Course Notes for CS310 – Dynamic Programming

Nurit Haspel (notes adapted from Prof. Betty O'Neil and Prof. Carl Offner)

Problem – Making Change

Let me give you a task – you should go to the cafeteria and buy a cup of coffee. Let's say the coffee costs 63 cents (yeah, right). You are given an unlimited number of coins of all the common types – 1, 5, 10, and 25 cents (forget the other types), and your only limitation is that you have to pay the exact change. Let's think – What is the combination of coins you'd use?

I bet most of you thought something like "two quarters, one dime, three cents", right? This is a correct answer, but it is by no means the only correct answer! You can also pay six dimes and three cents, or even 63 1-cent coins! Why, then, did you almost certainly think of the combination of "three quarters, one dime, four cents"? Moreover, why in this particular order, from the largest coin to the smallest one? The answer lies in our tendency for optimization and "greedy thinking".

Greedy Algorithms – Change Making

Logically, we select the combination that *minimizes* the number of coins. This is the optimization part – we want to count change with the fewest number of coins.

Where is the "greedy" part then? The answer lies in the order which we count the coins – from the large to the small. Clearly we want to first use as many large-value coins to minimize the total number. When we can't use the largest value we move on to the next largest etc. What about 63? Use as many 25s as fit, $63 = 2(25) + 13$, then as many 10s as fit in the remainder: $63 = 2(25) + 1(10) + 3$, no 5's fit, so we have $63 = 2(25) + 1(10) + 3(1)$, 6 coins.

This method is called "a greedy algorithm". A greedy person grabs everything they can as soon as possible. Similarly, a greedy algorithm makes locally optimized decisions that appear to be the best thing to do at each step, not necessarily thinking about the global solution.

Example: Change-making greedy algorithm for "change" amount, given many coins of each size:

- Loop until change == 0:
- Find largest-valued coin less than change, use it.
- change = change - coin-value;

Does the greedy method work? That is, will it always give us the minimum number of coins? The answer is yes for US coinage, but not in general. That is, it will always give us a way to count change, but for some combination of coins we can't guarantee optimality.

For example, let's say we had an additional 21 cent coin. Then we know that the optimal solution is $63 = 3(21)$, but the greedy method above says to use 2 25s, 1 10, and 3 1's, a total of 6 coins. Why didn't it work now? The coin values need to be spread out enough to make greedy work. In this case the quarter "overshadows" the 21-cent coin, so that it will almost always get selected by a greedy method, not giving the 21-cent a chance. But even some spread-out cases don't work.

Consider having pennies, dimes and quarters, but no nickels. Then if we want to count 30 cents, the greedy scheme will select a quarter and five pennies, ignoring the optimal solution of three dimes. Greedy algorithms are a very interesting class of algorithms. They are very often used as approximation methods when the exact solution is too hard to calculate. There are some cases when a greedy algorithm is guaranteed to give the optimal solution, like the example with the US coins above and some other examples we will see later on in the course. But for now we want an algorithm that will guarantee the minimum number of coins for any combination of coins, and the greedy method is clearly not the answer. So let's abandon greedy for now and try to find another way.

(Very bad) Recursive Solution

One way that guarantees an optimal solution is a rather brute-force recursive counting of all the possible combinations. Example: change for 63 cents with coins = {25, 10, 5, 1, 21} no order required in array.

```
makeChange(63)
minCoins = 63
loop over j from 1 to 63/2 = 31
   thisCoins = makeChange(j) + makeChange(63-j)
```

Notice that we only go halfway, because makeChange(63-j) covers the other half. Why is it very bad? Remember the double recursion examples from the beginning of the semester!
makeChange(63) calculates makeChange(62), makeChange(61), makeChange(60) etc.
makeChange(62) in turn calculates makeChange(61), makeChange(60), makeChange(59) etc.
We make lots and lots of redundant calls! If we want to express the runtime as a recursive formula, we get something like:

$$T(n) = T(n-1) + T(n-2) + T(n-3) + ... + T(n/2) + ... \tag{1}$$

This is far worse than T(n) = T(n-1) + T(n-2) the famous Fibonacci sequence discussed on pg. 242 (and hw1). Fibonacci is exponential, so this certainly is.

Here is a Better Idea: We know we only have 1,5,10,21 and 25 as coins. Therefore, the optimal solution for 63 cents must be the minimum of the following:

- 1 (A 1 cent) + optimal solution for 62.
- 1 (A 5 cent) + optimal solution for 58.
- 1 (A 10 cent) + optimal solution for 53.
- 1 (A 21 cent) + optimal solution for 42.
- 1 (A 25 cent) + optimal solution for 38.

This reduces the number of recursive calls drastically, since at every stage of the recursion we only have to make five calculation instead of up to 62. This is still a Naïve implementation that makes lots of redundant calls.

Dynamic Programming for Change Problem

The idea – instead of performing the same calculation over and over again, we realize that we can calculate them just once and save pre-calculated results to an array. The important insight is that the answer to a large change depends only on results of smaller calculations, so we can calculate the optimal answer for all the smaller change first, and save it to an array. Then go over the array and minimize on:

(2) $$change(n) = \min\{change(n - K) + 1\}$$

For all K types of coins
The runtime is $O(n * K)$, which is practically linear since K is usually small and independent of the input size. This is quite a decrease, from exponential to linear!

For the implementation we define two arrays: One for the minimum number of coins for each of 1,2,...n. The other is for the actual last coin that brought us there. This way we can backtrack and find the actual coin combination, not just the number of coins.

```
public static void makeChange( int [ ] coins, int differentCoins,
             int maxChange, int [ ] coinsUsed, int [ ] lastCoin )
   {
       coinsUsed[ 0 ] = 0; lastCoin[ 0 ] = 1;

       for( int cents = 1; cents <= maxChange; cents++ ){
           int minCoins = cents;
           int newCoin = 1;

           for( int j = 0; j < differentCoins; j++ ) {
               if( coins[ j ] > cents ) // Cannot use coin j
                   continue;
               if( coinsUsed[ cents - coins[ j ] ] + 1 < minCoins ) {
                   minCoins = coinsUsed[ cents - coins[ j ] ] + 1;
                   newCoin = coins[ j ];
               }
           }

           coinsUsed[ cents ] = minCoins;
           lastCoin[ cents ] = newCoin;
       }
   }
```

What is Dynamic Programming (DP)?

It is a common and very useful algorithm design technique for **optimization problems**: often minimizing or maximizing. Notice that we are "reversing" our original way of thinking about the problem. Instead of going top down – starting from the goal change and calculating smaller solutions on our way, we go bottom up – calculate small solutions independently of the big problem. This is a rather subtle but important distinction. It is also important to realize that even if the formulation above looks recursive, it is not!

Like divide and conquer, DP solves problems by combining solutions to subproblems. Unlike divide and conquer, subproblems are not independent and may share subsubproblems, However, solution to one subproblem may not affect the solutions to other subproblems of the same problem. (More on this later.) DP reduces computation by Solving subproblems in a bottom-up fashion. We do it by storing solution to a subproblem the first time it is solved, and looking up the solution when subproblem is encountered again. **Key:** determine the structure of optimal solutions. Find out how to build optimal solutions from solutions to subproblems. In the case of the change making problem, we figured out that every solution is the minimum of five possible smaller subproblems plus one coin.

Dynamic Programming – Two Conditions

DP cannot be applied to every optimization problem. Two conditions have to hold: One is the *optimal substructure property*: a solution contains within it the optimal solutions to subproblems – in this case, the minimum number of coins for smaller change. In other words, subproblems are just "smaller versions" of the main problem.

The second is the *overlapping subproblems*: There are only $O(n)$ distinct solutions, but they may appear multiple times on our way to solving the original problem. Therefore, we only have to compute each subproblem once, and save the result so we can use it again. This trick is called *memoization*, which refers to the process of saving (i.e., making a "memo") of a intermediate result so that it can be used again without recomputing it. Of course the words "memoize" and "memorize" are related etymologically, but they are different words, so don't mix them up.

Another Application – Sequence Alignment

This may seem like a pretty artificial problem, but it is part of a class of what are called *alignment problems*, which are extremely important in modern biology. Now that we can sequence the entire genomes of many organisms, we can use this information to deduce quite accurately how closely related different organisms are, and to infer the real "tree of life". Trees showing the evolutionary development of classes of organisms are called "phylogenetic trees", and they are computed by looking at the DNA sequences of different organisms and comparing them to see how close they are and where the differences are. A lot of this kind of comparison amounts to finding common subsequences, just as we are doing here. This is a tremendously exciting field.

See Figure 1.

Definition 1 *A* subsequence *of a sequence $A = \{a_1, a_2, \ldots, a_n\}$ is a sequence $B = \{b_1, b_2, \ldots, b_m\}$ (with $m \leq n$) such that*

- *Each b_i is an element of A.*
- *If b_i occurs before b_j in B (i.e., if $i < j$) then it also occurs before b_j in A.*

Note that in particular, we do *not* assume that the elements of B are consecutive elements of A. For instance, here is an example, where each sequence is an ordinary string of letters of the alphabet:

- "axdy" is a subsequence of "baxefdoym"

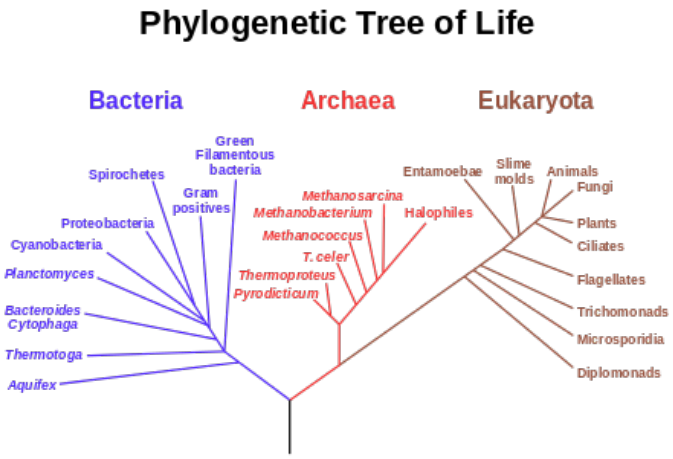
The "longest common subsequence" problem is simply this:

> Given two sequences $X = \{x_1, x_2, \ldots, x_m\}$ and $Y = \{y_1, y_2, \ldots, y_n\}$ (note that the sequences may have different lengths), find a subsequence common to both whose length is longest.

For example see Figure 2. We will use the abbreviation LCS to mean "longest common subsequence".

So how do we solve this problem? Suppose we try the obvious approach: list all the subsequences of X and check each to see if it is a subsequence of Y, and pick the longest one that is.

There are 2^m subsequences of X. To check to see if a subsequence of X is also a subsequence of Y will take time $O(n)$. (Is this obvious?) Picking the longest one is really just an $O(1)$ job, since we can keep track as we proceed of the longest subsequence that we have found so far that works. So the cost of this method is $O(n2^m)$.




```
A5ASC3.1    14 SIKLWPPSQTTRLLLVERMANNLST..PSIFTRK..YGSLSKEEARENAKQIEEVACSTANQ.....HYEKEPDGDGGSAVQLYAKECSKLILEVLK 101
B4F917.1    13 SIKLWPPSESTRIMLVDRMTNNLST..ESIFSRK..YRLLGKQEAHENAKTIEELCFALADE.....HFREEPDGDGSSAVQLYAKETSKMMLEVLK 100
A9S1V2.1    23 VFKLWPPSQGTREAVRQKMALKLSS..ACFESQS..FARIELADAQEHARAIEEVAFGAAQE......ADSGGDKTGSAVVMVYAKHASKLMLETLR 109
B9GSN7.1    13 SVKLWPPGQSTRLMLVERMTKNFIT..PSFISRK..YGLLSKEEAEEDAKKIEEVAFAAANQ.....HYEKQPDGDGSSAVQIYAKESSRLMLEVLK 100
Q8H056.1    30 SFSIWPPTQRTRDAVVRRLVDTLGG..DTILCKR..YGAVPAADAEPAARGIEAEAFDAAAA..SGEAAATASVEEGIKALQLYSKEVSRRLLDFVK 120
Q0D4Z3.2    44 SLSIWPPSQRTRDAVVRRLVQTLVA..PSILSQR..YGAVPEAEAGRAAAAVEAEAYAAVTES.SSAAAAPASVEDGIEVLQAYSKEVSRRLLELAK 135
B9MVW8.1    56 SFSIWPPTQRTRDAIISRLIETLST..TSVLSKR..YGTIPKEEASEASRRIEEEAFSGAST.......VASSEKDGLEVLQLYSKEISKRMLETVK 141
Q0IYC5.1    29 SFAVWPPTRRTRDAVVRRLVAVLSGDTTTALRKRYRYGAVPAADAERAARAVEAQAFDAASA....SSSSSSSVEDGIETLQLYSREVSNRLLAFVR 121
A9NW46.1    13 SIKLWPPSESTRLMLVERMTDNLSS..VSFFSRK..YGLLSKEEAAENAKRIEETAFLAAND.....HEAKEPNLDDSSVVQFYAREASKLMLEALK 100
Q9C500.1    57 SLRIWPPTQKTRDAVLNRLIETLST..ESILSKR..YGTLKSDDATTVAKLIEEEAYGVASN.......AVSSDDDGIKILELYSKEISKRMLESVK 142
Q2HRI7.1    25 NYSIWPPKQRTRDAVKNRLIETLST..PSVLTKR..YGTMSADEASAAAIQIEDEAFSVANA.......SSSTSNDNVTILEVYSKEISKRMIETVK 110
Q9M7N3.1    28 SFKIWPPTQRTREAVVRRLVETLTS..QSVLSKR..YGVIPEEDATSAARIIEEEAFSVASV.ASAASTGGRPEDEWIEVLHIYSQEIXQRVVESAK 119
Q9M7N6.1    25 SFSIWPPTQRTRDAVINRLIESLST..PSILSKR..YGTLPQDEASETARLIEEEAFAAAGS.......TASDADDGIEILQVYSKEISKRMIDTVK 110
Q9LE82.1    14 SVKMWPPSKSTRLMLVERMTKNITT..PSIFSRK..YGLLSVEEAEQDAKRIEDLAFATANK.....HFQNEPDGDGTSAVHVYAKESSKLMLDVIK 101
Q9M651.2    13 SIKLWPPSLPTRKALIERITNNFSS..KTIFTEK..YGSLTKDQATENAKRIEDIAFSTANQ.....QFEREPDGDGGSAVQLYAKECSKLILEVLK 100
B9R748.1    48 SLSIWPPTQRTRDAVITRLIETLSS..PSVLSKR..YGTISHDEAESAARRIEDEAFGVANT.......ATSAEDDGLEILQLYSKEISRRMLDTVK 133
```

Figure 1: Top: The "Tree of life". Bottom: An example of multiple sequence alignment.

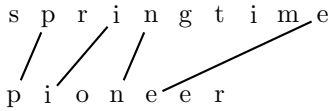


Figure 2: An example of two sequences and their longest common subsequence (LCS)

That's pretty awful. It's so bad, in fact, that it's completely useless. The strings that we are concerned with in biology have hundreds or thousands of elements *at least*. So we really need a better algorithm.
Here is one. It depends on a couple of really important properties of the problem:

Optimal substructure
Again, let us say we have two strings, with possibly different lengths:

$$X = \{[x_1, x_2, \ldots, x_m\}$$
$$Y = \{y_1, y_2, \ldots, y_n\}$$

A *prefix* of a string is an initial segment. So we define for each i less than or equal to the length of the string the prefix of length i:

$$X_i = \{x_1, x_2, \ldots, x_i\}$$
$$Y_i = \{y_1, y_2, \ldots, y_i\}$$

Now the point of what we are going to prove is that a solution of our problem reflects itself in solutions of prefixes of X and Y.

Theorem 1 *Let $Z = \{z_1, z_2, \ldots, z_k\}$ be any LCS of X and Y.*

1. *If $x_m = y_n$, then $z_k = x_m = y_n$, and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.*
2. *If $x_m \neq y_n$, then $z_k \neq x_m \implies Z$ is an LCS of X_{m-1} and Y.*
3. *If $x_m \neq y_n$, then $z_k \neq y_n \implies Z$ is an LCS of X and Y_{n-1}.*

Proof 1

1. *By assumption $x_m = y_n$. If z_k does not equal this value, then Z must be a common subsequence of X_{m-1} and Y_{n-1}, and so the sequence $Z' = \{z_1, z_2, \ldots, z_k, x_m\}$ would be a common subsequence of X and Y. But this is a longer common subsequence than Z, and this is a contradiction.*

2. *If $z_k \neq x_m$, then Z must be a subsequence of X_{m-1}, and so it is a common subsequence of X_{m-1} and Y. If there were a longer one, then it would also be a common subsequence of X and Y, which would be a contradiction.*

3. *This is really the same as 2.*

Note that conclusions 2 and 3 of the Theorem could be summarized as follows:

Corollary 1 *If $x_m \neq y_n$, then either*

- *Z is an LCS of X_{m-1} and Y, or*
- *Z is an LCS of X and Y_{n-1}.*

Thus, the LCS problem has the *optimal substructure property* –in this case, to subproblems constructed from prefixes of the original data. This is one of the two keys to the success of a dynamic programming solution.

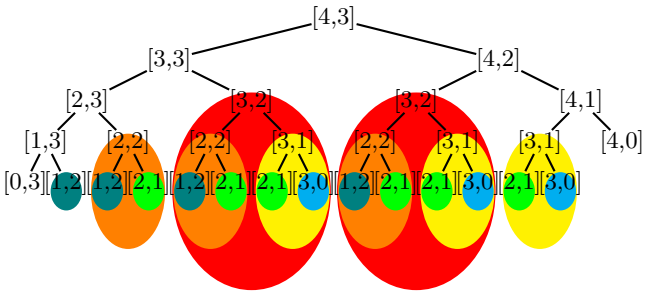


Figure 3: A recursive tree for the LCS problem.

Recursive solution

Let $c[i, j]$ be the length of the LCS of X_i and Y_j. Based on Theorem 1, we can write the following recurrence:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{c[i-1, j], c[i, j-1]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

This is nice—the optimal substructure property allows us to write down an elegant recursive algorithm. However, the cost is still far too great—we can see that there are $\Omega(2^{\min\{m,n\}})$ nodes in the tree, which is still a killer. But at least we have an algorithm.

Overlapping subproblems

What saves us is that there really aren't that many *distinct* nodes in the tree. In fact, there are only $O(mn)$ distinct nodes. Its just that individual nodes tend to occur lots of times. (This is similar to the tree that you get for a naive recursive algorithm for computing the Fibonacci numbers.)

So the point is that we only have to compute each subproblem once, and save the result so we can use it again. This is again *memoization*. See Algorithm 1 for how it's done.

Figure 4 gives an example showing how these tables are constructed.

The length of an LCS of the two sequences is just $c[m, n]$; that is, it is the number found in the lower right-hand entry of the array. To construct the actual LCS, we can walk backward, following the arrows, from that entry. Each time we encounter a $\nwarrow$, we know that we are at an element of the LCS. The pseudo-code that implements this is shown in Algorithm 2.

Algorithm 1 LCSLength(X,Y,m,n)

```
1: for i ← 1...m do
2:    c[i,0] ← 0
3: end for
4: for j ← 0...n do
5:    c[0,j] ← 0
6: end for
7: for i ← 1...m do
8:    for j ← 1...n do
9:       if x_i == y_j then
10:         c[i,j] ← c[i−1,j−1] + 1; b[i,j] ← "↖"
11:      else
12:         if c[i−1,j] ≥ c[i,j−1] then
13:            c[i,j] ← c[i−1,j]; b[i,j] ← "↑"
14:         else
15:            c[i,j] ← c[i,j−1]; b[i,j] ← "←"
16:         end if
17:      end if
18:   end for
19: end for
20: return c and b
```

Algorithm 2 PrintLCS(b, X, i, j)

```
1: if i = 0 or j = 0 then
2:    return
3: end if
4: if b[i,j] == "↖" then
5:    PrintLCS(b, X, i−1, j−1)
6:    PRINT x_i
7: else
8:    if b[i,j] == "↑" then
9:       PrintLCS(b, X, i−1, j)
10:   else
11:      PrintLCS(b, X, i, j−1)
12:   end if
13: end if
```

And it is clear that the cost of doing all this is $O(mn)$.

Binomial Coefficients

Another famous example is the sequence of binomial coefficients. These are the coefficients of the powers of the series generated by $(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$. For example –

$$(x+y)^4 = x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4 \tag{3}$$

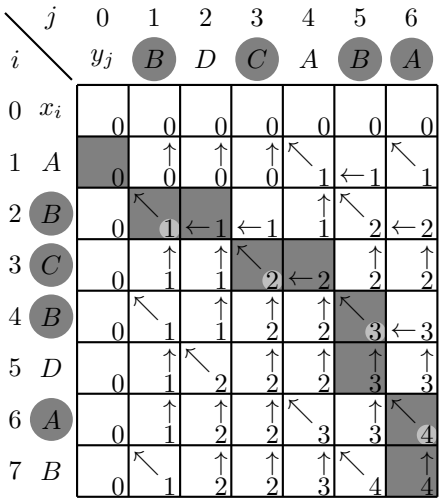


Figure 4: The c and b tables computed by Algorithm 1 on the sequences $X = \{A, B, C, B, D, A, B\}$ and $Y = \{B, D, C, A, B, A\}$.

They can also be generated by Pascal's triangle as seen in Figure 5. To draw the triangle, start a new row with 1's on the edges. Otherwise, every number is the sum of the two above it.

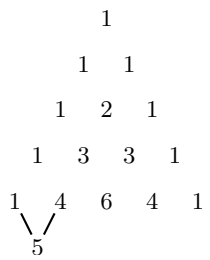


Figure 5: Pascal's triangle

The row number is N, and the entries are k=0, k=1, ..., k=N across a row, so for example let us denote C(4,0) = 1, C(4,1) = 4, C(4,2) = 6, C(4,3) = 4, C(4,4) = 1. (incidentally, the rows are also the powers of 11...).
Another way to look at the binomial coefficients is remembering that C(N, k) = number of ways to choose a set of k objects from N without order or repetitions. It is often denoted $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ (read N choose K). For ex. C(4, 2) = 6 The 2-sets of 4 numbers are the 6 sets: $\{1, 2\}$, $\{1, 3\}$, $\{1, 4\}$, $\{2, 3\}$, $\{2, 4\}$, $\{3, 4\}$

n Choose k – Rationale: Many people struggle with the actual "meaning" or intuition behind $\binom{n}{k}$. Let me give you some pointers that you may find helpful in future courses (Discrete math, prob. and stat. etc.):
Base cases: C(N, 0) = 1, C(N, N) = 1. There is only one way to choose zero numbers out of N (the empty set), and only one way to choose N numbers out of N (the entire set).

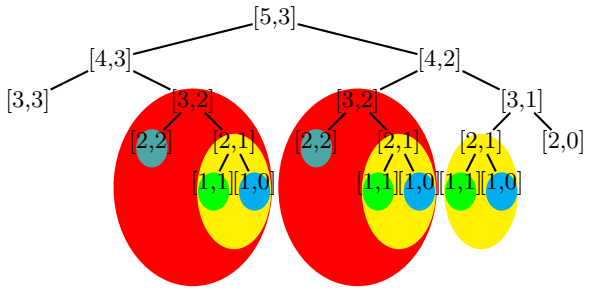


Figure 6: An example of double recursion in binomial coefficients.

Any other case: To choose k objects from N, set one object x aside and find all the ways of choosing k objects from the remaining N-1. We have two cases:

1. These are all the sets we want that don't include x, C(N-1, k).

2. The sets that do include x also need k-1 other objects from the other N-1, C(N-1, k-1).

Another non-recursive, explicit way to think about it: If we need to choose k objects out of N: There are N possibilities to select the first object of the k. There are N-1 possibilities to select the second out of the remaining N-1 objects. There are N-2 possible selections for the third object etc. There are N-k+1 possible selections for the k^{th} object. These selections are independent of one another, so we multiply them to obtain the total number of ways to choose the k objects out of N:

$$N * (N-1) * ... * (N-k+1) = \frac{N!}{(N-k)!} \tag{4}$$

Notice, however, that we assume a particular order to the k objects as we selected them. Since the order does not matter, we could choose the k objects in any order out of the possible k! ways to order (permute) them. So we have a k! possible ways to get to the equation above, all result in the same selection. Therefore we should divide the equation above by k! to get $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.

Binomial Coefficient – Recursion

Notice that since every number in the triangle can be expressed as the sum of the two above it, this is basically a recursive definition: C(N, k) = C(N-1, k) + C(N-1, k-1)
We can write a naïve recursive function:

```
combo(N, k):
if (k == 0) return 1
if (k == N) return 1
return combo(N-1, k) + combo(N-1, k-1)
```

Note the double recursion, without halving the "N" value, so we have a dangerous recursion (remember double recursion from the beginning of the semester! It's what happens when we repeat calculations). See Figure 6. So, we get an exponential runtime for T(N): T(N, k) = T(N-1, k) + T(N-1, k-1) – 2 terms in N-1
= T(N-2, k) + ... 4 terms in N-2
= ... some of these hit base cases and stop

Efficient Calculation of Binomial Coefficients

Notice that as before, if we go bottom-up – save and reuse values, it's much faster. In other words, use Pascal's triangle to generate all the coefficients.

One way to do it: set up a table and use it for each N in turn.

```
C[1][0] = 1
C[1][1] = 1
for n up to N
   for k up to n
      C[n][ k] = C[n-1][ k] + C[n-1][k-1]
```

It takes $O(1)$ to fill each spot in the upper half of an NxN array, so overall $O(N^2)$

Map Approach to Binomial Coefficients

Another approach: set up Map from (N, k) to value. if N and k are both ints, then a long key = N + (long)k>>32 will fit both This is a case of classic dynamic programming, saving partial results along the way, even though the implementation, again, looks recursive. It actually is, but most recursive calls end up looking up an existing value in a map.

```
combo(N, k):
val = M.get(key(N,k))
if (val != null) return val
if (k == 0) val = 1
if (k == N) val = 1
else val = combo(N-1, k) + combo(N-1, k-1)
M.put(key(N, k), val)
return val
```

once this recursion reaches a cell, fills it in, so work bounded by number of cells below (N, k), which is $< N^2$.

Maximum Contiguous Subsequence Sum

Given a sequence of integers $(A_1, A_2, ..., A_N)$, possibly negative numbers, otherwise the problem is not interesting (why?). The task is to identify the subsequence $(A_i, ..., A_j)$ that corresponds to the maximum value of $\sum_{i}^{j} A_k$

The naïve approach is cubic (examine all $O(N^2)$ sequences and sum each one).

Divide-and-Conquer Algorithm

The divide and conquer algorithm is shown in Figure 7. The sample input is {4, -3, 5, -2, -1, 2, 6, -2} Notice that there are three possible cases:

1. The maximum sum is contained entirely in the first half
2. The maximum sum is contained entirely in the second half
3. The maximum sum begins in the first half and ends in the second half

Notice that case 3 can be solved in linear time by running a loop forward in the right half and backwards in the left half, computing the running sum as we loop. Therefore, we can apply case 3's strategy to solve case 1 and 2

Summary:

For case 3: sum = sum 1^{st} + sum 2^{nd}

$\leftarrow \quad \rightarrow$

First Half				Second Half				
4	–3	5	–2	–1	2	6	–2	Values
4*	0	3	–2	–1	1	7*	5	Running sums

Running sum from the center (*denotes maximum for each half).

figure 7.19

Dividing the maximum contiguous subsequence problem into halves

Figure 7: Divide and conquer example for the contiguous subset-sum

- Recursively compute the max subsequence sum in the first half
- Recursively compute the max subsequence sum in the second half
- Compute, via two consecutive loops, the max subsequence sum that begins in the first half but ends in the second half
- Choose the largest of the 3 sums

The runtime is $O(n \log n)$, since it's basically the same formula as MergeSort.

```
/**
 * Recursive maximum contiguous subsequence sum algorithm.
 * Finds maximum sum in subarray spanning a[left..right].
 * Does not attempt to maintain actual best sequence.
 */
private static int maxSumRec( int [ ] a, int left, int right )
{
    int maxLeftBorderSum = 0, maxRightBorderSum = 0;
    int leftBorderSum = 0, rightBorderSum = 0;
    int center = ( left + right ) / 2;

    if( left == right ) // Base case
        return a[ left ] > 0 ? a[ left ] : 0;

    int maxLeftSum = maxSumRec( a, left, center );
    int maxRightSum = maxSumRec( a, center + 1, right );

    for( int i = center; i >= left; i-- )
    {
        leftBorderSum += a[ i ];
        if( leftBorderSum > maxLeftBorderSum )
            maxLeftBorderSum = leftBorderSum;
    }

    for( int i = center + 1; i <= right; i++ )
    {
        rightBorderSum += a[ i ];
```

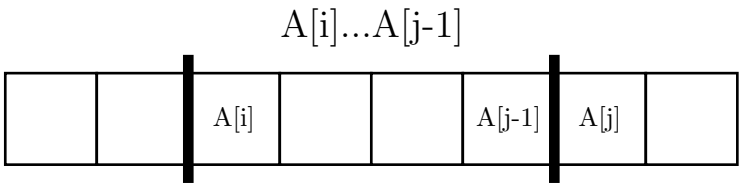


Figure 8: Dynamic programming illustration of the maximum contiguous subset sum

```
        if( rightBorderSum > maxRightBorderSum )
            maxRightBorderSum = rightBorderSum;
    }

    return max3( maxLeftSum, maxRightSum,
                 maxLeftBorderSum + maxRightBorderSum );
}

/**
 * Return maximum of three integers.
 */
private static int max3( int a, int b, int c )
{
    return a > b ? a > c ? a : c : b > c ? b : c;
}
```

Dynamic Programming Solution

Notice that we can use DP for this problem too. Let's look at index j. The maximum contiguous subsequence ending at index j (denoted MaxSum(j)) either extends a previous maximum subsequence (ending at j-1) or starts a new sum. The former happens if MaxSum(j-1) is positive. The latter happens if MaxSum(j-1) is non-positive. See Figure 8. Notice the optimal substructure (maxsum(j) builds on maxsum(j-1) or starts its own). There are also overlapping subproblems, which are the sub-subsequences that are parts of bigger subsequences.
Therefore a dynamic programming solution for Max(j) is:

$$Max(j) = \max\{Max(j-1) + a[j], a[j]\} \tag{5}$$

(constant computing time for each j, considering that Max(j-1) was already computed).
The overall solution to the problem is $\max_j Max(j)$. So the run time is:

- T(1) = O(1) – the maximum sum for a[0] is $\max\{a[0], 0\}$.
- T(N) = T(N-1) + O(1) – maximizing over two O(1) expressions.

We already know that this is O(N).