Advanced Programming Techniques

PART III

Sorting Algorithms

Beniamin Bogosel

Ecole Polytechnique
Department of Applied Mathematics

Sorting

- One of the most fundamental algorithmic problems
- Generally: sort some data with respect to a given key
- Here we ignore the data and just focus on the sorting part.
- Sorting problem:
  - **Input**: a sequence of n numbers: $\langle a_1, ..., a_n \rangle$.
  - **Output**: a permutation of the starting sequence $\langle a'_1, ..., a'_n \rangle$ such that

$$a'_1 \leq a'_2 \leq ... \leq a'_n.$$

Applications

- Sort emails with respect to various criteria
- Sort search engine results
- Sort object facets for 3D rendering
- Manage banking operations
- ...

Applications

[Skiena, *The Algorithm Design Manual*

$\star$ **Searching**: unsorted array $O(n)$; sorted array $O(\log n)$. Search preprocessing is one of the most important applications of sorting

$\star$ **Closest pair**: Given a set of n numbers, find the pair having the smallest difference. Once the array is sorted, the closest pair consists of numbers lying on consecutive positions somewhere in the sorted table. Total complexity: $O(n \log n)$ including sorting; Compare this with the $O(n^2)$ brute force approach!

$\star$ **Element uniqueness**: Test for duplicates in a set. If unsorted brute force needs $O(n^2)$ operations. If the array is sorted, non-unique elements are consecutive! Complexity $O(n \log n)$ including sorting.

$\star$ **Frequency distribution:** Find which element occurs the largest number of times! In a sorted array this is easy! Sort, count. $O(n \log n)$.

Applications

⋆ **Selection**: What is the kth largest element in an array? In a sorted array this is obvious.
⋆ **Convex hulls**: Find the convex polygon with the smallest area containing a sequence of given points. Process points in the order of the x-coordinate (for example...). Applications in graphics processing.

Take-Home lesson

Sorting lies at the heart of many algorithms. Sorting the data is one of the first things any algorithm designer should try in the quest for efficiency.

Different types of sorting algorithms

- **Iterative sorting**: based on iterating the table once or more
- **Recursive sorting**: using a recursive procedure
- **In place sorting**: modifying directly the structure, without needing extra memory space
- **Stable sorting**: preserve relative order of equal elements
  - Useful if sorting with respect to multiple criteria is needed
  - Example: a table of students is sorted lexicographically regarding their names
  - If the table is sorted in a stable way concerning some other criteria (grade, course options,...) then the alphabetical order is preserved in sub classes

What we saw previously?

⋆ Insertion sort:

- Assuming $A[1..j-1]$ is sorted, insert $A[j]$ on the correct position

⋆ Merge sort:

- Assuming $A[p..q]$ and $A[q+1..r]$ are sorted, merge them into the sorted interval $A[p..r]$
- Do this recursively!

Other algorithms

⋆ Bubble sort:

- Loop for i from 1 to n
- Loop for j from 1 to $n - i$
- If $A[j] > A[j+1]$ swap them
- At each iteration in the outer loop, the $n - i + 1$ greatest element reaches its position
- If no swaps occur we can stop: the array is sorted
- In place; complexity $O(n^2)$...

⋆ Selection sort:

- Loop for i from 1 to n
- Find the minimal element in $A[i ... A.length]$
- Swap it with the element $A[i]$
- In place; complexity $O(n^2)$

Up to this point

Algorithm	Complexity			In place?
	Worst	Average	Best	
INSERTION-SORT	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
SELECTION-SORT	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
BUBBLE-SORT	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
MERGE-SORT	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	no

Can we do better? $\Theta(n \log n)$? in place?

Quick sort

- Invented by Hoare in 1960
- Top 10 algorithms of the 20th century (SIAM)
- Example of the "divide and conquer" technique
- In place
- Complexity? $\Theta(n^2)$ in worst case, $\Theta(n \log n)$ on average

Quick Sort: main lines

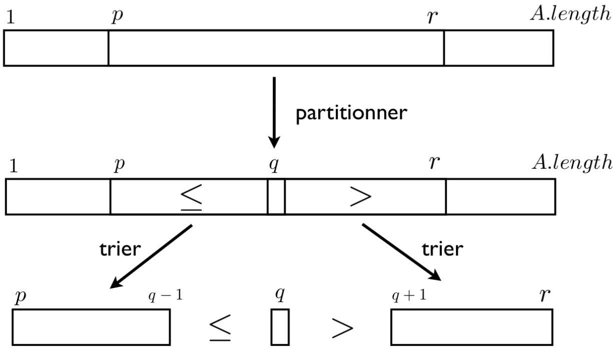
To sort the sub-array $A[p..r]$:

- Split the interval in two parts: $A[p..q-1]$ and $A[q+1..r]$ such that
  - All elements in $A[p..q-1]$ are $\leq A[q]$
  - All elements in $A[q+1..r]$ are $\geq A[q]$
- Call the algorithm recursively to sort $A[p..q-1]$ and $A[q+1..r]$.

Remarks:

- $A[q]$ is called "pivot"
- compared with merge sort, there is no merging operation

Quick Sort: graphical view



Quick Sort: pseudo-code

QuickSort(A, p, r)

if $p < r$ **then**
    $q =$ Partition(A, p, r)
    QuickSort($A, p, q - 1$)
    QuickSort($A, q + 1, r$)

$\star$ Initial call: QuickSort($A, 1, A.length$)

Partition function: main lines

Find a pivot in $A[p..r]$

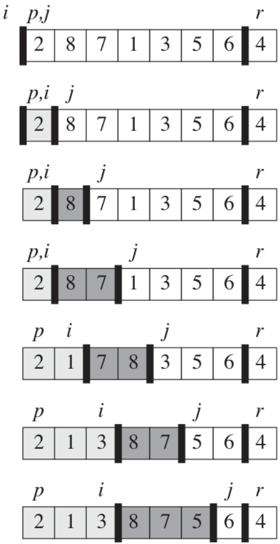
- Select last element $A[r]$ as a pivot
- Initialize $i = p - 1$
- Iterate with j from p to $r - 1$
- If $A[j]$ is smaller than the pivot $i = i + 1$ and swap $A[i]$ with $A[j]$
- Move the pivot index forward $i = i + 1$
- At the end of the loop swap $A[i]$ with $A[r]$
- The pivot index will be i

$\star$ elements smaller than the pivot are put in the front of the array

$\star$ when we reach the end of the loop, the pivot is put after the list of elements smaller than itself

$\star$ all remaining elements are bigger than the pivot!

Partition: illustration



- $A[r]$ is the pivot
- $A[p..i]$ contains elements $\leq$ than the pivot
- $A[i+1..j-1]$ contains elements $>$ than the pivot
- $A[j..r-1]$ is not yet examined

Partition Pseudo-Code

```
PARTITION(A, p, r)
    x = A[r]
    i = p - 1
    for j = p to r - 1
        if A[j] ≤ x
            i = i + 1
            swap(A[i], A[j])
    swap(A[i + 1], A[r])
    return i + 1
```

Is the Partition code correct?

Pre-condition: $A[p..r]$ is a table of numbers
Post-condition: $A[p..i] \leq A[i+1] < A[i+2..r]$
Invariant:

- The values $A[p..i]$ are $\leq$ than the pivot
- The values $A[i+1..j-1]$ are $>$ than the pivot
- $A[r]$=pivot

$\star$ easy to check that the invariant is preserved inside the loop.

Complete algorithm

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
        if A[j] ≤ x
                i = i + 1
                swap(A[i], A[j])
swap(A[i + 1], A[r])
return i + 1
```

```
QUICKSORT(A, p, r)
if p < r
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q - 1)
        QUICKSORT(A, q + 1, r)
```

Complexity of the Partition function

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
        if A[j] ≤ x
                i = i + 1
                swap(A[i], A[j])
swap(A[i + 1], A[r])
return i + 1
```

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

Complexity of QuickSort

```
QuickSort(A, p, r)
if p < r
    q = Partition(A, p, r)
    QuickSort(A, p, q − 1)
    QuickSort(A, q + 1, r)
```

⋆ Worst case:

- $q = p$ or $q = r$ (the array is not split in "equal halfs")
- Partitioning transforms a problem of size n into one of size $n - 1$

$$T(n) = T(n-1) + \Theta(n).$$

- Same complexity as insertion sort:

$$T(n) = \Theta(n^2).$$

Complexity of QuickSort

```
QuickSort(A, p, r)
if p < r
    q = Partition(A, p, r)
    QuickSort(A, p, q − 1)
    QuickSort(A, q + 1, r)
```

⋆ Best case:

- $q = n/2$ (the array is split in "equal halfs")
- Partitioning transforms a problem of size n into two problems of size $n/2$

$$T(n) = 2T(n/2) + \Theta(n).$$

- Same complexity as merge sort:

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

Average Complexity of QuickSort: intuition

- Average complexity correspondes to best case

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

- Intuitively:
  - On average we expect an alternance between "good" and "bad" partitionings
  - The complexity of a bad partitioning followed by a good one is the same as the complexity of a good partitioning directly.

Complexity of QuickSort: mathematically

- number of comparisons for partitioning: $n+1$
- probability that the pivot is at position k: $1/n$
- size of sub-arrays in that case: $k-1$ and $n-k$
- the sub-arrays may be sorted randomly

The *average* number of comparisons used by quick sort is given by the following recurrence:

$$C_1 = 0$$

$$C_n = n - 1 + \sum_{k=1}^{n} \frac{1}{n}(C_{k-1} + C_{n-k}), \text{ if } n > 1$$

Analytical formulas

$$C_n = n - 1 + \sum_{k=1}^{n} \frac{1}{n}(C_{k-1} + C_{n-k}), \text{ if } n > 1$$

and using symmetry:

$$C_n = n - 1 + \frac{2}{n} \sum_{k=1}^{n} C_{k-1}.$$

multiply by n:

$$nC_n = n(n-1) + 2\sum_{k=1}^{n} C_{k-1}$$

Subtracting the same formula for $n-1$:

$$nC_n - (n-1)C_n = 2(n-1) + 2C_{n-1}.$$

Regrouping:

$$nC_n = (n+1)C_{n-1} + 2(n-1).$$

$$C_n = 2(n+1)\sum_{k=1}^{n} \frac{1}{k} - 3(n+1).$$

For the harmonic series we have

$$\sum_{k=1}^{n} \frac{1}{k} \sim \ln n.$$

Finally:

$$C_n \sim 2n \ln n \in \Theta(n \log n).$$

- choosing the pivot differently: randomly (not the last), median of extremes: decreases drastically the chances of being in the worst case
- quicksort is slow for small tables
- Better use a simpler sort (insertion sort) for smaller tables ($n \leq k$)

Conclusion on QuickSort

- quick algorithm on average $\Theta(n \log n)$
- worst case $\Theta(n^2)$, but it is not likely if the pivot is well chosen
- Sorting in place
- Not stable (can change the order of equal elements in the original table)
- A bit quicker in practice than Merge-Sort

Up to this point...

Algorithm	Complexity			In place?
	Worst	Average	Best	
INSERTION-SORT	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
SELECTION-SORT	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
BUBBLE-SORT	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
MERGE-SORT	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	no
QUICK-SORT	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	yes

Heap Sort: Introduction

- invented by Williams in 1964
- based on a useful data structure: the heap
- complexity bounded by $\Theta(n \log n)$ (in all cases)
- sorting in place
- simple to implement

In the following:

- introduction to trees
- heaps
- heap sort

Selection sort: alternative point of view

$\star$ can be improved by using an appropriate data structure!
Variant of selection sort:

SELECTION-SORT2(A)

```
repeat the following recursively
for i = 1 to n do
    Find minimum of A
    Delete the minimum
```

$\star$ if A is an array finding the minimum costs $O(n)$ time.
$\star$ if A is represented using a priority queue or a heap then complexity drops from $O(n^2)$ to $O(n \log n)$
$\star$ heap sort is just a selection sort using the right data structure!

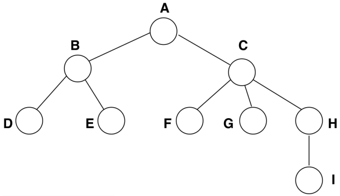
Trees: definition

⋆ **Definition**: A tree T is a directed graph (N, E) where:

- N is a set of nodes
- $E \subset N \times N$ is a set of arcs

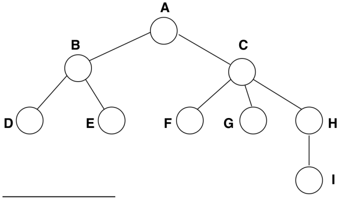
A tree has the following properties:

- T is connected and has no cycles
- if T is not void then it has a distinguished node called root. This root is unique.
- For every arc $(n_1, n_2) \in E$ the node n_1 is the parent of n_2.
    - The root of T does not have a parent
    - Every other node of T has one and only one parent

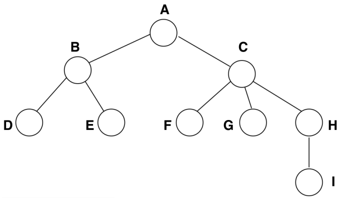


Trees: terminology

- If n_2 is the parent of n_1 then n_1 is the child of n_2
- Two nodes n_1, n_2 which have the same parent are siblings
- A node having at least a child is called internal
- An external node (not internal) is called a leaf
- A node n_2 is an ancestor of a node n_1 if n_2 is the parent of an ancestor of n_1
- n_2 is a descendant of n_1 if n_1 is an ancestor of n_2



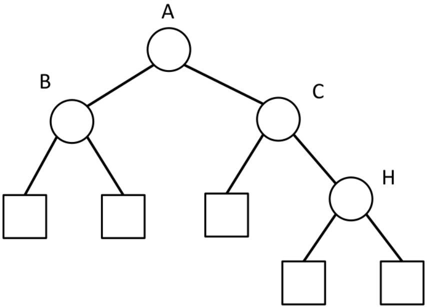
- A path is a sequence of nodes $n_1, n_2, ..., n_m$ such that for $i \in [1, m-1]$ (n_i, n_{i+1}) is an arc of the three; **Remark:** a path cannot connect two distinct leafs.
- The height of a node is the number of the longest path connected to a leaf. The height of the tree is the height of the root.
- The depth of a node is the number of arcs needed to connect it to the root



Binary trees

- An ordered tree is a tree in which the set of children for each one of the nodes is ordered
- A binary tree is an ordered tree with the following properties
  - Each node has at most two children
  - Each child is either a left or a right child
  - The left child is before the right child in the ordering
- A full binary tree is a binary tree in which every inner node has exactly two children
- A perfect binary tree is a full binary tree in which all leaves have the same depth

Properties for full binary trees



- Number of external nodes equal to number of inner nodes plus one
- The number of inner nodes is $(n-1)/2$
- Number of nodes at depth (or level) i is $\leq 2^i$
- The height h of the tree is less than the number of inner nodes
- The link between the number of nodes n and the height is

$$n \in \Omega(h); n \in O(2^h)$$

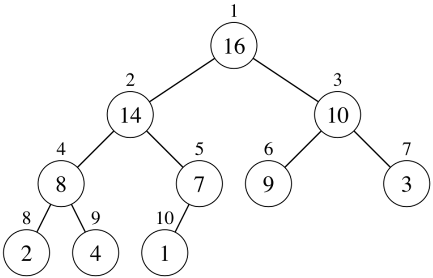
Heap: definition

A complete binary tree is a binary tree such that

- If h is the height of the tree:
  - For $i \leq h-1$ there are 2^i nodes at depth i
  - A leaf has depth h or $h-1$
  - All maximal depth leaves are on the left

A binary heap is a complete binary tree such that:

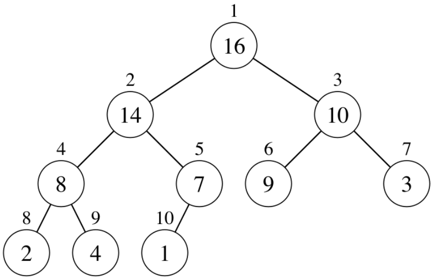
- To each node is assigned a key
- The key for a node is larger than the key of its children (order property for the heap)



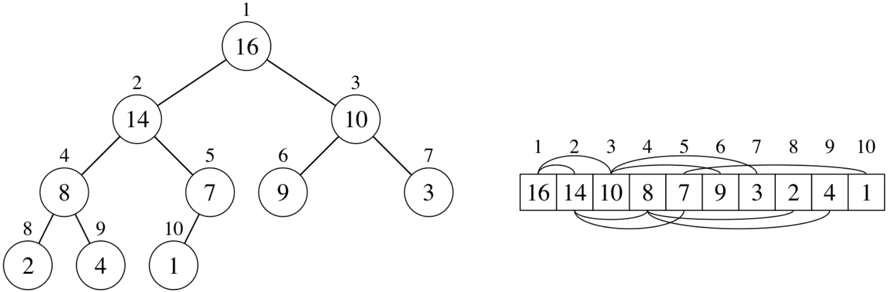
Properties for a heap

⋆ if T is a complete binary tree containing n nodes, having height h:

- $n \geq 2^{h-1}$, the height of the perfect tree of height $h - 1$ +1
- n is smaller than the size of the perfect tree with height h: $n \leq 2^{h+1} - 1$



Implementing a heap through an array



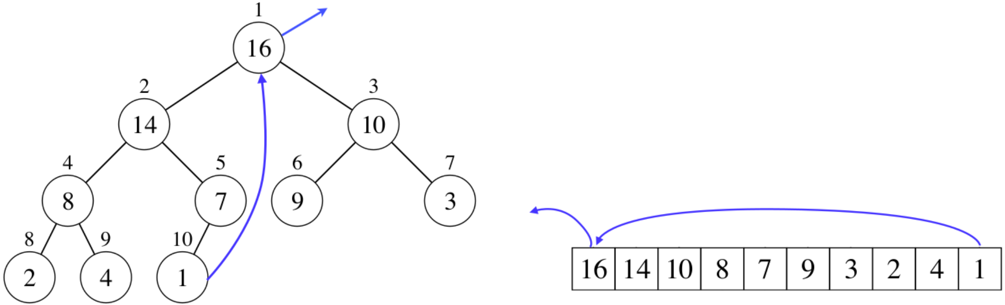
A heap can be represented in a compact way through an array A:

- The root is the first element of the array
- $\textsc{Parent}(i) = \lfloor i/2 \rfloor$
- $\textsc{Left}(i) = 2i$
- $\textsc{Right}(i) = 2i + 1$

Order property for a heap: for each i we have $A[\textsc{Parent}(i)] \geq A[i]$.

Principle of heap sort

- Build a heap from an array to be sorted: $\textsc{Build-max-heap}(A)$.
- While the heap contains elements:
    - Extract the root of the heap, put it in the sorted array; replace it by the right-most element
    - Re-establish the heap property, keeping in mind that the left and right sub-trees are heaps: $\textsc{Max-Heapify}(A, 1)$.



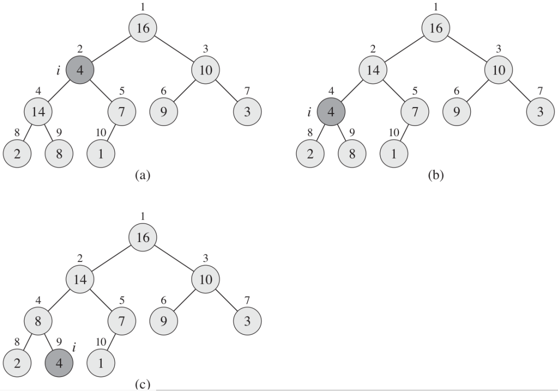
Everything is done in the original table: in place

Max-Heapify

⋆ Procedure Max-Heapify(A, i):

- Assume that the left sub-tree of node i is a heap
- Assume that the right sub-tree of node i is a heap
- Objective: rearrange the heap to maintain the order properties

⋆ Example: Max-Heapify$(A, 2)$



Max-Heapify: pseudo-code

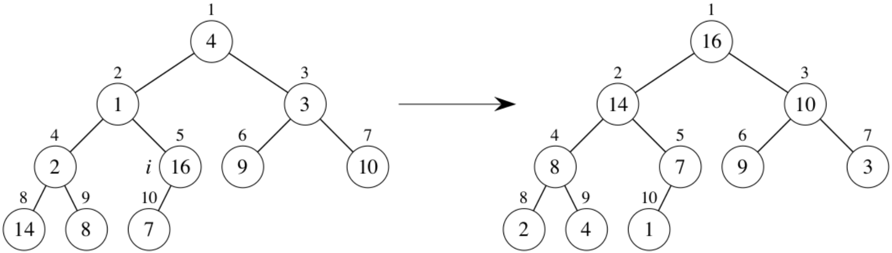
```
MAX-HEAPIFY(A, i)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ A.heap-size ∧ A[l] > A[i]
        largest = l
  else largest = i
  if r ≤ A.heap-size ∧ A[r] > A[largest]
        largest = r
  if largest ≠ i
        swap(A[i], A[largest])
        MAX-HEAPIFY(A, largest)
```

Complexity: $T(n) = O(\log n)$: height of the node

BUILD-MAX-HEAP(A)

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

Invariant: every node $i, i+1, ..., n$ is the root of a heap



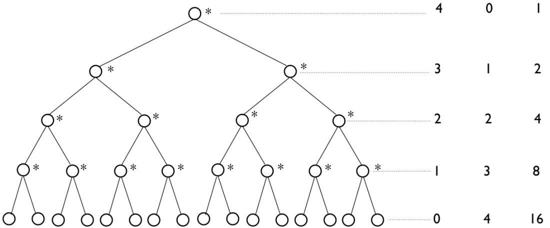
- The initial array is interpreted as a complete binary tree

Complexity of BUILD-MAX-HEAP

Simple bound: $O(n)$ calls to MAX-HEAPIFY each one being $O(\log n)$: $O(n \log n)$

Finer analysis:

- to simplify the analysis, assume the binary tree is perfect/complete
- $n = 2^{h+1} - 1$ for a given $h \geq 0$, the height of the resulting tree



- There are 2^i nodes at depth i
- We must call Max-Heapify on all of them
- Each call is at worst $\Theta(h-i)$
- Number of operations in terms of h:

$$T(h)=\sum_{i=0}^{h-1} 2^i\Theta(h-i)=\Theta(\sum_{i=0}^{h-1} 2^i(h-i))=\Theta(2^{h+1}-h-2).$$

- $T(n)\in\Theta(n)$.

Heap sort: algorithm

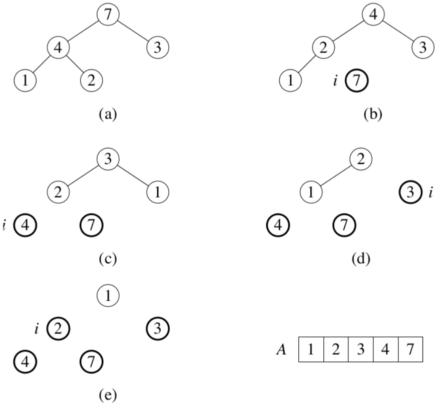
```
HEAP-SORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
        swap(A[i], A[1])
        A.heap-size = A.heap-size − 1
        MAX-HEAPIFY(A, 1)
```

Invariant:

$A[1..i]$ is a heap containing i elements smallest from $A[1..A.length]$

$A[i+1..A.length]$ has the $n-i$ largest elements in $A[1..A.length]$ sorted.

Initial array: $A = [7, 4, 3, 1, 2]$.



Complexity of Heap-Sort

```
Heap-Sort(A)
Build-Max-Heap(A)
for i = A.length downto 2
        swap(A[i], A[1])
        A.heap-size = A.heap-size − 1
        Max-Heapify(A, 1)
```

- Build-Max-Heap: $O(n)$
- **for** loop: $n-1$ times
- swapping elements $O(1)$
- Max-Heapify: $O(\log n)$

Total $O(n \log n)$ (worst and average cases)
Heap sort is generally beaten by quick sort.

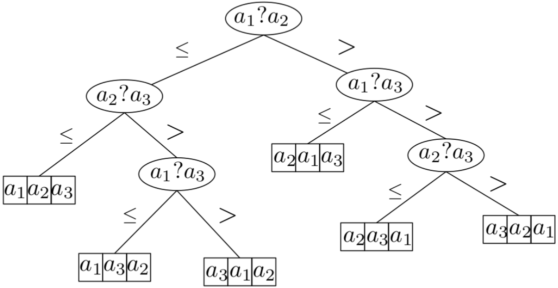
Algorithm		Complexity		In place?
	Worst	Average	Best	
INSERTION-SORT	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
SELECTION-SORT	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
BUBBLE-SORT	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
MERGE-SORT	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	no
QUICK-SORT	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$	yes
HEAP-SORT	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	yes

Can we do better than $O(n \log n)$?

- No, if we use *comparative* sorting
  - no hypothesis on elements to sort
  - we need to compare elements
- Complexity of a problem vs complexity of an algorithm?
- In our case a sorting algorithm is
  - a sequence of comparisons
  - a procedure for transforming a table into another one, which is sorted

Decision tree: example

An optimization algorithm = a binary decision tree



⋆ decision tree for sorting the table $[a_1, a_2, a_3]$

⋆ number of leaves: $n!$ = number of permutations of the original array

Decision tree: definition

Sorting algorithm = a binary decision tree

- a leaf of the tree: a permutation of the original table
- sorting: a path from the root to the leaf corresponding to a sorted table
- height of the tree: the worst case for the sorting algorithm
- shortest path: the best case for the sorting algorithm
- average height: average complexity of the sorting algorithm

Decision tree: properties

- A binary tree with height h has at most 2^h leaves
- The number of leaves in the decision tree is $n!$ (the number of permutations of the original table)
- We find

$$n! \leq 2^h.$$

- Stirling formula: $n! \sim (n/e)^n$.
- Therefore:

$$h \geq \log(n!) \sim n \log n - n = \Omega(n \log n)$$

Conclusion: We cannot do better than $n \log n$.

The comparative sorting **problem** is $\Omega(n \log n)$.

Conclusion: we have seen

- a categorization of the sorting algorithm
- QuickSort: in place, $\Theta(n \log n)$
- Analysis of the average case for an algorithm
- A first data structure: the heap
- HeapSort: in place, $\Theta(n \log n)$
- A lower bound for comparative sorting