COMP4500/7500 Advanced Algorithms & Data Structures
Tutorial Exercises 6 (2014/2)*

School of Information Technology and Electrical Engineering, University of Queensland

September 2, 2014

This material aims to familiarise you with dynamic programming algorithms. A good treatment of dynamic programming may be found in CLRS chapter 15; CLR chapter 16.

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.

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

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

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

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

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

   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) $\text{SUBSETSUM}(i, c)$ 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.

   (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.

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

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

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

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

(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.