演算法 #Hw3

賴冠廷 陳昀靖

Notes

- Pseudo code
  - 縮排有意義，不可以跑掉
  - 括號：要就全都要寫、不然就都不要，改用縮排替代
  - 投影片是因為有篇幅限制所以才放左右，因有縮排希望同學列出完整的整頁
  - 變數的定義：不要過程中突然出現新的變數，又沒解釋是什麼
  - for loop 注意 0 to n-1 or 1 to n
- 建議同學多練習Pseudo code
- 作業若評論有「找助教解釋」，可來找助教解釋後獲得部分分數

Q1.

- Consider algorithm *Mapping* (see slides). Is it possible that the set *S* will become empty at the end of the algorithm? Show an example, or prove that it cannot happen.

Q1.

- 利用反證法: 假設 S 最後會是空集合
- 根據 Mapping 演算法，每次會消去一個沒被連到的點
- 令空集合前一步驟的點為 e, 並因為下一步驟會被消除 c[e] = 0
- 根據定義，A 中的每個點必然連到 A 中的點，得知 f(e) != e
- 但 e 所連到的點不應該在 e 之前就被消除掉了
- 矛盾! 因此證明 S 不會是空集合

Q2.

- Write a program (or modify the code discussed in class) to recover the solution (i.e., enumerate the elements in the solution) to a knapsack problem using the *belong* flag. You should make your algorithm as efficient as possible.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$k_1=2$	O	-	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-
$k_2=3$	O	-	O	I	-	I	-	-	-	-	-	-	-	-	-	-	-
$k_3=5$	O	-	O	O	-	O	-	I	I	-	I	-	-	-	-	-	-
$k_4=6$	O	-	O	O	-	O	I	O	O	I	O	I	-	I	I	-	I

I: a solution containing this item has been found

O: a solution without this item has been found

-: no solution has not yet been found

Q2.

```
Knapsack_Output (n, S, K, P):
begin
    if !P[n, K].exist then
          print “No solution!”
    else
          i := n;
          k := K;
          while k > 0 do
                if P[i, k].belong then
                      print S[i];
                      k := k - S[i];
                i := i – 1
end
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
k_1=2	O	-	I	-	-	-	-	-	-	-	-	-	-	-	-	-	-
k_2=3	O	-	O	I	-	I	-	-	-	-	-	-	-	-	-	-	-
k_3=5	O	-	O	O	-	O	-	I	I	-	I	-	-	-	-	-	-
k_4=6	O	-	O	O	-	O	I	O	O	I	O	I	-	I	I	-	I

I: a solution containing this item has been found

O: a solution without this item has been found

-: no solution has not yet been found

Q3.

- Let $x_1, x_2, \cdots, x_n$ be a sequence of real numbers (not necessarily positive). Design an $O(n)$ algorithm to find the subsequence $x_i, x_{i+1}, \cdots, x_j$ (of consecutive elements) such that the product of the numbers in it is maximum over all consecutive subsequences. The product of the empty subsequence is defined as 1.

Q3.

```
Find_Maximum_Product_Sequence(n ,X):
maxNum[0] = 1
minNum[0] = 1
maxProductSequence = 1
for i := 1 to n
        maxNum[i] = max(X[i], maxNum[i-1]*X[i], minNum[i-1]*X[i])
        minNum[i] = min (X[i], maxNum[i-1]*X[i], minNum[i-1]*X[i])
        maxProductSequence = max(maxProductSequence, maxNum[i])
return maxProductSequence
```

Q4.

- The Knapsack Problem that we discussed in class is defined as follows: Given a set *S* of *n* items, where the *i*-th item has an integer size *S[i]*, and an integer *K*, find a subset of the items whose sizes sum to exactly *K* or determine that no such subset exists. We have described in class an algorithm to solve the problem. Modify the algorithm to solve a variation of the knapsack problem where each item has an unlimited supply. In your algorithm, please change the type of *P[i, k].belong* into integer and use it to record the number of copies of item *i* needed.

Q4.

```
Multiple_Knapsack (S, K):
begin
        P[0, 0].exist := true;
        P[0, 0].belong := 0;
        for k := 1 to K do
                P[0, k].exist := false;
        for i := 1 to n do
                for k := 0 to K do
                        P[i, k].exist := false;
                if P[i – 1, k].exist then
                        P[i, k].exist := true;
                        P[i, k].belong := 0;
                else if k - S[i] ≥ 0 then
                        if P[i, k - S[i]].exist then
                                P[i, k].exist := true;
                                P[i, k].belong :=
                                P[i, k - S[i]].belong + 1;
end
```

Q4. Wrong

```
Multiple_Knapsack (S, K):
begin
        P[0, 0].exist := true;
        P[0, 0].belong := 0;
        for k := 1 to K do
                P[0, k].exist := false;
        for i := 1 to n do
                for k := 0 to K do
                        P[i, k].exist := false;
                if P[i – 1, k].exist then
                        P[i, k].exist := true;
                        P[i, k].belong := 0;
                else if k - S[i] ≥ 0 then
                        if P[i-1, k - S[i]].exist then
                                P[i, k].exist := true;
                                P[i, k].belong :=
                                P[i, k - S[i]].belong + 1;
end
```

Q4.

Wrong	0	1	2	3	4	5	6
	0	-	-	-	-	-	-
$k_1 = 2$	0	-	1	-	-	-	-
$k_2 = 3$	0	-	0	1	-	1	-

Correct	0	1	2	3	4	5	6
	0	-	-	-	-	-	-
$k_1 = 2$	0	-	1	-	2	-	3
$k_2 = 3$	0	-	0	1	0	1	0

Q5.

- Let $x_1, x_2, \cdots, x_n$ be a set of integers, and let $S = \sum_{i=1}^{n} x_i$. Design an algorithm to partition the set into two subsets of equal sum, or determine that it is impossible to do so. The algorithm should run in time *O(nS).*

Q5.

```
Equal_Subset (n, X, S):
if S is odd then
        print ”No solution!”;  // 沒有判斷odd的話，除以2會出錯 e.g. 9/2 = 4 or array index
else then                                出現小數 e.g. P[n, 4.5]
        let halfSum = S/2;
        set1 = {}
        set2 = {}
        Build knapsack table P[n, halfSum];
        if P[n, halfSum].exist then
                while n > 0 do
                        if P[n, halfSum].belong then
                                put X[n] in set1;
                                halfSum -= X[n];
                        else
                                put X[n] in set2;
                        n --
        else
                print “No solution!”;
```