | S |
| **7** | A | A | I | L | N | S | S | Y | |

i. Run the algorithm on the input array: `[A N A L Y S I S]`

ii. What is the time complexity of the algorithm?

iii. Is the sorting algorithm stable?

iv. Does the algorithm sort in-place?

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

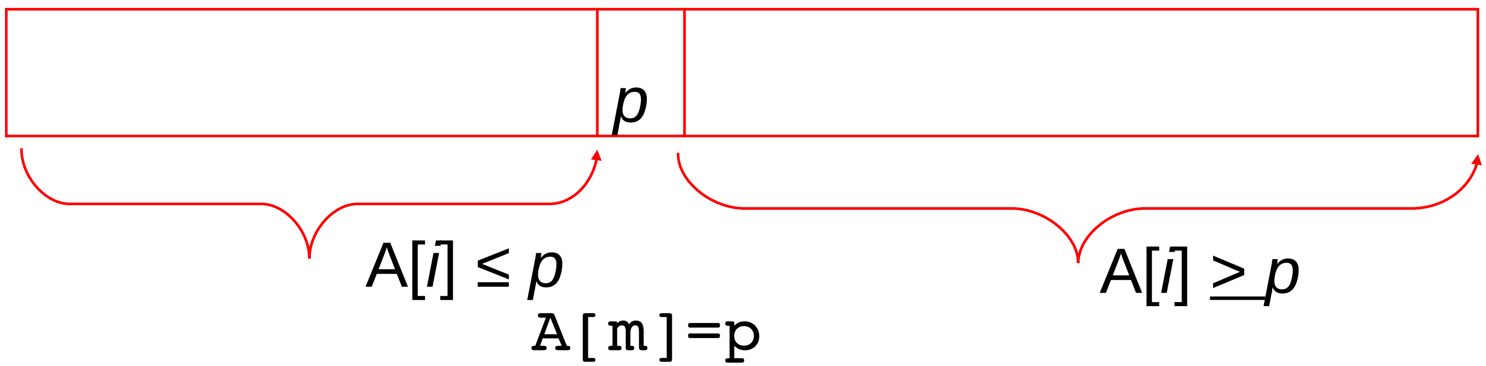
**function** $\text{Quicksort}(A[l..r])$
    **if** $l < r$ **then**
        $s \leftarrow \text{Partition}(A[l..r])$
        $\text{Quicksort}(A[l..s-1])$
        $\text{Quicksort}(A[s+1..r])$

```
Partition(A[l..r])
      <=A[s]         >=A[s]

    A[l..s-1   s   s+1..r   ]

   return s
```

# Partitioning (using the rightmost element as pivot)

**Input:** Given an unsorted slice `A[l..r]`,

**Output:** A value `m` and re-arrangement of `A` so that:




Algorithm:

1. set `p= A[r] (=rightmost),` keep `A[r]` un`changed
2. Start with `i= l`, `j=r-1`
3. Moving `i` forward, stop when `A[i]>=p`
4. Moving `j` backward, stop when `A[j]<=p`
5. If `i<j: swap A[i], A[j]` then go back to step 3
6. `Swap A[r] and A[i]`, return `m= i`

# Partitioning (using the rightmost element as pivot)

1. set `p= A[r] (=rightmost),` keep `A[r]` un`changed
2. Start with `i= l`, `j=r-1`
3. Moving `i` forward, stop when `A[i]>=p`
4. Moving `j` backward, stop when `A[j]<=p`
5. If `i<j: swap A[i], A[j]` then go back to step 3
6. `Swap A[r] and A[i]`, return `m= i`

Example

start 1 2 3 4 5 6 **7**

moved 1 2 3 4 5 6 **7**

**i>j: no swap**

**Step 6:** 1 2 3 4 5 6 **7** **(swap 7 with 7)**

[1 2 3 4 5 6] **7** [ ]

# Partition ivot)

1. set `p= A[r] (=rightmost),` keep `A[r]` un`changed
2. Start with `i= l`, `j=r-1`
3. Moving `i` forward, stop when `A[i]>=p`
4. Moving `j` backward, stop when `A[j]<=p`
5. If `i<j: swap A[i], A[j]` then go back to step 3
6. `Swap A[r] and A[i]`, return `m= i`

## Example

start A N A L Y S **N**

moved A N A L Y S **N**

**i<j: swap** A L A N Y S **N**

**moved:** A L A N Y S **N**

**i>=j: stops looping**

**Step 6: swap N with N, return m= i= 3**

A L A N Y S **N**

**answer=3, [left]m[right]= [A L A] N [Y S N]**

# Q 6.1

You are asked to show the operation of quicksort on the following keys. For simplicity, use the rightmost element as the partition element:

2 3 97 23 15 21 4 23 29 37 5 23

Comment on the stability of quicksort and its behavior on almost sorted inputs.

# Quiz 1

The best case of Selection sort is:

*a. O(n log n).*

*b. O(n).*

*c.* $O(n^2)$.

*d. O(log n).*

*When?*

# Quiz 2

The best case of Insertion Sort is:

*a. O(n log n).*

*b. O(n).*

*c.* $O(n^2)$*.*

*d. O(log n).*

*When?*

# Quiz 3

The average case of Insertion Sort is:

*a. $O(n \log n)$.*

*b. $O(n)$.*

*c. $O(n^2)$.*

*d. $O(\log n)$.*

# Quiz 4

The average case of Quick Sort is:

*a. $O(n \log n)$.*

*b. $O(n)$.*

*c. $O(n^2)$.*

*d. $O(\log n)$.*

# Quiz 5

The big-O complexity of Quick Sort is:

*a. $O(n \log n)$.*

*b. $O(n)$.*

*c. $O(n^2)$.*

*d. $O(\log n)$.*

# Sorting algorithms: complexity; being stable, in-place

| | **Selection** | **Insertion** | **Quick** | **Merge (top-down)** |
|---|---|---|---|---|
| Basic Idea | For A[0..n-1] Identify the smallest and swap it with A[0]. Repeat for A[1..n-1] and so on. | From $2^{nd}$ element, insert it to the left sub-array so that the extended left sub-array remains sorted. | Choose a pivot, partition array into a *lesser* and a *greater* (than pivot) halves. Do recursively with each half. | Split to equal-size halves, sort them then merge them. |
| Complexity | | | | |
| Best case | | | | |
| Worst case | | | | |
| Average | | | | |
| In-place? | | | | |
| Stable? | | | | |

# Sorting algorithms: complexity; being stable, in-place

| | Selection | Insertion | Quick | Merge (top-down) |
|---|---|---|---|---|
| Basic Idea | For A[0..n-1] Identify the smallest and swap it with A[0]. Repeat for A[1..n-1] and so on. | From $2^{nd}$ element, insert it to the left sub-array so that the extended left sub-array remains sorted. | Choose a pivot, partition array into a *lesser* and a *greater* (than pivot) halves. Do recursively with each half. | Split to equal-size halves, sort them then merge them. |
| Complexity | $\Theta(n^2)$ | $O(n^2)$ | $O(n^2)$ | |
| Best case | $O(n^2)$ | $O(n)$ | $O(n \log n)$ | |
| Worst case | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | |
| Average | $O(n^2)$ | $O(n^2)$ | $O(n \log n)$ | |
| In-place? | | | | |
| Stable? | | | | |

# Sorting

Concepts: sorting, stable sort, in-place sort
Some sorting algorithm (on an array of $n$ elements):

| | **Selection** | **Insertion** | **Quick** | **Merge** |
|---|---|---|---|---|
| Basic Idea | Identify the smallest and swap it with the first… | From 2nd element, insert it to the left sub-array so that the extended left sub-array remains sorted. | Choose a pivot, partition array into a *lesser* and a *greater* (than pivot) halves… | Split to equal-size halves, sort them then merge them. |
| Best case | $O(n^2)$ | $O(n)$ | $O(n \log n)$ | $O(n \log n)$ |
| Worst case | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(n \log n)$ |
| Average | $O(n^2)$ | $O(n^2)$ | $O(n \log n)$ | $O(n \log n)$ |
| In-place? | | | | |
| Stable? | | | | |

# Assignment 2: Report

A report should

- be concise :  <2 pages including table/graphs
- be clear as instructed in  specs
- have:
  - proper introduction (purpose of the report)
  - expts description (data, queries, outcome)
  - expts outcome (graphs/table)
  - discussion: compare with theory, compare stage 1 and 2
  - short conclusion

# Experiments

What to show? How?

What can affect the performance?

How to (automatically) organize experiments? Create data files? Accumulate the results? Build table/graphs?

?

# Experiments

```
n=   1000   2000   4000   8000   16000 ?
n=    3000 6000 9000 12000 15000 18000 ?
```

Data: sorted, random [no need median]

query: run 100 queries each times and compute average?

Stage 1 vs stage 2?

| stage | data_type | n | comparison/query |
|---|---|---|---|
| 1 | sortx | 1000 | ??? |
| 1 | sortx | 2000 | ??? |
| … | | | |
| 1 | rand | 1000 | ??? |

# Creating data file

1. Create a sorted data file with no header line

```
tail -n +2 CLUEdata2018_sortx.csv  > sortx_all.csv
```

2. Create 2 data files of 1000 lines:

```
head -n 1000 sortx_all.csv > sortx_1000.csv
shuf sortx_1000.csv >  rand_1000.csv
```

Repeat step 2 for all the sizes you want. You can also run the following single-line command:

```
for size in 1000 10000 100000;
do head -n $size sortx_all.csv > sortx_$size.csv;
shuf sortx_$size.csv >  rand_$size.csv; done
```

**Note:** The above 3 parts are in a single line, if you want to break into sub-lines, add a single slash \ to the end of each sub-line.

The data files don't have the header line. It's OK for the experiment purpose.

# Creating query files

Create 2 query files of 100 lines each (q1.txt and q2.txt for Stage 1 and Stage 2) from sortx_all.csv :

```
cat sortx_all.csv
     | awk -F ',' '{if (NF==12) print $9,$10}'
     | shuf | head -n 100
     | awk '{printf("%lf %lf\n", $1, $2)}'
     > q1.txt
```

For Stage 2 queries, change the last 2 components to

```
     | awk '{printf("%lf %lf 0.0005\n", $1, $2)}'
          > q2.txt
```

Note: each command is in a single line, if you want to break into sub-lines, add a single slash \ to the end of each sub-line.

# Summing up the output of you program

Suppose that the output into stdout of Stage 1 are *only* of format:

```
144.959522 -37.800095 --> 4000
```

```
0 0 --> 300
```

And we run, say:

```
./dict1 sortx-1000 o.txt > S1out_sortx_1000.txt
```

Then (**$4** for "column 4", which is the number 4000 and 300 in the top 2 lines):

```
cat S1out_sortx_1000.txt | awk 'BEGIN{sum=0; n=0}{sum = sum+$4;
n++}END{print "1000 " sum/n}'     >> Stage1_sortx.txt
```

will *append* line

```
1000 2150
```

to file `Stage1_sortx.txt`. This line represents:

```
n     average_cmp_per_search
```

**Note:**

- Make sure that your output to stdout is in correct format: `0<SPACE>0<SPACE>--><SPACE>300`
- For Stage 2 output, we need to relace **$4** by **$5** because the output has format where num_cmp is in column 5:

```
144.959522 -37.800095 0.0005 --> 4000
```

# SUMMARY: Process for expts with sizes 100 200 300

Creating a sorted all-record data file:

```
tail -n +2 CLUEdata2018_sortx.csv  > sortx_all.csv
```

Creating data files for experiments with n= **100  200  300** (you should change these to your designed values such as : **3000 6000 9000 12000 15000 18000**

```
for size in 100 200 300; do head -n $size sortx_all.csv > sortx_$size.csv; shuf sortx_$size.csv >  rand_$size.csv; done
```

Run experiments (warnings: it might take long, depending on how fast is your code):

```
for dict in 1 2; do for size in 100 200 300; do for type in sortx rand; do ./dict$dict $type-$size.csv out-$dict-$type-$size.txt < q$dict.txt > Stage$dict-$type-$size.txt; done ; done; done
```

Summary all expt outcome into file **expts.txt**

```
for dict in 1 2; do for type in sortx rand ;do for size in 100 200 300; do cat Stage$dict-$type-$size.txt | awk -v size=$size -v dict=$dict -v type=$type 'BEGIN{sum=0; n=0}{sum = sum+$4; n++}END{print "dict"dict, type, size, sum/n}' >> expts.txt ; done ; done; done
```

# MST ?

# 2015/2017/2019 MST

On Canvas

- 2015 Mid-sem
  - Warning: Way too hard!
- 2017 Mid-sem
- 2019 Mid-sem

# P6.1 Implementing a simple hash table, OR group work:

Choices:

- Implementing a simple hash table
- Individual working on ass2
- Group work with sample MST questions and/or programming tasks, for example:
  - Big-O questions
  - Hashing examples
  - AVL rotations
  - Programming: dynamic arrays and strings
  - Programming: linked lists