Sorting algorithms

Input: a sequence of items that can be compared (integers, strings...)

Output: a sequence of the same items, sorted according to the comparison function

4 3 5 7 12 8

→ 3 4 5 7 8 12

Which algorithms?
What are their complexities?
Can we do better?

Insertion sort (cardplayer sorting algorithm) (41 swaps)

Input: Array T of n elements

```
For i ← 1 to n-1                          } n times
    For j ← i-1 to 1 decreasing
        If T[j] ≤ T[j+1]
            Break;
        Else
            Swap T[j] and T[j+1]          } O(1)
```

Complexity

* Best case (array already sorted) $\Theta(n)$
* Worst case (array in reverse order)
  $1+2+3+\dots+n-1$ swaps
  $= \frac{n(n-1)}{2} = \Theta(n^2)$
* Average-case : $\boxed{\Theta(n^2)}$ (not proved)

Merge sort (divide and conquer) (39 swaps)

Base case: If $n = 1$, then already sorted $O(1)$

Recursive case

* **Divide** the array in two arrays of near-identical size $\sim n/2, n/2$ $O(1)$ ← if array $\Theta(n)$ ← if linked list
* **Apply** recursively merge sort on the two subarrays $2 \times T(n/2)$
* **Combine** the two results by iterating in order and taking each time the smaller of the smallest element of the two arrays $\Theta(n)$

$$\begin{matrix} \underline{1} & \underline{4} & \underline{5} & \underline{9} & \underline{12} \\ \underline{2} & \underline{3} & \underline{7} & \underline{8} & \underline{10} \end{matrix}$$

$1\ 2\ 3\ 4\ 5\ 7\ 8\ 9\ 10\ 12$

$$\begin{cases} T(1) = O(1) \\ T(n) = 2\,T(n/2) + \Theta(n) \end{cases}$$

Master theorem: $a = 2$, $b = 2$, $f(n) = \Theta(n^1)$

$$c = \log_b a = \log_2 2 = 1$$

2nd case of the master theorem $\longrightarrow$ $\boxed{T(n) = \Theta(n \log n)}$

(but you need extra space in memory for recombining arrays)

Quicksort (divide and conquer) 13 swaps

Base case: If $n = 1$ then already sorted $O(1)$

Recursive case:

* Choose a pivot element x (e.g., the first element of the array) $O(1)$
* Build two subarrays: all elements greater than x (size n_2), and all elements $\leq x$ (size n_1) $\Theta(n)$
* Recursive quicksort on these two subarrays (without the pivot, in between the two subarrays) $T(n_1) + T(n_2)$ with n_1, n_2 sizes of the subarrays

Complexity

Worst case: pivot is either always the first or the last element of the subarray (will in particular happen if we choose as pivot the first element and the array is already sorted): $\Theta(n^2)$

Best case

Every time we choose a pivot, it is in the middle of the array.

$$T(n) \simeq 2 \times T(n/2) + \Theta(n)$$
$$\rightarrow \boxed{T(n) = \Theta(n \log n)}$$

Optimization: Randomly choose the pivot uniformly among the elements

On average, (not proved) $T(n) = \Theta(n \log n)$

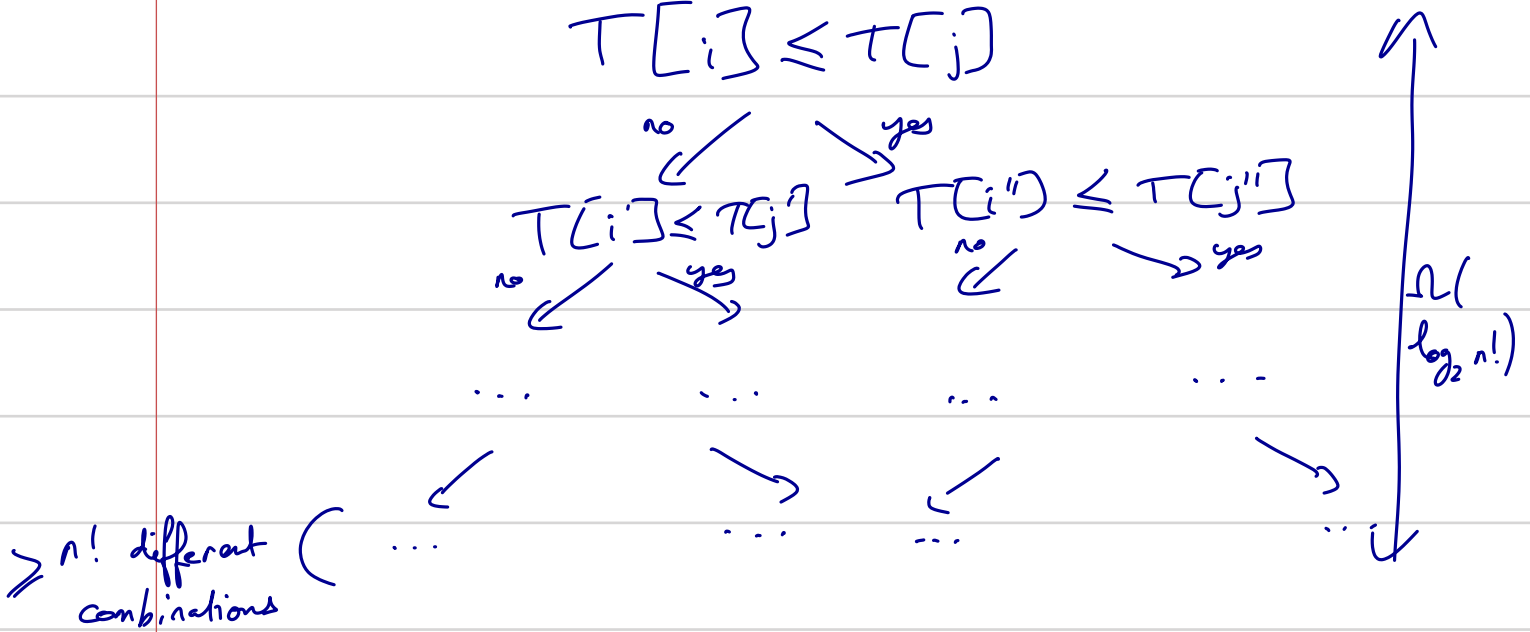
In practice, very efficient and the algorithm for sorting that is the most commonly used / implemented.

Lower bound

Does there exist an algorithm with a complexity better than $\Theta(n \log n)$ to sort an array of n elements by comparing these elements two-by-two?

Sorting algorithm → Doing comparisons ← focus on this

→ Doing swaps, constructing new arrays, insert in a linked list...



Stirling's equivalent $n! \sim \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$

So $\log n! \sim n \log n - n \log e + \frac{1}{2} \log(2\pi n)$

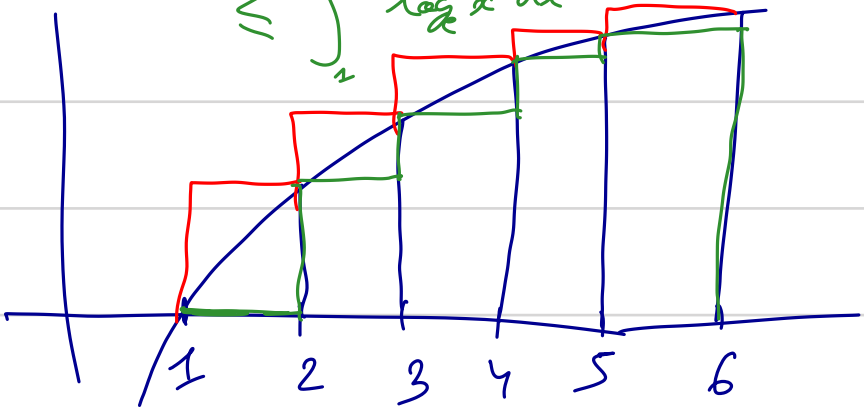
$\sim n \log n$

$\log_2 n! = \Theta(n \log n)$

Without Stirling's equivalent:

$$\log_e n! = \sum_{k=2}^{n} \log_e k \geq \int_1^n \log_e x \, dx$$

$$\leq \int_1^{n+1} \log_e x \, dx$$



$$\int_1^N \log_e x \, dx = \left[x \log_e x - x \right]_1^N$$

$$= N \log_e N - N + 1$$

$$= \Theta(N \log N)$$

So $\log n! = \Theta(n \log n)$.

A comparison-based sorting algorithm cannot have a complexity better than $\Omega(n \log n)$.

Count sort. Assume we know that all values to be sorted are integers between 1 and m

Counting array :

1	2	3	4	5	6	7	8	9	10	J	Q	K
|	|		||	|	||		||		|		|	||

1 2 4 4 5 6 6 8 8 10 Q K K

Complexity : $\Theta(n+m) = \Theta(\max(n,m))$

- n : # elements to sort
- m : # possible values

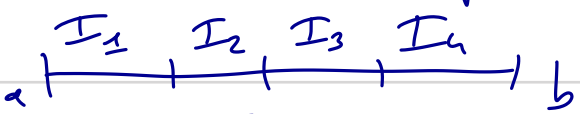
Other example of non-comparison sort algorithm : **bucket sort**

$B = \alpha n$ with $\alpha \leq n$

B : # buckets

Assume values to be sorted are uniformly distributed within $[a;b]$.

* We build B buckets of the same size within $[a;b]$.



$B = 4$

* We put each element to sort in the corresponding I_j interval.
* We sort (e.g., by insertion sort) each I_j.

Complexity : $\Theta\left(n + B \times \left(\frac{n}{B}\right)^2\right)$

If $\alpha = \frac{B}{n}$ is a constant, this becomes $\Theta\left(n + \frac{B}{\alpha^2}\right) = \Theta(n)$

Heap sort

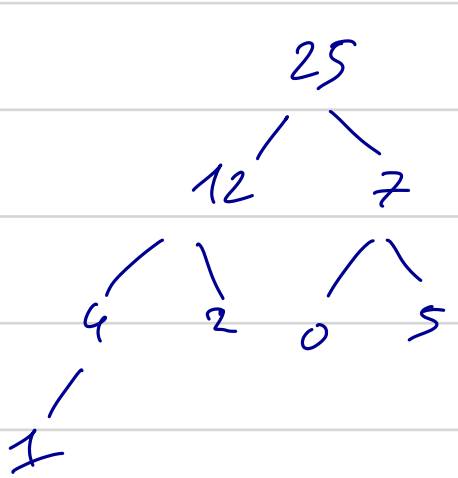
A heap is a special data structure which is formed (not a search tree) of a binary tree encoded into an array.

→ Balanced but not necessarily complete

→ Almost complete: all levels of the tree are complete except the last one where all the leaves are at the leftmost position.

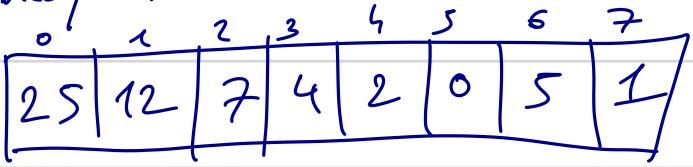
→ Value attached to each node.

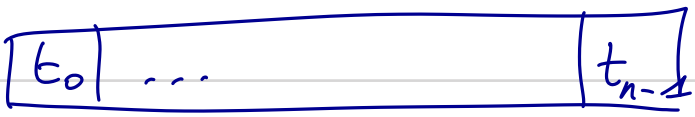
→ If u has children u_l, u_r then $v(u) \geq v(u_l)$ and $v(u) \geq v(u_r)$



depth	
0	complete
1	complete
2	complete
3	not complete / leftmost leaves

A heap is encoded into an array:





value of the root

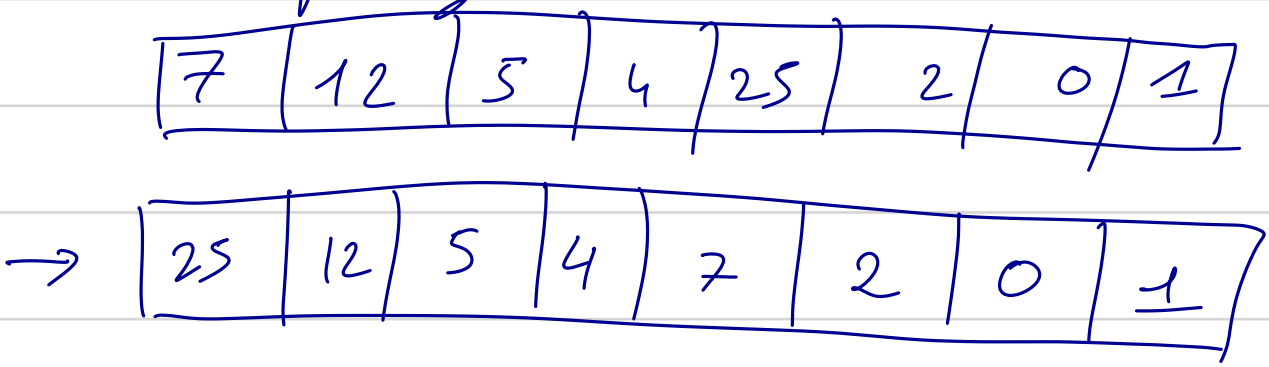
value of the last leaf

If u is at position k ($v(u) = t_k$) and u has children u_l, u_r and parent u_p then:

$v(u_l) = t_{2k+1}$ $v(u_r) = t_{2k+2}$ $v(u_p) = t_{\lfloor \frac{k-1}{2} \rfloor}$

Building a heap

For every element (from the last to the first) apply a make-heap function that turns the subtree rooted at this node into a heap by swapping values lower than their children values repeatedly.



→

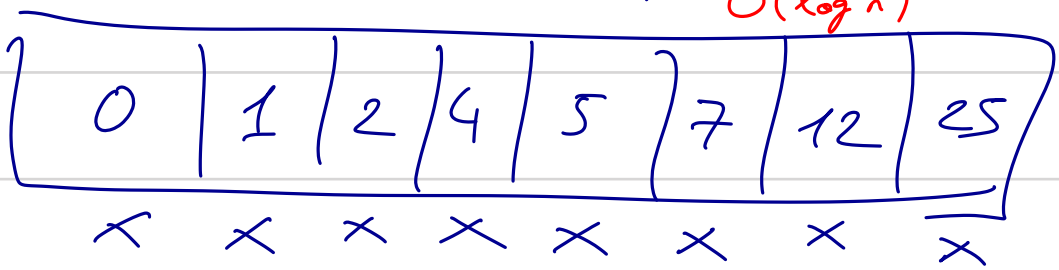
Complexity of building the heap : $\Theta(n \log n)$
(actually, $\Theta(n)$, see the textbook)

HeapSort

→ Build a heap Θ(n log n) (or even Θ(n))

→ Repeatedly: (n-1 times) n times

* Swap the value of the root with the last element of the heap O(1)
* Note that the last element of the array is out of the heap O(1)
* Apply make-heap to the root O(log n)



Complexity:

Θ(n log n)