Part I

Algorithms

Chapter 1

Computational Complexity

1.1 Asymptotic Notation

1.1.1 Asymptotic Analysis

Definition 1.1.1 (asymptotically positive function)**.** $f : \mathbb{N} \to \mathbb{R}$ is an asymptotically positive function if $\exists n_0 > 0$ such that $\forall n > n_0$ we have $f(n) > 0$

1.1.2 O-Notation, Ω-Notation and Θ-Notation

Definition 1.1.2 (O-Notation)**.** For a function $g(n)$,

$$O(g(n)) = \{f : \exists c > 0, n_0 > 0 \text{ such that } f(n) \leq cg(n), \forall n \geq n_0\}$$

O-Notation also known as asymptotic upper bound.

Definition 1.1.3 (Ω-Notation)**.** For a function $g(n)$,

$$\Omega(g(n)) = \{f : \exists c > 0, n_0 > 0 \text{ such that } f(n) \geq cg(n), \forall n \geq n_0\}$$

Ω-Notation also known as asymptotic lower bound.

Definition 1.1.4 (Θ-Notation)**.** For a function $g(n)$,

$$\Theta(g(n)) = \{f : \exists c > 0, n_0 > 0 \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$$

Ω-Notation and Θ-Notation are not used very often when we talk about running times.

Definition 1.1.5 (o-Notation)**.** For a function $g(n)$,

$$o(g(n)) = \{f : \exists c > 0, n_0 > 0 \text{ such that } f(n) < cg(n), \forall n \geq n_0\}$$

Definition 1.1.6 (ω-Notation)**.** For a function $g(n)$,

$$\omega(g(n)) = \{f : \exists c > 0, n_0 > 0 \text{ such that } f(n) > cg(n), \forall n \geq n_0\}$$

Notice: $O(g)$, $\Omega(g)$ and $\Theta(g)$ are sets, we use "=" to represent "$\in$". In here "=" is asymmetric. Equality such as $O(n^3) = n^3 + n$ is incorrect.

Example. The following are some examples

$f(n)$	$g(n)$	O	Ω	Θ
$4n^2 + 3n$	$n^3 - 2n + 3$	Yes	No	No
$\lg^{10} n$	$n^{0.1}$	Yes	No	No
$\log_{10} n$	$\lg(n^3)$	Yes	Yes	Yes
$\lceil\sqrt{10n + 100}\rceil$	n	Yes	No	No
$n^3 - 100n$	$10n^2 \lg n$	No	Yes	No
2^n	$2^{\frac{n}{2}}$	No	Yes	No
$\sqrt{n}$	$n^{\sin n}$	No	No	No

(1.1)

Theorem 1.1. *Let f and g be two functions that*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0$$

Then $f(n) = \Theta(g(n))$

Proof. Since $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists and positive, there is some n_0 beyond which the ratio is always between $\frac{1}{2}c$ and 2c. Thus,

$$\forall n > n_0, f(n) \leq 2cg(n) \Rightarrow f(n) = O(g(n)) \tag{1.2}$$
$$\forall n > n_0, f(n) \geq \frac{1}{2}cg(n) \Rightarrow f(n) = \Omega(g(n)) \tag{1.3}$$

□

A set of properties:

$$f(n) = O(g(n)) \iff g(n) = \Omega(f(n)) \tag{1.4}$$
$$f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \text{ and } g(n) = O(f(n)) \tag{1.5}$$

Another set of properties:

$$f = O(g), g = O(h) \Rightarrow f = O(h) \tag{1.6}$$
$$f = \Omega(g), g = \Omega(h) \Rightarrow f = \Omega(h) \tag{1.7}$$
$$f = \Theta(g), g = \Theta(h) \Rightarrow f = \Theta(h) \tag{1.8}$$

Theorem 1.2. *If $f_i = O(h)$, for finite number of $i \in K$, then $\sum_{i\in K} f_i = O(h)$*

Proof is trivia.

Notice: Function f and g not necessarily have relation $f = O(g)$ or $g = O(f)$. E.g., for $f = \sqrt{n}$ and $g = n^{\sin n}$, $f \notin O(g)$ and $g \notin O(f)$

1.2 Common Running Times

The following are examples of common running times.

Running Time	Examples
$O(n)$	Scan through a list to find a element matching with input
$O(\lg n)$	Binary search
$O(n^2)$	Scan through every pair of elements, $\binom{n}{2}$
$O(n^3)$	Matrix multiplication by definition
$O(n \lg n)$	Many divide and conquer algorithm which in each step iteratively divide the problem into two part and solve the subproblem, for example mergesort.
$O(n!)$	Enumerate all permutation, for example Hamiltonian Cycle Problem
$O(c^n)$	Enumerate all elements in power set. ($O(2^n)$)
$O(n^n)$	Enumerate all combinations. (Can't find good example yet)

Here is a comparison between running times: $\lg n < n < n \lg n < n^2 < n^{\sqrt{n}} < 2^n < e^n < n! < n^n$

Chapter 2

General Strategies

2.1 Greedy Algorithms

2.1.1 Introduction

Greedy algorithm solve the problem incrementally. Its often for optimization problems. Solving optimization problem typically requires a sequences of steps, at each step, an irrevocable decision will be made, and makes the choice looks the best at the moment. Based on that, a small instance will be added to the problem and we solve it again.
Greedy algorithm do not always yield optimal solution, but it usually can provide a relatively acceptable computational complexity. They often run in polynomial time due to the incrementally of instances. Sometimes even if we cannot guarantee the solution is optimal, we still use it in optimization, because of its cheap computation burden. One of the examples will be the constructive heuristic of VRP problems.
Greedy algorithm usually gives polynomial time complexity, but that is not all the cases. In simplex method, each pivot is greedily searching through for the extreme point. Although simplex method usually gives us traceable computation running time, however, it is not a polynomial algorithm.
The following is a very rough sketch of generic greedy algorithm

Algorithm 1 Generic Greedy Algorithm

```
1: while The instance is non-trivial do
2:     Make the choice using the greedy strategy
3:     Reduce the instance
```

If we can prove the following, we can claim the greedy algorithm. First, it need to prove that for "current moment", the strategy is safe, i.e., there is always an optimum solution that agrees with the decision made according to the strategy, this is usually difficult. Then, it need to show that the remaining task after applying the strategy is to solve a/many smaller instance(s) of the same problem.

2.1.2 Examples

Interval Scheduling

For n jobs, job i starts at time s_i, and finishes at time f_i. i and j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ are disjoint. Find the maximum-size of mutually compatible jobs.
The following is the greedy algorithm to solve this problem
Now we proof the that it is safe to schedule the job j with the earliest finish time, i.e., there is an optimum solution where the job j with the earliest finish time is scheduled.

Proof. For arbitary optimum solution S, one of the following cases will happen:
Case 1: S contains j, done
Case 2: S does not contain j. Then the first job in S can be replaced by j to obtain another optimum schedule S'. □

Algorithm 2 Interval Scheduling, $S(s, f, n)$

```
Sort jobs by f
t ← 0, S ← ∅
for every j ∈ [n] according to non-decreasing order of f_j do
    if s_j ≥ t then
        S ← S ∪ {j}
        t ← f_j
return S
```

Unit-length interval covering

Given a set of n points $X = \{x_1, x_2, \cdots, x_n\}$ on the real line, WLOG, assuming the points have already been sorted. We want to use the smallest number of unit-length closed intervals to cover all the points in X.
The following is a greedy algorithm to find the set of unit-length intervals that cover all the points in real line.

Algorithm 3 Cover points with unit-length intervals

```
Initial, S ← ∅
for i = 1, 2, ..., n do
    if x_i is not covered by any unit-length intervals then
        Add one unit-length interval starting from x_i, i.e., S ← S ∪ {x_i}
```

This strategy (algorithm) is a greedy algorithm because it build up the solution in steps, in each iteration it considers one more point to be covered, i.e., it is optimal for each step in the iteration. Now we prove this algorithm is "safe".

Proof. First we consider the case where there is only one point on the real line. Then the optimal number of unit-length interval will be 1, according to the algorithm, that interval will be started at that point.
Then assuming for the case where there are k points from left to right, i.e., $X = \{x_1, x_2, \cdots, x_k\}$, and p unit-intervals is already the minimal number and placed by the algorithm, then the $(k+1)^{th}$ point can only be one of the following cases:
Case 1: The $(k+1)^{th}$ point is covered by the p^{th} unit-length interval. According to the strategy, no new unit-length interval will be needed, the number of unit-length interval for $k+1$ points will be the same as when there are k points. Therefore in this case for $k+1$ points, p is the minimal (optimal) number. So the strategy is "safe" in this case.
Case 2: The $(k+1)^{th}$ point is not covered by the p^{th} unit-length interval. According to the strategy, there will be one new unit-length interval added. Notice that p unit-length intervals will not be feasible to cover $k+1$ points in this case, because if we move the p^{th} unit-length interval to the right, it will not be able to cover at least one point which overlapped with the starting point of that unit-length interval. Since p is infeasible and $p+1$ unit-length interval is feasible, then $p+1$ is the minimal (optimal) number. So the strategy is "safe" in this case.
Notice that for the k we have mentioned above, k can start from 1 to infinite number of integer. So this strategy is "safe" in every cases. □

2.2 Divide and Conquer

2.2.1 Introduction

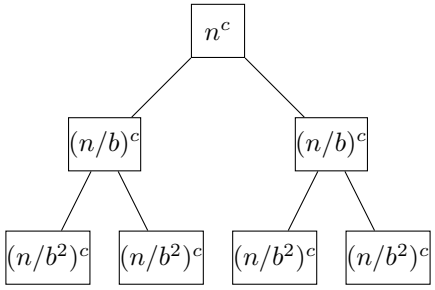
The divide-and-conquer algorithm contains three steps: divide, conquer and combine. Step one, divide instances into many smaller instances. Step two, conquer small instance by solving each of smaller instances recursively and separately. Step three, combine solutions to small instances to obtain a solution for the origin large instance.

2.2.2 Master Theorem

Theorem 2.1 (Master Theorem)**.** *For running time in forms of* $T(n) = aT(n/b) + O(n^c)$*, where* $a \geq 1$*,* $b > 1$*,* $c \geq 0$ *are constants. Then*

$$T(n) = \begin{cases} O(n^{\lg_b a}), & c < \lg_b a \\ O(n^c \lg n), & c = \lg_b a \\ O(n^c), & c > \lg_b a \end{cases} \tag{2.1}$$

Proof of Master theorem using recursion tree



Proof. The i^{th} level has a^{i-1} nodes. For the following cases, we can derive the time complexity:
Case 1: $c < \lg_b a$ bottom-level dominates $(\frac{a}{b^c}^{\lg_b n})n^c = O(n^{\lg_b a})$
Case 2: $c = \lg_b a$ all levels have same time $n^c \lg_b n = O(n^c \lg n)$
Case 3: $c > \lg_b a$ top-level dominates $O(n^c)$ □

2.2.3 Examples

Counting Inversions

2.3 Dynamic Programming

2.3.1 Introduction

2.4 Compare between three paradigms

Greedy algorithm

- Make a greedy choice
- Prove that the greedy choice is safe
- Reduce the problem to a sub-problem and solve it iteratively
- Usually for optimization problems.

Divide-and-Conquer

- Break a problem into many independent sub-problems
- Solve each sub-problem separately
- Combine solutions for sub-problems to form a solution for the origin one
- Usually used to design more efficient algorithm

Dynamic Programming

- Break up a problem into many overlapping sub-problems
- Build solutions for larger and larger sub-problems
- Use a table to store solutions for sub-problems for reuse

Chapter 3

Sorting

3.1 Exchange Sorts

3.1.1 Bubble Sort

3.1.2 Cocktail Shaker Sort

3.1.3 Odd-even Sort

3.1.4 Comb Sort

3.1.5 Gnome Sort

3.1.6 Quicksort

3.2 Selection Sorts

3.2.1 Selection Sort

3.2.2 Heapsort

3.2.3 Smoothsort

3.2.4 Cartesian Tree Sort

3.2.5 Tournament Sort

3.2.6 Cycle Sort

3.2.7 Weak-heap Sort

3.3 Insertion Sorts

3.3.1 Insertion Sort

3.3.2 Shell Sort

3.3.3 Splaysort

3.3.4 Tree Sort

3.3.5 Library Sort

3.3.6 Patience Sorting

3.4 Merge Sorts

3.4.1 Merge Sort

3.4.2 Cascade Merge Sort

3.4.3 Oscillating Merge Sort

Chapter 4

Mathematical Algorithm

4.1 Polynomial Multiplication

For given two polynomials of degree $n-1$, the algorithm outputs the product of two polynomials.

Example.

$$(3x^3 + 2x^2 - 6x + 9) \times (-2x^3 + 7x^2 - 8x + 4) \tag{4.1}$$
$$= -6x^6 + 17x^5 + 24x^4 - 60x^3 + 119x^2 - 96x + 36 \tag{4.2}$$

Then for input as (3, 2, -6, 9) and (-2, 7, -8, 4), the output will be (-6, 17, 2, -60, 119, -96, 36)

A naive algorithm to solve this problem will be $O(n^2)$

Algorithm 4 PolyMultNaive(A, B, n)

```
Let C[k] = 0 for every k = 0, 1, ··· , 2n − 2
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i + j] ← C[i + j] + A[i] × B[j]
return C
```

Use divide and conquer can reduce the running time. The idea is to divide the polynomial with degree of $n-1$ (WLOG, let n be even number) by two polynomials, i.e.

$$p(x) = p_H(x)x^{\frac{n}{2}} + p_L(x) \tag{4.3}$$

Both p_H and p_L are polynomials with degree of $\frac{n}{2} - 1$, then

$$p(x)q(x) = (p_H(x)x^{\frac{n}{2}} + p_L(x)) \times (q_H(x)x^{\frac{n}{2}} + q_L(x)) \tag{4.4}$$
$$= p_H q_H x^n + (p_H q_L + p_L q_H)x^{\frac{n}{2}} + p_L q_L \tag{4.5}$$

Therefore

$$multiply(p, q) = multiply(p_H, q_H)x^n \tag{4.6}$$
$$+ (multiply(p_H, q_L) + multiply(p_L, q_H))x^{\frac{n}{2}} + multiply(p_L, q_L) \tag{4.7}$$
$$= multiply(p_H, q_H)x^n \tag{4.8}$$
$$+ (multiply(p_H + p_L, q_H + q_L) - multiply(p_H, q_H) - multiply(p_L, q_L))x^{\frac{n}{2}} \tag{4.9}$$
$$+ multiply(p_L, q_L) \tag{4.10}$$
$$\tag{4.11}$$

The algorithm is as following
The running time $T(n) = 3T(n/2) + O(n)$, $T(n) = O(n^{\lg_2 3})$

Algorithm 5 PolyMultiDC(A, B, n)

1: **if** $n = 1$ **then return** $A[0]B[0]$
2: $A_L \leftarrow A[0..n/2-1]$, $A_H \leftarrow A[n/2..n-1]$
3: $B_L \leftarrow B[0..n/2-1]$, $B_H \leftarrow B[n/2..n-1]$
4: $C_L \leftarrow$ `PolyMultiDC(`A_L`, `B_L`, n/2)`
5: $C_H \leftarrow$ `PolyMultiDC(`A_H`, `B_H`, n/2)`
6: $C_M \leftarrow$ `PolyMultiDC(`A_H `+` A_L`, `B_H `+` B_L`, n/2)`
7: $C \leftarrow 0$ array of length $2n-1$
8: **for** $i \leftarrow 0$ to $n-2$ **do**
9: $C[i] \leftarrow C[i] + C_L[i]$
10: $C[i+n] \leftarrow C[i+n] + C_H[i]$
11: $C[i+n/2] \leftarrow C[i+n/2] + C_M[i] - C_L[i] - C_H[i]$
return C

4.2 Matrices Multiplication

4.3 Gaussian Elimination

4.4 Curve Fitting

4.5 Integration

Chapter 5

Searching

Chapter 6

String

6.1 String Searching

6.2 Pattern Matching

6.3 Parse

6.4 Optimal Caching

6.4.1 Offline Caching

Definition 6.4.1 (Furthest-in-Future)**.** (Les Belady, 1960s) When

6.5 File Compression

6.6 Cryptology

Chapter 7

Data Structures

Chapter 8

NP and PSPACE