Telerik Academy

Dynamic Programming

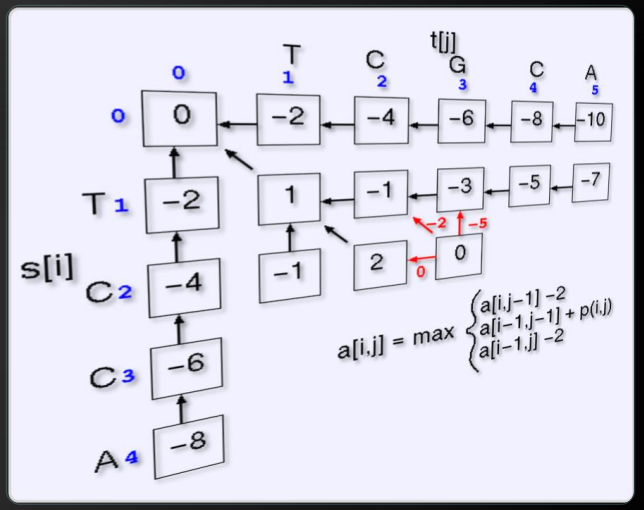
Brief Introduction in Problem Solving using Dynamic Programming and Memoization

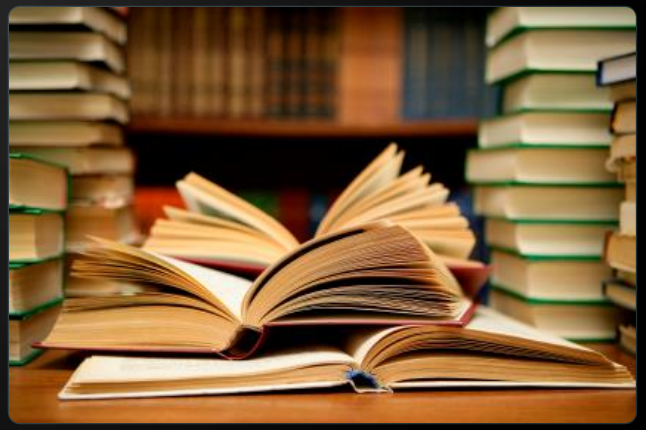
Data Structures & Algorithms

Telerik Software Academy
http://academy.telerik.com

Telerik Academy

Table of Contents





Telerik Academy

Minimum and Maximum

Telerik Academy

Minimum and Maximum

- The minimum of a set of N elements
  - The first element in list of elements ordered in incremental order (index = 1)
- The maximum of a set of N elements
  - The last element in list of elements ordered in incremental order (index = N)
- The median is the "halfway point" of the set
  - When N is odd, index = (N+1) / 2 = unique value
  - When N is even, index = $\lfloor (n+1)/2 \rfloor$ (lower median) or index = $\lceil (n+1)/2 \rceil$ (upper median)

Telerik Academy

Finding Min and Max Element

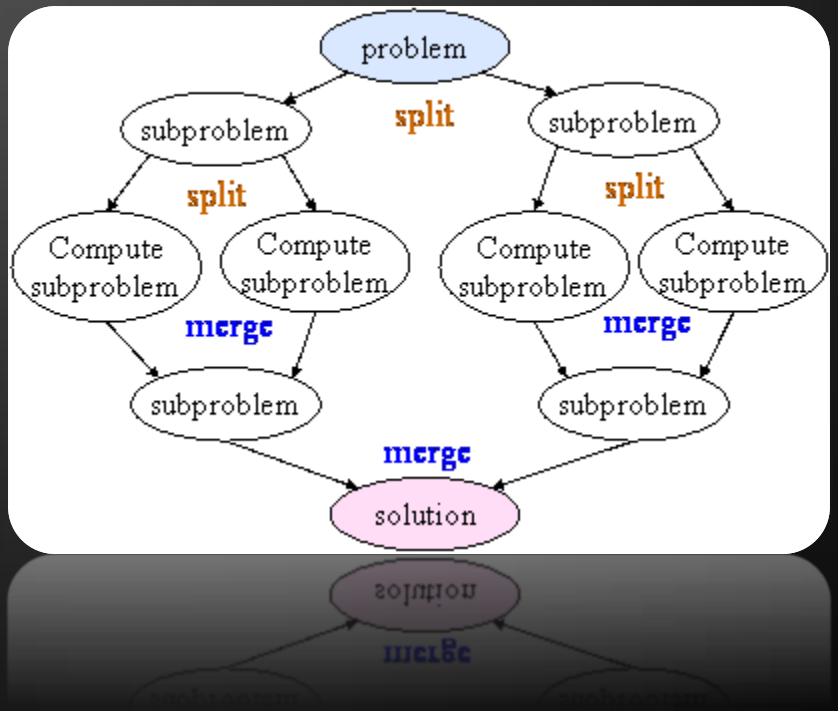
- Minimum element

```
int FindMin(int[] arr)
{
    int min = arr[0];
    for (int i = 1; i < arr.Length; i++)
        if (arr[i] < min) min = arr[i];
    return min;
}
```

- Maximum element

```
int FindMax(int arr[])
{
    int max = arr[0];
    for (int i = 1; i < arr.Length; i++)
        if (arr[i] > max) max = arr[i];
    return max;
}
```

Divide-and-Conquer



Divide-and-Conquer

- Divide: If the input size is too large to deal with in a straightforward manner
  - Divide the problem into two or more disjoint subproblems
- Conquer: conquer recursively to solve the subproblems
- Combine: Take the solutions to the subproblems and "merge" these solutions into a solution for the original problem

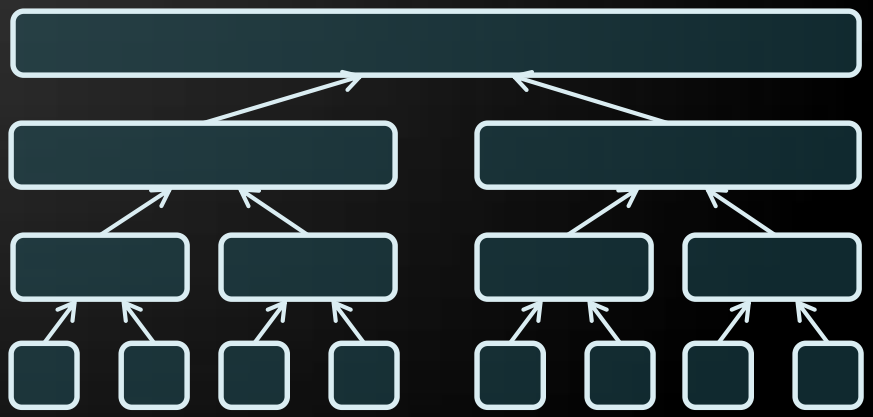
Telerik Academy

Divide-and-Conquer Example

- Merge Sort

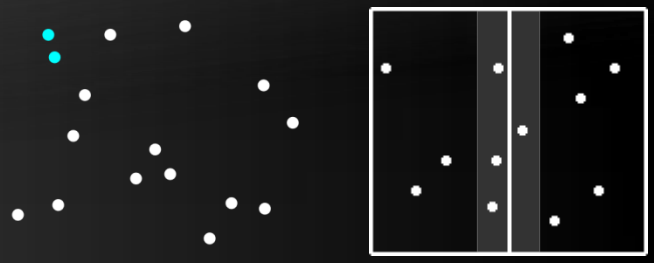
```
void MergeSort(int[] arr, int left, int right)
{
  if (right > left)
  {
    int mid = (right + left) / 2;
    MergeSort(arr, left, mid);
    MergeSort(arr, (mid+1), right);
    Merge(arr, left, (mid+1), right);
  }
}
```

- The subproblems are independent, all different

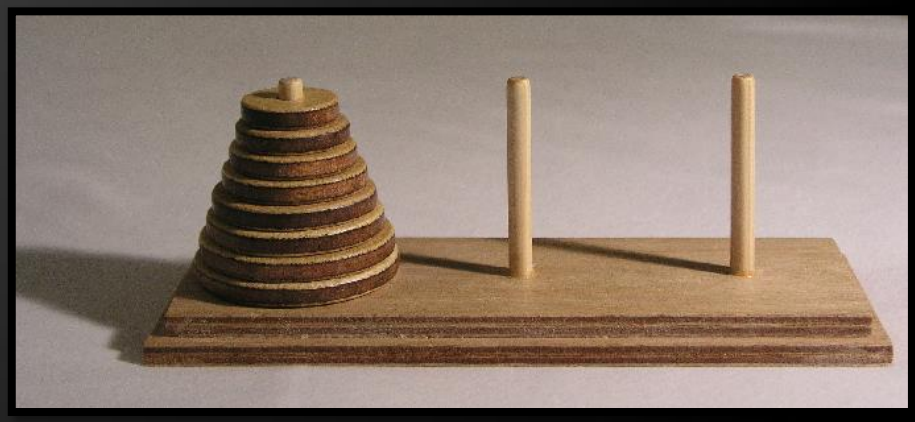


Divide-and-Conquer Algorithms

- Binary search
  - Closest pair in 2D geometry



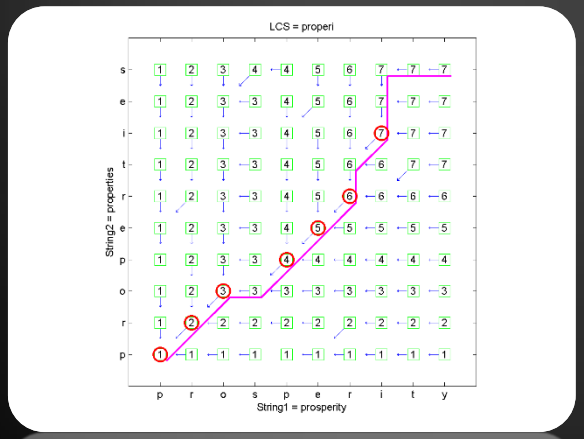
- Quick sort
- Merging arrays
  - Merge sort

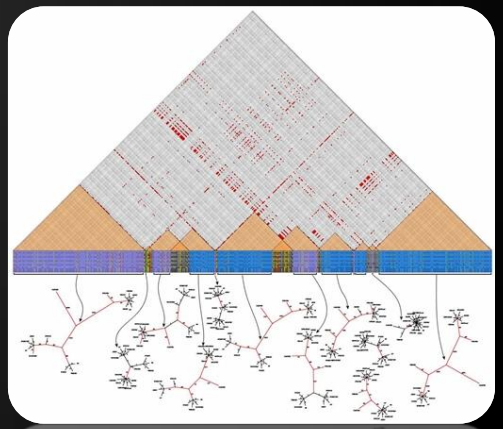


- Finding majorant
- Tower of Hanoi
- Fast multiplication
  - Strassen's Matrix Multiplication

Telerik Academy

Dynamic Programming





Dynamic Programming

- How dynamic programming (DP) works?
  - Approach to solve problems
  - Store partial solutions of the smaller problems
  - Usually they are solved bottom-up
- Steps to designing a DP algorithm:
  1. Characterize optimal substructure
  2. Recursively define the value of an optimal solution
  3. Compute the value bottom up
  4. (if needed) Construct an optimal solution

Elements of DP

- DP has the following characteristics
  - Simple subproblems
    - We break the original problem to smaller sub-problems that have the same structure
  - Optimal substructure of the problems
    - The optimal solution to the problem contains within optimal solutions to its sub-problems
  - Overlapping sub-problems
    - There exist some places where we solve the same sub-problem more than once

Common Characteristics

- The problem can be divided into *levels* with a *decision* required at each level
- Each level has a number of *states* associated with it
- The decision at one level transforms one state into a state in the next level
- Given the current state, the optimal decision for each of the remaining states does not depend on the previous states or decisions

Difference between DP and Divide-and-Conquer

- Using Divide-and-Conquer to solve problems (that can be solved using DP) is inefficient
  - Because the same common sub-problems have to be solved many times
- DP will solve each of them once and their answers are stored in a table for future use
  - Technique known as "memoization"

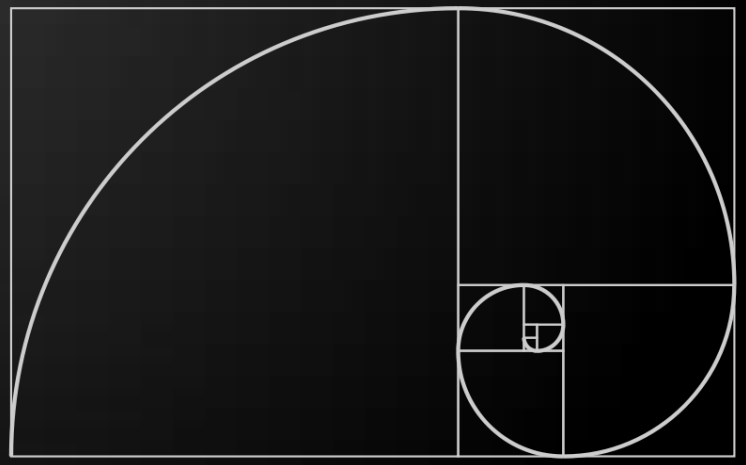
Telerik Academy

Fibonacci Numbers

From "divide and conquer" to dynamic programming

Fibonacci sequence

- The Fibonacci numbers are the numbers in the following integer sequence:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …
  - The first two numbers are 0 and 1
  - Each subsequent number is the sum of the previous two numbers
- In mathematical terms:
  - $F_n = F_{n-1} + F_{n-2}$
  - $F_0 = 0, F_1 = 1$

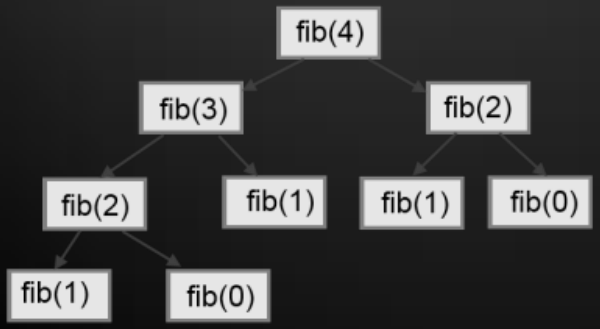


Telerik Academy

Divide and Conquer Approach

- How can we find the n^{th} Fibonacci number using recursion ("divide and conquer")
- Directly applying the recurrence formula

```
decimal Fibonacci(int n)
{
    if (n == 0) return 0;
    if (n == 1) return 1;
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```



Telerik Academy

Fibonacci and Memoization

- We can save the results from each function call
- Every time when we call the function we check if the value is already calculated
- This saves a lot of useless calculations!
- http://en.wikipedia.org/wiki/Memoization

```
decimal Fibonacci(int n)
{
    if (memo[n] != 0) return memo[n];
    if (n == 0) return 0;
    if (n == 1) return 1;
    memo[n] = Fibonacci(n - 1) + Fibonacci(n - 2);
    return memo[n];
}
```

Fibonacci and DP

- How to find the n^{th} Fibonacci number using the dynamic programming approach?
  - We can start solving the Fibonacci problem from bottom-up calculating partial solutions
  - We know the answer for the 0^{th} and the 1^{st} number of the Fibonacci sequence

0	1	2	3	4	5	6	...	i^{th}	...	n^{th}
0	1	1	2	3	5	8	...	$F_{i-1} + F_{i-2}$	...	...

  - And we know the formula to calculate each of the next numbers ($F_i = F_{i-1} + F_{i-2}$)

Compare Fibonacci Solutions

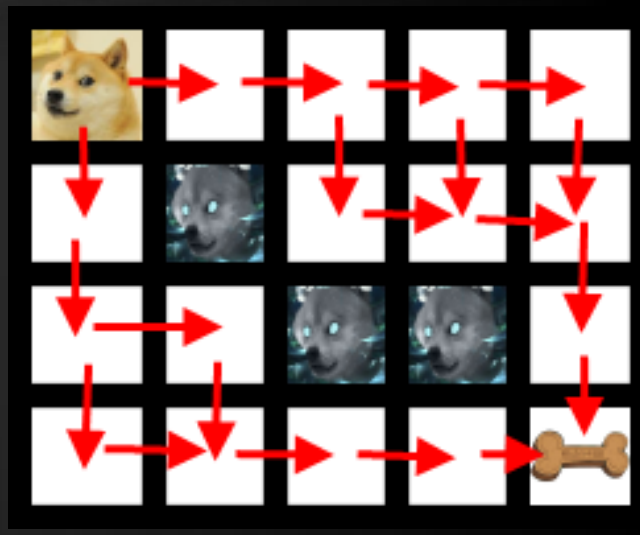
- Recurrent solution
  - Complexity: ~ $O(1.6^n)$
- DP or memoization solution
  - Complexity: ~ O(n)
- Dynamic programming solutions is way faster than the recurrent solution
  - If we want to find the 36^{th} Fibonacci number:
    - Recurrent solution takes ~48 315 633 steps
    - Dynamic programming solution takes ~36 steps

Telerik Academy

Moving Problem

Moving Problem

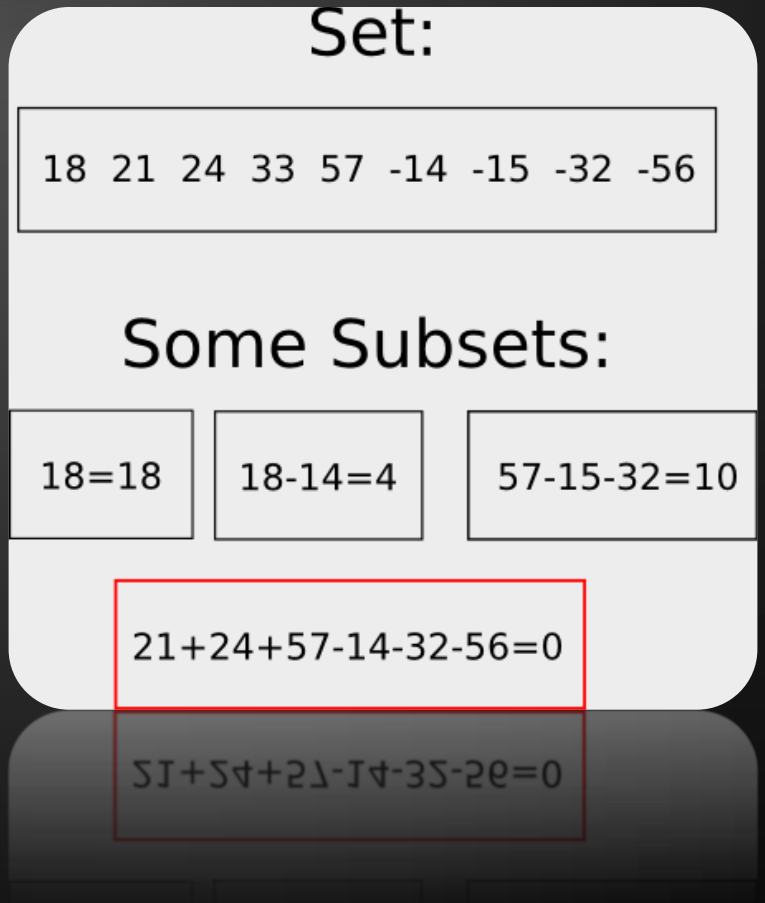
- In many DP problems there is a moving object with some restrictions
- For example - in how many ways you can reach from top left corner of a grid to the bottom right



- You can move only right and down
- Some cells are unreachable

Telerik Academy

Subset Sum Problem



Telerik Academy

Subset Sum Problems

- Given a set of integers, is there a non-empty subset whose sum is zero?
- Given a set of integers and an integer S, does any non-empty subset sum to S?
- Given a set of integers, find all possible sums
- Can you equally separate the value of coins?

Subset Sum Problem

- Solving the subset sum problem:
  - numbers = { 3, 5, -1, 4, 2 }, sum = 6
  - start with possible = { 0 }
- Step 1: obtain all possible sums of { 3 }
  - possible = { 0 } ∪ { 0+3 } = { 0, 3 }
- Step 2: obtain all possible sums of { 3, 5 }
  - possible = { 0, 3 } ∪ { 0+5, 3+5 } = { 0, 3, 5, 8 }
- Step 3: obtain all possible sums of { 3, 5, -1 }
  - possible = { 0, 3, 5, 8 } ∪ { 0-1, 3-1, 5-1, 8-1 } = { -1, 0, 2, 3, 4, 5, 7, 8 }
- …

Subset Sum Problem – C++

```
for(int i = 0; i < N; i++)
{
   int newminpos = minpos, newmaxpos = maxpos;
   int newpossible[OFFSET + OFFSET] = { 0 };
   for(int j = maxpos; j >= minpos; j--) // j = one possible sum
   {
      if (possible[j+OFFSET]) newpossible[j+nums[i]+OFFSET] = 1;
      if (j+nums[i] > newmaxpos) newmaxpos = j+nums[i];
      if (j+nums[i] < newminpos) newminpos = j+nums[i];
   }
   minpos = newminpos;
   maxpos = newmaxpos;
   for(int j = maxpos; j >= minpos; j--)
      if (newpossible[j+OFFSET] == 1)
         possible[j+OFFSET] = 1;
   if (nums[i] > maxpos) maxpos = nums[i];
   if (nums[i] < minpos) minpos = nums[i];
   possible[nums[i]+OFFSET] = 1;
}
```

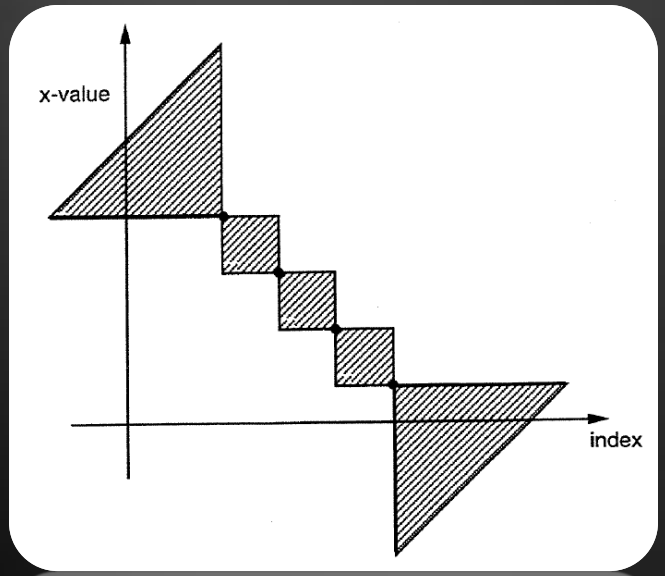
Telerik Academy

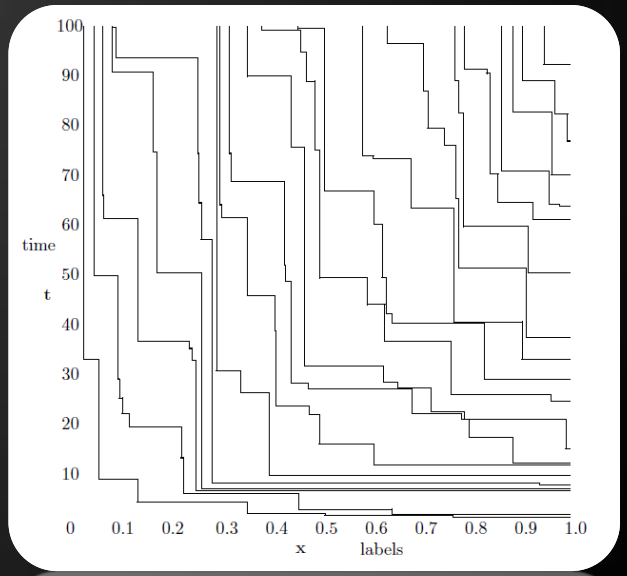
Subset Sum Problem – Answer

```
int S = 5;

if (possible[0+OFFSET]) cout << "Sum 0 is possible" << endl;
else cout << "Sum 0 is not possible" << endl;

if (possible[S+OFFSET]) cout << "Sum " << S << " is possible" <<
endl;
else cout << "Sum " << S << " is not possible" << endl;

cout << "Possible sums:";
for(int i = minpos; i <= maxpos; i++)
{
    if (possible[i+OFFSET] == 1) cout << " " << i;
}
```

```
Numbers: 2 7 -3
Sum 0 is not possible
Sum 6 is possible
Possible sums: -3 -1 2 4 6 7 9
```

Longest Increasing Subsequence





Telerik Academy Longest Increasing Subsequence

- Find a subsequence of a given sequence in which the subsequence elements are in sorted order, lowest to highest, and in which the subsequence is as long as possible
- This subsequence is not necessarily contiguous, or unique
- The longest increasing subsequence problem is solvable in time O(n log n)
- We will review one simple DP algorithm with complexity O(n * n)

LIS – C++ Solution

```
L[0] = 1; P[0] = -1;
for (int i = 1; i < N; i++)
{
    L[i] = 1;
    P[i] = -1;
    for (int j = i - 1; j >= 0; j--)
    {
        if (L[j] + 1 > L[i] && S[j] < S[i])
        {
            L[i] = L[j] + 1;
            P[i] = j;
        }
    }
    if (L[i] > maxLength)
    {
        bestEnd = i;
        maxLength = L[i];
    }
}
```

i =	0	1	2	3	4	5	6	7	8
Sequence S_i =	2	4	3	5	1	7	6	9	8
Length L_i =	1	2	2	3	1	4	4	5	5
Predecessor P_i =	-1	1	0	2	-1	3	3	6	6

Telerik Academy

LIS – Restore the Sequence

```
cout << "Max length: " << maxLength << endl;
cout << "Sequence end index: " << bestEnd << endl;
cout << "Longest subsequence:";
int ind = bestEnd;
while(ind != -1)
{
    cout << " " << S[ind];
    ind = P[ind];
}
cout << endl;
```

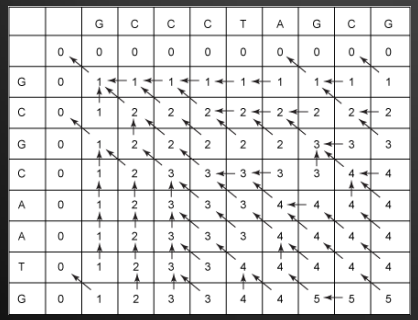
```
Max length: 5
Sequence end index: 7
Longest subsequence: 9 6 5 3 2
```

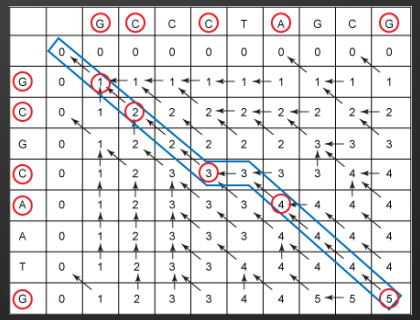
i =	0	1	2	3	4	5	6	7	8
Sequence S_i =	2	4	3	5	1	7	6	9	8
Length L_i =	1	2	2	3	1	4	4	5	5
Predecessor P_i =	-1	1	0	2	-1	3	3	6	6

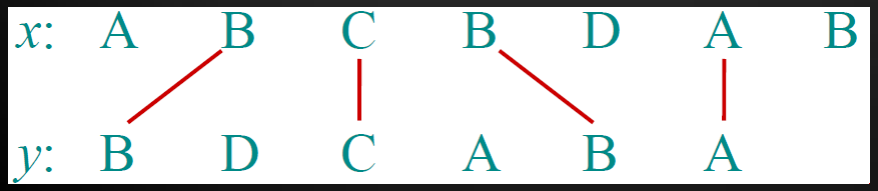
Telerik Academy

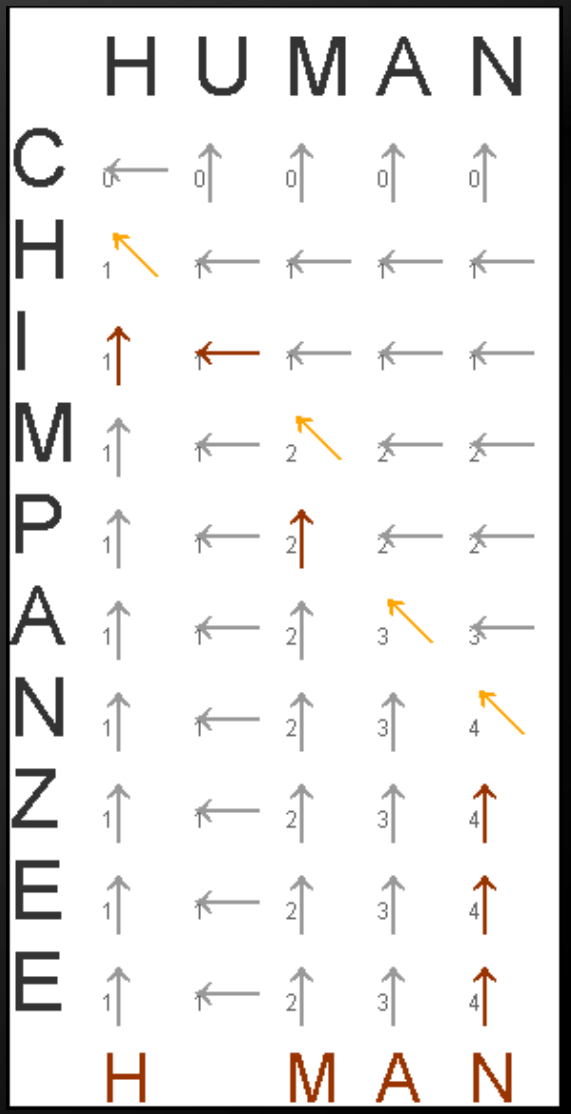
Longest Common Subsequence

Recursive and DP Approach





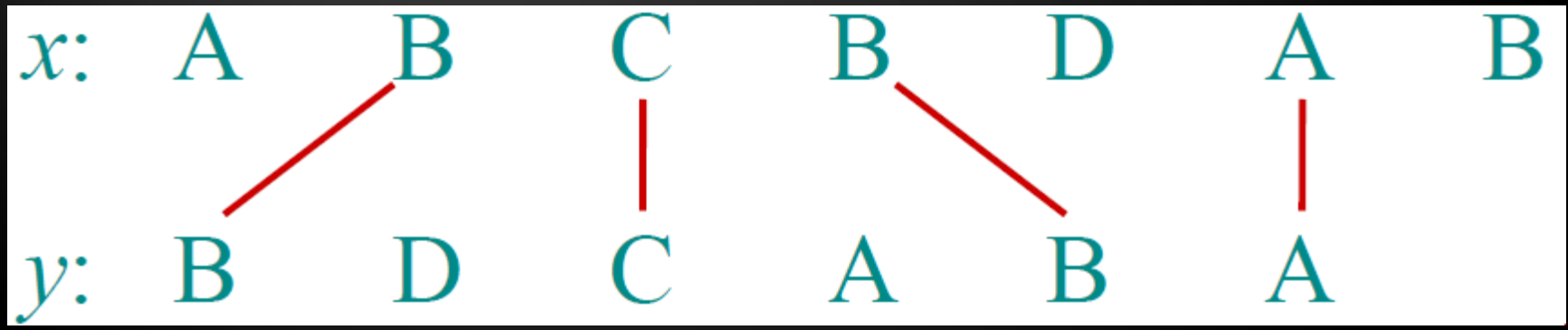




Telerik Academy

Longest Common Subsequence

- Given two sequences x[1 . . m] and y[1 . . n], find a longest common subsequence (LCS) to them both
- For example if we have x = "ABCBDAB" and y = "BDCABA" their longest common subsequence will be "BCBA"



LCS – Recursive Approach

- S_1 = GCCCTAGCG, S_2 = GCGCAATG
  - Let C_1 = the right-most character of S_1
  - Let C_2 = the right-most character of S_2
  - Let S_1' = S_1 with C_1 "chopped-off"
  - Let S_2' = S_2 with C_2 "chopped-off"
- There are three recursive subproblems:
  - L_1 = LCS(S_1', S_2)
  - L_2 = LCS(S_1, S_2')
  - L_3 = LCS(S_1', S_2')

LCS – Recursive Approach (2)

- The solution to the original problem is whichever of these is the longest:
  - L_1
  - L_2
  - If C_1 is not equal to C_2, then L_3
  - If C_1 equals C_2, then L_3 appended with C_1
- This recursive solution requires multiple computations of the same sub-problems
- This recursive solution can be replaced with DP

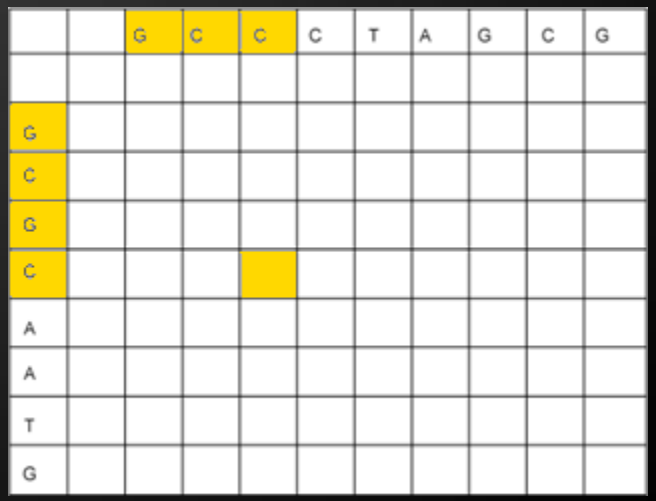
Initial LCS table

- To compute the LCS efficiently using dynamic programming we start by constructing a table in which we build up partial results

		G	C	C	C	T	A	G	C	G
G										
C										
G										
C										
A										
A										
T										
G										

Initial LCS table (2)

- We'll fill up the table from top to bottom, and from left to right
- Each cell = the length of an LCS of the two string prefixes up to that row and column
- Each cell will contain a solution to a sub-problem of the original problem
  - S_1 = GCCCTAGCG
  - S_2 = GCGCAATG



		G	C	C	C	T	A	G	C	G
G										
C										
G										
C										
A										
A										
T										
G										

Telerik Academy

LCS table – base cases filled in

- Each empty string has nothing in common with any other string, therefor the 0-length strings will have values 0 in the LCS table

		G	C	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0	0
G	0									
C	0									
G	0									
C	0									
A	0									
A	0									
T	0									
G	0									

```
for (i=0; i<=n; i++)
{
    c[i][0] = 0;
}
```

```
for (i=0; i<=m; i++)
{
    c[0][i] = 0;
}
```

Telerik Academy

LCS – C++ Code Solution

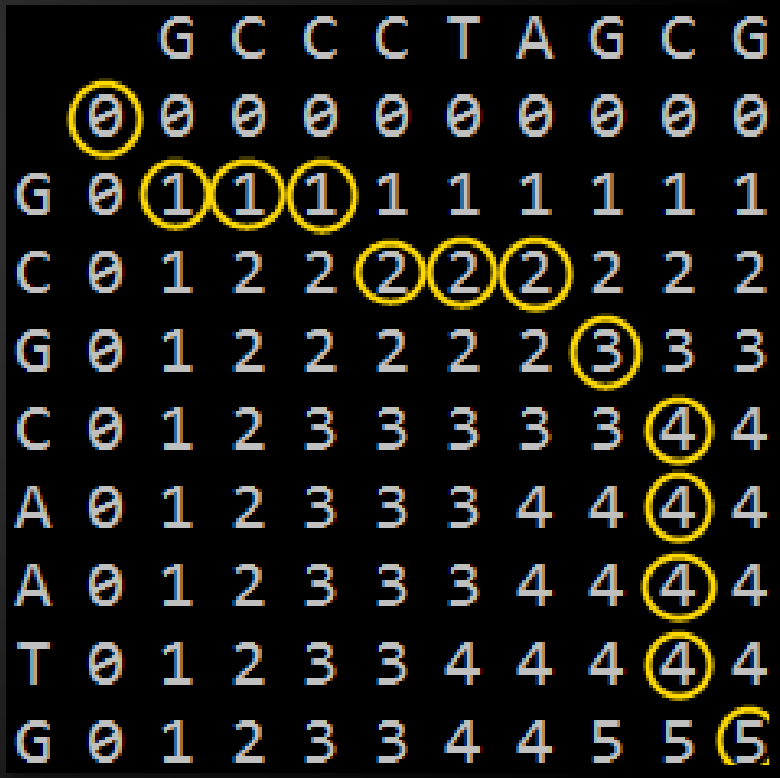
```
int LCS(string X, string Y)
{
    int m = X.length();
    int n = Y.length();
    for (int i = 1; i <= m; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            if (X[i-1] == Y[j-1])
            {
                c[i][j] = c[i-1][j-1] + 1;
            }
            else
            {
                c[i][j] = max(c[i][j-1], c[i-1][j]);
            }
        }
    }
    return c[m][n];
}
```

		G	C	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1
C	0	1	2	2	2	2	2	2	2	2
G	0	1	2	2	2	2	2	3	3	3
C	0	1	2	3	3	3	3	3	4	4
A	0	1	2	3	3	3	4	4	4	4
A	0	1	2	3	3	3	4	4	4	4
T	0	1	2	3	3	4	4	4	4	4
G	0	1	2	3	3	4	4	5	5	5

Telerik Academy

LCS – Reconstruct the Answer

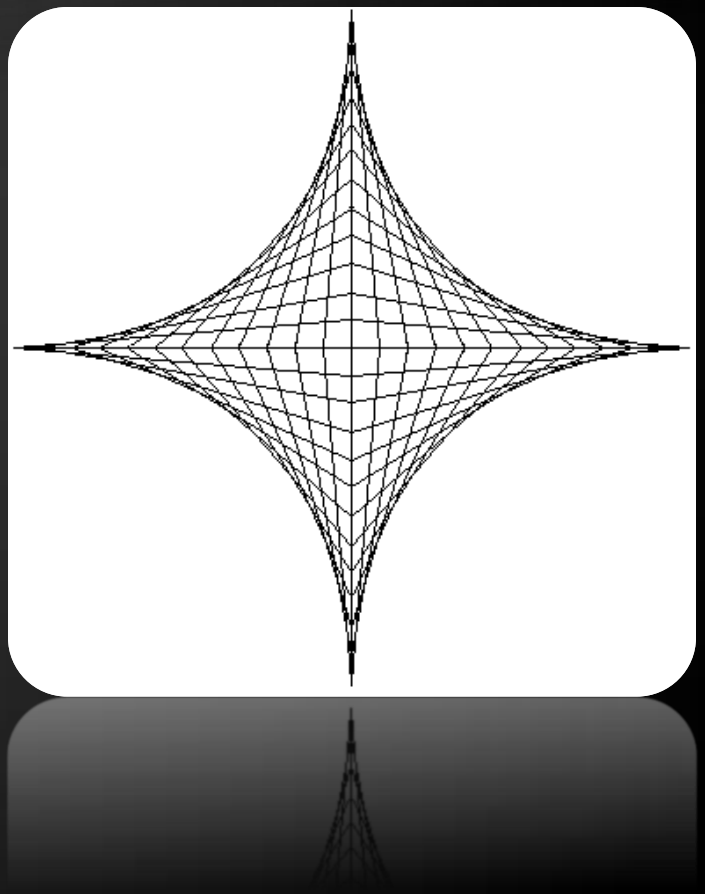
```
void printLCS(int i,int j)
{
    if (i==0 || j==0) return;
    if (X[i-1] == Y[j-1])
    {
       printLCS(i-1, j-1);
       cout << X[i-1];
    }
    else if (c[i][j] == c[i-1][j])
    {
        printLCS(i-1, j);
    }
    else
    {
        printLCS(i, j-1);
    }
}
```



		G	C	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1
C	0	1	2	2	2	2	2	2	2	2
G	0	1	2	2	2	2	2	3	3	3
C	0	1	2	3	3	3	3	3	4	4
A	0	1	2	3	3	3	4	4	4	4
A	0	1	2	3	3	3	4	4	4	4
T	0	1	2	3	3	4	4	4	4	4
G	0	1	2	3	3	4	4	5	5	5

DP Applications

- Areas
  - Bioinformatics
  - Control theory
  - Information theory
  - Operations research
  - Computer science:
    - Theory
    - Graphics
    - AI



Some Famous Dynamic Programming Algorithms

- Integer Knapsack Problem
- Unix diff for comparing two files
- Bellman–Ford algorithm for finding the shortest distance in a graph
- Floyd's All-Pairs shortest path algorithm
- Cocke-Kasami-Younger for parsing context free grammars
- en.wikipedia.org/wiki/Dynamic_programming#Algorithms_that_use_dynamic_programming

Summary

- Divide-and-conquer method for algorithm design
- Dynamic programming is a way of improving on inefficient divide-and-conquer algorithms
- Dynamic programming is applicable when the sub-problems are dependent, that is, when sub-problems share sub-sub-problem
- Recurrent functions can be solved efficiently
- Longest increasing subsequence and Longest common subsequence problems can be solved efficiently using dynamic programming approach

Telerik Academy

Dynamic Programming

http://academy.telerik.com

Homework

1. Write a program based on dynamic programming to solve the "Knapsack Problem": you are given N products, each has weight W_i and costs C_i and a knapsack of capacity M and you want to put inside a subset of the products with highest cost and weight ≤ M. The numbers N, M, W_i and C_i are integers in the range [1..500]. Example: M=10 kg, N=6, products:
   - beer – weight=3, cost=2
   - vodka – weight=8, cost=12
   - cheese – weight=4, cost=5
   - nuts – weight=1, cost=4
   - ham – weight=2, cost=3
   - whiskey – weight=8, cost=13

Optimal solution:
- nuts + whiskey
- weight = 9
- cost = 17

Homework (2)

2. Write a program to calculate the "Minimum Edit Distance" (MED) between two words. MED(x, y) is the minimal sum of costs of edit operations used to transform x to y. Sample costs are given below:
   - cost (replace a letter) = 1
   - cost (delete a letter) = 0.9
   - cost (insert a letter) = 0.8

   Example: x = "developer", y = "enveloped" → cost = 2.7
   - delete 'd': "developer" → "eveloper", cost = 0.9
   - insert 'n': "eveloper" → "enveloper", cost = 0.8
   - replace 'r' → 'd': "enveloper" → "enveloped", cost = 1

Free Trainings @ Telerik Academy

- C# Programming @ Telerik Academy
  - csharpfundamentals.telerik.com

- Telerik Software Academy
  - academy.telerik.com

- Telerik Academy @ Facebook
  - facebook.com/TelerikAcademy

- Telerik Software Academy Forums
  - forums.academy.telerik.com