MIT6.046J Design and Analysis of Algorithms

Xiyuan Yang
2025.10.23

Lecture Notes for advanced algorithms for Open lecture MIT 6.046J

Contents

§1. Introduction

§1.1. Course Overview

1. Divide and Conquer - FFT, Randomized algorithms
2. Optimization - greedy and dynamic programming
3. Network Flow
4. Intractibility (and dealing with it)
5. Linear programming
6. Sublinear algorithms, approximation algorithms
7. Advanced topics

§1.2. Complexity Recall

- **P**: class of problems **solvable** in polynomial time. $O(n^k)$ for some constant k.
  - P 类问题可以使用确定性图灵机在多项式时间内解决的问题集合
- **NP**: class od problems **verifiable** in polynomial time.

> **Example** (Hamiltonian Cycle).
> Find a simple cycle to contain each vertex in V.
> - Easy to evaluate, but hard to calculate!
>
> We have P $\subset$ NP.
> - 在多项式时间内找到正确答案,那么肯定可以在多项式时间内验证答案是否正确。(默认比较两个答案是否相同是可以在多项式实现内实现的)

- **NPC**: NP Complete
  - 问题本身属于 NP 复杂度类
  - 为 NP 困难问题:
    - 所有的 NP 问题都可以在多项式时间内归约到问题 C 上。

§1.3. Interval Scheduling

Requests $1, 2, ..., n$: single resource.

- $s(i)$: the start time
- $f(i)$: the finish time
- $s(i) < f(i)$
- two requests are compatible: $[s(i), f(i)] \cap [s(j), f(j)] \neq \emptyset$

Goal: select a compatible subset of requests with the maximum size.

Solving: Greedy Search!

- Use a simple rule to select a request i.
- Reject all requests incompatible with i.
- Repeat until all requests are processed.

§2. Divide and Conquer

§2.1. Paradigm

Intuition: Splitting bigger problems into smaller problems.

- Solve the sub-problems recursively
- Combine solutions of sub-problems to get overall solutions.

$$T(n) = aT\left(\frac{n}{b}\right) + [\text{work for merge}]$$

- a: The number of sub-problems during recursion.
- b: The size of each sub-problems

For example, for the merge sort:

$$T(n) = aT\left(\frac{n}{2}\right) + O(n)$$

§2.2. Convex Hall

§2.2.1. Brute force for Convex Hull

C_n^2 segments, testing each segment:

- All other points are on the single side: correct
- Else: false

Time Complexity: $O(n^3)$

§2.2.2. Gift Wrapping Algorithms

Given n points in the plane, the goal is to find the smallest polygon containing all points in $S = \{(x_i, y_i) | i = 1, 2, ..., n\}$. We ensure no two points share the same x coordinates or the y coordinates, no three points are in the same line.

Intuition: Gift wrapping algorithms.

Recordings (Simple Gift Wrapping Algorithms).
- Select the initial point
  - Find the point which has the smallest x coordinates or the smallest y coordinates.
- 找到旋转角度最大的点，作为凸包上的点选入
  - 这也可以看做是一种橡皮筋手搓生成凸包的过程
- Time Complexity: $O(n \cdot h)$

§2.2.3. Divide and Conquer for Convex Hall

Recordings (When to use Divide and Conquer).
- 分治最关键的是两个步骤：
  - 分解成若干个子问题（递）
  - 把子问题的结果合并起来（归）
- 如果使用分治法，那务必重视的一点是递归的终点（最简单的情况）必须是简单可解的。($O(1)$ Time Complexity)

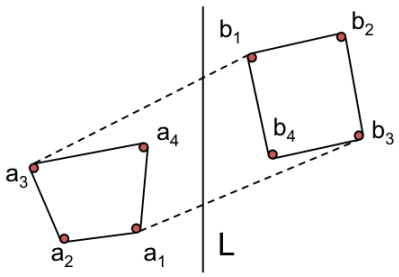
For simple condition: when $n \leq 3$, the convex hall is quite simple! All the points are the vertices of the convex hall.

Now, we need to solve two things:

- When to divide
  - With x coordinates
  - More like half splitting!
- When to conquer
  - The most critical step!
  - We need to finding the bridges (Upper Bridge and Lower Bridge) to form the bigger convex hall.
    - **Two Finger Algorithms**

Recordings (Two Finger Algorithms).
- 基本思路类似于双指针法实现线性扫描
- 基本思想还是不断旋转找到最外部的切线

Example



a_3, b_1 is upper tangent. $a_4 > a_3$, $b_2 > b_1$ in terms of Y coordinates.
a_1, b_3 is lower tangent, $a_2 < a_1$, $b_4 < b_3$ in terms of Y coordinates.

a_i, b_j is an upper tangent. Does not mean that a_i or b_j is the highest point. Similarly, for lower tangent.

Figure 1: Convex Hall Conquering Steps

Time Complexity:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Thus total time complexity: $\Theta(n \log n)$

For the compute of this time complexity, we can use Master Theorem.

§2.3. Master Theorem

For simple cases:

$$T(n) = aT\left(\frac{n}{b}\right)$$

To compute this complexity, we use the recursive tree to solve this:

$$T(n) = a^k T\left(\frac{n}{b^k}\right)$$

For the recursion endpoint, $\frac{n}{b^k} = 1$, we can compute:

$$T(n) = a^{\log_b n} T(1) = n^{\log_b a} T(1) = O\left(n^{\log_b a}\right)$$

For general cases:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

We need to compare $f(n)$ and $n^{\log_b a}$.

§2.3.1. The Work at the Leaves Dominates

$$f(n) = O\left(n^{\log_b (a-\varepsilon)}\right)$$

Then it means that recursion part dominates! ($T(n) = aT\left(\frac{n}{b}\right)$). Thus, the time complexity is:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) = aT\left(\frac{n}{b}\right) + O\left(n^{\log_b (a-\varepsilon)}\right) = \Theta\left(n^{\log_b a}\right)$$

§2.3.2. The Work is Balanced

$$f(n) = \Theta\left(n^{\log_b a} \cdot \log^k n\right)$$

Then it means the two parts are both the dominant parts! The total time complexity remains the same.

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) = aT\left(\frac{n}{b}\right) + \Theta\left(n^{\log_b a} \cdot \log^k n\right) = \Theta\left(n^{\log_b a} \cdot \log^{k+1} n\right)$$

§2.3.3. The Work at the Root Dominates

$$f(n) = \Omega\left(n^{\log_b a+\varepsilon}\right)$$

and:

$$\exists c \in R, \exists N_0 \in \mathbb{N}, \forall n > N_0 : af\left(\frac{n}{b}\right) \leq cf(n)$$

Recordings (Regular Condition).

- 这个条件说明划归到子问题的时候时间复杂度可能很大，但是对于大问题“分而治之”的复杂度是非常昂贵的。
- 总复杂度有递归的最高层（根节点）的代价决定，这也保证该情况下时间复杂度的量级为 $\Theta(f(n))$

Then the total time complexity:

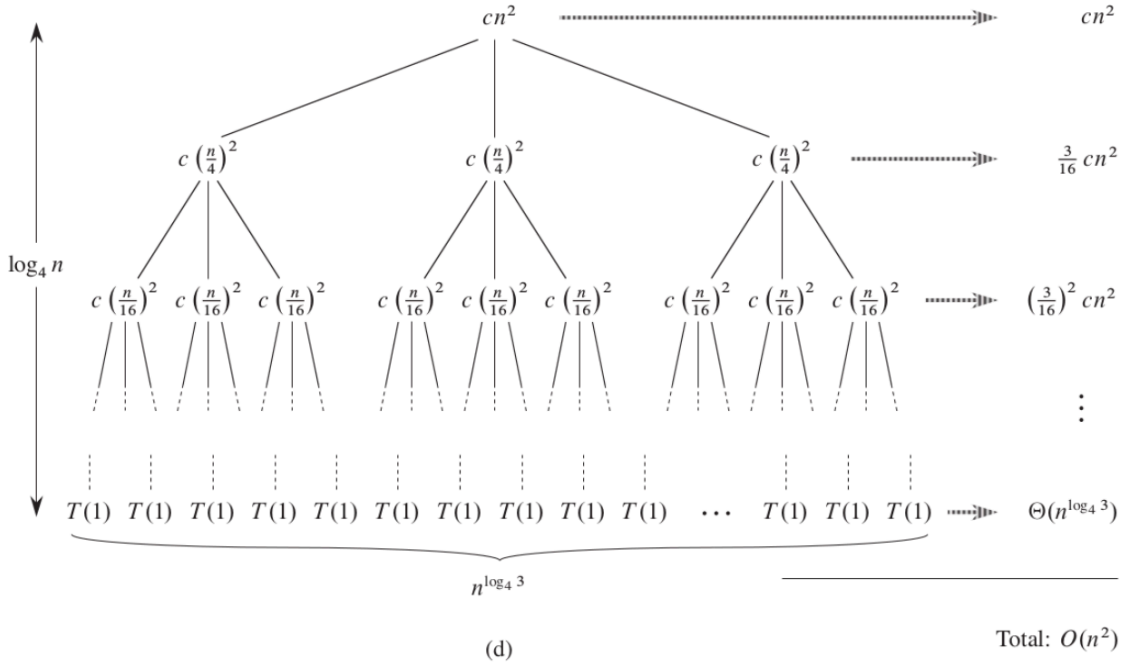
$$T(n) = \Theta(f(n))$$

Example (Example for the work at the root dominates).
例如如果递归的时间复杂度为：

$$T(n) = 3T\left(\frac{n}{4}\right) + n^2$$

$a = 3, b = 4$, $T(n) = \Theta(n^2)$

§2.3.4. Solving Master Theorem Using Recursive Tree



- 每一个叶子结点代表被递归分解的小问题的规模
- 每一个非叶子节点的时间复杂度为 $cf(n')$ (n' 代表当前层数 k 的规模 $\frac{n}{b^{k-1}}$)
- 最终总的时间复杂度就是这个递归树的所有节点的时间复杂度的总和
- 因此，只需要把这个递归树展开到最底层，并计算所有结点的代价，就是最终的时间复杂度。

使用等比数列求和：

$$T(n) = \Theta\left(n^{\log_b a}\right) + \sum_{j=0}^{\log_b(n)-1} a^j f\left(\frac{n}{b^j}\right)$$

这就是为什么主定理需要比较 f 和这个的大小，在这里仅仅做主要阐述：特别考虑 boundary $f(n) = \Theta\left(n^{\log_b a}\right)$

$$T(n) = \Theta\left(n^{\log_b a}\right) + \sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a} = \log_b n \times n^{\log_b a} = \Theta\left(n^{\log_b a} \log n\right)$$

§2.4. Median Finding

Problem 2.4.1 (Median Finding).
Given set of n numbers, define rank(x) as number of numbers in the set that are $\leq x$. Find element of rank $\lfloor \frac{n+1}{2} \rfloor$ (lower median) and $\lceil \frac{n+1}{2} \rceil$ (upper median).

Obviously, we can use **sorting algorithms** to solve this! The time complexity is $\Theta(n \log n)$.

Simple Algorithms: Define problem as `Select(S,i)` to find the `i` th element value in the set S.

- Pick $x \in S$
  - We just pick it cleverly
- Compute $k = \text{rank}(x)$
- $B = \{y \in S | y < x\}$
- $C = \{y \in S | y > x\}$
- algorithms:
  - If $k = i$: `return x`
  - If $k < i$: `return Select(C,i-k)`
  - If $k > i$: `return Select(B,i)`

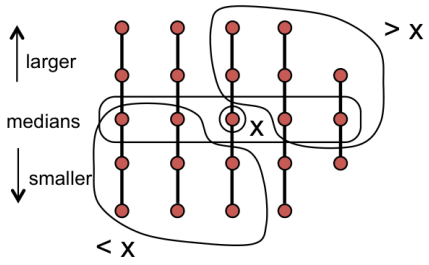
For dummy choices for selecting $x \in S$, for the worse case, the time complexity is $\Theta(n^2)$.

§2.4.1. Picking x cleverly

- Arrange S into columns of size 5 ($\lceil \frac{n}{5} \rceil$ cols).
- Sort each columns in linear time.
- Find **medians of medians** as the selected x.

Recordings (Why selecting this?).

- 对于简单的取常数或者中间值的方法在极端情况下会退化到平方时间复杂度，因为我们难以知道全局数据的分布特征，因此我们很难选择一个好的 splitting
- 和快速排序很类似！我们希望选择一个好的 splitting ,这样让递归算法变成对数级别的。
- 而下面的选择可以保证 splitting 的效率，即至少有 $3\left(\lceil \frac{n}{10} \rceil - 2\right)$ 的点被分到左边并且至少有 $3\left(\lceil \frac{n}{10} \rceil - 2\right)$ 的点被分到右边。



How many elements are guaranteed to be $> x$?
Half of the $\lceil \frac{n}{5} \rceil$ groups contribute at least 3 elements $> x$ except for 1 group with less than 5 elements and 1 group that contains x.
At lease $3(\lceil \frac{n}{10} \rceil - 2)$ elements are $> x$, and at least $3(\lceil \frac{n}{10} \rceil - 2)$ elements are $< x$

Figure 3: SELECT for medians of medians

Recurrence:

$$T(n) = T\left(\lceil \frac{n}{5} \rceil\right) + T\left(\frac{7n}{10} + 6\right) + \Theta(n)$$

- $T\left(\lceil \frac{n}{5} \rceil\right)$ 是找到中位数的中位数的算法时间
- $\Theta(n)$ 是分组线性扫描需要的时间复杂度
- $\frac{7n}{10} + 6$ 代表子问题的规模，因为我们保证$3\left(\lceil \frac{n}{10} \rceil - 2\right)$ 会被分到对应的组，因此最坏情况就是 $\frac{7n}{10} + 6$

Solving this recurrence.

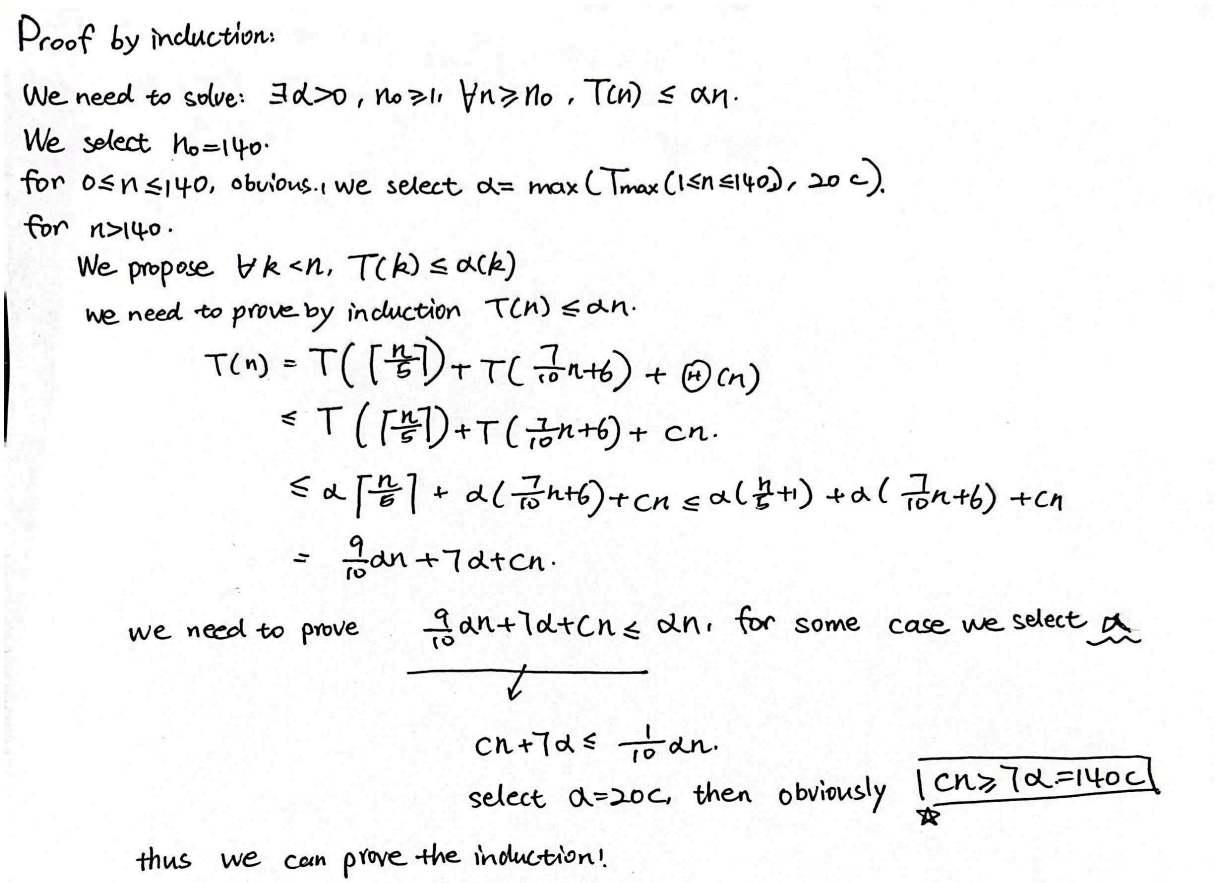


Figure 4: Induction proof for median finding algorithms

Example (Medians of Medians).
给定一个数组和 k 值，尝试求解这个数组中的第 k 大的元素。要求保证时间复杂度为 $O(n)$.

```
from typing import List

class Solution:
    def find_median_of_small_array(self, arr: List[int]) -> int:
        # O(1) for constant length arrays
        arr.sort()
        return arr[len(arr) // 2]

    def select_pivot(self, arr: List[int]) -> int:
        n = len(arr)
        if n <= 5:
            return self.find_median_of_small_array(arr)

        # splitting into sub-lists
        sublists = [arr[i : i + 5] for i in range(0, n, 5)]
        medians = [self.find_median_of_small_array(sublist) for sublist in sublists]
```

```
        # ! very important recursion!
        # That is the T(n/5) part
        return self.findKthSmallest(medians, len(medians) // 2 + 1)

    def findKthSmallest(self, arr: List[int], k: int) -> int:
        n = len(arr)
        if n == 1:
            return arr[0]

        pivot = self.select_pivot(arr)

        # do partition based on selected pivot
        less = [x for x in arr if x < pivot]
        equal = [x for x in arr if x == pivot]
        greater = [x for x in arr if x > pivot]

        len_less = len(less)
        len_equal = len(equal)

        # the core recursion part remains unchanged
        if k <= len_less:
            return self.findKthSmallest(less, k)
        elif k <= len_less + len_equal:
            return pivot
        else:
            new_k = k - len_less - len_equal
            return self.findKthSmallest(greater, new_k)

    def findKthLargest(self, nums: List[int], k: int) -> int:
        n = len(nums)
        k_smallest = n - k + 1
        return self.findKthSmallest(nums, k_smallest)
```

§2.5. Matrix Multiplication

For simple matrix multiplication, the time complexity is $O(n^3)$.

$$c_{i,j} = \sum_{k=1}^{p} a_{i,k} b_{k,j}$$

- n^3 times multiplication.
- $n^3 - n^2$ times addiction.

§2.5.1. Strassen Algorithms

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

For simple divide and conquer algorithms:

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

这个是基本的分治算法，对于子矩阵，需要进行 8 次 子矩阵的乘法和 4 次子矩阵的加法。

$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$$

Based on Master theorem, the time complexity is $O(n^3)$, remains unchanged!

The break through for strassen algorithms are reducing matrix multiplication from 8 times into 7 times by reducing repeated computation!

$$\begin{aligned} M_1 &= (A_{11} + B_{22})(B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22})B_{11} \\ M_3 &= A_{11}(B_{12} - B_{21}) \\ M_4 &= A_{22}(B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12})B_{22} \\ M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

7 times matrix multiplication ($\frac{n}{2} \times \frac{n}{2}$), and 18 times addiction.

$$\begin{aligned} C_{11} &= M_1 + M_4 - M_5 + M_7 \\ C_{12} &= M_3 + M_5 \\ C_{21} &= M_2 + M_4 \\ C_{22} &= M_1 - M_2 + M_3 + M_6 \end{aligned}$$

Thus the time complexity:

$$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2) = \Theta\left(n^{\log_2 7}\right) \approx \Theta(n^{2.807})$$

§2.6. FFT

§2.6.1. Polynomials

All about polynomial:

$$A(x) = \sum_{k=1}^{n-1} a_k x^k = [a_0, a_k, ..., a_{n-1}]$$

§2.6.2. Operations for $A(x)$

§2.6.2.1. Evaluation

Definition 2.6.2.1.1 (Evaluation)**.**
Given x, calculate $A(x)$.

Theorem 2.6.2.1.1 (Horner's Rule)**.**
我们希望更少次数的乘法和加法

$$A(x) = a_0 + x(a_1 + x(a_2 + ... + x(a_{n-1})))$$

- Before: $\Theta(n^2)$ times multiplication and $\Theta(n)$ times addition.

- After: $\Theta(n)$ times multiplication and $\Theta(n)$ tines addition.
- Thus the time complexity: $O(n^2) \to O(n)$

§2.6.2.2. Addition

Definition 2.6.2.2.1 (Addition).

$$C(x) = A(x) + B(x)$$

Obviously, the time complexity is $O(n)$ for addition.

§2.6.2.3. Multiplication

Definition 2.6.2.3.1 (Multiplication).

$$C(x) = A(x) \times B(x), \forall x \in X$$

- Naive calculation: $O(n^2)$

$$c_k = \sum_{j=0}^{K} a_j b_{K-j}$$

We want to achieve $O(n \log n)$

Algorithms vs.	Representations		
	Coefficients	Roots	Samples
Evaluation	$O(n)$	$O(n)$	$O(n^2)$
Addition	$O(n)$	∞	$O(n)$
Multiplication	$O(n^2)$	$O(n)$	$O(n)$

§2.6.3. Representations

- Coefficient Vectors
- Roots and a scale term
- Samples

§2.6.4. Vendermonde Matrix

$$V \cdot A = \begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \dots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \dots \\ y_{n-1} \end{pmatrix}$$

Recordings (多项式插值).
- 这个本质上也可以看做是一种多项式插值的手段
- 我们希望从 sample 的形式转变为 coefficient 的形式

- 根据线性代数的知识，范德蒙行列式只有在 sample 的点均不相同的情况下才是不可逆的，因此只要 sample 了 n 个不相同的点，就可以保证能够求解可逆矩阵，但是可逆矩阵的时间复杂度是 $O(n^3)$，因此实际插值并不会采用这个原始的算法。

$$\Pi_{0\le i<j\le n-1}(x_j - x_i) = 0$$

We want to calculate A, thus we need to calculate:

$$A = V^{-1}Y$$

§2.6.5. Divide and Conquer for FFT

The original input: A_{eff} and B_{eff} as two vectors, and we need to calculate C_{eff} for the new coefficients after polynomial multiplications.

We know, if we have N samples for two polynomials, just calculate $A(x_k) \cdot B(x_k)$.

因此，如果我们需要解决多项式乘法的问题，实际上我们可以将问题拆分为：

- 预处理计算 FFT(A) and FFT(B): we want it to be $O(N \log N)$
- $C_{\text{point}} = A_{\text{point}} \odot B_{\text{point}}$: it is $O(N)$, not the bottleneck
- IFFT(C_{point}): we want it to be $O(N \log N)$

§2.6.5.1. How to divide?

Divide into Even and Odd Coefficients $O(n)$

$$A_{\text{even}} = \sum_{k=0}^{\lceil \frac{n}{2}-1 \rceil} a_{2k}x^k = [a_0, a_2, ..., a_{2l}]$$

$$A_{\text{odd}} = \sum_{k=0}^{\lceil \frac{n}{2}-1 \rceil} a_{2k+1}x^k = [a_1, a_3, ..., a_{2l}]$$

§2.6.5.2. How to conquer

$$A(x) = A_{\text{even}}(x^2) + x \cdot A_{\text{odd}}(x^2) \text{ for } x \in X$$

Thus, we need to recursively calculate $A_{\text{even}(y)}, y \in X^2 = \{x^2 \mid x \in X\}$

$$T(n, |X|) = 2T\left(\frac{n}{2}, |X|\right) + O(n + |X|) = O(n^2)$$

Actually, the time complexity does not change... However, if we can achieve the conquer time complexity as follows, we can do a great improvement:

$$T(n, |X|) = 2T\left(\frac{n}{2}, \frac{|X|}{2}\right) + O(n + |X|)$$

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Thus, the time complexity is $O(n \log n)$! We should select X in a clever way in which it is **collapsing**, or we can say:

$$|X^2| = \frac{|X|}{2}$$

for the base case, $X = \{1\}$ when $|X| = 1$.

Recordings (Roots of Unity).
- 从数值来讲，就是在复平面内不断求根
- 在复平面内就是不断的取中间的过程

$$(\cos\theta, \sin\theta) = \cos\theta + i\sin\theta = e^{i\theta}$$
$$\theta = 0, \frac{1}{n}\tau, \frac{2}{n}\tau, ..., \frac{n-1}{n}\tau(\tau = 2\pi)$$

Then, we successfully implement FFT with the time complexity for $O(n \log n)$!

- Well defined recursion
- Selected X, for $x_k = e^{\frac{i\tau k}{n}}$

§2.6.6. Inverse Discrete Fourier Transform (IFFT)

We want to return the coefficients from the multiplied samples. The transformation of this form is $A = V^{-1}Y$. Thus, all we need to do is calculate V^{-1}, or the inverse of Vendermonde Matrix!

Theorem 2.6.6.1 (Calculate the inverse).

$$V^{-1} = \frac{1}{n}\overline{V}$$

where $\overline{V}$ is the complex conjugate of V.

Proof. We claim that $P = V \cdot \bar{V} = nI$:

$$\begin{aligned} p_{jk} &= (\text{row } j \text{ of } V) \cdot (\text{col. } k \text{ of } \bar{V}) \\ &= \sum_{m=0}^{n-1} e^{ij\tau m/n}\overline{e^{ik\tau m/n}} \\ &= \sum_{m=0}^{n-1} e^{ij\tau m/n} e^{-ik\tau m/n} \\ &= \sum_{m=0}^{n-1} e^{i(j-k)\tau m/n} \end{aligned}$$

Now if $j = k$, $p_{jk} = \sum_{m=0}^{n-1} = n$. Otherwise it forms a geometric series.

$$\begin{aligned} p_{jk} = &= \sum_{m=0}^{n-1} (e^{i(j-k)\tau/n})^m \\ &= \frac{(e^{i\tau(j-k)/n})^n - 1}{e^{i\tau(j-k)/n} - 1} \\ &= 0 \end{aligned}$$

because $e^{i\tau} = 1$. Thus $V^{-1} = \frac{1}{n}\bar{V}$, because $V \cdot \bar{V} = nI$. □

This claim says that the Inverse Discrete Fourier Transform is equivalent to the Discrete Fourier Transform, but changing x_k from $e^{ik\tau/n}$ to its complex conjugate $e^{-ik\tau/n}$, and dividing the resulting vector by n. The algorithm for IFFT is analogous to that for FFT, and the result is an $O(n \lg n)$ algorithm for IDFT.

Recordings (Inverse).
- 换句话说，范德蒙行列式是可以快速求解的，因为旋转基本根的良好性质。

§2.7. van Emde Boas Tree

Definition 2.7.1 (vEB Tree).
Goal: We want to maintain n elements of a **set** in the range $\{0, 1, 2, ..., u-1\}$ and perform **Insert**, **Delete** and **Successor** Operations in $O(\log \log u)$ time.
- Successor: 后继操作，即找到严格大于 x 的最小元素。

By using an ordered binary search tree like AVL Tree, we can implement this query in $O(\log n)$ time on average.

For example, this data structure can be used in Network Routing Tables, where u = Range of IP Addresses $\rightarrow$ port to send. (u = 2^{32} in IPv4)

§2.7.1. Intuition

Where might the $O(\log \log u)$ bound arise?

Binary search over $O(\log u)$ elements.

$$T(k) = T\left(\frac{k}{2}\right) + O(1) = \Theta(\log n)$$

$$T(\log u) = T\left(\frac{\log u}{2}\right) + O(1)$$

$$T(u) = T\left(\sqrt{u}\right) + O(1)$$

Thus, we want to find the recurrence like:

$$T(u) = T\left(\sqrt{u}\right) + O(1)$$

§2.7.2. Improvement

§2.7.2.1. Bit Vector (Hash and Bucket)

Define a vector of size u, where $V[x] = 1$ iff $x \in S$.

- Insert & Delete: $O(1)$
- Successor/Predecessor: Need to traverse all the bit vector, requires worst $O(n)$.

§2.7.2.2. Split Universe into Clusters

Recordings (It is Chunking!).
- 经典的分块思想
- 牺牲空间换取查询的更快时间复杂度
- 在这个具体的问题场景下，选择分块可以避免一些重复计算，先对整体的块进行操作而加速。

Split Universe into Clusters

We can improve performance by splitting up the range $\{0, 1, 2, \ldots, u-1\}$ into $\sqrt{u}$ clusters of size $\sqrt{u}$. If $x = i\sqrt{u} + j$, then $V[x] = V.Cluster[i][j]$.

$$low(x) = x \bmod \sqrt{u} = j$$
$$high(x) = \left\lfloor \frac{x}{\sqrt{u}} \right\rfloor = i$$
$$index(i, j) = i\sqrt{u} + j$$

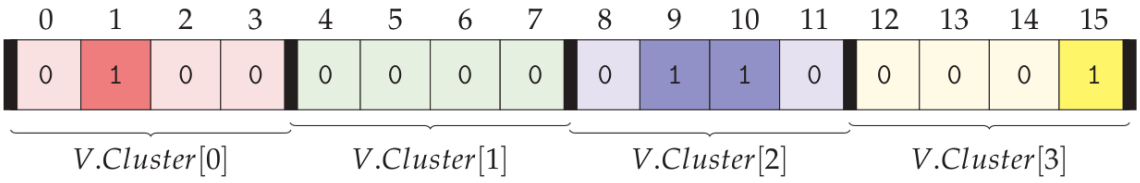


Figure 2: Bit vector ($u = 16$) split into $\sqrt{16} = 4$ clusters of size 4.

- Insert: Set `V.cluster[i][j]` to 1, then mark cluster `high(x)` as no empty. $O(1)$
- Successor: $O(\sqrt{u})$
  - Look within the cluster `i`
  - Else, find next non-empty cluster `i`
  - Find Minimum Entry `j` for that cluster
  - Return `index(i,j)`

§2.7.2.3. Recurse

- `V.cluster[i]` is a size-$\sqrt{u}$ van Emde Boas structure ($\forall 0 \le i < u$)
- `V.summary` is a size-$\sqrt{u}$ van Emde Boas structure
- `V.summary[i]` indicates whether `V.cluster[i]` is nonempty

§2.7.2.3.1. Insert

`Insert(V,x)`:

- `Insert(V.cluster[high(x)], low[x])`
- Remark that this cluster is non-empty: `Insert(V.summary, high[x])`.

Time complexity: $T(u) - 2T(\sqrt{u}) + O(1)$, thus the time complexity is $O(\log u)$.

§2.7.2.3.2. Successor

`SUCCESSOR(V, x)`:

- i = high(x)
- j = Successor(V.cluster[i], j)
  - if j == ∞
    - i = Successor(V.summary, i)
    - j = Successor(V.cluster[i], $-\infty$)
- return index(i, j)

Time complexity for the worse case:

$$T(u) = 3T(\sqrt{u}) + O(1)$$

$$T(u) = O((\log u)^{\log_2 3}) \approx O((\log u)^{1.585})$$

Recordings (Time complexity is not enough!).

- 分块的思想在 Bit Vector 的基础之上空间换时间达到了 log 级别的时间复杂度优化，但是仍然不够！
- 如何进一步优化，根据主定理，我们优化的关键在于递归的分路数 a，缩减 a 就有可能进一步缩减时间复杂度！

§2.7.2.4. Maintain Min and Max

We store the minimum and maximum entry in each structure. This gives an $O(1)$ time overhead for each Insert operation.

SUCCESSOR(V, x):

- if x < V.min return V.min
- i = high(x)
- if low(x) < V.cluster[i].max: j = Successor(V.cluster[i], low(x))
- else:
  - i = Successor(V.summary, i) $T(\sqrt{n})$
  - j = V.cluster[i].min $O(1)$
- return index(i, j)

Now the time complexity is:

$$T(n) = T(\sqrt{n}) + O(1) = O(\log \log u)$$

§2.7.2.5. Don't Store min recursively

INSERT(V, x)

```
1  if V.min == None
2      V.min = V.max = x        ▷ O(1) time
3      return
4  if x < V.min
5      swap(x ↔ V.min)
6  if x > V.max
7      V.max = x)
8  if V.cluster[high(x) == None
9      Insert(V.summary, high(x))      ▷ First Call
10 Insert(V.cluster[high(x)], low(x))    ▷ Second Call
```

If the **first call** is executed, the **second call** only takes $\mathcal{O}(1)$ time. So

$$\begin{aligned} & T(u) = T(\sqrt{u}) + \mathcal{O}(1) \\ \Longrightarrow\ & T(u) = \mathcal{O}(\log \log u) \end{aligned}$$

Figure 8: Divide and Conquer Algorithms for veb Tree insertion.

Recordings (Pay Attention to min and max).

- vEB 树最关键的地方在于每一个 vEB 树的最大值和最小值的管理是不在分块数组内部的，而是作为当前树的缓存额外存储。
- 在插入过程中，如果最小值被替换，原先的最小值需要被插入，因此可以等效的交换两者
- 如果最大值被替换，因为最大值被正常的插入，因此只需要正常的更新就可以了。
- 注意，一棵树的最小值的节点是不存储在 Bit Vector 里面的！这样是为了保证对于空树的递归调用实现不要太多次。

§2.7.2.6. Deletions Operations

Recordings (Deletions).

- 如果需要删除最小值
  - vEB 树的精妙之处，额外缓存数据结构的最小值，因此这样就可以找到第二小的元素！
    - i = V.summary.min
    - second_min = index(i, V.cluster[i].min)
  - 接着，删除这个第二小的元素，因为他不可以在 Vector Bit 中出现，然后把这个元素的值更新到最小缓存中。
- 如果删除中间值或者最大值：
  - 直接递归调用子簇 `Delete(T.cluster[i], low(x))`
- 如果删除后的子簇是空的：
  - 直接删掉 Summary 对应的部分 (Second Call)
- 如果删除最大值
  - 因为 max 是正常被储存的，因此我们需要更新 max
  - 找到新的 max：
    - 如果 Summary 是空的，那么新的 T.max 就是 T.min
    - 如果不是，就更新为最大簇的最大元素

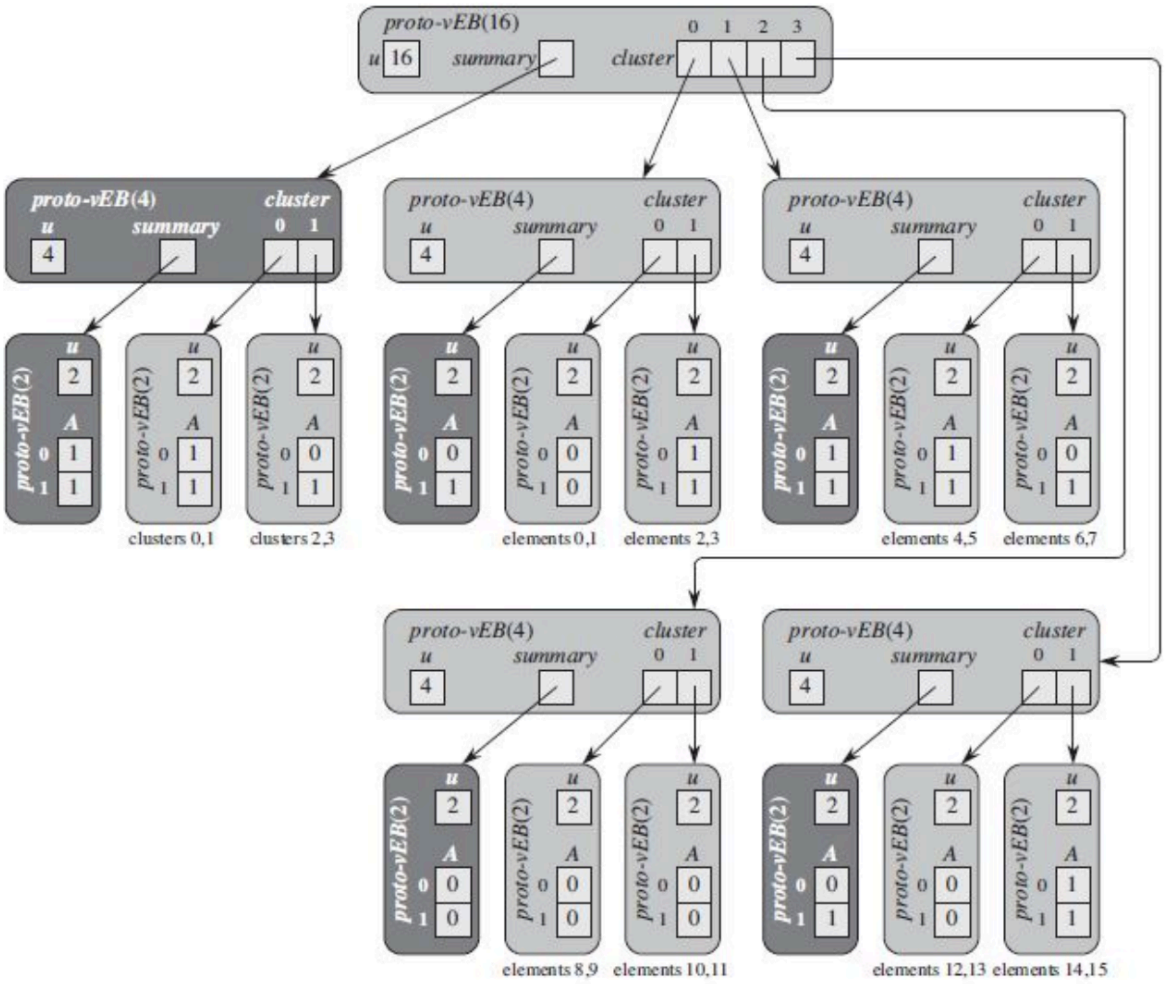


Figure 20.4 A *proto-vEB*(16) structure representing the set $\{2, 3, 4, 5, 7, 14, 15\}$. It points to four *proto-vEB*(4) structures in *cluster*[0..3], and to a summary structure, which is also a *proto-vEB*(4). Each *proto-vEB*(4) structure points to two *proto-vEB*(2) structures in *cluster*[0..1], and to a *proto-vEB*(2) summary. Each *proto-vEB*(2) structure contains just an array *A*[0..1] of two bits. The *proto-vEB*(2) structures above "elements i, j" store bits i and j of the actual dynamic set, and the *proto-vEB*(2) structures above "clusters i, j" store the summary bits for clusters i and j in the top-level *proto-vEB*(16) structure. For clarity, heavy shading indicates the top level of a proto-vEB structure that stores summary information for its parent structure; such a proto-vEB structure is otherwise identical to any other proto-vEB structure with the same universe size.

§3. Amortization

Definition 3.1 (Amortization).

- 摊还分析 (Amortized Analysis) 的基本思想是：分析一系列操作的总成本，并将这个总成本平摊（平均）到每一个操作上。
- 它关注的不是单个操作的"最坏情况"耗时，而是保证即使某个操作（例如每隔一段时间发生的重置/扩容操作）成本非常高，在考虑了所有操作的成本后，平均到每个操作上的成本仍然很低。

Example (Table Doubling).
- Like the rehash operations in hashmap.

为什么要双倍扩容？因为这样可以保证在单次扩容是 $O(N)$ 的，但是均摊的每一次操作就是 $O(1)$的。

§3.1. Aggregate Method

Definition 3.1.1 (Aggregate Method).

$$\text{Amortized Operation Cost} = \sum_{i=0}^{K} \frac{\text{Cost}(\text{operation}_i)}{K}$$

§3.2. Amortized Bound Definition

assign amortized cost for each operations, and we need to ensure it is a upperbound!

Recordings (Amortized Bound Definition).

$$\sum_{\text{amortized cost}} \geq \sum_{\text{actual cost}}$$

Recordings (To be done in the future.).
To be done in the future.

§4. Randomization and Randomized Algorithms

Recordings (Generating a random numbers).
- 计算机中生成随机数往往是根据随机数种子而生成伪随机数
- 在这里，认为这个操作不会随着数据的规模而不断变大，因此可以认为是 $O(1)$ 操作。

Definition 4.1 (Randomized Algorithms).
- Will generate a random number $r \in \{1, 2, 3, ..., R\}$ and make decisions based on r's value.
- Same input, different results.
- It will be classified into:
  - **Monte Carlo**: runs in polynomial time always output is correct with high probability
  - **Las Vegas**: runs in expected polynomial time output always correct

Recordings (Monte Carlo Tree Search).
Sometime, we just need to ensure the algorithms are correct **enough**, not absolutely correct.

§4.1. Matrix Product

§4.1.1. Matrix Product Checker

Simple Algorithms: $O(n^3)$.

Strassen Algorithms using divide and conquer optimization: $O(n^{\log_2 7}) \approx O(n^{2.81})$.

Coppersmith-Winograd: $O(n^{2.2376})$

But considering the constant factors, this optimization is not really useful.

Goal: Get an $O(n^2)$ algorithms:

- If $A \times B = C$, then the Pr[output = Yes] = 1.
- If $A \times B \neq C$, then the Pr[output = Yes] $\leq \frac{1}{2}$ (We get the upper-bound).

We can run this checker in many times! $O(kn^2)$ (For these are independent.)

§4.1.2. Frievald's Algorithm

Choose a random binary vector $r[1...n]$ which sampled from $\{0, 1\}$, such that $\Pr[r_i = 1] = 1/2$ independently for $r = 1, ..., n$. The algorithm will output 'YES' if A(Br) = Cr and 'NO' otherwise.

Recordings ().
- 使用 向量和矩阵的乘法
- The validation time complexity is $O(n^2)$ instead of $O(n^3)$.
- three times of matrix-vector multiplications.

§4.1.3. Analysis

- No False Negative Here.
  - We will ensure for the correct answer, the final result will respond to yes!
- We need to prove the upper-bound for the True-Negative.

Lemma 4.1.3.1.
We need to claim:

$$\text{If } AB \neq C, \text{then } \Pr[A(Br) \neq Cr] \geq \frac{1}{2}$$

Proof. Define $D = AB - C$.

$A(Br) = Cr \to Dr = 0$

Then, we need to prove:

$$\text{If } D \neq 0, \text{then } \Pr[Dr = 0] \leq \frac{1}{2}$$

For $D \neq 0$, we assume and consider there $\exists j$, $d_j \neq 0$.

Then, we just need to prove:

$$\Pr[d_1 r = 0] \leq \frac{1}{2}$$

for we have: $\Pr[Dr = 0] \leq \Pr[d_1 r = 0]$

Then, we can use the **Principle of Deferred Decisions** and prove it! □

Recordings (Principle of Deferred Decisions).
在分析一个涉及一系列随机选择的算法或过程时，不需要假设所有的随机选择都是在开始时一次性确定的。相反，可以想象这些随机选择是随着算法的执行一步一步、在需要用到它们的时候才进行决定的，而这种"延迟"并不会改变最终的概率分布或结果。

Analysis of Correctness if $AB \neq C$

Claim. If $AB \neq C$, then $Pr[ABr \neq Cr] \geq 1/2$.

Let $D = AB - C$. Our hypothesis is thus that $D \neq 0$. Clearly, there exists r such that $Dr \neq 0$. Our goal is to show that there are many r such that $Dr \neq 0$. Specifically, $Pr[Dr \neq 0] \geq 1/2$ for randomly chosen r.

$D = AB - C \neq 0 \implies \exists\ i, j$ s.t. $d_{ij} \neq 0$. Fix vector v which is 0 in all coordinates except for $v_j = 1$. $(Dv)_i = d_{ij} \neq 0$ implying $Dv \neq 0$. Take any r that can be chosen by our algorithm. We are looking at the case where $Dr = 0$. Let

$$r' = r + v$$

Since v is 0 everywhere except v_j, r' is the same as r exept $r'_j = (r_j + v_j) \mod 2$. Thus, $Dr' = D(r + v) = 0 + Dv \neq 0$. We see that there is a 1 to 1 correspondence between r and r', as if $r' = r + V = r'' + V$ then $r = r''$. This implies that

$$\text{number of } r' \text{ for which } Dr' \neq 0 \geq \text{number of } r \text{ for which } Dr = 0$$

From this we conclude that $Pr[Dr \neq 0] \geq 1/2$

Figure 10: Analysis of correctness if $AB \neq C$

Recordings (The basic principle).
从 $D \neq 0$ 的条件出发，确定关键的非零元素

- 从 $Dr = 0$，根据 r 构建一个新的向量 r'，形成一个映射，并且可以保证每一个 r 都可以找到一个 $r' = r + v$ s.t. $Dr' = D(r + v) = Dv \neq 0$.
- Then we can construct an injection which guarantee that $|\{Dr = 0\}| \leq |\{Dr \neq 0\}|$.
  - For the vector r is arbitrarily chosen, thus $P \geq \frac{1}{2}$.

§4.2. QuickSort

- In place sorting. (Compared with merge sort.)
- All the work is the divide step.
  - For the merge sort, all the work is all the merge step!

§4.3. Basic Quick Sort

For the worst case:

$$\begin{aligned} T(n) &= T(0) + T(n-1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

Thus, for the worst cases, its time complexity is $\Theta(n^2)$.

Recordings (The core is pivot).
- 和之前分治的基本分析情况保持一致。
- 关键在于子问题的切分需要保证足够均衡。

§4.4. Pivot Selection Using Median Finding

Using pivot selection optimization like Median Findings, we can select the pivot:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + \underbrace{\Theta(n)}_{\text{The time cost for median findings}} + \underbrace{\Theta(n)}_{\text{The time cost for dividing.}} \\ &= \Theta(n \log n) \end{aligned}$$

It has the better asymptotic complexity, the performance of it loses in practice. For finding the median will brings a relative large constant numbers.

§4.5. Randomized QuickSort

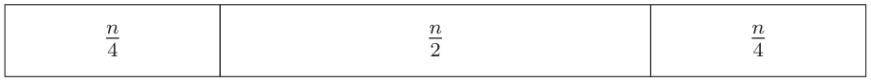
The core selection is **select pivot intelligently**!

$$\frac{n}{4} \le |L| \le \frac{3n}{4}$$
$$\frac{n}{4} \le |G| \le \frac{3n}{4}$$

"Paranoid" Quicksort Analysis

Let's define a "good pivot" and a "bad pivot"-
Good pivot: sizes of L and $G \le \frac{3}{4}n$ each
Bad pivot: one of L and G is $\le \frac{3}{4}n$ each

bad pivots good pivots bad pivots



We see that a pivot is good with probability $> 1/2$.
Let $T(n)$ be an upper bound on the expected running time on any array of n size.
T(n) comprises:

- time needed to sort left sub-array.
- time needed to sort right sub-array.
- the number of iterations to get a good call. Denote as cn the cost of the

partition step.

We can use the expectations:

$$T(n) \leq \max_{\frac{n}{4} \leq i \leq \frac{3n}{4}} (T(i), T(n-i)) + \mathbb{E}(\text{iterations}) \cdot cn$$

For the probability of the good pivot $p \geq \frac{1}{2}$, thus the expectations is:

$$\mathbb{E}(\text{iterations}) \leq 2$$

Thus:

$$T(n) \leq \max_{\frac{n}{4} \leq i \leq \frac{3n}{4}} (T(i), T(n-i)) + 2cn$$

Drawing the recursion tree, we can get the height of the tree is at most $\log_{\frac{4}{3}}(2cn)$ and at each level, we do the total of $2cn$ work. Thus the total time complexity is:

$$T(n) = \Theta(n \log n)$$

§4.6. SkipList

Recordings (SkipList).
To be done in the future.

§4.7. Hashing

§4.7.1. ADT and Dictionary

For abstract data type **dictionary**, this data structure supports several operations:

- `insert(item)`
- `delete(item)`
- `search(key)`

By implementing this ADT, we can use:

- We can use the `AVL Tree` or `RB Tree` to implement the data structure from scratch, which is called `set` in C++ STL. Several operations achieve $O(\log n)$ time.
- We can use hash-table to implement the data structure for unsorted conditions. Time complexity is average $O(1)$ but worst $O(n)$.

§4.7.2. Simple Hashing

Definition 4.7.2.1 (Several Definitions).
- u = number of keys over all possible items
- n = number of keys/items currently in the table
- m = number of slots in the table

具体的数据结构：Bucket and chaining.

§4.7.3. Simple Uniform Hashing

$$\Pr()_{k_1 \neq k_2}\{h(k_1) = h(k_2)\} = \frac{1}{m}$$

We achieve time complexity for $O(1+\alpha)$, $\alpha = \frac{n}{m}$ which is called load factor.

Recordings (Simple Uniform Hashing).
- 使用 Linked HashList 实现的哈希表可以达到接近常数的均摊时间复杂度，但是这是在最理想不产生哈希冲突的情况下才会出现的。
- 因此保证简单一致哈希需要保证数据分布必须是均匀并且随机的，或者所选择的哈希函数可以保证均匀的处理数据分布到不同的桶中。
- 因此，我们需要 Universal Hashing ，这个技术可以保证不需要雨来键的输入的随机性而保证非常快速的平均性能。

§4.7.4. Universal Hashing

- choose a random hash function h from $\mathcal{H}$
- require $\mathcal{H}$ to be a **universal hash family** such that:

$$\Pr_{h \in \mathcal{H}}\{h(k) = h(k')\} \leq \frac{1}{m} \ \forall k \neq k'$$

Theorem 4.7.4.1.
For n arbitrary distinct keys and random $h \in \mathcal{H}$, where $\mathcal{H}$ is a universal hashing family:

$$\mathbb{E}[\text{numbers of keys colliding in a slot}] \leq 1 + \frac{n}{m} = 1 + \alpha$$

Proof.

$$\mathbb{E}[\text{keys hashing to the same slot as } k_i] = \mathbb{E}\left[\sum_{j=1}^{n} I_{i,j}\right]$$

Then, due to the linearity of expectations and the independence, we can have:

$$\mathbb{E}\left[\sum_{j=1}^{n} I_{i,j}\right] = \sum_{j=1}^{n} \mathbb{E}[I_{i,j}] = \sum_{j \neq i} \mathbb{E}[I_{i,j}] + \mathbb{E}[I_{i,i}] \leq \frac{n}{m} + 1$$

□

Thus, we need to define the hash family $\mathcal{H} = \{\text{all hash functions } h : \{0, 1, ..., u-1\} \to \{0, 1, ..., m-1\}\}$.

We want to store and select the hash function in an efficient way.

§4.7.4.1. Dot Product Hash Family

We assume m is an integer and $u = m^r$.

We view the key in the hash function $k = (k_0, k_1, ..., k_{r-1})$ and each hash function in the hash family is defined by a randomized parameter vector $a = (a_0, a_1, ..., a_{r-1})$.

$$h_a(k) = \left(\sum_{i=0}^{r-1} a_i k_i\right) (\text{mod } m)$$

Thus, the hash family is:

$$\mathcal{H} = \{h_a | a \in \{0, 1, ..., u-1\}\}$$

In the word RAM model, manipulating $O(1)$ machine words takes O(1) time and "objects of interest" (here, keys) fit into a machine word. Thus computing ha(k) takes O(1) time.

简单来说，我们只需要保证 r 是 $O(1)$ 的。在具体的操作过程中，对于超大整数的切分也是根据字长来实现的，这样能保证单个字节的运算是 $O(1)$ 的操作。

Theorem: Dot-product hash family $\mathcal{H}$ is universal.

Proof: Take any two keys $k \neq k'$. They must differ in some digits. Say $k_d \neq k_{d'}$. Define *not* $d = \{0, 1, \ldots, r-1\} \setminus \{d\}$. Now we have

$$\begin{aligned}
\Pr_a\{h_a(k) = h_a(k')\} &= \Pr_a\left\{\sum_{i=0}^{r-1} a_i k_i = \sum_{i=0}^{r-1} a_i k'_i \ (\text{mod } m)\right\} \\
&= \Pr_a\left\{\sum_{i\neq d} a_i k_i + a_d k_d = \sum_{i\neq d} a_i k'_i + a_d k'_d \ (\text{mod } m)\right\} \\
&= \Pr_a\left\{\sum_{i\neq d} a_i (k_i - k'_i) + a_d (k_d - k'_d) = 0 \ (\text{mod } m)\right\} \\
&= \Pr_a\left\{a_d = -(k_d - k'_d)^{-1} \sum_{i\neq d} a_i (k_i - k'_i) \ (\text{mod } m)\right\} \qquad (4) \\
&(m \text{ is prime} \Rightarrow \mathbb{Z}_m \text{ has multiplicative inverses}) \\
&= \underset{a_{not\ d}}{E}[\Pr_{a_d}\{a_d = f(k, k', a_{not\ d})\}] \\
&(= \sum_x \Pr\{a_{not\ d} = x\} \Pr_{a_d}\{a_d = f(k, k', x)\}) \\
&= \underset{a_{not\ d}}{E}\left[\frac{1}{m}\right] \\
&= \frac{1}{m}
\end{aligned}$$

□

Recordings (Another universal hash family).

$$h_{ab}(k) = [(ak + b) \bmod p] \bmod m$$

And let $\mathcal{H}$ to be: $\{h_{ab} | a, b \in \{0, 1, ..., u-1\}\}$

§4.7.5. Perfect Hashing

In the worst case of universal hashing, it has little probability for hashing collusion. (For every time we build a hash table or rehash a hash table, we will choose a hash function from the hash family randomly.) But in the worse case, the time complexity is still $O(n)$.

For dynamic dictionary problem, we cannot achieve perfect hashing. But we can do this for static ones!

Static dictionary problem: Given n keys to store in table, only need to support `search(k)`. No insertion or deletion will happen. We can achieve searching with $O(1)$ time complexity!

§5. Advanced Dynamic Programming

§6. Greedy Algorithms

§7. Graph Algorithms

§8. Linear Programming

§9. Complexity

§10. More Advanced Algorithms

> **Recordings** (More advanced algorithms)**.**
> You know, algorithms are fascinating...

§11. Conclusion