ADA Hw2 p5-6

Problem 5 Toyz's Dog

Subproblem (a)

(1)

if $i > j$

$$dp(i,j) = \begin{cases} dp(i-1,j) + E(i-1,i), & \text{if } |i-j| > 1 \\ min_{0 \le i' < j}(dp(i',j) + E(i',i)) & \text{if } |i-j| = 1 \end{cases}$$

else if $j > i$

$$dp(i,j) = \begin{cases} dp(i,j-1) + E(j,j-1), & \text{if } |i-j| > 1 \\ min_{0 \le j' < i}(dp(i,j') + E(j,j')) & \text{if } |i-j| = 1 \end{cases}$$

else

$$dp(i,i) = (\text{Not Define})$$

我們先從定義轉移式開始，題目已定義說$dp(i,j)$ 為 $\text{minimal cost of the trip} - (E(i,N) + E(N,j))$，且比$max(i,j)$小的石頭都已經是最佳的位置，因此我們最需要確定的是假定目前狀態是最佳的情況(OPT)，這狀況的子問題也必定是最佳狀況。從這份定義出發我們可以先將case分為兩大部分，$i > j$ 與 $i < j$，會這樣分是因為在處理石頭的過程中，將同樣的石頭放在去程以及放在回程是不同的意義，因此需要分開討論，而$dp(i,i)$並沒有被定義，因為同一顆石頭無法同時成為去程與回程最靠近S_N的石頭。

我們在這邊先假設 $dp(i,j)$ 是OPT：
在開始討論各項case之前，我們需要定義此遞迴式的base case，若我們最少一定一顆石頭的話，最基本的一定就是 $dp(1,0)$ (第一顆石頭放去程)，與 $dp(0,1)$ (第一顆石頭放回程)了。 $dp(1,0)$ 的值是 $E(1,0)$，而 $dp(0,1)$ 的值會是 $E(1,0)$ 。

有了base case後，我們先討論 $i > j$ 。在這情況，我們觀察到說若 i 與 j 的差額大於1，那 $dp(i, j)$ 只能從$dp(i-1, j) + E(i-1, j)$ 得到，這代表 $dp(i-1, j)$ 是 $dp(i, j)$ 的唯一子問題，我無法從其他狀態轉移到這狀態(例如 $dp(i, j-1)$)，因 $j-1 < max(i, j-1) = max(i, j)$ ，所以 $j-1$ 已經是在最佳的位子上了，我若更動其位子，意義上他就不是$dp(i, j-1)$的optimal substructure了。而因 $dp(i, j)$ 與 $dp(i-1, j)$ 只差一個常數 $E(i-1, i)$ ，若 $dp(i-1, j)$ 不是optimal substructure的話，$dp(i, j)$ 也不會是OPT，而這跟我的假設矛盾，因此此dp轉移式在 $|i-j| > 1$ 的狀況下必定是OPT。

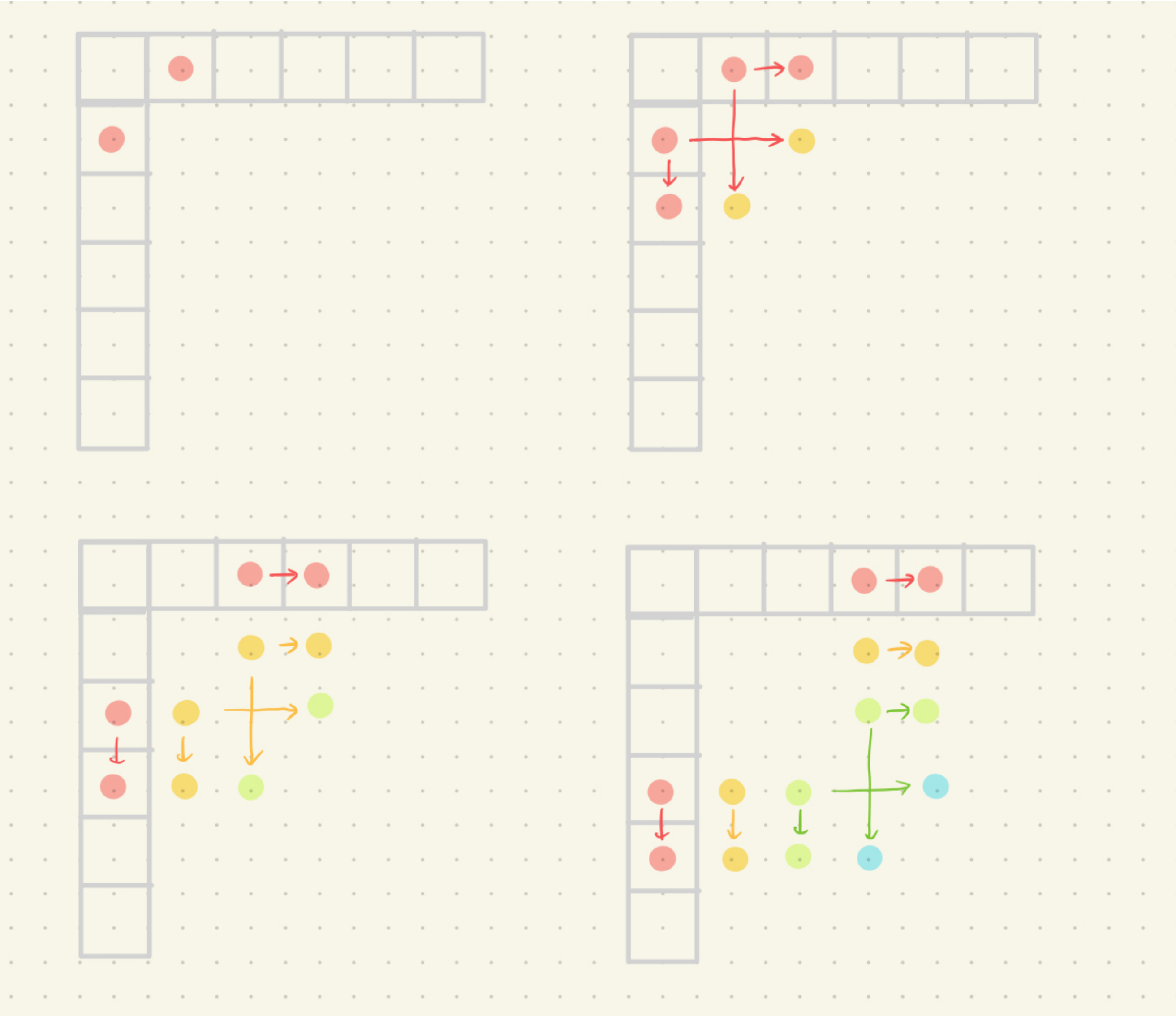
然而這狀況在 $|i-j| = 1$ 的時候需要多做討論，因為若按照原本的轉移式，$dp(i, i-1) = dp(i-1, i-1) + E(i-1, i)$ ，但 $dp(i-1, i-1)$ 並沒有定義，所以他只能從固定j，並且從 $i < j$ 的OPT裡面尋找最佳的來轉移。而 $dp(i', j)$ 總共有 j 種可能 $(i' = 0\text{~}(j-1))$，我必須枚舉這些OPT並且轉移來判斷哪一個 $dp(i', j)$ 會是 $dp(i, j)$ 的optimal substucture。因為我所有的子結構都是OPT，只要我選擇一個轉移後最小的存進 $dp(i, j)$ ，那 $dp(i, j)$ 本身也必定是OPT，符合原本的假設。

同樣的情況也適用於 $j > i$ ，將這個case分開來討論是因為，當 $j > i$且$|j-i| > 1$ ，我在討論 $dp(i, j)$ 將S_j安排於回程會類似於上面講的$|i-j| > 1$，都只有一種OPT($dp(i, j-1)$)能夠轉移，然而他所轉移的E值會是$E(j, j-1)$，為遞減順序，因此分開討論。而當 $|j-1| = 1$ 時，也需要枚舉 $dp(i, j')\text{for } 0 \leq j' < i$ 的OPT來比較最小值。同樣的因為子結構一定是OPT，所以藉由這種方式得到的dp也會是OPT。

而這轉移式的時間複雜度，$|i-j| > 1$ 的case是單純加法，$\mathcal{O}(1)$轉移，而 $|i-j| = 1$ 需要枚舉 0至i-1或j-1的case，所以是$\mathcal{O}(n)$，整體加起來，$\mathcal{O}(1)$的case有N^2個，$\mathcal{O}(n)$的case有2N個，所以整體複雜度是$N^2 \times \mathcal{O}(1) + 2N \times \mathcal{O}(N^2) = \mathcal{O}(N^2)$。

(2)

ref: b07207063 廖政華



這一題若單純按照dp式填表的話，其空間複雜度是$\mathcal{O}(N^2)$，然後最終解會是從 $dp(i', N-1) + E(i', N) + E(N, N-1) \text{ for } 0 \leq i' < N-1$ 與 $dp(N-1, j') + E(N-1, N) + E(N, j') \text{ for } 0 \leq j' < N-1$ 中選一個最小值來輸出。然而事實上我們只需要兩條N大小的陣列就可以完成這件事了。

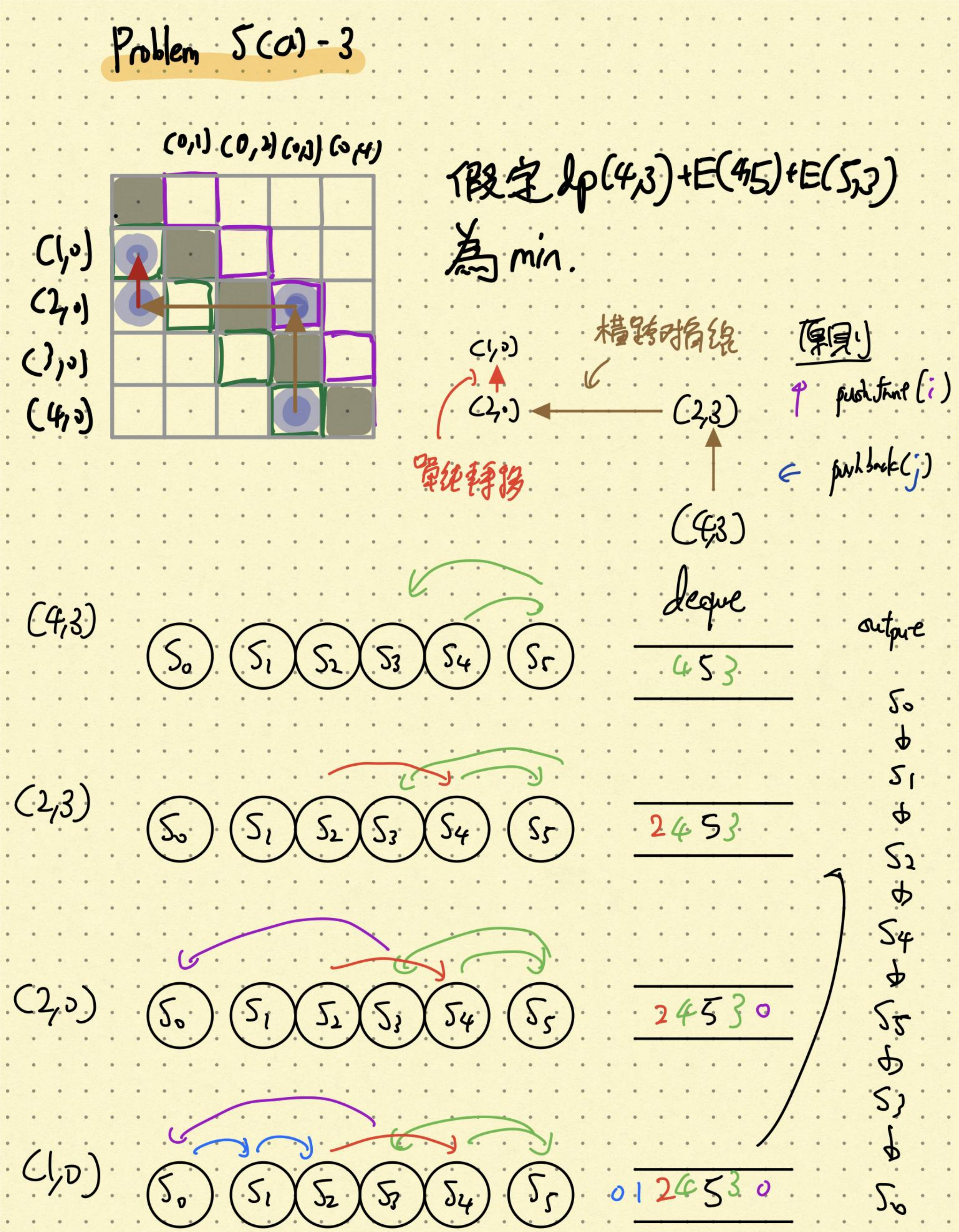
我們的觀察是，定義的遞迴式中， $dp(i,j)$ 只與他在table中左方與上方的dp值相關(如圖)，若 $|i-j| > 1$，那dp只與i-1或j-1的dp式有關；而如果$|i-j| = 1$，以$i > j$為例，此dp值也只與 $dp(i', j) \text{ for } 0 \leq i' < j$有關。在圖表中，計算dp值都不會使用到斜上方的區域，利用此性質，我們規劃出以下算法：

1. 首先先new出兩條array，在兩條index=0的地方，放上base case $dp(0,1)$ 和 $dp(1,0)$的值。而在推演dp值的過程中，我們使用$max(i,j)$這個變數來斷定目前陣列是在哪個狀態，我們會從$max(i,j) = 1$推到$max(i,j) = N-1$。圖上方的橫條陣列代表的是$dp(i, max(i,j))$，直條陣列代表的則是$dp(max(i,j), j)$。
2. 我們每往下推一階層的時候，都要將max(i,j)+1，並觀察說格子的狀態屬於哪一個dp轉換式。舉例來說原本我們存$dp(0,1)$的位置，就會變成$dp(0,2)$，並且他符合$j > i$且$|j-i| > 1$的狀況，所以$dp(0,2) = dp(0,1) + E(2,1)$；那如果是$dp(1,2)$的位置，他就符合$j > i$且$|j-i| = 1$的狀況，那他就要遍歷$j' < i$的所有狀況，並從中轉移中取得min值填入。需要注意的是，在更新此種狀況時，我們須將dp(i,j')的值先更新才能進行計算，不然裡面的dp值會是錯誤的狀態。
3. 照著step 2的方式將$max(i,j)$更新到N-1時，我們也基本上將所有OPT遍歷並且集合成最終的狀態了，最後只要從陣列取$dp(0, N-1)$到$dp(N-2, N-1)$以及$dp(N-1, 0)$到$dp(N-1, N-2)$這些狀態中各自加上$(E(i,N) + E(N, N-1))$與$(E(N-1, N) + E(N, j))$後取最小值，這個答案就會是minimum cost of the trip。

在這樣的流程我們只需要兩條$\mathcal{O}(N)$陣列，因此空間複雜度為$\mathcal{O}(N)$。

(3)

ref: b07207063 廖政華



取得路徑同樣只需要兩條$\mathcal{O}(N)$的陣列就可以算。這兩條陣列在Table上的位置是兩條紫色與綠色的$|i-j=1|$對角線。在這兩條陣列中，它代表著在那次轉移中，哪個dp值是他的parent，存下那個parent的座標後就可以進行backtrace。

舉例來說，紫色的陣列代表$|j>i|=1$，所以他會存$dp(0,1)$，$dp(1,2)$，$dp(2,3)$...的parent。其中$dp(0,1)$與$dp(1,0)$為base case，所以當路徑回溯到那邊演算法就停止，只需要將 S_0補進相對應的起點或終點即可。

因此在(2)的演算法裡，我們需要加上一條步驟，每次在$|i-j|=1$的case裡比較完min之後，需要將parent的座標記進對應的對角線陣列。

我們利用deque來存最終的路徑答案，並且利用push_front()與push_back()來進行更新，push_front()代表將去程的石頭記錄起來，push_back()則是記錄回程的石頭。

有了以上基礎後，建構路徑的基本算法如下：

1. 初始化一條deque，並且在裡面push N 這個點，因每一條路徑都一定會踩到S_N。
2. 從最後判斷最終答案的式子裡挑出最小的$dp(i,j)$值，由那個$(i.j)$作為回溯的起點。並且將i與j分別push_front()與push_back()進deque。
3. (a) 假設$dp(i,j)$的$|i-j|>1$，代表他的轉換式是唯一的，我可以藉由dp轉換式推回其parent並且照定義push進deque裡。若$i>j$，我就將i' for $i>i'>j$依序push_front()進deque裡，直到$|i'-j|=1$為止。
   (b)若$dp(i,j)$的$|i-j|=1$，代表其轉換式的parent並非唯一，我們必須查我們先前的存的對角線座標陣列裡來看他的parent是誰，並且將現在關注的$dp(i,j)$更新成parent的值後push其i或j進deque裡。
4. 重複 step(3)直到dp值的座標更新成$dp(0,1)$或$dp(1,0)$，若最終值是$dp(1,0)$就將S_0 push_front()進deque；相反的如果是$dp(0,1)$就將其push_back()進deque。
5. 最終將deque裡的元素從index 0依序pop出來，就會是路徑答案了。

以上圖為例，我最終答案是由$dp(4,3)$取得，因此我先將S_4與S_3 push_front()與push_back()進deque，而$dp(4,3)$符合$i-j=1$，所以我需要查綠色陣列來看其parent是誰。他假定他的parent是$dp(2,3)$，我就將dp值更新成$dp(2,3)$並且將S_2 push_front()進deque(因為只有i值被更新)。之後$dp(2,3)$符合$j-i=1$，因此我查紫色陣列來看其parent是誰。假定parent為 $dp(2,0)$，我就將S_0 push_back()進deque裡(因為只有j值被更新)。$dp(2,0)$符合$i-j>1$，因此我減少 i 並且依序將i值push_front()進deque，最後dp座標為$(1,0)$，演算法結束並且將S_0 push_front()進deque。

而最終的路徑就會是 $S_0->S_1->S_4->S_5->S_3->S_0$。

以上算法只多用到兩條$\mathcal{O}(N)$對角線陣列，因此空間複雜度依舊為$\mathcal{O}(N)$，回溯本身時間複雜度也只需$\mathcal{O}(N)$。

Subproblem (b)

(1)

此題我們可以視為10背包問題的變形，在這個例子裡，去程的石頭為物件本身，石頭的體積為D_i，背包的總量為血量H，物件的價值為cost of the trip(也就是上題定義的dp)，每次對每個石頭我只有放在去程或不放在去程兩個選項。有了這個觀點後，我們從背包問題的角度看待這題。

這題的$dp(i, j, h)$定義為在h的血量中前max(i,j)件石頭所需花費的最小花費(在實作上或許需要特意將cost轉成負值來實現背包問題的最大價值)；同樣的，在$dp(i, j, h)$裡，我們已經固定S_1~$S_{max(i,j)}$的位置。

接著不同於上一題的看法，我們從關注此dp值的來源，轉換成此dp值接下來的轉換途徑。我們假設此dp式為OPT，此dp定義式的base case為 $dp(0, 0, H) = 0$(不取任何石頭，也不會有任何傷害值)。

$\forall i, j \in \mathcal{N}, 0 \le i, j < N:$

$\text{if } i > j, \text{then } i + 1 \text{ would be the max element we focus on:}$

$\text{We could simply place the max element at the farthest part of the departure path or the return pat}$

$dp(i + 1, j, h - D_{i+1}) = dp(i, j, h) + E(i, i + 1) => \text{departure case}$

$dp(i, i + 1, h) = min(dp(i, i + 1, h), dp(i, j, h) + E(i + 1, j)) => \text{return case}$

$\text{if } j > i, \text{then } j + 1 \text{ would be the max element we focus on:}$

$dp(i, j + 1, h) = dp(i, j, h) + E(j + 1, j) => \text{return case}$

$dp(j + 1, j, h - D_{j+1}) = min(dp(j + 1, j, h - D_{j+1}), dp(i, j, h) + E(i, j + 1)) => \text{departure case}$

$\text{if any } h - D_x \le 0 \; \forall x \in \mathcal{N}, 0 < x < N \text{ , return } \infty$

$dp(i, i, h) \text{ is not defined.}$

(2)

此方法的正確性我們以數學歸納法證明，

1. base case: $dp(1,0,H-D_1)=E(0,1)$ if $H-D_1>0$ 與 $dp(0,1,H)=E(1,0)$，前者因去程有S_1因此要扣D_1血量，而後者將S_1置於回程，無須扣血。兩者皆只有S_1，故cost只有$E(0,1)$或$E(1,0)$，若$H-D_1\leq 0$，$dp(1,0,H-D_1)=\infty$，得證。
2. 我們假定$dp(i,j,h)$為OPT，代表在$dp(i,j,h)$裡，在h的血量裡，S_1到$S_{max(i,j)}$裡都已經被放在會讓cost最低的位置。
3. 在$dp(i,j,h)$為OPT的狀態，根據ij大小，最多會有四種轉換式，$dp(i+1,j,h-D_{i+1})$，$dp(i,i+1,h)$，$dp(i,j+1,h)$與$dp(j+1,j,h-D_{j+1})$。

我們假設$dp(i,j,h)$為當下的最佳狀態，而這時我們選擇$S_{max(i,j)+1}$的放置地點，總共只會有兩個狀況，將$S_{max(i,j)+1}$放置於去程或者是放置於回程。在此時與上題一樣，將case分為 $i>j$ 與 $i<j$，$dp(i,i,h)$的狀態一樣沒有定義，因為同樣的無法將同一顆石頭同時放置於去程和回程。

若$i>j$，此時將$S_{max(i,j)+1}$放置於去程，此狀態必定為單一轉換，因$dp(i,j,h)$中，我們已固定比$S_{max(i,j)}$前面的石頭位置，因此將$S_{max(i,j)+1}$放置於去程也會是OPT，而因為將石頭放置於去程需要扣掉生命值，因此定義式須將$h-D_{i+1}$才是正確的轉換式。若將石頭放置於回程，則轉換式會變成$dp(i,i+1,h)$，同樣的若是從$dp(i,j,h)$轉換也是單一轉換。然而我們不難發現在此定義裡j為非固定值，因此我們需要遍歷所有j' s.t. $0\leq j'<max(i,j)$才能決定此$dp(i,i+1,h)$的值，因此在定義式裡，我們須與$dp(i,i+1,h)$本身的值做比較，若由此$dp(i,j,h)$轉換的值比原值小，則做更新的動作，因我們將石頭放置於回程，所以h值不受影響。

同理可證，若$i<j$，將$S_{max(i,j)+1}$放置於回程也是單一轉換。而將其放置於啟程，我們也需要考慮其他也可能轉換至$dp(j+1,j,h-D_{j+1})$的狀態，因此也需要與本身的值做最小值比較才可以維持其OPT正確性。

當在轉換過程的h值小於等於0，代表爆炸值大於生命值，此狀態無法再繼續前進，因此設定為∞，代表無限的cost，此狀態不會是OPT。而最後同樣的，dp(i,i,h)沒有定義，因為無法將同一顆石頭同時放置於去程與回程。

根據上述，我們可證明說當dp(i,j,k)為OPT時，根據ij大小，他所轉移的四個狀態皆是OPT，因此根據數學歸納法，由此dp轉移式得到的$dp(N-1,j',H)$ for $0\leq j'<N-1$ 與
$dp(i',N-1,H)$ for $0\leq i'<N-1$ 皆會是OPT，並且由此解算出的包含終點跳躍cost的值會是最佳解。

其時間複雜度為$\mathcal{O}(N^2H)$，因我們須遞迴將i,j,h遍歷才能得出最佳解。

Problem 6 - Howard's Desiring Order of Course

(1)

If no assumption: 98141210
with assumption 3: 981120

(2)

利用merge sort排序後串接就會是maximum satisfying value。

證明：

1. 假設此作法A所產出的答案S非最佳解，那麼就存在真正的最佳解S'，且$S' > S$。
2. 假設有兩個CoursesC_1C_2的preferenceP_1P_2，P_1<P_2，既然最優解$S' \neq S$，所以S'裡肯定存在P_1在P_2前面的，也就是逆序數對。
3. 現在我們將P_1與P_2對換，獲得方案S_1'。
4. 我們假設最終的maximum satisfying value的位數為N個，最高位的位數為D_1，個位數的位數為D_N，則maximum satisfying value 可以為 $\sum\limits_{x=1}^{N} 10^{x-1} \times D_x$。利用此定義，在$S'$裡必定存在$P_1 \times 10^x + P_2 \times 10^{x-k}$，$x$與$k$皆為正整數且$k < x$。$S_1'$則包含$P_2 \times 10^x + P_1 \times 10^{x-k}$。
5. 因$P_1 < P_2$，所以$P_2 \times 10^x + P_1 \times 10^{x-k}$恆大於$P_1 \times 10^x + P_2 \times 10^{x-k}$，代表$S'$並非最佳解。
6. 若S_1'裡仍存在逆序數對，則重複步驟三得到S_2'，重複直到S_x'不存在逆序數對為止。
7. 最後$S \geq S_x' \geq S_{x-1}' \geq .... \geq S_1' \geq S'$，即$S \geq S'$，與假設矛盾，因此$S$為最佳解。

merge sort時間複雜度為$\mathcal{O}(N \log N)$，因此此算法複雜度也是$\mathcal{O}(N \log N)$。

(3)

基本上其做法與第二題相似，只是在merge sort時的compare function為：
我們定義$A \circ B$為$A * len(B) + B$，代表串接後的整數，則當$A \circ B > B \circ A$時$A > B$。

利用此comparator得到的最終序列會是最佳解。

在證明前我們要有一個先備知識，也就是兩個相同位數的數字，其大小是由高順位下第一個不同的數的大小決定，而$A \circ B$ 與 $B \circ A$的位數相同，而$A \circ B > B \circ A$也代表$A \circ B$的高順位數字比$B \circ A$大。再者，若$A \circ B > B \circ A$且$B \circ C > C \circ B$，則$A \circ C > C \circ A$。

小證明

令字串A的長度為$len(A)$，則

$$A \circ B > B \circ A$$
$$\Rightarrow A \times 10^{len(B)} + B > B \times 10^{len(A)} + A$$
$$\Rightarrow A \times (10^{len(B)} - 1) > B \times (10^{len(A)} - 1)$$

同理$B \circ C > C \circ B$
$$\Rightarrow B \times (10^{len(C)} - 1) > C \times (10^{len(B)} - 1)$$

因A，B，C，$10^{len(A)} - 1$，$10^{len(B)} - 1$，$10^{len(C)} - 1$皆為正數，因此將兩不等式同乘得

$$A \times B \times (10^{len(B)} - 1) \times (10^{len(C)} - 1) > B \times C \times (10^{len(A)} - 1) \times (10^{len(B)} - 1)$$

$$\Rightarrow \text{(相消得)} A \times (10^{len(C)} - 1) > C \times (10^{len(A)} - 1)$$

$$\Rightarrow A \times 10^{len(C)} + C > C \times 10^{len(A)} + A$$
$$\Rightarrow A \circ C > C \circ A$$

利用此性質，我們將這集合內的字串利用此性質做divide and conquer，將每個陣列分成最少兩個元素取比較，排序後再集合成更大的子陣列，最終集成大陣列後output。

Merge sort時間複雜度為$\mathcal{O}(N \log N)$，而每次比較時串接時若將$A \circ B$定義為$A \times 10^{len(B)} + B$，其計算複雜度是$\mathcal{O}(1)$，比較也是$\mathcal{O}(1)$，因此此算法複雜度也是$\mathcal{O}(N \log N)$。

(4)

在這我們使用counting sort的技巧：

1. 遍歷一次所有element，並且將每個digit的出現頻率記錄在一個0~9的table裡，另外在遍歷的過程中我們也將所有digit的和加起來。
2. 將和 mod 3 以下將分為3種情況：
   a. 餘數為零：代表此集合內的所有數字都應該串接，我們就查table裡每個digit的出現頻率，並且按照9~0的順序照頻率放入串接的答案中。
   b.餘數為一：代表有數字導致餘數為一需要剔除，我們就依1,4,7的順序查表，若這三個數字有出現過的就將其頻率減1後就輸出答案。若都沒出現過就查2，5，8的表，我們需要從這些數字裡挑兩個最小的剔除，做法與剛剛查1,4,7的表一樣，在做完剔除的動作，將digit依照新table裡的頻率按照9~0的順序照頻率放入串接的答案中。
   c.餘數為二：代表有數字導致餘數為二需要剔除，我們就依2,5,8的順序查表，若這三個數字有出現過的就將其頻率減1後就輸出答案。若都沒出現過就查1，4，7的表，我們需要從這些數字裡挑兩個最小的剔除。在做完剔除的動作，將digit依照新table裡的頻率按照9~0的順序照頻率放入串接的答案中。
   d. 若在剔除兩個數之後其總和仍不是3的倍數的話，直接輸出0。

證明：(有點累了，寫簡單一些)
首先先說明為何最多只挑兩個數字，若照演算法挑不到數字就直接輸出0：

1. 不管裡面的數字組成是怎樣，只要是正數，比較多位數的數字一定比較少位數的大。
2. 假設每個digit的總和是3的倍數，那我完全不必挑出任何數字，這樣絕對比挑出一個以上數字的還要大。
3. 若總和mod1，那他總共有兩種可能：
   a. 一個mod 1的數字加上一串總和mod 0的數字集合。
   b. 兩個mod 2的數字加上一串總和mod 0的數字集合(代表集合沒有任何mod 1的數)。
   而若有取三個數以上的集合，其中肯定能以以上這兩種case去減少取的數量。因為若取三個數內的某一個數為mod 1，那我只需取出那個數即可達成，且位數也比取三個的多，肯定大於取三個數的。那三個數內的沒任何一個數是mod 1，那肯定都是mod 2，而兩個mod 2的數的總和就是mod 1了，所以我只需要取兩個mod 2的數就可以使總和為三的倍數了。若存在這樣取都取不出來的那只說明，它是由兩個mod 2的數組成的集合，只能輸出零才能達成題目要求。
4. 若總和為mod 2，也總共有兩種可能
   a. 一個mod 2的數字加上一串總和mod 0的數字集合。
   b. 兩個mod 1的數字加上一串總和mod 0的數字集合(代表集合沒有任何mod 2的數)。
   其理由與第三點一樣，若有一個mod 2的數存在，取他就會是最大值。而如果沒有mod 2，那肯定也是兩個mod 1的數的總和才可以使這個數mod 2，取出兩個數即可。同樣若取完後沒有任何數字存留，那就直接輸出零。

上面已經證明了取的數字數量，接下來說明挑的順位：

1. 若數字總合為三的倍數，我全部的數字都要串接。而在相同位數下，高順位的位數決定了這個數與其他相同位數的數字的大小比較，因此若將集合內的大數字依序由高位數排到低位數，其組成的數字一定是最大的。(事實上前幾題都一直在說明這件事)
2. 若不得已需要取出數字來符合題目要求，我會希望取越少數字越好，這點在上面已解釋清楚。而取出的數字越小越好，因為同樣是取出一個數字，取出大的與取出小的的差別在於，在依序組成字串時，若將取出大的數字與取出小的數字的最高不同位數來比較，因為取出大的數字的那個數少一個，所以在比較到取出數字的最後一個位數時，取出小的數字還有最後一個大數，但取出大的已經沒有數字，所以只能那更小的數字來填補空缺。因此最高不同位數的數字比較，一定是取出比較小的數字會大於取出大的數字。
3. 結合以上兩點，我們維持取出最少數字以及最小數字的greedy property後按序輸出就會是OPT。

此作法只需遍歷一次即可算完0~9的出現頻率以及總和，而照著上述的演算法取出數字是O(1)的事情。因此只需O(n)的時間複雜度與空間即可完成。

(5)

Yes, I Can!!!

98653 です。

(6)

ref: https://web.archive.org/web/20160120093629/http://algobox.org/create-maximum-number/ (https://web.archive.org/web/20160120093629/http://algobox.org/create-maximum-number/)

這問題可以分成兩個大子問題的結合，分別是：

- 給定一個陣列長度n，要取出最大的長度k的數字，且不能破壞順序。
- 給定兩個長度m+n的陣列，取出最大的數字，其長度k=m+n。(要兩個陣列的數字全部取完，且不能破壞在陣列裡的順序)

我們先解決第一個子問題：
我們利用stack的特性來實現greedy property，stack的特性是FILO，這特性使的我們push進去的數字必須pop掉之前的所有數字才可以被pop掉，且若依序push & pop，後面的數字是不可能跑到前面的數字之前的，利用這特性output出的序列都會維持在原序列的順序。

我們在決定push 跟pop的機制在於，我現在要push進的digit與stack.top()裡的數字大小比較，若top()比要push的數字小就pop掉，直到top()比要push的數字大或者stack為空；反之，就push進stack裡。如此原因在於我們要盡力維持stack裡的單調性，只有stack最深處的digit是最大的數字，其output才會是最大的(根據上面最高位數的digit決定數字大小的討論)。

然而因為要output長度為k的數字，因此我pop的數量是有限度的(n-k)。若在演算法執行過程中pop的quota到達上限，代表剩下的數字我必須全部放入stack，即使會破壞單調性。若在演算法執行完後stack的size大於k的數量，此時stack應照著大小有大至小從stack的深處排到stack的top，根據我們前幾題討論的觀點(越高順位的digit越大，騎術字本身越大)，那我們也只需從stack裡的最底層依序output k個數字即是答案。

因為每個digit最多會被push pop 一次，因此其時間複雜度為$\mathcal{O}(N)$。

接著來解決第二個子問題：

因為我兩個陣列都要全部取完，且不能破壞彼此在原陣列的順序，因此題目可轉型成，我總共要從這兩個陣列中各取出一個現在可以取的最高位數做k次比較，並且將它依序放在合併後的陣列的高順位數處。因為我們要放的位子都會是最高位數處，因此我們會希望這個位數的數字越大越好，我們在這裡因入我第三小題時用的運算子。

假定現在有A，B兩陣列，A有n項digit，B有m項digit，我們要依序取出所有AB陣列裡的digit組成新的$n+m$長度的C陣列，若$A \circ B > B \circ A$則將A的最高順位數放進C裡，並將此位數在A中移除，反之將B的最高順位數放進C裡並將此位數從B移除，反覆比較至AB皆為空。

小證明

令現在這個OPT的output為S，假定有另一OPT'的 output S'並沒有遵照greedy property，且$S' > S$。則OPT'在$A \circ B > B \circ A$時將B的最高順位放入C。因$A \circ B > B \circ A$，A的高順位位數a必定大於B的同位位數b，而OPT'優先選取B的高順位位數，僅會讓a出現在比b更後面的順位。而根據上述高位數的digit決定數字的大小的規則，遵循OPT'的S'裡的a順位必定不高於S，因此$S \geq S'$，與假設矛盾，因此遵循此greedy property 的OPT為最佳解。

此步驟時間複雜度因每次比較最多需要遍歷所有字串共k項，而需要比較k次，因此複雜度為$\mathcal{O}(k^2) = \mathcal{O}(N^2)$。

回到這題本身，我們要2個n個字串A與B裡選出k個位數組成maximum satisfying value S，此k位數必定由A陣列的i個位數與B陣列的$k-i$個位數所組成，因此我們遍歷所有情況由i從0到k，共k+1項，而每次決定每個陣列要output的個數後，需兩個陣列先各跑一次第一個子問題輸出字串之後，再將兩個輸出放入第二個子問題內合併。當遍歷完所有可能後，比較合併後的值並選擇最大者輸出。

此演算法複雜度為$(k+1) \times (2\mathcal{O}(N) + \mathcal{O}(N^2)) = \mathcal{O}(kN^2)$。