算法设计与分析

Lecture 3: Algorithm Analysis

卢杨
厦门大学信息学院计算机科学系
luyang@xmu.edu.cn

PROBABILISTIC ANALYSIS

Probabilistic Analysis

- Average-case analysis determines the average (or expected) performance.
  - 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.
- 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.

Linear Search

- 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)

$k \leftarrow 1$

while $k \leq n$ and $x \neq A[k]$ **do**

$\quad k \leftarrow k + 1$

if $k > n$ **then return** 0

else return k

Probabilistic Analysis of Linear Search

- To simplify the analysis, let us assume:
  - $A[1 \dots n]$ contains the numbers 1 through n, which implies that all elements of A are distinct.
  - The search key x is in A.
  - The search key x is uniformly drawn from $[1 \dots n]$.
  - We only count the number of key comparisons.

```
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

- Probability of x being found at index k is $1/n$ for each value of k.
- If $x = A[k]$, then the number of comparison is k.
- 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 $\Theta(n)$.
- Think: What if the key x is not uniform distributed?

Probabilistic Analysis of Insertion Sort

- 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.
  - All $n!$ permutations of A appear with equal probability as the input.
  - We only count the number of key comparisons.

```
InsertSort(A)
for j ← 2 to n do
        key ← A[j]
        i ← j − 1
        while i > 0 and A[i] > key do
                A[i + 1] ← A[i]
                i ← i − 1
        A[i + 1] ← 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.
- Now we consider inserting $key = A[j]$ in the proper position in $A[1 \dots j]$.
- If its proper position is $k(1 \le k \le j)$, then the number of comparisons performed in order to insert key in $A[k]$ is:

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

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

Probabilistic Analysis of Insertion Sort

- Since the probability that its proper positions in $A[1 \dots j]$ is $1/j$, so the number of comparisons needed to insert $A[j]$ in its proper position in $A[1 \dots j]$ 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 $\mathrm{InsertSort}(A)$ is:

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

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

What is the order of this term?

The Hiring Problem

- The problem scenario:
  - You are using an employment agency to hire a new office assistant.
  - 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.

The Hiring Problem

```
HireAssistant(n)
best ← 0
for i ← 1 to n do
    interview candidate i
    if candidate i is better than candidate best then
        best ← i
        hire candidate i.
```

Analysis of the Hiring Problem

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

Probabilistic Analysis of the Hiring Problem

- 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 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 $n!$ permutations appears with equal probability.

Probabilistic Analysis of the Hiring Problem

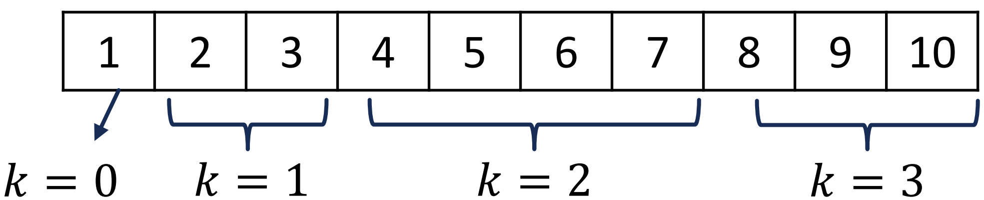
- Candidate i is hired if and only if candidate i is better than each of candidates $1, 2, \ldots, i-1$.
- Base on the assumption that the candidates arrive in random order, any one of these i candidates is equally likely to be the best one so far.
- Thus, the probability of hiring candidate i is $1/i$.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

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

- 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 $O(\lg n)$.



$$
\begin{aligned}
\sum_{i=1}^{n} \frac{1}{i} &\le \sum_{k=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^k - 1} \frac{1}{2^k + j} \\
&\le \sum_{k=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^k - 1} \frac{1}{2^k} \\
&= \sum_{k=0}^{\lfloor \lg n \rfloor} 1 \\
&\le \lg n + 1 .
\end{aligned}
$$

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?

Examples of Probabilistic Analysis

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

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?

Examples of Probabilistic Analysis

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

- Let $A[1 \dots n]$ 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

Amortized Analysis

- In some algorithms, the average-case performance is difficult to be determined because each operation takes different time.
- 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.
- An amortized analysis guarantees the average performance of each operation in the worst case.

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.

- 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

- 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.
- In the worst case, the average cost, or amortized cost, per operation is therefore $T(n)/n$.
- Note that this amortized cost applies to each operation, even when there are several types of operations in the sequence.

MultiPop Operation

- 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

- Now we add a new stack operation $\text{MultiPop}(S, k)$: 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 $\text{MultiPop}(S, k)$ on a stack of s objects?
  - It varies for different S.

```
MultiPop(S, k)
while not StackEmpty(S) and k ≠ 0 do
      Pop(S)
      k ← k − 1
```

top → 23			
17			
6			
39			
10	top → 10		
47	47		
MultiPop(S, 4)	MultiPop(S, 7)		

Aggregate Method for MultiPop Operation

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

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

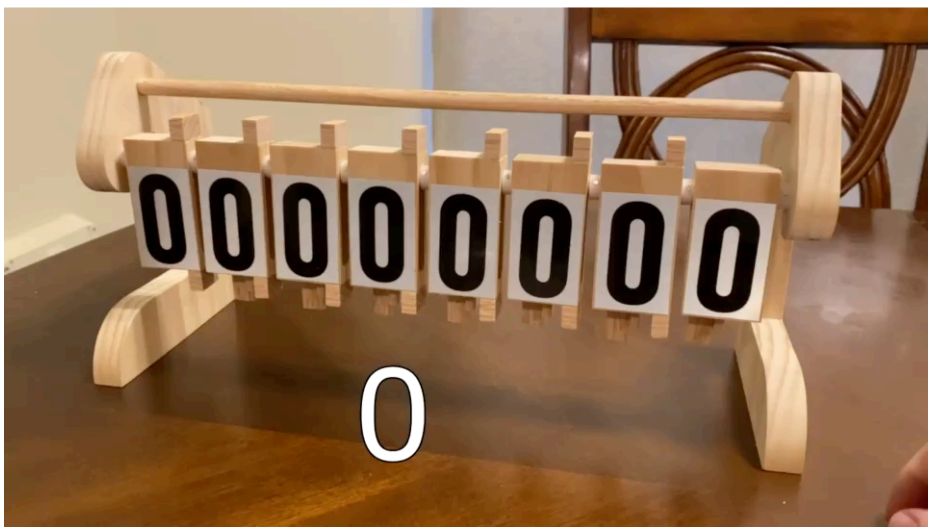
- For a stack with at most n elements, the worst-case time of MultiPop is $O(n)$, and we may have $O(n)$ MultiPop operations . Hence a sequence of n MultiPop operations costs $O(n^2)$.
- This analysis is correct but the upper bound is too high. We have at most n elements to pop. How does $O(n^2)$ come?
  - This upper bound situation will never be happened, because it is impossible to pop n elements in MultiPop for n times.

Aggregate Method for MultiPop Operation

- Notice: each element is popped at most once after it is pushed into a stack.
- Therefore, the total number of Pop (include the ones in MultiPop) operations is at most n.
- Therefore, any sequence of n Push, Pop, and MultiPop operations on an initially empty stack can cost at most $O(n)$.
- The average cost of an operation is $O(n)/n = O(1)$.
  - Although it looks like $O(n)$.

Binary Counter

- 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 $A[0 \ldots k-1]$ 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 ← 0
while i < n and A[i] = 1 do
        A[i] ← 0
        i ← i + 1
if i < n then
        A[i] ← 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

Aggregate Method for Binary Counter

- What is the average cost of a single execution of Increment, if we count the number of bits flipped as the cost?
- Follow the idea of amortized analysis, we consider a sequence of n Increment operations on an initially zero counter.
- 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?

Aggregate Method for Binary Counter

- 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.

Aggregate Method for Binary Counter

- 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

- 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.
- 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).
- Credit can be used later on to help pay for operations whose amortized cost is less than their actual cost.

Accounting Method

- 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.

Accounting Method for MultiPop Operation

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:

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).
  - At any point in time, every element on the stack has \$1 of credit on it, which is for the cost of popping it.
  - To pop (from Pop or $\mathrm{MultiPop}$) 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 $\mathrm{MultiPop}$ operations, the total amortized cost is $O(n)$.

Accounting Method for Binary Counter

- Let us once again use \$1 to represent each unit of cost.
- For the accounting method, let us charge an amortized cost of \$2 to set a bit to 1.
  - 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.
  - 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.
  - 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 $\mathrm{Increment}$ is \$2.
- Thus, for n $\mathrm{Increment}$ operations, the total amortized cost is $O(n)$, which bounds the total actual cost.

Potential Method

- In accounting method, we associate credits with elements in the data structure.
- Similarly, in potential method (势能方法), we store "potential" of the data structure for future operations.
  - We start with an initial data structure D_0 on which n operations are performed.
  - Let D_i be the data structure that results after applying the ith operation to data structure D_{i-1}, for each $i = 1, 2, \ldots, n$.
  - 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.

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

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

- The total amortized cost of the n operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right)$$
$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0).$$

Potential Method

- Just like accounting method, we can pay for future operations by potential in potential method.
- 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.
- 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.
  - 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.

Potential Method for MultiPop Operation

- Define the potential function:

$\Phi(D_i)$ = 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 $0 \leq i \leq n$.
- The total amortized cost of n operations with respect to Φ therefore represents an upper bound on the actual cost.

Potential Method for MultiPop Operation

- If the ith operation on a stack containing s objects is a Push operation:
  - The potential difference is
$$\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - s = 1.$$
  - The amortized cost is
$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2.$$
- 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
$$\Phi(D_i) - \Phi(D_{i-1}) = -k'.$$
  - The amortized cost is
$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0.$$
- 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.

Potential Method for Binary Counter

- Define the potential function:

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

- Suppose that the ith Increment operation sets t_i bits to 0.
  - If $\Phi(D_i) = 0$, then the ith operation resets all k bits, and so $\Phi(D_{i-1}) = t_i = k$.
  - If $\Phi(D_i) > 0$, then $\Phi(D_i) = \Phi(D_{i-1}) - t_i + 1$.
- In either case, we have $\Phi(D_i) \leq \Phi(D_{i-1}) - t_i + 1$.

Potential Method for Binary Counter

- The actual cost c_i is at most $t_i + 1$ (set t_i bits to 0, and set at most one bit to 1).
- The potential difference 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
  $$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (t_i + 1) + (1 - t_i) = 2.$$
- 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 $O(n)$.

Classroom Exercise

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).

Classroom Exercise

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

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

- The total cost of a sequence of n dynamic table insertion operations is

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

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

Classroom Exercise

Solution (accounting method):

- Assume that m is an power of 2.
- When we are inserting the $(m+1)$th element in the table, we expand the table to $2m$.
- We charge each insertion operation \$3 (amortized cost).
  - Use \$1 to perform immediate insert.
  - Store \$2 as credit for future use.
- When we have $2m$ elements, we expand the table to $4m$:
  - \$1 is used to re-insert the item itself (items from $m+1$ to $2m$).
  - \$1 is used to re-insert another old item (items from 1 to m).

Classroom Exercise

Solution (potential method):

- Define the potential function:

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

  - $num[T]$ is the number of elements in T.
  - $size[T]$ is the size of the table.
- $\Phi(T_0) = 0$ 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

- If the ith TABLE-INSERT operation does not trigger an expansion, then we have $size[T_i] = size[T_{i-1}]$ and the amortized cost of the operation is

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

- 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.

EMPIRICAL ANALYSIS

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.
- Which of the algorithm with the following complexity will you choose?

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

  - 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

- Empirical analysis (实验分析) is most useful for hard problem or randomized algorithm.
  - Data generation (benchmark).
  - Algorithm implement (software and hardware).
  - Result analysis (visualization).

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

  3.6

  3.8

谢谢

有问题欢迎随时跟我讨论