CSc 3102: Dynamic Programming

Supplementary Notes

- Recursion
- Dynamic Programming

Introduction

Definition 1. Dynamic programming is an algorithmic technique in which an optimization problem is solved, often recursively, by caching subproblem solutions. The solutions to the subproblems overlap.

Definition 2. Memoization is an algorithmic technique which saves (memoizes) a computed answer for later reuse, rather than recomputing the answer. The term comes from "memo".

Dynamic programming may use memoization, a top-down approach, or tabulation a bottom-up approach. That is memoization is top-down dynamic programming while tabulation is bottom-up dynamic programming.

Fibonacci Numbers

Definition 3. A **Fibonacci number** is a member of the sequence of numbers such that each number is the sum of the preceding two. The first five numbers are 1, 1, 2, 3, and 5.
Formally, the n^{th} Fibonacci number is

- $F(n) = F(n-1) + F(n-2),\ where\ \ F(1) = 1\ \ and\ \ F(2) = 1$

Iterative Version

Discuss the iterative implementation in class. How much space is used to store values when computing $F(n)$. Is it O(1)?

Recursive Version

```
Algorithm: fib(n)
{ n - a positive integer}
   if n = 1  or n = 2
      return 1
   else
      return fib(n-1) + fib(n-2)
```

Elegant solution but how much space will this use? Considerably more. Successive recursive calls cause stack frames (activation records) to be mounted on the runtime stack. These are not removed until the recursion begins to unwind. (Draw a box trace diagram to illustrate this).

Dynamic Programming Versions

A dynamic programming version does a lookup in a table to determine whether the n^{th} value was already calculated. If so, it returns it. If not, it calculates it and saves it. This saves computation time; however, there is a trade-off being made. Additional storage is required to cache all calculated values. There are two approaches. One approach is memoization, a top-down approach, and the other is the tabulation, a bottom-up approach. We give the pseudocode descriptions of both approaches for calculating the n^{th} term of the Fibonacci sequence.

Memoization

```
Algorithm: memFib(n)
{ n - a positive integer
  table - previously calculated values}
  if table[n] <> 0
     return table[n]
  if n = 1  or n = 2
     table[n-1] <- 1;
     return 1
  else
     table[n] <- memFib(n-1) + memFib(n-2)
     return table[n]
```

Tabulation

Another approach to dynamic programming is to use tabulation, a bottom-up approach. The goal is to fill the table and this is usually done iteratively.

```
Algorithm: tabFib(n)
{ n - a positive integer
  table - previously calculated values
  tabSize - current table size
  table[0] = 1 and table[1] = 1}
  if n <= tabSize
     return table[n-1]
  for i <- tabSize:n
        table[i] <- table[i-2] + table[i-1]
  return table[n-1]
```

The Knapsack Problem

Definition 4. The Knapsack Problem is an algorithmic problem in combinatorial optimization in which the goal is to fill a container with various items chosen in such a way to maximize the value of the selected items. Given a set of items each of which has an associated *weight* and *value* and a knapsack with a specified capacity, the Knapsack Problem is one of finding a subset of items from the set that optimizes the value subject to the capacity constraint of the knapsack.

Suppose the set of pairs $X = \{(w_1, v_1), (w_2, v_2), \cdots, (w_n, v_n)\}$ represent n items and the knapsack has a capacity of W_{max}, then the problem is one of finding $S \subseteq X$ such that $OPT\left(\sum_{j \in S} v_j\right)$ and $\sum_{j \in S} w_j \leq W_{\max}$. Each item $x_i \in X$ has a weight w_i and a value v_i.

Continuous Knapsack

In the continuous variation of the problem, fractional amounts of different materials may be chosen to maximize the value of the selected materials. This variant of the problem can be solved in polynomial time using a greedy algorithm. When selecting items, the choice is based on value per weight, $\frac{v_i}{w_i}$. A fraction of the last item may be chosen to fill the knapsack.

Discrete Knapsack

In the classic variation of the problem, also called the 0/1 Knapsack Problem, the items are indivisible and all or none of each item may be chosen while attempting to maximize the value of the selected items. A pseudo-polynomial time Dynamic Programming algorithm can be used to solve the 0/1 Knapsack problem. We construct an $n \times m$ table, where m is $W_{max} + 1$. The table is then filled using the piece-wise function below.

$$V[i, j] = \begin{cases} \max\left(V[i-1, j], V[i-1, j-w_i] + v_i\right), & w_i \leq W_j \\ V[i-1, j], & otherwise \end{cases} \tag{1}$$

Example 1. Suppose $X = \{(2, 3), (3, 2), (5, 4)\}$ represents three items denoted by their weight-value pairs that we would like to fit into a knapsack with a capacity of 9.

Recursion

In class we will do box traces of several recursive algorithms. We will discuss space complexity of recursion when the parameters require constant and non-constant space. We will discuss an elementary example of mutual recursion. We will also discuss a simple tail-recursive function and a related non-tail-recursive function.

A *tail-recursive* algorithm is one whose only recursive call is its last call. Most modern compilers are optimized to recognize and convert tail-recursive functions to iterative ones. An iterative solution, one that uses a loop, runs in one stack frame and generally saves space relative to its recursive equivalent. The version of the Fibonacci algorithm on page 2, *fib*, is non-tail recursive. Observe the addition in the last return statement. This means that stack frames, including the one from the initial call, in the recursion chain must be preserved in memory and that the return statement in the initial frame is executed only after all subsequent stack frames created from successive calls have been popped from memory as the algorithm begins to backtrack. This can be very taxing on memory. Here is a tail recursive version of the Fibonacci algorithm.

```
Algorithm: tRFib(i, prev, cur, n)
{ n - the position of the term to be generated
  i - position of the current term
  prev - term preceding the current term
  cur - the current term
}
if i = n
   return cur
cur <- cur + prev
return tRFib(i+1,cur-prev,cur,n)

tRFibCaller(n)
{ A wrapper function that calls tRfib
  n - a positive integer
}
if n = 1 or n = 2
   return 1
return tRFib(3,1,2,n)
```

Observe how this can easily be rewritten using a loop.

```
Algorithm: iterFib(n)
{ n - the position of the term to be generated
i - position of the current term
prev - term preceding the current term
cur - the current term
}
if n = 1 or n = 2
   return 1
i <- 3
prev <- 1
cur <- 2
while i <> n
   i <- i + 1
   cur <- cur + prev
   prev <- cur - prev
endWhile
return cur
```

Problem 1. Draw a box trace diagram for *fact(5)*, the factorial function evaluated at $n = 5$. How many calls are made to *fact*?

$$fact(n) = \begin{cases} undefined, & n < 0 \\ 1, & n = 0 \ or \ 1 \\ n \times fact(n-1), & otherwise \end{cases} \tag{2}$$

Problem 2. Draw a box trace diagram for α*(2,1)*, the Ackermann function evaluated at $m = 2$ and $n = 1$. How many calls are made to α? List the calls for each $m - n$ pair and its multiplicity. Draw a box trace diagram for a memoized version of the function. How many fewer calls will this version make?

$$\alpha(m,n) = \begin{cases} n+1, & m = 0 \\ \alpha(m-1,1), & n = 0 \\ \alpha(m-1,\alpha(m,n-1)), & otherwise \end{cases} \tag{3}$$

Problem 3. Suppose $X = \{(2,3),(3,5),(4,7),(5,6)\}$ represents four items denoted by their weight-value pairs that we would like to fit into a knapsack with a capacity of 8.

1. What items and which fraction of each item selected must be placed into the knapsack to optimize the total value of the items that are selected? Show how the optimal value is obtained.

2. Suppose the items are indivisible, which items must be selected to optimize the total value? Trace the action of the dynamic programming-based solution to this problem by showing the initial contents of the table and how each entry of the table is updated as the algorithm is executed.