# 9. Sorting

Please read Part II Sorting and Order Statistics in (Corman et al., 2009).

## Ineffective Sorts

**Problem 4: Sorting**

**Decision Problem:** Given a list of keys, is it sorted?

**Function Problem:** Given a list of keys, sort it into ascending (or descending) order.

Pretend you are given a file of records containing keys that can be ordered. That is, the keys are from a totally ordered set where the
common relations and functions

\(<\), \(\leq\), \(=\), \(\geq\), \(>\), \(\neq\), \(\text{min}\), \(\text{max}\)

are defined on the keys.

The function \texttt{sorted} witnesses that a list is sorted in ascending order or not.

\textbf{Listing 23: The Sorted Decision Problem}

\begin{verbatim}
  (Sorted list or not? 106)
  sorted :: (Ord a) => [a] -> Bool
  sorted [] = True
  sorted [x] = True
  sorted (x:y:ys) = (x <= y) && sorted (y:ys)
\end{verbatim}

The sorted function has time complexity \(O(n)\) and this shows the Sorted can be decided in polynomial time. That is, Sorted is in NP, the class of decision problems that can be solved in polynomial time using a non-deterministic Turing machine. Of course, the function problem: Sort this list, can be solved in polynomial time and is in P,

Two factors dominate the time spent sorting by comparing keys:

1. The number of comparisons made, and
2. The amount of data moved

When an algorithm requires duplicating the records, space complexity can become an issue too.

- Some comparison-based sorting algorithms have \(O(n^2)\) worst case time complexity. For example: bubble, insertion, and selection sorts.
- Other comparison-based sorting algorithms have \(O(n \lg n)\) worst case time complexity. For example: merge and heap sorts.
- Quicksort is famous for almost always being fastest, but in some races it does not win.

\textit{Sentinels:} To terminate a sort, some algorithms benefit from \textit{sentinels} at an ends of records \((A[0] \text{ or } A[n + 1])\). Sentinels are typically below or above every valid value.

\textit{Comparison sorts:} The most common sorts require comparing keys. The lower bound time complexity for comparison sorts is \(\Omega(n \lg n)\).

\textit{Sorts without Compares:} Time complexity can be reduced when data properties are known. There are several algorithms that sort in \(O(n)\) time without comparing keys. For example, counting, radix, and bin sorts.
**Internal and External Sorts:** Sorting algorithms can also be classified by the size of the file to be sorted. Internal sorting processes files that fit into main memory. External sorting processes files too large to fit in main memory. The files are stored in external memory: magnetic tapes, disk, or on a network (in the cloud).

**Stability:** A sorting algorithm is *stable* if it preserves relative order of equal keys. For example, if an alphabetized file of names is sorted by salary, those names with the same salary will remain in alphabetical order.

**Sorting Algorithms**

The following algorithms are presented and analyzed in these notes.

- Bubble Sort
- Insertion Sort
- Selection Sort
- Shell Sort
- Merge Sort
- Quicksort
- Heap Sort
- Counting Sort
- Radix Sort
- Bucket (Bin) Sort

Some nice demonstrations of several sorting algorithms can be found [at this external site](#). Know thy complexities! Below is a list of sorting algorithms and their complexities. I found it [here](#). The value $n$ is the length of the list (number of keys in the file) to be sorted.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quicksort</td>
<td>$O(n \log(n))$</td>
<td>$O(n \log(n))$</td>
<td>$O(n^2)$</td>
<td>$O(\log(n))$</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log(n))$</td>
<td>$O(n \log(n))$</td>
<td>$O(n \log(n))$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Timsort</td>
<td>$O(n)$</td>
<td>$O(n \log(n))$</td>
<td>$O(n \log(n))$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$O(n \log(n))$</td>
<td>$O(n \log(n))$</td>
<td>$O(n \log(n))$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Shell Sort</td>
<td>$O(n)$</td>
<td>$O((n \log(n))^2)$</td>
<td>$O((n \log(n))^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Bucket Sort</td>
<td>$O(n + k)$</td>
<td>$O(n + k)$</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Radix Sort</td>
<td>$O(nk)$</td>
<td>$O(nk)$</td>
<td>$O(nk)$</td>
<td>$O(n + k)$</td>
</tr>
</tbody>
</table>

**Bubble Sort**

The heuristic is: Repeatedly pass through the records exchanging adjacent elements that are out of order. When no exchanges are needed
the list is sorted. The insight for bubblesort is the bubble function that maps a list into a tuple containing a bubbled list and a Boolean flag indicating whether an exchange was made or not.

**Bubble**

Before learning to sort, learn to bubble. The bubble function will move the smallest value in a list to the head.

### Listing 24: Functional Bubbling

A pattern for defining a function is shown in the code outline.

\[
\begin{align*}
\langle & \text{Bubble smallest to head} \rangle \\
\langle & \text{Define the function’s type} \rangle \\
\langle & \text{Define base computations} \rangle \\
\langle & \text{Define the recursion} \rangle
\end{align*}
\]

The bubble function maps a list \([a]\) of values from a totally ordered set to an ordered pair: A permutation of the input list and a Boolean flag that signals if the input was changed or not. Not explicit in the type declaration is the post condition: The smallest value in the input list is the head of the output list.

\[
\begin{align*}
\langle & \text{Define the function’s type} \rangle \\
\langle & \text{Define base computations} \rangle \\
\langle & \text{Define the recursion} \rangle
\end{align*}
\]

As base cases, the empty list and a singleton are sorted and need no bubbling or changes to the list.

\[
\begin{align*}
\langle & \text{Define the function’s type} \rangle \\
\langle & \text{Define base computations} \rangle \\
\langle & \text{Define the recursion} \rangle
\end{align*}
\]

Otherwise, on a longer list \((x:xs)\), there are two objectives: Permute the input, if necessary, so its smallest value becomes the head of the list. And, record whether or not the input list was changed.

To bubble the smallest value to the head in a list \((x:xs)\), bubble the tail \(xs\) returning a permuted list \((y:ys)\) where \(y\) is the smallest value in \(xs\). If the permutation is the identity, the changed flag is False, otherwise \(\text{changed} = \text{True}\).

Now that \(x\) and \(y\) are the two smallest values in \((x:xs)\), they can be ordered, setting and returning the changed flag appropriately.

\[
\begin{align*}
\langle & \text{Define the function’s type} \rangle \\
\langle & \text{Define base computations} \rangle \\
\langle & \text{Define the recursion} \rangle
\end{align*}
\]

It may or may not be clear that the time complexity of bubble is

Inductive and equational reasoning is useful here. Trace the executions

- bubble \([2, 5]\) =
  - let \((5, False) = \text{bubble}[5]\)
  - in if \(2 > 5\)
  - then \((2, 5), False\)

- bubble \([7, 2, 5]\) =
  - let \((2, 5), False = \text{bubble}[2, 5]\)
  - in if \(7 > 2\)
  - else \((2, 7, 5), \text{True}\)

- bubble \([4, 7, 2, 5]\) =
  - let \((2, 7, 5), \text{True} = \text{bubble}[7, 2, 5]\)
  - in if \(4 > 2\)
  - then \((2, 4, 7, 5), \text{True}\)

Convince yourself: One call to bubble \(zs\) places the smallest value in \(zs\) at the head of the list.

A second call to bubble \(zs\) places the next smallest value in \(zs\) in the second position of the list.

Each subsequence call places another value in its correct position.
The time complexity of bubble can be described by the recurrence

\[ T(n) = T(n-1) + c, \quad T(0) = c \]

Which has solution

\[ T(n) = c(n + 1) = O(n) \]

Now bubble can be used repeated to sort a list.

### Listing 25: Functional Bubble Sort

```haskell
bubble_sort :: Ord a => [a] -> [a]
bubble_sort xs = 
  let (zs, changed) = bubble xs 
  in if changed 
     then bubble_sort zs 
     else zs
```

Each call to bubble places at least one element where it belongs. Therefore, `bubble_sort` will be called at most \( n \) times on a list of length \( n \).

- In the best case, `bubble_sort xs` calls `bubble xs` which returns `(zs, False)`, and `bubble_sort` immediately returns `zs`. In this case the time complexity is \( T(n) = c(n + 1) \).

- In the worst case, `bubble xs` returns `(zs, True)` each of \( n - 1 \) times and `(zs, False)` on the last pass. In this case, the time cost is \( T(n) = cn(n + 1) = O(n^2) \).

Here is an imperative implementation of bubble sort. It is naive in that it does not halt once it determines the array is sorted.

### Listing 26: Imperative Bubble Sort

```c
#include <stdio.h>

void bubblesort(int A[], int n) 
{
    int tmp; // for swapping
    for (int i = n-1; i > 0; i--) 
    { 
```
for (int j = 1; j <= i; j++) {
    if (A[j-1] > A[j]) {
        tmp = A[j-1];
        A[j] = tmp;
    }
}

int main () {
    int A[10] = {21, 8, 13, 55, 34, 5, 3, 2, 0, 1};
    bubblesort(A, 10);
    for (int i = 0; i < 10; i++) {
        printf("A[%d] = %d \n", i, A[i]);
    }
}

**Bubble Sort – Analysis of Complexity**

- Bubble sort uses about $n^2/2$ compares and $n^2/2$ data exchanges in the worst and average cases.

- The comparison $A[j-1] > A[j]$ is always executed inside the `for` loops on $i$ and $j$. The cost of the compares can be calculated using summation notation

\[
\sum_{i=1}^{n-1} \sum_{j=1}^{i} = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \binom{n}{2}
\]

- Bubble sort always makes $O(n^2)$ compares.

- In the worst case for swaps, the file is in reverse order, and $O(n^2)$ swaps are required.

- In the best case for swaps, the file is in sorted order, and no swaps are required.

- In the average case for swaps, we need the probability that the `if` test evaluates to True.
Example: Bubble sort operations

Consider the six permutations of \( \{2, 4, 7\} \). Pretend each permutation of the keys occurs with equal likelihood \( \frac{1}{6} \).

\[
\begin{array}{c|c|c|c|c|c}
\text{order} & \text{swaps} & \text{reorder} & \text{swaps} & \text{reorder} & \text{swaps} \\
2 & 5 & 7 & 0 & 0 & 0 \\
2 & 7 & 5 & 0 & 0 & 1 \\
5 & 2 & 7 & 0 & 0 & 0 \\
5 & 7 & 2 & 1 & 2 & 5 \\
7 & 2 & 5 & 1 & 2 & 5 \\
7 & 5 & 2 & 1 & 2 & 5 \\
\end{array}
\]

\[
P(\text{swap}) = \frac{1}{2} \quad P(\text{swap}) = \frac{2}{3} \quad P(\text{swap}) = \frac{1}{4}
\]

There are \((3+4+2) = 9\) swaps for the 6 cases. The average number of swaps is

\[
\frac{9}{6} = \frac{3}{2} = \frac{3(3-1)}{4} = \frac{n(n-1)}{4}
\]

Let’s see if this example provides the general rule.

- On the first pass of the outer loop \( i = (n-1) \) and \( j = 1, \ldots, (n-1) \). The if test will be True at any given \( 1 \leq j \leq i \) if and only if

  \[
  \max\{A[0], A[1], \ldots, A[j-1]\} > A[j]
  \]

- This will be True if the largest value from the set

  \[
  \{A[0], A[1], \ldots, A[j]\}
  \]

  is in any of the first \( j \) of \( j+1 \) positions

- Therefore, the probability that the if test evaluates to True, during the first pass on \( i=n-1 \) for \( j = 1, 2, \ldots, (n-1) \) is

  \[
  \frac{j}{j+1} = \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \ldots, \frac{n-1}{n}
  \]

- On the next pass of the outer loop \( i = (n-2) \) and the if test will be True if and only if

  \[
  \max\{A[0], A[1], \ldots, A[j-1]\} > A[j]
  \]

  and

  \[
  \max\{A'[0], A'[1], \ldots, A'[j]\} > A'[j+1]
  \]

where the primed values refer to the original array values.
• The probability that the if test evaluates to True on this second pass is
\[
\frac{j + 1}{j + 2} \cdot \frac{j}{j + 1} = \frac{j}{j + 2}
\]
\[
= \frac{1}{3}, \frac{2}{4}, \frac{3}{5}, \ldots, \frac{n - 2}{n} \quad \text{for } j = 1, \ldots, n - 2
\]

• In general, the probability that the if test evaluates to True on the \(k\)th pass is
\[
j = 1, 2, \ldots, n - k
\]

• Average case cost is computed by the formula
\[
\sum \text{Prob(case)} \cdot \text{Work(case)}
\]

• The average number of swaps is
\[
\sum_{i=1}^{n-1} \sum_{j=1}^{i} \frac{j}{j + (n - i)} = \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{n-1}{n} \quad \text{for } i = n - 1, j = 1, \ldots, (n - 1)
\]
\[
+ \frac{n-2}{n} \quad \text{for } i = n - 2, j = 1, \ldots, (n - 2)
\]
\[
+ \frac{n-3}{n} \quad \text{for } i = n - 3, j = 1, \ldots, (n - 3)
\]
\[
\vdots
\]
\[
+ \frac{1}{n-1} + \frac{2}{n} \quad \text{for } i = 2, j = 1, 2
\]
\[
+ \frac{2}{n} \quad \text{for } i = 1, j = 1
\]
\[
= \frac{1}{2} + \frac{1}{3} \left(1 + 2\right) + \frac{1}{4} \left(1 + 2 + 3\right) + \ldots + \frac{1}{n} \left(1 + 2 + 3 + \cdots + (n - 1)\right)
\]
\[
= \frac{1}{2} + \frac{1}{3} \left(\frac{2 \cdot 3}{2}\right) + \frac{1}{4} \left(\frac{3 \cdot 4}{2}\right) + \ldots + \frac{1}{n} \left(\frac{n(n-1)}{2}\right)
\]
\[
= \frac{1}{2} (1 + 2 + 3 + \cdots + (n - 1))
\]
\[
= \frac{n(n - 1)}{4}
\]

**Exercises**

1. Consider the insertion sort algorithm below. Assume \(A[0]\) is a
   *sentinel*, smaller every other element in the array.
(a) Compute the average number of comparisons (executions of the while loop) on the 6 permutations of \(\{0, 2, 5, 7\}\), where 0 is held fixed as the first element, the sentinel. Construct a table, similar to the table in example to show your computations.

(b) Find a general formula for the average number of comparisons in the insertion sort algorithm.
Bibliography


Corman, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, third edition. [page 9], [page 14], [page 18], [page 21], [page 31], [page 33], [page 39], [page 53], [page 93], [page 105], [page 135], [page 141], [page 163], [page 169], [page 178], [page 187], [page 193], [page 201]


# Index

**Haskell** programming language, 20, 31, 44, 116, 137  
**Java** programming language, 31  

Ackermann function, 174  
Axioms  
  Peano, 192  

Bézout, Étienne, 47  

C programming language, 30, 31, 42, 43, 55, 109, 152, 153  
Calendar, 7  
Cartesian product, 142  
Catalan numbers, 156, 161  
Computability, 191  
Computable function, 28  
Computer Language Benchmark Game, 30  
Contradiction, 193  

Dijkstra’s algorithm, 183  
DNA, 146  

Empty  
  String λ, 61  

Fibonacci, 44, 46, 142  
Floating point  
  Machine epsilon, 50  
Function  
  Ackermann, 174  
Functional programming, 31  

Gödel, Kurt, 192  
Golden ratio, 142  
Greatest common divisor, 47  
Gutenberg Project, 84  

Hash Tables, 93  
Hofstadter, Douglas, 25  

Kleene, Stephen, 53  
Knuth, Donald, 30  

Lamé, Gabriël, 44  
λ calculus, 26, 28  
Linear congruence, 47  
Listing  
  C  
    Matrix Chain Multiplication, 158  
Listings  
  C  
    A Minimum Algorithm, 135  
    Aho-Corasick, 89  
    Boyer-Moore Pattern Matching, 72  
    Brute-force Left-to-Right Pattern Matching, 56  
    Bubble Sort, 109  
    Build a Heap, 126  
    Counting Sort, 128  
    Direct Address Table Operations, 94  
    Edit Distance, 152  
    Find Set with Path Compression, 177  
    Greatest Common Divisor, 42  
    Hash Table with Chaining, 95  
    Heap Sort, 127  
    Heapify a List, 126  
    Inner Product, 155  
    Insertion Sort, 112  
    Knuth–Morris–Pratt Pattern Matching, 67  
    Maximum Subsequence Sum (Cubic), 34  
    Maximum Subsequence Sum (Linear), 35  
    Merge Sort, 118  
    Morris–Pratt Pattern Matching, 58  
    Partition about a pivot, 120  
    Quick Sort, 120  
    Radix Exchange Sort, 132  
    Radix Sort, 129  
    Randomized Partition, 136  
    Randomized Selection, 138  
    Rational Knapsack, 167  
    Selection Sort, 114  
    Shell Sort, 116  
    Union by rank, 177  

Haskell  
  Ackermann function, 174  
  Apply a Function Repeatedly, 50  
  Brute-force Left-to-Right Pattern Matching, 57  
  Bubble Function, 108  
  Bubble Sort, 109  
  Extended Euclidean Algorithm, 48  
  Fold from the left, 37  
  Greatest Common Divisor, 44  
  Inner Product, 155  
  Insertion Sort, 112  
  Matrix Multiplication, 155  
  Maximum Subsequence Sum, 37  
  Merge Sort, 117  
  Merging Sorted Lists, 117  
  Newton’s Square Root Method, 51  
  Newton’s Square Root Recurrence, 50  
  Quicksort, 120  
  Randomized Partition, 137  
  Relative Error Convergence Test, 50  
  Sample Code Header, 31  
  Second of a pair, 36  
  Selection Sort, 114  
  Sorted Decision Problem, 106
Splitting a List, 117
Swapping head with Another Element, 137
Pseudocode
Activity Selection, 170
Bucket Sort, 130
Dijkstra’s Algorithm, 186
Kruskal Minimum Spanning Tree, 172
Prim Minimal Spanning Tree, 181
Lucas, Édouard, 44
Memorization, 141
Metric space, 152
Newton’s method, 13
Newton, Issac, 49
Noweb, 30, 55
Object-oriented programming, 31
Pattern Matching, 53, 54
Boyer-Moore, 71
Brute Force, 55
Knuth–Morris–Pratt, 67
Morris-Pratt, 58
Right-to-Left Brute Force, 70
φ, 142
Presburger arithmetic, 191
RAM, 26, 29
Sequence, 19
Lucas, 44
Sets
Cartesian product, 142
Simpson’s rule, 51
Sorting Algorithms
Bubble Sort, 107
Bucket Sort, 129
Counting Sort, 128
Heap Sort, 124
Insertion Sort, 111
Merge Sort, 116
Quick Sort, 119
Radix Sort, 129
Selection Sort, 114
Shell Sort, 115
Time complexity, 127
String
Border, 61
Period, 61
Prefix, 61
Suffix, 61
Syllabus, 5

Theorems
Bezout’s, 47
Euclidean division, 41
Lamé’s, 45
Taylor’s, 49, 51
Turing Machine, 9, 26, 28, 192
Wikipedia, 201

xkcd
Algorithms by Complexity, 25
Calendar of meaningful dates, 7
Haskell, 30
Ineffective Sorts, 105
Traveling Salesman Problem, 141