COS 397 – (week 02) DYNAMIC PROGRAMMING

Computations with *memoization*.

Memoization is a technique used to speed up computer programs by storing the results of functions for later reuse, rather than recomputing them (the word *memoization* is often confused with *memorization*, which, although a good description of the process, is not limited to this specific meaning).

Fibonacci sequence:

```
typedef unsigned long long int LONG;
LONG a[94];
LONG f(int i)
{if(a[i]>0) return a[i];
 a[i]=f(i-2)+f(i-1);
 return a[i];
}
int main()
{a[1]=1; a[2]=1;
 for(int i=1;i<94;i++) cout << f(i) << " ";
}
```

Why do we need `typedef unsigned long long int LONG;` ?
What is the purpose of the magic number 94 here?

Typical case study: Finding the longest increasing subsequence.

Example: Given a sequence of integers 9,5,2,8,7,3,1,6,4, one possible solution is 2,3,4.

Algorithm: Let us denote the given sequence by a[i], i = 1, 2, ..., *n*. Indicating by s[i] the *length* of the longest increasing subsequence that ends at a[i], our goal is to compute successively for i = 1, 2, ..., *n*, all the values of s[i].

First, it is clear that s[1] = 1. Next, if we have already computed all values up to the i-th position: s[1], s[2], ..., s[i], then s[i+1] = 1 + max {s[j] : $0 \le j \le i$, a[j] < a[i+1]}
(in case this set is empty, we assume s[i+1] = 1).

The length of the longest subsequence can be found by computing the maximal value of s[j] over all j = 1, 2,.., *n*. How we can find the longest increasing subsequence, see Exercise 2.

Following the algorithm, we make out the table below for the above example:

i =	1	2	3	4	5	6	7	8	9
a[i] =	**9**	**5**	**2**	**8**	**7**	**3**	**1**	**6**	**4**
s[i] =	1	1	1	2	2	2	1	3	3
j_max =	-	-	-	2	2	3	-	6	6
backward			*			*			*

Exercises:

1. Implement a program to find the length of the longest increasing subsequence for a given integer sequence as input data.
2. Extent the previous program to find also the longest increasing subsequence itself (one instance, if there is more then one solution). Hint: As is seen in the table, start at the position `i` for which the value of `s[i]` is the largest. The backward step is: move to the new position `i_new` for which `i_new = j_max[i]`, and assign `i=i_max` to do the next step.

Longest Common Subsequence

Given two sequences of items, find the longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, in the string *abcdefg*, "abc", "abg", "bdf", "aeg" are all subsequences.
A naive exponential algorithm is to notice that a string of length n has $O(2^n)$ different subsequences, so we can take the shorter string of both given, and test each of its subsequences for presence in the other string.
Recursive solution
We can try to solve the problem in terms of smaller subproblems. We are given two strings x and y, of length n and m respectively. We solve the problem of finding the longest common subsequence of $x = x_{1...n}$ and $y = y_{1...m}$ by taking the best of the three possible cases:

1. The longest common subsequence of the strings $x_{1...n-1}$ and $y_{1...m}$
2. The longest common subsequence of the strings $x_{1...n}$ and $y_{1...m-1}$
3. If x_n is the same as y_m, the longest common subsequence of the strings $x_{1...n-1}$ and $y_{1...m-1}$, followed by the common last character.

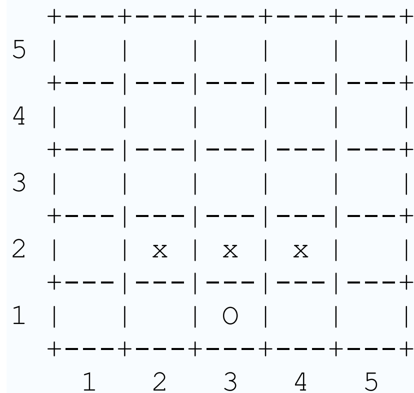
```
string lcs(string x, string y)
{
  int n=x.size();
  int m=y.size();
  string best="";
  if ((n==0) || (m==0)) return best;
  string s1 = lcs(x.substr(0,n-1),y.substr(0,m));
  string s2 = lcs(x.substr(0,n),y.substr(0,m-1));
  string ss = lcs(x.substr(0,n-1),y.substr(0,m-1));
  if(s1.size()>s2.size()) best=s1; else best=s2;
  if((x[n-1]==y[m-1]) && (best.size()<ss.size()+1))best = ss+x[n-1];
  return best;
}
int main() {cout << lcs("abcde","axcye") << endl;}
```

Obviously, this is still not very efficient. But because the subproblems are repeated, we can use memoization. An efficient way, which avoids the overhead of function calls, is to order the computation in such a way that whenever the results of subproblems are needed, they have already been computed, and can simply be looked up in a table. This is the idea of the Dynamic Programming. We find lcs($x_{1..i}$,$y_{1..j}$) for every i and j, starting from smaller ones, storing the results in an array at index (i,j) as we go along.
Dynamic programming solution closely parallels the recursive solution above, while entirely eliminating recursive calls. This "small" change makes the difference between exponential time and polynomial time.

Checkerboard
Consider a checkerboard with $n \times n$ squares and a cost-function $c(i, j)$ which returns a cost associated with square i,j (i being the row, j being the column). For instance (on a 5×5 checkerboard),

```
  +---+---+---+---+---+
5 | 6 | 7 | 4 | 7 | 8 |
  +---|---|---|---|---+
4 | 7 | 6 | 1 | 1 | 4 |
  +---|---|---|---|---+
3 | 3 | 5 | 7 | 8 | 2 |
  +---|---|---|---|---+
2 | 2 | 6 | 7 | 0 | 2 |
  +---|---|---|---|---+
1 | 7 | 3 | 5 | 6 | 1 |
  +---+---+---+---+---+
    1   2   3   4   5
```

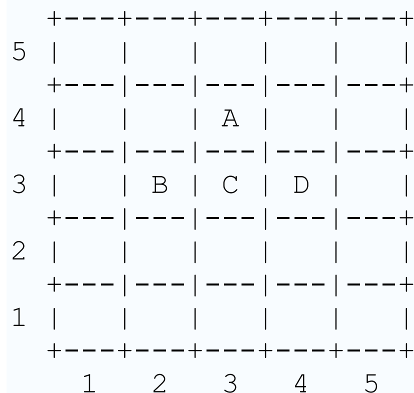
Thus $c(1, 3) = 5$
Let us say you had a checker that could start at any square on the first rank (i.e., row) and you wanted to know the shortest path (sum of the costs of the visited squares are at a minimum) to get to the last rank, assuming the checker could **move only diagonally left forward, diagonally right forward, or straight forward.** That is, a checker on (1,3) can move to (2,2), (2,3) or (2,4).



This problem exhibits **optimal substructure**. That is, the solution to the entire problem relies on solutions to subproblems. Let us define a function $q(i, j)$ as

$$q(i, j) = \text{the minimum cost to reach square } (i, j)$$

If we can find the values of this function for all the squares at rank n, we pick the minimum and follow that path backwards to get the shortest path.
It is easy to see that $q(i, j)$ is equal to the minimum cost to get to any of the three squares below it (since those are the only squares that can reach it) plus $c(i, j)$. For instance:



$$q(A) = \min(q(B),\ q(C),\ q(D))\ +\ c(A)$$

Now, let us define $q(i, j)$ in little more general terms:

$$q(i,j) = \begin{cases} \infty & j < 1 \text{ or } j > n \\ c(i,j) & i = 1 \\ \min(q(i-1,j-1), q(i-1,j), q(i-1,j+1)) + c(i,j) & \text{otherwise.} \end{cases}$$

This equation is pretty straightforward. The first line is simply there to make the recursive property simpler (when dealing with the edges, so we need only one recursion). The second line says what happens in the first rank, so we have something to start with. The third line, the recursion, is the important part. It is basically the same as the A,B,C,D example. From this definition we can make a

straightforward recursive code for $q(i, j)$. In the following pseudocode, n is the size of the board, `c(i, j)` is the cost-function, and `min()` returns the minimum of a number of values:

```
int minCost(int i, int j)
{if ((j < 1)||(j > n))
  return infinity;
 else if (i == 1)
  return c(i, j);
 else
  return
   min(minCost(i-1,j-1),minCost(i-1,j),minCost(i-1,j+1))+c(i,j);
}
```

It should be noted that this function just computes the path-cost, not the actual path. We will get to the path soon. This, like the Fibonacci-numbers example, is horribly slow since it spends mountains of time recomputing the same shortest paths over and over. However, we can compute it much faster in a bottom up-fashion if we use a two-dimensional array `q[i, j]` instead of a function. Why do we do that? Simply because when using a function we recompute the same path over and over, and we can choose what values to compute first.

We also need to know what the actual path is. The path problem we can solve using another array `p[i,j]`, a *predecessor array*. This array basically says where paths come from. Consider the following code:

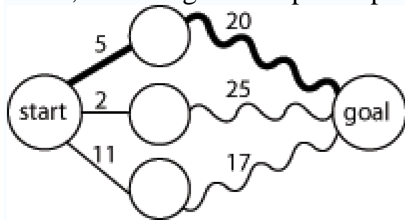
```
function computeShortestPathArrays()
     for x from 1 to n
         q[1, x] := c(1, x)
     for y from 1 to n
         q[y, 0]     := infinity
         q[y, n + 1] := infinity
     for y from 2 to n
         for x from 1 to n
             m := min(q[y-1, x-1], q[y-1, x], q[y-1, x+1])
             q[y, x] := m + c(y, x)
             if m = q[y-1, x-1]
                 p[y, x] := -1
             else if m = q[y-1, x]
                 p[y, x] :=  0
             else
                 p[y, x] :=  1
```

Now the rest is a simple matter of finding the minimum and printing it.

```
function computeShortestPath()
     computeShortestPathArrays()
     minIndex := 1
     min := q[n, 1]
     for i from 2 to n
         if q[n, i] < min
             minIndex := i
             min := q[n, i]
     printPath(n, minIndex)
 function printPath(y, x)
     print(x)
     print("<-")
     if y = 2
         print(x + p[y, x])
     else
         printPath(y-1, x + p[y, x])
```

Dynamic programming

In computer science, **dynamic programming** is a method of solving problems exhibiting the properties of overlapping subproblems and optimal substructure that takes much less time than naive methods.
The term was originally used in the 1940s by Richard Bellman to describe the process of solving problems where one needs to find the best decisions one after another. By 1953, he had refined this to the modern meaning. The field was founded as a systems analysis and engineering topic which is recognized by the IEEE.
The word "programming" in "dynamic programming" has no particular connection to computer programming at all. A program is, instead, the plan for action that is produced. For instance, a finalized schedule of events at an exhibition is sometimes called a program. Programming, in this sense, is finding an acceptable plan of action.

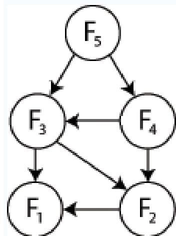


On the Figure: Finding the shortest path in a graph using optimal substructure; a wavy line indicates a shortest path between the two vertices it connects; the bold line is the overall shortest path from start to goal.

Optimal substructure means that optimal solutions of subproblems can be used to find the optimal solutions of the overall problem. For example, the shortest path to a goal from a vertex in an acyclic graph can be found by first computing the shortest path to the goal from all adjacent vertices, and then using this to pick the best overall path, as shown in Figure 1. In general, we can solve a problem with optimal substructure using a three-step process:

1. Break the problem into smaller subproblems.
2. Solve these problems optimally using this three-step process recursively.
3. Use these optimal solutions to construct an optimal solution for the original problem.

The subproblems are, themselves, solved by dividing them into sub-subproblems, and so on, until we reach some simple case that is easy to solve.



On the Figure. The subproblem graph for the Fibonacci sequence. That it is not a tree but a DAG indicates overlapping subproblems.

To say that a problem has *overlapping subproblems* is to say that the same subproblems are used to solve many different larger problems. For example, in the Fibonacci sequence, $F_3 = F_1 + F_2$ and $F_4 = F_2 + F_3$ — computing each number involves computing F_2. Because both F_3 and F_4 are needed to compute F_5, a naïve approach to computing F_5 may end up computing F_2 twice or more. This applies whenever overlapping subproblems are present: a naïve approach may waste time recomputing optimal solutions to subproblems it has already solved.
In order to avoid this, we instead save the solutions to problems we have already solved. Then, if we need to solve the same problem later, we can retrieve and reuse our already-computed solution. This approach is called *memoization* (not *memorization*, although this term also fits). If we are sure we

won't need a particular solution anymore, we can throw it away to save space. In some cases, we can even compute the solutions to subproblems we know that we'll need in advance.
In summary, dynamic programming makes use of:

- Overlapping subproblems
- Optimal substructure
- Memoization

Dynamic programming usually takes one of two approaches:

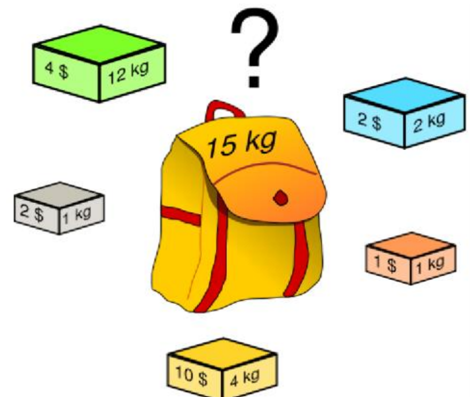
- **Top-down approach**: The problem is broken into subproblems, and these subproblems are solved and the solutions remembered, in case they need to be solved again. This is recursion and memoization combined together.
- **Bottom-up approach**: All subproblems that might be needed are solved in advance and then used to build up solutions to larger problems. This approach is slightly better in stack space and number of function calls, but it is sometimes not intuitive to figure out all the subproblems needed for solving the given problem.

Some programming languages with special extensions:

http://www.apl.jhu.edu/~paulmac/c++-memoization.html

can automatically memoize the result of a function call with a particular set of arguments, in order to speed up call-by-name evaluation (this mechanism is referred to as *call-by-need*). Some languages (e.g., Maple) have automatic memoization builtin. This is only possible for a function which has no side-effects, which is always true in pure functional languages but seldom true in imperative languages.

Knapsack problem



Example of a one-dimensional (constraint) knapsack problem: which boxes should be chosen to maximize the amount of money while still keeping the overall weight under 15 kg? A multi dimensional problem could consider the density or dimensions of the boxes, the latter a typical packing problem
The **knapsack problem** is a problem in combinatorial optimization. It derives its name from the maximization problem of choosing possible essentials that can fit into one bag (of maximum weight) to be carried on a trip. A similar problem very often appears in business, combinatorics, complexity theory, cryptography and applied mathematics. Given a set of items, each with a cost and a value, then determine the number of each item to include in a collection so that the total cost is less than some given cost and the total value is as large as possible.
The decision problem form of the knapsack problem is the question "can a value of at least V be achieved without exceeding the cost C?"
In the following, we have n kinds of items, x_1 through x_n. Each item x_j has a value p_j and a weight w_j. The maximum weight that we can carry in the bag is C.
The **0-1 knapsack problem** restricts the number of each kind of item to zero or one.
Mathematically the 0-1-knapsack problem can be formulated as:

$$\text{maximize} \sum_{j=1}^{n} p_j x_j.$$

$$\text{subject to} \sum_{j=1}^{n} w_j x_j \leq c, \qquad x_j = 0 \text{ or } 1, \quad j = 1, \ldots, n.$$

The **bounded knapsack problem** restricts the number of each item to a specific value.
Mathematically the bounded knapsack problem can be formulated as:

$$\text{maximize} \sum_{j=1}^{n} p_j x_j.$$

$$\text{subject to} \sum_{j=1}^{n} w_j x_j \leq c, \qquad 0 \leq x_j \leq b_j, \quad j = 1, \ldots, n.$$

The **unbounded knapsack problem** places no bounds on the number of each item.
Of particular interest is the special case of the problem with these properties:

- It is a decision problem
- It is a 0/1 problem
- For each item, the cost equals the value: $C = V$

Notice that in this special case, the problem is equivalent to this: given a set of integers, does any subset of it add up to exactly *C*? Or, if negative costs are allowed and *C* is chosen to be zero, the problem is: given a set of integers, does any subset add up to exactly 0? This special case is called the subset sum problem. For some reason, it is traditional in cryptography to say "knapsack problem" when it is actually the "subset sum problem" that is meant.
The knapsack problem is often solved using dynamic programming, though no polynomial-time algorithm is known for the general problem. Both the general knapsack problem and the subset sum problem are NP-hard, and this has led to attempts to use subset sum as the basis for public key cryptography systems, such as Merkle-Hellman.

Dynamic programming solution

The knapsack problem can be solved in pseudo-polynomial time using dynamic programming. The following depicts a dynamic programming solution for the *unbounded knapsack problem*.
In computational complexity theory, a numeric algorithm runs in **pseudo-polynomial time** if its running time is polynomial in the *numeric value* of the input (which is exponential in the *length* of the input -- its number of digits).

Let the costs be c_1, ..., c_n and the corresponding values v_1, ..., v_n. We wish to maximize total value subject to the constraint that total cost is less than or equal to *C*. Then for each $i \leq C$, define $A(i)$ to be the maximum value that can be attained with total cost less than or equal to *i*. Clearly, $A(C)$ is the solution to the problem.
Define $A(i)$ recursively as follows:

- $A(0) = 0$
- $A(i) = \max \{ v_j + A(i - c_j) \mid c_j \leq i \}$

Here the maximum of the empty set is taken to be zero. Tabulating the results from $A(0)$ up through $A(C)$ gives the solution. Since the calculation of each $A(i)$ involves examining *n* items (all of which have been previously computed), and there are *C* values of $A(i)$ to calculate, the running time of the dynamic programming solution is thus $O(nC)$.
This does not contradict the fact that the knapsack problem is NP-complete, since *C*, unlike *n*, is not polynomial in the length of the input to the problem. The length of the input to the problem is proportional to the number of bits in *C*, not to *C* itself.
A similar dynamic programming solution for the *0-1 knapsack problem* also runs in pseudo-polynomial time. As above, let the costs be c_1, ..., c_n and the corresponding values v_1, ..., v_n. We wish to maximize total value subject to the constraint that total cost is less than *C*. Define a recursive function, $A(i, j)$ to be the maximum value that can be attained with cost less than or equal to *j* using items up to *i*.
We can define $A(i,j)$ recursively as follows:

- $A(0, j) = 0$
- $A(i, 0) = 0$
- $A(i, j) = A(i - 1, j)$ if $c_i > j$
- $A(i, j) = \max(A(i - 1, j), v_i + A(i - 1, j - c_i))$ if $c_i \leq j$

The solution can then be found by calculating $A(n, C)$. To do this efficiently we can use a table to store previous computations. This solution will therefore run in $O(nC)$ time and $O(nC)$ space, though with some slight modifications we can reduce the space complexity to $O(C)$.

Matrix chain multiplication

Matrix chain multiplication is an optimization problem that can be solved using dynamic programming. Given a sequence of matrices, we want to find the most efficient way to multiply these matrices together. The problem is not actually to *perform* the multiplications, but merely to decide in what order to perform the multiplications.

We have many options because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices *A*, *B*, *C*, and *D*, we would have:

$$(ABC)D = (AB)(CD) = A(BCD) = A(BC)D = ...$$

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the *efficiency*. For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix. Then,

$(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500$ operations

$A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000$ operations

Clearly the first method is the more efficient. Now that we have identified the problem, how do we determine the optimal parenthesization of a product of n matrices? We could go through each possible parenthesization (brute force), but this would require time $O(2^n)$, which is very slow and impractical for large n. The solution, as we will see, is to break up the problem into a set of related subproblems. By solving subproblems one time and reusing these solutions many times, we can drastically reduce the time required. This is known as dynamic programming.

The algorithm

To begin, let's assume that all we really want to know is the minimum cost, or minimum number of arithmetic operations, needed to multiply out the matrices. If we're only multiplying two matrices, there's only one way to multiply them, so the minimum cost is the cost of doing this. In general, we can find the minimum cost using the following recursive algorithm:

- Take the sequence of matrices and separate it into two subsequences.
- Find the minimum cost of multiplying out each subsequence.
- Add these costs together, and add in the cost of multiplying the two result matrices.
- Do this for each possible position at which the sequence of matrices can be split, and take the minimum over all of them.

For example, if we have four matrices *ABCD*, we compute the cost required to find each of (*A*)(*BCD*), (*AB*)(*CD*), and (*ABC*)(*D*), making recursive calls to find the minimum cost to compute *ABC*, *AB*, *CD*, and *BCD*. We then choose the best one. Better still, this yields not only the minimum cost, but also demonstrates the best way of doing the multiplication: just group it the way that yields the lowest total cost, and do the same for each factor.

Unfortunately, if we implement this algorithm we discover that it's just as slow as the naive way of trying all permutations! What went wrong? The answer is that we're doing a lot of redundant work. For example, above we made a recursive call to find the best cost for computing both *ABC* and *AB*. But finding the best cost for computing ABC also requires finding the best cost for *AB*. As the recursion grows deeper, more and more of this type of unnecessary repetition occurs.

One simple solution is memoization: each time we compute the minimum cost needed to multiply out a specific subsequence, we save it. If we are ever asked to compute it again, we simply give the saved answer, and do not recompute it. Since there are about $n^2/2$ different subsequences, where n is the number of matrices, the space required to do this is reasonable. It can be shown that this simple trick brings the runtime down from $O(2^n)$ to $O(n^3)$, which is more than efficient enough for real applications. This is top-down dynamic programming.

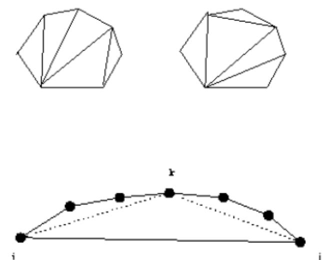
Pseudocode:
```
Matrix-Chain-Order(int p[])
{
  n = p.length - 1;
  for (i = 1; i <= n; i++) m[i,i] = 0;
  for (l=2; l<=n; l++)
  { // l is chain length
   for (i=1; i<=n-l+1; i++)
    { j = i+l-1;
      m[i,j] = MAXINT;
      for (k=i; k<=j-1; k++)
       {
         q = m[i,k] + m[k+1,j] + p[i-1]*p[k]*p[j];
         if (q < m[i,j]) {m[i,j] = q; s[i,j] = k;}
       }
    }
  }
}
```
Another solution is to anticipate which costs we will need and precompute them. It works like this:

- For each k from 2 to n, the number of matrices:
  - Compute the minimum costs of each subsequence of length k, using the costs already computed.

When we're done, we have the minimum cost for the full sequence. Although it also requires $O(n^3)$ time, this approach has the practical advantages that it requires no recursion, no testing if a value has already been computed, and we can save space by throwing away some of the subresults that are no longer needed. This is bottom-up dynamic programming: a second way by which this problem can be solved.

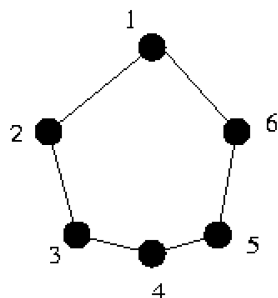
Minimum Length Triangulation

A triangulation of a polygon is a set of non-intersecting diagonals which partitions the polygon into diagonals. The length of a triangulation is the sum of the diagonal lengths. We seek to find the minimum length triangulation. Once we identify the correct connecting vertex, the polygon is partitioned into two smaller pieces, both of which must be triangulated optimally.



$t[i, i + 1] = 0, \ \ t[i, j] = \min\{t[i, k] + t[k, j] + L(i,k) + L(k,j), k = i, \ldots, j\}$

Evaluation proceeds as in the matrix multiplication example: $n(n-1)/2$ values of t, each of which takes $O(j-i)$ time if we evaluate the sections in order of increasing size.



J-i = 2
13, 24, 35, 46, 51, 62

J-i = 3
14, 25, 36, 41, 52, 63

J-i = 4
15, 26, 31, 42, 53, 64
Finish with 16