

UNIVERSITÉ
PARIS SCIENCES
& LETTRES

Sorting Algorithms

The Art of Computer Programming

Pierre Senellart



16 November 2026

Outline

- The sorting problem
- Insertion sort
- Merge sort
- Quicksort
- A lower bound
- Sorting without comparisons
- Heapsort
- Summary
- Coda

UNIVERSITÉ PARIS SCIENCES & LETTRES

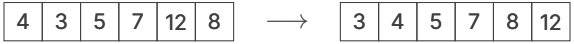
The sorting problem



Sorting

Input a sequence of n items that can be **compared** (integers, floating-point numbers, strings, records with a key, ...)

Output a sequence of the **same items**, in **increasing order** for the comparison function



- One of the most **fundamental** problems of computer science: a building block of countless algorithms (binary search, duplicate removal, joins in databases, computing medians and quantiles, ...)
- One of the **best-studied**: the whole of Knuth [1998] is about it
- Three questions for today:
  - Which **algorithms**?
  - What are their **complexities**?
  - Can we do **better**?

Vocabulary

- **Comparison-based**: the algorithm only learns about the items by **comparing two of them**
- **In place**: the algorithm sorts within the input array, using only $O(1)$ or $O(\log n)$ extra memory
- **Stable**: items with **equal keys** keep their relative order, e.g., sorting students by name and then by grade keeps the names in order within each grade
- **Cost**: number of **comparisons** and of **swaps** or moves, as a function of n

A useful helper

Most algorithms of today exchange two elements of an array:

Python

```python
# Python: multiple assignment
t[i], t[j] = t[j], t[i]
```

C++

```cpp
// C++: from <utility>
std::swap(t[i], t[j]);
```

C

```c
void swap(int *t, int i, int j) {
  int tmp = t[i];
  t[i] = t[j];
  t[j] = tmp;
}
```

Reminders from previous lectures:

- $O()$, $\Omega()$, $\Theta()$ notation; worst-case and average-case complexity (lecture 3)
- Recursion and the call stack (lecture 2); divide and conquer and the **master theorem** (lecture 7)
- Binary trees (lecture 6)

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Insertion sort



Insertion sort: the card player's algorithm

- How do you sort a hand of cards? Take the cards one by one and **insert** each one at its place among the cards already in your hand
- Invariant: after step i, the first $i + 1$ elements of the array are **sorted** (but not necessarily at their final position)
- Insertion of element i: move it to the left, one swap at a time, as long as its left neighbor is larger

Input: Array T of n elements
Output: T sorted in place

```
1: for i ← 1 to n − 1 do
2:     j ← i
3:     while j > 0 and T[j − 1] > T[j] do
4:         swap T[j − 1] and T[j]
5:         j ← j − 1
6:     end while
7: end for
```

Insertion sort: example

Step	Array (sorted prefix highlighted)
start	4 3 5 7 12 8
$i = 1$: insert 3, 1 swap	3 4 5 7 12 8
$i = 2$: insert 5, 0 swap	3 4 5 7 12 8
$i = 3$: insert 7, 0 swap	3 4 5 7 12 8
$i = 4$: insert 12, 0 swap	3 4 5 7 12 8
$i = 5$: insert 8, 1 swap	3 4 5 7 8 12

Number of swaps: the number of **inversions** of the input, i.e., of pairs (i, j) with $i < j$ and $T[i] > T[j]$: here 2, at most $n(n-1)/2$.

Insertion sort: code

Python

```python
def insertion_sort(t):
  for i in range(1, len(t)):
    j = i
    while j > 0 and t[j - 1] > t[j]:
      t[j - 1], t[j] = t[j], t[j - 1]
      j -= 1
```

C

```c
void insertion_sort(int *t, int n) {
  for (int i = 1; i < n; i++) {
    int j = i;
    while (j > 0 && t[j - 1] > t[j]) {
      swap(t, j - 1, j);
      j--;
    }
  }
}
```

- Note the **short-circuit** evaluation of the condition: `t[j - 1]` is only accessed when `j > 0`

Insertion sort: complexity

- The outer loop runs $n - 1$ times; each iteration of the inner loop costs $O(1)$
- **Best case** (array already sorted): the inner loop stops immediately, $n - 1$ comparisons, 0 swap: **$\Theta(n)$**
- **Worst case** (array in decreasing order): element i moves all the way to position 0,

$$1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} \text{ swaps: } \mathbf{\Theta(n^2)}$$

- **Average case** (random order): half of the pairs are inverted on average, $n(n-1)/4$ swaps: **$\Theta(n^2)$** (not proved here)
- **Properties**: in place, stable (equal elements are never swapped); cost $\Theta(n + \text{inversions})$
- Excellent for **small** or **nearly sorted** arrays, hopeless for large random ones: $n = 10^6$ means $\approx 2.5 \times 10^{11}$ swaps

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Merge sort



Merge sort: divide and conquer

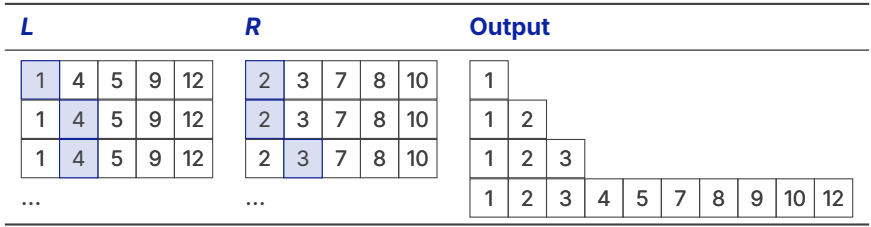
- **Base case**: an array of size $n \leqslant 1$ is already sorted
- **Recursive case**:
  1. **Divide** the array into two halves of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$
  2. **Sort** each half recursively
  3. **Merge** the two sorted halves into one sorted array

Input: Array T of n elements
Output: A sorted array with the same elements

1: **if** $n \leqslant 1$ **then**
2: **return** T
3: **end if**
4: $L \leftarrow \text{MergeSort}(T[0 \dots \lfloor n/2 \rfloor - 1])$
5: $R \leftarrow \text{MergeSort}(T[\lfloor n/2 \rfloor \dots n - 1])$
6: **return** $\text{Merge}(L, R)$

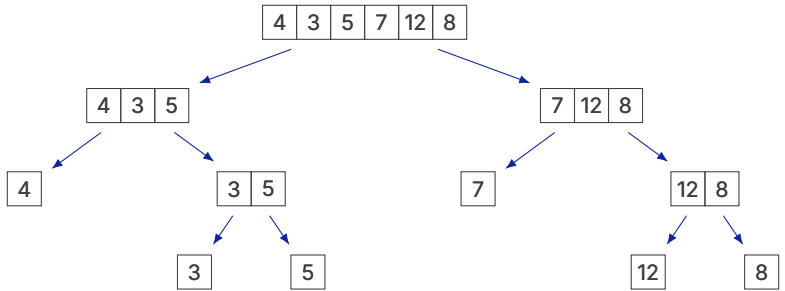
Merging two sorted arrays

- Walk through both arrays **from left to right** simultaneously, each time taking the **smaller** of the two current elements
- When one array is exhausted, copy the rest of the other



Each step outputs one element after one comparison: **$\Theta(n)$** for n elements in total, but requires an **extra array** of size n.

Merge sort: example



Then merge on the way back up: 3 5, 3 4 5; 8 12, 7 8 12; finally 3 4 5 7 8 12.

Merge sort in Python

Python

```
def merge(left, right):
    result = []
    i, j = 0, 0
    while i < len(left) and j <
    ↪  len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

Python

```
def merge_sort(t):
    if len(t) <= 1:
        return t
    mid = len(t) // 2
    left = merge_sort(t[:mid])
    right = merge_sort(t[mid:])
    return merge(left, right)
```

- Returns a **new** list, the input is untouched
- The `<=` in `merge` makes the sort **stable**: on ties, the left element comes first

Merge sort: complexity

- Let $T(n)$ be the cost of sorting n elements:

$$T(1) = \Theta(1), \qquad T(n) = 2\,T(n/2) + \Theta(n)$$

  (dividing is $O(1)$ for an array, sorting the halves is $2\,T(n/2)$, merging is $\Theta(n)$)
- **Master theorem** with $a = 2$, $b = 2$, $f(n) = \Theta(n)$: $c = \log_b a = 1$ and $f(n) = \Theta(n^c)$, second case

$$\boldsymbol{T(n) = \Theta(n \log n)}$$

- Directly: the recursion tree has $\log_2 n$ levels, and the merges of each level cost $\Theta(n)$ in total
- **Same cost** in the best, worst, and average cases, even on an already sorted array
- **Stable**; **not in place**: $\Theta(n)$ extra memory for merging (plus $\Theta(\log n)$ for the call stack)

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Quicksort



Quicksort: divide and conquer, the other way around

- Merge sort does the work **after** the recursive calls (merging); quicksort does it **before** (partitioning), so that nothing remains to be done afterwards [Hoare, 1962]
- **Base case**: an array of size $n \leqslant 1$ is sorted
- **Recursive case**:
  1. Choose an element of the array, the **pivot** x (e.g., the last one)
  2. **Partition** the array: elements $\leqslant x$ to the left, the pivot, elements $> x$ to the right
  3. Sort the left part and the right part recursively



After partitioning, the pivot is at its **final** position; the two parts are sorted independently and **in place**.

Partitioning in place

With pivot $x = T[hi]$: scan the array from left to right, maintaining the invariant that $T[lo \ldots i]$ contains the elements $\leqslant x$ seen so far, and $T[i+1 \ldots j-1]$ those $> x$.

Input: Array T, bounds $lo \leqslant hi$
Output: Position p of the pivot after partitioning

```
1: x ← T[hi]; i ← lo − 1
2: for j ← lo to hi − 1 do
3:     if T[j] ⩽ x then
4:         i ← i + 1; swap T[i] and T[j]
5:     end if
6: end for
7: swap T[i + 1] and T[hi]
8: return i + 1
```

Partitioning: example

Step	Array, pivot 7
start, $i = -1$	4 12 5 8 3 7
$j = 0$: $4 \leqslant 7$, $i = 0$	4 12 5 8 3 7
$j = 1$: $12 > 7$; $j = 2$: $5 \leqslant 7$, $i = 1$, swap	4 5 12 8 3 7
$j = 3$: $8 > 7$; $j = 4$: $3 \leqslant 7$, $i = 2$, swap	4 5 3 8 12 7
final swap, return 3	4 5 3 7 12 8

Highlighted: the elements known to be $\leqslant$ the pivot, and the pivot. Each element is compared to the pivot exactly once: $\Theta(n)$ comparisons, at most n swaps, no extra memory.

Quicksort: code

Python

```python
def partition(t, lo, hi):
    x = t[hi]
    i = lo - 1
    for j in range(lo, hi):
        if t[j] <= x:
            i += 1
            t[i], t[j] = t[j], t[i]
    t[i + 1], t[hi] = t[hi], t[i + 1]
    return i + 1
def quicksort(t, lo, hi):
    if lo < hi:
        p = partition(t, lo, hi)
        quicksort(t, lo, p - 1)
        quicksort(t, p + 1, hi)
```

C

```c
int partition(int *t, int lo, int hi) {
  int x = t[hi];
  int i = lo - 1;
  for (int j = lo; j < hi; j++) {
    if (t[j] <= x) {
      i++;
      swap(t, i, j);
    }
  }
  swap(t, i + 1, hi);
  return i + 1;
}
void quicksort(int *t, int lo, int hi) {
  if (lo < hi) {
    int p = partition(t, lo, hi);
    quicksort(t, lo, p - 1);
    quicksort(t, p + 1, hi);
  }
}
```

Called as `quicksort(t, 0, n - 1)`: `hi` is **included**.

Quicksort: complexity

- Partitioning n elements costs $\Theta(n)$; then $T(n) = T(n_1) + T(n_2) + \Theta(n)$ with $n_1 + n_2 = n - 1$ the sizes of the two parts
- **Best case**: the pivot always lands in the middle, $T(n) \approx 2\,T(n/2) + \Theta(n)$, hence **$\Theta(n \log n)$** as for merge sort
- **Worst case**: the pivot is always the smallest or largest element, $n_1 = 0$, $n_2 = n - 1$:

$$T(n) = T(n-1) + \Theta(n) = \Theta(n + (n-1) + \cdots + 1) = \mathbf{\Theta(n^2)}$$

  This happens with the pivot $T[hi]$ on an array that is **already sorted**, hardly a rare input!
- **Average case** over random inputs, or for any input with a pivot chosen **uniformly at random**: **$\Theta(n \log n)$**, with about $1.39\,n \log_2 n$ comparisons (proof in Cormen et al., 2022)
- **In place**, **not stable**
- In practice the most commonly used and implemented general sorting algorithm: few moves, excellent use of CPU caches; always randomize the pivot or take the median of three elements

UNIVERSITÉ PARIS SCIENCES & LETTRES

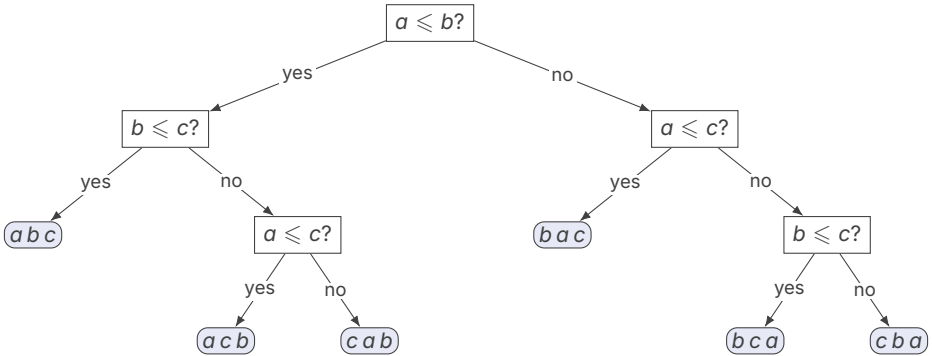
A lower bound



Can we do better than $n \log n$?

- Insertion sort: $\Theta(n^2)$; merge sort, heapsort (later) and quicksort on average: $\Theta(n \log n)$
- Is there a comparison-based algorithm in, say, $\Theta(n)$ or $\Theta(n\sqrt{\log n})$?
- To answer, we abstract away everything an algorithm does except its **comparisons**: swaps, copies, insertions in lists, etc., are ignored, which can only make the algorithm look faster
- Any comparison-based algorithm, run on inputs of size n, can be seen as a **decision tree**: each internal node is a comparison $T[i] \leqslant T[j]$, with two children for the two outcomes; each leaf is a final answer, i.e., a permutation of the input
- The number of comparisons on a given input is the **depth** of the leaf reached; the worst case is the **height** of the tree

The decision tree for $n = 3$



Sorting (a, b, c): $3! = 6$ possible outputs, so 6 leaves; the height is 3, and indeed 3 comparisons are needed in the worst case.

The lower bound

- There are $n!$ possible orderings of n distinct items, and the algorithm must be able to output each of them: the decision tree has **at least $n!$ leaves**
- A binary tree of height h has at most 2^h leaves, so

$$2^h \geqslant n! \quad \Longrightarrow \quad h \geqslant \log_2(n!)$$

- **Stirling's formula**: $n! \sim \sqrt{2\pi n}\,(n/e)^n$, hence

$$\log_2(n!) = n \log_2 n - n \log_2 e + \tfrac{1}{2} \log_2(2\pi n) + o(1) = \Theta(n \log n)$$

- **Theorem.** Any comparison-based sorting algorithm performs **$\Omega(n \log n)$** comparisons in the worst case (and also on average, with a bit more work)
- Merge sort is therefore **optimal** up to a constant factor; no clever trick will give $\Theta(n)$ comparisons

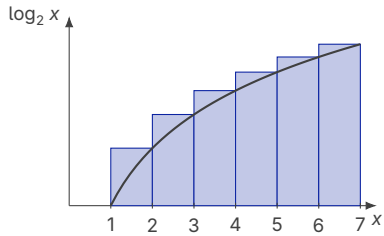
$\log_2(n!) = \Theta(n \log n)$ without Stirling

- Upper bound: $n! \leqslant n^n$, so $\log_2 n! \leqslant n \log_2 n$
- Lower bound: $\log_2 n! = \sum_{k=2}^{n} \log_2 k$, and each term $\log_2 k$ is at least $\int_{k-1}^{k} \log_2 x \, \mathrm{d}x$ (see the figure), so

$$\log_2 n! \geqslant \int_1^n \log_2 x \, \mathrm{d}x = n \log_2 n - \frac{n-1}{\ln 2}$$

since $\log_2 x = \frac{\ln x}{\ln 2}$ and $\frac{\mathrm{d}}{\mathrm{d}x}(x \ln x) = \ln x + 1$, so $\frac{\mathrm{d}}{\mathrm{d}x}(x \ln x - x) = \ln x$

- Both bounds are $\Theta(n \log n)$



The rectangle of height $\log_2 k$ on $[k-1, k]$ lies above the curve, since $\log_2$ is increasing.

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Sorting without comparisons



Counting sort

- The lower bound only applies to **comparison-based** algorithms; with more knowledge about the keys, one can do better
- Assume all keys are **integers between 0 and $m - 1$**, with m not too large (grades, ages, bytes, ...)
- **Counting sort**: count how many times each value occurs, then output each value as many times as it was counted

Input:

4	1	6	8	2	10	4	8	5	6

values from 0 to $m - 1 = 10$

value	0	1	2	3	4	5	6	7	8	9	10
count	0	1	1	0	2	1	2	0	2	0	1

Output:

1	2	4	4	5	6	6	8	8	10

Counting sort: code and complexity

Python

```
def counting_sort(t, m):
    count = [0] * m
    for x in t:
        count[x] += 1
    k = 0
    for v in range(m):
        for _ in range(count[v]):
            t[k] = v
            k += 1
```

_: by convention, a variable whose value is never used (lecture 5)

- Counting: $\Theta(n)$; output: $\Theta(n + m)$; total **$\Theta(n + m)$**, **linear** when $m = O(n)$
- No comparison between elements at all: the lower bound does not apply
- Extra memory $\Theta(m)$; useless if $m \gg n$, e.g., for arbitrary 64-bit integers
- To sort **records** by an integer key, a variant computes the position of each key from the cumulative counts, and is **stable**; this is the base of **radix sort**, which sorts on one digit (or byte) at a time

Bucket sort

- Assume the keys are real numbers **uniformly distributed** in an interval $[a, b)$
- Split $[a, b)$ into B **buckets** $I_1, \ldots, I_B$ of equal width; put each element into the bucket of its key ($\Theta(n)$); sort each bucket, e.g., by insertion sort; concatenate the buckets



$B = 4$ buckets

- If bucket j receives n_j elements, sorting it costs $\Theta(n_j^2)$; the total is $\Theta(n + \sum_j n_j^2)$
- With uniform keys, each bucket receives n/B elements on average, hence $\Theta(n + B\,(n/B)^2) = \Theta(n + n^2/B)$
- Taking $B = \alpha n$ buckets for a constant α: **$\Theta(n)$ on average**; but $\Theta(n^2)$ in the worst case (all keys in one bucket) and the assumption on the distribution is essential

UNIVERSITÉ PARIS SCIENCES & LETTRES

Heapsort

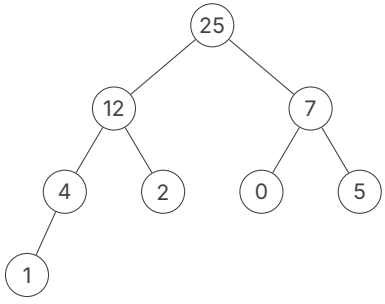


Heaps

A **(binary max-)heap** is a binary tree (lecture 6) such that:

- it is **almost complete**: every level is full except possibly the last, whose nodes are all at the leftmost positions (thus the tree is balanced, of height $\lfloor \log_2 n \rfloor$)
- each node carries a value, and the **heap property** holds: the value of a node is $\geqslant$ **the values of its children**

Consequences: the **maximum** is at the root; every subtree is a heap; there is **no** order between siblings or between subtrees (a heap is not a search tree)



A heap in an array

- Since the tree is almost complete, it can be stored in an **array**, level by level, from left to right, **without any pointer**:

0	1	2	3	4	5	6	7
25	12	7	4	2	0	5	1

- The root is at position 0; the last leaf at position $n - 1$
- If a node is at position k:
  - its left child is at position **$2k + 1$**, its right child at **$2k + 2$** (if these are $< n$)
  - its parent is at position **$\lfloor (k - 1)/2 \rfloor$** (if $k > 0$)

  Check: the children of 12 (position 1) are 4 and 2 (positions 3 and 4); the parent of 1 (position 7) is 4 (position 3)
- The heap property reads: $T[k] \geqslant T[2k + 1]$ and $T[k] \geqslant T[2k + 2]$ whenever these positions exist

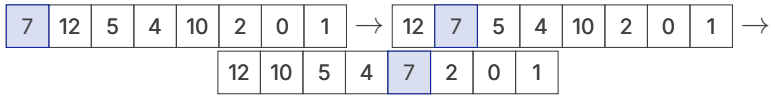
Restoring the heap property: sift down

Suppose, in a heap of size n, that the subtrees of node k are heaps but the value at k may be too small; **sift it down**: swap it with the larger of its children, and repeat from there, until it is $\geqslant$ both children or reaches a leaf. Cost: proportional to the height of the subtree, $O(\log n)$.

1: **loop**
2: $largest \leftarrow k$
3: **if** $2k + 1 < n$ **and** $T[2k + 1] > T[largest]$ **then** $largest \leftarrow 2k + 1$
4: **if** $2k + 2 < n$ **and** $T[2k + 2] > T[largest]$ **then** $largest \leftarrow 2k + 2$
5: **if** $largest = k$ **then return**
6: swap $T[k]$ and $T[largest]$; $k \leftarrow largest$
7: **end loop**

Sift down: example

Sifting down the value 7 from the root, whose two subtrees are heaps:

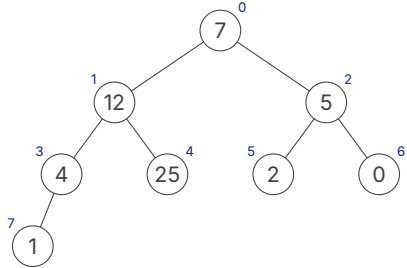


- At position 0, the children are 12 (position 1) and 5 (position 2): 12 is the largest, swap 7 and 12
- At position 1, the children are 4 (position 3) and 10 (position 4): swap 7 and 10
- At position 4, there is no child (positions 9 and 10 are $\geqslant n = 8$): stop

Two swaps for a tree of height 3: the cost is bounded by the height.

Building a heap

- Given an arbitrary array, make it a heap by sifting down every node, **from the last one to the root**: when node k is processed, its subtrees are already heaps
- Nodes in the second half of the array are leaves and need no work: start at $\lfloor n/2 \rfloor - 1$
- Cost: $n/2$ sift-downs of $O(\log n)$ each, so $\boldsymbol{O(n \log n)}$; a finer analysis, summing the heights of the subtrees, gives $\boldsymbol{\Theta(n)}$ (see Cormen et al., 2022)

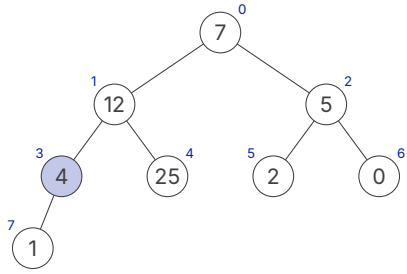


7	12	5	4	25	2	0	1

Start: an arbitrary array

Building a heap

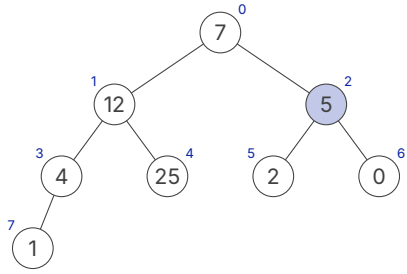
- Given an arbitrary array, make it a heap by sifting down every node, **from the last one to the root**: when node k is processed, its subtrees are already heaps
- Nodes in the second half of the array are leaves and need no work: start at $\lfloor n/2 \rfloor - 1$
- Cost: $n/2$ sift-downs of $O(\log n)$ each, so $\boldsymbol{O(n \log n)}$; a finer analysis, summing the heights of the subtrees, gives $\boldsymbol{\Theta(n)}$ (see Cormen et al., 2022)



Sift down position 3 (value 4, child 1): no change

Building a heap

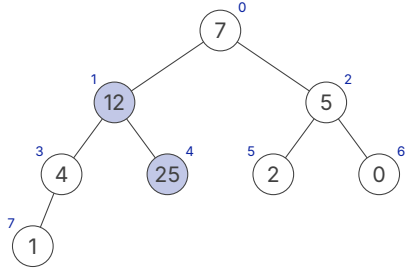
- Given an arbitrary array, make it a heap by sifting down every node, **from the last one to the root**: when node k is processed, its subtrees are already heaps
- Nodes in the second half of the array are leaves and need no work: start at $\lfloor n/2 \rfloor - 1$
- Cost: $n/2$ sift-downs of $O(\log n)$ each, so $\boldsymbol{O(n \log n)}$; a finer analysis, summing the heights of the subtrees, gives $\boldsymbol{\Theta(n)}$ (see Cormen et al., 2022)



Sift down position 2 (value 5, children 2 and 0): no change

Building a heap

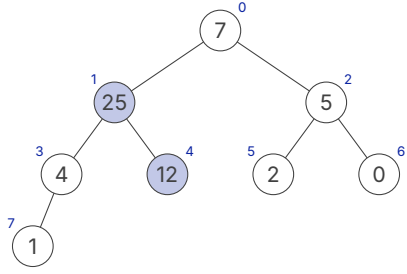
- Given an arbitrary array, make it a heap by sifting down every node, **from the last one to the root**: when node k is processed, its subtrees are already heaps
- Nodes in the second half of the array are leaves and need no work: start at $\lfloor n/2 \rfloor - 1$
- Cost: $n/2$ sift-downs of $O(\log n)$ each, so $\boldsymbol{O(n \log n)}$; a finer analysis, summing the heights of the subtrees, gives $\boldsymbol{\Theta(n)}$ (see Cormen et al., 2022)



Sift down position 1 (value 12, children 4 and 25): 25 is the largest, swap 12 and 25

Building a heap

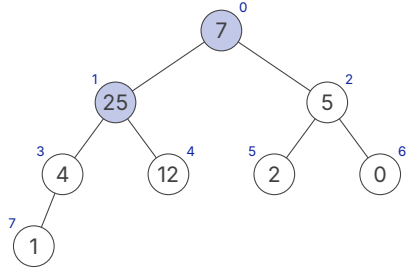
- Given an arbitrary array, make it a heap by sifting down every node, **from the last one to the root**: when node k is processed, its subtrees are already heaps
- Nodes in the second half of the array are leaves and need no work: start at $\lfloor n/2 \rfloor - 1$
- Cost: $n/2$ sift-downs of $O(\log n)$ each, so $\boldsymbol{O(n \log n)}$; a finer analysis, summing the heights of the subtrees, gives $\boldsymbol{\Theta(n)}$ (see Cormen et al., 2022)



12 and 25 are swapped; position 4 is a leaf: stop

Building a heap

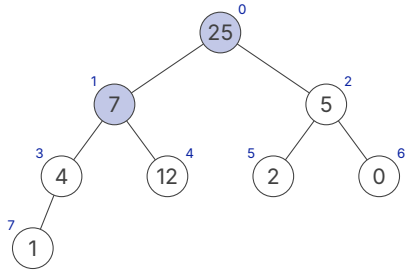
- Given an arbitrary array, make it a heap by sifting down every node, **from the last one to the root**: when node k is processed, its subtrees are already heaps
- Nodes in the second half of the array are leaves and need no work: start at $\lfloor n/2 \rfloor - 1$
- Cost: $n/2$ sift-downs of $O(\log n)$ each, so $\boldsymbol{O(n \log n)}$; a finer analysis, summing the heights of the subtrees, gives $\boldsymbol{\Theta(n)}$ (see Cormen et al., 2022)



Sift down position 0 (value 7, children 25 and 5): 25 is the largest, swap 7 and 25

Building a heap

- Given an arbitrary array, make it a heap by sifting down every node, **from the last one to the root**: when node k is processed, its subtrees are already heaps
- Nodes in the second half of the array are leaves and need no work: start at $\lfloor n/2 \rfloor - 1$
- Cost: $n/2$ sift-downs of $O(\log n)$ each, so $\boldsymbol{O(n \log n)}$; a finer analysis, summing the heights of the subtrees, gives $\boldsymbol{\Theta(n)}$ (see Cormen et al., 2022)

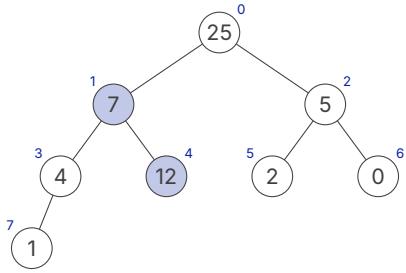


25	7	5	4	12	2	0	1

7 and 25 are swapped; continue from position 1

Building a heap

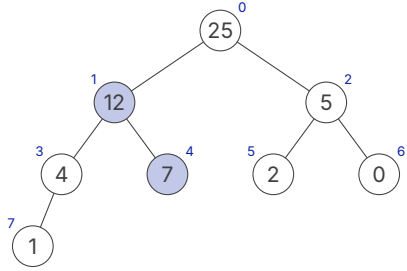
- Given an arbitrary array, make it a heap by sifting down every node, **from the last one to the root**: when node k is processed, its subtrees are already heaps
- Nodes in the second half of the array are leaves and need no work: start at $\lfloor n/2 \rfloor - 1$
- Cost: $n/2$ sift-downs of $O(\log n)$ each, so $\boldsymbol{O(n \log n)}$; a finer analysis, summing the heights of the subtrees, gives $\boldsymbol{\Theta(n)}$ (see Cormen et al., 2022)



Position 1 (value 7, children 4 and 12): 12 is the largest, swap 7 and 12

Building a heap

- Given an arbitrary array, make it a heap by sifting down every node, **from the last one to the root**: when node k is processed, its subtrees are already heaps
- Nodes in the second half of the array are leaves and need no work: start at $\lfloor n/2 \rfloor - 1$
- Cost: $n/2$ sift-downs of $O(\log n)$ each, so $\boldsymbol{O(n \log n)}$; a finer analysis, summing the heights of the subtrees, gives $\boldsymbol{\Theta(n)}$ (see Cormen et al., 2022)



7 and 12 are swapped; position 4 is a leaf: stop. The heap is built

Heapsort

- Williams [1964]: sort by repeatedly **extracting the maximum** of a heap
  1. Build a heap from the array
  2. Repeat $n - 1$ times: **swap** the root (the maximum) with the last element of the heap; **shrink** the heap by one (the maximum is now at its final position, outside the heap); **sift down** the new root

Step	Array (sorted suffix highlighted)
heap built	25 12 5 4 7 2 0 1
swap 25 and 1, sift 1 down ($n = 7$)	12 7 5 4 1 2 0 25
swap 12 and 0, sift 0 down ($n = 6$)	7 4 5 0 1 2 12 25
swap 7 and 2, sift 2 down ($n = 5$)	5 4 2 0 1 7 12 25
... 4 more steps	0 1 2 4 5 7 12 25

Heapsort: code

C

```c
void sift_down(int *t, int n,
               int k) {
  while (true) {
    int largest = k;
    int l = 2 * k + 1, r = 2 * k + 2;
    if (l < n && t[l] > t[largest])
      largest = l;
    if (r < n && t[r] > t[largest])
      largest = r;
    if (largest == k) return;
    swap(t, k, largest);
    k = largest;
  }
}
```

C

```c
void heapsort(int *t, int n) {
  for (int k = n / 2 - 1;
       k >= 0; k--)
    sift_down(t, n, k);
  for (int end = n - 1;
       end > 0; end--) {
    swap(t, 0, end);
    sift_down(t, end, 0);
  }
}
```

Heapsort: complexity

- Building the heap: $O(n \log n)$, even $\Theta(n)$
- Then $n - 1$ times: one swap, $O(1)$, and one sift-down, $O(\log n)$
- Total: **$\Theta(n \log n)$** in all cases, **in place**, **not stable**
- Optimal like merge sort, without the extra array; in practice slower than quicksort (poor use of CPU caches), hence used as a fallback when quicksort behaves badly
- The heap is a data structure in its own right: a **priority queue** with insertion and extraction of the maximum in $O(\log n)$ (Python: `heapq`; C++: `std::priority_queue`)

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Summary

Comparison of sorting algorithms

Algorithm	Best	Average	Worst	Extra memory	Stable	In place
Insertion sort	n	n^2	n^2	$O(1)$	yes	yes
Merge sort	$n \log n$	$n \log n$	$n \log n$	$O(n)$	yes	no
Quicksort	$n \log n$	$n \log n$	n^2	$O(\log n)$	no	yes
Heapsort	$n \log n$	$n \log n$	$n \log n$	$O(1)$	no	yes
Counting sort	$n + m$	$n + m$	$n + m$	$O(n + m)$	yes	no
Bucket sort	n	n	n^2	$O(n)$	yes	no

- Complexities are in $\Theta()$ of the given quantity; the first four are comparison-based, and $\Omega(n \log n)$ in the worst case is unavoidable for them
- Counting sort needs integer keys in $[0, m)$; bucket sort needs uniformly distributed keys
- Other classical algorithms: selection sort and bubble sort ($\Theta(n^2)$, of pedagogical interest only), Shell sort, and radix sort

In practice: use the library (Python)

- **Python:** `sorted(t)` returns a new sorted list, `t.sort()` sorts in place; both use **Timsort** [Peters, 2002], a stable merge sort exploiting already sorted runs, $O(n \log n)$ worst case, $O(n)$ on sorted data

Python

```
students.sort(key=lambda s: s.grade, reverse=True)  # by decreasing grade
sorted(words, key=len)                              # by length, stable
```

- Write your own sort only to learn, or when the keys allow a linear-time algorithm

In practice: use the library (C and C++)

- **C:** `qsort` from `<stdlib.h>`, with a comparison function; in spite of its name, not necessarily a quicksort (glibc uses a merge sort when memory allows)

C

```c
int compare_ints(const void *a, const void *b) {
  int x = *(const int *) a, y = *(const int *) b;
  return (x > y) - (x < y);   // negative, zero, or positive
}
qsort(t, n, sizeof(int), compare_ints);
```

- **C++:** `std::sort(v.begin(), v.end())` from `<algorithm>`, typically an **introsort** (quicksort, switching to heapsort in bad cases and to insertion sort on small ranges); `std::stable_sort` is a merge sort

UNIVERSITÉ
PARIS
SCIENCES
& LETTRES

Coda

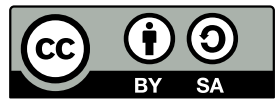


Wednesday's tutorial and references

- Wednesday's tutorial: implementing and **measuring** some of these algorithms, in Python and C, on random and on sorted inputs; watching quicksort degrade on sorted input; using a heap as a priority queue
- Reference: Cormen et al. [2022], chapters 2 (insertion sort, merge sort), 6 (heapsort), 7 (quicksort), and 8 (lower bound, counting sort, bucket sort)
- Run and compare the algorithms of this lecture yourself:
  `https://pierre.senellart.com/other/deck/`

Licensing

This work is licensed under a Creative Commons “Attribution-ShareAlike 4.0 International” license.



Bibliography I

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 4th edition, 2022. ISBN 978-0-262-04630-5. URL https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/.

C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962. doi: 10.1093/comjnl/5.1.10.

Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998. ISBN 978-0-201-89685-5.

Tim Peters. Timsort description. https://github.com/python/cpython/blob/main/Objects/listsort.txt, 2002.

J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6): 347–348, 1964. doi: 10.1145/512274.512284.