Algorithm

Chapter 1

Foundations

1.1 Computational Problems and Algorithms

Definition 1.1. A **computational problem** is a relation

$$P \subseteq X \times Y,$$

where X is called the set of **instances** and Y is called the sets of **solutions**.

Definition 1.2. We will assume the **random-access machine (RAM)** model of computation as our implementaion technology for most of this note. In this model, we have an infinite sequence of w-bit words, and we assume $w = \lceil c \lg n \rceil$ for some constant $c \geq 1$, where n is the input size. We can perform some basic operations on these words, including

- arithmetic operations (e.g., addition, subtraction, multiplication, division),
- data movement operations (e.g., load, store, copy), and
- control operations (e.g., branch, subroutine call, return).

Definition 1.3. Given a computational model, an **algorithm** is defined as a finite sequence of basic operations that transforms a given input into a unique output.

- We say that an algorithm **solves** a computational problem $P \subseteq X \times Y$ if it transforms every instance $x \in X$ into a solution $y \in Y$ such that $(x, y) \in P$.
- The **running time** of an algorithm on a specific input is defined as the number of basic operations performed.

1.2 Asymptotic Notations

Definition 1.4. Let $f(n)$ and $g(n)$ be functions.

- We write $f(n) = O(g(n))$ if there exists positive constants c and n_0 such that
$$0 \leq f(n) \leq cg(n)$$
holds for any integer $n \geq n_0$.
- We write $f(n) = \Omega(g(n))$ if $g(n) = O(f(n))$.
- We write $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $g(n) = O(f(n))$.
- We write $f(n) = o(g(n))$ if for any real number $c > 0$, there exists a positive constant n_0 such that
$$0 \leq f(n) < cg(n)$$
holds for any integer $n \geq n_0$.
- We write $f(n) = \omega(g(n))$ if $g(n) = o(f(n))$.

Chapter 2

Sorting

2.1 Insertion Sort

In this chapter, we focus on the sorting problem. An algorithm that solves the sorting problem is usually called a sorting algorithm.

Problem 2.A (Sorting Problem).

- Input: An array $A[1 \mathinner{..} n]$ of numbers.
- Output: A permutation of A that is nondecreasing.

Algorithm 2.1. Insertion-Sort is an efficient sorting algorithm if the size of input array is small.

```
Insertion-Sort(A[1 .. n])
 1  for i ← 2 to n
 2      τ ← A[i]
 3      j ← i
 4      φ ← TRUE
 5      while φ
 6          if j = 1 or A[j − 1] ≤ τ
 7              φ ← FALSE
 8          else
 9              A[j] ← A[j − 1]
10              j ← j − 1
11      A[j] ← τ
```

Theorem 2.2. The algorithm Insertion-Sort correctly solves the sorting problem.

Proof. We prove the loop invariant that at the start of each iteration of the **for** loop of lines 1 – 11, the subarray $A[1 \mathinner{..} i-1]$ is a nondecreasing permutation of the elements originally in $A[1 \mathinner{..} i-1]$. The loop invariant is trivially true for $i = 2$, and we show that each iteration maintains the loop invariant.

First, we set $\tau \leftarrow A[i]$ and $j \leftarrow i$. Then the **while** loop of lines 5 – 10 maintains the loop invariant that at the start of each iteration, $A[1 \mathinner{..} j-1]$ remains unchanged, and the elements in $A[j+1 \mathinner{..} i]$ are the elements originally in $A[j \mathinner{..} i-1]$, each at its corresponding position. It can be shown that when the **while** loop of lines 5 – 10 terminates, each element in $A[1 \,..\, j-1]$ is less than or equal to $A[j]$, and each element in $A[j+1 \,..\, i]$ is greater than $A[j]$. Thus, after we set $A[j] \leftarrow \tau$, the subarray $A[1 \,..\, i]$ becomes a sorted permutation of the elements originally in $A[1 \,..\, i]$, implying that the loop invariant holds after the increment of i.

When the **for** loop of lines 1 – 11 terminates, we have $i = n + 1$. Due to the loop invariant, the entire array is a nondecreasing permutation of the original input array, which completes the proof. □

Theorem 2.3. The worst-case running time of INSERTION-SORT is $\Theta(n^2)$.

Proof. It is easy to verify that the **while** loop of lines 5 – 10 takes $O(i)$ time. Thus, the overall running time is $O(n^2)$.

However, if the input array is strictly decreasing, then the **while** loop of lines 5 – 10 will takes $\Omega(i)$ time. In this case, the overall running time is $\Omega(n^2)$. Thus, the worst-case running time of INSERTION-SORT is $\Theta(n^2)$. □

2.2 Heapsort

Definition 2.4. A **binary heap** is a complete binary tree such that the value of each node is not less than the values of its children.

We can use an array to represent a complete binary tree, such that $A[1]$ is the root of the tree, and $A[2i]$ and $A[2i+1]$ are the left child and the right child of $A[i]$.

Algorithm 2.5. Suppose that $A[1\,..\,n]$ is an array representing a complete binary tree. If the subtrees rooted at $A[2i]$ and $A[2i+1]$ are already heapfied, then we can use Heapify-Down to heapify the subtree rooted at $A[i]$.

```
Heapify-Down(A[1..n], i)
φ ← TRUE
while φ
    ℓ ← 2i
    r ← 2i + 1
    j ← i
    if ℓ ≤ n and A[ℓ] > A[j]
        j ← ℓ
    if r ≤ n and A[r] > A[j]
        j ← r
    if j = i
        φ ← FALSE
    else
        swap A[i] and A[j]
        i ← j
```

```
Heapify-Up(A[1..n], i)
j ← ⌊i/2⌋
while j ≥ 1 and A[i] > A[j]
    swap A[i] and A[j]
    i ← j
    j ← ⌊j/2⌋
```

```
Heapsort(A[1..n])
for i ← ⌊n/2⌋ downto 1
    Heapify-Down(A, i)
for j ← n downto 2
    swap A[1] and A[j]
    Heapify-Down(A[1..j − 1], 1)
```

Chapter 3

Divide and Conquer

3.1 Selection

Problem 3.A (Selection Problem).

- Input: An array A of n numbers and an integer k with $1 \le k \le n$.
- Output: The kth smallest number of A.

```
PARTITION(A)
n ← |A|
i ← 1
for j ← 1 to n − 1
      if A[j] ≤ A[n]
            swap A[i] and A[j]
            i ← i + 1
swap A[i] and A[n]
return i
```

```
SELECT(A, i)
n ← |A|
if n ≤ 5
      INSERTION-SORT(A)
else
      ℓ ← ⌊n/5⌋
      for i ← 1 to ℓ
            INSERTION-SORT(A[(5i − 4) .. 5i])
            swap A[i] and A[5i − 2]
      m ← ⌈ℓ/2⌉
      SELECT(A[1 .. ℓ], m)
      swap A[m] and A[n]
      j ← PARTITION(A)
      if j > i
            SELECT(A[1 .. j − 1], i)
      elseif j < i
            SELECT(A[j + 1 .. n], i − j)
```

Chapter 10

Shortest Paths

10.1 Single-Source Shortest Paths

```
BELLMAN-FORD(G, w, s)
n ← |V(G)|
for each u ∈ V(G)
    u.d ← ∞
    u.π ← NIL
s.d ← 0
for i ← 1 to n − 1
    for each u ∈ V(G)
        for each v ∈ N_G(u)
            if v.d > u.d + w(u, v)
                v.d ← u.d + w(u, v)
                v.π ← u
for each u ∈ V(G)
    for each v ∈ N_G(u)
        if v.d > u.d + w(u, v)
            return FALSE
return TRUE
```

```
DIJKSTRA(G, w, s)
for each u ∈ V(G)
    u.d ← ∞
    u.π ← NIL
s.d ← 0
Q ← V(G)
while Q ≠ ∅
    u ← EXTRACT-MIN(Q)
    for each v ∈ N_G(u)
        if v.d > u.d + w(u, v)
            v.d ← u.d + w(u, v)
            v.π ← u
```