Sorting

Example

Students	Alice	Bob	Cecil	David
DSA Mark	85%	47%	60%	70%
Age	22	21	38	19
Distance	30 miles	120 miles	8 miles	0 miles

We can sort them

- by mark: Bob, Cecil, David, Alice
- by more complicated conditions – e.g. housing score:
  Prefer students with age < 20, then those with age < 30.
  If in the same age category, compare by distance from home.

  Result: David, Bob, Alice, Cecil.

Sorting (students) can be done as long as we can **compare**, that is, we can determine whether

$$X < Y \qquad \text{given (students) } X \text{ and } Y.$$

This idea can be applied for any collection of data. It could be a collection of students, numbers, train companies, ... We can sort any such collection as long as we can **compare** any two of its elements.

Moreover, as we saw in the case of students, we might have several ways of comparing elements of the collection (by mark, age, something more complicated, ...). Then, to sort the collection we have to choose only one comparison criterion and use that one only. The sorting criterion will depend on our application.

Selection Sort

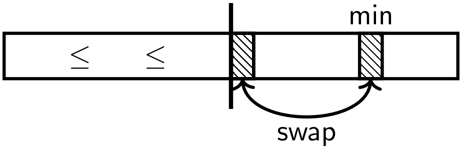
Algorithm:

1. Pick the smallest element from the second part, and
2. swap it with with the element on the position right after the sorted part.

Repeat until the sorted part extends to the end of the array.

Invariants: We split the array in two parts such that

- the first part of the array is sorted, and
- the second part contains elements $\geq$ than the elements in the first part.



The first (= the sorted) part is initially empty and the second part contains all elements. Then, every time we execute step 2. of the algorithm, the sorted part of the array extends by one and the second becomes one element shorter.

Note that the elements in the second part are in any order, i.e. they don't have to be sorted.

Example of a Selection Sort run

1. $\Big|\, 5, 12, 6, \underline{3}, 11, 8, 4$
2. $3 \,\Big|\, 12, 6, 5, 11, 8, \underline{4}$
3. $3, 4 \,\Big|\, 6, \underline{5}, 11, 8, 12$
4. $3, 4, 5 \,\Big|\, \underline{6}, 11, 8, 12$
5. $3, 4, 5, 6 \,\Big|\, 11, \underline{8}, 12$
6. $3, 4, 5, 6, 8 \,\Big|\, \underline{11}, 12$
7. $3, 4, 5, 6, 8, 11 \,\Big|\, \underline{12}$
8. $3, 4, 5, 6, 8, 11, 12 \,\Big|$

Selection Sort (pseudocode)

```
void selection_sort(int[] arr) {
  int n = arr.length;

  for (i=0; i<n; i++) {
    int min = i;

    // Find the smallest on positions i, i+1, ..., n-1
    for (j=i; j<n; j++) {
      if (arr[j] < arr[min])
        min = j;
    }

    // swap arr[min] and arr[i]
    int tmp = arr[min];
    arr[min] = arr[i];
    arr[i] = tmp;
  }
}
```

Note that the state of variable `i` marks the current position of the bar between the sorted and the unsorted parts of the array. For example, if `i` is equal to `3`, then the elements on positions `0`, `1` and `2` form a sorted sequence and the elements on positions `3` and further are all in the second (= unsorted) part of the array.

Time Complexity of Selection Sort

Looking up the smallest element is slow!

We do the following n-times:

1. Pick the smallest element from the second part, and
2. swap it with with the element on the position right after the sorted part.

1st iteration traverses n elements,
2nd iteration traverses $n-1$,
3rd iteration traverses $n-2$,
4th iteration traverses $n-3$,
5th iteration traverses $n-4$,
...
nth iteration traverses 1 element.

In total, we traverse

$$n + (n-1) + (n-2) + \ldots + 1 = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$$

elements.

$\Longrightarrow$ The time complexity is $\mathcal{O}(n^2)$.

When finding which element to swap (step 1 of the algorithm), we first traverse n elements, then $n - 1$, $n - 2$ and so on. In other words, the number of elements we need to traverse through is one element shorter after every iteration.

Heap Sort

Algorithm:

First, build a **(min) heap** from the elements in the array.

Then, do the following *n*-times:

1. Pick the smallest element from the **heap**, and
2. place it on the position right after the sorted part.

Time Complexity:

- Building the heap is in $\mathcal{O}(n)$.
- Deleting the minimum from a heap is in $\mathcal{O}(\log n)$ – we do this *n*-times: $\mathcal{O}(n \log n)$.

$\implies$ The time complexity is $\mathcal{O}(n \log n)$.

We have noticed that the reason why selection sort is slow is because it takes too long to find the next smallest element in the second part of the array. Next we describe *heap sort* which is just an improvement of *selection sort*. This algorithm is also gradually extending the first part of the array as before but instead of storing the remaining elements in the second half of the array we rather store them in a min heap.

In the previous lecture we discussed *(binary) max heaps*. Min heaps work exactly the same way but their first condition is "reversed" (it says that the priority of every node is *lower* than that of its children).

Recall that inserting elements into heaps is in $\mathcal{O}(\log n)$ and delete the minimum is also in $\mathcal{O}(\log n)$.

Building the heap can be done iteratively. We can just `insert` all elements of the array one by one and that would be in $\mathcal{O}(n \log n)$, which is still okay. However, last week we showed an algorithm called `heapify` which creates a heap from an array in $\mathcal{O}(n)$.

Heap Sort (pseudocode)

```
void heap_sort(int[] arr) {
  MinHeap h = new MinHeap<Int>();
  h.heapify(arr);

  for (i=0; i<arr.length; i++) {
    arr[i] = h.deleteMin();
  }
}
```

Theoretically best

Theorem

No deterministic comparison-based sorting algorithm has the Worst Case time complexity better than $\mathcal{O}(n \log n)$.

Proof:

(1) For a sequence $\langle x_1, x_2, ..., x_n \rangle$ on input, there is

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$$

many different ways to rearrange it.

The task of any sorting algorithm is to determine which of the many ways of rearranging leads to a sorted sequence.

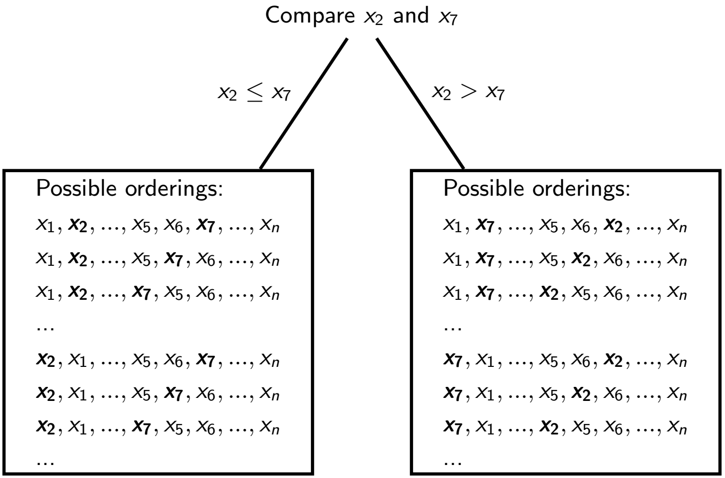
The number $n!$ is called the *nth factorial*. It represents in how many different ways we can rearrange n elements in sequence. For example for $n = 3$ we have six different rearrangements:

(i) 1, 2, 3
(ii) 2, 1, 3
(iii) 3, 1, 2
(iv) 1, 3, 2
(v) 2, 3, 1
(vi) 3, 2, 1

It is important to realise that $n!$ grows really fast. See its values for the first 9 elements:

n	1	2	3	4	5	6	7	8	9
$n!$	1	2	6	24	120	720	5040	40320	362880

(2) By comparing, e.g., x_2 and x_7, we eliminate many possibilities:



Compare x_2 and x_7
$x_2 \leq x_7$
$x_2 > x_7$
Possible orderings:
$x_1, \boldsymbol{x_2}, \ldots, x_5, x_6, \boldsymbol{x_7}, \ldots, x_n$
$x_1, \boldsymbol{x_2}, \ldots, x_5, \boldsymbol{x_7}, x_6, \ldots, x_n$
$x_1, \boldsymbol{x_2}, \ldots, \boldsymbol{x_7}, x_5, x_6, \ldots, x_n$
$\ldots$
$\boldsymbol{x_2}, x_1, \ldots, x_5, x_6, \boldsymbol{x_7}, \ldots, x_n$
$\boldsymbol{x_2}, x_1, \ldots, x_5, \boldsymbol{x_7}, x_6, \ldots, x_n$
$\boldsymbol{x_2}, x_1, \ldots, \boldsymbol{x_7}, x_5, x_6, \ldots, x_n$
$\ldots$
Possible orderings:
$x_1, \boldsymbol{x_7}, \ldots, x_5, x_6, \boldsymbol{x_2}, \ldots, x_n$
$x_1, \boldsymbol{x_7}, \ldots, x_5, \boldsymbol{x_2}, x_6, \ldots, x_n$
$x_1, \boldsymbol{x_7}, \ldots, \boldsymbol{x_2}, x_5, x_6, \ldots, x_n$
$\ldots$
$\boldsymbol{x_7}, x_1, \ldots, x_5, x_6, \boldsymbol{x_2}, \ldots, x_n$
$\boldsymbol{x_7}, x_1, \ldots, x_5, \boldsymbol{x_2}, x_6, \ldots, x_n$
$\boldsymbol{x_7}, x_1, \ldots, \boldsymbol{x_2}, x_5, x_6, \ldots, x_n$
$\ldots$

How many times do we need to compare to distinguish between the $n!$-many different possibilities?

If we compare x_2 with x_7 and obtain that $x_2 \leq x_7$, that means that reorderings which have x_2 before x_7 are possible candidates, such as:

$$x_1, \boldsymbol{x_2}, \ldots, x_5, x_6, \boldsymbol{x_7}, \ldots, x_n$$

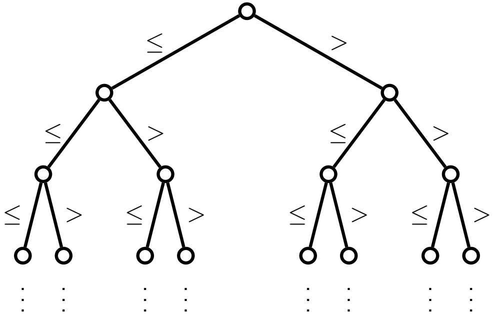
whereas those where x_7 precedes x_2 are illegal (and become "eliminated"), such as:

$$x_1, \boldsymbol{x_7}, \ldots, \boldsymbol{x_2}, x_5, x_6, \ldots, x_n.$$

Any algorithm which is sorting the sequence has to eventually eliminate all but the only one legal reordering. Any time it compares two elements it eliminates some possibilities until, eventually, it is left with only one. Even if the algorithm could keep track of all still not eliminated reorderings, how many times does it have to compare?

Remark: Since we are interested in the Worst Case complexity we can assume that no two elements in the sequence are equal. (Then only one reordering is legal.)

(3) This is determined by the *decision tree* of the algorithm (every node corresponds to one comparison):



Every comparison eliminates some possibilities. In the leaves we are left with only one of the $n!$-many options.

The height of the tree is the lower bound on the number of comparisons!

(4) In the best possible case, the decision tree is perfectly balanced. Then its height is $\mathcal{O}(\log(n!))$. But how big is $\log(n!)$?

We have a "sandwich" of inequalities:

$$
\begin{array}{ccc}
n^n \leq (n!)^2 & \qquad & n! \leq n^n \\
\Downarrow & & \Downarrow \\
n \log n \leq \log(n!)^2 = 2\log(n!) & & \log(n!) \leq \log(n^n) = n \log n
\end{array}
$$

Therefore, $\mathcal{O}(\log(n!)) = \mathcal{O}(n \log n)$. $\square$

Even though we can't compute $\log(n!)$ we can estimate it from the top and from the bottom (i.e. we have a sandwich).

To show $n^n \leq (n!)^2$ observe that

$$\begin{array}{rllll} (n!)^2 = n & \times(n-1) & \times(n-2) & \times \ \dots & \times 1 \\ \times\, 1 & \times 2 & \times 3 & \times \ \dots & \times n \end{array}$$

Consequently, in order to have $n^n \leq (n!)^2$ it is enough to show that $(n-k)(k+1) \geq n$ (for every k between 0 and $n-1$) and the latter inequality is equivalent to

$$(n-k)(k+1) - n \geq 0$$

(by subtracting n from both sides). Moreover, $(n-k)(k+1) - n$ is equal to

$$kn - k(k+1) = k[n - (k+1)]$$

which is always ≥ 0 since $0 \leq k \leq n-1$.

Summary

	Selection Sort	Heap Sort
Time Complexity	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$
Space Complexity	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Stable?	No	No

(The Average Case and Worst Case complexities are the same.)

Stability means that the order of the elements with the same key is preserved.

For example, selection sort, for $\langle (2, A), (2, B), (1, C) \rangle$, outputs

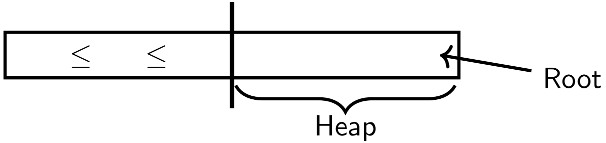
$$\langle (1, C), (2, B), (2, A) \rangle .$$

Bonus: In-place Heap Sort

With a little bit of extra effort, we can sort the array with no extra memory allocated (that is, in-place).

Invariants: We split the array in two parts such that

- the first part of the array is sorted,
- the second part contains elements which $\geq$ than elements in the first part and
- is organised as a **min heap**, stored in the reversed order, that is,
  1. the root is on position $n-1$,
  2. its children are on positions $n-3$ and $n-2$,
  3. their children are on positions $n-7$, $n-6$, $n-5$, and $n-4$,
  4. ...



Recall that `deleteMin` is done as follows:

1. Remember the value of the root node.
2. Move the last node on the last level to be the new root.
3. Bubble the new root down.

So after `deleteMin` is the position where we used to store the last node on the last level unused. Therefore, we can store there the value that we obtain in the first step (as described above). After we do that the first (= ordered) part of the array extends by one element.