ArsDigita University
Month 5: Algorithms - Professor Shai Simonson

Lectures – Second Two weeks

In this half of the course we discuss two topics. The first is techniques of algorithm design, comparing recursion (divide and conquer) to dynamic programming (bottom-up) and greedy strategies. The second is the idea of NP-complete problems and discovering the frontier between variations of problems – which are solvable in polynomial time and which are NP-complete.

Recursion versus Dynamic Programming– Fibonacci Numbers

We are ready now to move away from different applications of algorithms and concentrate on general techniques that show up in a variety of different applications. We start by comparing recursion with dynamic programming. The best way to appreciate the difference is to look at some simple examples before we attack more interesting problems.

Everyone know that the Fibonacci numbers are defined recursively as F(n) = F(n-1) + F(n-2), and F(1) = 1 F(2) = 1. However, a direct recursive implementation has running time bounded by a similar recurrence equation with an exponential solution. It is not hard to see that the time is between 2^n and 2^(n/2). The exact solution can be found in Rosen, Discrete Math.

Of course we would normally generate F(n) by a loop starting at 1 and moving up. This method takes O(n) time. Why is so much faster than the recursive method? With tail recursion, iteration and recursion converge, but with true recursion, there is the possibility that a subproblem will be called more than once. In the Fibonacci example this happens an exponential number of times. By running a loop from the bottom up, we guarantee that each subproblem is computed exactly once. As long as the total number of subproblems is polynomial in n, O(n) in this is case, then we are fine.

In a typical dynamic programming versus recursion scenario, the choice is between recursion that will compute some subproblems many times and some not at all, and dynamic programming that will compute each subproblem exactly once. Dynamic programming is a helpful alternative whenever the total number of subproblems is polynomial in the size of the input. The tricky part is trying to find a recursive solution to the problem where the total of subproblems is bounded this way.

Dynamic programming algorithms almost always have a simple array (sometimes multidimensional) to store the results of the problems. The generation of the subproblems in some order is done either by simple nested loops or if necessary a queue. A queue is used when the precise subproblems depend on that last ones that were just generated, rather than on some specific order.

Recursive algorithms use an implicit stack, and the control is handled by the underlying recursive structure and the compiler that handles it. I like to think of queues, breadth first processing and dynamic programming in one paradigm, with depth first processing, recursion and stacks in the other.

Binomial Coefficients

For example consider binomial coefficients C(n,m) defined to be C(n-1, m) + C(n-1, m-1), n>m>0, else equals 1. To calculate C(6,3) recursively we call C(5,2) + C(5,3) which calls C(4,2)+ C(4,1) and C(4,3) + C(4,2). This continues and in the end we make the following calls this many times.

C(6,3) – 1
C(5,3) – 1
C(5,2) – 1
C(4,3) – 1
C(4,2) – 2
C(4,1) – 1
C(3,3) – 1 *
C(3,2) – 3
C(3,1) – 3
C(3,0) – 1 *
C(2,2) – 3 *
C(2,1) – 5
C(2,0) – 3 *
C(1,1) – 6 *
C(1,0) – 6 *

The calls with stars are bottom level calls which return 1, and the sum of these is C(6,3) = 20. Note that we do NOT call every possible subproblem. In particular, C(6,4), C(5,1), C(4,0) among many others are never called at all.

Nevertheless, if you analyze the recurrence equation we get T(x) = T(x-1) + T(x-2), where x = m+n, and this is the same as the bad Fibonacci time algorithm of the previous paragraph.

There is of course a faster way to compute binomial coefficients from the bottom up and it generates Pascal's triangle through the nth row.

```
For j = 1 to n {B(j, 0) = B(j,j) = 1};
For j = 2 to n {
        For k = 1 to j-1 {
                B(j,k) = B(j-1, k) + B(j-1, k-1);
        }
}
```

It is pretty clear that this algorithm uses at most n^2 time, but it does actually compute every subproblem B(j,k) for all j and k such that n>=j>=k.

This is a classic success of dynamic programming over recursion.

Summary

Both recursion and dynamic programming are based are a recursive solution to the problem at hand. Recursion assumes a recursive implementation which may call duplicate subproblems too many times. Dynamic Programming avoids this pitfall by methodically generating every possible subproblem exactly once. It is crucial that the total number of subproblems be polynomial in n.

Now let's head toward to some real life examples from a variety of applications starting with graph algorithms, and moving toward mathematical, geometric, optimization, and parsing algorithms

All Pairs Shortest Path Algorithm – An O(n^4) Dynamic Programming Attempt

In this problem we are trying to find the shortest paths between every pair of nodes in a graph. This means the construction of n shortest path trees. Now we can certainly do this by running our single source shortest path algorithms n times, but this takes O(ne log n) for Dijkstra's algorithm and

O(n^2e) for Bellman-Ford. We would like to get this down to O(n^3) with a dynamic programming technique.

Our first try is based on a recursive formulation for shortest paths.

Let d(x,y,k) be the shortest length path from node x to node y that uses k edges or less.

if k=0, d(x,y,0) = 0 if x=y else d(x,y,0) = MaxInt;

If k>0, d(x,y,k) = min { d(x, m, k-1) + weight(m,y) } for all nodes m

This means in English that the shortest path from x to y using k edges or less, is computable by calculating the shortest paths from x to a node m which is adjacent to y, using k-1 edges or less, and adding in the weight on the edge (m,y).

Note that implementing this directly gives us a hideous exponential time algorithm, hence we will try to calculate d(x,y,k) bottom up using a sequence of loops. We initialize d(x,y,0) and work up from there to d(x,y,1), d(x,y,2) … d(x,y, n) where n is the number of nodes in the graph. Each improvement from d(x,y,k-1) to d(x,y,k) needs a loop on k, and this loops must be computed for each pair x,y. IT results in a triple loop each of which is 1 to n, giving O(n^3). Since there are at most n-1 edges in any shortest path (no negative cycles allowed ensures this) we must do this improvement from d(x,y,0) to d(x,y,n-1), and we get O(n) repetitions of an O(n^3) process which results in O(n^4). The code can be found in your text on page 554-555.

To keep track of the actual shortest path trees, we need to store the appropriate parent pointers as we go along. We will have a shortest path tree for each node. The shortest path trees are stored in a two dimensional array p(x,y), where p(x,y) is the parent of y in the shortest path tree rooted at x. This means that if d(x,m,k-1) + weight(m,y) is the minimum over all nodes m, and hence d(x,y,k) = d(x,m,k-1) + weight(m,y), we must also set parent(x,y) = m. The text does not mention any of this, so be careful to study the details here.

Note that although the text leaves out the important but easy detail of keeping track of the shortest path trees, it does conclude this section with a neat explnation of how to knock the O(n^4) down to O(n^3 log n). This method which the authors call *repeated squaring,* is actually a variation on the ancient Egyptian way to do integer multiplication.

I will explain the Egyptian method in class, and leave the details of the application here for you to read in the text or to be discussed in recitation. The basic notion is that what the Egyptians did with multiplying integers can be applied to multiplying matrices, and our algorithm is really multiplying matrices in disguise.

Floyd-Warshall – An O(n^3) Improved Version of All Pairs Shortest Path Dynamic Programming Algorithm

This algorithm is an improvement to the previous algorithm and uses a recursive solution that is a little more clever. Instead of considering the shortest paths that use at most k edges, we consider the shortest paths that use intermediate edges in the set {1 .. k}. In recitation, generalizations of this will be discussed, where the skeleton of the algorithm is modified to compute solutions to a host of other interesting problems, including transitive closure. It turns out that the skeleton can be used for any problem whose structure can be modeled by an algebraic notion called a *closed semiring.*

Let d(x,y,k) be the shortest path from x to y using intermediate nodes from the set {1 .. k}. The shortest distance from x to y using no intermediate nodes is just weight(x,y). The shortest path from x to y using nodes from the set {1 .. k} either uses node k or doesn't. If it doesn't then d(x,y,k) = d(x,y,k-1). If it does, the d(x,y,k) = d(x,k,k-1) + d(k,y,k-1). Hence d(x,y,0) = weight(x,y), and d(x,y,k) equals min{d(x,y,k-1), d(x,k,k-1) + d(k,y,k-1)}.

This is very elegant and somewhat simpler than the previous formulation. Once we know the d(x,y,k) we can calculate the d(x,y,k+1) values. We therefore calculate the values for d(x,y,k) with a triply nested loop with k on the outside and x,y on the inside. Code can be found in your text on page 560, with a detailed example on 561. The parent trees are stored in a two dimensional array as before. There is a different to actually store and retrieve the paths, that is discussed on page 565 problem 26.2-6. This second method is one that is more commonly seen in dynamic programming methods, however we use the first method because it is consistent with how we would do it for n iteration of the single source shortest path algorithm. You will get plenty of practice with the other method. The example coming up next will show you the idea.

Matrix-Chain Multiplication – Another Example

In every dynamic programming problem there are two parts, the calculation of the minimum or maximum measure, and the retrieval of the solution that achieves this measure. For shortest paths this corresponds to the shortest path lengths and the paths themselves. The retrieval of the solution that achieves the best measure is accomplished by storing appropriate information in the calculation phase, and then running a recursive algorithm to extract this information after the calculation phase has ended. The example we are about to do is an excellent illustration.

As you know, matrix multiplication is associative but not commutative. Hence if you are multiplying a chain of matrices together there are many ways to order the pairs of matrix multiplications that need to get done. The number of ways to multiply n matrices is equal to the number of different strings of balanced parentheses with n-1 pairs of parentheses. We discussed this in discrete math (month 2) and recall that these numbers are called the Catalan numbers and also happen to represent the number of different spanning trees on n nodes. Also recall that the Catalan numbers are approximately O(4^n), which we proved in class in month 2. See problem 13-4 on page 262, for a sketch and review.

We would like to choose a parentheses structure on the matrices that minimizes the total number of scalar multiplications done. Recall that for each multiplication of two matrices, the number of columns in the left matrix must equal the number of rows in the right matrix. Multiplying two matrices of sizes m by n, and n by p, requires mnp scalar multiplications. This assumes we use the standard algorithm that just calculates each pf the n^2 entries with a linear time dot product. Therefore, depending on what order we do the matrix multiplications, the total number of scalar multiplications will vary.

Let M(x,y) be the minimum of multiplications to compute the product of the matrix x through matrix y inclusive. If we first multiply the arrays x through k, and k+1 through y, and then multiply their results, the cost is the sum of M(x,k) + row(x)column(k)column(y) + M(k+1, y). The actual M(x,y) will be the minimum. Our problem asks for M(1,n).

M(x,y) = minimum over all k from x to y-1 inclusive, of M(x,k) + row(x)column(k)column(y) + M(k+1, y). The base case is when x=y whence M(x,y) =0. If you calculate M(x,y) you get a large number of multiple subproblem computations. The recurrence equation gives the horrible looking $T(n) = (n-1) T(n-1) + n$.

However, the total number of subproblems is proportional to n^2. Hence we will use dynamic programming and calculate each subproblem exactly once storing the results in a 2-dimensional array called M. Each subproblem requires linear time for the minimum calculation of n other subproblems, so this gives an O(n^3) algorithm. We carefully order the computation so that all the subproblems needed for the next step are ready.

Each subproblem M(x,y) requires the results of all other subproblems M(u,v) where |u-v| < |x-y|. We compute M(x,y) = 0 when x=y, and then we compute M(x,y), where |x-y| = 1, etc.

The code is given in your text on page 306, but here is perhaps an easier to read version:

```
N = the number of arrays;
For  I = 1 to N set M(I,I) = 0;

For difference = 1 to N-1
For x = 1 to N-difference       {
y = x + difference;
            M(x, y) = MaxInt;
            For middle = x to y-1 {
Temp = M(x, middle) + M(middle+1, y) + row(x)column(middle)column(y) ;
If Temp < M(x,y) then M(x,y) = Temp; }
}
```

The analysis of this triple loop is reviewed in detail in problem 16.1-3 on page 309, and involves the sum of the first n squares. A simpler more naïve analysis notes that we never do worse than a triply nested loop each of which is O(n), thereby giving O(n^3). It turns out that the more careful analysis uing the sum of the first n squares is also O(n^3).

An example is done in your text on page 307 and we will do one in class as well.

Returning the Actual Ordering of the Arrays

It is useful not just to calculate the min cost M(x,y) but also to be able to retrieve what ordering of arrays give us that cost. In order to do this, we will remember in a two-dimensional array s(x,y) which one of the *middle* indices was the one that gave us the minimum cost. Hence after M(x,y) = Temp; we add the line s(x,y) = middle.

We can reconstruct the ordering with the following simple recursive algorithm. We actually call PrintOrder(s, 1, N).

```
PrintOrder (s, x, y);

        If x==y then Print x;
        Temp = s(x,y);
        Print '(';  PrintOrder(s, x, Temp);
        Print '*';
        PrintOrder(s, Temp+1, y); Print ')';
```

The analysis of PrintOrder is worst case O(n^2). This happens when the two recursive calls repeatedly split into sizes of n-1 and 1.

This idea of storing information during the dynamic programming and then retrieving the actual minimum cost order with a recursive lookup, is a theme commonly used. You will see it again and again in the forthcoming examples.

Polygon Triangulation – Another Dynamic Programming Problem

This problem is essentially a non-obvious generalization of the matrix order problem of the previous section. The similar structure of the two problems is hidden because one is a mathematical algorithm and one is geometric. This section emphasizes a mathematical idea that is useful in algorithms – that is, noticing when two different things are really the same.

The Polygon Triangulation asks for the best way to triangulate a given polygon. The *best* way is defined to be the minimum of the sums of some function of the triangles. This function can be area, perimeter, product of the sides etc.

The similarity to matrix multiplication order, is that the number of different triangulations of an n-sided polygon is equal to the number of different ways to order n-1 arrays with parentheses. The n-1 arrays have a total of n dimensions, and the polygon has a total of n points. If we associate each point p_j with a dimension d_j then if the function of each triangle is the product of the nodes, we get exactly the matrix multiplication order problem. In class, we will show an example illustrating the connection between parentheses, trees, arrays and polygons.

The technique in general is exactly the same as before except that in our new problem any function can replace the simple product of the nodes example that gives the old matrix order problem.

Cocke-Younger-Kasimi (CYK) Parsing Method – An O(n^3) Dynamic Programming Algorithm

This problem breaks new ground in our coverage of algorithms. It solves the problem of parsing strings in a *context free language*. It is used in the early stages of building a compiler. A compiler very simply can be split into three stages:

1. Token recognition – This groups the characters into larger *tokens* and is done using a finite state machine.
2. Parsing – This checks whether the tokens are syntactically consistent with the description of your programming language, given by a context free language. There are linear time methods to do this for restricted kinds of context free language. Almost every practical programming language can describe its syntax using these restricted types.
3. Translating – This generates the machine code and does the semantic interpreting of the correctly parsed code.

The CYK algorithm solves part two of this process for any context free language.

Fast Introduction to Context Free Grammars and Languages

A context free language is a collection of strings described by a *context free grammar*. In the application to compilers, each string represents a legal computer program. However, for our purposes we can consider each string a binary string. A context free grammar describes a context free language by the strings that it can generate. For example, the following context free grammar generates all binary strings with an even number 0's.

S → 0A | 1S | e
A → 0S | 1A | 0

The convention is that S is the start symbol, and we *generate* strings by applying the productions of the *grammar*. For example, the following sequence of productions generates the string 01110.

S → 0A → 01A → 011A → 0111A → 01110S → 01110.

The symbol *e* denotes the empty string. The idea is that we start from the start symbol and little by little generate all binary digits while the S's and A's disappear. The capital letters (S and A) are called *non-terminal* symbols and the alphabet (0's and 1's in this case) consists of *terminal* symbols. In a programming language the alphabet would include many more non-terminal and terminal symbols. Below is a fragment of the context free grammar from an old language called Pascal.

S → **program (** I)**;** B**; end;**
I → I**,**I | LJ | L
J → LJ | DJ | e
L → **a**|**b**|**c**| **…** |**z**
D → **0**|**1**|**2**| **…** |**9**

…

It means that every Pascal program stars with the word **program** and continues with an open paren followed by an identifier list (represented by the non-terminal symbol I). An identifier list is a sequence of identifiers separated by commas. A single identifier starts with a letter and continues with any letter or digit. Other common programming constructs including **if, while** etc. can be described similarly.

Every context free language is required to have exactly one non-terminal symbol on the left side of every production. There are less restrictive grammars (context sensitive) that allow multiple symbols on the left side, and they are harder to parse. The syntax of any programming language can be generated by a restricted type of context free language. The CYK algorithm takes a context free grammar and a string of terminal symbols, and gives a method of determining whether or not a candidate string can be generated by a particular context free grammar.

It is convenient for the purposes of CYK to assume that the context free grammar is given in a particular normal form, called *Chomsky Normal form*. You will learn in month 8 (theory of computation) that any context free language can be put into Chomsky Normal form. In this form, every production is of the form A → BC or A → a, where A, B and C are non-terminal symbols and a is a terminal symbol.

For example, the even number of zeros grammar would look like this in Chomsky Normal form:

S → ZA | OS | e
A → ZS | OA | 0
Z → 0
O → 1

(Actually it would look a little different because of the e-production at the start symbol – but that is a technical detail very far away from our main considerations here).

Let's look at an example and describe the general algorithm afterwards. Consider the grammar below in Chomsky Normal form.

S → AB | BC A → BA | 0 B → CC | 1 C → AB | 0

Then consider the string s =10010. This example comes from Hopcroft and Ullman's text on Automata Theory and Formal Languages.

Let V(x,y) be the set of non-terminal symbols that can generate the substring of *s* that starts at position x and is y symbols long. For example, V(2,3) is the set of all non-terminals that can generate the string 001. This string starts at the second symbol of *s* and continues for three symbols.

To determine whether s can be generated by the grammar above, we need to see whether S is contained in V(1,5).

We need to figure out a recursive relationship for V.

V(x,y) = {A | A → BC is a production, B is in V(x,k), and C is in V(x+k, y-k)}, for some k between 1 and y-1 inclusive.

For example, V(1,5) depends on four pairs of other V values: V(1,1) and V(2,4); V(1,2) and V(3,5); V(1,3) and V(4,2); V(1,4) and V(5,1). In general V(x,y) depends on y-1 different pairs of V(r,k) values, where the second parameter k ranges from 1 to y-1. If we compute the recursive calls, we end up computing many of the same subproblems over and over again.

This suggests that we compute the V(x,y) values in order from y = 1 to n. The base case is when y =1, and V(x,1) = {A | A → c is a production in the grammar, and c is the xth symbol in the given string s.}

The complete code is shown below:

```
For x = 1 to n {
V(x,1) = {A | A → c is a production in the grammar, and c is the xth symbol in the given string s.}
}

for y = 2 to n {
        for x = 1 to n-y+1 {
        V(x,y) = {};
                For k = 1 to y-1 {
V(x,y) = V(x,y) U {A | A → BC is a production, B is in V(x,k), and C is in V(x+k, y-k)},
                }
        }
}
```

It would be worthwhile to store information that allows us to recover the actual sequence of productions that parse the strings when it is actually generate by the language. I leave this problem for the Pset.

The table below shows the result of the algorithm after it has been executed for the string 10010. The computation proceeds top to bottom and left to right. Each new V(x,y) looks at pairs of other V values, where one of the pairs comes from the column above (x,y) moving down from the top, and the other pair comes the northeast diagonal moving up and to the left. I will draw the picture in class.

Here is the grammar and the string again for reference:

S → AB | BC A → BA | 0 B → CC | 1 C → AB | 0

s=10010

		X 1	2	3	4	5
Y	1	B	A,C	A,C	B	A,C
	2	S,A	B	S,C	S,A	
	3	0	B	B		
	4	0	S,A,C			
	5	S,A,C				

Since S, the start symbol, is in the set V(1,5), then we conclude that the grammar does indeed generate the string 10010.

Longest Common Subsequence

In recitation we will review one more dynamic programming idea that is related to string matching, and has applications in Biology. Given two sequences of symbols, we wish to find the longest common subsequence. This is useful when trying to match long subsections of DNA strings. It is also useful when one is trying to determine the age of a particular piece of wood by its *ring* patterns. There is a huge library of ring patterns for different geographical areas and different trees, which we try to match up with a sequence from our sample. The longer the common sequence, the more likely we have a correct match of time frame.

The Knapsack Problem – An NP-Complete Problem with a Pseudo-Polynomial Time Dynamic Programming Algorithm

The Knapsack problem is NP-Complete, but in special cases it can be solved in polynomial time. The algorithm to do this is a dynamic programming algorithm. There are also greedy algorithms that work in other special cases.

Imagine you are a robber in the twilight zone, and when you enter a house you see it has been prepared to help you choose your loot! Dozens of boxes are in the house, and each has an unlimited supply of objects all of the same kind. Each box is labeled with the size and value of its objects. You have come with a knapsack (hence the problem name) of a fixed size. All you have to do now is choose the combination of objects that will let you walk away with the most loot. That means a knapsack that is worth the most money – even if for example it is not completely full. For example if you objects of size two each worth two dollars, and objects of size 15 each worth 20 dollars, and your knapsack has a capacity of 16, then you are better off taking one object of size 15 leaving some empty space, rather than taking eight objects of size two and filling the knapsack completely.

We will describe an algorithm that solves the knapsack problem in O(nM), where n is the number of boxes of objects, and M is the size of the knapsack's capacity. This results in a polynomial time algorithm whenever M is a polynomial function of n. When the parameters to a problem include a main parameter (like n) and some extra numbers (like M), and the problem can be solved in polynomial time if the numbers are restricted in size, then the problem is said to be solved in *pseudo-polynomial* time. Note that the input size of a number M is considered to be the number of digits it contains. This is O(log M) because the number of digits in a number is proportional to the log of the number. Hence our O(nM) algorithm is not polynomial time in n and M, unless M is polynomial in n. This means O(nM) is not polynomial time unless the number of digits in M is O(log n).

The algorithm is reminiscent of the recursive relationship you saw in month 2 (discrete math) for calculating the number of ways to make change of m cents using pennies, nickels, dimes and quarters.

Let P(m,k) = the most loot we can fit in a knapsack with capacity m, using objects from boxes 1 through k. When k=0, we set P(m,k) = 0. This says that taking no objects gets us no loot.

When k>0, then P(m,k) can be computed by considering two possibilities. This should remind you of the proof that C(n,m) = C(n-1, m) + C(n-1, m-1). To choose m from n, either we include the mth object C(n-1,m-1) or we do not C(n-1, m).

For P(m,k). either we include items from box k or we don't. If we include items from box k, then P(m,k) = P(m-size(k), k). If we do not, then P(m,k) = P(m, k-1).

Hence P(m,k) = max {value(k) + P(m-size(k), k), P(m, k-1)}.

If k=0 or m-size(k) < 0 then P(m,k) = 0.

We can set up the calculation with appropriate loops and construct the P(m,k) values in a careful order. We can also remember which one of the two choices gave us the minimum for each P(m,k), allowing us afterwards to extract the actual objects that fill up the knapsack. I will leave these details out for now.

Greedy Strategy for *Liquid* Knapsack

In your Pset, I ask you to look at the knapsack problem again where you are allowed to choose fractions of objects. This variation of the problem can be solved with a greedy strategy.

Bandwidth Minimization – Another Pseudo Polynomial Time Dynamic Programming Algorithm

One final example of a dynamic programming problem comes from VLSI layout issues. The problem is called the bandwidth minimization problem and is represented as a graph problem. Like Knapsack, this problem is NP-complete in general but is solvable in pseudo-polynomial time. To determine the minimum bandwidth is NP-complete but to determine whether a graph has bandwidth can be done in O(n^(k+1)).

To solve the problem in general, we would have to use brute force and generate all n! linear layouts of the graph and then check each one in time O(e) to make sure that no edge was stretched too far. This is too slow of course, so let's focus on a special case when k=2. That is, we want to find out whether a layout is possible such that the longest stretch is at most two.

We begin by considering two nodes in order, node one and node two, and the set of edges connected to them. Some of these edges are *dangling* in the sense that they connect to nodes that must be laid out further on, that is they do not connect one node to the other. Note that to make the layout have bandwidth two, there had better not be two edges connected to node one. Moreover, if there is one edge connected to node one then we must lay that node out next. If there are no edges connected to node one then we have O(n) choices for the next node. Once we lay out the next node, we remember only the last two nodes in the layout, because the first one cannot possibly have any more edges dangling. Hence at each stage when we lay out a new node, we must consider O(n) choices in the worst case—giving an O(n!) brute force approach.

However, it is important, that altogether there are only O(n^2) *partial layouts.* A partial layout is a subset of two nodes with a subset of its adjacent edges marked as dangling. The edges not marked as dangling are connected to the left in an area that has already been laid out correctly. We can solve the bandwidth problem for k=2, by checking each partial layout at most once.

We start by placing all the partial layouts that have its dangling edges all off the right side, on a queue. Then while the queue is not empty, we remove a partial layout from the queue and try to extend it. If we can successfully extend it, then we place the new resulting partial layout on the queue, and continue. If a partial layout cannot be extended we simply continue with the next partial layout on the queue. If a partial layout with no dangling edges comes off the queue, then we answer yes to the question of bandwidth two. If we empty out the queue and find no partial layouts with no dangling edges, then we say no to the question of bandwidth two.

The complexity of this algorithm depends on the fact that we will never put a partial layout on the queue more than once, and there are at most O(n^2) partial layouts. For each partial layout there are at most O(n) possible extension possibilities. This makes the total complexity of the algorithm O(n^3). We must keep a Boolean array for each partial layouts that stores whether or not it has been put on the queue yet, to make sure we never put anything on twice.

I will discuss this more in detail in class and do a small example.

Greedy Strategy vs. Dynamic Programming

In recitation, we will discuss the problem of finding the minimum number of coins to make change for n cents. In the USA, the greedy strategy solves this correctly. We simply try the largest coin possible until we can't use it any more. There are however, other denominations of coins for which the greedy strategy would fail, and we will present such a set in recitation. We will discuss a dynamic programming approach that always solves the problem correctly. By the way, given a set of coin denominations, the question of whether the greedy strategy works is NP-complete!

Activity Selection – A Greedy Method

Huffman Codes and Compression– Another Greedy Algorithm

NP-Complete Problems and Reductions