Hw 9

Page245: 16.2-2，16.2-4

Page249: 16.3-3，16.3-4

Page260：17.1-2

Page262：17.2-3

Page264：17.3-3

16.2

Page245: 16.2-2，16.2-4

16. 2-2 设计动态规划算法求解 0-1 背包问题，要求运行时间为 $O(nW)$，n 为商品数量，W 是小偷能放进背包的最大商品总重量。

答:

假定$w_1, w_2, \ldots w_n$和W都是正整数

w_i：第i个物品的重量

p_i：第i个物品价值

$A(j, Y)$：在总重量不超过Y的前提下，前j种物品的总价格所能达到的最高值

$A(j, Y)$的递推关系式为：

$(1) A(0, Y) = 0$

(2)如果$w_j > Y$, 则$A(j, Y) = A(j-1, Y)$

(3)如果$w_j \leq Y$, 则$A(j, Y) = max\{A(j-1, Y), p_j + A(j-1, Y-w_j)\}$

通过计算$A(n, W)$即得到最终结果。

参考#149号同学

```
0-1-KNAPSACK(n,W,p,w)
    let K[0..n][0..W] be a new table    #最优解值
    let T[1..n][1..W] be a new table    #最优解
    for i = 1 to n
        k[i][0] = 0
    for i = 0 to W
        K[0][i] = 0
    for i = 1 to n
        for j = 1 to W
            if j < w[i]
                K[i][j] = k[i-1][j]
                T[i][j] = 0
            else
                if K[i-1][j] > K[i-1][j-w[i]] + p[i]
                    K[i][j] = K[i-1][j]
                    T[i][j] = 0
                else
                    K[i][j] = K[i-1][j-w[i]] + p[i]
                    T[i][j] = 1
```

16. 2-4 Gekko 教授一直梦想用直排轮滑的方式横穿北达科他州。他计划沿 U. S. 2 号高速公路横穿，这条高速公路从明尼苏达州东部边境的大福克斯市到靠近蒙大拿州西部边境的威利斯顿市。教授计划带两公升水，在喝光水之前能滑行 m 英里(由于北达科他州地势相对平坦，教授无需担心在上坡路段喝水速度比平地或下坡路段快)。教授从大福克斯市出发时带整整两公升水。他携带的北达科他州官方地图显示了 U. S. 2 号公路上所有可以补充水的地点，以及这些地点间的距离。

教授的目标是最小化横穿途中补充水的次数。设计一个高效的方法，以帮助教授确定应该在哪些地点补充水。证明你的策略会生成最优解，分析其运行时间。

采用贪心的策略

```
marked_stop = n*[0]
Min_Water_Stop(n):
    distance_current = 0
    number_of_water_stops = 0
    for stops = 1 to n:
        if distance_current + distance_to_next_stop > 2 mile:
            # 记录停止位置,保存在mark_stop矩阵中
            stop()
            number_of_water_stops++
            distance_current = 0
        else:
            distance_current = distance_current + distance_to_next_stop
    return marked_stop
```

Step 1 of 3

Algorithm to find the shortest possible root using greedy approach

The optimal strategy of the given scenario should be a Greedy one.

Suppose there is k refilling locations beyond the start that are within m miles of the start. The greedy solution chooses the k^{th} location as its first stop. No station beyond the k^{th} works as a first stop.

Since Professor Gekko would run out of water first. The professor will try to cover the maximum distance before his water runs out.

Step 2 of 3

At Grand Forks, the professor has all the possible routes in front of him.He now first eliminates the stops which are more than *m* miles away from him.

From the rest of the stops, he will choose the stop that traverses the maximum distance that is less than *m*.

Doing this for all the further stops, the Professor will be able to stop at minimum stops.

The algorithm for this is given below:

The algorithm below gives the minimum number of water stops that the professorcan go through.

MIN-WATER-STOP(*N*)

1 //Initialize the distance of current stops to 0

distance_current = 0;

2 //for loop to trace all the stops

For stops 1 **to** *n*

3 //condition to check whether the distance is more than 2 miles.

if (distance_current + distance_to_next_stop> 2 mile)

4 //Mark the current stop, means the stop should be there

mark_current_stop();

5 //Make increment in the number of stops

number_of_water_stops++;

6 //Make the distance_current equal to 0 so that in other iteration the distance

```
//remain 0
distance_current = 0
7 else
8 //add the distance of next stop in the current distance.
distance_current = distance_current + distance_to_next_stop;
9 //after calculating the stops, return the number of marked stops.
Return marked_stop
```

Step 3 of 3

In the algorithm given above, the water stop is made by professor if the distance between the upcoming refill and last refill is more than 2 miles. If he can easily reach to next stop, there is no need to make any stop there.

So, this leads to the property of sub-optimality, this property makes *m* number of minimal stops.

Suppose that there exists another solution with requirement of *m*–1 stops. It is not possible to cover the distance when a stop is omitted because the water will not be there. It is because the distance will become more than 2 miles.

If the number of available stops is represented by *n*, the complexity of the algorithm can be calculated as below:

If *n*=1, then the loop would be running only one time making the time:

$$T_1 = 1$$

If *n*=2, then there would be 2 repetitions of the loop and the time taken would be:

$$T_2 = 1 + 1$$
$$= 2$$

...

...

For the n repetitions of the loop there would be *n* calculations and the time taken would be equal to

$$T_n = 1 + 1 + 1 + \ldots + 1$$
$$= n$$

Thus the running time of the algorithm for *n* stops would be $O(n)$.

16.3

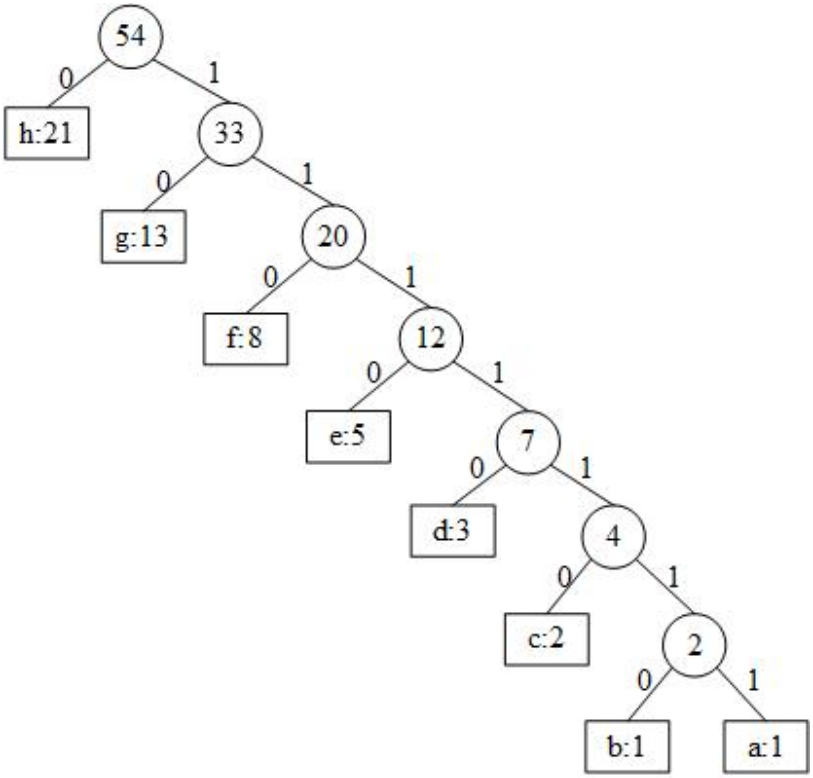
Page249: 16.3-3，16.3-4

16.3-3 如下所示，8 个字符对应的出现频率是斐波那契数列的前 8 个数，此频率集合的赫夫曼编码是怎样的？

a：1　b：1　c：2　d：3　e：5　f：8　g：13　h：21

你能否推广你的结论，求频率集为前 n 个斐波那契数的最优前缀码？

赫夫曼树：



赫夫曼编码：

$$
\begin{aligned}
h &: 0 \\
g &: 10 \\
f &: 110 \\
e &: 1110 \\
d &: 11110 \\
c &: 111110 \\
b &: 1111110 \\
a &: 1111111
\end{aligned}
$$

推广，在字符频率集对应前n个斐波那契数的集合中，第i个字符的赫夫曼编码如下：

$if\ i = 1$ 则第i个字符有$n - i$位且全为1

$if\ i > 1$

(1)其有$n - i + 1$位

(2)其最后一位为0

(3)其前$n - i$位为1

即：

$$
\begin{aligned}
Fib(n) &: 0 \\
Fib(n-1) &: 10 \\
Fib(n-2) &: 110 \\
&\cdots \\
Fib(2) &: 1^{n-2}0 \\
Fib(1) &: 1^{n-1}
\end{aligned}
$$

16.3-4　证明：编码树的总代价还可以表示为所有内部结点的两个孩子结点的联合频率之和。

Step 1 of 5

Let tree be a full binary tree with n leaves. Apply Induction Hypothesis on the number of leaves in T. When n = 2 (the case n = 1 is trivially true). There are two leaves x and y with the same parent z, then the cost of T is

$$B(T) = f(x)d_T(x)+f[y]d_T[y]$$
$$=f[x]+f[x] \text{ since } d_T(x)=d_T(y)=1$$
$$= f[\text{child1 of z}]+f[\text{child2 of z}]$$

Step 2 of 5

Thus, the statement of theorem is true. Now suppose n > 2 and also suppose that theorem is true for trees on n − 1 leaves. Let C_1 and C_2 are two sibling leaves in T such that they have the same parent P. Letting T be the tree obtained by deleting C_1 and C_2, we know by induction that

$$B(T) = S \text{ leaves i' in T' } f[1']dT[1']$$
$$= S \text{ internal nodes i' in T' } f[\text{child1 of i'}]+f[\text{child2 of i'}]$$

Step 3 of 5

Using this information calculate the cost of T,

$$B(T) = S\text{leaves i' in T' } f[1']d_T[1']$$
$$=S_1 \text{ not} = C_1$$
$$= C_2 f[1]d_T(1)+f[C_1]d_T(C_1)\text{-}1+f$$

Step 4 of 5

$$[C_2]d_T((C_2)\text{-}1)+f[C_1]+f[C_2]$$
$$= S\text{leaves 1in T' } f[1]d_T(1)+f[C_1]+f[C_2]$$
$$= S \text{ internal nodes i' in T' } f[\text{child1 of i'}]+f$$

Step 5 of 5

$$[\text{child2 of i'}]+ f[C_1]+f[C_2]$$
$$= S \text{ internal nodes i in T } f[\text{child1 of i}]+f[\text{child1 of i}]$$

Thus the statement is true and this completes the proof.

17.1

Page260：17.1-2

17.1-2 证明：如果 k 位计数器的例子中允许 DECREMENT 操作，那么 n 个操作的运行时间可能达到 $\Theta(nk)$。

下界：只有Incre，或只有Decre操作：Decre操作是Incre逆过程，所以只有Decre时的摊还代价也是O(n)。

上界：最坏情况，k bits初始全0，**交替执行Decre、Incre**。执行Decre负向溢出，k bits由0置1，$\Theta(k)$；执行Incre又会导致正向溢出，k bits由1复0，$\Theta(k)$。这样n个操作后，总代价$\Theta(nk)$。

综上，$\Theta(nk)$。

17.2

17.2-3 假定我们不仅对计数器进行增 1 操作，还会进行置 0 操作(即将所有位复位)。设检测或修改一个位的时间为 $\Theta(1)$，说明如何用一个位数组来实现计数器，使得对一个初值为 0 的计数器执行一个由任意 n 个 INCREMENT 和 RESET 操作组成的序列花费时间 $O(n)$。(提示：维护一个指针一直指向最高位的 1。)

首先，作两点说明：

1. “维护一个指针指向最高位‘1’”的目的：

   Reset要将计数器中所有的“1”复为“0”，有了这个指针后，Reset每次不用检测所有位，只要检测到指针所指的位置即可，因为指针后全“0”，无需复位。

2. “检测或修改1 bit的时间为$\Theta(1)$”：

   - Reset算法：从低位到指针进行检测，遇“0”只是检测，遇“1”还要复位。
   - Incre算法：分两种情况：（1）无溢出：从低位到高位，遇“1”复位，直至遇到第一个“0”，将其置“1”后停止。在此期间，都是检测并修改。（2）计数器全是“1”时溢出：将“1”全复为“0”结束。在此期间，都是检测并复位。
   - 检测并修改的代价：考虑到有的同学将这个代价记为1，有的记为2，其实1还是2分析是一样的，我们这里记检测并修改的代价为a（$1 \leq a \leq 2$）。

法1. Accounting method

书上只有Incre时，accounting的设计为：将1 bit置位的代价记为2，置位时消耗1个代价，复位时取走剩下的1个代价。

增加Reset后，**不能采用原设计**，因为这里检测也会产生代价。**新accounting设计**：检测并置位的代价记为a+1+a，当前检测并置位消耗a个代价，剩余1+a个代价储存在该bit，由后续操作消费——只检测不修改取走1个代价，检测并复位时取走a个代价。

新accounting的正确性：若操作过程中，出现bit负债，则accounting设计错误。只有对同一bit连续的检测或检测并复位，会导致bit负债。而（1）由于存在指向最高位“1”的指针，故不会出现对同一位连续的Reset，即不会出现对同一位连续的检测。（2）Reset会将Incre终止的“0”降到低位，故不会出现对同一位连续的检测、检测并复位，或连续的检测并复位。所以，新accounting不会导致bit负债，正确。

这样设计accounting，使得**只有Incre增加摊还代价，Reset不增加摊还代价。**又Incre每次最多置1个“1”，n次操作后摊还代价$= O((2a+1)n) = O(n)$。

开始的说明里给了**Incre和Reset**的算法，但最好以**伪码**形式给出（by #84）：

我们引入一个新的字段$max[A]$来保存A的高阶1的索引。开始$max[A]$被设为-1，因为A的低阶位在索引0处，并且初始在A中没有1。$max[A]$的值是递增或重置的，我们使用这个值来限制必须查看多少A才能重置它。通过这种方式控制重置代价，我们可以将其限制在一个可以由早期 INCREMENT 的信贷覆盖的金额内。

```
1  INCREMENT(A)
2      i = 0
3      while i<A.length and A[i]==1
4          A[i] = 0
5          i = i+1
6      if i < A.length
7          A[i] = 1
8      /*增加的代码段*/
9      if  i > max[A]
10         max[A] = i
11      else
12         max[A] = -1
13 /*RESET函数*/
14 RESET(A)
15     for i=0 to max[A]
16         A[i] = 0;
17     max[A] = -1
```

我们假设翻转一点需要花费1美元。另外，我们假设花费1美元来更新$max[A]$。通过 INCREMENT 设置和重置比特将如原始计数器一样运行：支付1美元设置一位为1，1将被放置在被设置为1的位上作为信贷，每个1上的积分将用来支付递增过程中重置位。

此外，我们将使用1美元来支付更新max，如果max增加，我们将在新的高阶1上增加1美元的信用。(如果max不增加，我们可以浪费那1美元——就不需要了)。由于复位操作位只在$max[A]$的位置，并且由于每一位到那里必须在高阶1达到$max[A]$之前的某个时间变成高阶1，复位看到的每一位都有1美元的信用。因此，通过复位对A位的归零可以完全由存储在位上的信用支付。我们只需要1美元来重置max。每次 INCREMENT 收费4美元，每次重置收费1美元，因此n个 INCREMENT 和 RESET 操作的序列花费$O(n)$时间。

法2. 直接分析

考虑k_i次incre后作一次reset：

（1）k_i次连续incre的代价$= O(k_i)$

（2）一次reset的代价：k_i次连续incre后，计数器的值$= k_i$时reset代价最大，换成二进制，最高位"1"最高出现在（从低往高）第$\lceil \log_2 k_i \rceil + 1$位，故reset的代价为$O(\log k_i)$。

设n次操作中依次有$k_1, .. k_l$次连续incre，则n次操作总代价$= O(\Sigma k_i) + O(\Sigma \log k_i)$，因为$\Sigma k_i \le n$，$\Sigma \log k_i \le \Sigma k_i \le n$，故总代价$= O(n)$。

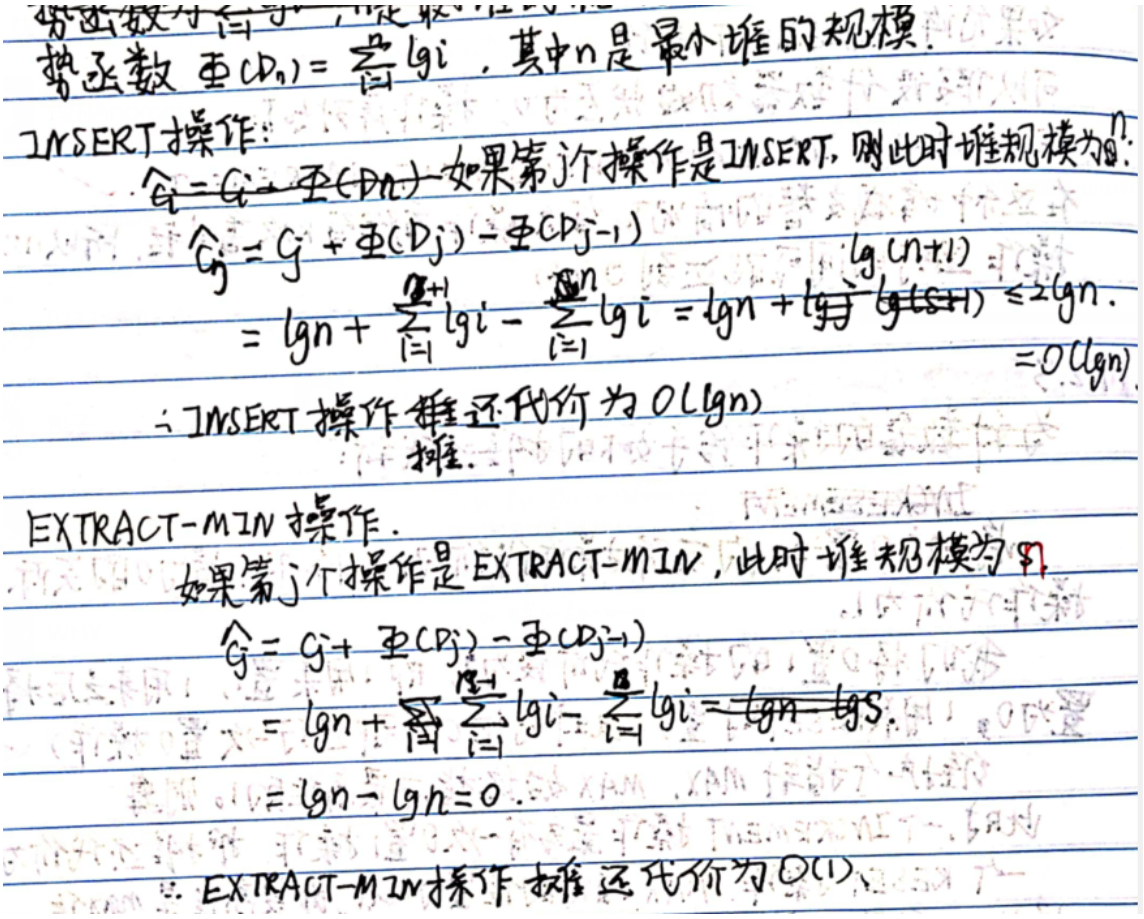
17.3

Page264：17.3-3

17.3-3 考虑一个包含 n 个元素的普通二叉最小堆数据结构，它支持 INSERT 和 EXTRACT-MIN 操作，最坏情况时间均为 $O(\lg n)$。给出一个势函数 Φ，使得 INSERT 操作的摊还代价为 $O(\lg n)$，而 EXTRACT-MIN 操作的摊还代价为 $O(1)$，证明它是正确的。

思路：摊还代价$\hat{c}_i =$ 实际代价$c_i + \Phi(D_i) - \Phi(D_{i-1})$，注意到Ext $\hat{c}$减到了$O(1)$量级，而Ins $\hat{c}$ 仍是$O(\lg n)$量级，故$\Delta\Phi$一定也是$O(\lg n)$量级。

法1. 将势函数定义为$\lg i$的累和



法1并不严谨，但它简单直观。

法2. 法1的严谨版

(1) 设实际代价c的系数为k，给出potential func

Let D_i be the heap after the ith operation, and let D_i consist of n_i elements. Also, let k be a constant such that each INSERT or EXTRACT-MIN operation takes at most $k \ln n$ time, where $n = \max(n_{i-1}, n_i)$. (We don't want to worry about taking the log of 0, and at least one of n_{i-1} and n_i is at least 1. We'll see later why we use the natural log.)

Define

$$\Phi(D_i) = \begin{cases} 0 & \text{if } n_i = 0\,, \\ kn_i \ln n_i & \text{if } n_i > 0\,. \end{cases}$$

This function exhibits the characteristics we like in a potential function: if we start with an empty heap, then $\Phi(D_0) = 0$, and we always maintain that $\Phi(D_i) \geq 0$.

这里设$c = k \ln n$，取$\ln()$而不是其他$\log()$函数是必要的，因为要推出摊还代价，需要$\ln()$函数的一个性质:

Before proving that we achieve the desired amortized times, we show that if $n \geq 2$, then $n \ln \frac{n}{n-1} \leq 2$. We have

$$\begin{aligned}
n \ln \frac{n}{n-1} &= n \ln \left(1 + \frac{1}{n-1}\right) \\
&= \ln \left(1 + \frac{1}{n-1}\right)^n \\
&\leq \ln \left(e^{\frac{1}{n-1}}\right)^n && \text{(since } 1 + x \leq e^x \text{ for all real } x\text{)} \\
&= \ln e^{\frac{n}{n-1}} \\
&= \frac{n}{n-1} \\
&\leq 2 ,
\end{aligned}$$

assuming that $n \geq 2$. (The equation $\ln e^{\frac{n}{n-1}} = \frac{n}{n-1}$ is why we use the natural log.)

(2) 推导摊还代价

先推Insert的摊还代价$O(\lg n)$:

要分类讨论，insert to an empty heap时，

If the ith operation is an INSERT, then $n_i = n_{i-1} + 1$. If the ith operation inserts into an empty heap, then $n_i = 1$, $n_{i-1} = 0$, and the amortized cost is

$$\begin{aligned}
\widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&\leq k \ln 1 + k \cdot 1 \ln 1 - 0 \\
&= 0 .
\end{aligned}$$

insert to a non-empty heap时，

If the ith operation inserts into a nonempty heap, then $n_i = n_{i-1} + 1$, and the amortized cost is

$$\begin{aligned}
\widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&\leq k \ln n_i + k n_i \ln n_i - k n_{i-1} \ln n_{i-1} \\
&= k \ln n_i + k n_i \ln n_i - k(n_i - 1) \ln(n_i - 1) \\
&= k \ln n_i + k n_i \ln n_i - k n_i \ln(n_i - 1) + k \ln(n_i - 1) \\
&< 2k \ln n_i + k n_i \ln \frac{n_i}{n_i - 1} \\
&\leq 2k \ln n_i + 2k \\
&= O(\lg n_i) .
\end{aligned}$$

下面推导Extract_min的摊还代价$O(1)$:

也要分类讨论，extract heap中唯一元素时，

If the ith operation is an EXTRACT-MIN, then $n_i = n_{i-1} - 1$. If the ith operation extracts the one and only heap item, then $n_i = 0$, $n_{i-1} = 1$, and the amortized cost

is

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\le k \ln 1 + 0 - k \cdot 1 \ln 1 \\ &= 0 .\end{aligned}$$

heap有多个元素时，

If the ith operation extracts from a heap with more than 1 item, then $n_i = n_{i-1} - 1$ and $n_{i-1} \ge 2$, and the amortized cost is

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\le k \ln n_{i-1} + k n_i \ln n_i - k n_{i-1} \ln n_{i-1} \\ &= k \ln n_{i-1} + k(n_{i-1} - 1) \ln(n_{i-1} - 1) - k n_{i-1} \ln n_{i-1} \\ &= k \ln n_{i-1} + k n_{i-1} \ln(n_{i-1} - 1) - k \ln(n_{i-1} - 1) - k n_{i-1} \ln n_{i-1} \\ &= k \ln \frac{n_{i-1}}{n_{i-1} - 1} + k n_{i-1} \ln \frac{n_{i-1} - 1}{n_{i-1}} \\ &< k \ln \frac{n_{i-1}}{n_{i-1} - 1} + k n_{i-1} \ln 1 \\ &= k \ln \frac{n_{i-1}}{n_{i-1} - 1} \\ &\le k \ln 2 \qquad (\text{since } n_{i-1} \ge 2) \\ &= O(1) .\end{aligned}$$

法3. 在法2势函数的基础上，定义一个更方便理解和推导的势函数

A slightly different potential function—which may be easier to work with—is as follows. For each node x in the heap, let $d_i(x)$ be the depth of x in D_i. Define

$$\begin{aligned}\Phi(D_i) &= \sum_{x \in D_i} k(d_i(x) + 1) \\ &= k\left(n_i + \sum_{x \in D_i} d_i(x)\right) ,\end{aligned}$$

where k is defined as before.

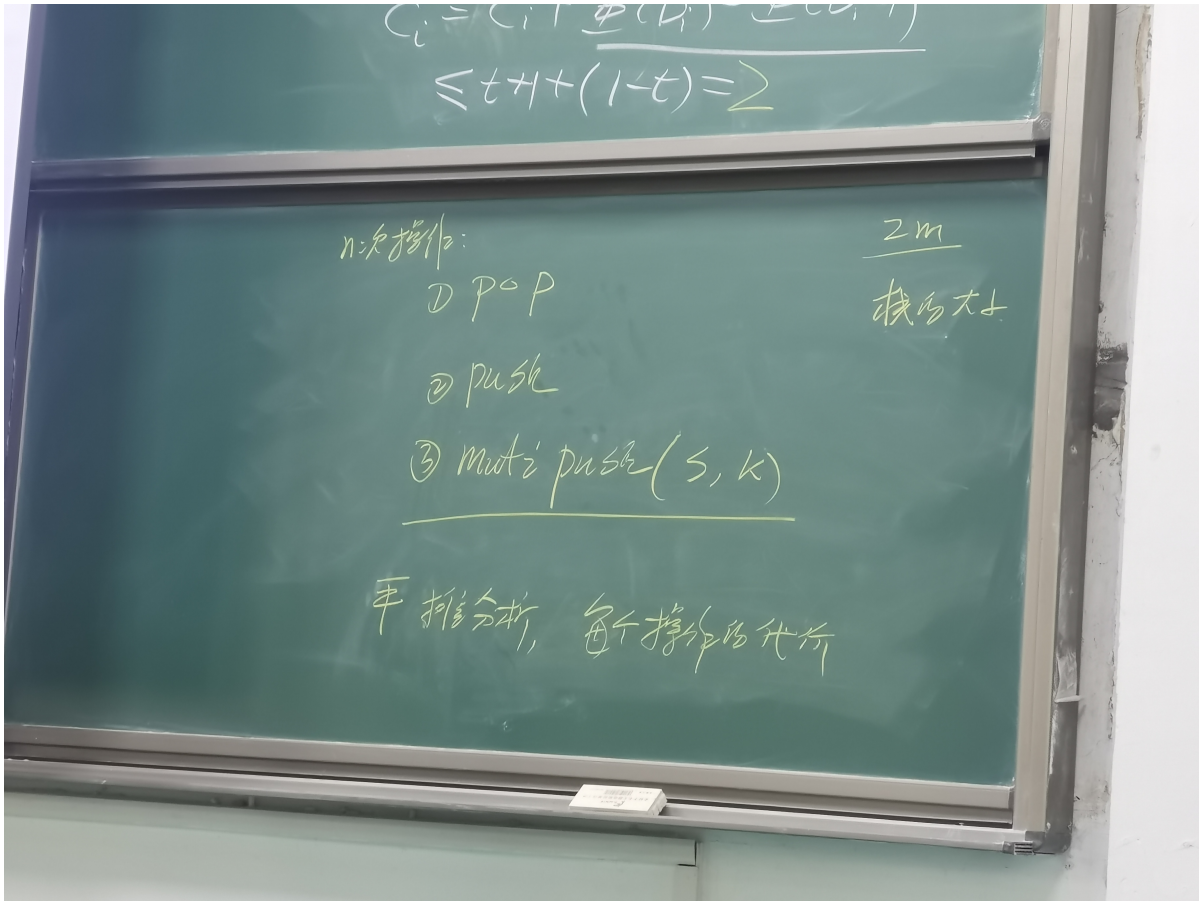
Initially, the heap has no items, which means that the sum is over an empty set, and so $\Phi(D_0) = 0$. We always have $\Phi(D_i) \ge 0$, as required.

Observe that after an INSERT, the sum changes only by an amount equal to the depth of the new last node of the heap, which is $\lfloor \lg n_i \rfloor$. Thus, the change in potential due to an INSERT is $k(1 + \lfloor \lg n_i \rfloor)$, and so the amortized cost is $O(\lg n_i) + O(\lg n_i) = O(\lg n_i) = O(\lg n)$.

After an EXTRACT-MIN, the sum changes by the negative of the depth of the old last node in the heap, and so the potential *decreases* by $k(1 + \lfloor \lg n_{i-1} \rfloor)$. The amortized cost is at most $k \lg n_{i-1} - k(1 + \lfloor \lg n_{i-1} \rfloor) = O(1)$.

第三次随测

问：假设栈大小为2m，考虑栈上操作pop，push，multipush作摊还分析，每个操作的代价。



为什么做摊还分析?

单个操作的平均代价不好求，最坏代价又不能很好反映该操作实际使用中的代价，如multipush(k个元素)，最坏代价O(2m)，其实际代价一般没那么大，故联合其数据结构上的其他操作做摊还分析，求n个多种操作的总代价，然后摊到单个操作上作为其摊还代价。

方法一：记账法（最简单的）

出现一个元素记下代价2，其中一个代价支付入栈操作，另一个代价支付出栈操作。

若最后栈空，则n个操作最多出现n-1个元素，即开始时一次multipush(n-1个元素)，后续(n-1)次pop将栈清空，记账代价为O(2(n-1))。

若最后栈不空，这时的总代价一定小于等于在栈空基础上增加2m次push操作将栈填满，故摊还总代价为O(2(n-1)+2m)，一个操作的摊还代价O((2(n-1)+2m)/n) = O((n+m)/n)。

方法二：聚合分析（参考119号同学）

一次multipush代价最多O(2m)，最坏情况下假设执行k次pop，n-k次multipush，则有：

$$-k+(n-k)2m=2m$$
$$2m(n-1)=(2m+1)k$$
$$k=\frac{2m(n-1)}{2m+1}$$
$$n-k=\frac{2mn+n-(2mn-2m)}{2m+1}=\frac{2m+n}{2m+1}$$

最坏情况时间：

$$kO(1)+(n-k)O(2m) = \frac{2m(n-1)}{2m+1}+2m*\frac{2m+n}{2m+1}$$
$$= O(m+n)$$

故摊还代价为$O(\frac{m+n}{n})=O(\frac{m}{n})$

大部分用聚合分析的同学：

（1）假设开始一次multipush，代价O(2m)，后续（n-1）次pop和push，代价O(n)，摊还代价O((2m+n)/n)，这样是不合适的，multipush就一次太少了。

（2）一次multipush(2m个元素)+2m次pop构成一轮，这是片面的，n次操作共n/(2m+1)轮，总代价O(((2m+2m)*n)/(2m+1))，但这样的前提是m<<n，故摊还代价为O(4mn/(2m+1)n)=O(2)

方法三：势函数法（目前还未解决）

假设$\Phi(D_i)=$栈D_i状态下的元素个数，故其与书上例题中势函数定义相同。故符合势函数要求

$$\begin{cases}\Phi(D_0)=0\\ \Phi(D_i)\$\geqslant 0\end{cases}$$

其中pop，push的摊还代价仍是0和2。

对于multipush(k个元素)：

$$\hat{C}_i=C_i+\Phi(D_i)-\Phi(D_{i-1})=2C_i\leq 4m$$

$$\text{其中，}C_i=min(k,2m-\Phi(D_{i-1}))\text{，}\Delta=C_i$$

$$\text{摊还代价}\frac{\sum\hat{C}_i}{n}\leq\frac{4mn}{n}=O(m)$$

这样做太大了，说明势函数这样定义不合适。