0、改变一下 RadixSort 的操作，得到排序算法 HabitSort，按照我们可能的习惯，首先用 Counting Sort 排序最高位，然后按照最高位的取值将数据划分成不同的组，再递归地用 HabitSort 排序各个组内元素，继而得到最终的排序。这样修改后的 HabitSort 算法的复杂性还是 $O(d(n+k))$吗？

解：排序最高位的代价是 O(n+k)，最高位最多有 k 种不同的取值。因此，需要将所有数据分组到 k 组中，每组的最高位相同，分组代价为 O(n)，将分组代价计入排序最高位后，排序最高位代价仍为 O(n+k)。

随后，对每组分别根据次高位进行排序，第 i 组的排序代为 O(n_i+k)，其中 n_i 为第 i 组中元素数量，排序 k 组的次高位，代价为$\sum_{i=1}^{k} O(n_i + k) = O\left(\sum_{i=1}^{k}(n_i + k)\right) = O(n + k^2)$。根据前两位对数据分组，分组代价 O($n$)，将分组代价计入排序最高位后，排序次高位代价仍为$O(n + k^2)$。

以此类推，排序第 3 高位时最多有 k^2 个分组，排序代价为$O(n + k^3)$；

排序第 j 高位时最多有 k^{j-1} 各分组，排序代价为$O(n + k^j)$；

排序最低位时最多有 k^{d-1} 各分组，排序代价为$O(n + k^d)$。

当然，想要让排序次高位到最低位时都有上述数量的分组，需要输入中有足够多的不同数字，这也对 n 的大小有了要求。

当$n = \theta(k^{d-0.5})$时，可以使得各个位的排序过程中产生上述数量的分组，而且排序最低位时每个分组中有$k^{0.5}$个元素（而非常数）。此时，排序所有位的代价为:$T(n) = O\left(\sum_{i=1}^{d}\left(n + k^d\right)\right) = O\left(dn + \frac{k^{d+1}-1}{k-1}\right) = O(dn + k^{d+1})$。

当$k = \Omega(\sqrt[d]{d})$时，$T(n) \neq O(d(n + k))$。

1、有n个大小不同的杯子和与之匹配的 n 个杯盖，你可以尝试一个杯子和一个杯盖是否匹配，尝试结果有三种：（1）杯子太大；（2）匹配成功；（3）杯盖太大．请设计一个分治算法完成所有杯子和杯盖的匹配，算法的时间复杂性用匹配尝试的次数来衡量。

解：（1）叙述算法设计思路. (请叙述以下方面)

边界条件：当只有一个杯子和一个杯盖时，匹配杯子和杯盖

Divide：从杯子集合中随机选择一个杯子 x，将 x 与所有杯盖进行匹配，把结果为杯子太大的杯盖放入 G1，把结果为杯盖太大的杯盖放入 G2，找到和 x 匹配成功的杯盖记为 y。将 y 与 x 以外所有杯子进行匹配，把结果为杯盖太大的杯子放入 B1，把匹配结果为杯子太大的杯盖放入 B2。

Conquer：递归地对 B1 和 G1、B2 和 G2 进行匹配.

Merge：返回 B1 和 G1、B2 和 G2 的匹配结果以及(x, y).

（2）写出算法伪代码.

Match(B, G)

Input: 杯子集合 B，杯盖集合 G, |B|=|G|=n, B 中每个杯子都只和 G 中一个杯盖匹配成功

Output: {(x, y)| x∈B, y∈G, x 与 y 匹配成功}

```
1.  if |B|=1
2.      return B×G;
3.  初始化 B1, B2, G1, G2, M=∅; y=NIL;
4.  随机选择杯子集合 B 中的一个元素 x;
5.  for ∀g ∈G
6.      if g 和 x 的匹配结果为“杯子太大”
```

7. G1=G1∪{g};
8. else if g 和 x 的匹配结果为“杯盖太大”
9. G2=G2∪{g};
10. else
11. y=g;
12. M=M∪{(x, y)};
13. for ∀b ∈B/{x}
14. if y 和 b 的匹配结果为“杯盖太大”
15. B1=B1∪{b};
16. else if g 和 x 的匹配结果为“杯子太大”
17. B2=B2∪{b};
18. if B1≠ ∅
19. M=M∪Match(B1, G1);
20. if B2≠ ∅
21. M=M∪Match(B2, G2);
22. return M;

（3）分析算法的时间复杂性.

记算法时间复杂性 T(n). 将 x 与所有杯盖匹配代价为 $O(n)$; 将 y 与 x 以外所有杯盖匹配的代价为 $O(n)$; 综上, 划分代价为 $O(n)$. Conquer 代价为 T(|B1|)+T(n-1-|B1|), 其中|B1|为 0、1、…、n-1 的概率都是 1/n. Combine（Merge）代价不计（算法的时间复杂性用匹配尝试的次数来衡量）. 总体代价与快速排序相同, 最坏情况为 $O(n^2)$; 期望为 O($n$log$n$).

2、对于两个二维数据元素$\boldsymbol{p}=(\boldsymbol{x_p},\boldsymbol{y_p})$和$\boldsymbol{q}=(\boldsymbol{x_q},\boldsymbol{y_q})$，如果(1) $\boldsymbol{x_p}\geq\boldsymbol{x_q}$, $\boldsymbol{y_p}>\boldsymbol{y_q}$ 或者(2) $\boldsymbol{x_p}>\boldsymbol{x_q}$, $\boldsymbol{y_p}\geq\boldsymbol{y_q}$，则$\boldsymbol{p}$支配$\boldsymbol{q}$，记为$\boldsymbol{p}\rightarrow\boldsymbol{q}$。二维数据集合$\boldsymbol{D}$的$\boldsymbol{Skyline}$定义如下：$\boldsymbol{SKL(D)}=\{\boldsymbol{p}|\boldsymbol{p}\in\boldsymbol{D},\nexists\boldsymbol{q}\in\boldsymbol{D},\boldsymbol{q}\rightarrow\boldsymbol{p}\}$。本部分内容计算二维数据集合的$\boldsymbol{Skyline}$。设计基于分治的二维数据 Skyline 求解算法。

（1）叙述算法设计思路. (请叙述以下方面)

边界条件: 数据集D中只有一个点p，则$SKL(D)=\{p\}$;

Divide:

1. 按二维键值(x, y)使用线性时间算法计算D的中位数$p(x_m,y_m)$，两个点比较大小时首先比较x值，若x值相等时比较y值（注: $|D|$为偶数时求第$|D|/2$ 个数作为中位数）；用p将D划分为两个大小最多相差 1 的子集合D_L和D_R;

Conquer: 递归地在D_L和D_R中计算 $SKL(D_L)$和$SKL(D_R)$;

Merge:

1. 扫描$SKL(D_R)$，若$SKL(D_R)$中存在 x 值等于x_m的点，计算$y_1=max\{y|(x,y)\in SKL(D_R),x=x_m\}$，否则$y_1=-\infty$；若$SKL(D_R)$中存在 x 值大于x_m的点，计算$y_2=max\{y|(x,y)\in SKL(D_R),x>x_m\}$，否则$y_2=-\infty$;
2. 初始化$SKL(D)\leftarrow SKL(D_R)$，对于$SKL(D_L)$中任一点$p(x,y)$:

Case 1: $p.x<x_m$，如果$p.y>y_1$且$p.y>y_2$，$SKL(D)=SKL(D)\cup\{p\}$;

Case 2: $p.x=x_m$，如果$p.y=y_1$且$p.y>y_2$，$SKL(D)=SKL(D)\cup\{p\}$;

（注：$SKL(D_L)$中的点 x 值都不大于x_m，因此只需要讨论$p.x<x_m$和$p.x=x_m$两种情况。）

（2）写出算法伪代码.

$SKL(D)$

1. if $|D| < 3$
2. 比较 D 中每个点对，并返回 Skyline 集合;
3. $p \leftarrow$D 中第$\lfloor |D|/2 \rfloor$的点（点与点比较时首先比较 x 值, x 值相同时比较 y 值）;
4. $D_L,\ D_R,\ S \leftarrow \emptyset$;
5. for $\forall q \in D$
6. if $q < p$
7. $D_L = D_L \cup \{q\}$;
8. if $q > p$
9. $D_R = D_R \cup \{q\}$;
10. if $|D_L| < \lfloor |D|/2 \rfloor$
11. for $j = 1$ to $\lfloor |D|/2 \rfloor - |D_L|$
12. $D_L = D_L \cup \{(p.x, p.y)\}$;
13. if $|D_R| < \lceil |D|/2 \rceil$
14. for $j = 1$ to $\lceil |D|/2 \rceil - |D_R|$
15. $D_R = D_R \cup \{(p.x, p.y)\}$;
16. $S \leftarrow SKL(D_R)$;
17. $L \leftarrow SKL(D_L)$;
18. $y_1,\ y_2 = -\infty$;
19. for $\forall q \in S$
20. if $q.x = p.x$ and $q.y > y_1$
21. $y_1 = q.y$;
22. else if $q.x > p.x$ and $q.y > y_2$
23. $y_2 = q.y$;
24. for $\forall q \in L$
25. if $q.x = p.x$ and $q.y = y_1$ and $q.y > y_2$
26. $S = S \cup \{q\}$;
27. if $q.x < p.x$ and $q.y > y_1$ and $q.y > y_2$
28. $S = S \cup \{q\}$;
29. return S;

（3）分析算法的时间复杂性.

边界条件处理代价为 $O(1)$;

Divide 代价为 $O(n)$;

Merge 代价中计算y_1和y_2的代价 O($|SKL(D_R)|$)，初始化$SKL(D) \leftarrow SKL(D_R)$的代价 $O(|SKL(D_R)|)$，扫描$SKL(D_L)$中每个点并更新$SKL(D)$的代价为 $O(|SKL(D_L)|)$，因此 Merge 代价为 $O(n)$;

算法代价的递归方程为 T(n)=2T(n/2)+$O(n)$，T(n)=$O(n\log n)$。

4、$X[0:n-1]$和$Y[0:n-1]$为两个数组，每个数组中的n个均已经排好序，试设计一个$O(\log n)$的分治算法，找出 X 和 Y 中$2n$个数的中位数。

（1）叙述算法设计思路.

需找到 X 和 Y 合并后排序第 n 和第 n+1 的元素. 将 X 划分为$X_L = X[0:\lfloor n/2 \rfloor - 1]$

和$X_R = X[\lfloor n/2 \rfloor : n-1]$, 将 Y 划分为$Y_L = Y[0:\lceil n/2 \rceil - 1]$和$Y_R = Y[\lceil n/2 \rceil : n-1]$. 这样$X_L$和$Y_L$中共有 n 个元素, X_R和Y_R中共有 n 个元素. 与此同时, X_L和Y_R中元素数量相等, X_R和Y_L中元素数量相等. 比较$X[\lfloor n/2 \rfloor - 1]$和$Y[\lceil n/2 \rceil - 1]$:

Case 1: 如果$X[\lfloor n/2 \rfloor - 1] = Y[\lceil n/2 \rceil - 1]$, 则 X 和 Y 合并后排序第 n 个元素必为$Y[\lceil n/2 \rceil - 1]$, 并且第$n+1$个元素为$\min\{X[\lfloor n/2 \rfloor], Y[\lceil n/2 \rceil]\}$;

Case 2: 如果$X[\lfloor n/2 \rfloor - 1] > Y[\lceil n/2 \rceil - 1]$, 则$X[\lfloor n/2 \rfloor - 1]$在 X 和 Y 合并后排序至少为 n, 而且$Y[\lceil n/2 \rceil - 1]$ 在 X 和 Y 合并后排序小于 n. 进一步比较$X[\lfloor n/2 \rfloor - 1]$和$Y[\lceil n/2 \rceil]$:

Case 2.1: 如果$X[\lfloor n/2 \rfloor - 1] \le Y[\lceil n/2 \rceil]$, 则$X[\lfloor n/2 \rfloor - 1]$在 X 和 Y 合并后排序为 n, 而$\min\{X[\lfloor n/2 \rfloor], Y[\lceil n/2 \rceil]\}$的排序为$n+1$;

Case 2.2: 如果$X[\lfloor n/2 \rfloor - 1] > Y[\lceil n/2 \rceil]$, 则$X[\lfloor n/2 \rfloor - 1]$在 X 和 Y 合并后排序至少为$n+1$, 此时Y_L中元素排序位置均小于n并且X_R中元素排序均大于$n+1$, 由此可知X和 Y 合并后排序第 n 和第 n+1 的元素在X_L和Y_R中仍为中位数. 原始问题转化为在X_L和Y_R中找到中位数.

Case 3: 如果$X[\lfloor n/2 \rfloor - 1] < Y[\lceil n/2 \rceil - 1]$, 则 $Y[\lceil n/2 \rceil - 1]$在 X 和 Y 合并后排序至少为 n, 而且$X[\lfloor n/2 \rfloor - 1]$ 在 X 和 Y 合并后排序小于 n. 进一步比较$Y[\lceil n/2 \rceil - 1]$和$X[\lfloor n/2 \rfloor]$:

Case 3.1: 如果$Y[\lceil n/2 \rceil - 1] \le X[\lfloor n/2 \rfloor]$, 则$Y[\lceil n/2 \rceil - 1]$在 X 和 Y 合并后排序为 n, 而$\min\{X[\lfloor n/2 \rfloor], Y[\lceil n/2 \rceil]\}$的排序为$n+1$;

Case 3.2: 如果$Y[\lceil n/2 \rceil - 1] > X[\lfloor n/2 \rfloor]$, 则$Y[\lceil n/2 \rceil - 1]$在 X 和 Y 合并后排序至少为$n+1$, 此时X_L中元素排序均小于n并且Y_R中元素排序均大于$n+1$, 由此可知X和 Y 合并后排序第 n 和第 n+1 的元素在X_R和Y_L中仍为中位数. 原始问题转化为在X_R和Y_L中找到中位数.

（2）写出算法伪代码.

FindMedium($X[l_X : u_X]$, $Y[l_Y : u_Y]$)

1. $m \leftarrow u_X - l_X + 1$;
2. if $m < 3$
3. 按大小合并$X[l_X : u_X]$和$Y[l_Y : u_Y]$到数组$M[0 : 2m-1]$;
4. return $M[m-1]$, $M[m]$;
5. if $X[l_X + \lfloor m/2 \rfloor - 1] = Y[l_Y + \lceil m/2 \rceil - 1]$
6. return $X[l_X + \lfloor m/2 \rfloor - 1]$, $\min\{X[l_X + \lfloor m/2 \rfloor], Y[l_Y + \lceil m/2 \rceil]\}$;
7. else if $X[l_X + \lfloor m/2 \rfloor - 1] > Y[l_Y + \lceil m/2 \rceil - 1]$
8. if $X[l_X + \lfloor m/2 \rfloor - 1] \le Y[l_Y + \lceil m/2 \rceil]$
9. return $X[l_X + \lfloor m/2 \rfloor - 1]$, $\min\{X[l_X + \lfloor m/2 \rfloor], Y[l_Y + \lceil m/2 \rceil]\}$;
10. else
11. return FindMedium($X[l_X : l_X + \lfloor m/2 \rfloor - 1], Y[l_Y + \lceil m/2 \rceil : u_Y]$);
12. else
13. if $Y[l_Y + \lceil m/2 \rceil - 1] \le X[l_X + \lfloor m/2 \rfloor]$
14. return $Y[l_Y + \lceil m/2 \rceil - 1]$, $\min\{X[l_X + \lfloor m/2 \rfloor], Y[l_Y + \lceil m/2 \rceil]\}$;
15. else
16. return FindMedium($X[l_X + \lfloor m/2 \rfloor : u_x], Y[l_Y : l_Y + \lceil m/2 \rceil - 1]$);

运行 FindeMedium($X[0 : n-1]$, $Y[0 : n-1]$)求解原始问题.

（3）分析算法的时间复杂性.

$T(n) \leq T\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(1),\ T(n) = O(\log n).$

1、证明：对于任意正整数d和任意常数a_d >0，有：$P(n)=\Sigma_{i=0}^{d}a_i n^i=\theta(n^d)$.

证明：对于$0\le i\le d-1$，可以通过n的增长使得$|a_i n^i|\le\frac{a_d n^d}{d+1}$，这要求$n\ge\sqrt[d-i]{\frac{(d+1)|a_i|}{a_d}}$，此时有$-\frac{a_d n^d}{d+1}\le a_i n^i\le\frac{a_d n^d}{d+1}$。于是，当$n\ge\max\limits_{i=0,1,\dots,d-1}\left\{\sqrt[d-i]{\frac{|a_i|}{a_d}}\right\}$时，$a_d n^d-\frac{da_d n^d}{d+1}\le\Sigma_{i=0}^{d}a_i n^i\le a_d n^d+\frac{da_d n^d}{d+1}$。因此，当$c_1=\frac{a_d}{d+1}$，$c_2=\frac{(2d+1)a_d}{d+1}$，$n\ge\max\limits_{i=0,1,\dots,d-1}\left\{\sqrt[d-i]{\frac{(d+1)|a_i|}{a_d}}\right\}$时，$c_1 n^d\le P(n)\le c_2 n^d$，得证。

2、证明：（1）$O\big(f(n)\big)+O\big(g(n)\big)=O(f(n)+g(n))$;

（2）$O\big(f(n)\big)\times O\big(g(n)\big)=O(f(n)\times g(n))$.

证明：(1) 对于任意$p(n)=O(f(n))$，存在$c_p>0$，当$n\ge n_p$时，$p(n)\le c_p f(n)$；对于任意$q(n)=O\big(g(n)\big)$，存在$c_q>0$，当$n\ge n_q$时，$q(n)\le c_q g(n)$。令$c=\max\{c_p,c_q\}>0$，当$n\ge n_0=\max\{n_p,n_q\}$时，有$p(n)+q(n)\le c_p f(n)+c_q g(n)\le c(f(n)+g(n))$，得证。

（2）对于任意$p(n)=O(f(n))$，存在$c_p>0$，当$n\ge n_p$时，$p(n)\le c_p f(n)$；对于任意$q(n)=O\big(g(n)\big)$，存在$c_q>0$，当$n\ge n_q$时，$q(n)\le c_q g(n)$。令$c=c_p\times c_q>0$，当$n\ge n_0=\max\{n_p,n_q\}$时，有$p(n)\times q(n)\le c_p f(n)\times c_q g(n)=c(f(n)\times g(n))$，得证。

3、证明：$\sum_{j=0}^{\lfloor\log_b n\rfloor}(\log_b n-j)^k=\Omega((\log_b n)^{k+1})$，$k$为大于 0 的常数.

证明：$\sum_{j=0}^{\lfloor\log_b n\rfloor}(\log_b n-j)^k=(\log_b n)^k+(\log_b n-1)^k+(\log_b n-2)^k+\cdots+\left(\log_b n-\left\lfloor\frac{1}{2}\log_b n\right\rfloor\right)^k+\left(\log_b n-\left\lfloor\frac{1}{2}\log_b n\right\rfloor-1\right)^k+\cdots+(\log_b n-\lfloor\log_b n\rfloor)^k\ge(\log_b n)^k+(\log_b n-1)^k+(\log_b n-2)^k+\cdots+\left(\log_b n-\left\lfloor\frac{1}{2}\log_b n\right\rfloor\right)^k\ge\left(\log_b n-\left\lfloor\frac{1}{2}\log_b n\right\rfloor\right)^k\times\left(1+\left\lfloor\frac{1}{2}\log_b n\right\rfloor\right)$

$\ge\left(\frac{1}{2}\log_b n\right)^k\frac{1}{2}\log_b n=(\frac{1}{2})^{k+1}(\log_b n)^{k+1}$。

即当$c=(\frac{1}{2})^{k+1}$为正常数时，有$\sum_{j=0}^{\lfloor\log_b n\rfloor}(\log_b n-j)^k\ge c(\log_b n)^{k+1}$，得证。

4、解递归方程：$T(n)=T(an)+T(bn)+n$，其中$a,b>0$，$a+b=1$.

解：猜测$T(n)=O(n\log n)$，即当n足够大时，存在常数c使得$T(n)\le cn\log n$。

$$\begin{aligned}T(n)&=T(an)+T(bn)+n\\&\le can\log an+cbn\log bn+n\\&=can\log n-can\log\frac{1}{a}+cbn\log n-cbn\log\frac{1}{b}+n\\&=cn\log n-\left(c\left(a\log\frac{1}{a}+b\log\frac{1}{b}\right)-1\right)n\end{aligned}$$

当$c\ge\frac{1}{a\log\frac{1}{a}+b\log\frac{1}{b}}$时，有$T(n)\le cn\log n$，因此$T(n)=O(n\log n)$。

用类似的过程可以证明$T(n)=\Omega(n\log n)$。

5、解递归方程：$T(n)=T(an)+T(bn)+n$，其中$a,b>0$，$a+b<1$.

解：猜测$T(n)=O(n)$，即当n足够大时，存在常数c使得$T(n)\le cn$。

$$\begin{aligned}T(n)&=T(an)+T(bn)+n\\&\le can+cbn+n\\&=(c(a+b)+1)n\end{aligned}$$

当$c\ge\frac{1}{1-a-b}$时，有$T(n)\le cn$，因此$T(n)=O(n)$。

从$T(n)$的定义可知$T(n)=\Omega(n)$。

6、用 Master 方法求解：$T(n)=4T\left(\frac{n}{2}\right)+\theta(n\log n)$.

解：$a=4$，$b=2$，$\log_b a=2$，$n^{\log_b a}=n^2$，$f(n)=n\log n=O(n^{2-\varepsilon})$，其中$\varepsilon<1$即可。因此，$T(n)=\theta\left(n^{\log_b a}\right)=\theta(n^2)$。

下述问题中，请做如下部分回答：（1）用自然语言描述问题的优化解包含哪些子问题的优化解，注意说明用到的符号的含义；（2）写出优化解代价的递归方程，注意说明每个符号的含义；（3）写出算法伪代码；（4）分析算法的时间复杂性。

1、给定一个整数序列 $a_1,\ldots,a_n$。相邻两个整数可以合并，合并之后的结果是两个整数之和（用和替换原来的两个整数），合并两个整数的代价是这两个整数之和。通过不断合并最终可以将整个序列合并成一个整数，整个过程的总代价是每次合并操作代价之和。试设计一个动态规划算法给出 $a_1,\ldots,a_n$ 的一个合并方案使得该方案的总代价最大。

优化子结构：

所有合并操作中的最后一次合并将两个整数x和y合并为一个整数，其中x和y一定是合并输入中的连续整数序列所得，即存在整数k，使得x是 $a_1,\ldots,a_k$ 的合并结果，而y是 $a_{k+1},\ldots,a_n$ 的合并结果。由合并操作的说明可知，$x=\sum_{i=1}^{k}a_i$，$y=\sum_{k+1}^{n}a_i$，而且最后一次合并的代价为$x+y=\sum_{i=1}^{n}a_i$。因此，最优解的总代价是合并 $a_1,\ldots,a_k$ 得到x的最大代价、合并 $a_{k+1},\ldots,a_n$ 得到y的最大代价、以及$x+y=\sum_{i=1}^{n}a_i$的加和。其中包括合并 $a_1,\ldots,a_k$ 和 $a_{k+1},\ldots,a_n$ 对应的两个子问题的优化解代价。

优化解代价的递归方程：

令$M[i,j]$表示合并$a_i\ldots a_j$的优化解代价，$S[i,j]=\sum_{p=i}^{j}a_p$.

边界条件：$M[i,j]=0$, 如果$i=j$;

当$i>j$时，$M[i,j]=\min_{i\le k<j}\{M[i,k]+M[k+1,j]\}+S[i,j]$。

算法思路：

逐条对角线计算，首先用边界条件确定满足$i-j=0$的$M[i,j]$，随后增加$i-j$的差值，并计算满足该差值的$M[i,j]$。$M[1,n]$即位优化解的代价。

计算$M[i,j]$时，用$B[i,j]$记录最优的划分点k，并通过B构造优化的合并序列。

伪代码：略

代价：$O(n^3)$.

2、最长增长子序列问题定义如下：

输入：由 n 个数组成的一个序列 S：$a_1,a_2,\ldots,a_n$

输出：S 的子序列 $S'=b_1,b_2,\ldots,b_k$，满足：

(1) $b_1\le b_2\le\ldots\le b_k$，

(2) $|S'|$最大

使用动态规划技术设计算法求解最长增长子序列问题.

优化子结构：

令 $b_1,b_2,\ldots,b_k$ 是一个优化解，其中 b_k 对应 a_i，b_{k-1} 对应 a_j，则 $i>j$，$a_i\ge a_j$ 而且 $b_1,b_2,\ldots,b_{k-1}$ 必须是 $a_1,a_2,\ldots,a_i$ 的最长增长子序列。

优化解代价的递归方程：

令$L[i]$表示以 a_i 结尾的增长子序列的最大长度，最长增长子序列必定以某个元素结尾，因此 $\max\{L[i]\}$即最长增长子序列的长度。$L[i]$的求解如下：

边界条件：$L[0]=0$, $L[1]=1$;

当 $i>1$ 时，$L[i]=\max\limits_{\substack{0\le j\le i-1\\ a[i]\ge a[j]}}\{L[j]+1\}$.

算法思路：

从左向右计算 $L[i]$，计算过程中 a_i 的前序元素保存为使 $L[i]$最大的 a_j;

扫描所有 $L[i]$并取其最大值，即为最长增长子序列的长度，使用该长度计算过程中保存的前序元素构造最长增长子序列。

伪代码：略。

代价：$O(n^2)$。

（选做）3、最长公共增长子序列问题定义如下：

输入：由 n 个数组成的一个序列 S：$a_1,a_2,\ldots,a_n$,

由 m 个数组成的一个序列 T：$b_1,b_2,\ldots,b_m$.

输出：S 和 T 的公共子序列 $X=c_1,c_2,\ldots,c_k$，满足：

(1) $c_1\le c_2\le\ldots\le c_k$,

(2) $|X|$最大

使用动态规划技术设计算法求解最长公共增长子序列问题.

优化子结构：

令 $c_1, c_2, \ldots, c_{k-1}, c_k$ 为是一个优化解，而且 c_k 在 S 和 T 中分别对应 a_i 和 b_j，c_{k-1} 在 S 和 T 中分别对应 a_p 和 b_q，则 $a_i=b_j$，$c_1, c_2, \ldots, c_{k-1}, c_k$ 是 $a_1,a_2,\ldots,a_i$ 和 $b_1,b_2,\ldots,b_j$ 的包含它们各自最后一个字符的最长公共增长子序列；$a_p=b_q$，$c_1, c_2, \ldots, c_{k-1}$ 是 $a_1,a_2,\ldots,a_p$ 和 $b_1,b_2,\ldots,b_q$ 的包含它们各自最后一个字符的最长公共增长子序列。

优化解代价的递归方程：

令$L[i,j]$表示 $a_1,a_2,\ldots,a_i$ 和 $b_1,b_2,\ldots,b_j$ 的包含它们各自最后一个字符的最长公共增长子序列的长度，当$a_i\neq b_j$时，$L[i,j]=0$。$a_1,a_2,\ldots,a_n$ 和 $b_1,b_2,\ldots,b_m$ 的最长公共增长子序列的长度为 $\max\limits_{1\le i\le n,1\le j\le m}\{L[i,j]\}$。$L[i,j]$的求解过程如下：

边界条件：

$$L[i,1]=\begin{cases}1 & \text{如果}a_i=b_1;\\ 0 & \text{如果}a_i\neq b_1;\end{cases}\ 1\le i\le n;$$

$$L[1,j]=\begin{cases}1 & \text{如果}a_1=b_j;\\ 0 & \text{如果}a_1\neq b_j;\end{cases}\ 2\le j\le m.$$

当$i>1$而且$j>1$时，如果$a_i\neq b_j$时，则$L[i,j]=0$；否则，$L[i,j]=\max\limits_{\substack{p<i,q<j,\\ a_p=b_q,a_p\le a_i}}\{L[p,q]\}+1$。

算法思路：

逐行计算$L[i,j]$，并保留计算$L[i,j]$时使用的子问题代价$L[p,q]$。最终，$\max\limits_{1\le i\le n,1\le j\le m}\{L[i,j]\}$为优

化解代价，从优化解代价对应的$L[i,j]$开始，反向便利计算过程中使用的子问题，构建最长公共增长子序列。

以下为基本的方法：

伪代码：

LongestIncreasingSubsequenceBasic($a_1,a_2,\ldots,a_n, b_1,b_2,\ldots,b_m$)

```
1.  for i =1 to n
2.      if a_i= b_1;
3.          L[i,1] ←1;
        else
            L[i,1] ←0;
4.  for j =2 to m
5.      if a_1= b_j;
6.          L[1,j] ←1;
7.      else
8.          L[1,j] ←0;
9.  for i = 2 to n
10.     for j =2 to m
11.         if a_i = b_j
12.             L[i,j] ← 1;
13.             for p = 1 to i-1
14.                 for q =1 to j-1
15.                     if a_p ≤ a_i and L[p,q] + 1 > L[i,j]
16.                         L[i,j] ← L[p,q] + 1;
17.                         B[i,j] ← (p,q);
18.     else
19.         L[i,j] ← 0;
20. return L, B;
```

代价：$O(n^2m^2)$，该代价可以降低为 $O(n^2m)$或 $O(nm^2)$，并进一步降低为 $O(mn)$，详情见下一页。

以下为改进的方法：计算$L[i,j]$时不必扫描所有的$L[p,q]$，其中$1 \le p < i$，$1 \le q < j$。将 L 中每列中已经计算的最大值存储在$MAX[1]$~ $MAX[m]$中，计算该最大值所需的子问题存储在$SP[1]$~ $SP[m]$中。这样，计算$L[i,j]$时不必扫描左上角的整个方块中的所有$L[p,q]$，只需扫描$MAX[1]$~$MAX[j-1]$即可，如此可将算法代价降低为 $O(n^2m)$或 $O(nm^2)$。更进一步，可以避免每次计算都扫描$MAX[1]$~$MAX[j-1]$，在计算第 i 行时，用$MaxSPCost$和$MaxSPLoc$维护计算当前元素所需的子问题代价和子问题位置，计算$L[i,j]$时不必扫描$MAX[1]$~$MAX[j-1]$。通过上述方法，在子问题优化解代价已知的情况下，计算$L[i,j]$的代价从$O(ij)$降低为 $O(n)$或 $O(m)$，最后降低为$O(1)$，最终算法代价降低为 $O(mn)$。

伪代码：

LongestIncreasingSubsequenceImproved($a_1,a_2,\ldots,a_n, b_1,b_2,\ldots,b_m$)

1. **for** i =2 to n
2. **if** $a_i = b_1$;
3. $L[i,1] \leftarrow 1$;
4. else
5. $L[i,1] \leftarrow 0$;
6. **for** j =1 to m
7. $MAX[j] \leftarrow 0$;
8. $SP[j] \leftarrow NIL$;
9. **if** $a_1 = b_j$;
10. $L[1,j] \leftarrow 1$;
11. else
12. $L[1,j] \leftarrow 0$;
13. **for** i = 2 to n
14. **for** j= 1 to n
15. **if** $L[i-1,j] > MAX[j]$
16. $MAX[j] \leftarrow L[i-1,j]$;
17. $SP[j] \leftarrow B[i-1,j]$;
18. $MaxSPCost \leftarrow 0$;
19. **if** $a_i > b_1$
20. $MaxSPCost \leftarrow Max[1]$;
21. $MaxSPLoc \leftarrow SP[j]$;
22. **for** j =2 to m
23. **if** $a_i = b_j$
24. $L[i,j] \leftarrow 1$;
25. **if** $MaxSPCost > 0$
26. $L[i,j] \leftarrow MaxSPCost + 1$;
27. $B[i,j] \leftarrow MaxSPLoc$;
28. **else**
29. $L[i,j] \leftarrow 0$;
30. **if** $a_i > b_j$ **and** $MAX[j] > MaxSPCost$
31. $MaxSPCost \leftarrow MAX[j]$;
32. $MaxSPLoc \leftarrow SP[j]$;
33. **return** L, B;

代价：$O(mn)$。

1、存放于磁带上文件需要顺序访问。故假设磁带上依次存储了 n 个长度分别是 $L[1],\ldots,L[n]$ 的文件，则访问第 k 个文件的代价为 $\sum_{j=1}^{k} L[j]$。

现给定 n 个文件的长度 $L[1],\ldots,L[n]$，并假设每个文件被访问的概率相等，试给出这 n 个文件在磁带上的存储顺序使得平均访问代价最小。

（1）简述算法采用的贪心策略。

将文件按照从小到大的顺序存储在磁带上。

（2）表述并证明问题的贪心选择性。

给定 n 个文件的长度分别是 $L[1],\ldots,L[n]$，其中访问第 k 个文件的代价为 $\sum_{j=1}^{k} L[j]$，每个文件被访问的概率相等，令 $j = arg\min_i L[i]$，则存在一个存储方案 T，T 的平均访问代价是所有存储方案中最小的，而且 T 中 $L[j]$ 排在第一位.

证明：令 T' 是一个平均访问代价最小的存储方案，若 $L[j]$ 在 T' 中排在第一位，则得证。

否则，令文件 j（长度最小的文件）在 T' 中排在第 x 位，T' 中文件长度依次为 $L[i_1]$, $L[i_2]$, ..., $L[i_n]$，其中第 m 个文件的访问代价为 $\sum_{p=1}^{m} L[i_p]$。交换 T' 中排在第 1 的文件和文件 j（在 T' 中排在第 x 个）后，得到存储方案 T，往证 T 的平均访问代价不大于 T'。

T' 的平均访问代价如下：

$$C(T') = \frac{1}{n}\sum_{m=1}^{n}\sum_{k=1}^{m} L[i_k] = \frac{1}{n}\sum_{m=1}^{x-1}\sum_{k=1}^{m} L[i_k] + \frac{1}{n}\sum_{m=x}^{n}\sum_{k=1}^{m} L[i_k].$$

T 的平均访问代价如下：

$$C(T) = \frac{1}{n}\sum_{m=1}^{x-1}\left(L[j] + \sum_{k=2}^{m} L[i_k]\right) + \frac{1}{n}\sum_{m=x}^{n}\sum_{k=1}^{m} L[i_k]$$

$$\leq \frac{1}{n}\sum_{m=1}^{x-1}\sum_{k=1}^{m} L[i_k] + \frac{1}{n}\sum_{m=x}^{n}\sum_{k=1}^{m} L[i_k] = C(T').$$

由此得证 T 具有最小的平均访问代价，而且 T 中 $L[j]$ 排在第一位.

（3）表述并证明问题的优化子结构。

给定包含 n 个文件的集合 S，其中文件的长度分别是 $L[1],\ldots,L[n]$，访问第 k 个文件的代价为 $\sum_{j=1}^{k} L[j]$，每个文件被访问的概率相等，令 $j = arg\min_i L[i]$，令 T 是平均访问代价最小的存储方案，而且 T 中 $L[j]$ 排在第一位，记 $T = j, i_1, \ldots, i_{n-1}$，则 $T' = i_1, \ldots, i_{n-1}$ 是 $S - \{j\}$ 的平均访问代价最小的存储方案.

证明：记 $C(P)$ 为存储代价 P 的平均访问代价，则有

$$C(T) = \frac{1}{n}\left(L[j] + \sum_{m=1}^{n-1}\left(L[j] + \sum_{k=1}^{m} L[i_k]\right)\right)$$

$$= L[j] + \frac{1}{n}\sum_{m=1}^{n-1}\sum_{k=1}^{m} L[i_k] =$$

$$L[j] + \frac{n-1}{n}C(T').$$

若 $T' = i_1, \ldots, i_{n-1}$ 不是 $S - \{j\}$ 的平均访问代价最小的存储方案，则存在 T'' 是 $S - \{j\}$ 的平均访问代价最小的存储方案，通过与上式类似的推导，可以得到文件 j 和 T'' 组成的存储方案的平均访问代价为

$$L[j] + \frac{n-1}{n}C(T'') < L[j] + \frac{n-1}{n}C(T') = C(T)。$$

与 T 是平均访问代价最小的存储方案矛盾，因此 $T' = i_1, \ldots, i_{n-1}$ 是 $S - \{j\}$ 的平均访问代价最小的存储方案。

（4）写出算法的伪代码。

略。

（5）分析算法的时间复杂度。

$O(n\log n)$.

2、程序员接到 n 项编程任务，第 i 项任务需要在时间点 E_i 之前完成，完成第 i 项任务需要的时间（预计）为 t_i。每个任务顺利完成之后，程序员将得到固定的报酬 a；如果未能按时完成某项任务，则程序员不能得到该任务的酬金。试设计一个贪心算法，为程序员安排工作计划，使其获得酬金最大。

（1）简述算法采用的贪心策略。

每次选择当前可以选择的(即$t_i \le E_i$)长度最小的任务，放入任务队列中，已经选择的任务按照最晚结束时间从小到大排序.

（2）表述并证明问题的贪心选择性。

设$A=\{a_1, a_2, \dots, a_n\}$, $a_i=(t_i, T_i)$, $p=arg_i \min\limits_{1\le i\le n} t_i$. 必然存在$S=\{a_{j_1}, a_{j_2}, \dots, a_{j_k}\}$是$A$的最优调度，$S$包含$a_p$，并且$S$中的任务按照最晚完成时间升序排列.

证明：首先证明存在优化解包含任务p. 令T是A的一个优化调度，如果T包含任务p，则得证优化解包含任务p. 否则，用任务p替换T中的第一个任务，因为任务p的长度不超过T中第一个任务的长度，T中的其余任务的完成时间不会后延，于是得到一个新的调度P，其中包含任务数量和最优解T相同，因此P也是一个最优调度，并且包含任务p.

若P中的任务按照最晚完成时间从小到大排序，则贪心选择性得证. 否则，考虑 P 中任意相邻的两个任务a_x和a_y，其中a_x在前，而且任务a_y的完成时间必然不超过E_y. 如果$E_x > E_y$，则交换任务a_x和a_y后，其他任务的完成时间不变，a_y的完成时间提前，a_x的完成时间后延至交换前a_y的完成时间. 由于交换前a_y的完成时间必然不超过$E_y < E_x$，交换后a_x必然能够按时完成，其他任务显然也可以按时完成. 因此，对于任意相邻的两个任务a_x和a_y，其中a_x在前，$E_x > E_y$，则交换任务a_x和a_y后，所有任务依旧可以按时完成，可调度的任务数量不变. 可以从P开始，不断地交换最晚完成时间逆序的相邻任务，直至得到按照完成时间排序的调度S，此时S中按时完成的任务数量与P相同，S也是优化解，即存在优化解S包含任务p，而且S中的任务按照最晚完成时间排序.

（3）表述并证明问题的优化子结构。

设$A=\{a_1, a_2, \dots, a_n\}$, $a_i=(t_i, T_i)$, $p=arg_i \min\limits_{1\le i\le n} t_i$. S是A的最优调度，S包含a_p，并且S中的任务按照最晚完成时间升序排列.复制$A/\{a_p\}$中的元素到A'，修改A'中的元素，对于$a_q \in A'$：如果$T_q > T_p$，则$T_q \leftarrow T_q - t_p$；如果$T_q \le T_p$，则$T_q \leftarrow min\{T_q, T_p - t_p\}$；如果$T_q < t_q$，则从$A'$中去掉$a_q$，即$A' \leftarrow A'/\{a_q\}$. 从$S$中去掉$a_p$，其余任务顺序不变得到$S'$，则$S'$是$A'$的最优调度，而且$S'$中任务按照最晚完成时间升序排列.

证明：易知S'中任务按照最晚完成时间升序排列. 如果S'不是A'的最优调度，令S''是A'的最优调度，而且S''中任务按照最晚完成时间升序排列. 于是$|S''| > |S'|$. 将S''中最晚完成时间大于$T_p - t_p$任务向后延时t_p，将得到三个部分： S''中最晚完成时间不大于$T_p - t_p$的任务序列；长度为t_p的空闲时间； S''中最晚完成时间大于$T_p - t_p$的任务序列. 将a_p置于空闲时间将得到一个A的调度S'''，而且$|S'''| = |S''| + 1 > |S'| + 1 = |S|$ ，与S是A的最优调度矛盾！ 因此S'是A'的最优调度，而且S'中任务按照最晚完成时间升序排列.

（4）写出算法的伪代码。

Optimal-plan(A)

Input: A=$\{a_i = (t_i,\ T_i) \mid\ i = 1,2, \dots,\ n\}$

Output: $P = \{a_{j_1}, a_{j_2}, \dots, a_{j_k}\}$

1. $S \leftarrow \emptyset$
2. $A' = \{1, \dots, n\}$
3. While $A' \neq \emptyset$
4. $p \leftarrow arg_i \min_{i \in A'} t_i$
5. $S \leftarrow S \cup \{p\}$
5. For $q \in A'$
6. If $T_q \leq T_p - t_p$
7. $T_q \leftarrow min\{T_q,\ T_p - t_p\}$
8. Else if $T_q > T_p - t_p$
9. $T_q \leftarrow T_q - t_p$
10. If $T_q < t_q$
11. $A' \leftarrow A'/\{q\}$
12. $P = \{a_j | j \in S\}$并将P中元素按照T_j升序排序
15. Return P

（5）分析算法的时间复杂度。

$O(n^2)$.

1、对支持插入、删除的动态表中的操作进行平摊分析:

(1) 第 i 次操作是 TABLE-DELETE: 未收缩;

(2) 第 i 次操作是 TABLE-DELETE: 收缩.

解: 令f_i为动态表在第 i 个操作之后的装载因子, ϕ_i是第 i 个操作后的势能, c_i是第 i 个操作的真实代价, α_i是第 i 个操作的平摊代价.

(1). 由于没有收缩, 因此$size_i = size_{i-1}$, $num_i = num_{i-1} - 1$, $c_i = 1$.

情况一: $f_{i-1} < 1/2$, $f_i < 1/2$并且未收缩. $\phi_i = \frac{1}{2}size_i - num_i$, $\phi_{i-1} = \frac{1}{2}size_{i-1} - num_{i-1}$,

$\alpha_i = c_i + \phi_i - \phi_{i-1} = 1 + \left(\frac{1}{2}size_i - num_i\right) - \left(\frac{1}{2}size_{i-1} - num_{i-1}\right) = 2.$

情况二: $f_{i-1} \geq 1/2$, $f_i < 1/2$并且未收缩. $\phi_i = \frac{1}{2}size_i - num_i$, $\phi_{i-1} = 2num_{i-1} - size_{i-1}$,

$\alpha_i = c_i + \phi_i - \phi_{i-1} = 1 + \left(\frac{1}{2}size_i - num_i\right) - (2num_{i-1} - size_{i-1}) = 2 + \frac{3}{2}size_{i-1} - 3num_{i-1} \leq 2.$

情况三: $f_{i-1} > 1/2$, $f_i \geq 1/2$并且未收缩. $\phi_i = 2num_i - size_i$, $\phi_{i-1} = 2num_{i-1} - size_{i-1}$.

$\alpha_i = c_i + \phi_i - \phi_{i-1} = 1 + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) = -1.$

(2). 由于删除后收缩, 因此$f_{i-1} = 1/4$, $f_i < 1/2$, $size_i = \frac{1}{2}size_{i-1}$, $num_i = num_{i-1} - 1$, $c_i = num_{i-1}$ (删掉一个元素, 再拷贝剩余的$num_{i-1} - 1$个元素; 当然, 这个代价还可以是$c_i = num_{i-1} - 1$或者$c_i = num_{i-1} + 1$, 例如只拷贝未删除的元素或者先拷贝所有元素再删除一个). $\phi_i = \frac{1}{2}size_i - num_i$, $\phi_{i-1} = \frac{1}{2}size_{i-1} - num_{i-1}$, $\alpha_i = c_i + \phi_i - \phi_{i-1} = num_{i-1} + \left(\frac{1}{2}size_i - num_i\right) - \left(\frac{1}{2}size_{i-1} - num_{i-1}\right) = 2 + num_i - \frac{1}{2}size_i < 2.$

下面补充第 i 次操作是 TABLE-INSERTION 的情况, $num_i = num_{i-1} + 1$.

(3). 第 i 次操作是 TABLE-INSERTION, 未发生扩张; 由于没有扩张, 因此$size_i = size_{i-1}$, $num_i = num_{i-1} + 1$, $c_i = 1$.

情况一: $f_{i-1} < 1/2$, $f_i < 1/2$并且未扩张. $\phi_i = \frac{1}{2}size_i - num_i$, $\phi_{i-1} = \frac{1}{2}size_{i-1} - num_{i-1}$,

$\alpha_i = c_i + \phi_i - \phi_{i-1} = 1 + \left(\frac{1}{2}size_i - num_i\right) - \left(\frac{1}{2}size_{i-1} - num_{i-1}\right) = 0.$

情况二: $f_{i-1} < 1/2$, $f_i \geq 1/2$并且未扩张. $\phi_{i-1} = \frac{1}{2}size_{i-1} - num_{i-1}$, $\phi_i = 2num_i - size_i$,

$\alpha_i = c_i + \phi_i - \phi_{i-1} = 1 + (2num_i - size_i) - \left(\frac{1}{2}size_{i-1} - num_{i-1}\right) = 3 + 3num_{i-1} - \frac{3}{2}size_{i-1} < 3.$

情况三: $f_{i-1} \geq 1/2$, $f_i > 1/2$并且未扩张. $\phi_i = 2num_i - size_i$, $\phi_{i-1} = 2num_{i-1} - size_{i-1}$.

$\alpha_i = c_i + \phi_i - \phi_{i-1} = 1 + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) = 3.$

(4). 第 i 次操作是 TABLE-INSERTION, 发生扩张. $f_{i-1} = 1$, $f_i > \frac{1}{2}$, $size_i = 2size_{i-1}$, $c_i = num_i$, $\phi_i = 2num_i - size_i$, $\phi_{i-1} = 2num_{i-1} - size_{i-1}$. $\alpha_i = c_i + \phi_i - \phi_{i-1} = num_i +$

$(2num_i - size_i) - (2num_{i-1} - size_{i-1}) = 3.$

2、假设二元计数器初始时有 b 个 1，而不是全 0. 给定$n = \Omega(b)$，说明执行 n 个自增操作的代价为 O(n).

解：设置势能为二元计数器中 1 的个数，由于初始状态下 1 的个数$s_0 > 0$，这样的势能设置方法不能保证$\phi_i \geq \phi_0$，即平摊代价之和$\sum_1^n \alpha_i = \sum_1^n (c_i + \phi_i - \phi_{i-1}) = \sum_1^n c_i + \phi_n - \phi_0$不再是真实代价之和$\sum_1^n c_i$的上界，但是上述数学关系依旧是满足的。因此可以得到

$$\sum_1^n c_i = \sum_1^n \alpha_i - \phi_n + \phi_0.$$

根据势能设置方法得到$\sum_1^n \alpha_i = 2n$，此处与课堂上讲授的方法一致，但是要注意这里的$2n$是平摊代价之和的上界。

因此，$\sum_1^n c_i = \sum_1^n \alpha_i - \phi_n + \phi_0 \leq \sum_1^n \alpha_i + \phi_0 = 2n + b.$

当$n = \Omega(b)$时，显然可以得到$2n + b = O(n)$. 注意，如果$b = \Omega(n)$，则$2n + b = O(b)$.

3、用两个栈实现一个队列，使得 ENQUEUE 和 DEQUEUE 操作的平摊代价都是 O(1).

解：进队列操作将元素压入栈 A，出队列操作将栈 B 中栈顶元素弹出，若栈 B 中无元素，则将 A 中所有元素依次弹出并压入栈 B.

使用会计方法，进队列操作代价 4，其中 1 支付将元素压入 A 的代价，1 支付该元素从 A 中弹出的代价，1 支付该元素压入 B 的代价，1 支付该元素弹出 B 的代价；出队列操作平摊代价 0.

4、设计一个数据结构支持由整数组成的动态多重集合 S（包含可能重复整数的集合）中的操作：（1）Insert(S, x)将整数 x 插入 S 中；（2）Delete-Larger-Half(S)删除 S 中最大的$\lceil |S|/2 \rceil$个整数。对于任意由 Insert 和 Delete-Larger-Half 组成的长度为 m 的操作序列，要求其总体代价是 O(m)，并且可以在 O(|S|)代价内输出 S 的所有元素。

解：用一个链表存储所有数据，每个节点存储一个整数。Insert 操作在链表末尾添加一个节点，存储新插入的整数；Delete-Larger-Half 操作首先将链表中元素拷贝到一个数组中，该数组大小为当前链表中元素的数量，然后使用 selection 算法查找中位数，然后并遍历所有元素查看有多少个元素与中位数相同，以此断定需要删除多少个与中位数相同的整数和所有比中位数大的整数才能完成 Delete-Larger-Harf，随后遍历链表中的元素，删除比中位数大的元素，并删除相应数量的中位数元素。这个操作的代价为 O(n)，其中 n 是执行 Delete-Larger-Half 操作时链表中元素的数量。不妨设 Delete-Larger-Half 的代价不超过 cn (由其代价为 O(n)可得存在常数 c). 使用会计方法，Insert 的平摊代价设置为(1+2c)，其中 1 用于支付在链表中插入元素的代价 1，2c 附加新插入的元素上，当删除$\lceil n/2 \rceil$个元素时，这些元素上的代价共计不少于 cn，足以支付 Delete-Larger-Half 的代价（该代价不超过 cn）；Delete-Larger-Half 的平摊代价为 0。任意时刻，平摊代价之和与真实代价之和的插值不小于链表中元素数量的(1+2c)倍，一定非负。