COMP4500/7500 Advanced Algorithms & Data Structures
Sample Solution to Tutorial Exercise 6 (2014/2)*

School of Information Technology and Electrical Engineering, University of Queensland

September 2, 2014

1. (Aho, Hopcroft and Ullman, *Data Structures and Algorithms*, Exercise 10.5)

   The number of combinations of m things chosen from amongst a set of n things is denoted $C(n, m)$, for $n \geq 0$ and $0 \leq m \leq n$. We can give a recurrence for $C(n, m)$ as follows:

   $$\begin{array}{lcll} C(n,0) & = & C(n,n) = 1 & \\ C(n,m) & = & C(n-1,m) + C(n-1,m-1) & \text{for } 0 < m < n \end{array}$$

   $C(n, m)$ are also known as the binomial coefficients and are often written $\left(\begin{array}{c} n \\ m \end{array} \right)$.

   (a) Justify the above recurrence.

   Sample solution. There is only one way to choose a set of n items (or 0 items) from n items: one has to take the lot (or none). Consider choosing a set of m items from a set of n items, for $0 < m < n$. If one considers one particular item, either it is included in the chosen set or it is excluded. If it is excluded, then one has to choose m items from the remaining $n - 1$ items; there are $C(n - 1, m)$ combinations. If it is included, then one has to choose $m - 1$ items from the remaining $n - 1$ items; there are $C(n - 1, m - 1)$ combinations. The total number of combinations is the sum of these.

   (b) Give a recursive function to compute $C(n, m)$ in pseudocode or Java or C.

   Sample solution. The following Java function follows the structure of the recurrence above closely.

```
int C(int n, int m) {
  // Pre: 0 <= m <= n
  if( (m == 0) || (m == n) ) {
    return 1;
  } else {
    return C(n-1,m) + C(n-1, m-1);
  }
}
```

   (c) Give a dynamic programming algorithm to compute $C(n, m)$. Hint: The algorithm constructs Pascal's triangle.

   Sample solution. The value of $C(n, m)$ depends on values of $C(i, j)$ for $i < n$ and $j \leq m$. If we tabulate the values of $C(i, j)$ in increasing order of i (and then increasing order of j, although this is not significant), we can use dynamic programming. We store values in a two-dimensional array `SC` (zero-based).

   C(n, m)

```
1  // Pre: 0 ≤ m ≤ n
2  for i = 0 to n
3      SC[i,0] = 1
4      SC[i,i] = 1
5      for j = 1 to i − 1
6          SC[i,j] = SC[i − 1,j] + SC[i − 1,j − 1]
7  return SC[n,m]
```

   (d) What are your dynamic programming solution's worst-case time and space complexities as a function of n?

Sample solution. The inner **for** loop has a body that requires constant time. Therefore the loop's time complexity is

$$\sum_{j=1}^{i-1} \Theta(1) = \Theta\left(\sum_{j=1}^{i-1} 1\right) = \Theta(i-1).$$

Hence the complexity of the inner loop is $\Theta(i)$. Hence the time complexity of the outer loop is

$$\sum_{i=0}^{n} \Theta(i) = \Theta\left(\sum_{i=0}^{n} i\right) = \Theta\left(\frac{n(n+1)}{2}\right) = \Theta(n^2).$$

The dynamic programming solution is considerably more efficient than the straightforward recursive solution.

The space complexity is determined by the space required for the array SC. This is $(n+1)^2$ array entries, i.e., $\Theta(n^2)$.

Aside: there is a formula for computing combinations in terms of factorials $\begin{pmatrix} n \\ m \end{pmatrix} = \frac{n!}{(n-m)!\, m!}$ which leads to an $O(n)$ algorithm (but that would not illustrate dynamic programming).

2. (Kingston, Exercise 4.14) **The subset sum problem.** The following is a simple example of the problems that arise in making efficient use of a limited storage space, such as a computer's memory. Consider a set of n distinct items

$$A = \{a_1, a_2, \ldots, a_n\},$$

where each item a_i has a size, $s[i]$, which is a positive integer. The sizes of items are not necessarily distinct. The problem is to find a subset of A whose total size (that is, the sum of the sizes of its elements) is as large as possible, but not larger than a given integer *C*, the capacity of the storage space.

This problem may be solved by generating the 2^n subsets of *A*, eliminating all those whose total size exceeds *C*, and returning a remaining subset of maximum size. If we let T be a subset of $1..n$ and define the size of T by

$$size(T) \quad = \quad \sum_{j \in T} s[j]$$

then we want to calculate

$$\textsc{SubsetSum}(n, C) \quad = \quad max\{size(T) \mid T \subseteq 1..n \wedge size(T) \leq C\}$$

Unfortunately, basing an algorithm directly on this definition has exponential complexity because there are 2^n possible subsets of $1..n$. The problem is to find an algorithm which is substantially faster if *C* is not too large, and determine the complexity of the algorithm.

(a) Devise an (inefficient) recursive program (or just define a recurrence if you prefer) $\textsc{SubsetSum}(i, c)$ to find the size (only) of a maximal subset sum of the elements $a_1, a_2, ..., a_i$ that is no greater than c.

Hint: a_i is either in or out. Divide the problem into these two cases, and solve the resulting smaller instances.

Sample solution. The following recursive algorithm computes the maximal subset sum. The initial call on the procedure is of the form $\textsc{SubsetSum}(n, C)$.

If the set is empty ($i = 0$) then the maximal subset sum must be zero. That gives the first alternative in the procedure.

If the size of a_i is greater than c, then there is no way it can be included. That gives the second alternative in which we determine the maximal subset sum of elements with indices less than i, and total bound c.

If the size of a_i is less than or equal to c, then it may be included. To determine whether or not including a_i gives a maximal subset sum, we try both alternatives of including and excluding a_i. For the case when a_i is included, we use a recursive call to calculate the maximal subset sum of elements with indices less than i, and total bound $c - s[i]$. The value returned plus the size of a_i gives the maximal subset sum of the first i

elements when a_i is included ($with$). For the case when a_i is excluded, we use a recursive call to calculate the maximal subset sum of elements with indices less than i, and total bound c. The value returned is the maximal subset sum of the first i elements for the case when i is excluded ($without$). The maximal subset sum of the first i elements is then just the maximum of these two possible alternatives.

```
SUBSETSUM(i, c)
 1  // Pre: i ≤ n
 2  if i == 0
 3      return 0
 4  else if s[i] > c
 5      // we cannot possibly include item i
 6      return SUBSETSUM(i − 1, c)
 7  else // s[i] ≤ c
 8      with = SUBSETSUM(i − 1, c − s[i]) + s[i]
 9      without = SUBSETSUM(i − 1, c)
10      return MAX(with, without)
```

(b) Give a tight bound on the worst-case time complexity of your recursive algorithm. Hint: Consider the case when c is greater than the sum of the sizes of all the elements in A.

Sample solution. If $i = 0$ then SUBSETSUM takes constant time:

$$T(0) = \Theta(1).$$

For $i > 0$, if we assume that c is greater than the sum of the sizes of the first i elements of A, then the last branch of the **if** is always chosen. That gives the worst case because the last branch contains the same call as the middle branch as well as a second call. The last branch contains two recursive calls with first parameter $i - 1$, plus statements of constant time complexity. Hence the worst-case time complexity of the above algorithm is captured by the following equation.

$$T(i) = 2T(i-1) + \Theta(1) \quad \text{for } i > 0.$$

We can solve this by iteration as follows:

$$\begin{aligned}
T(i) &= 2T(i-1) + \Theta(1) && \text{for } i \geq 1 \\
&= 2(2T(i-2) + \Theta(1)) + \Theta(1) && \text{for } i \geq 2 \\
&= 2^2T(i-2) + 2\Theta(1) + \Theta(1) && \text{for } i \geq 2 \\
&= 2^2(2T(i-3) + \Theta(1)) + 2\Theta(1) + \Theta(1) && \text{for } i \geq 3 \\
&= 2^3T(i-3) + 2^2\Theta(1) + 2\Theta(1) + \Theta(1) && \text{for } i \geq 3 \\
&\vdots \\
&= 2^kT(i-k) + \sum_{j=0}^{k-1} 2^j\Theta(1). && \text{for } i \geq k
\end{aligned}$$

Substituting $i - k = 0$, that is, $k = i$, in this gives,

$$T(i) = 2^iT(0) + \sum_{j=0}^{i-1} 2^j\Theta(1) = 2^i\Theta(1) + (2^i - 1)\Theta(1) = \Theta(2^i).$$

Hence, the time complexity of above algorithm is $\Theta(2^i)$.

(c) What is the worst-case space complexity of your recursive algorithm?

Sample solution. The space complexity is determined by the depth of nesting of recursive calls. As a call with parameter i only makes calls with parameter $i - 1$, the maximum depth of nesting is i. Each call only requires constant space. Hence, the space complexity is $\Theta(i)$.

(d) Give a dynamic programming algorithm that matches your recursive algorithm.

Sample solution. If we examine the calling dependencies in the above procedure, we find that the recursive calls on SUBSETSUM have a first argument that is one smaller than i and a second argument that is less than or equal to c. That is SUBSETSUM(i, c) depends on SUBSETSUM$(i-1, c')$ for c' such that $0 \leq c' \leq c$. This gives a suitable order for the dynamic programming version of the algorithm. It uses a table ($(n+1) \times (C+1)$) of results for the solutions of the subset sum problem. It begins by initialising the table for empty sets to zero.

```
SUBSETSUM(n, C)
 1  for ci = 0 to C
 2      table[0, ci] = 0
 3  for i = 1 to n
 4      for ci = 0 to C
 5          if s[i] > ci
 6              // Cannot include item i
 7              table[i, ci] = table[i − 1, ci]
 8          else
 9              // s[i] ≤ ci
10              with = table[i − 1, ci − s[i]] + s[i]
11              without = table[i − 1, ci]
12              table[i, ci] = MAX(with, without)
13  return table[n, C]
```

(e) What is the worst-case time complexity of your dynamic programming algorithm?

Sample solution. The first loop has time complexity

$$\sum_{ci=0}^{C} \Theta(1) = \Theta\left(\sum_{ci=0}^{C} 1\right) = \Theta(C).$$

The body of the inner loop of the second loop consists of an **if** statement that has two branches each of constant time. Hence the worst case of the body of the inner loop is constant time, and the inner loop has time complexity

$$\sum_{ci=0}^{C} \Theta(1) = \Theta(C).$$

Hence the outer loop has time complexity

$$\sum_{i=1}^{n} \Theta(C) = \Theta\left(\sum_{i=1}^{n} C\right) = \Theta(n \cdot C).$$

Hence the time complexity of the dynamic programming algorithm is $\Theta(n \cdot C)$.

(f) What is the worst-case space complexity of your dynamic programming algorithm?

Sample solution. The space requirements for the $(n+1)$ by $(C+1)$ table — assuming dynamic allocation — are $\Theta(n \cdot C)$.

(g) Extend your dynamic programming algorithm to return a (there may be more than one) maximal subset. You should add an array R to record whether or not each item is included, i.e., $R[i]$ is TRUE if and only if the ith element is included in the solution.

Hint: Having computed the size of the maximal subsets, start from the nth item and determine whether or not it should be included. The solution is not necessarily unique.

Sample solution. From the recurrence that defines the values in the table we know that if

$$table[i, ci] = table[i-1, ci - s[i]] + s[i]$$

then a maximal set can be achieved by including the ith element, and if

$table[i, ci] = table[i-1, ci]$

then a maximal set can be achieved by excluding the ith element. Note that both these conditions can be true simultaneously, so that there may be more than one maximal set.

The following code can be used to determine a suitable subset with the maximal sum. The code can be inserted in the dynamic programming version of SUBSETSUM just before the final return.

```
1  ci = table[n, C]
2  for i = n downto 1
3      if table[i, ci] == table[i − 1, ci]
4          // exclude the item
5          R[i] = FALSE
6      else
7          // include the item
8          R[i] = TRUE
9          ci = ci − s[i]
```

The extraction of a maximal subset starts by setting ci to the size of the maximal subsets. We have chosen the simpler of the two possible tests. If it is true, the item can be excluded and a maximal set formed from the remaining elements. If it is false, the item must be included to form a maximal set, and the elements (with smaller indices) chosen to be included in the set must have total size $ci - s[i]$.