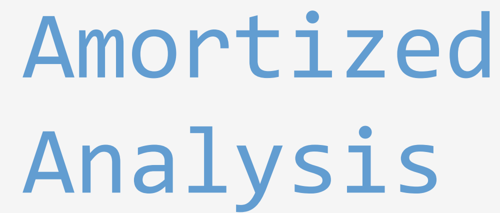
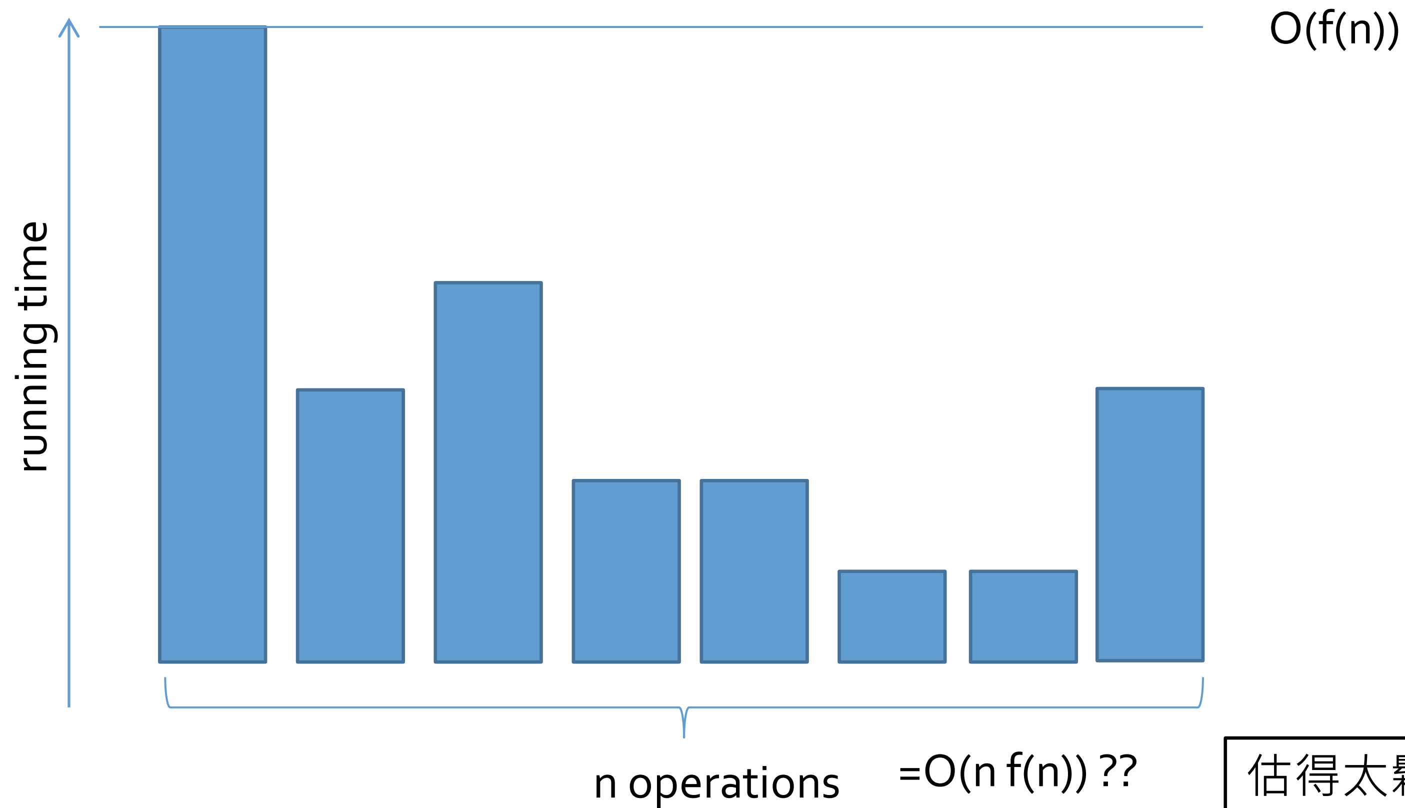


Michael Tsai

2011/12/2

為什麼需要Amortied Analysis



Amortized Analysis

- Amortized:
  - Reduce or extinguish (a debt) by money regularly put aside
  - 攤銷
- 可以想成是把一連串的operation一起考慮它的花費(也就是執行時間)
- 一起考慮的時候有時候估計比較準確(bound比較緊)

Aggregated Analysis

- T(n): n個operation最差狀況下所需花的時間
- 因此, 每個operation攤分的所需時間(amortized cost)為T(n)/n

- 在aggregated analysis中, 我們不區分不同operation的所需時間 (也就是amortized cost對不同operation都一樣)

一: Stack

- Stack的基本動作:
- Push(S,x) $O(1)$
- Pop(S) $O(1)$
- Multipop(S,k)

```
while not STACK_EMPTY(S) and k>0
      POP(S)
      k=k-1
```

$\min(s,k) \quad = O(n)$

- 使用平常的方法:

$$T(n) = O(n^2)$$

Aggregated Analysis

- 但是事實上，雖然單一Multipop operation要花很多時間
- 任何n個連續的operation最多只會花$O(n)$.
- 因為stack裡面最多只會放n個(n個push後)
- 而不管是pop或Multipop, 最多真正pop的數目也只會是n
- 因此總共花的時間不會超過$O(n)$.
- $T(n) = O(n)$
- 所以平均每個operation所花時間為$\frac{O(n)}{n} = O(1)$
- 在aggregated analysis裡面, 我們把每個operation的amortized cost設定為average cost.
- 在這個例子裡,也就說三種operation的amortized cost都是$O(1)$.

二：二進位計數器

- A[o..k-1]放 二進位數的bits
- k位數

A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
1	0	0	0	1	1

```
INCREMENT(A)
i=0
while A[i]==1 and i<A.length
     A[i]=0
     i=i+1
if i<A.length
     A[i]=1
```

- n個operation要花多少時間?

使用平常的方法

- 執行一次INCREMENT最多花$\Theta(k)$
- 最糟的狀況: 每一位都是1, 全部清成0
- 因此n次INCREMENT要花$O(nk)$
- 正確的bound, 但是太鬆

Aggregated Analysis

第i位數翻$\left\lfloor \frac{n}{2^i} \right\rfloor$次

翻$\left\lfloor \frac{n}{4} \right\rfloor$次

翻n次

Counter	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	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

翻$\left\lfloor \frac{n}{8} \right\rfloor$次

翻$\left\lfloor \frac{n}{2} \right\rfloor$次

Aggregated Analysis

- 所以n個operation的cost應為:
- $T(n) = \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n = O(n)$
- Amortized cost for each operation:
- $\frac{T(n)}{n} = \frac{O(n)}{n} = O(1)$

The accounting method

- “發明”一種給每個operation不同”虛擬cost”的方法
- (用這種方法可能分析T(n)比較容易)
- 每個operation給**不一樣的**amortized cost
- 每個operation的amortized cost可能比實際的cost少一點或多一點
- 可以把amortized cost > 實際cost想成是存錢
- 可以把amortized cost < 實際cost想成是花錢
- 有些operation可能多存一些錢, 供給後面的其他花錢的operation使用

限制

- 如果我們想要用amortized cost總和來分析T(n)最大有多少, 則必須滿足
- $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$
- 這樣的話, 如果$\sum_{i=1}^{n} \hat{c}_i$是$O(f(n))$則$\sum_{i=1}^{n} c_i$也是$O(f(n))$
- 注意上面的條件是"任何n個operation"都要符合喔!
- 如果不符合: 則不能保障"如果$\sum_{i=1}^{n} \hat{c}_i$是$O(f(n))$則$\sum_{i=1}^{n} c_i$也是$O(f(n))$"
- (也就是說, 不可以使某個operation多花的錢大於之前存的錢)

一：Stack

- 假設我們設定以下的amortized cost:
- Push: 2
- Pop: 0
- Multipop: 0
- 這樣, push先存下的錢, 足夠讓pop和multipop花嗎?
- Pop&Multipop的amortized cost設成0, 因此會花先存下來的錢

存錢和花錢的故事

Push: 付兩塊錢

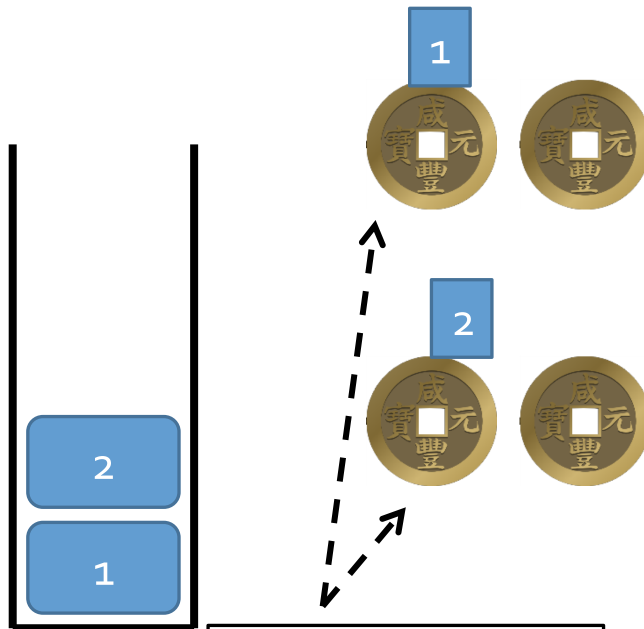
一塊錢付掉push本身花的時間

一塊錢先存起來
(跟著push進去stack的item)

Push: 付兩塊錢

一塊錢付掉push本身花的時間

一塊錢先存起來
(跟著push進去stack的item)



存起來的錢,
是用來給之後
pop出來的花的
(不管是pop or
Multipop)

也就是說, 不管n個operation的順序是怎麼樣,
永遠不會有情形是花錢花超過已經存下來的錢.
每個已經在stack裡面的item都已經預先存下了錢給pop or Multipop用.

→the total amortized cost is always a upper bound of the total actual cost.

Total Amortized Cost

- 然後呢?
- Total amortized cost=2×push operation數目=O(n)
- 因此total actual cost也是O(n)

二：二進位計數器

- 假設我們這樣定義amortized cost:
- 設定一個bit為1的時候: 2
- 其他都是0

- 這樣的設計會不會有錢不夠花的狀況?

存錢和花錢的故事 Part II

A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0

A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	1

A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	1	0

設A[0]=1: 付兩塊錢

一塊錢付掉設A[0]=1本身花的時間

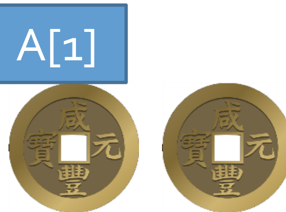
一塊錢先存起來 (跟著A[0]的bit 1)

A[0]

設A[0]=0: 不付錢

用存起來的一塊錢付

A[1]



設A[1]=1: 付兩塊錢

一塊錢付掉A[1]=1本身花的時間

一塊錢先存起來 (跟著A[1]的bit 1)

也就是說, 不管n個operation的順序是怎麼樣,永遠不會有情形是花錢花超過已經存下來的錢.每個bit=1都已經預先存下了錢給set bit=0的時候用.

→the total amortized cost is always a upper bound of the total actual cost.

Total Amortized Cost

- Total amortized cost=n個INCREMENT設定bit=1的次數
- 每次最多只設定一個bit為1
- 因此n次最多設定n個bit為1
- Total amortized cost=O(n)
- Total actual cost=O(n)

The potential method

- 比較:
- Accounting method是使用"先存後花"的方式
- Potential method則是用potential來計算"之前的operation使目前的資料結構有殘存多少能量"
- 物理定義:**Potential energy** is the energy stored in a body or in a system due to its position in a force field or due to its configuration*

*http://en.wikipedia.org/wiki/Potential_energy

定義

- D_i:做完第i個operation之後的data structure
- (一開始為D_0,結束n個operation後為D_n)
- $\Phi(D_i)$: D_i的potential (自己定義)
- 則第i個operation的amortized cost為
- $\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$
- amortized cost=actual cost加上potential的改變
- $\sum_{i=1}^{n} \hat{c_i} = \sum_{i=1}^{n}(c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$

限制

- 要怎麼確定total amortized cost是total actual cost的upper bound呢?
- 要確定:
- $\sum_{i=1}^{n} \hat{c_i} \geq \sum_{i=1}^{n} c_i$
- $\sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \geq \sum_{i=1}^{n} c_i$
- 也就是$\Phi(D_n) \geq \Phi(D_0)$
- 通常不知道n是多少
- 所以也就是必須對於任何i
- $\Phi(D_i) \geq \Phi(D_0)$
- 通常會希望$\Phi(D_0) = 0, \Phi(D_i) \geq 0$

概念

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

$$\Phi(D_i) - \Phi(D_{i-1}) \geq 0$$

amortized cost ≥ actual cost
則可以想成是存錢.
(存到potential裡面去)

$$\Phi(D_i) - \Phi(D_{i-1}) < 0$$

amortized cost < actual cost
則可以想成是花錢.
(從potential裡面拿出錢來)

一：Stack

- 定義Φ: stack裡面有多少個item
- 則$\Phi(D_0) = 0$ (一開始stack沒有東西)
- 且$\Phi(D_i) \geq \Phi(D_0) = 0$
- →total amortized cost是total actual cost的upper bound

Amortized costs

- 如果第i個operation是push, 目前stack有s個item
- 則$\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - s = 1$
- $\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$
- 如果第i個operation是pop, 目前stack有s個item
- 則$\Phi(D_i) - \Phi(D_{i-1}) = (s-1) - s = -1$
- $\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0$
- 如果第i個operation是Multipop(S,k) , 目前stack有s個item
- (請同學練習看看)
- 設$k' = \min(k, s)$
- 則$\Phi(D_i) - \Phi(D_{i-1}) = (s - k') - s = -k'$
- $\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = \mathrm{k}' - \mathrm{k}' = 0$

Total actual cost

- 因為三個operation的amortized cost都是O(1)
- 因此n個operation (由三種operation組成)的amortized cost是O(n)
- Total amortized cost是total actual cost的upper bound
- 因此total actual cost也是O(n)

二：二進位計數器

- 定義Φ: A[]裡面有幾個bit是1
- 假設b_i為做完第i個operation後A[]裡面bit是1的數目
- 假設第i個operation把t_i個bit從1設成0
- 則第i個operation的實際花費最多為$t_i + 1$
- (t_i個1→0, 最多1個0→1) (想想什麼時候沒有0→1)
- 如果$b_i = 0$, 則$b_{i-1} = t_i$
- 如果$b_i > 0$, 則$b_i = b_{i-1} - t_i + 1$
- →$b_i \leq b_{i-1} - t_i + 1$
- $\Phi(D_i) - \Phi(D_{i-1}) = b_i - b_{i-1} \leq 1 - t_i$

$\Phi(D_i) - \Phi(D_{i-1}) = b_i - b_{i-1} \leq 1 - t_i$

Amortized costs

- $\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (t_i + 1) + (1 - t_i) = 2$
- 如果一開始counter=o, 則$\Phi(D_i) \geq \Phi(D_0) = 0$ (所有i)
- →total amortized cost為total actual cost之upper bound
- n個INCREMENT之total amortized cost為O(n)
- 所以total actual cost也是O(n)

變形

- 如果一開始我們不從0開始count呢?
- $b_0 \neq 0, D_i \geq D_0$ 不一定成立, total amortized cost不一定是total actual cost的upper bound
- 使用原本的定義的話
- $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (t_i + 1) + (1 - t_i) = 2$
- $\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i - \Phi(D_n) + \Phi(D_0)$
  $\leq 2n - \Phi(D_n) + \Phi(D_0)$
  $= 2n - b_n + b_0$
- 因為只有k位數, $0 \leq b_0, b_n \leq k$
- 所以當k=$O(n)$的時候
- 我們知道total actual cost =$O(n)$!

來一個實際一點的例子...

- 大家很不愛用malloc, realloc, delete
- 陣列一開始就開很大
- 浪費記憶體空間

- Dynamic table: 怎麼隨時動態調整table大小, 使得不會有太多空間浪費, 平均起來也不會浪費很多時間?
- load-factor: $\alpha(T) = \frac{T.num}{T.size}$ (跟hash table的定義一樣)

只有insert的case

```
Table_Insert(T,x)
if T.size==0
      allocate T.table with 1 slot
      T.size=1
if T.num==T.size
      allocate new-table with 2*T.size slots
      insert all items in T.table into new-
table
      free T.table
      T.table=new-table
      T.size=2*T.size
insert x into T.table
T.num=T.num+1
```

這邊insert了T.size次

這邊insert了1次

假設主要花費時間是這些insertion

原本的方法 (非amortized analysis)

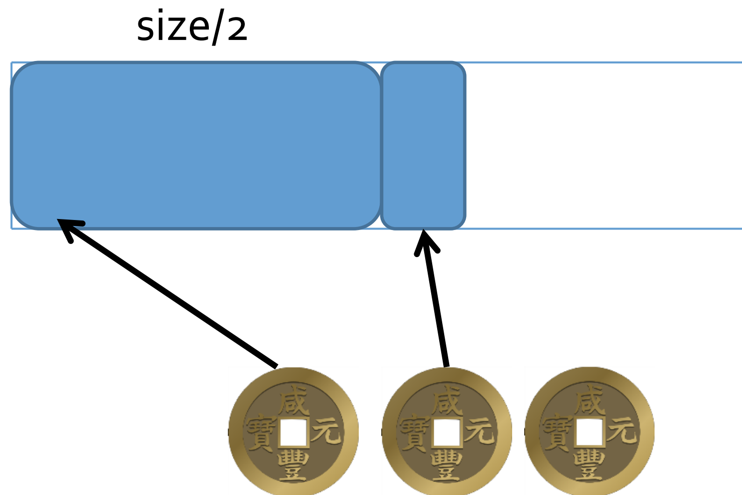
- $c_i = ?$
- 如果原本的table還有空間, 則$c_i = 1$
- 如果原本的table沒有空間, 則$c_i = i$
- (1個新的insertion跟i-1個搬移insert到新的table)
- 所以一個operation的cost為$O(n)$
- n個operation的cost為$O(n^2)$
- 一樣; 正確但是不tight

Aggregate analysis

- $c_i = \begin{cases} i & \text{, if i-1 is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$
- $\sum_{i=1}^{n} c_i \leq \mathrm{n} + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j = n + 2^{\lfloor \log n \rfloor + 1} \leq n + 2n = 3n$
- 所以n個operation的cost=3n=O(n)
- 平均每個operation的cost=3=O(1)

Accounting method

- 我們設計每個table-insert的amortized cost是3



insert的時候付3塊錢

一塊錢把最新的item insert到table的時候花掉

一塊錢給之前已經在裡面的其中一個item (準備當之後滿了需要移動的時候可以花掉)

一塊錢給自己
(準備當之後滿了需要移動的時候可以花掉)

當全滿的時候, 則每一個item都有預存了一塊錢可以拿來搬移(insert到新的table)

Potential method

- 定義$\Phi(T) = 2T.num - T.size$
- 則Table-insert沒有造成把table變大的時候:
- $\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) = 1 + (2num_i - size_i) - (2(num_i - 1) - size_i) = 3$
- 當Table-insert造成把table變大的時候:
- $\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = num_i + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) = num_i + \left(2num_i - 2(num_i - 1)\right) - \left(2(num_i - 1) - (num_i - 1)\right) = 3$

Reading assignment

- Section 17.4.2
- 可以長大也可以縮小的dynamic table
- 作業會有17.4-2 (延續你的閱讀的題目)