Homework 4

Yu Hsiao Yu-Hsuan Wu

Announcement

- Inquires About Assignments and Grades
  - We have distributed your graded Homework 2 (last TA session), Homework 3, and Homework 4.
  - If you have any inquiries, please discuss them with us later.
  - Also, there will be a TA Hour this Thursday. You are also welcome to come and discuss with us.
  - For more information, please refer to the announcement previously published on NTU COOL.
- Midterm is around the corner!
  - Time: Next Tuesday (10/29), 14:20-17:20
  - Location: Room 305, Management Building 2

Question 1

1. 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 also an analysis of its time complexity. The more efficient your algorithm is, the more points you will be credited for this problem.

Question 1

Thinking process:

1. Assume A is a sorted array with ascending order.
2. Let l, r be the indices of the first and the last element in Array A.
3. While $l < r$, we check the relation of each $A[l]$ and $A[r]$:
   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 is no solution, return *False* at the end of the algorithm.
5. We will go through the whole array at most once. Thus, the time complexity will be $O(n)$.

Question 1

```
1:  Algorithm TwoSum(A, x);
2:      l := 0;
3:      r := A.length − 1;
4:      while l < r do
5:          if A[l] + A[r] = x then
6:              return True;
7:          else if A[l] + A[r] < x then
8:              l := l + 1;
9:          else if A[l] + A[r] > x then
10:             r := r − 1;
11:         end if
12:     end while
13:     return False
14: end Algorithm
```

Question 2

2. (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.

Question 2

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:      end for
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 := false;
12:             else if k − S[i] ≥ 0 then
13:                 if P[i − 1, k − S[i]].exist then
14:                     P[i, k].exist := true;
15:                     P[i, k].belong := true;
16:                 end if
17:             end if
18:         end for
19:     end for
20: end Algorithm
```

Question 2

```
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:      end for
7:      for i := 1 to n do
8:          for k := 0 to K do
9:              P[i, k].exist := false;
10:             if P[i − 1, k].exist then
11:                 P[i, k].exist := true;
12:                 P[i, k].belong := 0;
13:             else if k − S[i] ≥ 0 then
14:                 if P[i, k − S[i]].exist then
15:                     P[i, k].exist := true;
16:                     P[i, k].belong := P[i, k − S[i]].belong + 1;
17:                 end if
18:             end if
19:         end for
20:     end for
21: end Algorithm
```

Question 2

The original Knapsack Problem:

	0	1	2	3	4	5	6	7	8	9	10	11	12
	O	–	–	–	–	–	–	–	–	–	–	–	–
$k_1 = 2$	O	–	I	–	–	–	–	–	–	–	–	–	–
$k_2 = 3$	O	–	O	I	–	I	–	–	–	–	–	–	–

Question 2

The revised Knapsack Problem with unlimited supply of items:

	0	1	2	3	4	5	6	7	8	9	10	11	12
	0	–	–	–	–	–	–	–	–	–	–	–	–
$k_1 = 2$	0	–	1	–	2	–	3	–	4	–	5	–	6
$k_2 = 3$	0	–	0	1	0	1	0	1	0	1	0	1	0

Question 3

3. (5.20 adapted) You are given a set of n integers, stored in an array X. Design an algorithm to partition the set into two subsets of equal sum, or determine that it is impossible to do so. Please present your algorithm in adequate pseudocode and make assumptions wherever necessary. Give also an analysis of its time complexity.

Question 3

```
1:  Algorithm PARTITION(X);
2:      totalSum := 0;
3:      for i := 1 to n do
4:          totalSum := totalSum + X[i];
5:      end for
6:      if totalSum is odd then
7:          return False;
8:      end if
9:      target := totalSum/2;
10:     P := Knapsack(X, target);
11:     if P[n, target].exist = False then
12:         return False;
13:     end if
14:
```

Question 3

15: **Algorithm** PARTITION(Continue)
16: $array1 := []$;
17: $array2 := []$;
18: **while** $n > 0$ **do**
19: **if** $P[n, target].belong$ **then**
20: $array1.push(X[n-1])$;
21: $target := target - X[n-1]$;
22: **else**
23: $array2.push(X[n-1])$;
24: **end if**
25: $n := n - 1$;
26: **end while**
27: return $array1$, $array2$;
28: **end Algorithm**

Question 4

4. Suppose that you are given an algorithm/function called *subsetSum* as a *black box* (you cannot see how it is designed) that has the following property: if you input any sequence X of real numbers and an integer k, $subsetSum(X, k)$ will answer "yes" or "no", indicating whether there is a subsequence of the numbers whose sum is exactly k. Show how to use this black box to find the subsequence whose sum is k, if it exists.

   Please present your algorithm in adequate pseudocode and make assumptions wherever necessary. You should use the black box $O(n)$ times (where n is the size of the sequence). Note that a sequence of n numbers are stored in an array of n elements. Note also that the input k must be an integer.

Question 4

Thinking process:

1. *subsetSum*(X, k) can tell us if there is a legal subsequence in X such that its sum equals to k.
2. If removing an element x from a sequence X would change the returning value of *subsetSum*, i.e., *subsetSum*$(X, k) =$"yes" and *subsetSum*$(X \setminus \{x\}, k) =$ "no", then x is required.
3. Similarly, if not, then x is not required.
4. Be careful about the condition for recovering the element.
5. We will use the *subsetSum* function $n + 1$ times, which satisfies the condition (that we should use the black box $O(n)$ times).

Question 4

```
1:  Algorithm FindSubset(X, k);
2:      if subsetSum(X, k) = "no" then
3:          return [];
4:      end if
5:      n := X.length;
6:      for i := 1 to n do
7:          x := X.head;
8:          X.pop_front();
9:          if subsetSum(X, k) = "no" then
10:             X.push_back(x);
11:         end if
12:     end for
13:     return X;
14: end Algorithm
```

Question 4

Some common mistakes include

1. Updating the value of k.
   $\rightarrow$ *subsetSum* can only accept a sequence of real numbers and an integer.
2. Temporary exclude one element from the list and always add it back later.
   $\rightarrow$ Try $X = [2, 2], k = 2$. You will get an empty list at the end of the execution.

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.

Question 5

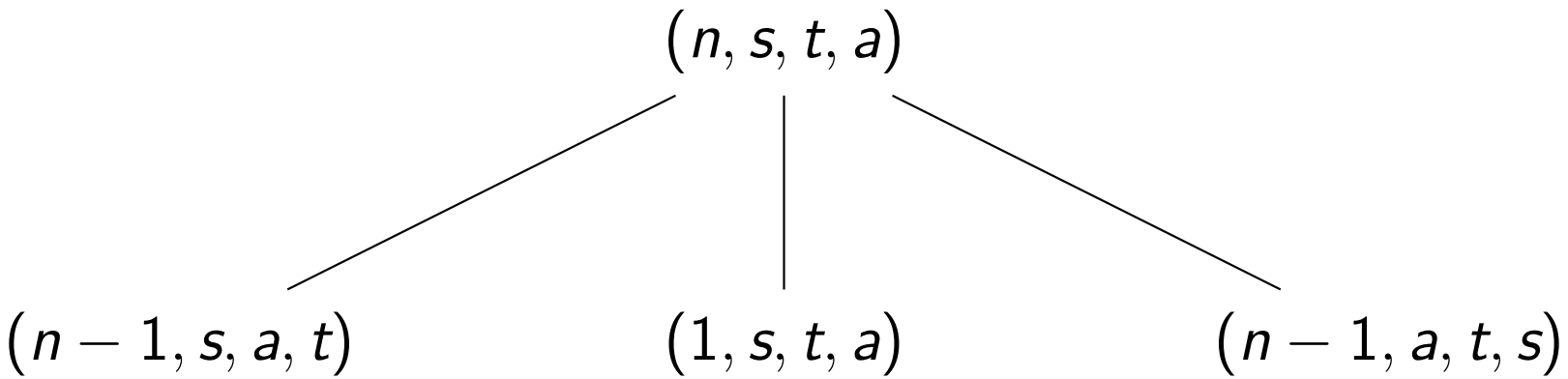
The step of solving a Hanoi puzzle with n disks can be simply written in 3 steps:

1. move the top $n-1$ disks from the source peg to the auxiliary peg.
2. move 1 disk (the bottom one) from source peg to target peg.
3. move the $n-1$ disks on the auxiliary peg to the target peg.

Question 5

```
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:      end if
7:      RecursiveHanoi(n − 1, s, a, t);
8:      RecursiveHanoi(   1   , s, t, a);
9:      RecursiveHanoi(n − 1, a, t, s);
10: end Algorithm
```

Question 5



Question 5

```
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)
14:         end if
15:     end while
16: end Algorithm
```