Problem 5 The Robbers

(1)

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+..+, 若第i個robber沒搶到，則。

設計演算法如下：

1. Make as close to T as possible, but less than T, assign those people to robber 1 to rob. //O(r1)
2. Then make 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()
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因為每次加時間都會計算到小於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()= 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，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

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

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。

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)

考慮在此

由以上推論可得，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(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)=

(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。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()

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()

(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], 跑一遍，找其中的max，即為解答。

1. Initialize dp[n][w][k]=,
2. dp[n][w][k]=0 if (n==0 or w==0),
3. 用(1)所示建表格之演算法先算dp[n][w][0] //O(NW)
4. for(n=1; n<=N; n++) //O(N)
5. for(w=1;w<=W;w++ ) //O(W)
6. for(k=1; k<=n; k++) //O(N), 有n個物品，最多只能取n個到vault
7. max\_val= max(dp[n-1][w-wt[i]][k-1]+val[i], dp[n-1][w][k-1])
8. max\_val= max( dp[n-1][w][k],max\_val)
9. dp[n][w][k]=max\_val
10. max=0;
11. for(k=1; k<=N; k++){ //O(N)
12. if(dp[N][W][k]>max){
13. max=dp[N][W][k];
14. }
15. }
16. return max

以上algorithm可在