




# 凸完全单调性的一个 加强与应用

西安市第一中学 杨哲



# 第一部分

四边形不等式、凸完全单调性与决策单调性以及凸完全单调性的一个加强

# 四边形不等式、凸完全单调性与决策单调性

- 对于一个权函数 $w(i, j)$ ，如果它满足 $w(x, i+1) - w(x, i)$  随 $x$ 单调不增，亦即 $w(x, i+1) + w(x+1, i) \geq w(x, i) + w(x+1, i+1)$ ，则称这个权函数满足凸完全单调性。
- 容易证明，当 $k > 0$  时， $w(x, i+k) - w(x, i)$  随 $x$ 单调不减， $w(i+k, x) - w(i, x)$  随 $x$ 单调不减。
- 所以对任意的 $a \leq b \leq c \leq d$ ，有 $w(a, d) + w(b, c) \geq w(a, c) + w(b, d)$ 。称此不等式为四边形不等式。
- 由四边形不等式也可推出凸完全单调性，所以“ $w$  满足四边形不等式”与“ $w$ 具有凸完全单调性”这两种说法是等价的。

# 四边形不等式、凸完全单调性与决策单调性

- 在一类要求将一段序列划分为若干子段，从 $i$ 到 $j$ 的一段的费用为 $w(i, j)$ ，要求出所有子段代价之和最小的划分方案的动态规划问题中，通常可以见到这样的状态转移方程：

$$f[x] = \min\{f[i] + w(i, x) : 1 \leq i < x\}$$

- 设 $t(i, x) = f[i] + w(i, x)$ ，如果对于某个 $x$ ， $t(i, x) \geq t(j, x)$  ( $i < j$ )，则对于任何 $y > x$ ，有 $t(i, y) \geq t(j, y)$ 。此式说明，对于 $i < j$ ，一旦某个时刻决策 $i$ 没有决策 $j$ 好，以后决策 $i$ 也不会比决策 $j$ 好。这说明， $f[x]$  的决策是随 $x$ 单调不减的，这就是决策的单调性。

# 四边形不等式、凸完全单调性与决策单调性

- 解决这类问题时，通常用 $B[i]$ 记录使决策 $i$ 比所有之前的决策 $j$  ( $j < i$ )要好的最小的 $x$ ，即 $B[i] = \min \{x : t(i, x) < t(j, x) \text{ 对所有 } j < i \text{ 均成立}\}$ 。
- 根据决策的单调性，决策 $i$ 比所有之前的决策 $j$  ( $j < i$ )要好等价于 $B[i] \leq x$ 。
- 如果对某个 $(i, j)$  ( $i < j$ )， $B[i] \geq B[j]$ ，则说明决策 $i$ 是无用的。
- 于是任何时刻，假设所有有用决策为  $i_1, i_2, \dots, i_k$ ，满足  $i_1 < i_2 < \dots < i_k$ ，则 $B[i_1] < B[i_2] < \dots < B[i_k]$ 。

# 四边形不等式、凸完全单调性与决策单调性

- 求解 $f[x]$ 时，如果 $j = \max \{j : B[i_j] \leq x, j \leq k\}$ ，则此时决策 $i_j$ 一定是最好的，即 $f[x] = t(i_j, x)$ 。利用决策的单调性，这个 $j$ 可以接着上次查找，所以 $n$ 次找 $j$ 的时间复杂度为 $O(n + \text{决策序列长度}) = O(n)$ 。
- 在求出了 $f[x]$ 之后， $x$ 将成为一个有用决策，我们需要求出 $B[x]$ ，以及维护这个决策序列。

# 四边形不等式、凸完全单调性与决策单调性

- 假设可以在 $T$ 单位时间内求出 $y = \min\{y : t(x, y) < t(i_k, y)\}$ 。
- 那么如果 $y \leq B[i_k]$ ，一定有 $B[x] \leq B[i_k]$ 。于是 $i_k$ 是无用决策，这时应当在决策序列中删去这个 $i_k$ （只需要让 $k \leftarrow k - 1$ ）。继续这个过程，直到 $\min\{y : t(x, y) < t(i_k, y)\} > B[i_k]$ 。此时， $B[x] = y$ （因为不可能再小），并且 $x$ 应当被添加到这个决策序列的末尾。
- 容易发现，用栈可以很好的完成这个序列的维护，由于每个决策至多进出栈一次，每个决策出栈至多消耗 $T$ 单位时间，于是维护序列的时间复杂度是 $O(nT)$ 。又因为决策的单调性，查找合适决策的时间复杂度是 $O(n)$ 的，所以总的时间复杂度是 $O(nT)$ 。

# 四边形不等式、凸完全单调性与决策单调性

- 有时，因为函数 $w$ 的表达式便于求出 $\min\{y : t(x, y) < t(i_k, y)\}$ ，所以 $T = O(1)$ ；一般情况下，根据 $w$ 的凸性可以得到 $t(x, y) - t(i_k, y)$ 关于 $y$ 单调不增，于是可以在 $O(\log n)$ 时间内用二分的方法求出 $y$ ，此时 $T = O(\log n)$ 。所以如果函数 $w$ 是凸的，那么这种动态规划问题最坏可以在 $O(n \log n)$ 时间内求解；最好时，可以在 $O(n)$ 时间内求解，可见这种方法是非常高效的。
- 然而，此种动态规划模型单一，对 $w$ 的限制有时又难以满足，所以应用范围也较为狭窄。但其思想是值得借鉴的。1




# 凸完全单调性的一个加强

- 假设我们要维护一个广义（不局限于动态规划问题）决策序列  $i_1, i_2, \dots, i_k$ ，满足  $i_1 < i_2 < \dots < i_k$ ，存在函数  $g$  和  $h$ ，满足在选取有关  $x$  的最优决策时，决策  $i$  不如决策  $j$  ( $i < j$ ) 等价于  $g(i, j) \leq h(x)$ 。
- 此时，决策仍然具有单调性，不过这不是关于  $x$  单调，而是关于  $h(x)$  单调。类比上面的知识，容易知道，这里维护的决策序列满足
$$-\infty = g(\text{nil}, i_1) < g(i_1, i_2) < \dots < g(i_{k-1}, i_k)$$
- 为了方便，我们记  $i_0 = \text{nil}$ 。则  $g(i_{k-1}, i_k) < g(i_k, i_{k+1})$ 。

# 凸完全单调性的一个加强

- 我们需要在这个序列上进行查找最小的 $j$ ，满足 $h(x) < g(i_j, i_{j+1})$ 。还需要在适当的时候添加一个候选决策。
- 函数 $g, h$ 的出现，使这个方法有了更大的弹性，所以这个决策序列的维护方法也多种多样，这些维护方法将在下一节中讨论。
- 在第一节介绍的 $w$ 凸的一类动态规划问题中，让 $h(x) = x$ ，容易证明 $g(i_{k-1}, i_k) = B[i_k]$ ，此方法就退化为第一节所讲方法。所以，利用凸完全单调性维护决策只是本方法的一个特例。



## 第二部分

对此加强的进一步分析

# 选取适当的 $h(x)$ 加速算法

- 例1 (CEOI'2004 Two Sawmills) 在一座山上分布着 $N$  ( $2 \leq N \leq 10000$ ) 棵树，由高到低依次编号为 $1, 2, \dots, N$ 。在山脚处还有一个伐木场。这 $N$ 棵树要被运往伐木场进行生产。每棵树只能从高处运往低处，而且这些树都分布在一条路旁。第 $i$ 棵树的重量为 $W_i$ ，它距离最山脚处的伐木场的距离为 $P_i$ 。运送每棵树的代价为它的重量与它到伐木场的距离的乘积。总代价为运送每棵树的代价之和。
- 伐木场的领导为了降低成本，决定在这条路旁的某两个位置再建两个伐木场。每棵树被运往不在它上方且离它最近的伐木场。现在要确定这两个新伐木场的位置，使得新的总代价最小，要求输出最小总代价。

# 选取适当的 $h(x)$ 加速算法

- 我们直接解决新建 $k$ 个伐木场的问题。
- 设  $S_i = \sum_{k=i}^n W_k$ ,  $T_i = \sum_{k=i}^n W_k P_k$ ,  $d[k, i]$  表示从 $i$ 到 $n$ 建造 $k$ 个伐木场的最小总代价。则

$$d[k, x] = \begin{cases} T_x & \text{当 } k = 0 \text{ 时} \\ \min \{d[k-1, i] + T_x - T_i - P_{i-1}(S_x - S_i) : x < i \leq n+1\} & \text{当 } k > 0 \text{ 时} \end{cases}$$

- 本题的答案就是 $d[k, 1]$ 。由于 $d[k]$  只与 $d[k-1]$  有关, 我们根据 $k$  来划分阶段。
- 假设 $U_i = d[k-1, i] - T_i + P_{i-1}S_i$ , 则

$$d[k, x] = \begin{cases} T_x & \text{当 } k = 0 \text{ 时} \\ \min \{U_i + T_x - P_{i-1}S_x : x < i \leq n+1\} & \text{当 } k > 0 \text{ 时} \end{cases}$$

# 选取适当的 $h(x)$ 加速算法

- 容易发现，这里的权函数是凸的，但是直接利用第一节的方法只能得到 $O(kn \log n)$ 的算法。利用第二节的方法可得，在计算 $d[k, x]$ 时

$$\text{决策 } j \text{ 比 } i \text{ 好} \Leftrightarrow \frac{T_j - T_i}{P_{j-1} - P_{i-1}} \leq S_x$$

- 所以令  $g(i, j) = \frac{T_j - T_i}{P_{j-1} - P_{i-1}}$ ， $h(x) = S_x$ 。
- 假设此时的决策序列为 $i_1, i_2, \dots, i_k$ ， $i_1 > i_2 > \dots > i_l$ ， $g(i_0, i_1) < g(i_1, i_2) < \dots < g(i_{l-1}, i_l)$ 。

# 选取适当的 $h(x)$ 加速算法

- 在计算 $d[k, x]$ 时，需要找到最小的 $j$ ，满足 $g(i_j, i_{j+1}) > S_x$ ，则 $d[k, x] = U_{i_j} + T_x - P_{i_j-1} S_x$ 。由于 $S_x$ 在的计算过程中是单调增的（按照 $x$ 从大到小的顺序计算），我们只需要接着上次向后查找，于是这 $n$ 次查找的总时间复杂度为 $O(n)$ 。
- 计算出 $d[k]$ 之后，我们需要建立一个决策序列供下一阶段选择。我们按照从大到小的顺序将这些候选决策加入决策序列。假设某个时刻决策序列为 $i_1, i_2, \dots, i_l$ ， $i_1 > i_2 > \dots > i_l$ ，此时要将 $x$  ( $i_l > x$ ) 添加到决策序列中。我们比较 $g(i_{l-1}, i_l)$ 与 $g(i_l, x)$ 的大小，如果 $g(i_l, x) \leq g(i_{l-1}, i_l)$ ，就删去决策 $i_l$ （令 $l \leftarrow l - 1$ ），继续这样的比较直到 $g(i_{l-1}, i_l) > g(i_l, x)$ ，最后再将 $x$  放到决策序列的末尾。

# 选取适当的 $h(x)$ 加速算法

- 容易看出，这是一个栈维护的问题。因为 $g$ 的计算是 $O(1)$ 的，所以整个过程的时间复杂度是 $O(n)$ 。
- 综上，每个阶段的计算是 $O(n)$ 的，一共有 $k$ 个阶段，故总的时间复杂度为 $O(kn)$ 。
- 对比这个方法与直接利用凸完全单调性的方法可以发现，直接利用凸完全单调性的方法实际上是在求出了 $g(i, j)$ 之后，用二分的方法求出了最小的满足 $g(i, j) \leq S_x$ 的 $x$ 。而这是完全没有必要的。如果 $w$ 本身是凸的，通过选择合适的 $h(x)$ （这里 $h(x) = S_x$ ），那么这个 $h(x)$ 与 $x$ 一定具有相同的单调性，这不会影响查找的总时间，而计算 $g$ 的复杂度由 $O(\log n)$ 降到了 $O(1)$ ，这就将维护决策序列时间复杂度从 $O(n \log n)$ 降到了 $O(n)$ ，从而提高了算法的效率。



# 用于解决一些权函数不具有凸完全单调性的问题

- 在刚才的例题中，这个方法只是扮演着优化的角色，并没有质的改变。下面，我们将利用这个方法解决一个权函数不满足四边形不等式的题目。
- 例2 (石阶)水平面上有个高度为 $h$ 的高台，你需要用 $n$  ( $n \leq 1000$ ) 块石块修建一个石阶，从这个高台通往地面。第 $i$ 个石阶的高度为 $h_i$ 。由于一些限制，你从高台出发，并且只能顺次处理每块石块，你要么将当前的石块摞在你前方的那一列，要么走到前方的那一列。在修建石阶的过程中你不能向回走。
- 为了让这个石阶不至于太单调，你希望这个石阶有所起伏，所以你将希望将这 $n$ 个石块都用上。为了让这个石阶安全可用，你希望相邻两列高度之差的最大值（视高台为第0列，地面为最后一列）尽可能小。也就是说，你需要将这 $n$ 个石块分成若干段，假设你将他们分成了 $k$ 段，则每段的高度为这段所有石块的高度总和。假设第 $i$ 段的高度为 $t_i$ ， $t_0 = h$ ， $t_{k+1} = 0$ 。你的目标就是选择一个合适的分段，来最小化 $\max_{i=0}^k |t_i - t_{i+1}|$ 。

# 用于解决一些权函数不具有凸完全单调性的问题

- 设 $h_0 = h, h_{n+1} = 0$ ,  $d[k, x]$ 为将石块0到石块 $x$ 分成若干段, 最后一段为石块 $k$ 到石块 $x$ 的方案中, 相邻两段高度差的最小值。设 $s_i = h_0 + \cdots + h_i$ ,  $diff[i, k, x] = |s_x - 2s_{k-1} + s_{i-1}|$ , 则

$$d[k, x] = \begin{cases} 0 & \text{当 } k = x = 0 \text{ 时} \\ \infty & \text{当 } 0 = k < x \text{ 时} \\ \min \{ \max(d[i, k-1], diff[i, k, x]) : 0 \leq i < k \} & \text{当 } 1 \leq k \leq x \leq n+1 \text{ 时} \end{cases}$$

- $d[n+1, n+1]$ 就是所求答案。

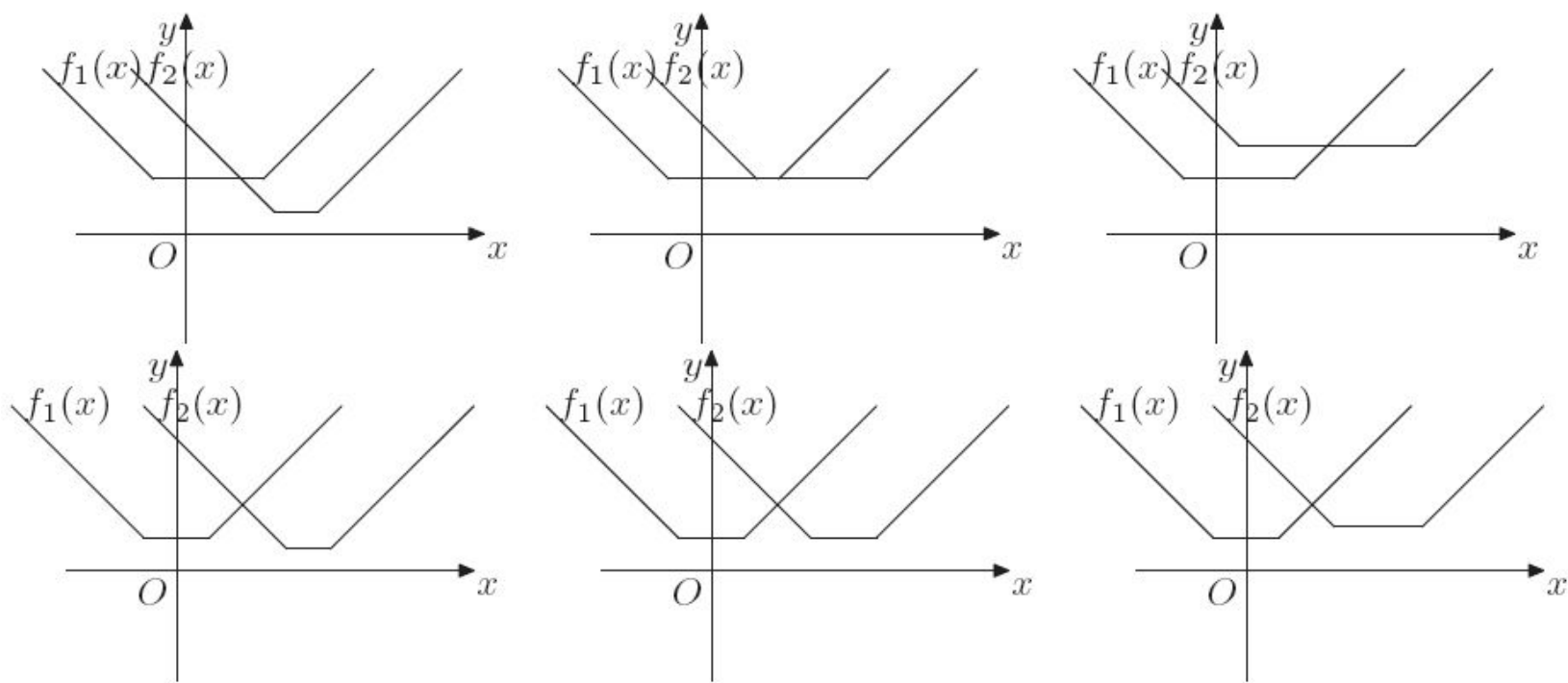
# 用于解决一些权函数不具有凸完全单调性的问题

- 这里，我们按照 $k$ 从小到大,  $x$ 从小到大的顺序计算 $d[k, x]$ 。我们需要维护 $n+1$ 个决策序列，第 $k$ 个决策序列存储 $d[i, k - 1]$ ，其中 $i$ 作为决策量。我们考察决策 $i$ 比决策 $j$ 更好的条件，以确定决策序列的各决策的序以及维护方法。
- 设 $h(x) = s_x - 2s_{k-1}$ ，则决策 $i$ 比决策 $j$  ( $i > j$ )好等价于
$$\max(|h(x) + s_{i-1}|, d[i, k - 1]) \leq \max(|h(x) + s_{j-1}|, d[j, k - 1])$$

# 用于解决一些权函数不具有凸完全单调性的问题

- 容易发现，如果 $h(x)$ 足够大，那么此式一定不成立。我们考察这类不等式的解集。考虑函数 $f_1(x) = \max(|x + a_1|, b_1)$ ， $f_2(x) = \max(|x + a_2|, b_2)$ 。这里 $a_1 > a_2$ ，但 $b_1$ 、 $b_2$ 间没有必然的大小关系。则这两个函数间可能有以下几种关系：

# 用于解决一些权函数不具有凸完全单调性的问题



# 用于解决一些权函数不具有凸完全单调性的问题

- 在图中我们可以看到,  $f_1(x) \leq f_2(x)$  的解集一定是  $x \leq g(a_2, b_2, a_1, b_1)$ 。故决策  $i$  比决策  $j$  ( $i > j$ ) 好等价于
$$h(x) \leq g(s_{j-1}, d[j, k-1], s_{i-1}, d[i, k-1])$$
- 简写为  $h(x) \leq g(j, i)$ 。这里  $g$  是可以在  $O(1)$  时间内求出的。

# 用于解决一些权函数不具有凸完全单调性的问题

- 所以我们维护的每个决策序列中的决策应当满足：

$$\begin{aligned} \text{nil} &= i_0 < i_1 < i_2 < \cdots < i_l \\ -\infty &= g(i_0, i_1) > g(i_1, i_2) > \cdots > g(i_{l-1}, i_l) \end{aligned}$$

- 查找决策时，应当找到最大的 $j$ ，满足 $h(x) \leq g(i_{j-1}, i_j)$ 。因为 $h(x) = s_x - 2s_{k-1}$ 关于 $x$ 单调增，我们可以接着上次向前查找，所以总的查找时间为 $O(n)$ 。
- 维护序列时，由于 $d[k, x]$ 是按照 $k$ 从小到大计算的，新的决策一定是被追加到决策序列的末尾，所以只需要反复比较 $g(i_{l-1}, i_l)$ 与 $g(i_b, k)$ ，删去序列的末尾决策直到满足 $g(i_{l-1}, i_l) > g(i_b, k)$ ，最后将 $k$ 追加到决策序列的末尾。

# 用于解决一些权函数不具有凸完全单调性的问题

- 这样，我们就得到了这个题目的 $O(n^2)$ 时间复杂度的解法。
- 这个题目中， $f_2(x) - f_1(x)$ 并不一定是单调增的，也就是权函数并不具有凸完全单调性，但我们还是利用这个方法解决了这个问题。其原因在于，我们利用的只是决策的单调性，并不需要在意是哪个具体的性质造成的这种单调性。对权函数要求过高，这便是凸完全单调性的局限，为一个不难达到的目的付出了过多的代价。



# 合理的序与维护方向

- 上面两个例子，尤其是例2，向我们展示了这种方法的灵活性。动态规划的方程并不像第一节中的那么死板，也并不只有一个决策序列，决策的添加顺序也没有固定的要求。一个决策甚至可以被加入多个决策序列！（但不能太多，否则就与动态规划的本质与维护决策的最终目的背道而驰；当然，在例2中并未出现此情况）然而，解决动态规划问题只是此类方法的一部分，此类方法还可以解决更为宽泛的问题。

# 合理的序与维护方向

- 例3 (经典问题) 要求维护一些不与 $y$ 轴平行的直线，为了方便，初始时只有一条直线 $y = \infty$ 。每次要么添加一条直线 $y = kx + b$ ，要么询问 $x = k$ 与这些直线的最低交点的坐标。假设询问的次数为 $q$ ，直线的条数为 $n$ ，请给出一个 $O((n + q) \log n)$ 的算法。
- 假设第 $i$ 条直线为 $y = k_i x + b_i$ 。考虑何时直线 $i$ 不如直线 $j$ 好。显然，这等价于

$$k_i x + b_i \geq k_j x + b_j \Leftrightarrow b_i - b_j \leq (k_i - k_j)x$$

但由于 $k_i - k_j$ 的正负不确定，所以无法继续变形下去。

# 合理的序与维护方向

- 回顾决策序列的维护过程，可以发现，此方法并没有对决策的顺序作任何显式的要求，只要求决策满足：

$$\text{决策}i\text{不如决策}j \Leftrightarrow g(i, j) \leq h(x)$$

- 回顾前两个例题可以发现，决策的顺序恰恰是暗含在函数 $g$ 中的。

# 合理的序与维护方向

- 在本题中为了得到 $g(i, j) \leq h(x)$ ，我们可以要求 $k_i \geq k_j$ 。这样就得到了一个合适的序。有了这个序，我们就可以得到：

$$\text{决策} i \text{ 不如决策 } j \Leftrightarrow \begin{cases} \frac{b_j - b_i}{k_i - k_j} \leq x & \text{当 } k_i \neq k_j \text{ 时} \\ -\infty \leq x & \text{当 } k_i = k_j, b_j \leq b_i \text{ 时} \\ \infty \leq x & \text{当 } k_i = k_j, b_j > b_i \text{ 时} \end{cases}$$

- 即 $h(x) = x$ ,

$$g(i, j) = \begin{cases} \frac{b_j - b_i}{k_i - k_j} & \text{当 } k_i \neq k_j \text{ 时} \\ -\infty & \text{当 } k_i = k_j, b_j \leq b_i \text{ 时} \\ \infty & \text{当 } k_i = k_j, b_j > b_i \text{ 时} \end{cases}$$

# 合理的序与维护方向

- 假如我们维护了一个决策序列  $i_1, i_2, \dots, i_l$ , 满足  $k_{i_1} < k_{i_2} < \dots < k_{i_l}$ ,  $-\infty = g(\text{nil}, k_{i_1}) < g(k_{i_1}, k_{i_2}) < \dots < g(k_{i_{l-1}}, k_{i_l})$
- 回答询问时, 只要在这个序列中找最大的  $j$ , 满足  $g(i_{j-1}, i_j) \leq x$ 。则所求交点一定在直线  $i_j$  上。
- 插入新直线时, 由于这条直线并不一定是斜率最小的直线, 所以我们不能在决策序列的末尾添加这个决策, 而是要根据这条直线的斜率来决定是在开头还是中间还是末尾来维护这个序列。为了使算法描述尽可能地简洁, 我们视序列的开头和结尾为一种退化了的中间。
- 假设  $k_{i_j} < k_n \leq k_{i_{j+1}}$ , 我们可以用类似栈维护的方法删去前面不需要保留的决策。假设这个时候  $n$  前方的决策变成了  $i_j$ , 我们比较  $g(i_j, x)$  与  $g(x, i_{j+1})$  来判断决策  $n$  是否需要保留。如果需要保留, 这时有可能  $g(n, i_{j+1}) \geq g(i_{j+1}, i_{j+2})$ , 那么决策  $i_{j+1}$  不需要被保留, 删去决策  $i_{j+1}$ , 再判断决策  $i_{j+2}$  是否不需要被保留。如此操作直到不需要删除决策为止。

# 合理的序与维护方向

- 容易知道，平衡树可以在 $O(\log n)$ 的时间内求前驱、后继，查找最大的不超过 $h(x)$ 的元素。因为每次插入新的决策只需要进行均摊 $O(1)$ 次操作（可用类似于栈维护的方法证明），所以维护序列的总时间是 $O(n \log n)$ 的。回答询问的总时间是 $O(q \log n)$ 的。所以总时间复杂度就是 $O((n+q) \log n)$ 的。
- 值得一提的是，只要对这个算法稍加改变，就可以在 $O(n \log n + \text{输出规模})$ 的时间内解决在线半平面交的问题，因为我们计算出的 $g$ 恰好就是这些直线组成的最低折线上的转折点。
- 虽然这种方法可以高效处理从中间维护决策序列的情况，但毕竟平衡树操作的常数因子不小，如果可以只在两头维护序列，就只需要用栈来维护，程序效率会大大提高。

# 合理的序与维护方向

- 考虑离线半平面交问题，给出 $n$ 个二元一次方程 $Ax + By \leq C$ ，求出可行域。我们只考虑这个问题的关键步骤：给出一些直线 $y = kx + b$ ，求出最低的一条折线（显然这是上凸的）。
- 由于最终的结果与处理直线的顺序无关，我们先将这些直线按照 $k$ 从大到小排序，然后应用上面的方法。这时，这些“决策”的插入过程中，只需要从序列的尾部维护，这个操作用栈就可以高效的完成。最后剩下的“决策”就是按照从左到右顺序的最低折线，而相邻两“决策”间的 $g$ ，就是这些折线交点的横坐标。排序之后的这些操作，其时间复杂度总计 $O(n)$ 。
- 这个几何问题，我们甚至没有画一张图，就用这种方法在 $O(n \log n)$ 的时间内高效的解决了。而算法效率的瓶颈是一开始的排序，如果排序可以在 $O(n)$ 时间内完成，或者直线就是按斜率从大到小或斜率从小到大的顺序给出，那么此方法的时间复杂度就是 $O(n)$ 。这足以说明此方法更应该被作为一种思想来看待。

# 合理的序与维护方向

- 例4 (USACO JAN07 schul) 要求维护一些不与y轴平行的直线。每次要么添加一条直线 $y = kx + b$ ，要么询问当 $x = p$ 与这些直线的所有交点中的纵坐标的最小的点。保证每条直线的斜率大于0，并且新加直线的横截距 $-b/k$ 比原来的直线的横截距都大。并且保证每次询问的 $p$ 总比最大横截距大。
- 由于直线并不是按照斜率递增或递减的顺序添加的，使用上面的方法只能得到 $O((n+q)\log n)$ 的算法，而对于本题的数据，这个算法虽然比 $O(qn)$ 的方法快了很多，但还是很慢。
- 我们不妨再仔细分析一下题目特点，按顺序将每条直线假如决策序列会怎样呢？
- 直线 $i$ 比直线 $j$  ( $i < j$ ) 优等价于

$$k_i x + b_i \leq k_j x + b_j \Leftrightarrow b_j - b_i \leq (k_j - k_i)x$$

- 当 $k_j > k_i$ 时，这等价于 $\frac{b_j - b_i}{k_j - k_i} \leq x$ ；当 $k_j \leq k_i$ 时，这等价于 $\frac{b_j - b_i}{k_j - k_i} \geq x$ ，然

而由于左边不大于最大横截距，而这个问题又保证每次询问的 $x$ 大于最大横截距，所以这个条件等价于  $x \leq \infty$  !



# 合理的序与维护方向

- 这样，我们就得到了一个合适的函数 $g(i, j)$ ，以及一个便于处理的序，这个序恰好就是直线插入的顺序。
- 所以这个问题就被转化为一个普通的栈维护问题，由于这个序列的维护方法已经在例2中给出，这里不再赘述。
- 这样，我们就在 $O(n + q)$ 的时间解决了这个问题。
- 而解决这个问题的关键是：有些结论虽然在全局并不成立，但在某个具有更多限制的局部内却可以满足。所以我们要充分利用题目特点，找到最适合的序，这个序有时就是以这个特殊的局部内的某些性质为基础的。

# 最优决策的查找方式

- 当这个决策序列是使用栈在两头维护时：
  - 如果 $h(x)$ 单调增或单调减，则可以利用这个单调性接着上次查找，在 $O(n + q)$ 时间内回答所有询问，如果这个 $h(x)$ 的反复次数不多（例如凸等），也可以用接着上次查找的方法，其时间复杂度为 $O(|ans_1 - ans_2| + |ans_1 - ans_2| + \dots + |ans_{q-1} - ans_q|)$ 。
  - 如果这个 $h(x)$ 的变化很不规律，则可以用二分查找的方法在 $O(q \log n)$ 时间内回答所有询问。
- 当这个决策序列是从中间维护时，由于使用了平衡树，我们也只能用平衡树上的查找方法。
- 相比之下，从中间维护决策序列无论是从决策的维护还是最优决策的查找，都有很大劣势，我们应当尽量避免（寻找适合的序）。

# 选择合适的规划方向

- 例5(旅行) 一条铁路线上有 $N$ 个城市，顺序编号为 $1, 2, \dots, N$ ，TT希望从城市1到城市 $N$ 。从城市 $i$ 出发只能到达后面的某个点 $j$ ，需要花费 $(j-i)A_i$ 的车票费与 $B_j$ 元的检票费。TT想知道从城市1出发到城市 $N$ 所需要的最少花费。
- 假设 $d[i]$ 为从城市1到城市 $i$ 所需的最小费用，则
$$d[x] = \min \{d[i] + (x-i)A_i + B_x : 0 \leq i < x\}$$
- 决策 $i$ 不比决策 $j$  ( $i < j$ )等价于
$$d[i] + (x-i)A_i + B_x \geq d[j] + (x-j)A_j + B_x$$
$$\Leftrightarrow d[j] - jA_j - d[i] + iA_i \leq x(A_i - A_j)$$
- 于是选择 $A_i$ 作为决策的序，就转化为使用平衡树从中间维护决策序列的问题。根据上面的讨论，这个算法的时间复杂度是 $O(n \log n)$ 的。但由于使用平衡树来维护，这个算法的常数因子不小。

# 选择合适的规划方向

- 但是，如果定义 $d[i]$ 为从车站 $i$ 到车站 $N$ 所花的最少代价，则

$$d[x] = \min \{d[i] + (i - x)A_x + B_i : x < i \leq n\}$$

- 此时，决策 $i$ 不如决策 $j$  ( $i > j$ )等价于

$$d[i] + (i - x)A_x + B_i \geq d[j] + (j - x)A_x + B_j$$

$$\Leftrightarrow \frac{d[j] + B_j - d[i] - B_i}{i - j} \leq A_x$$

- 这时，我们可以按处理顺序在决策序列末尾维护决策，这样只需要用栈就可以维护决策序列。在查找最优决策的时候，只需要使用二分查找的方法，就可以在 $O(n \log n)$ 时间内解决此问题。
- 虽然这个方法与上个方法具有一样的理论时间复杂度，但由于栈维护和二分查找相对于平衡树的常数因子非常小，在实践中，这个方法相比上个方法存在很大优势。

# 事先除去无用状态以获得合适的序

- 在上个问题中，如果令 $B_i = 0$ ，这个题目就可以很容易的用贪心算法解决，其时间复杂度是 $O(n)$ 的。
- 在这个特殊情况下，能否再进行一些优化来改进这个方法呢？答案是肯定的。
- 考虑这个题的贪心算法：如果当前车站的 $A_i$ 更小，那么就在这里倒车，直到到达城市 $N$ 。也就是说，决策序列中的 $A_i$ 是单调减的。
- 然而，对刚才的方法稍加分析，可以发现，有用的决策并不一定满足这个条件。因为我们计算了大量无用 $d[i]$ ！这些 $d[i]$ 不会作为以后的可行决策，但是为了计算它们，我们不得不使用一个更差的决策序列，或者不得不使用一种更差的最优决策的查找方法。

# 事先除去无用状态以获得合适的序

- 所以我们要事先知道哪些决策是无用的。假设 $d[i]$ 是从车站1到车站 $i$ 的最小花费。且 $d[i] = d[j] + (i-j)A_j$ 。则决策 $i$ 比决策 $j$ 好等价于

$$d[i] + (x-i)A_i \leq d[j] + (x-j)A_j \Leftrightarrow A_i \leq A_j$$

- 于是在这个问题中，任何时候决策序列中实际上只有一个决策，所以这个修改过的动态规划与贪心几乎没有区别了，可谓是殊途同归。

# 事先除去无用状态以获得合适的序

- 例6(Balkan OI'2003 欧元) 将一个由 $n$ 数组成的数列划分为若干段, 设前 $i$ 个数的和是 $s_i$ , 如果一段是从第 $i$ 个数到第 $j$ 个数, 那么这段的权值是 $(s_j - s_i)j - T$  ( $T > 0$ ), 求出一个权和最大的划分。
- 设 $d[i]$ 为从第1个数到第 $i$ 个数的权值最大的划分。则
$$d[x] = \max \{d[i] + (s_x - s_i)x - T : 0 \leq i < x\}$$
- 决策 $i$ 不如决策 $j$  ( $i < j$ ) 价等于
$$d[i] + (s_x - s_i)x - T \leq d[j] + (s_x - s_j)x - T$$
$$\Leftrightarrow d[i] - d[j] \leq x(s_i - s_j)$$
- 由此我们可以确定决策应当以 $s_i$ 从小到大为序。这样使用平衡树从中间维护决策序列, 就可以在 $O(n \log n)$ 时间内解决此问题。

# 事先除去无用状态以获得合适的序

- 但是，我们仍然求出了一些无用状态！
- 可以证明，如果决策序列的最后一个决策为 $j$ ，则决策 $i$ 只有当 $s_i$ 小于 $s_j$ 时才可能有用。这时，我们维护的决策序列中的决策恰好满足 $s_{i_k} > s_{i_{k+1}}$ 。于是刚才的方法实际上只用在序列的末尾进行维护，于是程序的时间复杂度就降到了 $O(N)$ 。



# 第三部分 总结

- 从上面的讨论可以看出，凸完全单调性的这个加强中函数 $g$ 和 $h$ 的出现带来了很大的灵活性，不仅可以用来加速求解权函数本身就凸的情形，还可以解决权函数并不凸的形式，而且其应用也不仅限于动态规划问题。
- 维护决策序列，更应当被看作一种思考问题的方法。

谢谢大家！