Introduction to Algorithms
3rd Edition

—

Thomas H. Cormen, Charles E. Leiserson,
Ronald L. Rivest, & Clifford Stein

Graham Strickland

July 5, 2025

1 The Role of Algorithms in Computing

1.1 Algorithms

1.1-1

Suppose we need to find the smallest number of buildings in a suburb of a city which need security huts for surveillance of vehicles entering the suburb. We can model this problem by trying to find the convex hull, where buildings are vertices are the n points in the plane.

1.1-2

Space required for data storage, since memory is finite in any computing system executing an algorithm.

1.1-3

Linked lists have the following strengths:

- They are easy to insert items into.
- They are easy to remove items from.
- They do not require a large block of contiguous memore to be allocated ahead of insertion.

They have the following limitations:

- Traversing the linked list is slow.

- Accessing an element is not easy and requires indirect methods.
- Deallocating the memory used is slow.

1.1-4

The traveling-salesman problem can be interpreted as a series of many different shortest-path problems which are combined into one. However, the shortest-path problem does not require that whoever traverses the path return to where they started. It can also be solved efficiently, whereas the traveling-salesman problem cannot.

1.1-5

If we are tring to find the optimal layout of train routes so that trains do not cross tracks at the same time, then only the best solution will do. On the other hand, if we are trying to minimize the cost of procuring parts for the train, then an approximation to the best solution may be good enough.

1.1 Algorithms as a technology

1.2-1

A Google search uses algorithmic content to find matches to a seqrch query in the shortest time possible. The algorithms used must find the shortest path between various networks of web pages as well as parse through large amounts of text data in the shortest time possible.

1.2-2

We are looking for the interval where

$$\begin{aligned} 8n^2 &< 64n \lg n \\ \Rightarrow n &< 8 \lg n \\ \Rightarrow \frac{n}{\lg n} &< 8 \\ \Rightarrow -n \lg n &< 8 \\ \Rightarrow n \lg n &\geq 8. \end{aligned}$$

For $n < 44$, we have insertion sort faster than merge sort.

1.2-3

We are looking for the interval where $100n^2 < 2^n$, thus for $n = 15$, n^2 runs faster than 2^n.

Problems

1-1

From the program in `rs/compruntime`, we have the following output:

```
----------------------------------------------------------
f(n)            1 s             1m              1h
----------------------------------------------------------
lg(n)               inf             inf             inf
sqrt(n)  1.00000000e+12  3.60000000e+15  1.29600000e+19
n               1000000        60000000  3.60000000e+09
nlg(n)            62746         2801417       133378058
n^2                1000            7745           60000
n^3                  99             391            1532
2^n                  19              25              31
n!                   10              12              13
----------------------------------------------------------
--------------------------------------------------------------
      1d              1m              1y              1c
--------------------------------------------------------------
           inf             inf             inf             inf
7.46496000e+21  6.71846400e+24  9.94519296e+26  9.94519296e+30
8.64000000e+10  2.59200000e+12  3.15360000e+13  3.15360000e+15
2.75514751e+09  7.18708564e+10  7.97633893e+11  6.86109568e+13
        293938         1609968         5615692        56156922
          4420           13736           31593          146645
            36              41              44              51
            14              16              17              18
--------------------------------------------------------------
```

In generating the above results, we have made use of the following two Rust functions:

```
pub fn inverse_nlogn(x: f64) -> f64 {
    let max_iters = 10;
    let mut a_0 = x / (x.log2());
    let mut a_1: f64 = 0.0;

    for _ in 0..max_iters {
        a_1 = a_0 - (a_0 * a_0.log2() - x) / ((1.0 / LN_2) + a_0.log2());
        if ((a_1 * a_1.log2()) - (a_0 * a_0.log2())).abs() < 1.0 {
            return a_1.floor();
        } else {
            a_0 = a_1;
        }
    }

    a_1.floor()
}
```

and

```
pub fn inverse_factorial(x: f64) -> f64 {
    let mut i = 0.0;
    let mut fact = 1.0;

    while fact < x {
        if i > 0.0 {
            fact *= i;
        }
        i += 1.0;
    }

    i - 1.0
}
```

`inverse_nlogn` makes use of Newton's method to calculate an approximate value $m(x)$ s.t.

$$n \lg n = x \Rightarrow m(x) \approx n,$$

while `inverse_factorial` generates an approximate value $m(x)$ s.t.

$$x = n! \Rightarrow m(x) = \lfloor n \rfloor.$$

2 Getting Started

2.1 Insertion sort

2.1-1

In Figure 1, we illustrate the operation of Insertion-Sort on the array $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

2.1-2

```
Insertion-Sort($A$)
1: for $j = 2$ to $A.length$ do
2:     $key = A[j]$
3:     // Insert $A[j]$ to the sorted
           sequence $A[1\,..\,j-1]$
4:     $i = j - 1$
5:     while $i > 0$ and $A[i] < key$ do
6:         $A[i+1] = A[i]$
7:         $i = i - 1$
8:     $A[i+1] = key$
```

The implementation can be seen in the following file: `src/algorithms/sorting/insertion_sort.h`.

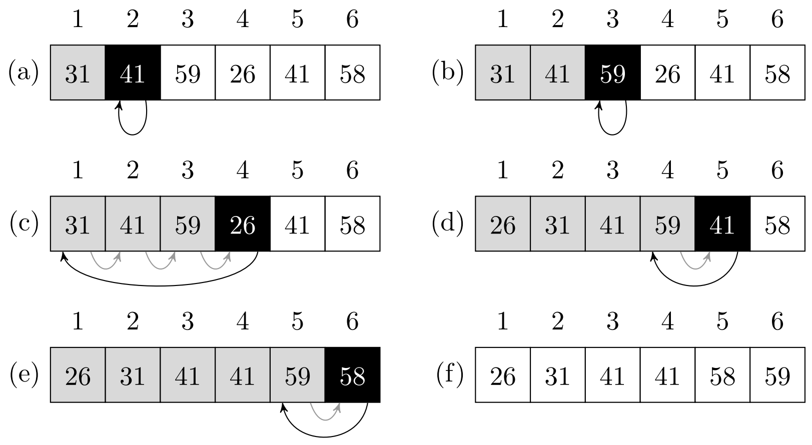

Figure 1: Illustration of INSERTION-SORT.

2.1-3

```
LINEAR-SEARCH($A, \nu$)
1: $j = 1$
2: while $j \neq A.length$ do
3:     if $A[j] == \nu$ then
4:         return $j$
5:     else
6:         $j = j + 1$
7: return NIL
```

Initialization: With the loop invariant being that $A[j]$ refers to an element of the sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$ and each element in $A[1 \mathinner{..} j-1]$ has been checked for equality, we have the loop invariant valid, since $j = 1$ and $A[1]$ is in A for $n \geq 1$, at the start of the **while** loop 1 – 6.

Maintenance: Either the **if** statement in line 3 returns $A[j]$ or the **else** statement in line 5 increments j by 1, so that after each pass of the **while** loop, we have checked whether or not $A[j] = \nu$, and the condition of the **while** loop ensures that $A[j]$ is in A.

Termination: If the **while** loop terminates, then either the **if** condition in line 3 is true, so that $A[j] = \nu$ or $j = A.length$, at which point we return the special value NIL.

Thus the algorithm is correct, since the loop invariant is initialized and maintained throughout, and the algorithm terminates with the correct output; if a value is returned, ν is in A, and occurs at $A[j]$ for return value j, otherwise the special value NIL was returned, and ν is not in A.

The implementation can be seen in the following file:
`src/algorithms/search/linear_search.h`.

2.1-4

We have the following addition problem:

Input: Two n-element arrays A and B containing binary digits.

Output: $(n+1)$-element array C containing the sum of A and B.

We have the following algorithm as a solution:

```
BINARY-ADDITION($A, B, n$)
1:  $carry = 0$
2:  for $i = 1$ to $n$ do
3:      if $A[i] = 1$ and $B[i] = 1$ then
4:          if $carry == 0$ then
5:              $C[i] = 0$
6:              $carry = 1$
7:          else
8:              $C[i] = 1$
9:              $carry = 1$
10:     else if $A[i] == 1$ and $B[i] == 0$ or
                $A[i] == 0$ and $B[i] == 1$ then
11:         if $carry == 0$ then
12:             $C[i] = 1$
13:         else
14:             $C[i] = 0$
15:             $carry = 1$
16:     else
17:         $C[i] = carry$
18:         $carry = 0$
19: $C[n+1] = carry$
20: return $C$
```

The implementation can be seen in the following file:
`src/algorithms/binary/binary_addtion.h`.

2.2 Analyzing algorithms

2.2-1

$$\frac{n^3}{1000} - 1000n^2 - 100n + 3 \approx \Theta(n^3).$$

2.2-2

```
SELECTION-SORT($A$)
1:  for $i = 1$ to $A.length - 1$ do
```

```
2:    $smallest = i$
3:    for $j = i$ to $A.length$ do
4:       if $A[j] < A[smallest]$ then
5:          $smallest = j$
6:    $A[smallest] = A[i]$
```

The **for** loop from lines 2-8 maintains the loop invariant that $A[1\,..\,i-1]$ contains sorted elements in ascending order.

The algorithm only needs to run for the the first $n-1$ elements since the nth element will already be the largest element in the array after each smaller element in $A[1\,..\,n]$ has been sorted.

Since there is no **while** loop and each search for the smallest item in the subarray $A[i+1\,..\,n-1]$ must occur, the best and wors-case running times are the same, i.e., $\Theta(n^2)$.

The implementation can be seen in the following file:
`src/algorithms/sorting/selection_sort.h`

2.2-3

Assuming the element being searched for is equally likely to be at any position in the array, we would on average search through

$$\frac{1+2+\cdots+n}{n} = \frac{1}{n}\sum_{k=1}^{n} k = \frac{n+1}{2}$$

elements for an input sequence of length n. In the worse case we would search through all n elements.

Thus the average- and worst-case running times are $\Theta(n)$, since

$$\frac{n+1}{2} = \frac{n}{2} + \frac{1}{2},$$

and the dominant term is $\frac{1}{2}n$, which is equivalent to $\Theta(n)$. Obviously, looping through n items sequentially is $\Theta(n)$, so the average- and worse-case running times are in an equivalent Θ-class.

2.2-4

By only inputting elements that are equivalent or close to the expected output.

2.3 Designing algorithms

2.3-1

In Figure 2, we illustrate the operation of MERGE-SORT on the array $A = \langle 3, 41, 52, 26, 38, 57, 9, 49\rangle$.

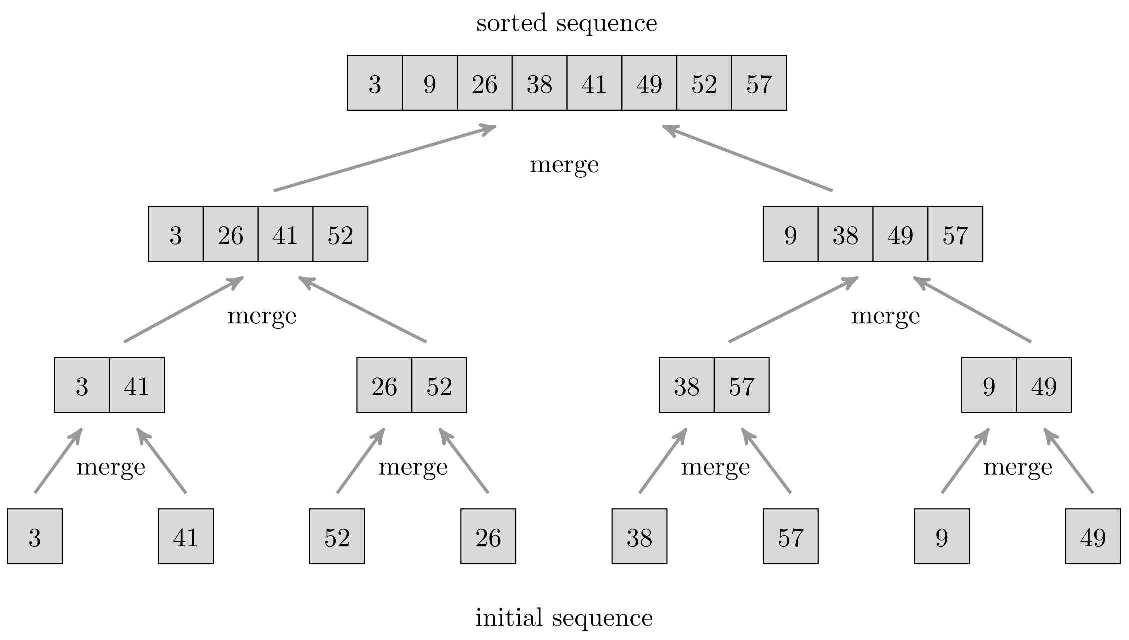

Figure 2: Illustration of MERGE-SORT.

2.3-2

```
MERGE($A, p, q, r$)
$n_1 = q - p + 1$
$n_2 = r - q$
let $L[1 \,.\,.\, n_1 + 1]$ and $R[1 \,.\,.\, n_2 + 1]$ be new arrays
for $i = 1$ to $n_1$ do
    $L[i] = A[p + i - 1]$
for $j = 1$ to $n_2$ do
    $R[j] = A[q + j]$
$i = 1$
$j = 1$
for $k = p$ to $r$ do
    if $i \leq n_1 + 1$ and $L[i] \leq R[j]$ then
        $A[k] = L[i]$
        $i = i + 1$
    else if $A[k] == R[j]$ then
        $j = j + 1$
```

The implementation can be seen in the following file:
`src/algorithms/sorting/merge_sort_no_sentinel.h`

2.3-3

Proof. We have the base case for $n = 2^1$ given by

$$T(n) = T(2) = 2 = 2\log_2 2^1 = 2 \cdot 1.$$

Now, we assume as our induction step that if $n = 2^k$ for $k > 1$,

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

Then, suppose that $n = 2^{k+1}$, so that we have

$$\begin{aligned}
T(n) &= 2T(2^{k+1}/2) + 2^{k+1} \\
&= 2T(2^k) + 2^{k+1} \\
&= 2(2^k \lg 2^k) + 2^{k+1} && \text{(by our induction assumption)} \\
&= 2^{k+1} \lg 2^k + 2^{k+1} \lg 2 && \text{(since } \log_2 2 = 1) \\
&= 2^{k+1} \lg 2^{k+1} && \text{(since } \log_2 a + \log_2 b = \log_2 ab)
\end{aligned}$$

Therefore, by induction on $n = 2^k$, $T(n) = n \lg n$. □

2.3-4

We let $T(n)$ be the running time of a problem of size n. For $n \leq 2$, we have $T(n) = \Theta(2) = \Theta(1)$, since this is just the process of inserting an item in either position 1 or 2, so a worst-case time is $\Theta(1)$.

If $n > 2$, our division of the problem yields $n-1$ subproblems, each of which is $\frac{n-1}{n}$ times the size of the original. It takes $T(\frac{n(n-1)}{n}) = T(n-1)$ to solve one subproblem, so it takes $(n-1)T(n-1)$ to solve $n-1$ of them. It then takes $\Theta(n)$ time to divide the problem into subproblems and insert the last item in the array into $A[1\,.\,.\,n-1]$, so we have

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 2 \\ (n-1)T(n-1) + \Theta(1) & \text{otherwise} \end{cases}.$$