算法设计与分析

Lecture 3: Algorithm Analysis

曹刘娟

厦门大学信息学院计算机系

caoliujuan@xmu.edu.cn

Review（复习）

什么是算法?

- An algorithm is a sequence of computational steps that transform the input into the output to solve a given problem.
- Example: The sorting （排序问题）problem.
  - Input: A sequence of n numbers $A = \langle a_1, a_2, \dots, a_n \rangle$.
  - Output: A permutation (reordering) $A' = \langle a_1', a_2', \dots, a_n' \rangle$ of the input sequence such that $a_1' \le a_2' \le \dots \le a_n'$.

LOOP INVARIANTS （循环不变量）

- At the start of each iteration of the for loop, the subarray $A[1 \ldots j-1]$ consists of the elements originally in $A[1 \ldots j-1]$ but in sorted order.
- We state these properties of $A[1 \ldots j-1]$ formally as a loop invariants (循环不变量).
- 使用loop invariants 来证明算法的正确性 why an algorithm is correct.
- 很多时候采用数学归纳法

插入排序算法的正确性

The loop invariants are:

$A[1 \ldots j-1]$ is sorted before each iteration.

The proof is similar to mathematical induction (数学归纳法):

- Initiation 初始化 (归纳基础): It is true prior to the first iteration of the loop.
- Maintenance (归纳步骤): If it is true before an iteration of the loop, it remains true before the next iteration.
- Termination(终止): When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

EFFICIENCY OF ALGORITHM

- Is correctness of an algorithm enough? How good is an algorithm?
- Time complexity (时间复杂度)
  - Indicates how fast an algorithm runs.
  - How many CPU cycles needed.
- Space complexity (空间复杂度)
  - Amount of memory units required by an algorithm.
- Does there exist a better algorithm?
- How to compare algorithms?

BIG O NOTATION 大O

Definition 2.2

For a given complexity function $g(n)$, $O\big(g(n)\big)$ is the set of complexity functions $f(n)$ for which there exists some positive real constant c and some nonnegative integer n_0 such that for all $n \geq n_0$,

$$0 \leq f(n) \leq cg(n).$$

- *上面的定义要注意，是存在正整数c和n_0使得对所有的n ≥n0都成立*
- $O\big(g(n)\big)$ is a set of functions in terms of $g(n)$ that satisfy the definition.
- If $f(n) = O\big(g(n)\big)$, it represents that $f(n)$ is an element in $O\big(g(n)\big)$. We say that $f(n)$ is "big O (大O)" of $g(n)$.
  - Strictly, we should use "∈" instead of "=". However, it is conventional to use "=" for asymptotic notations.

BIG O NOTATION

- 我们关心的是当算法规模n非常大的情况
- No matter how large $f(n)$ is, it will eventually be smaller than $cg(n)$ for some c and some n_0.
- Big O notation describes an upper bound （上界）. We use it to bound the worst-case running time（最差运行时间） of an algorithm on arbitrary inputs.

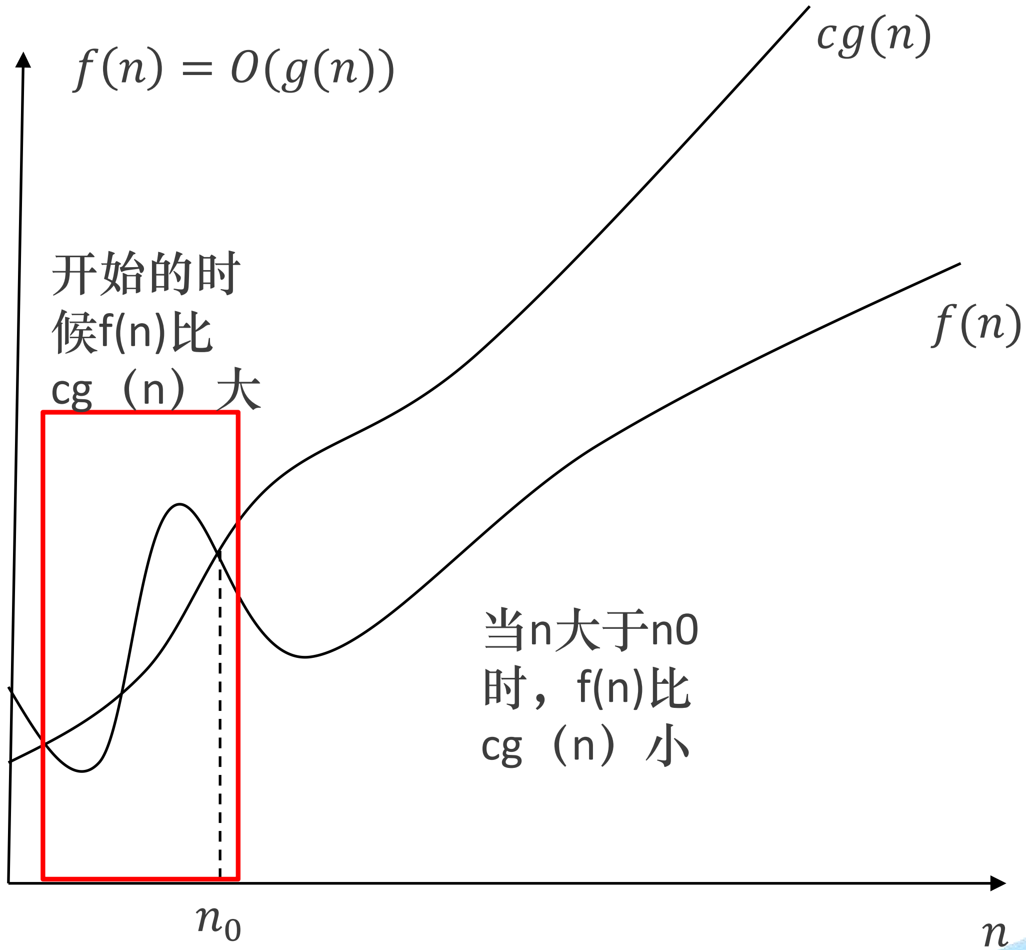


Image source: 图2.2, 张德富, 算法设计与分析, 国防工业出版社, 2009.

Definition 2.3

For a given complexity function $g(n)$, $\Omega\big(g(n)\big)$ is the set of complexity functions $f(n)$ for which there exists some positive real constant c and some nonnegative integer n_0 such that for all $n \geq n_0$,

$$0 \leq cg(n) \leq f(n).$$

- $\Omega\big(g(n)\big)$ is the opposite of $O\big(g(n)\big)$.
- If $f(n) = \Omega\big(g(n)\big)$, it represents that $f(n)$ is an element in $\Omega\big(g(n)\big)$. We say that $f(n)$ is “big Ω (大Ω)” of $g(n)$.

BIG Θ NOTATION (发音THETA)

Definition 2.1

For a given complexity function $g(n)$, $\Theta\big(g(n)\big)$ is the set of complexity functions $f(n)$ for which there exists some positive real constants c_1 and c_2 and some nonnegative integer n_0 such that, for all $n \geq n_0$,

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n).$$

- If $f(n) = \Theta\big(g(n)\big)$, we say that $f(n)$ is "big Θ (大Θ)" or has the same order (数量级) of $g(n)$.
- $\Theta\big(g(n)\big) = O\big(g(n)\big) \cap \Omega\big(g(n)\big)$.
- Θ 代表既是上界、又是下界

- Transitivity (传递性)
  - If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ then $f(n) = \Theta(h(n))$.
  - Same for O and Ω.
- Additivity (可加性)
  - If $f(n) = \Theta(h(n))$ and $g(n) = \Theta(h(n))$ then $f(n) + g(n) = \Theta(h(n))$.
  - Same for O and Ω.
- Reflexivity (自反性)
  - $f(n) = \Theta(f(n))$.
  - Same for O and Ω.
- Symmetry (对称性)
  - $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.
  - Not hold for O and Ω.

- In addition to proving by definition, we can also use limit to get asymptotic notations.

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}\begin{cases} = c & \text{implies } f(n)=\Theta\big(g(n)\big) \quad \text{if } 0<c<\infty \\ =0 & \text{implies } f(n)=O\big(g(n)\big) \\ =\infty & \text{implies } f(n)=\Omega\big(g(n)\big) \end{cases}$$

可以用洛必达法则

PROBABILISTIC ANALYSIS （概率分析）

Probabilistic Analysis

概率知识 分析 算法平均时间复杂度

- Average-case analysis determines the average (or expected) performance 概率分析分析的是算法的平均性能
  - The average time over all inputs of size n.
- The average-case analysis needs to know the probabilities of all input occurrences, i.e., it requires prior knowledge of the input distribution.所有输入发生的概率，即输入分布的先验知识
- Usually, to ease the analysis, we can use probabilistic analysis by simply assuming that all inputs of a given size appear with equal probability, i.e. draw from a uniform distribution. 一般是假设所有的输入用例的概率是一样的

Linear Search （线性搜索找特定值）

- The searching problem:
  Search an array A of size n to determine whether the array contains the value x; return index if found, 0 if not found.
- Recall the strategy 1 of the phonebook example in Lecture 1. We check the name from the top one by one. This algorithm is called linear search for the searching problem.

```
LinearSearch(A, x)
k ← 1
while k ≤ n and x ≠ A[k] do
        k ← k + 1
if k > n then return 0
else return k
```

Probabilistic Analysis of Linear Search

- To simplify the analysis, let us assume:
  - $A[1 \ldots n]$ contains the numbers 1 through n, which implies that all elements of A are distinct.(两两互不相同)
  - The search key x is in A.
  - The search key x is uniformly drawn from $[1 \ldots n]$.（x在任一位置出现机会相同）
  - We only count the number of key comparisons.（这里只算比较的次数）

LinearSearch(A, x)

$k \leftarrow 1$
while $k \leq n$ and $x \neq A[k]$ **do**
    $k \leftarrow k + 1$
if $k > n$ **then return** 0
else return k

Probabilistic Analysis of Linear Search

- Probability of x being found at index k is $1/n$ for each value of k.

假设x在第k个位置被发现的概率为1/n

- If $x = A[k]$, then the number of comparison is k. 第k个位置比较k次
- Therefore, we can calculate the expected number of comparison by multiplying k with its probability $1/n$ and then sum them up.
- So the number of comparison on the average is:

$$T(n) = \sum_{k=1}^{n} \frac{1}{n} \cdot k = \frac{1}{n} \sum_{k=1}^{n} k = \frac{1}{n} \frac{n(n+1)}{2} = \frac{n+1}{2}.$$

- Hence, the average-case time complexity of $\mathrm{LinearSearch}(A, x)$ is $\Theta(n)$.（平均时间复杂度）
- Think: What if the key x is not uniform distributed?

Probabilistic Analysis of Insertion Sort (插入排序)

- To simplify the analysis, let us assume:
  - $A[1..n]$ contains the numbers 1 through n, which implies that all elements of A are distinct 数组里面所有元素都不重复.
  - All $n!$ permutations of A appear with equal probability as the input. 所有可能的排列输入概率是一样的
  - We only count the number of key comparisons. 只考虑有多少次的和key进行比较

InsertSort(A)

for $j \leftarrow 2$ **to** n **do**
    $key \leftarrow A[j]$
    $i \leftarrow j - 1$
    while $i > 0$ and $A[i] > key$ **do**
        $A[i+1] \leftarrow A[i]$
        $i \leftarrow i - 1$
    $A[i+1] \leftarrow key$
return A

Probabilistic Analysis of Insertion Sort

- For different input, the difference of running time is from t_j, namely, how many comparisons do we need before inserting the key. 插入前计算比较次数
- Now we consider inserting $key = A[j]$ in the proper position in $A[1...j]$.
- If its proper position is $k (1 \leq k \leq j)$, then the number of comparisons performed in order to insert key in $A[k]$ is:

$$\begin{cases} j-1, & if\ k=1 \\ j-k+1, & if\ 2 \leq k \leq j \end{cases}.$$

  - If $k = 1$, the condition in while loop $i > 0$ is false and the comparison $A[i] > key$ is not triggered.
  - If $2 \leq k \leq j$, one more comparison $A[i] > key$ is needed.

Probabilistic Analysis of Insertion Sort

- Since the probability that its proper positions in $A[1...j]$ is $1/j$, so the number of comparisons needed to insert $A[j]$ in its proper position in $A[1...j]$ is:

$$\frac{1}{j} \cdot (j-1) + \frac{1}{j}\sum_{k=2}^{j}(j-k+1) = \frac{1}{j}(j-1+\sum_{k=1}^{j-1}k) = \frac{j}{2} - \frac{1}{j} + \frac{1}{2}.$$

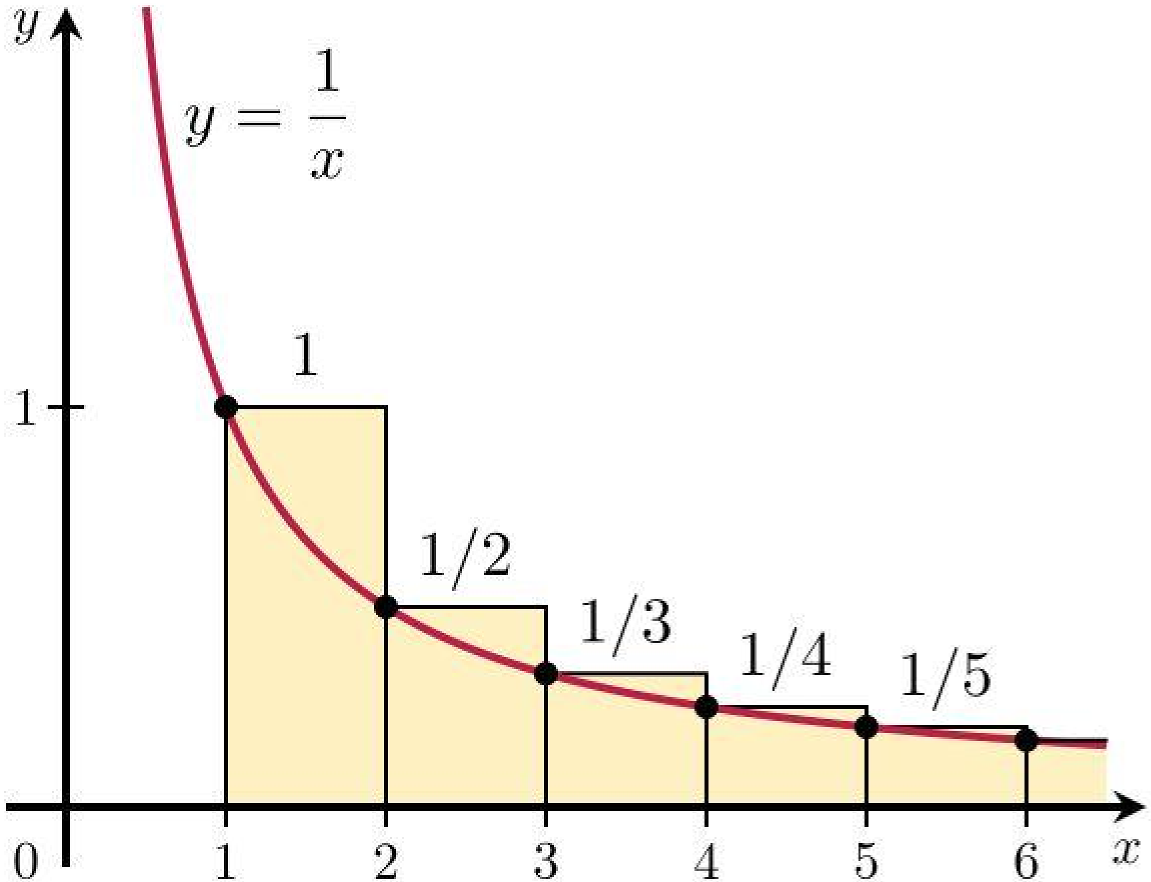
- Hence the average number of comparisons performed by InsertSort(A) is: 迭代n-1次

$$\sum_{j=2}^{n}\left(\frac{j}{2} - \frac{1}{j} + \frac{1}{2}\right) = \frac{n(n+1)}{4} - \frac{1}{2} - \sum_{j=2}^{n}\frac{1}{j} + \frac{n-1}{2}$$

$$= \frac{n^2}{4} + \frac{3n}{4} - \sum_{j=2}^{n}\frac{1}{j} = \Theta(n^2).$$

What is the order of this term?

调和级数



S = 1 + 1/2 + 1/3 + ... + 1/n ≈ ln(n) + γ

其中，S表示调和级数的和，n表示项数，ln(n)表示自然对数，γ表示欧拉常数。这个公式的推导涉及到数学分析的知识，可以使用积分的方法来证明。

The Hiring Problem 雇佣问题

- The problem scenario:
  - You are using an employment agency to hire a programmer.
  - The agency sends you one candidate each day.
  - You interview the candidate and must immediately hire the new one and fire the current one, if the new candidate is better.
  - **Cost of interview is C_i** and **cost of hiring is C_h**.
- If we hire m of n candidates finally, the cost will be $O(nC_i + mC_h)$.
- However, m varies with each run.
  - It depends on the order in which we interview the candidates.（取决于面试顺序）

The Hiring Problem

秘书从0开始，原本没有秘书，best表示当前最好秘书

HireProgrammer(n)
$best \leftarrow 0$
for $i \leftarrow 1$ to n **do**
  interview candidate i
  if candidate i is better than candidate *best* **then**
    $best \leftarrow i$
    hire candidate i.

$\boldsymbol{O}(\boldsymbol{n}\boldsymbol{C}_{\boldsymbol{i}} + \boldsymbol{m}\boldsymbol{C}_{\boldsymbol{h}})$.

Analysis of the Hiring Problem

- Best case
  - We just hire one candidate only.
    - The first is the best. Good luck thanks god.
  - Cost: $\boldsymbol{\Omega(nC_i + C_h)}$.
- Worst case
  - We hire all n candidates.
    - Each candidate is better than the current hired one. What a tough life!
  - Cost: $\boldsymbol{O(nC_i + nC_h)}$.
- What is the average case?

求当前求职者被聘用的概率，来估计平均聘用费用

- In general, we have no control over the order in which candidates appear.
- We just assume that they come in a **random order**.
  - The interview **score list $\boldsymbol{S}$** is equivalent to a permutation of the candidate numbers $\langle 1,2,3,\ldots,n\rangle$.
  - S is equally likely to be any one of the $n!$ permutations. Each of the possible **$\boldsymbol{n}$! permutations** appears with **equal probability**.

Probabilistic Analysis of the Hiring Problem

- Candidate i is hired if and only if candidate i is better than each of candidates $1,\ 2,\ .\ .\ .\ ,\ i-1.$
- Base on the assumption that the candidates arrive in random order, any one of these i candidates is equally likely to be the best one so far.
- Thus, the probability of hiring candidate i is $\mathbf{1/\boldsymbol{i}}$. The average cost of hiring is:

$$\sum_{i=1}^{n} \frac{1}{i} \cdot C_h = C_h \sum_{i=1}^{n} \frac{1}{i} \underset{???}{=} O(C_h \lg n).$$

- Thus, the averaged-case hiring cost is $O(\lg n)$, which is much better than the worst-case cost of $O(n)$.

Probabilistic Analysis of the Hiring Problem

- $\sum_{i=1}^{n} \frac{1}{i}$ is called the nth harmonic number (调和数).
- It has a bound of $O(\lg n)$.

1	2	3	4	5	6	7	8	9	10

$k = 0$ $\quad$ $k = 1$ $\quad$ $k = 2$ $\quad$ $k = 3$

$$\sum_{i=1}^{n} \frac{1}{i} \leq \sum_{k=1}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^k - 1} \frac{1}{2^k + j}$$

$$\leq \sum_{k=1}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^k - 1} \frac{1}{2^k}$$

$$= \sum_{k=1}^{\lfloor \lg n \rfloor} 1$$

$$\leq \lg n + 1 .$$

Examples of Probabilistic Analysis

Example 1: the Hat-Check Problem

- Each of n customers gives a hat to a hat-check person at a restaurant.
- The hat-check person gives the hats back to the customers in a random order.
- What is the expected number of customers that get back their own hat?

Example 1 (cont'd)

- Because there are n hats and the ordering of hats is random, each customer has a probability of $1/n$ of getting back his or her own hat.
- Now we can compute the expected number of all customers:

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

Examples of Probabilistic Analysis

Solution: Letting X denote the number of men that select their own hats, we can best compute $E[X]$ by noting that

$$X = X_1 + X_2 + \cdots + X_N$$

where

$$X_i = \begin{cases} 1, & \text{if the } i\text{th man selects his own hat} \\ 0, & \text{otherwise} \end{cases}$$

Now, because the ith man is equally likely to select any of the N hats, it follows that

$$P\{X_i = 1\} = P\{i\text{th man selects his own hat}\} = \frac{1}{N}$$

and so

$$E[X_i] = 1P\{X_i = 1\} + 0P\{X_i = 0\} = \frac{1}{N}$$

Hence, from Equation (2.11) we obtain

$$E[X] = E[X_1] + \cdots + E[X_N] = \left(\frac{1}{N}\right)N = 1$$

Hence, no matter how many people are at the party, on the average exactly one of the men will select his own hat.

结果很震惊！无论有多少人扔出自己的帽子，平均来看，总有1人能捡回自己的帽子！

Examples of Probabilistic Analysis

Example 2

- Assume that 12 passengers enter an elevator at the basement and independently choose to exit randomly at one of the 10 above-ground floors.
- What is the expected number of stops that the elevator will have to make?

Example 2 (cont'd)

- Denote the event that the elevator stops at the ith level as H_i.
- $\Pr\{H_i\} = 1 - \Pr\{\overline{H_i}\} = 1 - (1 - 1/10)^{12} = 1 - (9/10)^{12}$.
  - $\overline{H_i}$: the elevator does not stop (no passenger exit) at the ith level.
- Now we can compute expected number of stops:

$$\sum_{i=1}^{10} (1 - 0.9^{12}) = 10(1 - 0.9^{12}) \approx 7.176.$$

Classroom Exercise （课堂练习）

- Let $A[1 \ldots n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair $(i,\ j)$ is called an inversion of A.
- Suppose that each element of A is generated by randomly permutation. What is the expected number of inversions.

Classroom Exercise

Solution:

- Denote the event $i < j$ and $A[i] > A[j]$ as H_{ij}.
- Given two distinct random numbers, the probability that the first is bigger than the second is $1/2$. We have $\Pr\{H_{ij}\} = 1/2$.
- Now we can compute expected number of inversions by sum over of the pairs in the array:

$$\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{1}{2} = \frac{n(n-1)}{2}\cdot\frac{1}{2} = \frac{n(n-1)}{4}.$$

AMORTIZED ANALYSIS （分摊分析）

合计方法
记账方法
势能方法

概述

- 概率分析
  - 通过假设算法的各种输入用例的概率，计算出算法的平均情况复杂度
- 分摊分析

通过研究一系列结构运算所需要的费用，来研究各个运算之间的关系
一系列运算总费用是小的----->其中一个运算的分摊费用也是小的

  - 合计分析：将多个操作一起考虑，考虑总体平均的最坏情况，而不是只考虑单个操作最坏情况
  - 记账分析：将存款赋予一些操作，当其他操作执行时消耗这些操作的存款
  - 势能分析：将存款赋予整体数据结构

Amortized Analysis

- In some algorithms, the average-case performance is difficult to be determined because each operation takes different time. 在一些算法里面，由于不同的指令会消耗不同的时间，采用概率分析来分析算法时间复杂度较为困难。
- We can perform a sequence of such operations and average over the total time of all the operations performed. This is called amortized analysis (分摊分析).
- Amortized analysis differs from average-case analysis in that probability is not involved.（不涉及复杂的概率分析）
- An amortized analysis guarantees the average performance of each operation in the worst case. （保证了最坏情形下的每个操作的平均性能）

Amortized Analysis

- The key idea of amortized analysis:

If each single is different, but the total is fixed, we count the total and then calculate the average.

如果每个操作是不一样的，但是这些操作的总体是固定的，那么我们就可以计算这些操作的总体，然后取平均

- Base on this idea, there are three methods:
  - Aggregate method (合计方法)
  - Accounting Method (记账方法)
  - Potential method (势能方法)

Aggregate Method （合计方法）

- In aggregate method (合计方法), we show that for all n, a sequence of n operations takes worst-case time $T(n)$ in total.

  计算的是：这个n个操作合计的最坏情况$T(n)$

- 如果单纯的计算某个操作的最坏情况，并假设这个算法里面所有操作都是和最差情况一样，可能会过高估计了算法的复杂度，用合计方法估计的更准确些。
- In the worst case, the average cost, or amortized cost, per operation is therefore $T(n)/n$. 每个操作的分摊代价
- Note that this amortized cost applies to each operation, even when there are several types of operations in the sequence.（分摊费用计算方法对不同类型的每一个操作均成立）

MultiPop Operation

合计方法 堆栈操作

- Consider stack operations on stack S:
  - $\mathrm{Push}(S, x)$ pushes object x onto stack S.
  - $\mathrm{Pop}(S)$ pops the top of stack S and returns the popped object.
- Since each of these operations runs in $O(1)$ time, let us consider the cost of each to be 1.
- The total cost of a sequence of n Push and Pop operations is therefore n, and the actual running time for n operations is therefore $\Theta(n)$.

MultiPop Operation （多Pop操作）

- Now we add a new stack operation MultiPop(S, k): remove the k top objects of stack S or pop the entire stack if it contains fewer than k objects.
- What is the running time of MultiPop(S, k) on a stack of s objects?
  - It varies for different S.这个操作在不同栈大小下，计算量是不一样的

```
MultiPop(S, k)
while StackEmpty(S) ≠ Ø and k ≠ 0 do
        Pop(S)
        k ← k − 1
```

top → 23
17
6
39
10
47

top → 10
47

MultiPop(S, 4) 计算量为4

MultiPop(S, 7) 计算量为2

Aggregate Method for MultiPop Operation

- Let us analyze a sequence of n Push, Pop, and MultiPop operations on an initially empty stack.

$$\underbrace{\text{Push}(S, 1), \text{Push}(S, 2), \text{Pop}(S), \text{Push}(S, 4), \text{MultiPop}(S, 2), \ldots}_{n}$$

- For a stack with at most n elements, the worst-case time of MultiPop is $O(n)$, and we may have $O(n)$ MultiPop operations . Hence a sequence of $\boldsymbol{n}$ **MultiPop operations** costs $O(n^2)$.
- This analysis is correct but the upper bound is too high. We have at most n elements to pop. How does $O(n^2)$ come?
  - This upper bound situation will never be happened, because it is impossible to pop n elements in MultiPop for n times.
  - 每次估计**MultiPop**都用最差情况分析的话，过高估计了整体算法的复杂度（最坏情形上界）

Aggregate Method for MultiPop Operation

合计方法分析

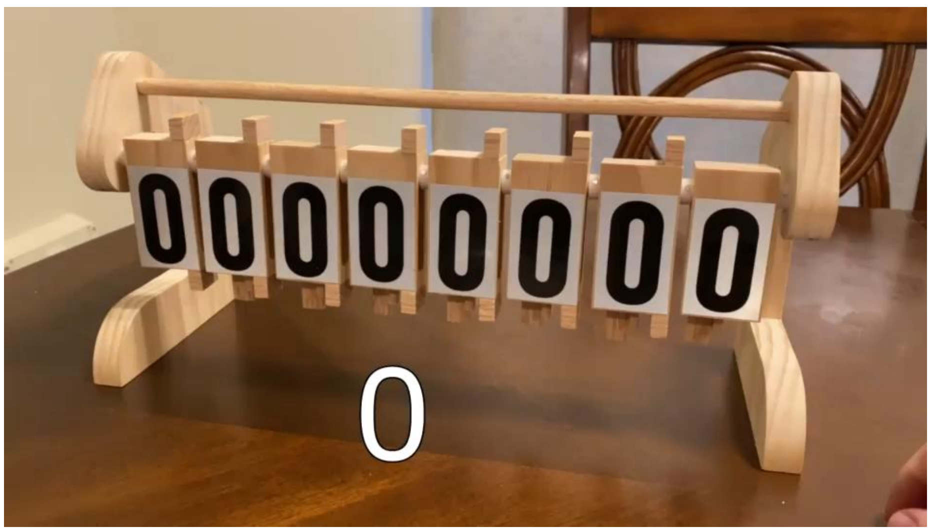
- Notice: each element is popped at most once after it is pushed into a stack. 当一个元素被 push到栈中时，这个元素最多被pop一次
- Therefore, the total number of Pop (include the ones in $\mathrm{MultiPop}$) operations is at most n. 因此在n个操作中，最多会有n次pop操作 （这里面包括MultiPop操作里面的pop）
- Therefore, any sequence of n Push, Pop, and $\mathrm{MultiPop}$ operations on an initially empty stack can cost at most $O(n)$. 因此合计起来这n个操作的cost是O(n)
- The average cost of **an operation** is $O(n)/n = O(1)$.
  - Although it looks like $O(n)$.

对比原来每个操作最坏情形 $O(n)$，明显降低

Binary Counter

合计方法 二进制加法

- Consider the problem of implementing a k-bit binary counter (k位二进制计数器) that counts upward from 0.
  - We use an array $A[0...k-1]$ of bits as the counter.
  - The lowest-order bit is in $A[0]$ and the highest-order bit is in $A[k-1]$.



A wooden 8-bit binary counter

Increment(A)

$i \leftarrow 0$

while $i < n$ and $A[i] = 1$ **do**

  $A[i] \leftarrow 0$

  $i \leftarrow i + 1$

if $i < n$ **then**

  $A[i] \leftarrow 1$

Video source: https://imgur.com/gallery/56LASVI

Binary Counter

Counter value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

红色字体代表如果下一次再加1，有几位要翻转（flip）的

Aggregate Method for Binary Counter计数器

- What is the average cost of a single execution of Increment, if we count the number of bits flipped as the cost?

  分析： 将有几位翻转了当做cost，每步操作的代价？

- Follow the idea of amortized analysis, we consider a sequence of n Increment operations on an initially zero counter. (分摊分析：将这n个加法操作一起考虑，初始化0)
- In the worst case, array A contains all 1. A single execution of Increment takes time $O(k)$. Thus, the whole sequence takes $O(nk)$.
- Will this worst case happen? （可进一步缩小上界）

Aggregate Method for Binary Counter

进一步分析:

- We can observe:
  - $A[0]$ is flipped for every execution.
  - $A[1]$ is flipped for every two executions, i.e. $A[1]$ is flipped $\lfloor n/2 \rfloor$ times for each execution.
  - $A[2]$ is flipped for every four executions, i.e. $A[2]$ is flipped $\lfloor n/4 \rfloor$ times for each execution.
  - …
  - $A[i]$ is flipped for every 2^i executions, i.e. $A[i]$ is flipped $\lfloor n/2^i \rfloor$ times for each execution.

初始为0的计数器，n次加运算，位$A[i]$翻转 $\lfloor n/2^i \rfloor$ 次，i=0,1,...$\lfloor lgn \rfloor$

Aggregate Method for Binary Counter

位$A[i]$翻转 $\lfloor n/2^i \rfloor$ 次，i=0,1,...$\lfloor lgn \rfloor$.执行n次，位反转总次数：

- Therefore, the total number of flips for n execution of Increment is:

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n.$$

- The worst-case time for a sequence of n Increment operations on an initially zero counter is therefore $O(n)$.
- The average cost of each operation, and therefore the amortized cost per operation, is $O(n)/n = O(1)$.

Accounting Method （记账方法）

- Accounting method (记账方法): Assign differing charges to different operations, with some operations charged more or less than they actually cost. The amount we charge an operation is called its amortized cost. 赋予不同的操作不同的费用，这些费用可能比实际的费用高或者低
- When an operation's amortized cost exceeds its actual cost, the difference is assigned to **specific objects** in the data structure as credit (存款). 当赋予更高的费用时，相当于将一些额外费用存储在这个数据里面
- Credit can be used later on to help pay for operations whose amortized cost is less than their actual cost. 额外存储的费用（存款）可以用于那些分摊费用比实际费用低的操作

对某一运算所赋予的费用，就记为该运算的分摊费用

Accounting Method （记账方法）

想要用分摊费用证明：最坏情形下，每个运算的平均费用是小的
序列总分摊费用必须为运算序列总实际费用的上界

- We denote:
  - c_i: the actual cost of the ith operation.
  - $\hat{c}_i$: the amortized cost of the ith operation.
- For the sequence of all n operations, we require:

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

- The total credit associated with the data structure must be nonnegative at all times.与数据结构相关联的**总存款**必须始终为非负 $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i$

记账方法 堆栈操作

Recall the stack operations. The actual costs of the operations are:

Push	1,
Pop	1,
MultiPop	$\min(k, s)$.

The amortized costs by accounting method are:（给分摊费用）

Push	2,
Pop	0,
MultiPop	0.

Accounting Method for MultiPop Operation

- Suppose we use a \$1 to represent each unit of cost. We start with an empty stack.
- When we push an element on the stack, we use \$1 to pay the actual cost of the push and are left with a credit of \$1 (out of the \$2 charged). （当push的时候，1用于actual cost, 另外一个1用于存款）
  - At any point in time, every element on the stack has \$1 of credit on it, which is for the cost of popping it. （这个1的存款是用于pop用的）
  - To pop (from Pop or MultiPop) an element, we take the dollar of credit off the element and use it to pay the actual cost of the operation.
  - Thus, by charging the Push operation a little bit more, we needn't charge the Pop operation anymore.
- Thus, for any sequence of n Push, Pop, and MultiPop operations, the total amortized cost is $O(n)$.

对于有n个操作的序列，总的分摊代价 $O(n)$，每个操作分摊 $O(1)$

Accounting Method for Binary Counter 二进制加法

记账方法 二进制加法

- Let us once again use \$1 to represent each unit of cost。（每个操作代价1）
- For the accounting method, let us charge an amortized cost of \$2 to set a bit to 1. 某比特位设为1的运算支付2元的分摊，0不分摊
  - When a bit is set to 1, we use \$1 to pay for the actual setting, and the other \$1 for preparing flipping the bit back to 0.（1用于实际费用，1用于转为0的费用）
  - The cost of setting the bits to 0 within the while loop is paid by the dollars on the bits when they are set to 1.（用存款支付）
  - Thus, the amortized cost for setting bits to 0 in the while loop becomes 0, and the amortized cost of setting bits to 1 in Line 6 of Increment is \$2.
- Thus, for n Increment operations, the total amortized cost is $\boldsymbol{O(n)}$, which bounds the total actual cost.

n个加操作的总分摊费用为$O(n)$也是实际费用的上界，每个操作。。

Potential Method （势能方法）

势能方法把每个运算的余款表示成势能，存储在整个数据结构中，它再需要时用来支付后面运算所需要的费用。

- In accounting method, we associate credits with elements in the data structure. 记账方法是把存款赋予给每个元素
- Similarly, in potential method (势能方法), we store "potential" of the data structure for future operations. 势能方法将存款赋予整个数据结构
  - We start with an initial data structure D_0 on which n operations are performed. 最开始将一个值赋给初始数据结构
  - Let D_i be the data structure that results after applying the ith operation to data structure D_{i-1}, for each $i = 1, 2, \ldots, n$. Di代表i操作之后，数据结构的势能值
  - A potential function Φ maps each data structure D_i to a real number $\Phi(D_i)$, which is the potential associated with data structure D_i. 势能函数是一个大于等于0的数, 把每个数据结构映射成一个实数。

Potential Method

- Let c_i be the actual cost of the ith operation.
- The amortized cost $\hat{c}_i$ of the ith operation with respect to potential function Φ is defined by （第i个运算的分摊费用）

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

- The total amortized cost of the n operations is

（n个运算的总分摊费用）

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right)$$

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0).$$

Potential Method

- Just like accounting method, we can pay for future operations by potential in potential method.像记账方法一样，我们可以预先支付
- If we can define a potential function Φ so that $\Phi(D_n) \geq \Phi(D_0)$, then the total amortized cost is an upper bound on the total actual cost.
  - It is often convenient to define $\Phi(D_0) = 0$ and the $\Phi(D_i) \geq 0$ for all i.

传统方法定义$\Phi(D_0) = 0$ 证明对所有i $\Phi(D_i) \geq 0$

- We consider the potential difference $\Phi(D_i) - \Phi(D_{i-1})$ for the ith operation:
  - If it is positive, $\hat{c}_i$ represents an overcharge to the ith operation, and the potential of the data structure increases.（如果差正，分摊过多，总势能增加）
  - If it is negative, $\hat{c}_i$ represents an undercharge to the ith operation, and the actual cost of the operation is paid by the decrease in the potential.

（如果差负，分摊过少，总势能支出一部分支付实际花销）

Potential Method for MultiPop Operation

势能方法 堆栈操作

- Define the potential function:（定义势能函数）
- 定义MultipPop例子中的势能函数为栈中元素的数量

  $\Phi(D_i)$ = number of objects in the stack after the ith operation.
- Starting from the empty stack D_0, we have $\Phi(D_0) = 0$.
- Since the number of objects in the stack is never negative, the stack D_i that results after the ith operation has nonnegative potential, and thus $\Phi(D_i) \geq 0 = \Phi(D_0)$ for all $0 \leq i \leq n$. 显然势能函数非负，满足定义
- The total amortized cost of n operations with respect to Φ therefore represents an upper bound on the actual cost.

  总分摊费用是实际费用的上界

Potential Method for MultiPop Operation

各种栈运算的分摊费用

- If the ith operation on a stack containing s objects is a Push operation: 计算Push操作基于势能函数的分摊价值
  - The potential difference is 经过push操作之后，势能函数的变化
    $$\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - s = 1.$$
  - The amortized cost is 分摊价值定义如下
    $$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2.$$
- If the ith operation on the stack is $\mathrm{MultiPop}(S, k)$ and that $k' = \min(k,\ s)$ objects are popped off the stack.
  - The potential difference is 经过multipop操作之后，势能函数的变化
    $$\Phi(D_i) - \Phi(D_{i-1}) =- k'.$$
  - The amortized cost is 分摊价值定义如下
    $$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0.$$
- Similarly, the amortized cost of a Pop operation is also 0.

Potential Method for MultiPop Operation

- The amortized cost of each of the three operations is $O(1)$, and thus the total amortized cost of a sequence of n operations is $O(n)$.
- Since we have already argued that $\Phi(D_i) \geq \Phi(D_0)$, the total amortized cost of n operations is an upper bound on the total actual cost.

因为该序列的最坏情形的费用$\boldsymbol{O(n)}$

Potential Method for Binary Counter（计数器）

势能方法 二进制计数器

- Define the potential function: 定义势能函数

$\Phi(D_i)$ = the number of 1's in the counter after the ith operation.

第i次运算后计数器中1的个数

- Suppose that the ith Increment operation sets t_i bits to 0. (ti个比特位变为0，实际费用至多ti+1）
  - If $\Phi(D_i) = 0$, then the ith operation resets all k bits, and so $\Phi(D_{i-1}) = t_i = k$. （第i次运算将所有K位都复位）
  - If $\Phi(D_i) > 0$, then $\Phi(D_i) = \Phi(D_{i-1}) - t_i + 1$.

    （第i次运算将ti位都复位置为0，有一位设置1）
- In either case, we have $\Phi(D_i) \leq \Phi(D_{i-1}) - t_i + 1$.

计数器从0开始，因此$\Phi(D_0) = 0$，计数器1的个数始终非负$\Phi(D_i) \geq 0$

Potential Method for Binary Counter

- The actual cost c_i is **at most** $\boldsymbol{t_i + 1}$ (set t_i bits to 0, and set at most one bit to 1).
- The potential difference （势能差）after the ith operation is $\Phi(D_i) - \Phi(D_{i-1}) \leq (\Phi(D_{i-1}) - t_i + 1) - \Phi(D_{i-1}) = 1 - t_i$.
- The amortized cost is therefore（i个操作分摊费用）
$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (\boldsymbol{t_i + 1}) + (1 - t_i) = 2.$$
- Since $\Phi(D_i) \geq 0$ for all i, the total amortized cost of a sequence of n Increment operations is an upper bound on the total actual cost, and so the worst-case cost of n Increment operations is $O(n)$

计数器从0开始，因此$\Phi(D_0) = 0$，计数器1的个数始终非负$\Phi(D_i) \geq 0$

Classroom Exercise （课堂练习）

Dynamic table insertion:

1. Initial table size $m = 1$;
2. Insert elements until the number of elements in the table $n > m$;
3. Generate a new table of size $2m$;
4. Reinsert the elements in old table into the new one;
5. Back to step 2.

For example, insert 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 one by one:

- insert 1: cost 1
- insert 2: cost 2
- insert 3: cost 3
- insert 4: cost 1
- insert 5: cost 5
- insert 6,7,8: cost 3
- insert 9: cost 9
- insert 10: cost 1

- Use amortized analysis to analyze the average cost of dynamic table insertion. We only consider the cost of insertion (no cost for table generation).

使用分摊分析来计算动态插入的费用，只考虑插入费用，不考虑表增长的费用。

Classroom Exercise （合计方法）

Solution (aggregate method):

- The ith operation causes an expansion only when $i-1$ is an exact power of 2. The cost of the ith operation is （只有当i-1是 2的次方的时候，数组才会发生扩展操作，这个额外的费用是i）

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of 2,} \\ 1 & \text{otherwise.} \end{cases}$$

- The total cost of a sequence of n dynamic table insertion operations is 总的费用就是那些不发生扩展操作的费用加上发生扩展操作的费用

$$\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n + 2n = 3n.$$

- Since the total cost of n operations is $O(n)$, the amortized cost of a single operation is $O(1)$.

备注：等比数列累加和：Sn=(a1-an*q)/(1-q)

Classroom Exercise （记账方法）

Solution (accounting method):

- Assume that $\boldsymbol{m}$ is an power of 2. （m=8为例）
- When we are inserting the $(m+1)$th element in the table, we expand the table to $2m$. （9.....16）
- We charge each insertion operation \$3 (amortized cost). （9.....16元素每个\$3）
  - Use \$1 to perform immediate insert.
  - Store \$2 as credit for future use.
- When we have $2m$ elements, we expand the table to $4m$:(每次扩展都清空)
  - \$1 is used to re-insert the item itself (items from $m+1$ to $2m$). 假设这个数据的下标是i，这个1用于自身重新插入
  - \$1 is used to re-insert another old item (items from 1 to m). 用于m之前的元素（下标为i-m）的重新插入

Solution (potential method):

- Define the potential function: 定义势能函数

$$\Phi(D_i) = 2 \cdot num[T] - size[T].$$

  - $num[T]$ is the number of elements in T. 表中元素个数
  - $size[T]$ is the size of the table. 表的大小
- $\Phi(T_0) = 0$ and $\Phi(T)$ is always ≥ 0.
  - Immediately after an expansion, we have $num[T] = size[T]/2$, and thus $\Phi(T) = 0$.
  - Immediately before an expansion, we have $num[T] = size[T]$, and thus $\Phi(T) = num[T]$.

Classroom Exercise

- 没有触发数组扩张的情况：If the ith TABLE-INSERT operation does not trigger an expansion, then we have $size[T_i] = size[T_{i-1}]$（表的大小没变） and the amortized cost of the operation is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\ &= 1 + \big(2 \cdot num(T_i) - size(T_i)\big) - \big(2 \cdot num(T_{i-1}) - size(T_{i-1})\big) \\ &= 1 + 2\big(num(T_i) - num(T_{i-1})\big) = 3.\end{aligned}$$

- 有触发数组扩张的情况： If the ith operation does trigger an expansion, then we have $size[T_i] = 2 \cdot size[T_{i-1}]$ and $num[T_{i-1}] = size[T_{i-1}]$. Thus, the amortized cost of the operation is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\ &= num[T_i] + (2 \cdot num[T_i] - size[T_i]) - (2 \cdot num[T_{i-1}] - size[T_{i-1}]) \\ &= num[T_i] + (2 \cdot num[T_i] - 2 \cdot num[T_{i-1}]) - num[T_{i-1}] \\ &= 3 \cdot num[T_i] - 3 \cdot num[T_{i-1}] = 3.\end{aligned}$$

Summary of Amortized Analysis

- When should we use amortized analysis, rather than probabilistic analysis? We can't determine each single, but we know the total.
  - Amortized analysis always gives the upper bound.
  - For accounting method and potential method, some tricky design is needed.
- For a sorting algorithm for n arrays, we can't determine each single, nor the total. Hence amortized analysis is not applicable for it. 对于排序问题，我们不能用分摊分析的方法

EMPIRICAL ANALYSIS （实验分析）

Problem of Theoretical Analysis

- Previous analysis are based on asymptotic notations. However, there are also some issues when we are dealing with real-world problems.
  - Asymptotic notations only consider the case when the size tends to infinity. 渐进分析主要考虑是问题规模趋近无穷大的情况
- Which of the algorithm with the following complexity will you choose? 当问题规模不大的情况下，考虑下面的两个复杂度

$$10^5 n \text{ vs. } n^2$$

  - Based on asymptotic notations, we choose the one with $10^5 n$.
  - However, if our input scale only range from 1 to 10^5, we should choose the one with n^2.

Empirical Analysis 实验分析

好的算法不仅要考虑计算速度，还要考虑解的质量

- Empirical analysis (实验分析) is most useful for hard problem or randomized algorithm.
  - Data generation (benchmark). 数据选择生成
  - Algorithm implement (software and hardware).实现算法
  - Result analysis (visualization). 计算结果分析

Conclusion

After this lecture, you should know:

- Why do we need probabilistic analysis?
- How to use probabilistic analysis for average case analysis?
- Which case is suitable for applying amortized analysis?
- What are the differences among three amortized analysis methods?

Homework

- Page 31

  3.1

  3.2

  3.4

  3.8

谢谢

有问题欢迎随时跟我讨论