算法设计与分析 第二部分

算法分析

本次课主要内容

- 排序问题
- 插入排序
- 合并排序
- 递归式
- 算法分析
- 平摊分析

排序问题

Input: sequence $\langle a_1, a_2, \ldots, a_n\rangle$ of numbers.

Output: permutation $\langle a'_1, a'_2, \ldots, a'_n\rangle$ Such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Example:

Input: 8 2 4 9 3 6

Output: 2 3 4 6 8 9

排序算法

- 插入排序
- 合并排序
- 冒泡排序
- 堆排序
- 快速排序
- 选择排序
- ……

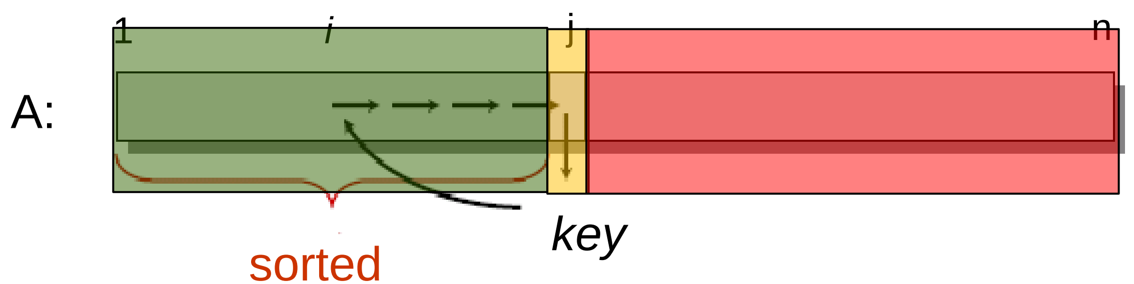
插入排序（INSERTION-SORT）

```
INSERTION-SORT (A, n)        ⊳ A[1 . . n]
    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
```

“pseudocode”

插入排序（INSERTION-SORT）

```
INSERTION-SORT (A, n)        ⊳ A[1 . . n]
    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
```



插入排序举例

8	2	4	9	3	6	
2	8	4	9	3	6	
2	4	8	9	3	6	
2	4	8	9	3	6	
2	3	4	8	9	6	
2	3	4	6	8	9	*done*

插入排序的分析

- 插入排序的时间开销
  - 与输入规模有关
  - 与输入序列的特性有关

插入排序的分析

- 最佳情况运行时间
  - 输入数组已经排好序
- 最坏情况运行时间
  - 问题要求最终按递增的顺序排列
  - 但输入数组按递减顺序排列

插入排序的分析

INSERTION-SORT (*A*, *n*)

```
for j ← 2 to n
    do key ← A[ j]
        ▷ Insert 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
```

插入排序的分析

- 为分析做的简化
  - 忽略每条语句的真实代价
  - 只考虑最高次项
  - 忽略最高次项的系数
- 插入排序的最坏情况时间复杂度
  - $\Theta(n^2)$

插入排序的C++示例代码

```
void InsertSort(int * Array,int Size)
{
        int j,t;
        for(int i = 1;i < Size;i++)
        {
                for( j = 0;j < i;j++)
                {
                        if(Array[j] > Array[i])
                                break;
                }
                t = Array[i];
                for(int k = i-1 ;k >= j;k--)
                {
                        Array[k+1] = Array[k];
                }
                Array[j] = t;
        }
}
```

插入排序示例

- 请对“3 23 25 8 1 23 16 15”应用插入排序算法进行排序，并给出排序过程

合并排序（Merge Sort）

MERGE-SORT A[1 . . n]

1. If n= 1, done.
2. Recursively sort A[1 . . .n/2.]and A[[n/2]+1 . . n] .
3. “Merge” the 2 sorted lists.

Key subroutine: MERGE

Merging two sorted arrays

20	12		20	12
13	11		13	11
7	9		7	9
2	1		2	1

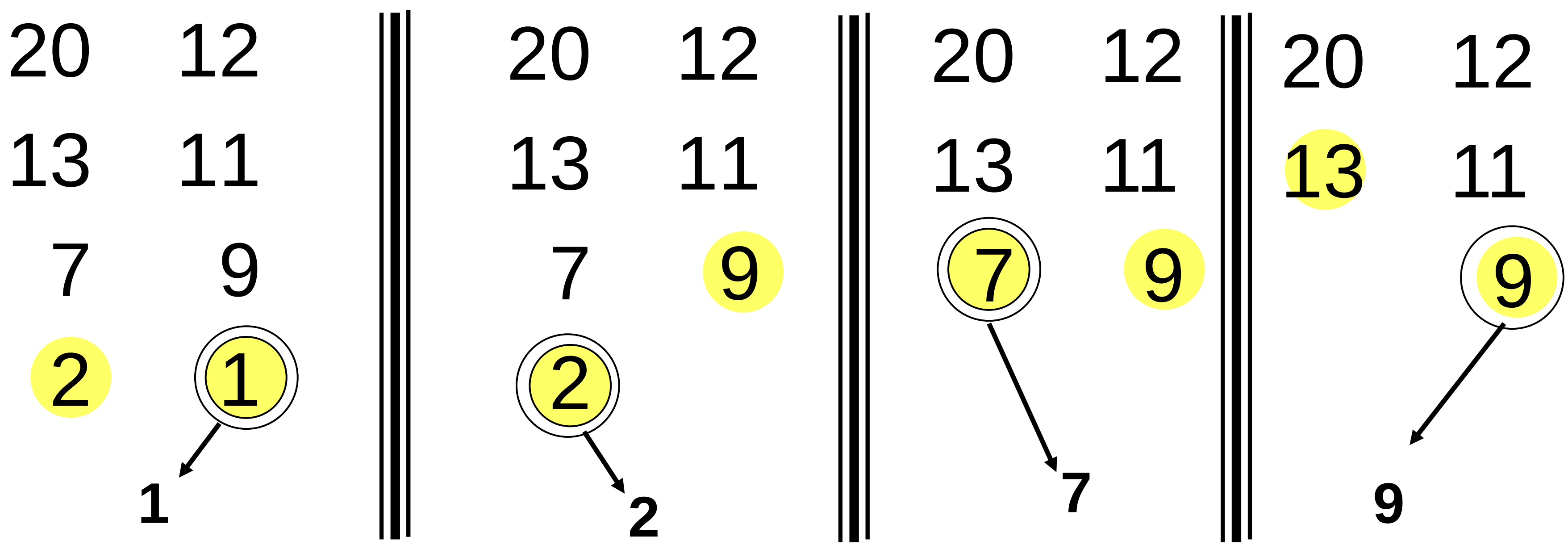
1

Merging two sorted arrays

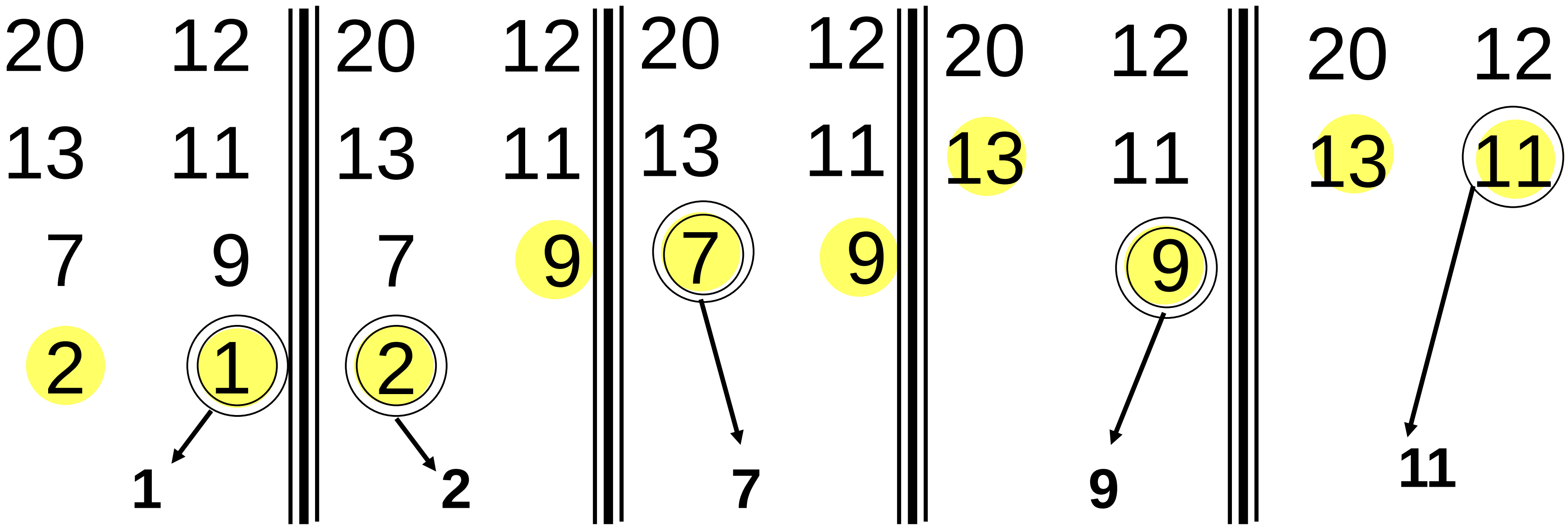
20	12		20	12
13	11		13	11
7	9		7	9
2	1		2	

1

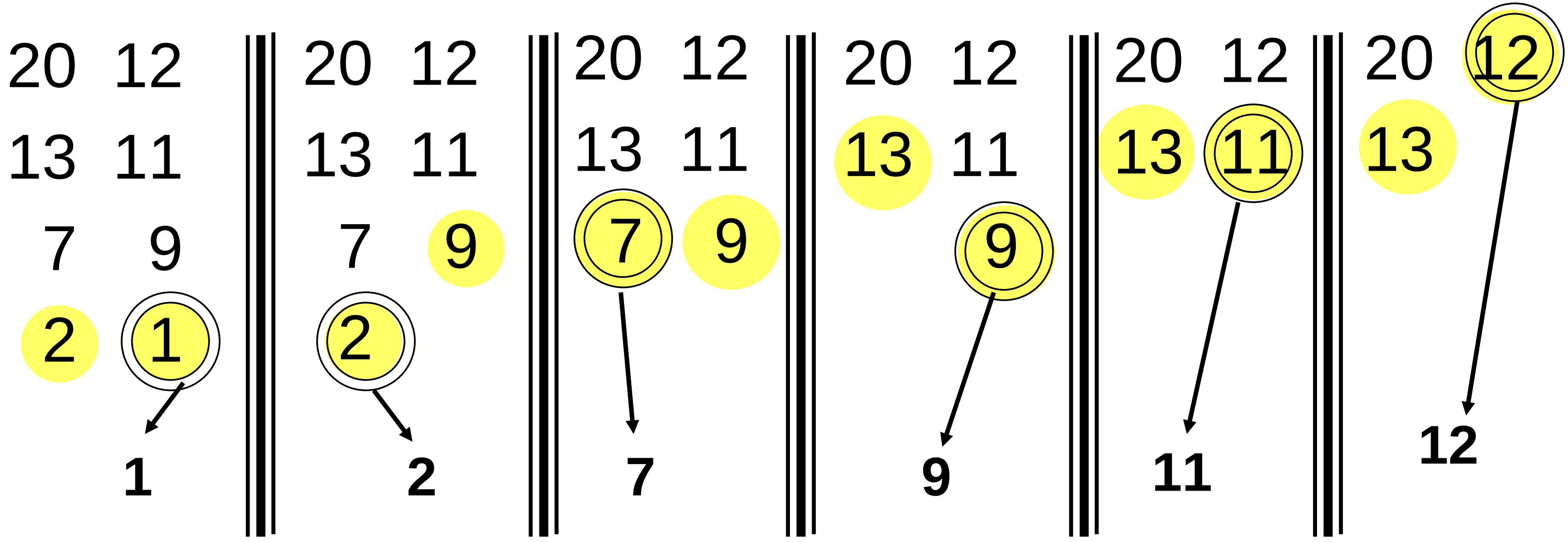
Merging two sorted arrays



Merging two sorted arrays



Merging two sorted arrays



Time = $\Theta(n)$ to merge a total of *n* elements (linear time).

合并排序分析

T(n)

Θ(1)	MERGE-SORTA[1 . . n] 1.If n= 1, done.
2T(n/2)	2.Recursively sort A[1 . . ⌈n/2⌉] and A[⌈n/2⌉+1 . . n] .
Θ(n)	3.“Merge”the 2sorted lists

Recursion tree

- Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.
- $\Theta(n \lg n)$
- $\Theta(n \lg n)$ grows more slowly than $\Theta(n^2)$
- Merge sort asymptotically beats insertion sort in the worst case
- In practice, merge sort beats insertion sort for n> 30 or so

平摊分析（Amortized Analysis）

平摊分析主要思路

- 平摊分析是一种算法分析的手法，其主要思路是：
  - 对若干条指令（通常O(n)条）整体进行考虑其时间复杂度（以获得更接近实际情况的时间复杂度）
  - 而不是逐一考虑执行每条指令所需的时间复杂度后再进行累加
- 利用平摊分析法来分析算法的最坏时间复杂度时，即使某些指令执行时具有较高的代价，但对算法总体而言，仍可求得较低的、更接近实际情况的时间复杂度
  - 注意：不是减小算法的时间复杂度，而是求得更准确的时间复杂度

平摊分析的常用方法

- 聚集方法（Aggregate method）
- 会计算法（Accounting method）
- 势能方法（Potential method）

聚集方法

- 平摊分析的最常用手法
  - 从全局来整体考虑时间复杂度
  - 把O(n)条指令的耗费（时间复杂度）分为几类
  - 分别计算O(n)条指令中每一类耗费的总和
  - 然后再把各类耗费总加起来

聚集方法 例：栈操作

- 栈操作：Push(x,S), Pop(S), 均只需要O(1)时间
- 引进一个新的栈操作Multi-Pop(S,k)：
  - 若S中元素个数≥ k, 则弹出k个元素,
  - 若小于k, 则弹出全部元素。
- 最坏情况下，Multi-Pop(S,k)执行一次需要O(n)时间

聚集方法 例：栈操作

- 考虑上述三种操作组成的序列，设共有n条指令
- 执行n条指令的最坏情况是否为$O(n^2)$？
- ——$O(n^2)$是上界，但不紧确。如何分析？

聚集方法 例：栈操作

- 把Pop, Multi-Pop的耗费分为两类：
  - 出栈操作需要的耗费
  - 非出栈操作（判断栈中的元素个数）需要的耗费
- 一条Multi-Pop指令的非出栈操作耗费为O(1)
- （Pop指令没有非出栈操作的耗费）
- 故n条指令的第2类（非出栈操作）的耗费为O(n)

聚集方法 例：栈操作

- 出栈操作需要耗费的分析：
  - ➢ 一个元素在出栈之前必定首先要进栈，
  - ➢ 故出栈元素的个数一定小于等于进栈元素的个数。
- ∵共有n条指令，∴进栈操作指令Push(x,S)最多n条。
- Push(x,S)指令每次只能使一个元素进栈，故进栈元素最多为n。
- 由于出栈元素个数 ≤ 进栈元素个数，因此全部Pop及Multi-Pop指令中取元素出栈的操作最多为n；故出栈操作需要的耗费也O(n)。
- 由此得总耗费也为O(n)。

聚集方法 例

- 一个程序，从0开始用增1方法逐个生成2进制数，放在k个二进制位中。
  - e.g. 0000, 0001, 0010, 0011, ……, 1111。
- 问：从0开始生成n个数 ($0 < n \le 2^k-1$)，在最坏情况下需要多少次位操作（从0变为1以及从1变为0）？

聚集方法 例

- 从当前数生成下一个数的算法：
  - 从最低位开始，如果当前检测位为1，则将其改为0，
  - 直到碰见第一个0时，将0改为1，一个数的生成工作完成。
- 简单分析
  - 在最坏情况下，生成1个数时k位都要改， 即需要O(k)时间，故生成n个数需O(n*k)时间。
  - ——作为上界是对的，但不紧确。如何分析才能得到紧确的上界？

聚集方法 例

- 平摊分析：把n条指令的耗费分为k类：
  - 第1类：最低位上的耗费（0变1或1变0的次数）；
  - 第2类：次低位上的耗费；
  - 第3类：右数第三位上的耗费；
  - ……
  - 第k类：最高位上的耗费。

聚集方法 例

- 各类耗费的具体量值：
  - 第1类：每生成一个新数最低位均需改变一次， 生成n个数的所需耗费为n=$\lfloor n/2^0 \rfloor$。
  - 第2类：每生成两个新数次低位均需改变一次，生成n个数的所需耗费为$\lfloor n/2 \rfloor$=$\lfloor n/2^1 \rfloor$。
  - 第3类：每生成四个新数右数第三位均需改变一次，生成n个数的所需耗费为$\lfloor n/4 \rfloor$=$\lfloor n/2^2 \rfloor$。
  - ……
  - 第k类：每生成2^{k-1}个新数最高(第k)位均需改变一次， 生成n个数的所需耗费为$\lfloor n/2^{k-1} \rfloor$。

聚集方法 例

- 由此得总耗费为 $\sum_{i=0}^{k-1} \lfloor n/2^i \rfloor \le \sum_{i=0}^{k-1} n/2^i = n^* \sum_{i=0}^{k-1} 1/2^i \le 2n$。
- 故在最坏情况下最多需要的位操作不超过2n。

会计算法

- 在计算A指令的耗费时，将B指令的全部或部分耗费提前计算，一并算作A指令的耗费。
- 于是计算B指令耗费时此部分就无需再算。

会计算法 例：栈操作

- Push、Pop、Multi-Pop的例子
- 栈操作的最主要工作是：进栈和出栈。
- 由于一个元素的出栈必然在进栈之后，故计算进栈指令耗费时可将进栈元素的出栈耗费一并算入。
    - 这样，执行任一元素的出栈指令时就不需要再计算出栈耗费了。（因为费用已提前计算过。）

会计算法 例：栈操作

- 给每个操作以下述平摊代价：
  - Push一次: 平摊代价为2（包括元素的进栈与该元素出栈的耗费）；
  - Pop一次：平摊代价为0（任一元素出栈的耗费已被Push指令提前计算）；
  - Multi-Pop一次：平摊代价为1（该代价为执行Multi-Pop时，对栈中元素个数进行判断所需的耗费，任一元素的出栈耗费已被指令Push提前计算了。）
- ∵一条指令的最大平摊代价为2，故n条指令的代价不超过2n。

会计算法 例：增1法生成二进制数

- 增1法逐个生成二进制数
- 从0开始用增1方法逐个生成2进制数，放在k个二进制位中。
  - e.g. 0000, 0001, 0010, 0011, ……, 1111。
- 问：从0开始生成n个数 (0 < n ≤ 2k-1)，在最坏情况下需要多少次位操作（从0变为1以及从1变为0）？

会计算法 例：增1法生成二进制数

- 会计方法求解：
  - 开始时是从0….0开始，任何一位在变为1之前必然是0。
  - 将0置成1时，把代价计为2，（把将来从1变为0所需要的1个代价先计算掉）
  - 这样，当某位从1变为0时，由于所需要的1个代价事先已经计算过了，故该代价无需再计算。

会计算法 例：增1法生成二进制数

- 会计方法求解：
  - 由于每生成1个数，只会碰到一个0，（从最低位开始，如果当前检测位为1，则将其改为0，碰见第一个0时，将0改为1，一个数的生成工作完成。）
  - 生成n个数，由0改为1 的操作次数为n，故代价为2n；
  - 而由1改为0的代价为0，故总的代价为2n。

势能方法

- 设C_i为第i个操作的实际代价，D_0为处理对象的数据结构的初始状态
- D_i为第i个操作施加于数据结构D_{i-1}之上后数据结构的状态，引入势函数Φ，$\Phi(D_i)$是与D_i相关的势。

势能方法

- 定义第i个操作的平摊代价为：

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

- （即实际代价加上势的变化），于是有：

$$\sum_{i=1}^{n} \hat{C}i = \sum_{i=1}^{n} Ci + (\Phi(D_n) - \Phi(D_0))$$

势能方法

如果我们总能保证$\Phi(D_i) \ge \Phi(D_0)$ $(1 \le i \le n)$，

即$\Phi(D_i) - \Phi(D_0) \ge 0$。

那么就有$\sum_{i=1}^{n} Ci \le \sum_{i=1}^{n} \hat{C}i$，即$\sum_{i=1}^{n} \hat{C}i$给出了$\sum_{i=1}^{n} Ci$的一个上界。

计算$\sum_{i=1}^{n} Ci$时，如果由于不确定性，使得计算有困难，

则可以通过计算$\sum_{i=1}^{n} \hat{C}i$，得到$\sum_{i=1}^{n} Ci$的一个上界。

势能方法 例 栈操作

- Push, Pop, Multi-Pop
- 作为操作对象的数据结构是栈，初始时为空栈D_0。
- 第i次操作后的栈为D_i。

势能方法 例 栈操作

- 引入势函数，
- $\Phi(D_i)$为第i次操作后栈中的元素的个数。
- 于是有$\Phi(D_0)=0$，$\Phi(D_i) \geq 0$ $(i=1,2,\ldots n)$。
- 设$\Phi(D_{i-1})=s$ $(s \geq 0)$。

势能方法 例 栈操作

若第 i 个操作为 Push，则有$\Phi(D_i)-\Phi(D_{i-1})=(s+1)-s=1$。

由于 Push 的实际代价 C_i 为 1，

故 Push 的平摊代价为：

$\hat{C}_i = C_i+(\Phi(D_i)-\Phi(D_{i-1}))=1+1=2$。

势能方法 例 栈操作

若第 i 个操作为 Multi-Pop，

并设该操作弹出了 k’个元素，

则此时有 $C_i = k' + 1$，$\Phi(D_i) = \Phi(D_{i-1}) - k'$，

即$\Phi(D_i) - \Phi(D_{i-1}) = -k'$，

于是，$\hat{C}_i = C_i + (\Phi(D_i) - \Phi(D_{i-1})) = k' + 1 - k' = 1$，

即当第 i 个操作为 Multi-Pop 时，平摊代价 $\hat{C}_i$ 为 1。

当第 i 个操作为 Pop 时，可证明平摊代价 $\hat{C}_i$ 为 0。

因此有：对任何 i，$\hat{C}_i \le 2$，故 $\sum_{i=1}^{n} Ci \le \sum_{i=1}^{n} \hat{C}i \le 2n$。

势能方法 例：二进制数的生成

- 数据结构为k个二进制位。
- 势函数$\Phi(D_i)$为第i次操作后，k个二进制位中1的个数。
- 于是，$\Phi(D_0)=0$ (设初始化为0)，
- 则有$\Phi(D_i) \geq \Phi(D_0)$ $(i=1,2,\ldots n)$。

势能方法 例：二进制数的生成

设第 i 次操作使右数 t_i 个连续的 1 变为 0($t_i \geq 0$),

右数第一个 0 变为 1,

则 $C_i=t_i+1$（但在全 1 情况下, $C_i=t_i$）。

另外，$\Phi(D_i)= \Phi(D_{i-1})-t_i+1$，即有$\Phi(D_i)-\Phi(D_{i-1}) = -t_i+1$。

（但在全 1 情况下, $\Phi(D_i)-\Phi(D_{i-1}) =-t_i$）。于是,

$\hat{C}_i =C_i+(\Phi(D_i)-\Phi(D_{i-1}))=(t_i+1)+(-t_i+1)=2$。

（但在全 1 情况下，$\hat{C}_i =t_i-t_i =0$）

故当 $n<2^k$ 时，有$\sum_{i=1}^{n} Ci \leq \sum_{i=1}^{n} \hat{C}i =2n$。

而当 $n\geq 2^k$ 时，有$\sum_{i=1}^{n} Ci \leq \sum_{i=1}^{n} \hat{C}i < 2n$。

势能方法 例：二进制数的生成

为了计算初始状态不为 0 的耗费，可将求和等式变形：

由 $\sum_{i=1}^{n}\hat{C}i = \sum_{i=1}^{n} Ci + \Phi(D_n) - \Phi(D_0)$，

可得 $\sum_{i=1}^{n} Ci = \sum_{i=1}^{n}\hat{C}i - \Phi(D_n) + \Phi(D_0)$。

由于 $\hat{C}_i \leq 2$**，故有** $\sum_{i=1}^{n} Ci \leq 2n - \Phi(D_n) + \Phi(D_0)$

势能方法 例：二进制数的生成

记$\Phi(D_0)=b_0$为初始时k位二进计数器中1的个数，

记$\Phi(D_n)=b_n$为执行n次操作后

k位二进计数器中1的个数。于是有

$\sum_{i=1}^{n} Ci \leq 2n-b_n+b_0$。若执行次数n<k，则有$\sum_{i=1}^{n} Ci \leq 2n+k$,

因$b_0 \leq k$即初始时二进计数器中1的个数不超过k，

而$b_n \geq 0$。

若执行次数$n \geq k$，则有$b_0 \leq k \leq n$，

于是可得$\sum_{i=1}^{n} Ci \leq 3n-b_n \leq 3n$，仍然是线性的。

平摊分析小结

- 平摊分析的概念
- 平摊分析的常用方法
- 平摊分析的应用