FIT2004: Supplementary notes for lecture 4

Dynamic Programming: Part I

DANIEL ANDERSON [1]

A powerful technique that we have already encountered for solving problems is to reduce a problem into smaller problems, and to then combine the solutions to those smaller problems into a solution to the larger problem. We have seen this for example in the recursive computation of Fibonacci numbers, and in the divide-and-conquer algorithms for sorting (Merge Sort and Quicksort). In some situations, the smaller problems that we have to solve might have to be solved multiple times, perhaps even exponentially many times (for example, recursive computation of the Fibonacci numbers). Dynamic programming involves employing a technique called *memoisation*, where we store the solutions to previously computed sub-problems in order to ensure that repeated problems are solved only once. Dynamic programming very frequently arises in optimisation problems (minimise or maximise some quantity) and counting problems (count how many ways we can do a particular thing).

Summary: Dynamic Programming: Part I

In this lecture, we cover:

- The key ideas behind dynamic programming
- Memoisation applied to computing Fibonacci numbers
- The "Rod Cutting" problem
- Top-down vs. bottom-up approach
- Reconstructing solutions via backtracking

Recommended Resources: Dynamic Programming

- `http://users.monash.edu/~lloyd/tildeAlgDS/Dynamic/`
- CLRS, Introduction to Algorithms, Chapter 15
- Weiss, Data Structures and Algorithm Analysis, Section 10.3
- Halim, Competitive Programming 3, Section 3.5

The Key Elements of Dynamic Programming

Repeated sub-problems and memoisation - Computing Fibonacci numbers

The essence of dynamic programming is to solve a problem by breaking it into smaller sub-problems and combining the solutions of those sub-problems into a solution to the greater problem. In contrast with the divide-and-conquer method which also employs the same idea, dynamic programming applies when the sub-problems **overlap** and are repeated multiple times. Recall the example of computing Fibonacci numbers recursively. Fibonacci numbers are defined by recurrence relation

$$F_n = F_{n-1} + F_{n-2}, \qquad F_0 = 0, \quad F_1 = 1,$$

and can be computed recursively with a very simple algorithm.

[1] Faculty of Information Technology, Monash University: `daniel.anderson@monash.edu`. These notes are based on the lecture slides developed by Arun Konagurthu for FIT2004.

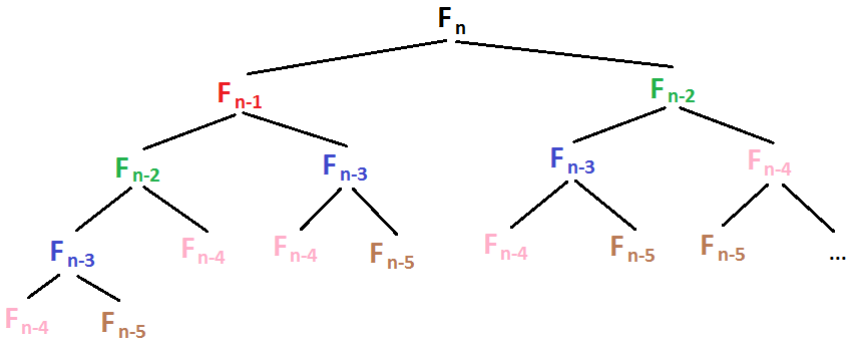
Algorithm: Compute Fibonacci Numbers

```
1: function FIBONACCI(N)
2:     if N ≤ 1 then
3:         return N
4:     else
5:         return fibonacci(N-1) + fibonacci(N-2)
6:     end if
7: end function
```

Although simple, this algorithm is extremely inefficient. What is its time complexity? Well, the amount of work taken T_n to compute the value of F_n is the sum of the times taken to compute the values of T_{n-1} and T_{n-2}, so the time complexity is given by the solution to the recurrence

$$T_n = T_{n-1} + T_{n-2}.$$

Hopefully we recognise the solution to this recurrence, its just the Fibonacci numbers! So the time complexity to compute F_n is $O(F_n)$. Hopefully you remember from your mathematics study, that the Fibonacci numbers grow exponentially, specifically they grow asymptotically like φ^n where φ is the golden ratio. This is a very interesting discussion itself, but we'll leave it there for now. The point is that the above algorithm has exponential time complexity! How does it get so bad? What exactly are we doing so wrong?



The call tree resulting from the recursive Fibonacci function for computing F_n.

The figure above depicts what happens when we try to compute F_n. F_n and F_{n-1} are computed once, F_{n-2} is computed twice, F_{n-3} is computed three times, F_{n-4} is computed five times... In general, F_{n-k} is computed F_{k+1} times, so the leaves of the call tree are computed exponentially many times, which explains why the algorithm is so slow.

The solution to this problem is simple but demonstrates the most powerful, core idea of dynamic programming. Instead of allowing ourselves to compute the same thing over and over again, we will simply store the result of each computation in a table, and if we are required to compute it again, we can look it up and return it straight from the table without any recomputation. This technique is called **memoisation** (no, that is not a typo).

Key Ideas: Memoisation

Memoisation is the process of remembering the solutions to previously computed sub-problems and storing them in a table for later lookup. If the same sub-problem needs to be computed multiple times, it need only be computed once, and then subsequently can be looked up from the memo table at no additional cost.

Dynamic programming exploits problems that exhibit overlapping or repeated sub-problems by using memoisation to avoid doing redundant work.

A memoised implementation of the recursive Fibonacci number calculation is shown bellow.

Algorithm: Memoised Fibonacci Numbers

```
1: function FIBONACCI_MEMO(N, memo)
2:     if N ≤ 1 then
3:         return N
4:     else if memo[N] != -1 then
5:         return memo[N]
6:     else
7:         memo[N] = fibonacci_memo(N-1, memo) + fibonacci_memo(N-2, memo)
8:         return memo[N]
9:     end if
10: end function
11:
12: function FIBONACCI(N)
13:     Set memo[1..N] = [-1,-1,...]
14:     return fibonacci_memo(N, memo)
15: end function
```

We store the Fibonacci numbers that have been computed in the memoisation table `memo`, and return straight from the table whenever we encounter a problem that we already have the solution to. The initial value of -1 in the table is used to indicate that we have not computed that number yet. If we were planning on computing lots of Fibonacci numbers, we could even keep the `memo` table between calls. Since each problem is never computed more than once and there are N problems, the total time complexity of this implementation is $O(N)$.

The approach that we have implemented above is an example of "top-down" dynamic programming since we first ask for the solution to the largest problem and recursively ask for the solutions to smaller problems while filling in the memoisation table. An alternative implementation would be to start from the smaller problems and work our way up to the big problems. This would be the "bottom-up" approach.

Algorithm: Bottom-up Fibonacci Numbers

```
1: function FIBONACCI(N)
2:     Set fib[0..N] = [0,1,0,0,...]
3:     for i = 2 to N do
4:         fib[i] = fib[i-1] + fib[i-2]
5:     end for
6:     return fib[N]
7: end function
```

We will explore the differences between the top-down and bottom-up approaches soon.

Optimal Substructure - The "Rod Cutting" Problem

A more interesting example of dynamic programming can be illustrated with the so called "rod cutting" problem. In this problem, we are given a steel rod with a certain length L cm. We can cut the rod into smaller pieces of any size (or keep it as a single rod) and sell all of the pieces. If we know the prices p_i for $1 \leq i \leq L$ that a rod of length i inches can be sold for, the problem is to determine the best way to cut up the rod to make the maximum amount of money. For example, consider the table of prices for rods of certain sizes given bellow.

length i	0	1	2	3	4	5	6	7	8	9	10
price p_i	0	1	4	6	8	8	9	15	17	18	19

By inspection, we can easily see that the optimal way to cut of rod of length 10cm is into two rods of length 2cm and 8cm which will sell for a total of \$21. If we had significantly longer rods, we would probably not be able to find the answer by hand so easily. Dynamic programming will help us to solve this problem.

Identifying the sub-problems

In order for dynamic programming to be applicable, the problem must exhibit sub-problems with **optimal substructure**. This means that solutions to the sub-problems must be able to be combined to form a solution to the larger problem. The optimal substructure for the rod cutting problem can be observed by thinking about what happens if we cut a single piece of rod off a larger piece. Suppose we have a rod of length L, then either we are going to keep it in tact and sell it for $p[L]$, or we are going to cut it somehow. If the optimal way to cut a length L rod involves cutting it into k pieces such that

$$L = i_1 + i_2 + i_3 + ...i_k$$

then it must be the case that $i_2 + i_3 + ... + i_k$ is also the optimal way to cut a rod of length $L - i_1$ into pieces. If it were not, then the above solution could be improved by substituting the true optimal way to cut a rod of length $L - i_1$ for the supposed partition $i_2 + i_3 + ...i_k$. This is the optimal substructure of the rod cutting problem.

Key Ideas: Optimal Substructure

A problem exhibits optimal substructure if the solution to smaller sub-problems can be combined to form a solution to the larger composite problem. Dynamic Programming exploits problems with optimal substructure to find efficient solutions to problems with otherwise daunting search spaces.

Deriving the recurrence

Given the optimal substructure that we have found, we can derive a recurrence relation for the optimal selling price of a rod of length L. Let r_L denote the maximum revenue that we can obtain from a rod of length L, then

$$r_L = \max_{1 \leq i \leq L} (p_i + r_{L-i}), \qquad r_0 = 0.$$

Implementing a solution

Using the above recurrence, we could easily implement a recursive function to compute r_L given the prices (p_i). Alone however, this would yield a very inefficient solution for the same reasons we discussed regarding the Fibonacci numbers. In order to prevent us from recomputing the solutions to the repeated sub-problems, we should employ memoisation.

Algorithm: Memoised Rod Cutting

```
1:  function CUT_ROD_MEMO(L, p, cut_memo)
2:      if L = 0 then
3:          return 0
4:      else if cut_memo[L] != -1 then
5:          return cut_memo[L]
6:      else
7:          Set best = p[L]
8:          for i = 1 to L - 1 do
9:              best = max(best, p[i] + cut_rod_memo(L - i))
10:         end for
11:         cut_memo[L] = best
12:         return best
13:     end if
14: end function
15:
16: function CUT_ROD(L, p)
17:     Set cut_memo[1..L] = [-1,-1,-1,...]
18:     return cut_rod_memo(L, p, cut_memo)
19: end function
```

Just as we did for Fibonacci numbers, we use -1 to denote a sub-problem that has yet to be computed. Rod cutting could also be implemented in a bottom-up fashion like so.

Algorithm: Bottom-up Rod Cutting

```
1: function CUT_ROD(L, p)
2:     Set r[0..L] = [0,0,0,...]
3:     for i = 1 to L do
4:         Set best = p[i]
5:         for j = 1 to i - 1 do
6:             best = max(best, p[j] + r[i-j])
7:         end for
8:         r[i] = best
9:     end for
10:    return r[L]
11: end function
```

In the bottom-up version, we compute the solutions to the sub-problems of size $i = 0, 1, 2, 3, ...$ in order. The reason for this is very important: The sub-problem of size i depends on knowing the solution to all of the sub-problems of size $j < i$, so we must compute them first, or we would not be able to compute the solution for i.

Since there are a total of N sub-problems and computing the solution to each sub-problem involves looping over all previous sub-problems, the time complexity of our dynamic programming solution to the rod cutting problem is

$$\begin{aligned} T_N &= \sum_{i=1}^{N} i \\ &= \frac{N(N+1)}{2} \\ &= O(N^2) \end{aligned}$$

Top-down vs. Bottom-Up Dynamic Programming

Both styles of dynamic programming, top-down and bottom-up involve using a table to remember the solutions to previously computed sub-problems. Their behaviour is quite different however. In the top-down approach, the memoisation table is filled on demand as the particular sub-problems that are required are computed. In the bottom-up approach, we fill the solution table one problem at a time in an order which ensures that any dependent sub-problems have already been computed before they are needed. The order in which we will the table in the bottom-up approach is the *reverse topological order* of the dependency graph of the sub-problems (we will learn about topological orderings later in the unit.) Most of the time, figuring out the order in which the sub-problems must be computed is simple, but in rare cases it might be more difficult.

The majority of the time, the two dynamic programming styles are equivalent and equally as good, but there are certain situations in which it may be preferable to favour one over the other. As eluded to above, one reason that the top-down approach might be preferable in some cases is that we do not need to know a-priori the order in which the sub-problems need to be computed, as the recursion takes care of ensuring that sub-problems are made available when required. The top-down approach also boasts the advantage that sub-problems are only computed if they are actually required, while the bottom-up approach strictly computes the solution to every possible sub-problem. If the solutions to only a small number of the possible sub-problems are required, then the top-down approach will avoid computing the ones that are not needed.

On the contrary, the bottom-up approach avoids recursion altogether and hence should be faster if sub-problems are frequently revisited since we drop the overhead on the program's call stack required to make the recursive calls. The bottom-up approach also allows us to more easily exploit the structure of specific dynamic programs in situations where we can improve the time and space complexity of the resulting program. We will see one such example, the space saving trick in the next section, which allows us to reduce the space complexity of various dynamic programs from $O(N^2)$ to $O(N)$, which is not possible with the top-down approach.

The following table summarises the pros and cons of both approaches.

Pros: Top-Down Approach	Pros: Bottom-Up Approach
1. No need to know the topological ordering of the sub-problem dependencies, as this is handled by the recursion 2. Avoids computing the solution to sub-problems that are not needed, hence faster if only a small number of sub-problems are visited 3. More intuitive for programmers who like to think recursively	1. Avoids the overhead of recursion, hence faster if sub-problems are frequently revisited 2. Allows for clever optimisations by exploiting particular properties of the problem that can not be done recursively (eg. space saving trick in the next section) 3. Might be more intuitive for programmers who don't like recursion

Reconstructing Optimal Solutions to Optimisation Problems

When we solved the rod cutting problem, we left out something rather important. Although we computed the maximum revenue that could be obtained by cutting a particular rod, we did not actually compute the other part of the solution, ie. a list of sizes into which the rod should actually be cut. We can extend the dynamic programming algorithm for the rod cutting problem to also construct and return a list of optimal cuts.

In general, there are two ways to approach constructing an optimal solution from a dynamic programming problem. One way is to simply *backtrack* through the entries of the table and figure out which choices were made that lead to the given solution. For example, in the rod cutting example given earlier, the optimal revenue was \$21, so we could backtrack through the entries of the dynamic programming table over values of `i = 1 to N` until we found a cut `i` such that `p[i] + r[L-i] = 21`. When we find such an `i`, the optimal substructure of the problem tells us that there must be a cut of size `i`. We then subtract `i` from `L` and repeat until we reach `L = 0` at which point we know we have constructed the entire solution. An example implementation using the bottom-up approach is given bellow.

Algorithm: Bottom-up Rod Cutting with Solution Reconstruction

```
1:  function CUT_ROD(L, p)
2:      Set r[0..L] = [0,0,0,...]
3:      for i = 1 to L do
4:          Set best = p[i]
5:          for j = 1 to i - 1 do
6:              best = max(best, p[j] + r[i-j])
7:          end for
8:          r[i] = best
9:      end for
10:     Set cuts = []
11:     Set i = L
12:     while i > 0 do
13:         Set best_cut = i
14:         Set best_value = p[i]
15:         for j = 1 to i - 1 do
16:             Set value = p[j] + r[i-j]
17:             if value > best_value then
18:                 best_cut = j
19:                 best_value = value
20:             end if
21:         end for
22:         cuts.append(best_cut)
23:         i -= best_cut
24:     end while
25:     return r[L], cuts
26: end function
```

The second approach often used for reconstructing solutions is to keep a second table in addition to the memoisation or bottom-up dynamic programming table which is used to remember the optimal decisions that were made by each sub-problem. Once the optimal value is found, the solution can then be reconstructed by looking up the choices that were made in this table.

Algorithm: Bottom-up Rod Cutting with Solution Reconstruction using Decision Table

```
1: function CUT_ROD(L, p)
2:     Set r[0..L] = [0,0,0,...]
3:     Set opt[1..L] = [0,0,0,...]
4:     for i = 1 to L do
5:         Set best_cut = i
6:         Set best_value = p[i]
7:         for j = 1 to i - 1 do
8:             Set value = p[j] + r[i-j]
9:             if value > best_value then
10:                best_cut = j
11:                best_value = value
12:            end if
13:        end for
14:        opt[i] = best_cut
15:        r[i] = best_value
16:    end for
17:    Set cuts = []
18:    Set i = L
19:    while i > 0 do
20:        cuts.append(opt[i])
21:        i -= opt[i]
22:    end while
23:    return r[L], cuts
24: end function
```

As can be seen from their pseudocode, both approaches are very similar. The first approach is advantaged by the fact that is uses less memory, since a second table is not required to remember the optimal cuts. The first approach should also usually be faster as a consequence.

The second approach however is arguably simpler to understand and hence might be preferable for some programmers. The second approach also allows us to do some more advanced optimisations that involve dynamically adjusting the search space for a particular sub-problem based on the optimal choices of previous sub-problems. These optimisations are however much more advanced than anything that we will cover in this unit.