Introduction to Algorithms Lecture 7

By: Darmen Kariboz

Dynamic Programming. Part I

- Overview
- Memoization
- Tabulation
- Fibonacci
- Money Problem
- Longest Increasing Subsequence
- Longest Common Subsequence

Overview

- Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again.
- In other words, DP algorithms where next step is calculated from results of previous steps.

- I will study the theory of Dynamic Programming, then I will practice some problems on classic DP and hence I will master Dynamic Programming.

- To Master Dynamic Programming, I would have to practice Dynamic problems and to practice problems – Firstly, I would have to study some theory of Dynamic Programming

- Both the above versions say the same thing, just the difference lies in the way of conveying the message and that's exactly what Bottom Up and Top Down DP do. Version 1 can be related to as Bottom Up DP and Version-2 can be related as Top Down Dp.

Fibonacci

- In fibonacci series all next values are evaluated from two previous values.
- Formula:
- $F_0 = 0; F_1 = 1;$
- $F_x = F_{x-1} + F_{x-2}$

- This formula can be calculated by recursion, but in this case we will do same calculations multiple times and it will consume time.

Memoization (Top Down):

- The memoized program for a problem is similar to the recursive version with a small modification that it looks into a lookup table before computing solutions.
- Whenever we need solution to a subproblem, we first look into the lookup table.
- If the precomputed value is there then we return that value, otherwise we calculate the value and put the result in lookup table so that it can be reused later.

```
int lookup[] = new int[n+1];
final int NIL = -1

int fib(int n)
  {
    if (lookup[n] == NIL)
    {
      if (n <= 1)
          lookup[n] = n;
      else
          lookup[n] = fib(n-1) + fib(n-2);
    }
    return lookup[n];
}
```

Tabulation (Bottom Up):

- The tabulated program for a given problem builds a table in bottom up fashion and returns the last entry from table.
- For example, for the same Fibonacci number, we first calculate fib(0) then fib(1) then fib(2) then fib(3) and so on.
- So literally, we are building the solutions of subproblems bottom-up.

```
int fib(int n)
{
  int f[] = new int[n+1];
  f[0] = 0;
  f[1] = 1;
  for (int i = 2; i <= n; i++)
        f[i] = f[i-1] + f[i-2];
  return f[n];
}
```

	Tabulation	Memoization
State	State Transition relation is difficult to think	State transition relation is easy to think
Code	Code gets complicated when lot of conditions are required	Code is easy and less complicated
Speed	Fast, as we directly access previous states from the table	Slow due to lot of recursive calls and return statements
Subproblem solving	If all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms a top-down memoized algorithm by a constant factor	If some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are definitely required
Table Entries	In Tabulated version, starting from the first entry, all entries are filled one by one	Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memoized version. The table is filled on demand.

0	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	3	2	5	6	8	14

Money problem

You have infinite number of coins of 2, 3, 5 cents.

Calculate how many sequences to collect N from that coins, if possible.

```
int solve(int n)
{
   // base case
   if (n < 0)
      return 0;
   if (n == 0)
      return 1;

   return solve(n-1) + solve(n-3) + solve(n-5);
}
```

- Now, you can make memoization or tabulation from this recursive code

	1	2	3	4	5	6	7	8	9
a[]	4	6	2	5	1	7	5	8	4
b[]	1	2	1	2	1	3	2	4	2

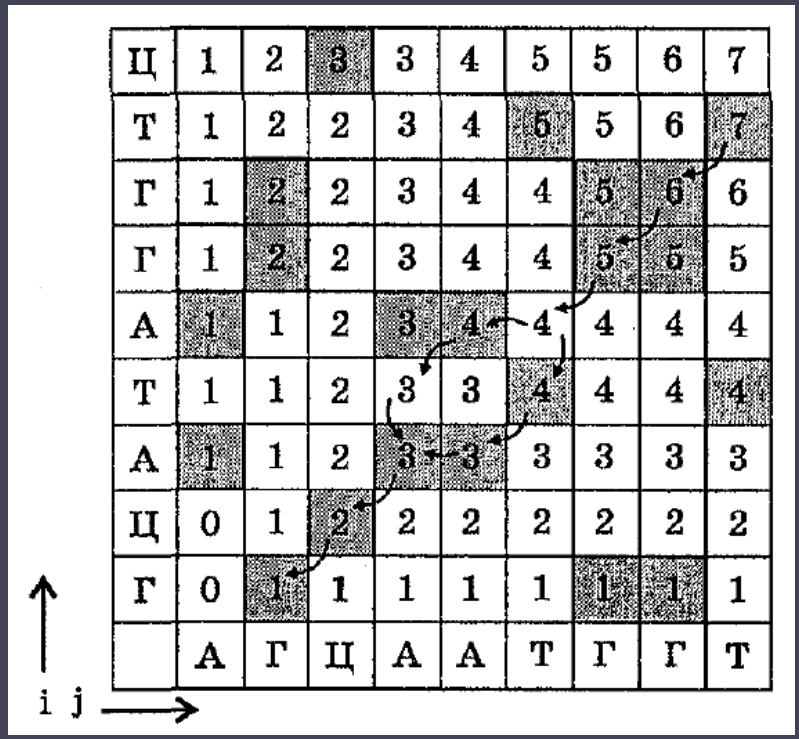
Longest increasing subsequence

* b[i] will store number of elements till i.

- **create** *b* filled with int 1
- **for** i = 1 to len(a) do
- **for** j = 1 to i-1 do
- **if** a [i]>a [j] do
- b[i]= maximum of b[i] or b[j]+1
- return max number in b

Longest Substring

Find longest substring of two given strings A and B



		А	Г	Ц	А	А	Т	Г	Г	Т
	0	0	0	0	0	0	0	0	0	0
Ц	0	0	1	2	2	2	2	2	2	2
Г	0	1	1	1	1	1	1	1	1	1
Т	1	1	2	2	3	3	4	4	4	4
А	1	1	2	3	3	4	4	4	4	4
Т	1	1	2	3	3	4	4	4	4	4
А	1	1	2	3	4	4	4	4	4	4
Г	1	2	2	3	4	4	5	5	5	5
Г	1	2	2	3	4	4	5	6	6	6
Т	1	2	2	3	4	5	5	6	7	7
Ц	1	2	3	3	4	5	5	6	7	7

i j

Longest substring problem

- T[i][j] will store length of maximum substring of $A_1 \ldots A_i$ and $B_1 \ldots B_j$.
- Each time check
  if $A_i = B_j$ then T[i][j] = T[i-1][j-1]+1,
  so if we have same letters take maximum substring without current letters and add this letter.
- Otherwise select maximum from previous results
  T[i][j] = Max(T[i-1][j],T[i][j-1])

Pseudocode of substring problem

```
For i:=l To Length (SI) Do
    For j:=l To Length (S2) Do
    Begin
        A[i,j] :=Max(A[i-l,j] ,A[i,j-l]) ;
        If SI [i]=S2 [j] Then
            A[i , j] :=Max (A[i , j] , A[i-l,j-l]+l) ;
    End;
```

Home Work

- **Find Longest decreasing sequence**
- Given numbers, find size of longest decreasing sequence in that numbers.

- **Largest rectangle**
- Given Matrix N,M filled with integer values. Find rectangle that starts in top-left corner and have maximum sum of elements inside.