Problem 5 The Robbers

(1)

(a)

採取的 greedy 策略是讓每一個 robber 搶越多人越好，但是搶人的時間不能超過 T。所以可能會有 robber 沒搶到。

假設 robber1 搶 r1 個人, robber2 搶 r2 個人，以此類推。

其中 r1+r2+..+$r_M = N$, 若第 i 個 robber 沒搶到，則$r_i = 0$。

設計演算法如下：

1. Make $t_1 + t_2 + .. + t_{r1}$ as close to T as possible, but less than T, assign those people to robber 1 to rob. //O(r1)
2. Then make $t_{r1+1} + t_{r1+2} + .. + t_{r1+r2}$ as close to T as possible, but less than T, assign those people to robber 2 to rob. //O(r2)
3. Repeat the process until all the people are robbed or there are not enough robbers //O(r3)+O(r4)+… +O(r_M)
4. if there are not enough robbers, return False, else return True //O(1)

根據上面的演算法，所會需要的總時間為

max ($\sum_{i=1}^{r1} t_i, \sum_{i=r1+1}^{r1+r2} t_i, \ldots, \sum_{N-rM+1}^{N} t_i$)因為每次加時間t_i都會計算到小於 T 為止，所以總時間不會超過 T，但是如果 robber 人數不夠，表示時間一定會超過 T。以上演算法的 time complexity 會在 O(r1)+O(r2)+..+O(r_M)= O(N)時間內跑完。

(b)

Optimal substructure:

假設今天只有一個 robber，依照上面的演算法，得到$\min(T - \sum_{i=1}^{r1} t_i) \geq 0$，那假如 r1>N，此時就可以在 T 時間內搶完。

假設今天有 M 個 robbers，我們所需要搶劫的時間為$T_{need} = \max\left(\min(T - \sum_{i=1}^{r1} t_i), \min(T - \sum_{i=r1+1}^{r2} t_i), \ldots, \min(T - \sum_{N-rM+1}^{N} t_i)\right)$

假設每一個 robber 都是按照(a)的想法來搶，每一個 robber 搶的時間都不會超過 T，那麼總體看來，當 robber 團體一起搶的時候，就算他們取最大值，也就是T_{need}，必會小於T。

因此證明，此問題具有 optimal substructure。

Greedy choice property:

假設今天我們讓每一個 robber 不用搶到最多人，也就是$T_{need}' = \max\left(T - \sum_{i=1}^{r1} t_i,\ T - \sum_{i=r1+1}^{r2} t_i, \ldots, T - \sum_{N-rM+1}^{N} t_i\right)$。

搶匪人數若在 greedy 策略下，人數如果不夠而超時，今天用非 greedy 策略一樣會超時。所以我們只討論搶匪人數在 greedy 策略下，人數足夠而且滿足時間限制 T 的情況。

考慮臨界情形，也就是採用 greedy 策略時，M 個搶匪都已經搶到最多人，而且每個搶匪的搶劫時間都非常接近 T。若在這個情況，我們不採用 greedy 策略，讓其中一個搶匪，比如說 robber1 少搶一個人，那剩下的 M-1 個搶匪一定要有一個人把 robber1 少搶的這一人吸收掉，此時在剩下的 M-1 個搶匪中，只要有人搶了那一人，時間都會超過 T。若 M-1 個搶匪都不搶，就會導致有一人沒被搶到。

以上可證明，此問題具有 greedy choice property。

(c)

考慮$T=\infty$的情形，$f(T)$必為 1，慢慢將T縮小，利用(a)
的演算法判斷是否能在T時間內搶完所有的人，應可得一最小值T_{min}。在此T_{min}以下，$f(T)=0$

由以上推論可得$f(T)=1\ if\ T\geq T_{min}, else\ f(T)=0$，故 f 為 monotonically increasing。

(d)

最少需 20s

robber1 搶 5, 8, 2, 1

robber2 搶 6, 4, 10

robber3 搶 9, 7, 3

(e)

1.令 T 為一足夠大的整數，最好稍微大於$\sum_{i=1}^{N} t_i$(正常情況下 T=O(N))，此時必定可以在 T 時間內做完，所以把此 T 當作 upper bound, r=T, l=0, middle=(r+l)/2

2. T=middle，利用(a)的演算法判斷是否可以在 T 時間內做完

3.若可以則 r=middle-1, middle=(r+l)/2

4.若不行，則 l=middle+1, middle=(r+l)/2

5.重複上述過程直到找到一最小的 T

以上是 binary search 的方法來找 T，花的時間不會超過 O(logN)

而(a)的演算法需時 O(N)

所以總共需要 O(NlogN)=$o(N^2)$

(2)

(1)

用一個 table，dp[n][w]表示 weight limit 為 w 且有 n 個物品時，所能得到的最大 value，wt[]存 w1, w2, ..., wN, val[]存 v1, v2, ..., vN

建立此表格需花費 O(NW)，演算法如下

```
for(i=0; i<=N; i++)
    for(w=0; w<=W; w++)
        if(i==0 ||w==0)
            dp[i][w]=0;
        else if(wt[i-1]<=w)
            dp[i][w]=max(dp[i-1][w], dp[i-1][w-wt[i-1]]+val[i-1])
        else
            dp[i][w]=dp[i-1][w]
```

利用此表格 backtracking 得到一組物品，以集合表示為 S，其數量為$|S|$。K=1 的情況下，丟到 vault 的物品必定會在 S 中。

1.令 min=INT_MAX

2.將 S 集合的物品依序一次取一個丟入 vault //O($|S|$)

3.重算 table(此時剩 N-1 個物品，重量限制仍為 W，val[]和 wt[]會少掉一組在 S 內的值。) // O(NW)

4.求得 dp[N-1][W]

5.若 dp[N-1][W]<min, min=dp[N-1][W]

6.重複 2~5 過程

以上演算法可得 min 為答案，且時間複雜度為 O($|S|NW$)

(2)

dp[n][w][k]表示表示 weight limit 為 w 且有 n 個物品，且最多取 k 個物品到 vault 時，所得到最大的 value

dp[n][w][0]表示不取任何物品到 vault 的情形，即為一般的 knapsack problem

因爲被搶的東西有保險，希望被搶的東西價值越大越好。而且要讓 k 至少等於 1，以免被保險公司懷疑詐保。

對於 dp[n][w][k]而言，有取物品放入 vault 的情形，和不取物品放入 vault 的情形。若有取物品放入 vault，此物一定不會被搶。若沒有取物品放入 vault，有被搶和不被搶兩種情況。依據以上推論，可得以下 recurrence function dp[n][w][k]= max(dp[n-1][w][k], max(dp[n-1][w-wt[i]][k-1]+val[i], dp[n-1][w][k-1]))

第二個 max 中表示不取到 vault 的情形，此時確定恰取到 k-1，而因為不取入 vault，所以有被搶或不被搶兩種情形，對於搶匪而言，要取 max。

第一個 max，對店家而言，要 maximize V 使保險公司賠最多錢，所以取 max。而 dp[n-1][w][k]表示取到 vault 的情形，此時表示最多取到 k，但物品只有 n-1 個。

最後要將 dp[N][W][k], $1 \leq k \leq N$跑一遍，找其中的 max，即為解答。

```
Initialize dp[n][w][k]=∞,  1 ≤ n ≤ N, 1 ≤ w ≤ W, 1 ≤ k ≤ N
dp[n][w][k]=0 if (n==0 or w==0),  1 ≤ k ≤ N
用(1)所示建表格之演算法先算 dp[n][w][0] //O(NW)
for(n=1; n<=N; n++)    //O(N)
    for(w=1;w<=W;w++ )    //O(W)
        for(k=1; k<=n; k++)      //O(N),  有 n 個物品，最多只能取 n 個到 vault
            max_val= max(dp[n-1][w-wt[i]][k-1]+val[i], dp[n-1][w][k-1])
            max_val= max( dp[n-1][w][k],max_val)
            dp[n][w][k]=max_val
max=0;
for(k=1; k<=N; k++){      //O(N)
        if(dp[N][W][k]>max){
        max=dp[N][W][k];
        }
}
return max
```

以上 algorithm 可在$O(N^2W) + O(N) = O((N+W)^3)$跑完