# Dynamic programming (algorithm type)

*Recursion, Rod cutting, Recurrence tree.*

## Recall algorithm design techniques

- Algorithm design techniques so far:
  - Brute-force algorithms
    - Linear search
  - Incremental algorithms
    - Insertion sort
  - Algorithms that use other ADTs (implemented using efficient data structures)
    - Heap sort
  - Divide-and-conquer algorithms
    - Binary search, merge sort, quick sort.

## 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** sub-problems.
- **Conquer**: Use divide-and-conquer recursively to solve the sub-problems.
- **Combine**: Take the solutions to the sub-problems and combine these solutions into a solution for the original problem.

## Dynamic Programming

What if the sub-problems overlap?

- Sub-problems share sub-sub-problems

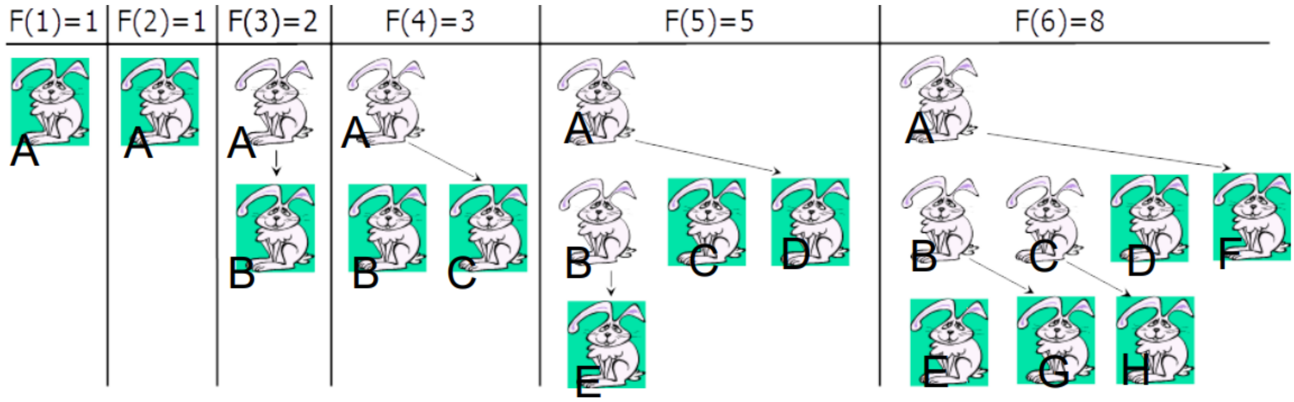
In this case, a divide-and-conquer algorithm does more work than necessary, because it needs to repeatedly solve the overlapped sub-sub-problems.

Let's see a concrete example - Fibonacci numbers.

## Fibonacci Numbers

Leonardo Fibinacci (1202):

- We have a rabbit in the beginning
- A rabbit starts producing offspring on the second generation after its birth and produces one child each generation.
- How many rabbits will there be after n generations?




$F(n) = F(n-1) + F(n-2)$

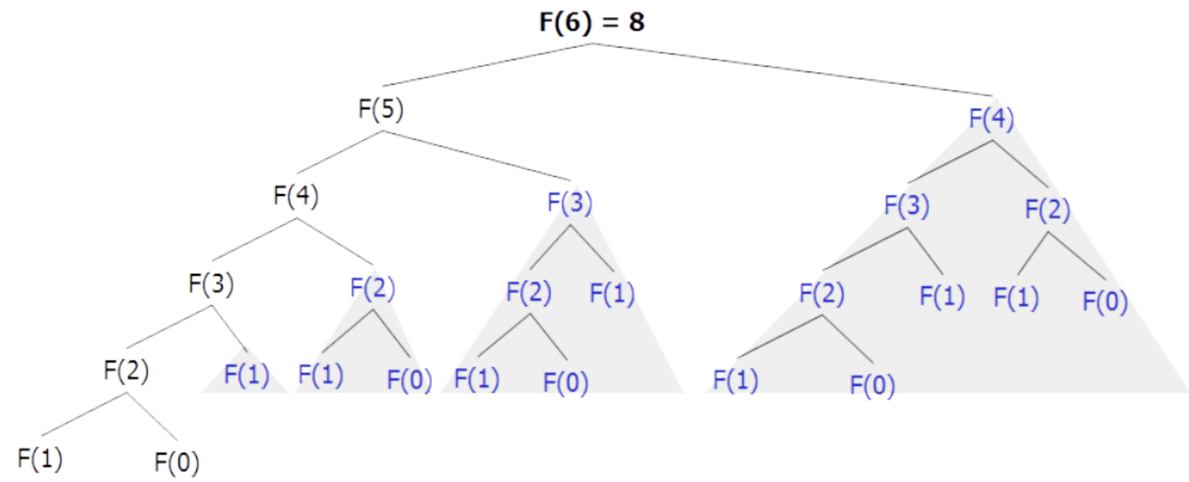
$F(0) = 0, F(1) = 1$

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34..$

```
FibonacciR(n)
if n <= 1 then return n
else return FibonacciR(n-1) + FibonacciR(n-2)
```

Straightforward recursive procedure is slow! We have two sub problems and their size is one smaller and two smaller.

Let's draw the recursion tree:




We keep calculating the same values!

- Recurrence
  - $T(n) = T(n-1) + T(n-2) + \Theta(1)$
- $T(n) \geq 2T(n-2) + a$ and $T(1) = T(0) = 1$
- Solving the recurrence.
  - Which method shall we use? Can we use the master method?
  - Reacted substitution method
  - $T(n)=2T(n-2)+a$
  - $T(n-2)=2T(n-4)+a$
  - $T(n)=2^2T(n-4)+(2+1)a$
  - $T(n-4)=2T(n-6)+a$
  - $T(n)=2^3T(n-6)+(2^2+2+1)a$
  - $T(n)= 2^iT(n-2*i)+(2^{i-1} +\ldots+2+1)a = 2^iT(n-2*i)+a \sum_{k=0}^{i-1} 2^k$
  - When $i=n/2$, we have $T(n-2*i)=T(0)$, i.e., the base case
  - $T(n)=2^{n/2}\, T(0) + a* 2^{n/2}= (a+1)\, 2^{n/2}$

For real $x \neq 1$, the summation

CLRS P1147 A.5

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n$$

is a ***geometric*** or ***exponential series*** and has the value

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1} .$$

- When $T(n)=2T(n-2)+a$, we get
  - $T(n)=2^{n/2}\, T(0) + a* 2^{n/2}= (a+1)\, 2^{n/2}$
- Recall the original recurrence $T(n) \geq 2T(n-2) + a$, then
  - $T(n) \geq (a+1)\, 2^{n/2} \approx (a+1)\, 1.4^n$
- Running time is at least ***exponential!***

- Dynamic programming
  - We can calculate F (n) in linear time by remembering solutions to the solved sub-problems
- Compute solution in a bottom-up fashion
- Trade space for time!
  - Linear time!

```
Fibonacci(n)
F[0]←0
F[1]←1
for i ← 2 to n do
   F[i] ← F[i-1] + F[i-2]
return F[n]
```

## Dynamic Programming

Why and when to use DP?

***When sub-problems overlap, a divide-and-conquer algorithm does more work than necessary, because it needs to repeatedly solve the overlapped sub-sub-problems.***

How does DP work?

**A dynamic programming algorithm solves each sub-sub-problem only once and then saves its result (in an array or a hash table), thus avoiding the work of repeatedly solving the common sub-subproblems.**

## Optimization Problems

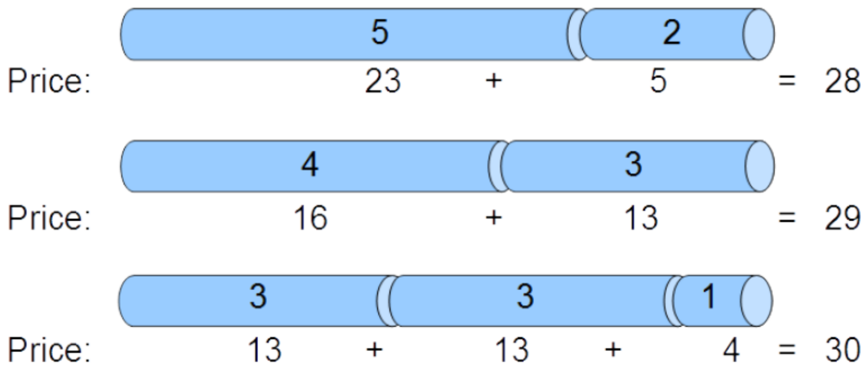
- Dynamic programming is typically applied to **optimization problems**.
- Optimization problems can have many possible solutions, each solution has a value, and we wish to find a solution with the optimal (i.e., minimum or maximum) value.
- An algorithm should compute the optimal value plus, if needed, **an** optimal solution.
- Let's see two concrete examples of optimization problems
  - Rod cutting
  - Longest common subsequences

## Rod Cutting

- The problem:
  - A steel rod of length n should be cut and sold in pieces.
  - Pieces sold only in integer sizes according to a price table P[1..n].
  - Goal: cut up the rod to maximize profit.

| *Length* | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| *Price* | 4 | 5 | 13 | 16 | 23 | 24 | 27 |

Max profit: 31
Optimal cut: 1, 1, 5




## How to solve rod cutting

- $r_n$ : the maximum profit of cutting a rod with length *n.*
- $r_n$=max (P[1]+ $r_{n-1}$, P[2]+ $r_{n-2}$, … , P[n-1]+ $r_1$ , P[n]+ $r_0$)
  - Get the maximum profit among the following combinations
    - Having a rod with length 1, i.e., P[1], and the maximum profit of the remaining rod with length n-1, i.e., $r_{n-1}$
    - Having a rod with length 2, i.e., P[2], and the maximum profit of the remaining rod with length n-2, i.e., $r_{n-2}$
    - …
    - Having a rod with length n, i.e., P[n], and the maximum profit of the remaining rod with length 0, i.e., $r_0$=0.
- For example, if n=7.
  - $r_7$ = max (P[1]+$r_6$, P[2]+$r_5$, P[3]+$r_4$, P[4]+$r_3$, P[5]+$r_2$, P[6]+$r_1$, P[7]+ $r_0$)
- We say that the rod cutting problem exhibits ***optimal substructure***
  - Optimal solutions to a problem incorporate optimal solutions to related subproblems.

### Rod Cutting Recursion

- Recursive top-down solution

```
Rod-Cut(P, n)
if n = 0 then return 0
q ← -∞
for i←1 to n do
    q ← max(q, P[i]+Rod-Cut(P, n-i))
return q
```

P, price table, an array
n, rod length, an integer

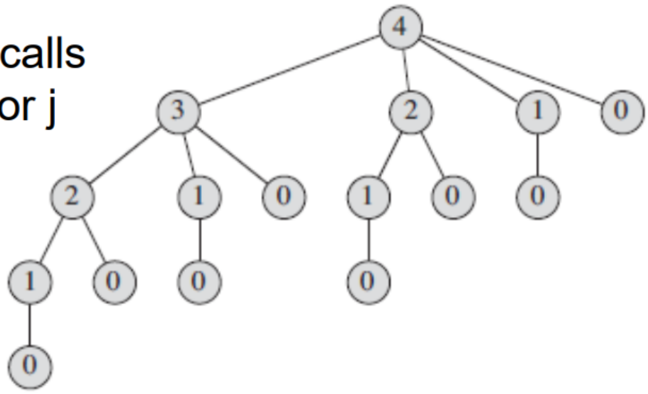
$r_n$=max (P[1]+ $r_{n-1}$, P[2]+ $r_{n-2}$, … , P[n-1]+ $r_1$ , P[n]+ $r_0$)

- Rod-Cut(P, n) calls Rod-Cut(P, *n-i*) n times where *i* starts from *1 to n*.
- Equivalently, Rod-Cut(P, n) calls Rod-Cut(P, *j*) for *j from n-1 to 0*.

### Recurrences Tree

- Let's consider the case that n = 4

Rod-Cut(P, n) calls Rod-Cut(P, j) for j from **n-1** to **0**.




Let's see the recursion tree for n and write down the recurrence.

How many sub-problems are there and what are the sizes of the sub-problems?

In order to solve a problem of size n, you need to solve **n** sub-problems whose sizes are **n-1, n-2, n-3, …, 0**, respectively.

- Recurrence:
  - $T(n)=\sum_{j=0}^{n-1} T(j)$
  - $T(0)=\Theta(1)$, constant time.

### Solving the recurrence

- $T(n)=\sum_{j=0}^{n-1} T(j)=T(0)+T(1)+\ldots+T(n-2)+T(n-1)$
- $T(n-1)=T(0)+T(1)+\ldots+T(n-2)$

- $T(n)=2T(n-1)$
- $T(n-1)=2T(n-2)$
- $T(n)=2^2T(n-2)$
- $T(n-2)=2T(n-3)$
- $T(n)=2^3T(n-3)$
- $T(n)=2^iT(n-i)$
- Base case $T(0)=a$, thus we need to make $i=n$.
- $T(n)=2^iT(n-i)=2^nT(n-n)=2^n*T(0)=a*2^n$
- Running time is ***exponential***!

### Rod cutting memorization

- Problem we have for the recursive version:
  - Solving the same sub-problems over and over.
- Dynamic programming – top-down with memoization
  - Solve each sub-problem only once and store the answers to the solved sub-problems in a table.
  - Next time, when you need to solve a solved sub-problem, just look up the table to get the answer.

- Let’s consider a memorization version of rod cutting algorithm.
- Remember the solutions in an array or a hash table.

R[n]: the maximum profit of cutting a rod with length *n*.

```
Rod-Cut-M(P, n)
  for i ←1 to n do
     R[i] ← -∞
  return Rod-Cut-M-Aux(P, n, R)

Rod-Cut-M-Aux(P, n, R)
  if R[n] ≥ 0 then return R[n]
  if n = 0 then q ← 0
  else
     q ← -∞
     for i ←1 to n do
        q ←max(q, P[i]+Rod-Cut-M-Aux(P, n-i, R))
     R[n] ←q
  return q
```

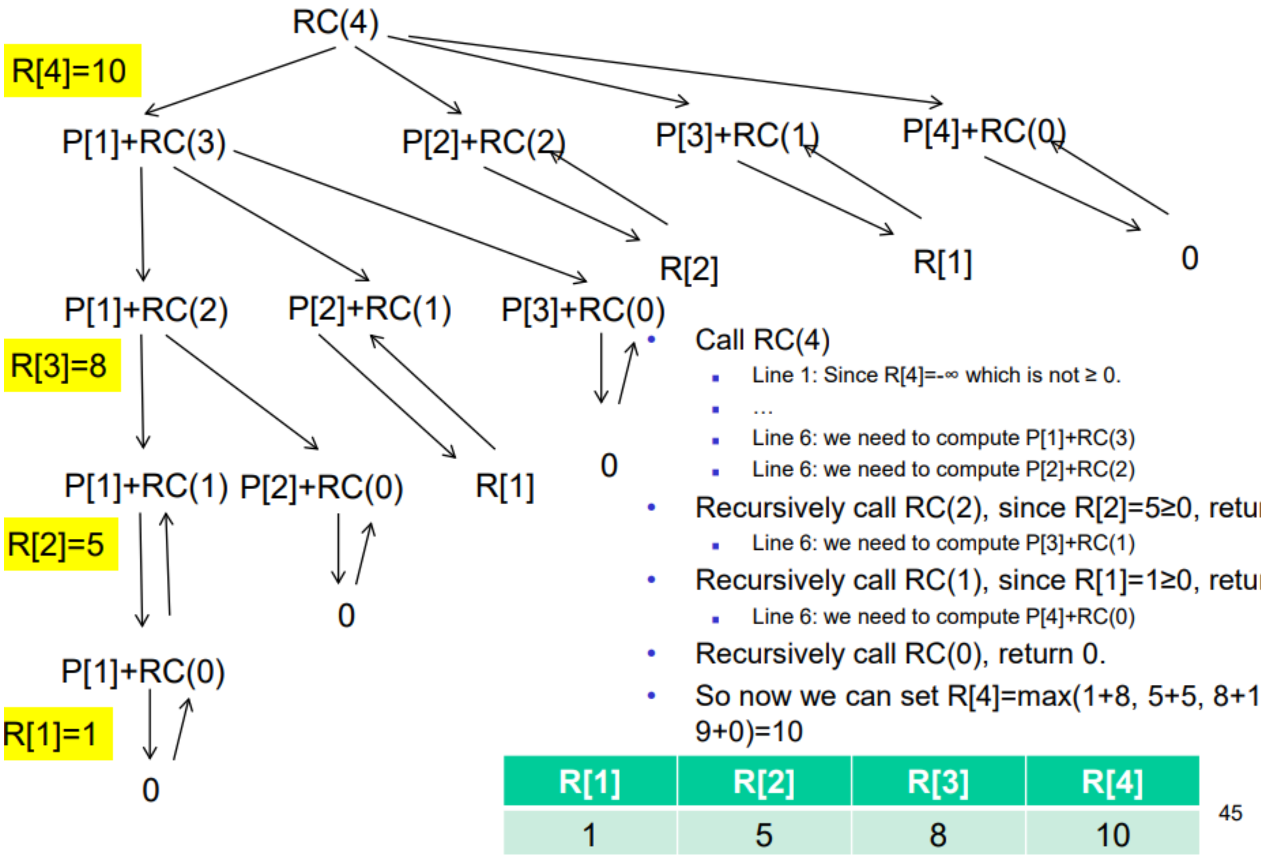
### Running example: n = 4

- Price table: P[1..4]

| Length | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Price | 1 | 5 | 8 | 9 |

- Let's see how the memorization version of rod cutting works?
  - When array R is assigned with values and what values?
  - When a subproblem is already solved so that you can directly get the value from array R?
- We denote Rod-Cut-M-Aux(P, n, R) as RC(n) for simplicity.
- We have array R in the beginning that looks like:

| R[1] | R[2] | R[3] | R[4] |
|---|---|---|---|
| -∞ | -∞ | -∞ | -∞ |




## Run time

- Recurrence: $T(n) = T(n-1) + (n-1)$
- Repeated substitution method
    - T(n-1)=T(n-2)+(n-2)
    - T(n)=T(n-2) + (n-1) + (n-2)
    - T(n-2)=T(n-3)+(n-3)
    - T(n)=T(n-3)+ (n-1) + (n-2) + (n-3)
    - …
    - T(n) = T(n-i) + (n-1) + (n-2) + (n-3) + … + (n-i)
    - To use based case T(0)=a, we set i=n, then we have
    - T(n) = T(n-n) + (n-1) + (n-2) + (n-3) + … + 2 +1 + 0
    - = a + n(n-1)/2
    - = $\Theta(n^2)$
- So So we reduce an **exponential** naive recursion to a **quadratic** recursion with memoization.
    - The overhead is an additional Θ(n) array R.

## Rod Cutting Bottom-up

- Problem we have for the recursive version:
    - Solving the same sub-problems over and over.
- Dynamic programming – bottom-up without recursion.

- Depending on some natural notion on the size of a sub-problem.
  - Solving any particular sub-problem depends only on solving **smaller** sub-problems.
  - Sort the sub-problems by size and solve them in size order, smallest first. And save the solutions.
- Let's consider a bottom-up version rod cutting algorithm.

```
Rod-Cut-B(P, n)
R[0] ← 0
for  j ←1 to n do
   q ← -∞
   for i ←1 to j do
      q ← max(q, P[i]+R[j-i])
   R[j] ← q
return R[n]
```

R[n]: the maximum profit of cutting a rod with length *n*.

We compute R[j], where j is from 1 to n, i.e., from small size to big size.

When you solve a sub-problem, e.g., R[j], you need to use the solutions to smaller sub-problems---that is why i is from 1 to j.

- Run time $\Theta(n^2)$

**Running example: n = 4**

- Price table: P[1..4]

| **Length** | **1** | **2** | **3** | **4** |
|---|---|---|---|---|
| Price | 1 | 5 | 8 | 9 |

- Let's see how the bottom-up version of rod cutting works.
  - How array R is populated with the right values?
- Cut it into "2,2".

## Getting an optimal solution

- So far, our dynamic programming solutions return the optimal profit.
- But, they do not return an actual optimal solution - a list of piece sizes.
  - To return an actual optimal solution, we have to record the choices that lead to optimal profit.
  - S[j]: to achieve the maximum profit of cutting a length j rod, we need to have a piece with length S[j].

```
Rod-Cut-B-Ext(P, n)
R[0] ← 0
for  j ←1 to n do
   q ← -∞
   for i ←1 to j do
      if q < P[i]+R[j-i] then
         q ← P[i]+R[j-i]
         S[j] ← i
   R[j] ← q
return (R[n], S)
```

```
Rod-Cut-B(P, n)
for i ←1 to j do
   q ← max(q, P[i]+R[j-i])
R[j] ← q
```

53

- R[0]=0, base case.
- R[1]=max(P[1]+R[0])=1
  - S[1]=1
- R[2]=max(P[1]+R[1], P[2]+R[0])=max(2, 5)=5
  - S[2]=2
- R[3]=max(P[1]+R[2], P[2]+R[1], P[3]+R[0])=max(6, 6, 8)=8
  - S[3]=3
- R[4]=max(P[1]+R[3], P[2]+R[2], P[3]+R[1] , P[4]+R[0])=max(9, 10, 9, 9)=10.
  - S[4]=2

### Reconstructing a solution

```
Print-Rod-Cut-Solution(P, n)
(Cost, S)← Rod-Cut-D-Ext(P,n)
while  n > 0 do
    print S[n]
    n ←n - S[n]
return Cost
```

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $r[i]$ | 0 | 1 | 5 | 8 | 10 | 13 | 17 | 18 | 22 | 25 | 30 |
| $s[i]$ | 0 | 1 | 2 | 3 | 2 | 2 | 6 | 1 | 2 | 3 | 10 |

- If n=4, the optimal cutting is 2, 2.
- If n=7, the optimal cutting is 1, 6.
- If n=9, the optimal cutting is 3, 6.
- If n=10, the optimal cutting is 10, i.e., not cutting.

### Memoization vs. Bottom-Up

- Pros and cons:
  - They should have the same asymptotic running time.
  - Recursion (Memorization) is usually slower than loops (Bottom-Up).
  - If not all sub-problems need to be solved, memorization only solves the necessary ones.

### Mini quiz

- If you have a rod of length 7, what is the optimal profit and what is the corresponding optimal cut?

| Length | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Price | 4 | 5 | 13 | 16 | 23 | 24 | 27 |

- R[0]=0, base case.
- R[1]=max(P[1]+R[0])=4
  - S[1]=1
- R[2]=max(P[1]+R[1], P[2]+R[0])=max(4+4, 5+0)=8
  - S[2]=1
- R[3]=max(P[1]+R[2], P[2]+R[1], P[3]+R[0])=max(4+8, 5+4, 13+0)=13
  - S[3]=3
- R[4]=max(P[1]+R[3], P[2]+R[2], P[3]+R[1], P[4]+R[0])=max(4+13, 5+8, 13+4,16+0)=17.
  - S[4]=1 or 3
- R[5]=max(P[1]+R[4], P[2]+R[3], P[3]+R[2], P[4]+R[1], P[5]+R[0])=max(4+17, 5+13, 13+8, 16+4, 23+0)=23.
  - S[5]=5
- R[6]=max(P[1]+R[5], P[2]+R[4], P[3]+R[3], P[4]+R[2], P[5]+R[1] , P[6]+R[0])=max(4+23, 5+17, 13+13, 16+8, 23+4, 24+0)=27.
  - S[6]=1 or 5
- R[7]=max(P[1]+R[6], P[2]+R[5], P[3]+R[4], P[4]+R[3], P[5]+R[2] , P[6]+R[1], p[7]+R[0])=max(4+27, 5+23, 13+17, 16+13, 23+8, 24+4, 27+0)=31.
  - S[6]=1 or 5

## Longest Common Subsequence

- Given two strings X and Y
- There is a need to quantify how similar they are:
  - Comparing DNA sequences in studies of evolution of different species.
  - Spell checkers.
- One of the measures of similarity is the length of the Longest Common Subsequence (LCS).
  - Z is a subsequence of X, if it is possible to generate Z by skipping some (possibly none) characters from X
  - X ="ACGGTTA", Z="CTA" is a subsequence of X.
  - Y ="CGTAT".
  - LCS(X,Y ) = "CGTA" or "CGTT".
- To solve LCS problem we have to find "skips" that generate LCS(X,Y ) from X , and "skips" that generate LCS(X,Y ) from Y

## Solution Outline

- Given $X_m =" x_1x_2x_3...x_{m-1}x_m ": \ and\, Y_n =" y_1y_2y_3...y_{n-1}y_n "$
- Brute-force solution
  - Enumerate all subsequences of $X_m$ and check each to see whether it is also a subsequence of $Y_n$.
  - Keep tracking the longest subsequence we found.
  - Exponential run time: $2^m$,because $X_m$ has $2^m$ subsequences.
- Recursive solution
  - We make Z to be empty and proceed from the ends of X and Y.
  - If $x_m = y_n$, append this symbol to the end of Z, and find LCS($X_{m-1}, Y_{n-1}$) as the beginning of Z.
  - If $x_m \neq y_n$, compute LCS($X_m, Y_{n-1}$) and LCS($X_{m-1}, Y_n$), and the longer one is the result.

**Recurrence**

- Let c[i, j] be the length of an LCS of $X_i$ and $Y_j$, where 1≤i ≤ m, 1 ≤ j ≤ n,

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

  - Finally, c[m, n] gives the length of the LCS between $X_m$ and $Y_n$

- Pseudo code is shown in P394, CLRS.