Sorting Algorithms

Sorting involves arranging n numbers in a specific order. For instance, sorting the sequence $6, 2, 9, 4, 5, 1, 4, 3$ would yield $1, 2, 3, 4, 4, 5, 6, 9$. Sorted data makes many tasks quicker, from dictionary lookups to address book searches. Sorting is a fundamental operation in computer science, serving as a building block for more complex algorithms. Numerous sorting algorithms exist, including:

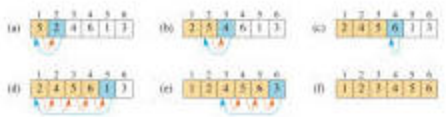
- Insertionsort
- Selectionsort
- Bubblesort
- Mergesort
- Quicksort
- Heapsort
- Radixsort
- Countingsort

These notes will cover the above algorithms. Unless specified otherwise, we will sort in non-decreasing order. To sort in descending order, simply reverse the comparisons. We assume the input is in a list $Python$ or an array $Java$, with elements composed of a sorting key $number$, $float$, $or string$ and additional information, represented here as integers.

Insertionsort

Insertionsort is similar to how one might sort a hand of cards. The algorithm works by iteratively inserting each element into its correct position within the already sorted portion of the array.

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



The image above illustrates the pseudocode for the Insertionsort algorithm, alongside a series of steps that visually demonstrate its operation on an array.

Analysis of Running Time for Insertionsort

The running time analysis of Insertionsort involves determining how many times the test in the inner while-loop is executed, denoted as t_i. Thus, $t_i - 1$ represents the number of times this loop runs, indicating how many elements the i-th element must pass during insertion. Note that $1 \leq t_i \leq i$.

Let $c = c_1 + c_2 + \cdots + c_8$ represent the sum of constant factors.

- Best Case: $t_i = 1$ for all i. The total time is $\leq c \cdot n$.
- Worst Case: $t_i = i$ for all i. The total time is $\leq c \cdot n^2$, since $\sum_{i=2}^{n} i \leq (1 + 2 + 3 + \cdots + n) = \frac{(n+1)n}{2} = \frac{n^2+n}{2} \leq \frac{2n^2}{2} = n^2$.

Selectionsort

Selectionsort is a straightforward sorting algorithm. The algorithm works as follows:

1. Take an input list
2. Create an empty list.
3. While the input list is not empty:
    - Find the smallest element x in the input list.
    - Move x from the input list to the end of the output list.

The algorithm is correct because each extracted element is less than or equal to all subsequent elements.

Running Time of Selection Sort

The algorithm finds the smallest element in the input list *n* times. Finding the smallest element can be done via linear search, which examines each remaining element.

Therefore, the total time is $\leq c \cdot (n + (n-1) + (n-2) + \cdots + 1) \leq c \cdot n^2$.

Merge

Merge combines two sorted sequences A and B into a single sorted sequence C. For example, merging $A = 2, 4, 5, 7, 8$ and $B = 1, 2, 3, 6$ yields $C = 1, 2, 2, 3, 4, 5, 6, 7, 8$.

Instead of simply sorting $A \cup B$, merging is faster.

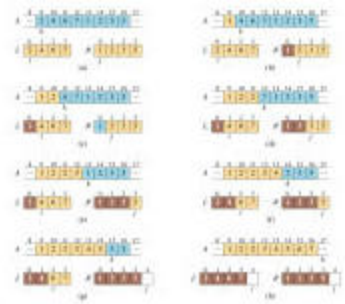
The algorithm repeatedly moves the smallest of the two front elements in A and B to the end of C. This takes $\leq c \cdot n$ time, where n is the total number of elements in A and B.

Correctness of Merge

Merge is a variant of Selectionsort that exploits the sorted nature of A and B. The smallest element in $theremainderof\ A \cup B$ can be found by only examining the front elements of A and B, taking constant time.

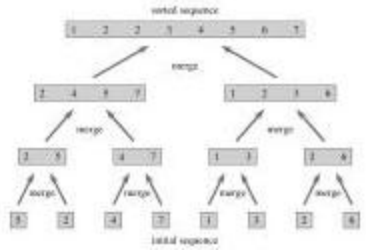
Pseudo-code for Merge

The following pseudo-code merges two input lists that are adjacent parts of the same list/array A, namely $A[p \ldots q]$ and $A[q+1 \ldots r]$. These are first copied into lists L and R.



Mergesort

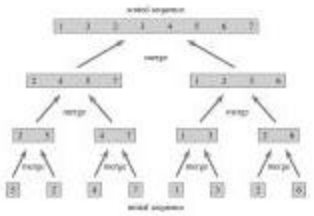
Mergesort builds longer sorted segments of the input by repeatedly using merge. The image below helps to illustrate the process.



Each merge uses at most $c \cdot n_1$ time, where n_1 is the number of elements being merged. All merge operations in a layer use at most $c \cdot (n_1 + n_2 + \ldots) = c \cdot n$. This holds for all layers. With a total of $\log_2 n$ layers, the overall time is at most $c \cdot n \cdot \log_2 n$.

Layers in Mergesort

After k merge layers, the number of sorted lists $assuming\$n\$isapowerof2$ can be described by the following image:



k	Antal lister
⋮	⋮
k	$n/2^k$
⋮	⋮
3	$n/2^3$
2	$n/2^2$
1	$n/2$
0	n

The algorithm stops when there is one sorted list. $n/2^k = 1 \Leftrightarrow n = 2^k \Leftrightarrow \log_2 n = k$

Mergesort with non-powers of 2

In each layer, the algorithm merges as many pairs as possible, leaving a single list unmerged if necessary $whichisthenincludedinthenextlayer$. For instance, 12 lists become $12/2 = 6$ lists, while 13 lists become $12/2 + 1 = 6 + 1 = 7$ lists.

If there are x lists before a merge layer, there are $\lceil x/2 \rceil$ lists after. Given two input sizes n and n', where $n \leq n'$, the number of lists in each layer cannot be smaller for n' than for n since $\lceil x/2 \rceil$ is an increasing function of x. Therefore, the number of layers cannot be smaller for n' than for n.

Set n' to the smallest power of two greater than or equal to n. There are exactly $\log_2 n'$ layers for n', and thus at most that many layers for n. Conversely, for $n = 2^k + 1$, there are $k + 1 = \lceil \log_2 n \rceil$ layers. So, there are $\lceil \log_2 n \rceil$ layers in general.

The table below shows a few examples of the relationship between input size n and the number of layers, in general:

n	7	8 = 2^3	9	10	11	12	13	14	15	16 = 2^4	17
$\log_2 n$	2.807	3	3.169	3.321	3.459	3.584	3.700	3.807	3.906	4	4.087
Number Layers	3	3	4	4	4	4	4	4	4	4	5

Mergesort Pseudo-code

Here is a recursive formulation of Mergesort. A call to `Merge-Sort(A, p, r)` sorts the elements in $A[p \ldots r]$. The initial call is `Merge-Sort(A, 1, n)`, which sorts the entire array A. A call to `Merge(A, p, q, r)` merges the two sorted subarrays/lists $A[p \ldots q]$ and $A[q + 1 \ldots r]$ into $A[p \ldots r]$.

Mergesort Example

Mergesort involves the following steps:

1. Split the input into two parts, X and Y $trivial$.
2. Sort each part recursively.
3. Merge the two sorted parts into one sorted part $realwork$.

The base case is $n \leq 1$, which is already sorted, so no action is needed.

Quicksort

Quicksort involves the following steps:

1. Divide the input into two parts X and Y such that $X \leq Y$ $realwork$.
2. Sort each part recursively.
3. Return X followed by Y $trivial$.

The base case is $n \leq 1$, which is already sorted, so no action is needed.

Quicksort Pseudo-code

A call to `Quicksort(A, p, r)` sorts the elements in $A[p \ldots r]$. The initial call is `Quicksort(A, 1, n)`, which sorts the entire array A. A call to `Partition(A, p, r)` selects an element $x \in A$ and divides $A[p \ldots r]$ such that:

$A[q] = x\ A[p \ldots q-1] \leq x\ A[q+1 \ldots r] > x$

Partition

To perform the partition, choose an element x from the input $here, the last element in the array$. Build the two parts during a scan of the array based on the following principle:

Partition Example

During the scan, the time complexity is $O(n)$, where n is the number of elements in $A[p \ldots r]$.

Partition Pseudo-code

The performance of Quicksort depends on how the partition divides the input during recursion.

Two extreme cases of recursive call sizes:

- **Unbalanced:** 0 and $n-1$
- **Balanced:** $\lceil (n-1)/2 \rceil$ and $\lfloor (n-1)/2 \rfloor$

If all partitions are balanced, the time complexity is $O(n \log n)$ $similar analysis to Mergesort$. If all partitions are unbalanced, the time complexity is $O(n + (n-1) + (n-2) + \cdots + 2 + 1) = O(n^2)$.

These represent the best-case and worst-case scenarios for Quicksort, respectively. In practice, Quicksort has a running time of $O(n \log n)$ for almost all inputs. However, note that already sorted input results in $\Theta(n^2)$ running time if the partitioning element x is chosen as the last element $as done in the book$. It is not recommended to choose the last element as the partitioning element in practice.

Suggestions for more robust choices of partitioning element x:

- Middle element
- Median of several elements
- Random element
- Median of several randomly chosen elements

Quicksort is *in-place*, meaning it does not require additional space beyond the input array. It is highly efficient in practice, and a well-implemented Quicksort is often the best all-around sorting algorithm, chosen in many libraries $e.g., Java and C++/STL$.

Heaps

A Heap is:

1. A binary tree
2. With heap order
3. And heap shape
4. Laid out in an array

Binary Tree

A binary tree is either:

- Empty
- A node v with two subtrees $right and left$

The node v is the *root* of the tree. If v has a non-empty subtree, the root u of that subtree is a *child* of v, and v is u's *parent*. If both of v's subtrees are empty, v is a *leaf*. The lines between children and parents are called *edges*. The parent/child concept generalizes naturally to ancestor and descendant.

- **Depth of Node:** Number of edges to root
- **Height of Node:** Max number of edges to leaf
- **Height of Tree:** Height of its root
- **Complete Binary Tree:** All layers are completely filled.

A complete binary tree of height h has $1 + 2 + 4 + 8 + \cdots + 2^h = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$ nodes, of which 2^h are leaves.

Heap Order

A binary tree with values in all nodes is max-heap-ordered if, for any node v with child u, the value in $v \geq$ value in u.

> Equivalently, for any node v with descendant u, the value in $v \geq$ value in u.

In a max-heap-ordered tree, the root contains the largest value in the entire heap. A tree is min-heap-ordered if the above holds with $\leq$ instead of $\geq$.

Heap Shape

A binary tree has heap shape if all layers are completely filled except the last layer, where all nodes are as far left as possible. $A complete tree has heap shape.$

For a tree of heap shape with height h and n nodes: $n >$ number of nodes in a complete tree of height $h - 1 = 2^h - 1$

$n > 2^h - 1 \Leftrightarrow n + 1 > 2^h \Leftrightarrow \log_2(n + 1) > h$

Heap Layout in Array

A binary tree in heap shape can be naturally laid out in an array by assigning array indices to nodes in a top-down, left-to-right traversal of the tree's layers.

Navigation between children and parents in the array version can be done with simple calculations: The node at position i has:

- Parent at position $\lfloor i/2 \rfloor$
- Children at positions $2i$ and $2i + 1$

Heap Operations

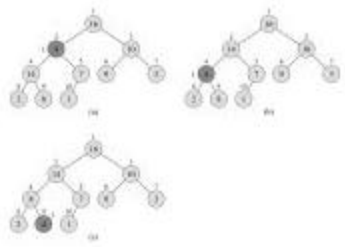
Operations include:

- **Max-Heapify:** Given a node with two subtrees, each obeying max-heap order, make the entire node's subtree obey max-heap order.
- **Build-Max-Heap:** Turn n input elements $unordered$ into a heap.

Max-Heapify

Given a node with two subtrees, each obeying max-heap order, make the entire node's tree obey max-heap order.

Problem: The node's value is smaller than one or both of its children's values.
Solution: Swap with the child with the largest value, then run Max-Heapify on that child.



The image above shows three binary tree diagrams and exemplifies how to maintain the **max-heap** property by moving larger elements up the tree.

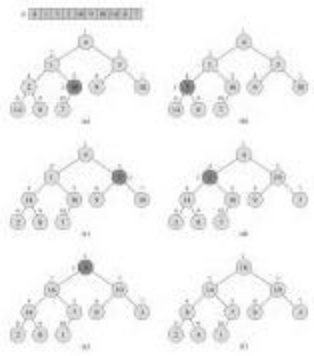
The time complexity is $O(\text{height of node})$.

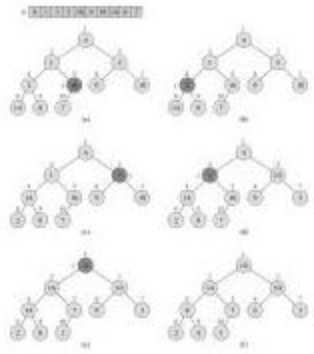
Max-Heapify Pseudo-code

The following pseudo-code includes a check to prevent the algorithm from looking "too far" in the array $i.e., beyondposition\$n\$$.

Build-Heap

Turn n input elements $unordered$ into a heap. Arrange the elements in heap shape, then bring the tree into heap order from the bottom up. A tree of size one always obeys heap order.



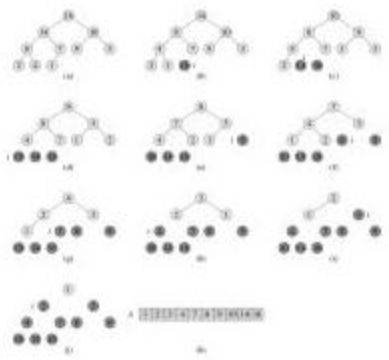


Build-Heap Pseudo-code

Heapsort

Heapsort is a selection sort variant that uses a heap to extract the largest remaining element.

1. Build a heap.
2. Repeat while the heap is not empty:
    - Extract the root $largestelementintheheap$.
    - Set the last element as the new root.
    - Restore heap structure using Max-Heapify on the new root.



Heapsort Pseudo-code

The time complexity is $O(n) + O(n \log n) = O(n \log n)$.

Comparison of $n \log n$ Sorting Algorithms

The table below compares Quicksort, Mergesort, and Heapsort with respect to worst-case time complexity and whether the algorithm is in-place:

Algorithm	Worst-Case	In-Place
Quicksort	√	√
Mergesort	√	
Heapsort	√	√

Heapsort runs slower than Mergesort and Quicksort due to inefficient memory usage $randomaccess$.

Introsort combines Quicksort and Heapsort. It starts with Quicksort but switches to Heapsort if the recursion becomes too deep, resulting in an in-place, worst-case $O(n \log n)$ algorithm with good practical running time. This is the sorting algorithm used in the C++ STL standard library.