Chapter 1
Introduction

What is an algorithm

The basic function of the computer is computing some results from input and make the correct output. An algorithm is the computational procedure that processing the input and produce the output.

Therefore algorithm is one of the most important core topics in computer science.

Algorithm: an abstraction of some computational procedures.

For example, a sorting algorithm is a computing procedure which solves the following problem:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A reordered sequence $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

When people use the algorithm, actually they use it to solve an instance of a problem. One might input $\langle 31, 41, 59, 26, 41, 58 \rangle$ and get the output $\langle 26, 31, 41, 41, 58, 59 \rangle$. An algorithm is not for a instance problem, but for a class of instance problems.

Correctness and efficiency

An algorithm is said to be correct if, for each input instance, it always finds out the correct output. To prove the correctness of an algorithm usually is not easy. Note that to try many instances is not a proof of the correctness of the according algorithm.

Usually, for one problem, there will be many correct algorithms to solve the problem. So we need to consider the efficiency of the algorithms.

- Some algorithm may not always output correct answers. Such an algorithm is called incorrect algorithm. Sometimes, an incorrect algorithm is also useful, if the probability of the wrong output is small.
- But in general, we consider correct algorithms.
- Not every problem has an efficient solution (of cause there are still many problems have no solution). So sometimes, we need to prove that a problem does not have an efficient solution. Or we need to prove that an existing algorithm is optimal, i.e., one cannot design a better algorithm.

- The efficiency of an algorithm is estimated by the running time and memory usage.
- We need to consider machine-independent efficiency.
- Machine-independent algorithm design depends upon a hypothetical computer called the Random Access Machine or RAM.

Under RAM model:

- Each simple operation (arithmetic operation $(+, -, \times, /, =)$, if, call) takes exactly one time step.
- Loops and subroutines are not considered simple operations. Instead, they are the composition of many single-step operations. The time it takes to run through a loop or execute a subprogram depends upon the number of loop iterations or the specific nature of the subprogram.
- Each memory access takes exactly one time step. Further, we have as much memory as we need. The RAM model takes no notice of whether an item is in cache or on the disk.

Note

- RAM is not a real computer and the different steps will take different time. So the RAM model will not give the exact time calculation.
- But that model simplifies the calculation and it still gives us a very good way to understanding the efficiency of the algorithms.
- More importantly, the model gives us a method to estimate the run time of the algorithm, that is machine independent.

Running time function $T(n)$

Since algorithm is used to treat the input, the running time depends on the length of input data. To estimate the running time, we define a function of the length of the data, $T(n)$, to denote the general running time, where n is the length of the data.

Example: Insertion Sort

Insertion	cost	times
for $j = 2$ to n	c_1	n
key $= A[j]$	c_2	$n-1$
$i = j - 1$	c_3	$n-1$
while $(i > 0)$ and $A[i] >$ key	c_4	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_5	$\sum_{j=2}^{n}(t_j - 1)$
$i = i - 1$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$A[i+1] =$ key //insert the key	c_7	$n-1$

Notation

In the algorithm, we use the summation formulas. In general,

$$\sum_{i=1}^{n} f(n) = f(1) + f(2) + \cdots + f(n).$$

For examples

$\sum_{i=1}^{n} 1 = n,$

$\sum_{i=1}^{n} i = n(n+1)/2.$

$\sum_{i=0}^{n} i^2 = n(n+1)(2n+1)/6.$

In the example, t_j means the number of times the `while` loop test is executed for that value of j. c_i denotes the execute constant time for that step. We have

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j \\ &\quad + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1). \end{aligned}$$

For different input, the running time will be different. Because the value of t_j will be different.

We can easily estimate the value of $T(n)$ for the best case and the worst case. In the best case, $t_j = 1$, i.e., the input is already sorted in the order from smallest to the largest. Therefore we have the best case running time:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) \end{aligned}$$

We can express it as: $T(n) = an + b$, where a and b are constants. It is thus a linear function of n.

If the input is in reverse sorted order (i.e., in decreasing order), then it is the worst case. In this case, $t_j = j$ because it will be compared with j numbers. Since $\sum_{j=2}^{n} j = n(n+1)/2 - 1$ and $\sum_{j=2}^{n}(j-1) = n(n-1)/2$,

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1) \\ &= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right) n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right) n \\ &\quad - (c_2 + c_3 + c_4 + c_7) \end{aligned}$$

So the worst case running time is $T(n) = an^2 + bn + c$, where a, b and c are constants. It is a quadratic function.

In general, we will focus on the complexity of **worst case**. The main reasons are

- The worst-case running time of an algorithm gives us an upper bound on the running time.
- For some algorithms, the worst case occurs fairly often.
- The "average" case is often roughly as bad as the worst case. The average case is not easy to define. Usually a probabilistic analysis is applied.

Order of growth

In the example, the running time of the algorithm in best case is a linear function of n, while in worst case is quadratic function of n.

Now we want to make further simplifying abstraction. We call it the rate of growth or order of growth or the running time. In general, we want to estimate the behavior of the algorithm when n is bigger.

We can consider the leading term of a formula. For example we just consider an^2 for $an^2 + bn + c$ and furthermore we ignore the leading terms constant coefficient, since the lower-order terms and the constant coefficient are relatively insignificant for large values of n.

This method is useful to study the the asymptotic efficiency of algorithms. So we basically consider how the running time of an algorithm increases with the size of the input size increases without bound.

Usually an algorithm that is asymptotically more efficient will be the best choice for applications treating large data.

Notations

The notations we used to describe the asymptotic running time of an algorithm are defined as functions of natural numbers. Such notations are convenient for describing the running time function $T(n)$, where n is the number of items of input.

We will introduce three notations which describe the asymptotic running time which are not only for worst case but for any kind of input.

Θ-notation

For a given function $g(n)$, $\Theta(g(n))$ is defined as the set of functions

$$\begin{aligned}\Theta(g(n)) \quad = \quad & \{f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \\ & \text{such that} 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}\end{aligned}$$

If we can find out positive constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$, then we say that $f(n) \in \Theta(g(n))$. Usually, we will use the notation $f(n) = \Theta(g(n))$ to denote the same notation.

Example

Let $g(n) = n^2, f(n) = 5.5n^2 + 3n + 1$. Then it is not difficult to prove that $f(n) = \Theta(n^2)$. In fact, for c_1 we can choose any positive constant less than 5.5, say 5. Now we choose $c_2 = 6$, because the leading coefficient determines the behavior of the polynomial.

Next we need to find n_0 such that for all $n \geq n_0$,
$5.5n^2 + 3n + 1 \leq 6n^2$, or equivalently $0.5n^2 \geq 3n + 1$. Dividing by $0.5n$ for both sides, we have $n \geq 6 + \frac{2}{n}$. So we can choose $n_0 = 7$. We have

$$5n^2 \leq 5.5n^2 + 3n + 1 \leq 6n^2 \text{ for all } n \geq 7.$$

Therefore $f(n) = \Theta(n^2)$.

In general, for a polynomial of n

$$f(n) = \sum_{i=0}^{t} a_i n^t = a_0 + a_1 n + \cdots + a_t n^t,$$

where $a_0, a_1, \ldots, a_t$ are constants and $a^t > 0$, we can use a similar method to prove that $f(n) = \Theta(n^t)$.

- $f(n) \in \Theta(g(n)$ means, there is positive integer n_0 such that when $n \geq n_0$, the function $f(n)$ is equal to $g(n)$ within a constant factor.

  So we say that $g(n)$ is an asymptotically tight bound for $f(n)$.

- The definition of $\Theta(g(n))$ requires that every member $f(n) \in \Theta(g(n))$ be asymptotically nonnegative, i.e., $f(n)$ is nonnegative for sufficiently large n.

- It is obvious that $g(n) = \Theta(g(n))$.

Example

We can also use the formal definition to prove that $2n^3 \neq \Theta(n^2)$. To show that, we should show that for any positive constant c and any large integer n_0, we can always find some integer $n > n_0$, such that $2cn^3 > n^2$. Because from the definition, there exists a positive c such that $2cn^3 \leq n^2$ for $n > n_0$, where n_0 is some large integer.

In fact, for any positive constant c, we can choose $n > \max\{\frac{1}{2c}, n_0\}$. Then $2cn^3 > n^2$, no matter how to choose n_0. Therefore $2n^3 \neq \Theta(n^2)$.

O-notation

When we have only an asymptotic upper bound, we use O-notation. $O(g(n))$ is also defined as a set of functions.

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

$f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$, because the definition of Θ notation is stronger than O notation. In the Θ notation, we need to find c_1, c_2 and n_0 and we can use c_2 and n_0 for the O notation.

Note

- For a linear function $f(n) = an + b$, we can prove that $f(n) = O(n)$. In fact we can let $c = a + |b|$ and $n_0 = 1$. It is also easy to see that $f(n) = O(n^2)$ by using the same constants. So O-notation just gives an upper bound which may not be tight.
- When we use O notation to describe the upper bound of the running time for the worst case of an algorithm, we have bounded on the running time of the algorithm for any input. But the Θ notation does not have the similar property. For example, the insertion sort for the best case is $\Theta(n)$ but not $\Theta(n^2)$.

Sometimes, we need to prove that $f(n) \neq O(g(n)$. In this case, we need to prove that for any constant c and n_0, we always can find some $n' > n_0$ such that $f(n') \geq cg(n')$.

Example

Suppose $f(n) = 5n^2 + n + 1$. We want to prove that $f(n) \neq O(n)$. For any constant c and n_0, we can try to find some $n' > n_0$ such that $5n'^2 + n' + 1 > cn'$. This is equivalent to that $n' + \frac{1}{5} + \frac{1}{5n'} \geq \frac{c}{5}$. So we can let $n' > \max\{n_0, \frac{c}{5}\}$, then $5n'^2 + n' + 1 > cn'$. So $f(n) \neq O(n)$.

Ω-notation

Ω notation gives an asymptotic lower bound for a function. The definition of Ω notation is defined as follows.

$$\begin{aligned}\Omega(g(n)) &= \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that} \\ &\quad 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}\end{aligned}$$

The running time of an algorithm being $\Omega(g(n))$ means that the running time, independent from the properties of input, in at least a constant times $g(n)$. Therefore this is also a lower bound of the best case.

From the definitions of the asymptotic notations we learned so far, the following theorem is not difficult to prove.

Theorem 1.2.1 For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n)$ and $f(n) = \Omega(g(n))$.

We may use asymptotic notations in equations and inequalities. For example, we may write $2n^2 + 3n + 1 = \Theta(n^2)$, we might also write it as $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$. Since we are considering the asymptotic behavior, there is no point to write it as $2n^2 + 3n + 1 = 3n + 1 + \Theta(n^2)$, though it is correct in logic.

Suppose that $f(n)$ and $g(n)$ are asymptotic positive. Then

- Transitivity

  $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ imply $f(n) = \Theta(h(n))$,

  $f(n) = O(g(n))$ and $g(n) = O(h(n))$ imply $f(n) = O(h(n))$,

  $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$ imply $f(n) = \Omega(h(n))$,.

- Reflexivity

  $f(n) = \Theta(f(n))$,

  $f(n) = O(f(n))$,

  $f(n) = \Omega(f(n))$.

- Transpose symmetry

  $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.

Data structure review

Data structure is a way to store and organize data in order to facilitate access and modifications.

Now we review some elementary data structures discussed in CS 2412. Some notations may different from the previous course.

A set manipulated by algorithms can grow, shrink or change over time. We call such a set dynamic set. Here we review some linear dynamic sets first.

Stack

A stack is a linear dynamic set with the property last-in first-out (LIFO). The operations of this data structure are at the top of the stack.

We can use an array to implement a stack. A stack consists of elements $S[1 \ldots S.top]$, where $S[1]$ is the element at the bottom of the stack and $S[S.top]$ is the element at the top. When $S.top == 0$, the stack is empty.

Main operations:

```
1: procedure PUSH(S, x)
2:     S.top = S.top + 1                ▷ Omitted the overflow checking
3:     S[S.top] = x
4: end procedure
```

```
1: procedure POP(S)
2:     if S is empty then
3:         error "underflow"
4:     else
5:         S.top = S.top − 1
6:         return S[S.top + 1]
7:     end if
8: end procedure
```

Queues

The queues have the property first-in first-out (FIFO). The operations on the queues are at two ends, the head and the tail. We also use an (circular) array to implement a queue.

Main operations:

```
1: procedure ENQUEUE(Q, x)
2:     Q[Q.tail] = x                ▷ omitted the check if the queue is full
3:     if Q.tail == Q.length then
4:         Q.tail = 1
5:     else
6:         Q.tail = Q.tail + 1
7:     end if
8: end procedure
```

```
1: procedure DEQUEUE(Q)
2:     x = Q[Q.head]        ▷ omitted the check if the queue is empty
3:     if Q.head == Q.length then
4:         Q.head = 1
5:     else
6:         Q.head = Q.head + 1
7:     end if
8:     return x
9: end procedure
```

Linked lists

Linked lists are more flexible representation for dynamic sets.

One example implementation is doubly linked list. Each element of a doubly linked list is an object with and attribute "key" and two other pointer attributes: *next* and *prev*. The object may also contain other data. In general a linked list may be single linked, doubly linked or circular list. The first element of the list is called the head and the last element is called the tail. The list can be sorted or unsorted.

Some operations:

```
1: procedure LIST-SEARCH(L, k)
2:     x = L.head
3:     while x ≠ NIL and x.key ≠ k do
4:         x = x.next
5:     end while
6:     return x
7: end procedure
```

```
1: procedure LIST-INSERT(L, x)
2:     x.next = L.head                    ▷ insert as the first element
3:     if L.head ≠ NIL then
4:         L.head.prev = x
5:     end if
6:     L.head = x
7:     x.prev = NIL
8: end procedure
```

```
1:  procedure LIST-DELETE(L, x)
2:      if x.prev ≠ NIL then
3:          x.prev.next = x.next
4:      else
5:          L.head = x.next
6:      end if
7:      if x.next ≠ NIL then
8:          x.next.prev = x.prev
9:      end if
10: end procedure
```

Heaps

The (binary) heap data structure is an array that we can view as a nearly complete binary tree.

An array A that represent a heap is an object with two attributes: $A.length$ which gives the number of elements in the array, and $A.heap\text{-}size$, which represents how many elements in the heap are stored within array A.

In the array $A[1..A.length]$, $A[1]$ is the root of the tree. Given the index i of a node, it is easily compute the indexes of its parent, left child and right child:

PARENT(i)

return $\lfloor i/2 \rfloor$

LEFT(i)

return $2i$

RIGHT(i)

return $2i + 1$

- On most computers, it is very efficient to compute $2i$ or $\lfloor i/2 \rfloor$, just shift the binary representation of i left, or right, by one bit position.
- There are two kinds of binary heaps: max-heaps and min-heaps. In a max-heap, $A[\textsc{Parent}(i) \geq A[i]$, while in a min-heap, $A[\textsc{Parent}(i) \leq A[i]$ for any node $A[i]$.

Main operations (use max-heap as example):

```
1: procedure MAX-HEAPIFY(A, i)
2:     l = LEFT(i)
3:     r = RIGHT(i)
4:     if l ≤ A.heap-size and A[l] > A[i] then
5:         largest = l
6:     else
7:         latgest = r
8:     end if
9:     if largest ≠ i then
10:        exchange A[i] with A[largest]
11:        MAX-HEAPIFY(A, largest)
12:    end if
13: end procedure
```

The above procedure "reheap down" an element if it damaged the max-heap property. The following procedure builds a heap on a existing array A.

```
1: procedure BUILD-MAX-HEAP(A)
2:     A.heap-size = A.length
3:     for i = ⌊A.length/2⌋ downto 1 do
4:         MAX-HEAPIFY(A, i)
5:     end for
6: end procedure
```

The above procedure "reheap down" an element if it damaged the max-heap property. The following procedure builds a heap on a existing array A.

```
1: procedure BUILD-MAX-HEAP(A)
2:     A.heap-size = A.length
3:     for i = ⌊A.length/2⌋ downto 1 do
4:         MAX-HEAPIFY(A, i)
5:     end for
6: end procedure
```

Since the running time for MAX-HEAPIFY is $O(\log n)$, the running time for procedure BUILD-MAX-HEAP has a upper bound $O(n \log n)$. A more careful discussion can show that the asymptotically running time is $O(n)$. We omitted the proof here.

```
1: procedure HEAPSORT(A)
2:     BUILD-MAX-HEAP(A)
3:     for i = A.length downto 2 do
4:         exchange A[1] with A[i]
5:         A.heap-size = A.heap-size − 1
6:         MAX-HEAPIFY(A, 1)
7:     end for
8: end procedure
```

The HEAPSORT procedure takes time $O(n \log n)$.

One important application for the heap is priority queues. There are two types of priority queues: max-priority queues and min-priority queues.

We use a max-priority queue to explain the main ideas. The min-priority queue is similar.

A max-priority queue maintains a set S and supports the following operations.

- INSERT(S, x) inserts the element x into the set S, which is equivalent to the set operation $S = S \cup \{x\}$.
- MAXIMUM(S) returns the element S with the largest key.
- EXTRACT-MAX(S) removes and returns the element of S with the largest key.
- INCREASE-KEY(S, x, k) increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

Now we explain in details about how to use a max-heap to implement the max-priority queue. Let A be a max-heap. Then we can use the following implementation of operations of a max-priority queue.

HEAP-MAXIMUM(A)

return $A[1]$

```
1: procedure HEAP-EXTRACT-MAX(A)
2:     if A.heap-size < 1 then
3:         error: "heap underflow"
4:     end if
5:     max = A[1]
6:     A[1] = A[A.heap-size]
7:     A.heap-size = A.heap-size − 1
8:     MAX-HEAPIFY(A, 1)
9: end procedure
```

Running time: $O(\log n)$.

```
1: procedure HEAP-INCREASE-KEY(A, i, key)
2:     if key < A[i] then
3:         error: "new key is smaller than current key"
4:     end if
5:     A[i] = key
6:     while i > 1 and A[PARENT(i)] < A[i] do
7:         exchange A[i] with A[PARENT(i)]
8:         i =PARENT(i)
9:     end while
10: end procedure
```

Running time: $O(\log n)$.

```
1: procedure MAX-HEAP-INSERT(A, key)
2:     A.heap-size = A.heap-size + 1
3:     A[A.heap-size] = −∞
4:     HEAP-INCREASE-KEY(A, A.heap-size, key)
5: end procedure
```

The running time: $O(\log n)$.