Algorithms

Xiyuan Yang

2025.11.03

SJTU Semester 2.1 Algorithms: Design and Analysis

目录

§ 1. Lec1 Introduction

§ 1.1. Problems and Computation

The basis of **AI**:
- Search
- Learning

Definition 1.1.1 计算问题.
- Given a **input**, $I, x \in I$

- output, O, $y \in O$
- relation: $f : x \to y$: use the algorithm!
  - we have some boundaries: (s.t.)

Definition 1.1.2 Problem Domain.
The set of all the problems. $< I, O, f >$

Definition 1.1.3 Problem Instance.
one simple case in the problem domain $< I, O, f, x >$

What is algorithm:
- a piece of code
- handling the mapping from x to y

§ 1.2. Algorithm

§ 1.2.1. Definition

Definition 1.2.1.1 Algorithm.
- Fixed length code
- accept input with any length (or we say it can scale up!)
- at finite time terminate

- Natural Language
- pseudocode
- Written Codes

For example, birthday matching problems.

§ 1.2.2. Pseudocode

- if else end
- foreach end
- init data structure (Use $\leftarrow$)

```
do something
do something else
while still something to do
    do even more
    if not done yet then
        wait a bit
        resume working
    else
        go home
```

```
    end
end
```

```
function BinarySearch(A, x)
low ← 0
high ← A.length - 1
while low <= high
    mid ← low + floor((high - low) / 2)
    if A[mid] == x then
        return mid
    else if A[mid] < x then
        low ← mid + 1
    else
        high ← mid - 1
    end
end
return −1
end
```

§ 1.3. RoadMap

Recordings RoadMaps.

- 基本概念，算法复杂度和正确性分析
- 分治法
- 排序算法
- 哈希表
- 贪心算法
- 动态规划
- 图搜索算法
- 回溯法
- 分支界限
- 启发式算法

§ 1.3.1. Divide and Conquer

- Like the merge sort and recursion.
- Split bigger problems into smaller ones.

§ 1.3.2. Greedy algorithms

- making the **locally optimal choice** at each stage with the hope of finding a global optimum.

§ 1.3.3. Dynamic Programming

- 最优子结构 optimal sub-structure
  - 这也是和 divide and conquer 算法之间最显著的区别

- 重叠的子问题 overlapping sub-problems
  - 这保证了动态规划的重复利用的部分，也是动态规划的高效性所在（不再重复计算）

§ 1.3.4. Back Tracking

- a brute-force searching algorithms with pruning.
- Like the DFS algorithm
  - N Queens Problems

§ 1.3.5. Heuristic Algorithms

- when encountering large solve space
- optimize (or tradeoff) for traditional searching algorithms.
- great for NP-hard problems.

§ 1.4. Correctness of the algorithms

给定输入-输出组 (x, y)，给出一个 judger function，返回一个布尔值是否正确。

Recordings Judger Functions.
- 一般而言，算法求解的复杂度是更被关注的部分，算法求解的复杂度会高于算法验证正确性的复杂度
- 但是 Evaluation is also important!

§ 1.5. 算法正确性证明

§ 1.5.1. 数学归纳法

归纳法将问题的结构简化为了两个部分的证明：

Definition 1.5.1.1 数学归纳法.
- 基础情况的证明成立
- 递推关系的证明成立
  - 在递推关系中，存在“假设”，相当于多添加了一个前提条件。

Example Birthday Example.

4.2 生日匹配算法正确性证明

证明：生日匹配算法的正确性

归纳基础：$k=0$，记录中前k 个学生不包含匹配，算法正确报告不匹配。

归纳假设：对于$k=k_0$ 个学生，如果前k_0 个包含匹配，算法在访问第k_0+1 个学生之前返回匹配。

归纳步骤：考虑$k=k_0+1$ 的情况

- 如果前k_0 个包含匹配，根据归纳假设，算法已经返回匹配
- 否则前k_0 个没有匹配，所以如果前k_0+1 个有匹配，匹配必须包含第k_0+1 个学生
- 然后算法直接检查第k_0+1 个学生的生日是否存在于前k_0 个学生中

图 1 Demo for the correctness of algorithms for birthday

§ 1.6. Complexity & Efficiency

时间复杂度的衡量为了摆脱硬件性能的约束和影响，在衡量算法复杂度的时候，往往使用原子操作来代表基本的时间步：

- Number of atomic operations.
- 常数开销 $O(1)$ 的操作：例如加减乘除
  - $O(1)$ 生万物

§ 1.7. Asymptotic Notation

Definition 1.7.1 Asymptotic Notation.
- O: Upper Bound
- Ω: Lower Bound
- Θ: 紧界
  - $f(n)=\Theta(g(n))$ 表示 $f(n)$ 和 $g(n)$ 的增长速度相同。

- Polynomial Complexity: $O(n^k)$
- Exponential Complexity: $O(k^n)$
  - X-hard problems

§ 1.8. Model of Computation

上述 $O(1)$ 生万物 的计算模型基于 WordRAM 计算模型：
- 整数运算
  - 浮点数？理论上不是，但是基本上是。
- 逻辑运算
- 位运算
- 内存访问（给定地址的特定内存块的读取和写入）

上述的运算都为 $O(1)$ 的时间复杂度。

§ 1.9. 系统字节数

32 位系统和 64 位系统标定的是内存地址的长度

- 32 位系统：4GB
- 64 位系统：16EB
  - 保证给 16 EB 的内存寻址，在 $O(1)$ 的时间复杂度进行存址

§ 1.10. 算法时间复杂度证明

记：

$$H_n = \sum_{i=1}^{n} \frac{1}{i}$$

求证：

$$H(N) = \Theta(\log N)$$

使用积分不等式：

$$\int_0^1 \frac{1}{x} \mathrm{d}x \leq \sum_{i=1}^{n} \frac{1}{i} \leq \int_0^1 \frac{1}{x} \mathrm{d}x + 1$$

§ 2. Simple Data Structure

Example 翻转单项链表.
给定一个长度为 N 的链表，在 $O(1)$ 空间复杂度和 $O(N)$ 的时间复杂度实现单项链表的翻转

Solution: 使用三指针实现

Recordings 三指针实现.
- 从最基本的情况开始，如果只有三个节点，很显然，只需要修改两条链表边的指向就可以
- 在跟一般的情况，认为第三个节点的后继结点被占满了，这只需要存储一个 variable 保证链表不断裂就可以

§ 3. Linear Time Sorting

Analysis and more advanced sorting algorithm with more constraints.

§ 3.1. Basic Analysis for Sorting

§ 3.1.1. The UpperBound Analysis

Consider the decision tree which contains all the output of all the possible inputs. (Based on selections.)

Formally, you are given an array with random values. (The candidate of the values are fixed but the sequence are random generated.) $[a_1, a_2, a_3, \ldots, a_{n-1}]$. At the leaf node of the decision binary tree, we know:

$$\text{Count}(\text{Leaf Node}) = n!$$

Thus, the optimal time complexity for general sorting algorithm (based on selections) are $O(\log(n!)) = O(n \log n)$ (Suppose the tree are balanced, like the process of merge sort.)

However, when we have a huge amount of data to be sorted, the time complexity $O(n \log n)$ is unaffordable, thus we can sacrifice generalization for speeding up.

§ 3.2. Direct Access Array sorting

We need to do following assumptions:

- We have the upperbound u: Suppose all keys are unique non-negative integers in range $\{0, ..., u - 1\}$, thus $u \geq n$.

Then, we can use the bucket, so called **direct access array**!

- Insert each item in the direct access array with size u in $\Theta(n)$.
- Return and sort the array with time $\Theta(u)$.

Thus, the total time complexity is $\Theta(n + u)$.

§ 3.3. Radix Sort or Tuple Sort

For case that $u = n^2$, the time complexity is $O(n^2)$.

We can split the number into a **tuple**, where $x = an + b$.

Then, we can do direct access array in 2 turns, in each turn, we ensure $u_0 - n$, ensuring the linear time complexity.

More generally, if we do c turns, then the time complexity is $O(cn)$, but the upperbound of u can up to n^c. Thus, the time complexity can be optimized into $O(n + n \log_n u)$

§ 3.4. Counting Sort

Define the maximum value and minimum value of the array $a[i]$, we can use linear time to scan them. The, we can define the maximum gap $d \leq u$, and we create an array with size d.

Then, we count all the numbers in the array and accumulate:

$$C[i] = C[i] + C[i - 1]$$

Finally, if $x = C[i]$, it means the last final positions for i should be x. **It can ensure the stability for the sorting.**

§ 4. Conclusion