Homework 4

Yu-Ju Teng Ling-Hsuan Chen

Question 1

1. (5.3 adapted) Consider algorithm *Mapping* (see notes/slides). The algorithm starts with a nonempty set A, of course, since a mapping from A to itself is considered. 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.

Question 1

One-to-One Mapping (cont.)

Algorithm Mapping (f, n);
begin
    $S := A$;
    for $j := 1$ **to** n **do** $c[j] := 0$;
    for $j := 1$ **to** n **do** increment $c[f[j]]$;
    for $j := 1$ **to** n **do**
        if $c[j] = 0$ **then** put j in Queue;
    while Queue not empty **do**
        remove i from the top of Queue;
        $S := S - \{i\}$;
        decrement $c[f[i]]$;
        if $c[f[i]] = 0$ **then** put $f[i]$ in Queue
end

Question 1

If A (original set) $= \emptyset$, then $S = \emptyset \subseteq A$ is still empty.

If $A \neq \emptyset$, suppose S will become empty at the end.
Before the set become to the empty set, the last step is eliminating the last element (called a) in S and $c[a] = 0$.

1. If $f(a) = a$, it is impossible because $c[a] = 0$.
2. If $f(a) \neq a$, it is impossible because a is the only one element in S.

Hence, we conclude that when $A \neq \emptyset$, S will not become empty.

Question 2

2. Design an efficient algorithm that, given a sorted array A of n integers and an integer x, determine whether A contains two integers whose sum is exactly x. Please present your algorithm in adequate pseudocode and make assumptions wherever necessary. Give an analysis of its time complexity. The more efficient your algorithm is, the more points you will be credited for this problem.

Question 2

The logic of the algorithm:

1. Assume A is a sorted array with increasing order.
2. Let l, r be the indexes of the first and the last element in Array A.
3. While $l < r$, do the check for each pair:
    1. If $A[l] + A[r] = x$,
    return True.
    2. else if $A[l] + A[r] < x$,
    increase l by 1.
    3. else if $A[l] + A[r] > x$,
    decrease r by 1.
4. If there are no solutions, return *False* at the end of the algorithm.

Question 2

The pseudocode are showed as followed:

```
1:  Algorithm SmartTwoSum(A, x);
2:      // Let n be the length of array A.
3:      l := 0;
4:      r := n − 1;
5:      while l < r do
6:          if A[l] + A[r] = x then
7:              return True;
8:          else if A[l] + A[r] < x then
9:              l := l + 1;
10:         else if A[l] + A[r] > x then
11:             r := r − 1;
12:     return False
```

Since in the worst case, we go through the whole array at most once. The time complexity of this algorithm is $O(n)$.

Question 3

3. (5.7) 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.

Question 3

Use the table P obtained from the **Knapsack** algorithm, where $P(n, K)$ denotes the solution to the problem instance with n as the number of the items and K as the size of the knapsack.

For $0 \leq i \leq n$ and $0 \leq k \leq K$,

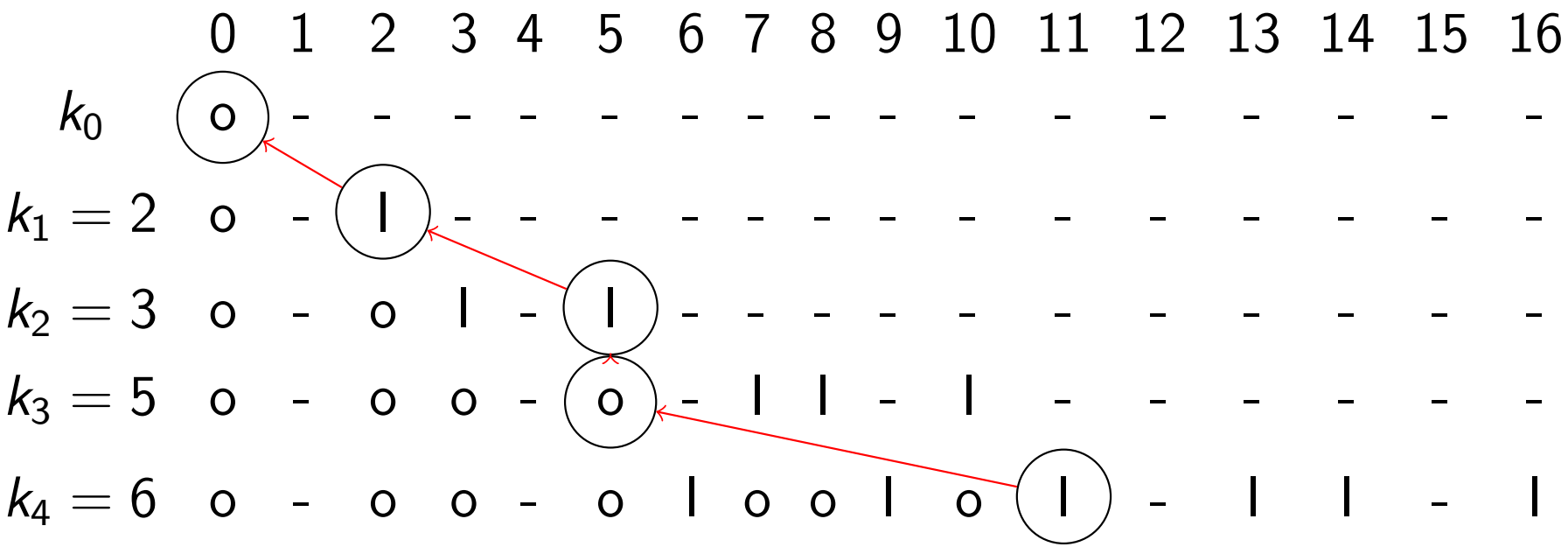
1. If $P[i, k].\mathit{exist}$ is *false*, then there's no solution.
2. If $P[i, k].\mathit{exist}$ is *true*, check whether $S[i]$ is in the solution (i.e., whether $P[i, k].\mathit{belong} = \mathit{true}$).
3. If so, put $S[i]$ into the solution and check for $P[i-1, k-S[i]]$.
4. if not, check for $P[i-1, k]$.

Question 3

```
1:  Algorithm Knapsack Recover(S, k, P);
2:      solution := [];
3:      i := n ;
4:      if P[i, k].exist = false then
5:          return "No such subset exists!";
6:      while k > 0 do
7:          if P[i, k].belong = true then
8:              solution.append(S[i]);
9:              k := k − S[i];
10:         i := i − 1;
11:     return solution;
```

Question 3

Recover(S, 11, P) = [6, 3, 2]



Question 4

4. (5.17) The Knapsack Problem that we discussed in class is defined as follows. Given a set S of n items, where the ith 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, change the type of $P[i, k].belong$ into integer and use it to record the number of copies of item i needed.

Question4

The original version of the algorithm from class:

```
1:  Algorithm KNAPSACK(S, K);
2:      P[0, 0].exist := true;
3:      for k := 1 to K do
4:          P[0, k].exist := false;
5:      for i := 1 to n do
6:          for k := 0 to K do
7:              P[i, k].exist := false;
8:              if P[i − 1, k].exist then
9:                  P[i, k].exist := true;
10:                 P[i, k].belong := false;
11:             else if k − S[i] ≥ 0 then
12:                 if P[i − 1, k − S[i]].exist then
13:                     P[i, k].exist := true;
14:                     P[i, k].belong := true;
```

Question 4

```
1:  Algorithm KNAPSACK UNLIMITED(S, K);
2:      P[0, 0].exist := true;
3:      P[0, 0].belong := 0;
4:      for k := 1 to K do
5:          P[0, k].exist := false;
6:      for i := 1 to n do
7:          for k := 0 to K do
8:              P[i, k].exist := false;
9:              if P[i − 1, k].exist then
10:                 P[i, k].exist := true;
11:                 P[i, k].belong := 0;
12:             else if k − S[i] ≥ 0 then
13:                 if P[i, k − S[i]].exist then
14:                     P[i, k].exist := true;
15:                     P[i, k].belong := P[i, k − S[i]].belong + 1;
```

Question 5

5. (5.23) Write a non-recursive program (in suitable pseudocode) that prints the moves of the solution to the towers of Hanoi puzzle. The three pegs are respectively named A, B, and C, with n (generalizing the original eight) disks of different sizes stacked in decreasing order on peg A.

Question5

The following pseudo-code shows how **recursive** Hanoi works.

```
1: Algorithm RecursiveHanoi(n, s, t, a);
2:     // s is source peg, t is target peg, and a is auxiliary peg.
3:     if n = 1 then
4:         print s + " to " + t;
5:         return ;
6:     RecursiveHanoi(n − 1, s, a, t);
7:     RecursiveHanoi(  1  , s, t, a);
8:     RecursiveHanoi(n − 1, a, t, s);
```

Question 5

We can use a stack to represent the steps in a recursive version. Due to the first-in-last-out (FILO) property of the stack, the sequence of operators in the stack version should be reversed.

```
1:  Algorithm HANOI(n, s, t, a);
2:      // s is source peg, t is target peg, and a is auxiliary peg.
3:      stk := empty stack;
4:      stk.push(⟨n, s, t, a⟩);
5:      while !stk.empty() do
6:          p := stk.top();
7:          stk.pop();
8:          if p.h = 1 then
9:              print p.s + " to " + p.t;
10:         else if p.h > 1 then
11:             stk.push(⟨ p.h-1, p.a, p.t, p.s ⟩); //RecursiveHanoi(n-1, a, t, s)
12:             stk.push(⟨   1   , p.s, p.t, p.a ⟩); //RecursiveHanoi(1, s, t, a)
13:             stk.push(⟨ p.h-1, p.s, p.a, p.t ⟩); //RecursiveHanoi(n-1, s, a, t)
```

$\langle h, s, t, a \rangle$ is a struct to represent the state of a Hanoi game, where h denotes the number of pegs to move, s denotes source peg, t denotes target peg, and a denotes auxiliary peg.