Chapter 17
Amortized Analysis
平摊（摊还、分摊、分期）分析

songyou@buaa.edu.cn

An example: Dynamic tables

void * malloc(size_t size);

c : malloc/free

c++: new/delete

- We might allocate space for a table, only to find out later that it is not enough. We must then reallocate the table with a larger size and copy all objects stored in the original table over into the new, larger table.
- Similarly, if many objects have been deleted from the table, it may be worthwhile to reallocate the table with a smaller size.
- We assume that the dynamic table supports the operations TABLE-INSERT and TABLE-DELETE.
  - TABLE-INSERT inserts into the table an item that occupies a single slot, that is, a space for one item.
  - Likewise, TABLE-DELETE removes an item from the table, thereby freeing a slot.

Amortized Analysis

- **In an amortized analysis, we average the time required to perform a sequence of data-structure operations over all the operations performed.**
  （在多个操作中，求一个操作的平均时间）
- **With amortized analysis, we can show that the average cost of an operation is small, if we average over a sequence of operations, even though a single operation within the sequence might be expensive.**
- **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 cost, 分摊消费：在一个操作上的平均消费 (cost)**

Amortized Analysis

17.1 Aggregate analysis (聚集分析)

17.2 The accounting method (记账法)

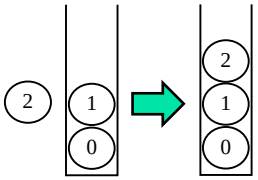
17.3 The potential method (势能法)

17.1 Aggregate analysis

- **In aggregate analysis, 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 $T(n)/n$.**

(1) Stack operations



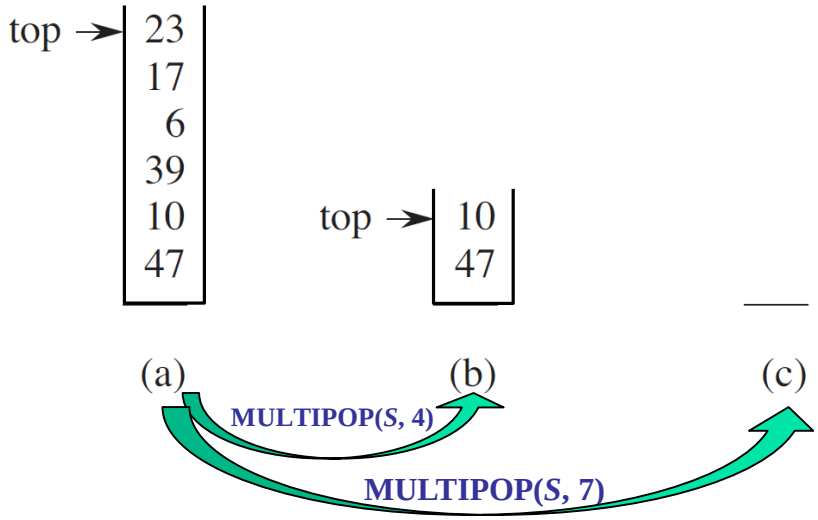
- **PUSH(*S*, *x*): pushes object *x* onto stack *S*.** **O(1)**
- **POP(*S*): pops the top of stack *S* and returns the popped object. Calling POP on an empty stack generates an error.** **O(1)**
- **MULTIPOP(*S*, *k*): removes the *k* top objects of stack *S*, popping the entire stack if the stack contains fewer than *k* objects.**

  (The stack has *s* objects.) **O(min(*s*, *k*))?**

(1) Stack operations

MULTIPOP(S, k): removes the k top objects of stack S, popping the entire stack if the stack contains fewer than k objects. (The stack has s objects.)

```
MULTIPOP(S, k)
1  while not STACK-EMPTY(S) and k > 0
2      POP(S)
3      k = k − 1
```



O(min(s, k))

(1) Stack operations

Let us analyze a sequence of *n* PUSH, POP, and MULTIPOP operations on an initially empty stack.

```
STACK(S, n)
1  S = NULL
2  for i←1  to  n
3      One of ( PUSH(S, i), POP(S), MULIPOP(S, k) )
```

```
MULTIPOP(S, k)
1  while not STACK-EMPTY(S) and k > 0
2      POP(S)
3      k = k − 1
```

- **Running time?**

  $O(n^2)$

- **Correct. Not tight.**
- **Running time tightly? $O(n)$**
- **We can pop each object from the stack at most once for each time we have pushed it onto the stack. Therefore, the number of times that POP can be called on a nonempty stack, including calls within MULTIPOP, is at most the number of PUSH operations, which is at most *n*.**

  push一个对象后，至多能被pop一次。但至多*n*次push。

$O(\min(s, k))$

worst-case: $O(n)$

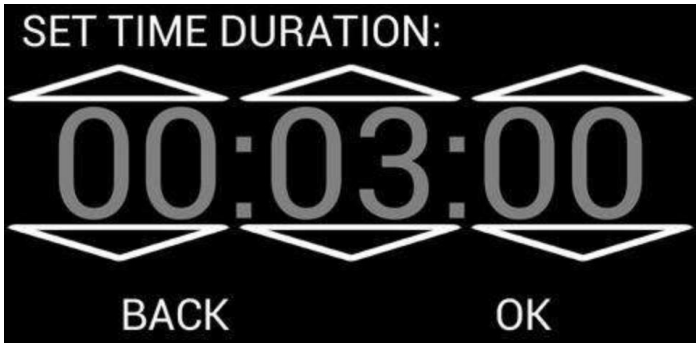
(1) Stack operations

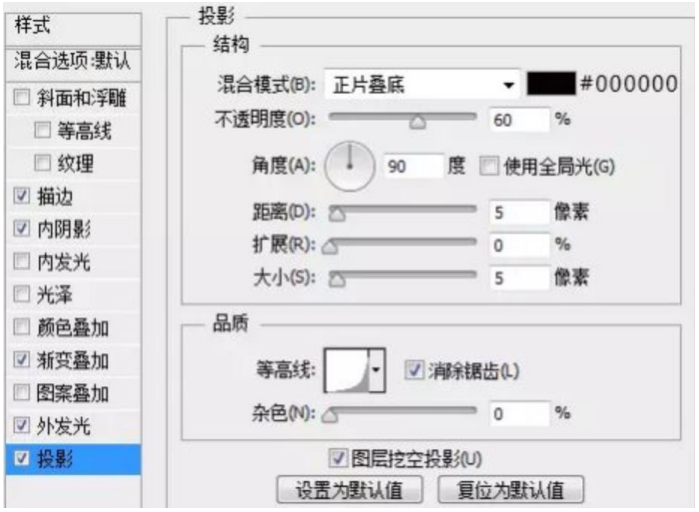
```
STACK(S, n)
1  S = NULL
2  for i←1  to  n
3       One of ( PUSH(S, i), POP(S), MULIPOP(S, k) )
```

- **Running time tightly. O(*n*)**
- **The average cost of an operation is O(*n*)/*n* = O(1)**
- **In aggregate analysis, we assign the amortized cost of each operation to be the average cost.**
- **For STACK operations, the average cost (the running time) of a stack operation is O(1), we did not use probabilistic reasoning.**
- **Dividing this total cost by *n* yielded the average cost per operation, or the amortized cost.**

(2) Incrementing a binary counter

Consider the problem of implementing a *k*-bit binary counter that counts upward from 0.





(2) Incrementing a binary counter

- **We use an array $A[0 .. k\text{-}1]$ of bits as the counter.** （使用位数组作为计数器）
- **A number x that is stored in the counter has its lowest-order bit in $A[0]$ and highest-order bit in $A[k\text{-}1]$, so that**

$$x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$$

- **Initially, $x = 0$, and thus $A[i] = 0$ for all i .**
- **To add 1 (modulo 2^k) to the value in the counter, we use the following procedures.**

```
INCREMENT(A)      // 加 1 算法
                  // index of A
1   i = 0
2   while i < A.length and A[i] == 1
3       A[i] = 0
4       i = i + 1
5   if i < A.length
6       A[i] = 1
```

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

(2) Incrementing a binary counter

- **Let us analyze a sequence of *n* INCREMENT operations on an initially zero counter.** （从0开始，计数到*n*）

```
COUNTER(A, n)
1  A = 0
2  for j←1  to  n
3      INCREMENT(A)
```

```
INCREMENT(A)
1  i = 0
2  while i < A.length and A[i] == 1
3      A[i] = 0
4      i = i + 1
5  if i < A.length
6      A[i] = 1
```

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

- **Running time of COUNTER?**

  O(*nk*), k 是 A 的位数

- **Correct. Not tight.**
- **Tight running time? O(*n*)**

(2) Incrementing a binary counter

```
COUNTER(A, n)
1  A = 0
2  for j←1  to  n
3      INCREMENT(A)
```

```
INCREMENT(A)
1  i = 0
2  while i < A.length and A[i] == 1
3      A[i] = 0
4      i = i + 1
5  if i < A.length
6      A[i] = 1
```

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

- **Not all bits flip each time INCREMENT is called**
- ***A*[0] does flip each time (*n* times) INCREMENT is called**
- ***A*[1], flips only every other time: *n*/2 times**
- ***A*[2], flips only every fourth time: *n*/4 times**
- ***A*[*i*] flips $n/2^i$ times**
- **The total number of flips in COUNTER**

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

The amortized cost per operation:
$O(n)/n = O(1)$

application of aggregate analysis -- convex hull

GRAHAM-SCAN(Q)
1 let p_0 be the point in Q with the minimum y-coordinate,
  or the leftmost such point in case of a tie
2 let $\langle p_1, p_2, ..., p_m \rangle$ be the remaining points in Q, sorted by
  polar angle in counterclockwise order around p_0
  (if more than one point has the same angle, remove all
  but the one that is farthest from p_0)
3 PUSH(p_0, S)
4 PUSH(p_1, S)
5 PUSH(p_2, S)
6 **for** $i \leftarrow 3$ **to** n
7 **while (** the consecutive segments formed by points
    NEXT-TO-TOP(S), TOP(S), and p_i
    make a nonleft turn**)**
8 POP(S)
9 **PUSH(p_i, S)**
10 **return** S

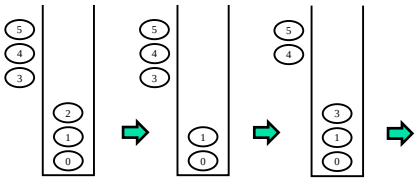
T(n) = ?

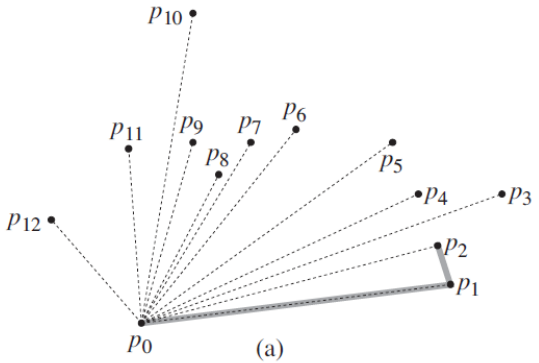
$\Theta(n)$

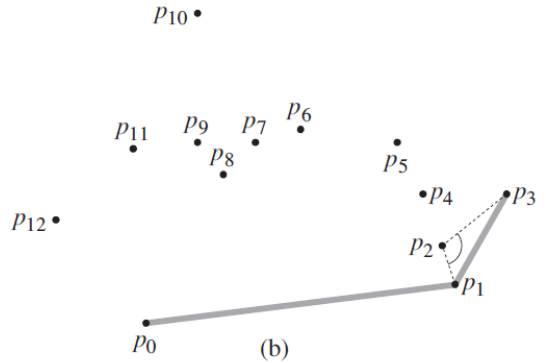
$O(n \lg n)$, using merge sort and the cross-product method to compare angles.

O(1)
O(1)
O(1)
$O(n-3)$

Aggregate analysis: **while** loop takes $O(n)$ time overall. For $i = 0, 1, ..., n$, each point p_i is pushed onto stack S exactly once, there is at most one POP operation for each PUSH operation. At least three points p_0, p_1, and p_n are never popped from the stack, so that in fact at most (n - 2) POP operations are performed in total?







17.2 The accounting method

- **Amortized cost** : the amount we charge an operation.
  - **Credit** : when an operation's amortized cost exceeds its actual cost, we assign the difference to specific objects in the data structure as credit.
  - Credit can help pay for later operations whose amortized cost is less than their actual cost.
- We denote the actual cost of the *i*th operation by c_i and the amortized cost of the *i*th operation by $\hat{c}_i$, we require

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

  For all sequences of *n* operations.
- The total credit $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i$

17.2 The accounting method - Stack operations

- **The actual cost of the *i*th operation: c_i**
- **The amortized cost of the *i*th operation: $\hat{c}_i$**

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

- **The actual costs of the operations**

Push	**1**
Pop	**1**
Multipop	**min(k, s)**

- **We assign the amortized costs**

Push	**2**
Pop	**0**
Multipop	**0**

- **We can pay for any sequence of stack operations by charging the amortized costs?**

17.2 The accounting method - Stack operations

- **The actual cost of the *i*th operation: c_i**
- **The amortized cost of the *i*th operation: $\hat{c}_i$**

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

- **The actual costs of the operations**

Push	**1**
Pop	**1**
Multipop	**min(*k*, *s*)**

- **We assign the amortized costs**

Push	**2**
Pop	**0**
Multipop	**0**

- **We can pay for any sequence of stack operations.**
- **For any sequence of *n* Push, Pop, and Multipop operations, the total amortized cost is O(*n*), so is the total actual cost.**

```
STACK(S, n)
1  S = NULL
2  for i←1  to  n
3       One of ( PUSH(S, i), POP(S), MULIPOP(S, k) )
```

17.2 The accounting method - Incrementing a binary counter

Charge an amortized cost of 2 dollars to set a bit from 0 to 1. When a bit is set, we use 1 dollar (out of the 2 dollars charged) to pay for the actual setting of the bit, and we place the other dollar on the bit as credit to be used later when we flip the bit back to 0. At any point in time, every 1 in the counter has a dollar of credit on it, and thus we can charge nothing to reset a bit to 0.

(1) 0到1时，支付分摊消费2元（其中1元用于实际转化0到1，结余1元作为信用）；
(2) 1到0时，分摊消费为0元，用信用来支付实际消费。

- The amortized cost of an INCREMENT is at most 2 dollars.
- For n INCREMENT operations, the total amortized cost is $O(n)$, which bounds the total actual cost.

```
COUNTER(A, n)
1  A = 0
2  for j←1  to  n
3      INCREMENT(A)
```

```
INCREMENT(A)
1  i = 0
2  while i < A.length and A[i] == 1
3      A[i] = 0
4      i = i + 1
5  if i < A.length
6      A[i] = 1
```

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

application of accounting method -- convex hull

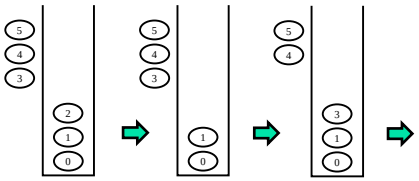
GRAHAM-SCAN(Q)
1 let p_0 be the point in Q with the minimum y-coordinate,
or the leftmost such point in case of a tie
2 let $\langle p_1, p_2, ..., p_m \rangle$ be the remaining points in Q, sorted by
polar angle in counterclockwise order around p_0
(if more than one point has the same angle, remove all
but the one that is farthest from p_0)
3 PUSH(p_0, S)
4 PUSH(p_1, S)
5 PUSH(p_2, S)
6 **for** $i \leftarrow 3$ **to** n
7 **while (** the consecutive segments formed by points
NEXT-TO-TOP(S), TOP(S), and p_i
make a nonleft turn**)**
8 POP(S)
9 PUSH(p_i, S)
10 **return** S

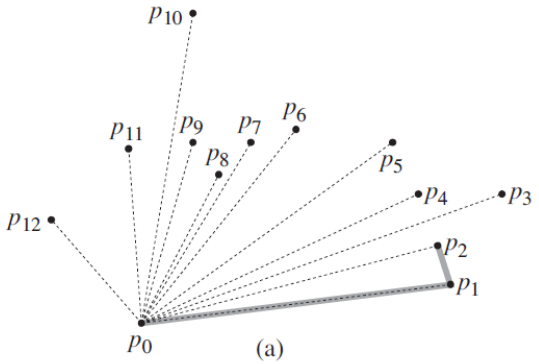
T(n) = ?

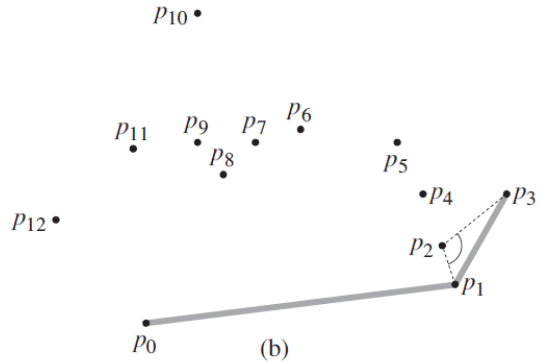
$\Theta(n)$

$O(n\lg n)$, using merge sort and the cross-product method to compare angles.

$O(1)$
$O(1)$
$O(1)$
$O(n\text{-}3)$
Accounting method: 每个顶点p_i都会且只会被PUSH一次，给予每个PUSH操作分摊消费2，其中1个用于实际PUSH，另1个作为“信用值”存储于该顶点上，POP操作的分摊消费为0，但POP能执行（用“信用值”进行支付）。







application of accounting method -- KMP

accounting: q（或 k） 加1时2个分摊消费（1个用于实际消费，1个是信用），减少时0个分摊消费（数q上有q个信用，最多减少到0，因此信用足够支付减少）

```
KMP-MATCHER(T, P)                                   Θ(n)
 1  n = T.length
 2  m = P.length
 3  π = COMPUTE-PREFIX-FUNCTION(P)
 4  q = 0
 5  for i = 1 to n
 6      while q > 0 and P[q + 1] ≠ T[i]
 7          q = π[q]    // cost 0
 8      if P[q + 1] == T[i]
 9          q = q + 1   // cost 2
10      if q == m
11          print "Pattern occurs with shift" i − m
12          q = π[q]
```

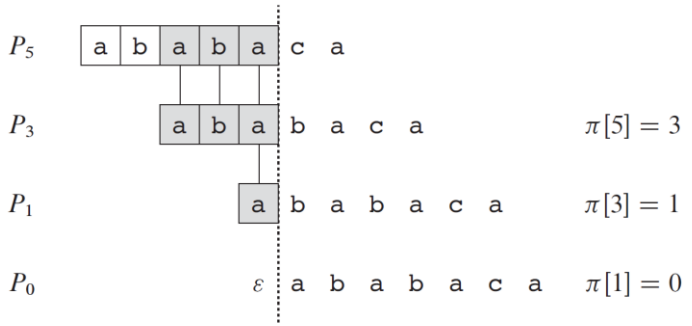
```
COMPUTE-PREFIX-FUNCTION(P)                          Θ(m)
 1  m = P.length
 2  let π[1 . . m] be a new array
 3  π[1] = 0
 4  k = 0
 5  for q = 2 to m
 6      while k > 0 and P[k + 1] ≠ P[q]
 7          k = π[k]
 8      if P[k + 1] == P[q]
 9          k = k + 1
10      π[q] = k
11  return π
```

prefix function:

$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$

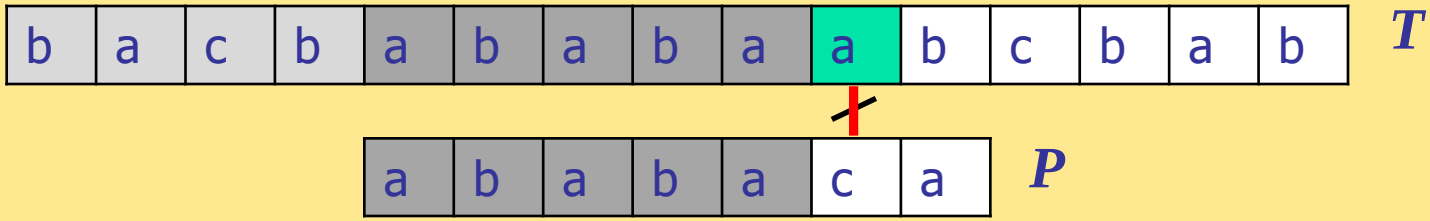
i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

(a)



(b)

KMP review

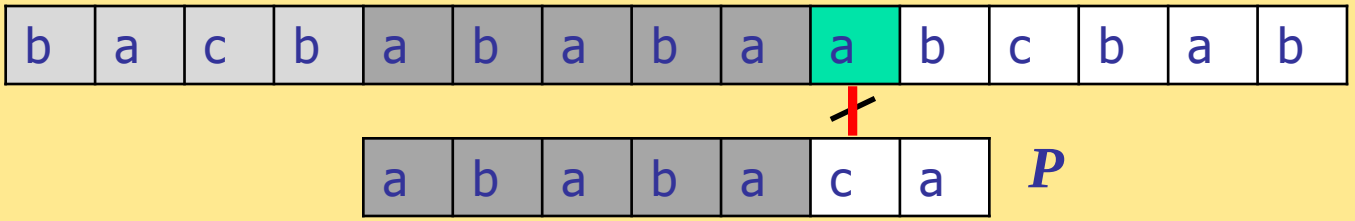


$P_5 \sqsupset T_{s+5}$, but, T[*s*+5+1] != *P*[5+1], ($q = 5$)

P 的前5个匹配，第6个不匹配，模版 *P* “漂移”到什么位置（然后继续匹配）？

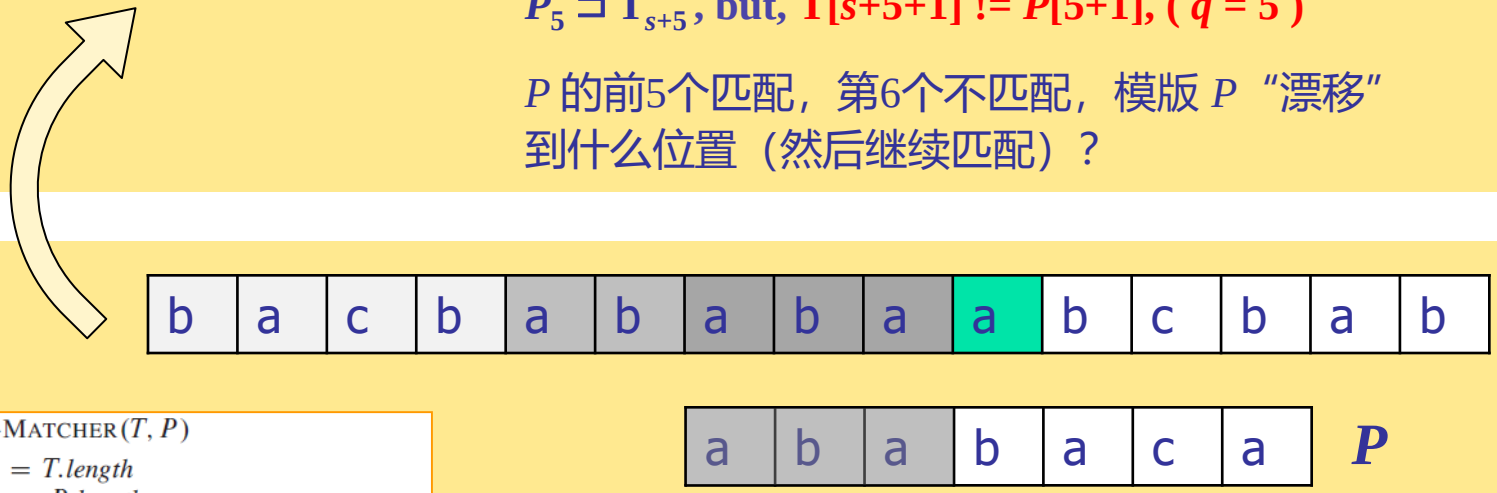
```
KMP-MATCHER(T, P)
 1  n = T.length
 2  m = P.length
 3  π = COMPUTE-PREFIX-FUNCTION(P)
 4  q = 0
 5  for i = 1 to n
 6      while q > 0 and P[q + 1] ≠ T[i]
 7          q = π[q]
 8      if P[q + 1] == T[i]
 9          q = q + 1
10      if q == m
11          print "Pattern occurs with shift" i − m
12          q = π[q]
```

KMP review



$P_5 \sqsupset T_{s+5}$, but, T[*s*+5+1] != *P*[5+1], ($q = 5$)

P 的前5个匹配，第6个不匹配，模版 *P* “漂移”到什么位置（然后继续匹配）？




```
KMP-MATCHER(T, P)
 1  n = T.length
 2  m = P.length
 3  π = COMPUTE-PREFIX-FUNCTION(P)
 4  q = 0
 5  for i = 1 to n
 6      while q > 0 and P[q + 1] ≠ T[i]
 7          q = π[q]
 8      if P[q + 1] == T[i]
 9          q = q + 1
10      if q == m
11          print "Pattern occurs with shift"
12          q = π[q]
```

P 的前缀 P_3 是已匹配的 P_5 的最大后缀，

$q = \pi[5] = 3$，即 P_5 的后缀且是 *P* 的最大前缀为 P_3 。即，P_3 是已扫描的子文本 T_{s+5} 的“最大”后缀（除已经处理过的 P_5 外），从此处开始继续判断 ***T[i]*** 是否与 ***P*[*q*+1]** 匹配）。

KMP review

如何求 $\pi[q]$？KMP-MATCHER，COMPUTE-PREFIX-FUNCTION，这两个函数的逻辑完全相同，第一个是找 T_x 最大后缀，第二个找 P_q 的最大后缀（两者都是 P 的前缀）。

```
KMP-MATCHER(T, P)
 1  n = T.length
 2  m = P.length
 3  π = COMPUTE-PREFIX-FUNCTION(P)
 4  q = 0
 5  for i = 1 to n
 6      while q > 0 and P[q + 1] ≠ T[i]
 7          q = π[q]
 8      if P[q + 1] == T[i]
 9          q = q + 1
10      if q == m
11          print "Pattern occurs with shift" i − m
12          q = π[q]
```

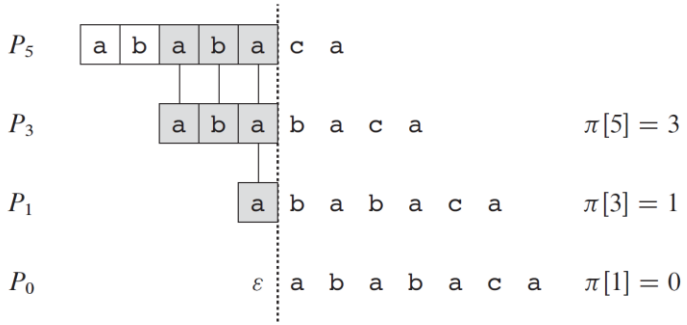
```
COMPUTE-PREFIX-FUNCTION(P)
 1  m = P.length
 2  let π[1 . . m] be a new array
 3  π[1] = 0
 4  k = 0
 5  for q = 2 to m
 6      while k > 0 and P[k + 1] ≠ P[q]
 7          k = π[k]
 8      if P[k + 1] == P[q]
 9          k = k + 1
10      π[q] = k
11  return π
```

prefix function:

$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

(a)



(b)

17.3 The potential method（势能法）

- The potential method of amortized analysis represents the prepaid work as "potential energy" (potential), which can be released to pay for future operations.
- We associate the potential with the data structure (DS) as a whole rather than with specific objects within the data structure.
- The potential method works as follows:
  - We will perform n operations, starting with an initial DS D_0.
  - c_i : the actual cost of the ith operation ($i = 1, 2, \ldots, n$).
  - D_i : the DS that results after applying the ith operation to DS D_{i-1}.
  - Φ : A potential function Φ maps each DS D_i to a real number $\Phi(D_i)$, which is the potential associated with DS D_i.
- The amortized cost $\hat{c}_i$ of the ith operation with respect to potential function Φ is defined by $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

17.3 The potential method（势能法）

- The potential method of amortized analysis represents the prepaid work as "potential energy" (potential), which can be released to pay for future operations.
- We associate the potential with the data structure (DS) as a whole rather than with specific objects within the data structure.
- The potential method works as follows:
  - We will perform n operations, starting with an initial DS D_0.
  - c_i : the actual cost of the ith operation ($i = 1, 2, \ldots, n$).
  - D_i : the DS that results after applying the ith operation to DS D_{i-1}.
  - Φ : A potential function Φ maps each DS D_i to a real number $\Phi(D_i)$, which is the potential associated with DS D_i.
- 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

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

17.3 The potential method

- 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

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

- We usually just define $\Phi(D_0)$ to be 0 and then show that $\Phi(D_i) \geq 0$ for all i .

- Different potential functions may yield different amortized costs.

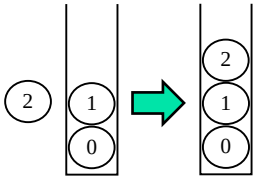
17.3 The potential method - Stack operations

- 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

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

- We **define the potential function Φ on a stack to be the number of objects** in the stack.
  - for the empty stack D_0 with which we start, we have $\Phi(D_0) = 0$
  - after the ith operation, for the stack D_i , $\Phi(D_i) \geq 0$?
- 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 of this **PUSH** operation is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$$



17.3 The potential method - Stack operations

- If the ith operation is **MULTIPOP(*S*, k)**, which causes $k' = \min(k, s)$ objects to be popped off the stack. The actual cost of the operation is k', and the potential difference is $\Phi(D_i) - \Phi(D_{i-1}) = -k'$
- The amortized cost of this MULTIPOP operation is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0$$

- Similarly, the amortized cost of an ordinary **POP** operation is 0.
- The worst-case cost of STACK is therefore O(n)?

```
STACK(S, n)
1  S = NULL
2  for i←1  to  n
3        One of ( PUSH(S, i), POP(S), MULIPOP(S, k) )  //  2  at most
```

17.3 The potential method - Incrementing a binary counter

- We define the potential Φ of the counter after the ith INCREMENT operation to be b_i , the number of 1s in the counter. (第 i 次操作后，势函数 $b_i = \Phi(D_i)$ 为计数器中 1 的个数)
- Suppose that the ith INCREMENT operation **resets** t_i bits (**from 1 to 0**). The actual cost of the operation is therefore at most, that is $c_i \le t_i + 1$, since in addition to resetting t_i bits, it sets at most one bit to 1.
  - If $b_i = 0$, then the ith operation resets all k bits (k位都是1，到最大数了，加1操作后越界限，变成 1000..00，最高位 1 舍去), and so $b_{i-1} = t_i = k$. (k位1全变为0)
  - If $b_i > 0$, then $b_i = b_{i-1} - t_i + 1$. (有效计数范围内，t_i 个1变为0 (while循环，即第2~4行)，1个0变为1 (加1算法的最后一行，即第6行))
  - In either case, $b_i \le b_{i-1} - t_i + 1$, and the potential difference is

$$\Phi(D_i) - \Phi(D_{i-1}) \le (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i$$

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \le (t_i + 1) + (1 - t_i) = 2$$

- The amortized cost is therefore 2.

```
COUNTER(A, n)
1 A = 0
2 for j←1 to n
3       INCREMENT(A)
```

```
INCREMENT(A)
1  i = 0
2  while i < A.length and A[i] == 1
3        A[i] = 0
4        i = i + 1
5  if i < A.length
6        A[i] = 1
```

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

17.3 The potential method - Incrementing a binary counter

- We define the potential Φ of the counter after the ith INCREMENT operation to be b_i , the number of 1s in the counter.
  第 i 次操作后，势函数 b_i 为计数器中 1 的个数
- The amortized cost is therefore

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \le (t_i + 1) + (1 - t_i) = 2$$

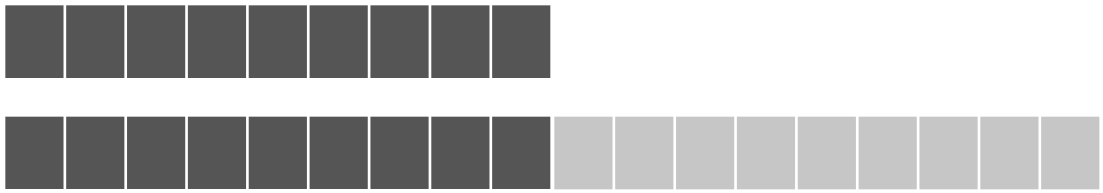
```
COUNTER(A, n)
1  A = 0
2  for j←1  to  n
3      INCREMENT(A)
```

```
INCREMENT(A)
1  i = 0
2  while i < A.length and A[i] == 1
3      A[i] = 0
4      i = i + 1
5  if i < A.length
6      A[i] = 1
```

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

- The worst-case cost of COUNTER is therefore O(n)?

17.4 Dynamic tables



```
void * malloc(size_t size);
```

c : malloc/free

c++: new/delete

- We might allocate space for a table, only to find out later that it is not enough. We must then reallocate the table with a larger size and copy all objects stored in the original table over into the new, larger table.
- Similarly, if many objects have been deleted from the table, it may be worthwhile to reallocate the table with a smaller size.
- We assume that the dynamic table supports the operations TABLE-INSERT and TABLE-DELETE.
  - TABLE-INSERT inserts into the table an item that occupies a single slot, that is, a space for one item.
  - Likewise, TABLE-DELETE removes an item from the table, thereby freeing a slot.

17.4 Dynamic tables

- The details of the data-structuring method used to organize the table are unimportant. It might be:

  a stack, or a heap, or a hash table, or an array

- load factor $\alpha(T)$ of a nonempty table T :

  $$\alpha(T) = T.num/T.size$$

- We assign an empty table (one with no items) size 0, and we define its load factor to be 1.

17.4.1 Table expansion

- We assume that storage for a table is allocated as an **array** of slots.
- Upon inserting an item into a full table, we can **expand** the table by allocating a new table with more slots than the old table had.
- A common heuristic allocates a new table with twice as many slots as the old one.

  If the only table operations are **insertions**, then the load factor of the table is always at least 1/2, and thus the amount of wasted space never exceeds half the total space in the table.

< *T.table* : a pointer to *T* >

```
TABLE-INSERT(T, x)
 1  if T.size == 0
 2       allocate T.table with 1 slot
 3       T.size = 1
 4  if T.num == T.size
 5       allocate new-table with 2 · T.size slots
 6       insert all items in T.table into new-table
 7       free T.table
 8       T.table = new-table
 9       T.size = 2 · T.size
10  insert x into T.table
11  T.num = T.num + 1
```

17.4.1 Table expansion

The running time of TABLE-EXPANSION ?

```
TABLE-EXPANSION(T, n)
1  T = NULL
2  for j←1  to  n
3      TABLE-INSERT(T, x)
```

```
TABLE-INSERT(T, x)
 1  if T.size == 0
 2        allocate T.table with 1 slot
 3        T.size = 1
 4  if T.num == T.size
 5        allocate new-table with 2 · T.size slots
 6        insert all items in T.table into new-table
 7        free T.table
 8        T.table = new-table
 9        T.size = 2 · T.size
10  insert x into T.table
11  T.num = T.num + 1
```

- What is the cost c_i of the ith operation?
  - If the current table has room for the new item (or the first operation), then $c_i = 1$?
  - If the current table is full, an expansion occurs, then $c_i = i$?
- The worst-case cost of an operation is $O(n)$?
- The upper bound of TABLE-EXPANSION is $O(n^2)$?
- Correct. Not tight ?

(1) aggregate analysis

The running time of TABLE-EXPANSION ?

```
TABLE-EXPANSION(T, n)
1  T = NULL
2  for j←1  to  n
3      TABLE-INSERT(T, x)
```

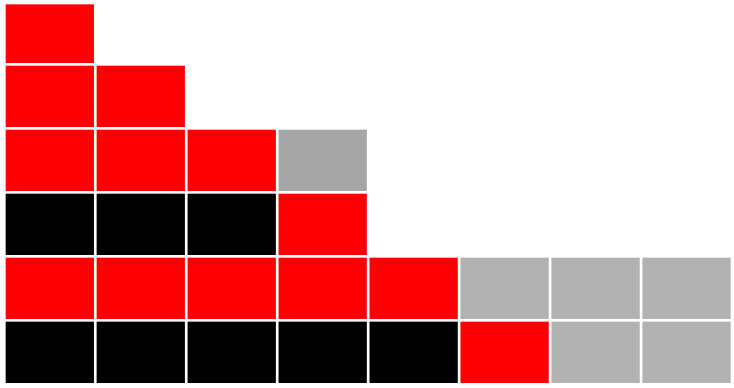
```
TABLE-INSERT(T, x)
 1  if T.size == 0
 2        allocate T.table with 1 slot
 3        T.size = 1
 4  if T.num == T.size
 5        allocate new-table with 2 · T.size slots
 6        insert all items in T.table into new-table
 7        free T.table
 8        T.table = new-table
 9        T.size = 2 · T.size
10  insert x into T.table
11  T.num = T.num + 1
```

What is the cost c_i of the ith operation?

$$c_i = \begin{cases} i, & \text{if } i\text{-}1 = 2^k, \\ 1, & \text{otherwise.} \end{cases}$$

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

No.	cost	T
1	1	
2	2	
3	3	
4	1	
5	5	
6	1	

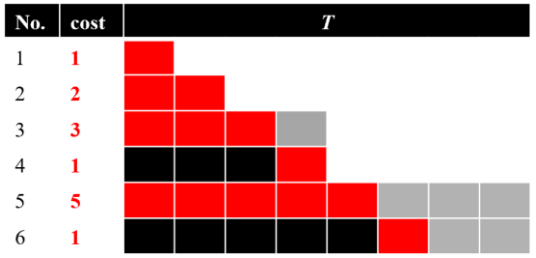


Red: 执行增加元素操作（次数）

Gray: 空槽 null slot

Black: 有元素的槽

(1) aggregate analysis



The running time of TABLE-EXPANSION?

```
TABLE-EXPANSION(T, n)
1  T = NULL
2  for j←1  to  n
3      TABLE-INSERT(T, x)
```

```
TABLE-INSERT(T, x)
 1  if T.size == 0
 2      allocate T.table with 1 slot
 3      T.size = 1
 4  if T.num == T.size
 5      allocate new-table with 2 · T.size slots
 6      insert all items in T.table into new-table
 7      free T.table
 8      T.table = new-table
 9      T.size = 2 · T.size
10  insert x into T.table
11  T.num = T.num + 1
```

What is the cost c_i of the ith operation?

插入每个元素所需的开销?

1	2	3	1	5	1	1	1	9	...
1	1	1	1	1	1	1	1	1	...
	1	2		4				8	...

$$c_i = \begin{cases} i\ , & \text{if } i\text{-}1=2^k, \\ 1\ , & \text{otherwise.} \end{cases}$$

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

(2) accounting method

The running time of TABLE-EXPANSION ?

```
TABLE-EXPANSION(T, n)
1  T = NULL
2  for j←1  to  n
3      TABLE-INSERT(T, x)
```

What is the cost c_i of the ith operation?

$$c_i = \begin{cases} i\ , & \text{if } i\text{-}1=2^k, \\ 1\ , & \text{otherwise.} \end{cases}$$

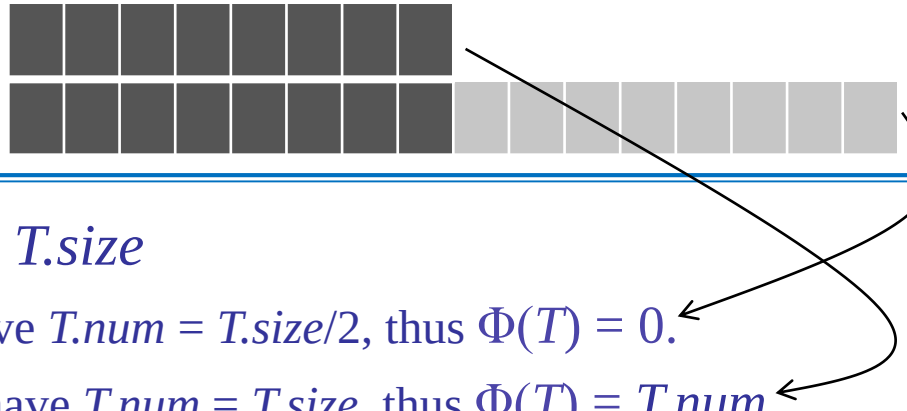
$$\begin{aligned} \sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n \end{aligned}$$

```
TABLE-INSERT(T, x)
 1  if T.size == 0
 2      allocate T.table with 1 slot
 3      T.size = 1
 4  if T.num == T.size
 5      allocate new-table with 2 · T.size slots
 6      insert all items in T.table into new-table
 7      free T.table
 8      T.table = new-table
 9      T.size = 2 · T.size
10  insert x into T.table
11  T.num = T.num + 1
```

We charge an amortized cost of 3 dollars for each insertion, 1 for actual insertion, 2 for credits.

No.	cost	T
1	1	2 — total is 3: 1 for cost, 2 left for credits.
2	2	1 2
3	3	0 1 2
4	1	0 1 2 2
5	5	-1 0 1 1 2
6	1	-1 0 1 1 2 2

(3) potential method



- potential function : $\Phi(T) = 2T.num - T.size$
  - Immediately after an expansion, we have $T.num = T.size/2$, thus $\Phi(T) = 0$.
  - Immediately before an expansion, we have $T.num = T.size$, thus $\Phi(T) = T.num$.
- If the ith TABLE-INSERT operation does not trigger an expansion, then we have $size_i = size_{i-1}$ and the amortized cost of the operation is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 1 + (2 \cdot num_i - size_i) - (2(num_i - 1) - size_i) \\ &= 3\end{aligned}$$

```
TABLE-EXPANSION(T, n)
1  T = NULL
2  for j←1  to  n
3      TABLE-INSERT(T, x)
```

No.	cost	T
1	1	
2	2	
3	3	
4	1	
5	5	
6	1	

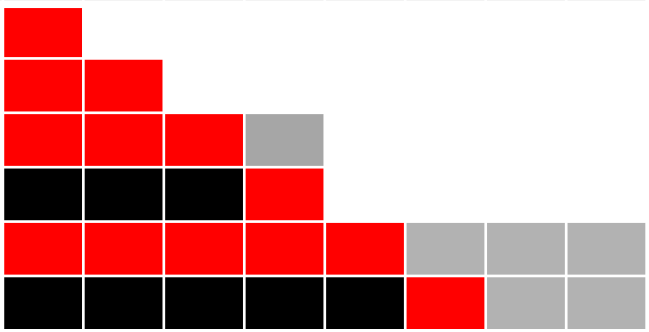
(3) potential method

- potential function : $\Phi(T) = 2T.num - T.size$
  - Immediately after an expansion, we have $T.num = T.size/2$, thus $\Phi(T) = 0$.
  - Immediately before an expansion, we have $T.num = T.size$, thus $\Phi(T) = T.num$.
- If the ith TABLE-INSERT operation trigger an expansion, $size_i = 2size_{i-1}$, and $size_{i-1} = num_{i-1} = num_i - 1$, thus

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\
&= num_i + (2 \cdot num_i - 2 \cdot (num_i - 1)) - (2(num_i - 1) - (num_i - 1)) \\
&= num_i + 2 - (num_i - 1) \\
&= 3
\end{aligned}$$

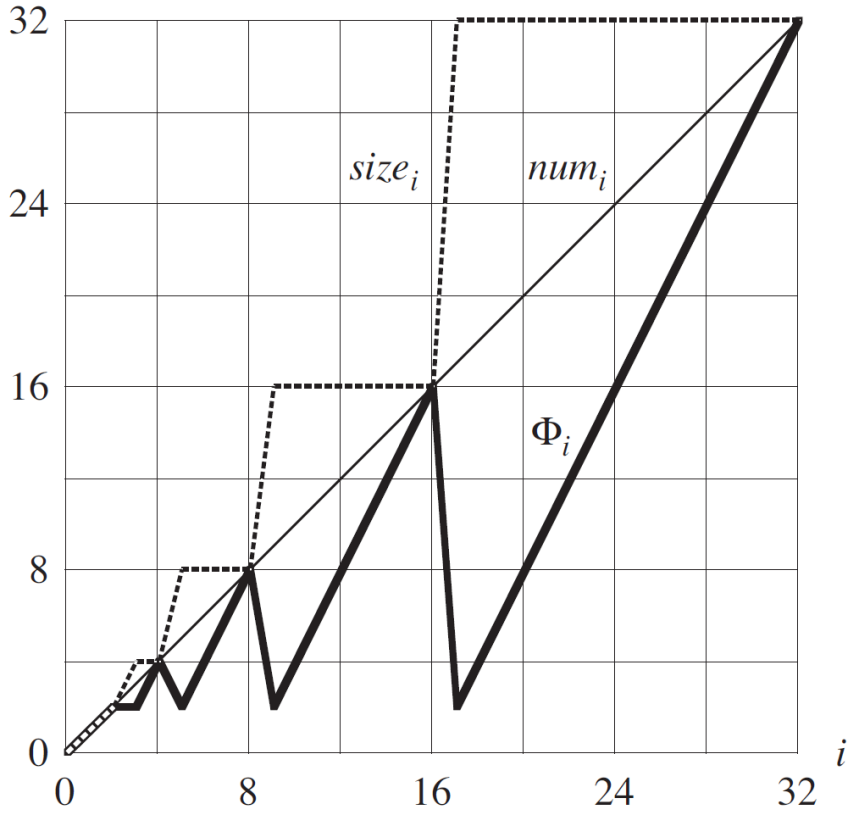
```
TABLE-EXPANSION(T, n)
1  T = NULL
2  for j←1  to  n
3      TABLE-INSERT(T, x)
```

No.	cost	T
1	1	
2	2	
3	3	
4	1	
5	5	
6	1	



(3) potential method

- potential function : $\Phi(T) = 2T.num - T.size$
  - Immediately after an expansion, we have $T.num = T.size/2$, thus $\Phi(T) = 0$.
  - Immediately before an expansion, we have $T.num = T.size$, thus $\Phi(T) = T.num$.
- For the ith TABLE-INSERT operation, the amortized cost is 3




```
TABLE-EXPANSION(T, n)
1  T = NULL
2  for j←1  to  n
3      TABLE-INSERT(T, x)
```

No.	cost	T
1	1	
2	2	
3	3	
4	1	
5	5	
6	1	

17.4.2 Table expansion and contraction(收缩)

自学

Conclusion and exercise-in-class

	Aggregate analysis	The accounting method	The potential method
Stack operations			
Incrementing a binary counter			
Dynamic Table expansion			
Dynamic Table contraction			
convex hull			
KMP			
……			

exercises

- 所有的课后习题。
- 再举几个例子（找凸包、KMP，我已经用过了，不能再用），说明如何用Amortized Analysis进行算法的复杂度分析。
- 把本书中所有用Amortized Analysis进行复杂度分析的算法找出来，并指出其分析方法与结果。

exercises

把本书中所有用Amortized Analysis进行复杂度分析是算法找出来，并指出其分析方法与结果。