COMP20007 Workshop Week 8

1	**T1 (APSP with Floyd's Algorithm)**
2	T2: Simple sorting algorithms
3	T3: Mergesort complexity
4	Lab: Topological Sort (and interesting graph visualization with `graphviz`)

T3: Warshall's Algorithm

- what is dynamic programming?
- a dynamic programming solution for n!, for Fibonacci?
- describe the (dynamic programming) Warshall's algorithm

Floyd's Algorithm

- Task: APSP – For a graph G, find shortest path between all pair of vertices.
- Algorithm: A dynamic programming algorithm, similar to Warhall's.

T4: Floyd's Algorithm (APSP)

Floyd's algorithm builds on Warshall's algorithm to solve the all pairs shortest path problem: computing the length of the shortest path between each pair of vertices in a graph.

Rather than an adjacency matrix, we will require a weights matrix W , where Wij indicates the weight of the edge from i to j (if there is no edge from i to j then then Wij = ∞). We will ultimately find a distance matrix D in which Dij indicates the cost of the shortest path from i to j.

The sub-problems in this case will be answering the following question: What's the shortest path from i to j using only nodes in {1, . . . , k} as intermediate nodes?

To perform the algorithm we find Dk for each k ∈ {0,...,n} and set D := Dn. The update rule becomes the following:

$$D^0_{ij} := W_{ij}, \qquad D^k_{ij} := \min\left\{D^{k-1}_{ij}, D^{k-1}_{ik} + D^{k-1}_{kj}\right\}$$

T4: Floyd's Algorithm (APSP)

Given weighted graph $G=(V,W)$, where

- $V=\{1,2,\ldots,n\}$, and
- $W=[\![\ W_{ij}\ ]\!]$ is the weight (adjacency) matrix.

The task is

- find the shortest path between every pair of nodes, using nodes in the set $\{1,2,\ldots,n\}$ as intermediate step stones.

The task is complicated. But it seems easy if we already has:

- the shortest path between every pair of nodes, using nodes in the set $\{1,2,\ldots,n-1\}$ as intermediate step stones

T4

$$D^0_{ij} := W_{ij}, \qquad D^k_{ij} := \min\left\{D^{k-1}_{ij}, D^{k-1}_{ik} + D^{k-1}_{kj}\right\}$$

Perform Floyd's algorithm on the graph given by the following weights matrix:

$$W = \begin{bmatrix} 0 & 3 & \infty & 4 \\ \infty & 0 & 5 & \infty \\ 2 & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix}.$$

Example: $D_{k-1} \rightarrow D_k$ when k=1

$$D^0_{ij} := W_{ij}, \qquad D^k_{ij} := \min\left\{D^{k-1}_{ij}, D^{k-1}_{ik} + D^{k-1}_{kj}\right\}$$

$$W = \begin{bmatrix} 0 & 3 & \infty & 4 \\ \infty & 0 & 5 & \infty \\ 2 & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix}$$

row **k**:
$A_{kj} < \infty$ if there is a path **k**→j

column **k**:
$A_{ik} < \infty$ if there is a path i→**k**

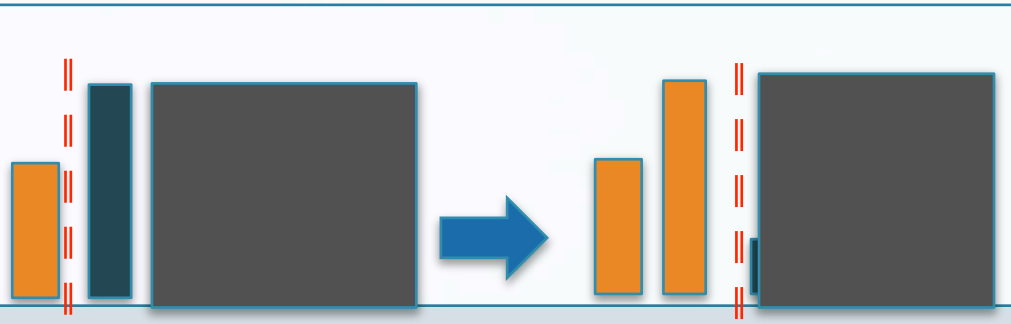
Simple Sorting Algorithms

- Any problem with Insertion Sort, Selection Sort, and especially, Counting Sort?

Insertion Sort: understanding (n=5)

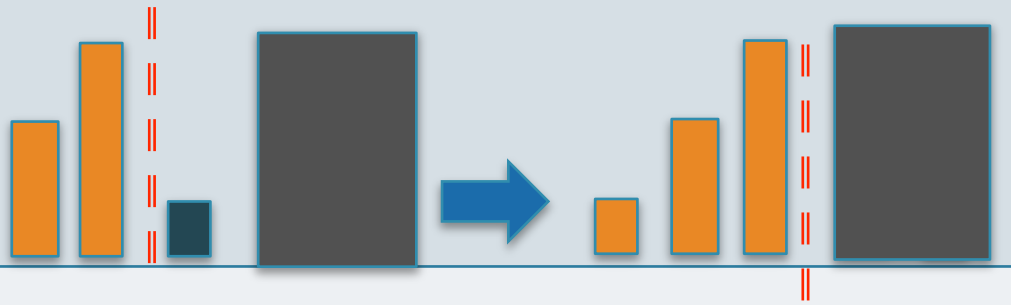
Round 1: consider `A[1]`

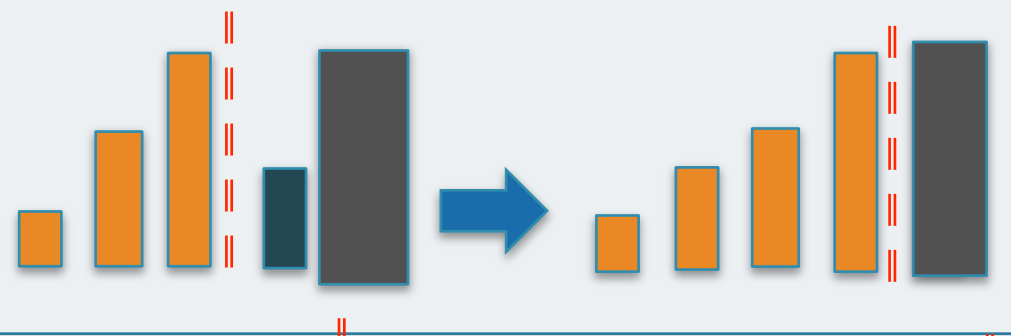
- `A[0..0]` *is sorted*
- *insert* `A[1]` *to the left so that* `A[0..1]` *is sorted*



Round 2: consider `A[2]`

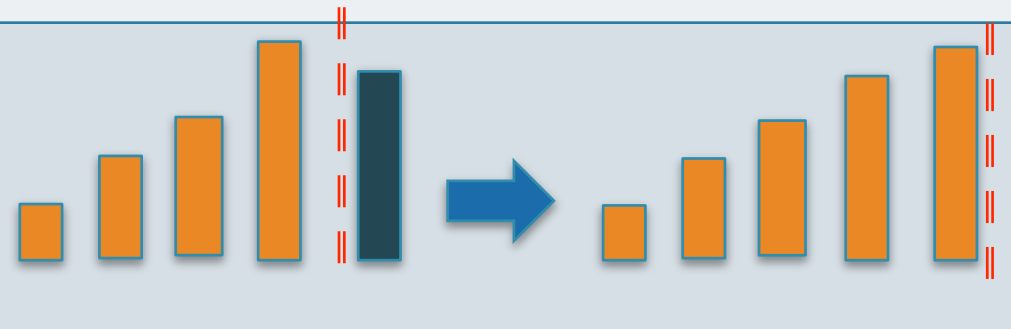
- `A[0..1]` *is sorted*
- *insert* `A[2]` *to the left so that* `A[0..2]` *is sorted*





Round i: consider `A[4]`

- `A[0..4]` *is sorted*
- *insert* `A[i]` *to the left so that* `A[0..4]` *is sorted*



Selection, Insertion, and Counting Sort

function InsertIntoSortedArray($A[0..(n-1)]$)

  $i \leftarrow (n-2)$

  while $i \geq 0$ **and** $A[i] < A[i+1]$ **do**

    $swap(A[i], A[i+1])$

    $i = i - 1$

function InsertionSort($A[0..(n-1)]$)

  for $i \leftarrow 1$ to $(n-1)$ **do**

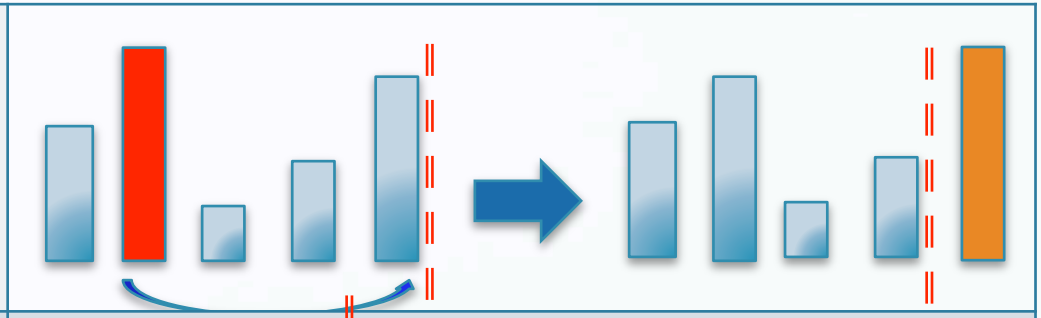
    InsertIntoSortedArray(A[0..i])

i. Run the algorithm on the input array: `[A N A L Y S I S] ,`
   Suppose: **Sort in increasing order.**
ii. Are there any restrictions on the input for the algorithm? If so, what?
iii. What is the time complexity of the algorithm?
iv. Is the sorting algorithm stable?
v. Does the algorithm sort in-place?
vi. Is the algorithm input sensitive?

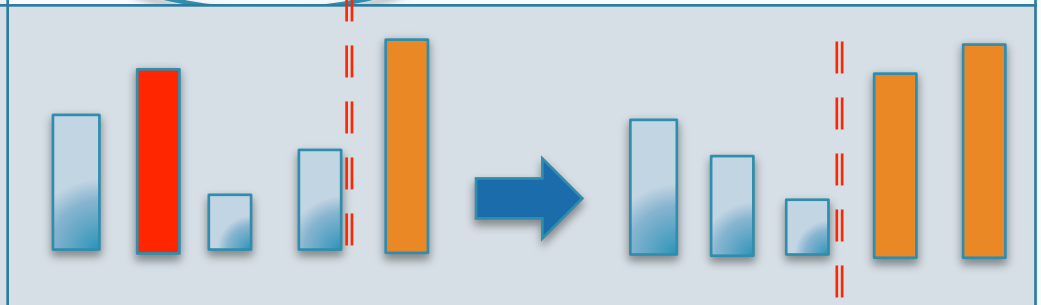
Selection Sort = ?

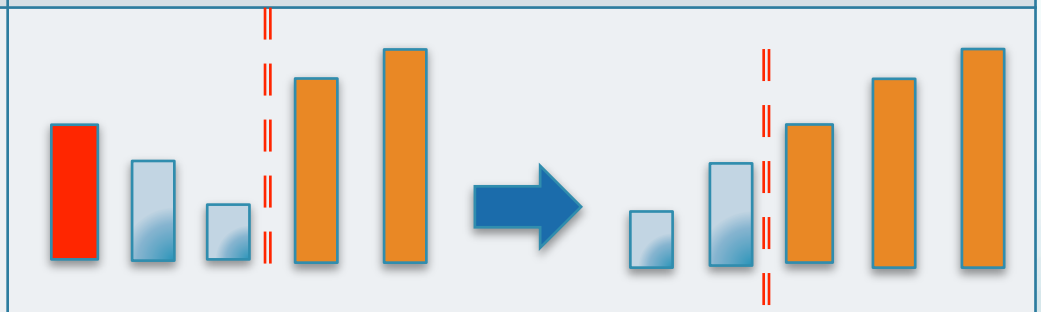
Selection Sort: `n=5, increasing order, select largest`

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

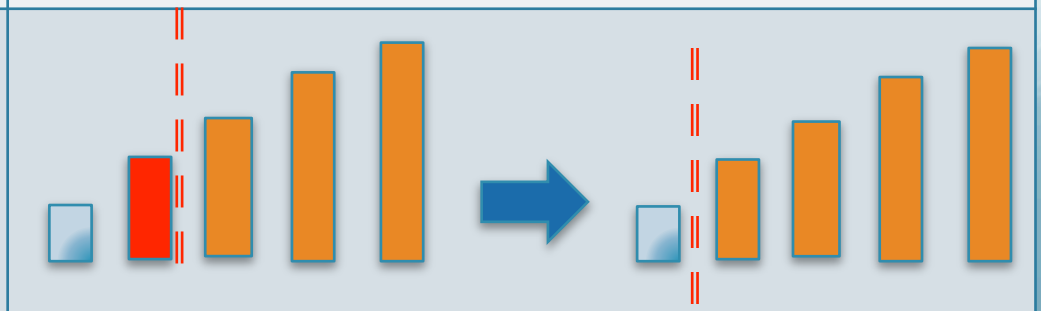


Round 2: consider `A[0..3]`
- *determine position of the largest*
- *swap with the last, ie.* `A[3]`





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



Selection, Insertion, and Counting Sort

function SELECTIONSORT($A[0..(n-1)]$)
  for $i \leftarrow 0$ to $n-2$ **do**
    $index\_max\_element \leftarrow IndexMaxElement(A[i..(n-1)])$
    swap $A[i]$ and $A[index\_max\_element]$

Answer the following questions about each algorithm:

i. Run the algorithm on the input array: `[A N A L Y S I S]`
   (Sort in increasing order, by selecting smallest)
ii. Are there any restrictions on the input for the algorithm? If so, what?
iii. What is the time complexity of the algorithm?
iv. Is the sorting algorithm stable?
v. Does the algorithm sort in-place?
vi. Is the algorithm input sensitive?

Counting Sort: Idea= ?

The task: Need to sort `A[]`. **Suppose that** `A` is an array of small integers, `0 <= A[i] <= d`, and the *range* `d` is small.

Counting Sort: for arrays of small integers

Approach: Make `A[]`= sorted version of `A[]`. Steps:

- Build the frequency array `f[]` of `d+1` element: `f[i]`= frequency of key value `i` in array `A[]`:

```
set f[j]= 0 for all j=0..d
for (i=0; i<n;  i++)
    f[A[i]] = f[A[i]] + 1;
```

- Scatter `f[0]` values `0`, `f[1]` values `1`… to `A[]`:

```
i=0;
for (j=0; j<=d; j++)
  for (k=0; k<f[j]; k++)
     A[i++]= j;
```

NOTE: this algorithm is essentially the same as the one provided in the lecture. It cannot be applied when A is an array of records and we need to sort according to a key in the records.

Selection, Insertion, and Counting Sort

```
function WriteValueInRange(A, value, start, end)
    for i ← start to (end − 1) do
        A[i] ← value

function ComputeHistogram(A[0..(n − 1)])
    WriteValueInRange(histogram, 0, 0, n)
    for i ← 0 to (n − 1) do
        histogram[A[i]] ← histogram[A[i]] + 1
    return histogram

function CountingSort(A[0..(n − 1)])
    histogram ← ComputeHistogram(A)
    i ← 0
    for j ← 0 to max(A) do
        WriteValueInRange(A, j, i, i + histogram[j])
        i ← i + histogram[j]
```

not `n`, *but* `max(A)+1`

Counting Sort: for arrays of of records that have keys of limited range

The task: Need to sort `A[]` according to A[i].key. **Suppose that** A[] is an array of small integers, `0min <= A[i] < d`, and the *range* `d` is small.

Approach: Build `B[]`= sorted version of `A[]`. Steps:

1. Build the frequency array `f[]` of `d` element: `f[i]`= frequency of key value `i` in array `A[]`
   → the first `f[0]` elements of `B[]` will be `0`, next `f[1] – 1`, and etc.
2. Transform `f[]` so that `f[i]=` starting position of value `i` in `B[]`, then we can build `B[]` with:

```
for (i=0; i<n;  i++)
   B[  f[A[i]]++ ] = A[i];
```

NOTES:

- the above algorithm can be adapted for a more general case when A[i] is a record with integer key and `min <= A[i].key <= max` for all `i`
- If A[i] is just a plain integer, we actually don't need B[],

Counting Sort: for arrays of of records that have keys <=d

The task: Need to sort `A[]` according to A[i].key. **Suppose that** A[] is an array of small integers, `0 <= A[i].key < d`, and the *range* `d` is small.

Approach: Build `B[]`= sorted version of `A[]`. Steps:

- Build the frequency array `f[]: f[i]`= frequency of key value `i`
- Transform `f[]` so that `f[i]=` starting position of key `i` in sorted `B[]`

```
for (c=0, i=0; i<= d; i++)
    tmp= f[i];
    f[i]= c;
    c += tmp;
```

- Build `B[]` with:

```
for (i=0; i<n;  i++)
    B[  C[A[i]-min]++ ] = A[i];
```

Counting Sort: for arrays of of records that have keys of limited range

The task: Need to sort `A[]` according to A[i].key. **Suppose that** A[] is an array of small integers, `min <= A[i].key < max`, and the *range* `d= max-min+1` is small.

Approach: Build `B[]`= sorted version of `A[]`. Steps:

- Build the frequency array `f[]: f[i]`= frequency of key value `i+min`
- Transform `f[]` so that `f[i]=` starting position of key `i+min in B`

```
for (c=0, i=0; i<= d; i++)
    tmp= f[i];
    f[i]= c;
    c += tmp;
```

- Build `B[]` with:

```
for (i=0; i<n;  i++)
    B[  C[A[i]-min]++ ] = A[i];
```

Selection, Insertion, and Counting Sort

Answer the following questions about each algorithm:

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

ii. Are there any restrictions on the input for the algorithm? If so, what?

iii. What is the time complexity of the algorithm?

iv. Is the sorting algorithm stable?

v. Does the algorithm sort in-place?

vi. Is the algorithm input sensitive?

T3: Mergesort Time Complexity

Mergesort is a divide-and-conquer sorting algorithm made up of three steps (in the recursive case):

1. Sort the left half of the input (using mergesort)
2. Sort the right half of the input (using mergesort)
3. Merge the two halves together (using a merge operation)

Construct a recurrence relation to describe the runtime of mergesort sorting n elements. Explain where each term in the recurrence relation comes from.

Use the Master Theorem to find the time complexity of Mergesort in Big-Theta terms.