Insertion and Merge Sort

Chapters 2.1, 2.3.1 of Cormen's book

Giulia Bernardini
giulia.bernardini@units.it

Algorithmic Design a.y. 2024/2025

Why Sorting?

- Endless obvious applications

- Many problems become easy when the input is sorted (e.g., finding the median of a set of items, finding the location of a specific item in the set)

- Not-so-obvious applications: e.g., data compression

Insertion Sort

```
INSERTION-SORT(A)
1   for j = 2 to A.length
2       key = A[j]
3       // Insert A[j] into the sorted sequence A[1 .. j − 1].
4       i = j − 1
5       while i > 0 and A[i] > key
6           A[i + 1] = A[i]
7           i = i − 1
8       A[i + 1] = key
```

Quiz Time

Please go to **www.wooclap.com,** use the code **BERNARDINI02** and answer the question (it is anonymous unless you decide to use your name). You do not need to create an account!

Insertion Sort

```
INSERTION-SORT(A)
1   for j = 2 to A.length
2       key = A[j]
3       // Insert A[j] into the sorted sequence A[1 .. j − 1].
4       i = j − 1
5       while i > 0 and A[i] > key
6           A[i + 1] = A[i]
7           i = i − 1
8       A[i + 1] = key
```

$\Theta(n^2)$ comparisons and swaps

Merge Sort

It is a divide and conquer algorithm.

```
MERGE-SORT(A, p, r)
1   if p < r
2       q = ⌊(p + r)/2⌋
3       MERGE-SORT(A, p, q)
4       MERGE-SORT(A, q + 1, r)
5       MERGE(A, p, q, r)
```

Input: an array A and two indices $p \leq q$ that determine the subarray $A[p \ldots q]$ to be sorted.

Merge-sort recursively calls itself on smaller and smaller subarrays, until they are of size 1.

Then it merges the solutions to the smaller subarrays until the solution to the whole A is obtained.

Merge Sort

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

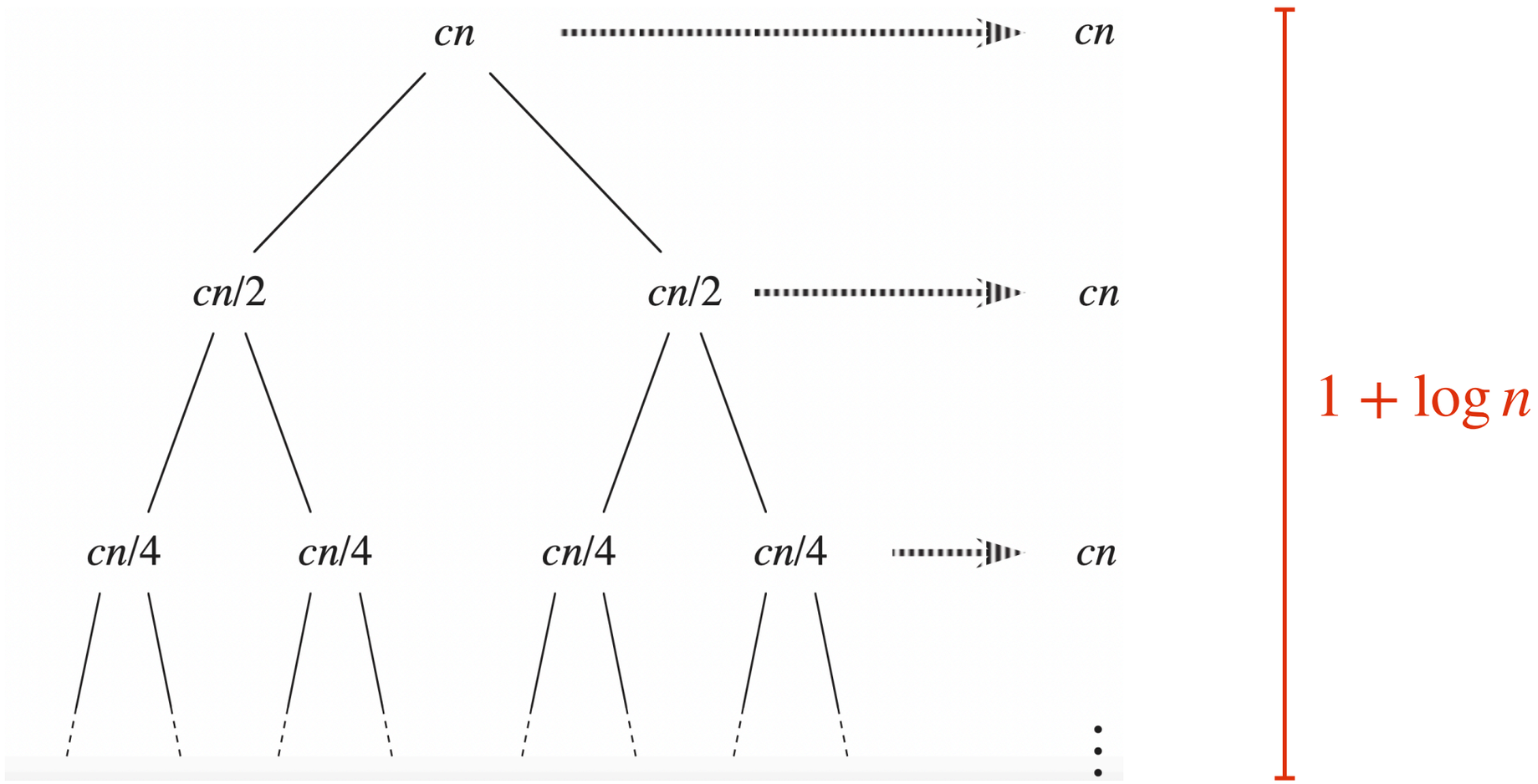
Copy the two portions of *A* to be merged into two new arrays *L* and *R*

Two fingers merge algorithm

Merge Sort: Analysis

The time complexity of Merge Sort can be written as

$T(n) = c_1 + cn + 2T\left(\frac{n}{2}\right)$ which is a recurrence equation.



$T(n) = \Theta(n \log n)$

Divide and Conquer Technique

Divide and conquer is a recursive algorithm design technique.

The approach is to break the problem into subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively calling the algorithm itself, and then combine these solutions to solve the original problem.

Divide and conquer algorithms have three main steps:

- Divide
- Conquer
- Combine

Insertion vs Merge Sort

Difference between $\Theta(n^2)$ and $\Theta(n \log n)$ time.

To sort $n = 10^7$ numbers on a machine that executes 10^7 instructions per second with Insertion Sort takes roughly

$$\frac{(10^7)^2}{10^7} = 10^7 \text{ seconds, which is more than 100 days.}$$

To sort the same 10^7 numbers with Merge Sort takes roughly

$$\frac{10^7 \log 10^7}{10^7} \leq 28 \text{ seconds.}$$

Implemented in Python, I.S. can take about $0.2n^2$ microseconds, while M.S. can take about $2.2n \log n$ microseconds.

Insertion vs Merge Sort

Insertion Sort sorts *A* in place: it rearranges the numbers within *A*, with only a constant number $\Theta(1)$ of them stored outside the array at any time.

This is not the case for Merge Sort, which makes copies of parts of *A* into new arrays *L* and *R*. Merge Sort needs $\Theta(n)$ auxiliary space to store *L* and *R*, so it is not in place.

Since the constants for Insertion Sort are very small, it outperforms Merge Sort on short arrays.

If you are sure you will sort only small arrays, using Insertion Sort is a good idea; otherwise, Merge Sort or other sorting algorithms we will see are much better suited.

Compare Sorting Algorithms

https://visualgo.net/en/sorting

Play with it!

Heaps and Heapsort

Chapter 6 of Cormen's book

Giulia Bernardini
giulia.bernardini@units.it

Algorithmic Design a.y. 2024/2025

Heaps

Heaps are data structures that model priority queues: sets of elements each associated with a key (their priority). Desired operations on a priority queue are:

- Pick the element with the max priority (key)
- Pick the element with the min priority
- Insert elements
- Delete elements
- Change the priorities of the elements

A heap is a visualization of an array as a nearly-complete binary tree.

Max Heaps

In a heap we have:

- Index 1 is the root
- parent(i)=i/2
- left(i)=2i
- right(i)=2i+1

Max heaps are heaps with the additional property that the key of each node is $\geq$ than the keys of its children.

Quiz Time

Please go to **www.wooclap.com,** use the code **BERNARDINI02** and answer the question (it is anonymous unless you decide to use your name). You do not need to create an account!

Building Max Heaps

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4       largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7       largest = r
 8  if largest ≠ i
 9       exchange A[i] with A[largest]
10       MAX-HEAPIFY(A, largest)
```

Building Max Heaps

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4       largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7       largest = r
 8  if largest ≠ i
 9       exchange A[i] with A[largest]
10       MAX-HEAPIFY(A, largest)
```

Complexity of max_heapify: $O(\log n)$ (length of a root-to-leaf path)

Building Max Heaps

BUILD-MAX-HEAP(A)

1 $A.heap\text{-}size = A.length$
2 **for** $i = \lfloor A.length/2 \rfloor$ **downto** 1
3 MAX-HEAPIFY(A, i)

Building Max Heaps

```
BUILD-MAX-HEAP(A)
1   A.heap-size = A.length
2   for i = ⌊A.length/2⌋ downto 1
3       MAX-HEAPIFY(A, i)
```

Straightforward analysis: $O(n \log n)$ time

Refined analysis with power series: $O(n)$ time

Heapsort

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

Time complexity: $O(n \log n)$ because we call max_heapify n times.

https://www.cs.usfca.edu/~galles/visualization/HeapSort.html

Quiz Time

Please go to **www.wooclap.com,** use the code **BERNARDINI02** and answer the question (it is anonymous unless you decide to use your name). You do not need to create an account!