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**.

Example. Let X be the set of nonempty sequence of distinct integers and let Y be the set of positive integers. Then we can define $P \subseteq X \times Y$ such that

$$((a_1, a_2, \ldots, a_n), i) \in P$$

if and only if $1 \leq i \leq n$ and $a_i \geq a_j$ for all $j \in \{1, \ldots, n\}$. We can write this problem as follows.

Problem 1.A (Champion Problem).

- Input: A sequence A of n distinct integers $A[1], \ldots, A[n]$.
- Output: The index of the maximum element of A.

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.

Example. We can find the index of maximum of a sequence by the following algorithm Max-Index.

```
Max-Index(A)
1  n ← A.length
2  i ← 1
3  for j ← 2 to n
4      if A[i] < A[j]
5          i ← j
6  return i
```

Note that we will describe algorithm is pseudocode so that implementaion details can be hidden.

Theorem 1.4. The algorithm Max-Index solves the champion problem using the least number of comparisons.

Proof. We show that Max-Index solves the champion problem as follows. We focus on the loop invariant that at the start of each iteration of the **for** loop of lines 3 – 5, $A[i]$ is the maximum of the subarray $A[1 \mathinner{..} j-1]$. The loop invariant is obviously true when entering the loop, and it remains true between iterations since $A[i] \geq A[j]$ must hold at the end of each iteration due to lines 4 and 5. Thus, when the **for** loop terminates, $A[i]$ should be the maximum of $A[1 \mathinner{..} n]$.

Furthremore, the algorithm Max-Index uses $n-1$ comparisons, and it is optimal with respect to the number of comparisons performed, since at least $n-1$ comparisons are necessary to determine the maximum. $\square$

Chapter 2

Sorting

2.1 Insertion Sort

In this chapter, we consider algorithms that solves the sorting problem, which is stated as follows.

Problem 2.A (Sorting Problem).

- Input: An array A of n integers.
- Output: A permutation of A that is non-decreasing.

Let us begin with **insertion sort**, which is a simple sorting algorithm.

```
INSERTION-SORT(A, n)
1  for i ← 2 to n
2      k ← A[i]
3      j ← i
4      while j > 1 and k < A[j − 1]
5          A[j] ← A[j − 1]
6          j ← j − 1
7      A[j] ← k
```

Theorem 2.1. Insertion-Sort solves the sorting problem in $\Theta(n^2)$ time in the worst case, where n is the length of the input array.

Proof. We prove the loop invariant that at the start of each iteration of the **for** loop, the subarray $A[1\,..\,i-1]$ is a non-decreasing permutation of the elements originally in $A[1\,..\,i-1]$.

The loop invariant is trivially true for $i = 2$, and is maintained by each iteration as follows. First, $A[i]$ is copied into k. The **while** loop of lines 4 – 6 moves the elements in $A[1\,..\,i-1]$ that are greater than k by one position to the right. Thus, at the end of itertaion, $A[1\,..\,j-1]$ stores the elements originally in $A[1\,..\,i-1]$ that are less than or equal to k, and $A[j+1\,..\,i]$ stores the elements originally in $A[1\,..\,i-1]$ that are greater than k. After storing k into $A[j]$, $A[1\,..\,i]$ is a nondecreasing permutation of the elements originally in $A[1\,..\,i]$, and incrementing i preserves the loop invariant.

When the **for** loop terminates, we have $j = n + 1$. Hence, the entire array $A[1\,..\,n]$ is sorted due to the loop invariant, implying that Insertion-Sort correctly solves the sorting problem.

Now we analyze the running time of Insertion-Sort. Let t_i denote the number of times the **while** loop test in line 4 is executed for that value of i. Then the running time $T(n)$ is given by

$$T(n) = \Theta\left(\sum_{i=2}^{n} t_i\right).$$

We have $T(n) = O(n^2)$ since $t_i \leq i$ for each $i \in \{2, \ldots, n\}$. If the original array is strictly decreasing, then $t_i = i$ for each $i \in \{2, \ldots, n\}$, and we have $T(n) = \Omega(n^2)$ in this case. Thus, $T(n) = \Theta(n^2)$ in the worst case, which completes the proof. $\square$

2.2 Heap Sort

This section is under construction.

Max-Heapify(A, i, n)

$l \leftarrow 2i$
$r \leftarrow 2i + 1$
$k \leftarrow i$
if $l \leq n$ and $A[l] > A[i]$
    $k \leftarrow l$
if $r \leq n$ and $A[r] > A[i]$
    $k \leftarrow r$
if $k \neq i$
    swap $A[i]$ and $A[k]$
    Max-Heapify(A, k)

Build-Max-Heap(A, n)

for $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1
    Max-Heapify(A, i, n)

Heap-Sort(A, n)

Build-Max-Heap(A, n)
for $i \leftarrow n$ **downto** 2
    swap $A[1]$ and $A[i]$
    Max-Heapify$(A, 1, i - 1)$