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: $\lg n < \sqrt{n} < n < n \lg n < n^2 < n^{\sqrt{n}} < 2^n < e^n < n! < n^n$

Chapter 2

General Strategies

This chapter we are going to discussion three types of strategies for algorithm design, greedy algorithm, divide-and-conquer, and dynamic programming.

- 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

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
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

Algorithm 1 Generic Greedy Algorithm

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

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.
Although greedy algorithm is intuitive and usually leads to satisfactory complexity, however, for most of the problems there is **no** natural greedy algortihm that works.

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

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
```

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'. □

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.
Divide and conquer can sometimes solve the problems that greedy algorithm cannot solve, but they often not strong enough to reduce exponential brute-force search down to polynomial time. What usually happen is that they reduce a running time that unnecessarily large, but already polynomial, down to a faster running time.

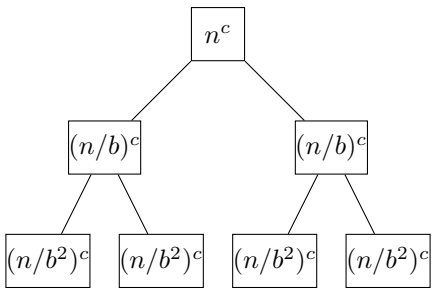
2.2.2 Master Theorem

The Master Theorem is useful in analyzing the running time of divide and conquer algorithm. Assume at each step we divide the origin problem of size n into subproblems of size n/b, run for a times of "itself" to conquer those subproblems with a combine operation of $O(n^c)$, then the total running time can be derived by the Master Theorem as following:

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

The principal of dynamic programming is essentially opposite of the greedy algorithm. Dynamic programming implicitly explores the space of all possible solutions, by carefully decomposing the origin problem into subproblems, and store the solution of those subproblems. Base on those subproblems, then build up larger and larger problem until the origin problem is solved.

2.3.2 Examples

Weighted Interval Scheduling

Consider n jobs, job i starts at time s_i and finishes at time f_i, each job has a weight of $v_i > 0$. Job i and job j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ are disjoint. Find the maximum-size subset of mutually compatible jobs. A special case of this problem is when the values of all jobs are equal, which will be the interval scheduling problem as discussed in the greedy algorithm examples.
Define $p(j)$ as for a job j, the largest index $i < j$ such that job j is disjoint with job i. Define $opt(i)$ as the optimal value for instance only containing jobs $\{1, 2, \cdots, i\}$.
Before the algorithm for weighted interval scheduling, sort the jobs by non-decreasing order of finishing time first, in $O(n \lg n)$ The dynamic programming algorithm is as following:

Algorithm 4 ComputeOpt(i)

```
1: if i == 0 then
2:     return 0
3: else
4:     return max{v_i + ComputeOpt(p(i)), ComputeOpt(i − 1)}
```

For finding $p(i)$ for one job, it takes $O(\lg n)$ by binary search. For n jobs the complexity will be $O(n \lg n)$.
The running time of this algorithm can be exponential in n, if each time $ComputeOpt(i)$ is computed repeatedly. However, if we store the value of each $ComputeOpt(i)$ and reuse it, we can reduce the running time to $O(n)$.
We can recover the set of jobs for given (valid) $opt(i)$ by the following algorithm, assuming the jobs has been sorted by non-decreasing order of finishing time.

Algorithm 5 RecoverJobs()

```
1: Compute p_1, p_2, ··· , p_n
2: opt(0) ← 0
3: for i ← 1 to n do
4:     if opt(i − 1) ≥ v_i + opt(p_i) then
5:         opt(i) ← opt(i − 1)
6:         b[i] ← N
7:     else
8:         opt(i) ← v_i + opt(p_i)
9:         b[i] ← Y
10: i ← n, S ← ∅
11: while i ≠ 0 do
12:     if b[i] == N then
13:         i ← i − 1
14:     else
15:         S ← S ∪ {i}
16:         i ← p_i
17: return S
```

The above algorithm is using memorized recursion to solve the problem, there is a second efficient algorithm to solve the Weighted Interval Scheduling Problem.

Subset Sum Problem

Given an integer bound $W > 0$ and a set of n items, each with an integer weight $w_i > 0$, find a subset S of items that

$$\max \quad \sum_{i \in S} w_i \tag{2.2}$$

$$\text{s.t.} \quad \sum_{i \in S} w_i \leq W \tag{2.3}$$

Consider this instance, i items, $(w_1, w_2, \cdots, w_i)$, budget is W'. For $opt[i, W']$ there can only be one of the following cases:
Case 1: The value of optimum solution does not contain w_i, then $opt[i, W'] = opt[i-1, W']$, else
Case 2: The value of optimum solution contains w_i, then $opt[i, W'] = opt[i-1, W'-w_i] + w_i$
The algorithm is as following

Algorithm 6 Optimum Subset

```
for W' ← 0 to W do
    opt[0, W'] ← 0
for i ← 1 to n do
    for W' ← 0 to W do
        opt[i, W'] ← opt[i − 1, W']
        b[i, W'] ← N
        if w_i ≤ W' and opt[i − 1, W' − w_i] + w_i ≥ opt[i, W'] then
            opt[i, W'] ← opt[i − 1, W' − w_i] + w_i
            b[i, W'] ← Y
return opt[n, W]
```

Algorithm 7 Recover the Optimum Set

```
i ← n, W' ← W, S ← 0
while i > 0 do
    if b[i, W'] == Y then
        W' ← W' − w_i
        S ← ∪{i}
    i ← i − 1
return S
```

Optimum Binary Search Tree

Given n elements $e_1 < e_2 < \cdots < e_n$, e_i has frequency f_i, the goal is to build a binary search tree for $\{e_1, e_2, \cdots, e_n\}$ with the minimum accessing cost

$$\sum_{i=1}^{n} f_i d_i \tag{2.4}$$

Where d_i is the depth of e_i in the tree.
Suppose we choose e_k to be the root, than $e_1, e_2, \cdots, e_{k-1}$ are in left-sub tree, and $e_{k+1}, \cdots, e_n$ will be in right-sub

tree, then, denote the cost for the tree and subtrees to be C, C_L, C_R respectively

$$C = \sum_{j=1}^{n} f_j d_j \tag{2.5}$$

$$= \sum_{j=1}^{n} f_j + \sum_{j=1}^{n} f_j (d_j - 1) \tag{2.6}$$

$$= \sum_{j=1}^{n} f_j + \sum_{j=1}^{k-1} f_j (d_j - 1) + \sum_{j=k+1}^{n} f_j (d_j - 1) \tag{2.7}$$

$$= \sum_{j=1}^{n} f_j + C_L + C_R \tag{2.8}$$

Denote $opt(i, j)$ to be the optimal value for the instance of $(e_i, e_{i+1}, \cdots, e_j)$, then, for every i, j such that $1 \leq i \leq j \leq n$

$$opt(i, j) = \sum_{k=i}^{j} f_k + \min_{k: i \leq k \leq j} \{opt(i, k-1) + opt(k+1, j)\} \tag{2.9}$$

Here is an example

Example. Consider the following optimum binary search tree instance. We have 5 elements e_1, e_2, e_3, e_4 and e_5 with $e_1 < e_2 < e_3 < e_4 < e_5$ and their frequencies are $f_1 = 5, f_2 = 25, f_3 = 15, f_4 = 10$ and $f_5 = 30$. Recall that the goal is to find a binary search tree for the 5 elements so as to minimize $\sum_{i=1}^{5} \text{depth}(e_i) f_i$, where $\text{depth}(e_i)$ is the depth of the element e_i in the tree. You need to output the best tree as well as its cost. You can try to complete the following tables and show the steps. In the two tables, $opt(i, j)$ is the cost of the best tree for the instance containing $e_i, e_{i+1}, ..., e_j$ and $\pi(i, j)$ is the root of the best tree.

$opt(i,j)$ i \ j	1	2	3	4	5
1	5	35	65	95	170
2		25	55	85	155
3			15	35	90
4				10	50
5					30

$\pi(i,j)$ i \ j	1	2	3	4	5
1	1	2	2	2	3
2		2	2	2 (or 3)	3
3			3	3	5
4				4	5
5					5

Table 2.1: *opt* and π tables for the optimum binary search tree instance. For cleanness of the table, we assume $opt(i, j) = 0$ if $j < i$ and there are not shown in the left table.

$$
\begin{aligned}
opt(1,2) &= \min\{0 + opt(2,2), opt(1,1) + 0\} + (f_1 + f_2) = \min\{25, 5\} + 5 + 25 = 35 \\
opt(2,3) &= \min\{0 + opt(3,3), opt(2,2) + 0\} + (f_2 + f_3) = \min\{15, 25\} + 25 + 15 = 55 \\
opt(3,4) &= \min\{0 + opt(4,4), opt(3,3) + 0\} + (f_3 + f_4) = \min\{10, 15\} + 15 + 10 = 35 \\
opt(4,5) &= \min\{0 + opt(5,5), opt(4,4) + 0\} + (f_4 + f_5) = \min\{30, 10\} + 10 + 30 = 50 \\
opt(1,3) &= \min\{0 + f(2,3), f(1,1) + f(3,3), f(1,2) + 0\} + (f_1 + f_2 + f_3) \\
&= \min\{55, 20, 35\} + 5 + 25 + 15 = 65 \\
opt(2,4) &= \min\{0 + f(3,4), f(2,2) + f(4,4), f(2,3) + 0\} + (f_2 + f_3 + f_4) \\
&= \min\{35, 35, 55\} + 25 + 15 + 10 = 85 \\
opt(3,5) &= \min\{0 + f(4,5), f(3,3) + f(5,5), f(3,4) + 0\} + (f_3 + f_4 + f_5) \\
&= \min\{50, 45, 35\} + 15 + 10 + 30 = 90 \\
opt(1,4) &= \min\{0 + f(2,4), f(1,1) + f(3,4), f(1,2) + f(4,4), f(1,3) + 0\} \\
&\quad + (f_1 + f_2 + f_3 + f_4) = 95 \\
&= \min\{85, 40, 45, 65\} + 5 + 25 + 15 + 10 \\
opt(2,5) &= \min\{0 + f(3,5), f(2,2) + f(4,5), f(2,3) + f(5,5), f(2,4) + 0\} \\
&\quad + (f_2 + f_3 + f_4 + f_5) = 155 \\
&= \min\{90, 75, 85, 85\} + 25 + 15 + 10 + 30 \\
opt(1,5) &= \min\{0 + f(2,5), f(1,1) + f(3,5), f(1,2) + f(4,5), f(1,3) + f(5,5) + f(1,4) + 0\} \\
&\quad + (f_1 + f_2 + f_3 + f_4 + f_5) \\
&= \min\{155, 95, 85, 95, 95\} + 5 + 25 + 15 + 10 + 30 = 170
\end{aligned}
$$

Matrix Chain Multiplication

Given n matrices $A_1, A_2, \cdots, A_n$ of sizes $r_1 \times c_1, r_2 \times c_2, \cdots, r_n \times c_n$ where $c_i = r_{i+1}$ for every $i = 1, 2, \cdots, n-1$. The Matrix Chain Multiplication finds the order of computing $A_1 A_1 \cdots A_n$ with the minimum number of multiplications. The idea is as following. Assume the last step in the multiplication is $(A_1 A_2 \cdots A_i)(A_{i+1} A_{i+2} \cdots A_n)$. The cost of this step will be $r_i \times c_i \times c_n$. So we need to optimally solve two sub-instances, i.e., $(A_1 A_2 \cdots A_i)$ and $(A_{i+1} A_{i+2} \cdots A_n)$.
Denote $opt[i, j]$ as the minimum cost of computing $A_i A_{i+1} \cdots A_j$, then

$$
opt[i,j] = \begin{cases} 0, & i = j \\ \min_{k:i \le k < j}(opt[i,k] + opt[k+1,j] + r_i \times c_k \times c_j), & j < j \end{cases} \tag{2.10}
$$

The algorithm is as following

Algorithm 8 MatrixChainMultiplication

```
1: opt[i, i] ← 0   ∀ i = 1, 2, ··· , n
2: for l ← 2 to n do
3:     for i ← 1 to n − l + 1 do
4:         j ← i + l − 1
5:         opt[i, j] ← ∞
6:         for k ← i to j − 1 do
7:             if opt[i, k] + opt[k + 1, j + r_i c_k c_j < opt[i, j] then
8:                 opt[i, j] ← opt[i, k] + opt[k + 1, j] + r_i c_k c_j
9:                 π[i, j] ← k
10: return opt[1, n]
```

With above algorithm, to construct the optimal solution, the follow algorithm is needed

2.4 Compare between three paradigms

Algorithm 9 PrintOptimalOrder(i, j)

```
1: if i == j then
2:     Print(A_i)
3: else
4:     Print("(")
5:     PrintOptimalOrder(i, π[i, j])
6:     PrintOptimalOrder(π[i, j] + 1, j)
7:     Print(")")
```

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

The idea of quicksort is to recursively divide an array into two subarrays, one with smaller number and one with large number, and concatenate the subarrays after all subarrays are singletons.
The most ideal way is to divide the subarrays equally, which requires an algorithm to find the median of an array of size n in $O(n)$ time.
The quicksort algorithm is as following

Algorithm 10 Quicksort(A, n)

1: **if** $n = 1$ **then return** A
2: Initial, $A_L \leftarrow \emptyset, A_H \leftarrow \emptyset$
3: $x \leftarrow$ `Median(A)`
4: **for** $element \in A$ **do**
5: **if** $element \leq x$ **then**
6: $A_L \leftarrow A_L \cup \{element\}$
7: **else**
8: $A_H \leftarrow A_H \cup \{element\}$
9: $B_L \leftarrow$ `Quicksort(`A_L`, `A_L`.size)`
10: $B_H \leftarrow$ `Quicksort(`A_H`, `A_H`.size)`
11: $t \leftarrow$ number of times $element$ appear in A **return** $B_L + element^t + B_H$

Running time $T(n) = 2T(n/2) + O(n)$, $T(n) = O(n \lg n)$
For the median finding algorithm

Algorithm 11 Median(A)

1: (To be finished)

If we don't use the median finding algorithm, we can modify the `Quicksort(A, n)` to be a random algorithm by replacing line 3 by $x \leftarrow$ `RandomElement(A)`. This modified algorithm has an expected running time of $O(n \lg n)$. The worse case running time is $O(n^2)$.

Based on Quicksort algorithm, we can define an $O(n)$ algorithm to find the ith smallest number in A, given that we have an $O(n)$ algorithm to find median of array.

The selection algorithm is as follows

Algorithm 12 Selection(A, n, i)

```
1:  if n = 1 then
2:      return A
3:  else
4:      x ← Median(A)
5:      for element ∈ A do
6:          if element ≤ x then
7:              A_L ← A_L ∪ {element}
8:          else
9:              A_H ← A_H ∪ {element}
10:     if i ≤ A_L.size then
11:         return Selection(A_L, A_L.size, i)
12:     else if i > n − A_R.size then
13:         return Selection(A_R, A_R.size, i − (n − A_R.size))
14:     else
15:         return x
```

Similarly, without `Median(A)`, we can replace line 4 by $x \leftarrow$ `RandomElement(A)`. Then the expected running time will be $O(n)$

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

Merge sort is a typical divide and conquer algorithm. It recursively separate an array into two subarrays, and sort while merging them. The algorithm is as following

Algorithm 13 MergeSort(A, n)

1: **if** $n = 1$ **then return** A
2: **else**
3: $B \leftarrow$ `MergeSort(`$A[0..\lfloor n/2 \rfloor], \lfloor n/2 \rfloor$`)`
4: $C \leftarrow$ `MergeSort(`$A[\lceil n/2 \rceil..n], \lceil n/2 \rceil$`)`
return `Merge(`$B, C, \lfloor n/2 \rfloor, \lceil n/2 \rceil$

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 14 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 15 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

Chapter 7

Data Structures

Chapter 8

NP and Computational Intractability

8.1 P, NP and Co-NP

Definition 8.1.1 (Decision Problem)**.** A problem X is a **decision problem** if the output is either 0 or 1 (yes/no). Further, a

The input of a problem can always be encoded into a binary string. The size of an input is the length of the encoded string s for the input.

Notice: For optimization problem X, we can always define a decision version X', by giving a threshold and ask if the objective function can satisfy that threshold or not. If that decision version X' can be solved in polynomial time, we can solve the original problem X in polynomial time.

Definition 8.1.2 (Polynomial running time)**.** An algorithm A has a **polynomial running time** if there is a polynomial function $p(\cdot)$ such that $\forall s$, the algorithm A terminates on s in at most $p(|\cdot|)$ steps.

Definition 8.1.3 (P)**.** The complexity class P is the set of decision problems X that can be solved in polynomial time.

Example. Shortest path, minimum spanning tree, determine if an integer is prime number, those are in P.

Definition 8.1.4 (Certificate, Certifier)**.** In order to check the an algorithm A, for a binary string s, such that $s \in A$, there is another separated algorithm B uses s and t, a separated string that contains the evidence that s is a "yes" instance of X, as input string and output (another) "yes". Then this string t is called **certificate**, this separated algorithm $B(s,t)$ is called **certifier**.

Notice: Certificate is like a solution, and certifier is the algorithm to prove solution is correct. But both with fancy terminology

Example. For Independent set problem. The input (s) is a graph, the certificate (t) will be a set of size k, and the certifier will be an algorithm to check the given set is an independent set.

Example. For 3-SAT problem. The input s will be the 3-CNF (conjunctive normal form), the certificate t will be the assignment of true values for each terms in the 3-CNF. The certifier will be the algorithm calculates the true value of the clause given the true values in t.

Definition 8.1.5 (Efficient Certifier)**.** B is an efficient certifier for problem X if

- B runs in polynomial time with input s and t. (s is the input of origin algorithm and t is the certificate)
- There is a polynomial function $p(|\cdot|)$ such that for every string s, we have $s \in X$ (s is a "yes" solution for X) and $B(s,t) = yes$ (the certifier returns "yes")

Definition 8.1.6 (NP)**.** The complexity class NP is the set of all problems for which there exists an efficient certifier.

From the definition of NP we can immediately know that $P \subseteq NP$. Because all the problems in P can satisfy the definition of NP. But is there a problem that $X \in NP$ and $X \notin P$? Solve that problem and you will win 100 million dollars and the chance to be remember forever.

Notice: NP stands for **Non-deterministic Polynomial** time

To prove if a problem is NP, we need to have a certifier that can output "yes" given the certificate in polynomial time. Otherwise it is not in NP

Example. Given a graph $G = (V, E)$ and an integer $t > 0$, whether the minimum vertex cover of G has size at least t. This problem is **unlikely** in NP. Reason is, the "yes" instance is "all minimum vertex cover has size at least t", that is not **likely to be** solvable in polynomial time for it need to find all minimum vertex covers.

Definition 8.1.7 (Co-NP)**.** For a problem X, the problem $\bar{X}$ is the problem such that $s \in \bar{X} \iff s \notin X$. Then **Co-NP** is the set of decision problems X such that $\bar{X} \in NP$.

Example (Tautology Problem)**.** Given a boolean formula, determine wither the formula is always evaluates to 1. This is a problem in Co-NP. Because we can have a polynomial time certifier to confirm that an instance is not a tautology.

Example. Given two boolean formulas, to determine whether or not they are equivalent. This is a problem in Co-NP. Because if we have one instance such that the output is "no", then we can easily prove there are counter examples in origin algorithm.

Notice: If the instance is "easy" to prove to be true, then it is in NP, if the counter instance is "easy" to prove to be the algorithm it be false, then it is in Co-NP

Relation between P, NP and $Co - NP$ is as following

- $P \subseteq NP$
- $P \subseteq Co - NP$
- $P = NP$? is not known
- $P = Co - NP$? is not known
- $NP = Co - NP$? is not known
- If $P = NP$ then $P = Co - NP$

8.2 Polynomial-Time Reductions

Definition 8.2.1 (Polynomial-time reducible)**.** Given an algorithm A that solves problem Y, if any instance of problem X can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to algorithm A, then we say X is **polynomial-time reducible** to Y, denoted as

$$X \leq_P Y \tag{8.1}$$

In a more intuitive way, $X \leq_P Y$ means X can't be more difficult than Y. Solving X can be "transforming" X into an equivalent Y in polynomial number of steps and then solve it by calling Y polynomial number of times, usually one time. Thus, if $X \leq_P Y$,

- If Y can be efficiently solved, X can be efficiently solved.
- If X cannot be efficiently solve, Y cannot be efficiently solved.

Notice: To prove $X \leq_P Y$, usually we already have an algorithm for Y, could be polynomial or not.

Lemma 8.1. *Hamiltonian-Path* $\leq_P$ *Hamiltonian-Cycle*

Proof. For graph $G = (V, E)$, s and t are two vertices that in V, define a new graph $G' = (V \cup \{v\}, E \cup \{(u, s)\} \cup \{(t, u)\}$. To solve the Hamilton-Path from s to t in graph G, say p_{st}, is equivalent to solving Hamilton-Cycle problem in G', i.e., find $sp_{st}te_{tu}ue_{us}$. □

Lemma 8.2. *Hamiltonian-Cycle* $\leq_P$ *Hamiltonian-Path*

Proof. For vertex s, make a copy and denote it as s', s' is connected to all the vertices that s connected. Solving the Hamiltonian-Cycle problem is equivalent to solving the Hamiltonian-Path problem from s to s' □

Lemma 8.3. *Hamiltonian-Path* $\leq_P$ *degree-3 spanning tree*

Proof. In graph G, for vertex s and t, add vertices s', s'', t', t'' and edges (s, s'), (s, s''), (t, t'), (t, t''). For all the other vertex $u \in V \setminus \{s\} \setminus \{t\}$, add vertices u' and edge (u, u') to the graph. Then solving Hamiltonian-Path problem is equivalent to solve the degree spanning tree problem in this new G. □

Lemma 8.4. *Vertex Cover* $\leq_P$ *Set Cover*

Proof. □

Lemma 8.5. *Set Cover* $\leq_P$ *Vertex Cover*

Proof. □

Lemma 8.6. *Clique* $\leq_P$ *Independent Set*

Proof. S is an independent set in $G = (V, E)$ iff S is a clique in $\bar{G} = (V, \bar{E})$ □

Lemma 8.7. *Independent Set* $\leq_P$ *Clique*

Proof. S is a clique in $G = (V, E)$ iff S is an independent set in $\bar{G} = (V, \bar{E})$ □

Lemma 8.8. *3-Coloring* $\leq_P$ *4-Coloring*

Proof. For a graph $G = (V, E)$, define a new graph $G' = (V \cup \{u\}, E \cup \{(u, v) | \forall\ v \in V\})$. Solving the 3-Coloring problem is equivalent to solving the 4-Coloring problem in G' □

Lemma 8.9. *Independent Set* $\leq_P$ *Set Packing*

8.3 NP-Completeness

Definition 8.3.1 (NP-Completeness)**.** A problem X is called **NP-Complete** if

- $X \in NP$, and
- $Y \leq_P X, \forall\ Y \in NP$

An intuitive explanation will be, we can regard problems that is NP-Complete to be the most difficult problems in NP. If any of those can be solved in polynomial time, then all problems in NP can be solved in polynomial time.

Theorem 8.10 (Cook's Theorem)**.** *Circuit Satisfiability is NP-Complete.*

Proof. □

Theorem 8.11. *If X is NP-Complete and $X \in P$, then $P = NP$*

Proof. Direct result from Cook's theorem. □

8.4 NP-Complete Problems