Dynamic Programming

(Most of the images and text are excerpted from Cormen et al.)

Up to now:

- ADTs
  - Stacks, Queues
  - Linked Lists
  - Trees (BST, RedBlack)
  - Etc.
- Sort/Search algorithm
  - Binary search
  - Merge Sort, InsertionSort
  - QuickSort
  - Etc.

- These are well-defined, straightforward algorithms.
- You don't need to implement these algorithms.
- You can download open source code and reuse it in your project.
- And, that's ok.

Now:

Up to now:

- ADTs
  - Stacks, Queues
  - Linked Lists
  - Trees (BST, RedBlack)
  - Etc.
- Sort/Search algorithm
  - Binary search
  - Merge Sort, InsertionSort
  - QuickSort
  - Etc.

- These are well-defined, straightforward algorithms.
- You don't need to implement these algorithms.
- You can download open source code and reuse it in your project.
- And, that's ok.

Now:

- Design&Analysis Techniques
  - Dynamic Programming
  - Greedy Algorithms
  - Amortized Analysis
  - Divide & Conquer
  - Etc.

- These are not well-defined, ready-to-use algorithms.
- You can not find a dynamic programming algorithm from Internet and reuse it.
- It is a design pattern, that you can apply to your specific problem.

How can we apply a design technique?

- First we need to understand our specific problem. It may be a
  - business problem
  - scientific problem
  - etc.
- And if our problem exhibits some particular characteristics.
- Then we can apply Dynamic Programming to improve its running-time.
- We will study these particular characteristics.

Outline of the lecture

- Overview of the Dynamic Programming
- Learning by example
  - The case of rod-cutting
  - Implementation of the rod-cutting problem
- Elements of the Dynamic Programming
- Other examples
  - Matrix-chain multiplication
  - Longest common subsequence

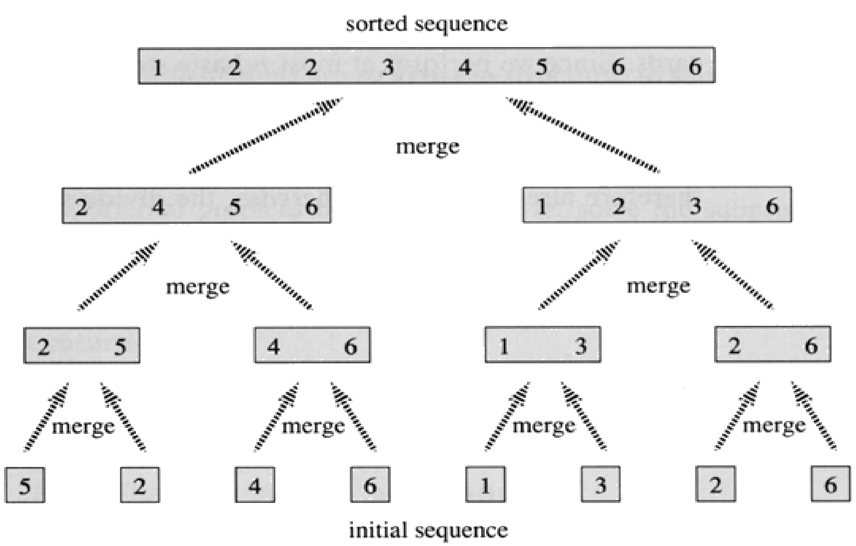
Overview: Optimization Problems

- Dynamic programming typically applies to optimization problems.
- What is an optimization problem?
  - Selection of a best element from a set of alternatives.
  - Example: The shortest path from dorm to class
  - Optimization problems can have many possible solutions.
    - First we make a set of choices for the shortest path:
      - Path A: 75 meters   Path B: 90 meters
      - Path C: 105 meters   Path D: 70 meters
    - Then we want to find the optimal(best) solution out of possible solutions.
      - In this case, the optimal (best) solution is Path D.

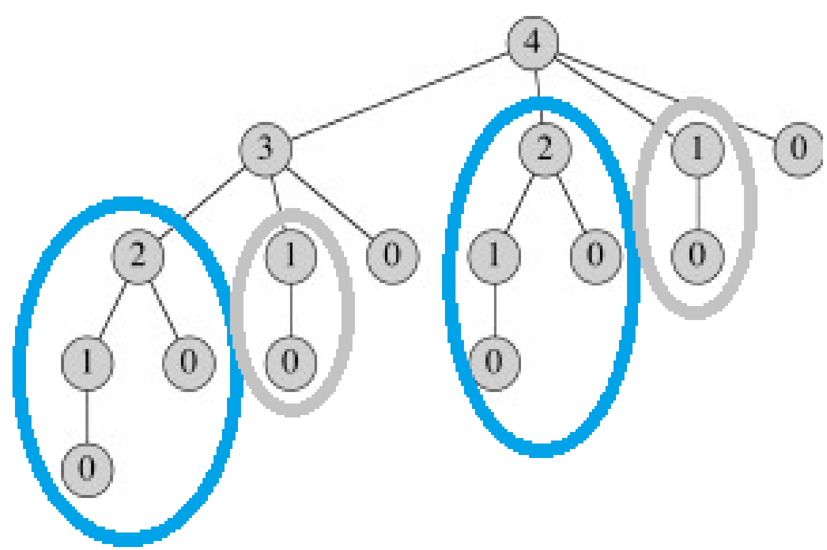
Overview: Dynamic Programming

- Dynamic programming, solves problems by combining solutions to subproblems.
- It is similar to divide-and-conquer method, but

Divide and conquer method applies when the subproblems are disjoint.

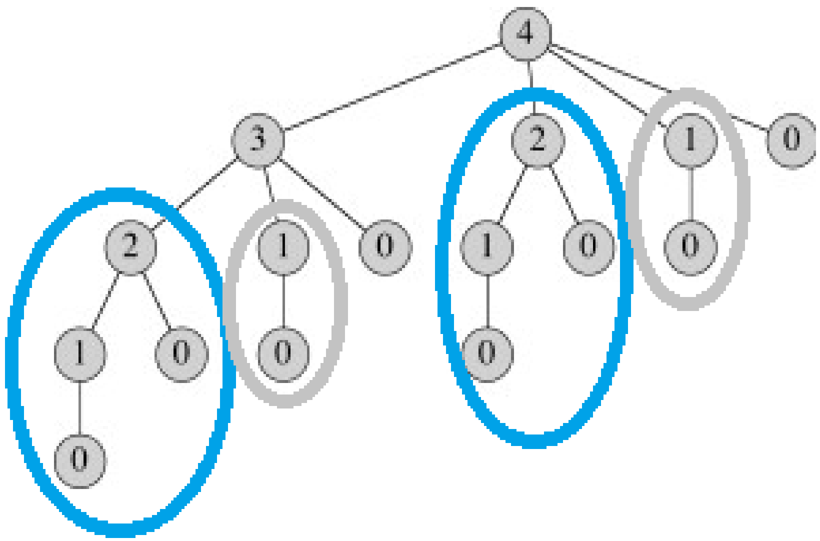


Dynamic programming method applies when the subproblems overlap. (subproblems share subsubproblems)



Overview: Dynamic Programming

- As we make each choice, subproblems of the same form often arise.
- Dynamic programming is effective when the same subproblem reappears more than once.



- <u>A dynamic programming method:</u>
  - solves each subproblem just once and
  - saves its answer in a table, and
  - it avoids the work of recomputing the answer every time it arises.
- The key technique is to store the solution to each subproblem, in case it should reappear.

Learning by example: Rod cutting

- **Problem:** A company buys long steel rods and cuts them into shorter rods, and then sell these shorter steel rods to their customers.
- Each cut is free.
- The price of the rod is not directly proportional to the length of the rod.

length i	1	2	3	4	5	6	7	8	9	10	inches
price p_i	1	5	8	9	10	17	17	20	24	30	dollars

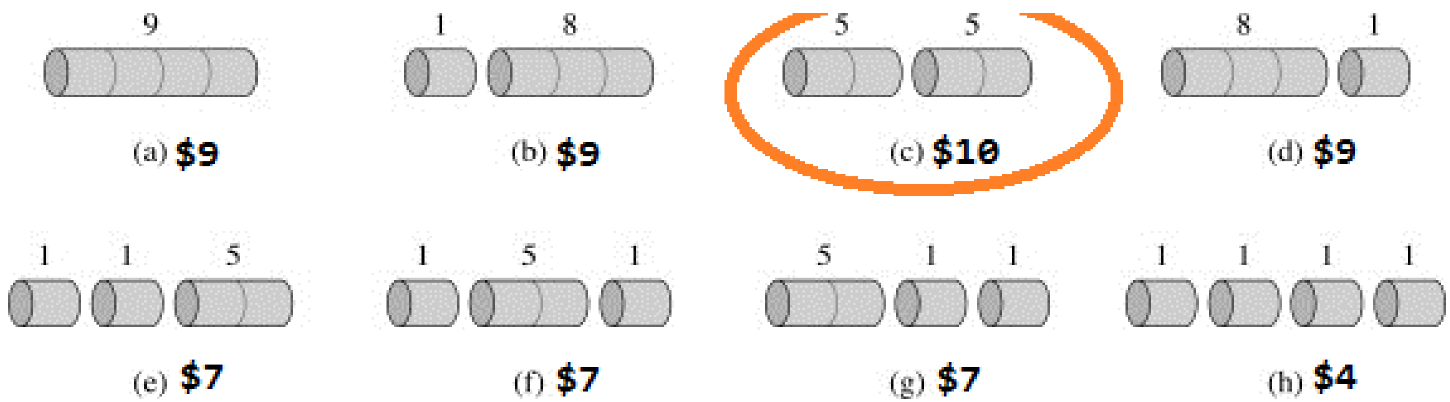
- **Problem:** Given a rod of length n and a table of prices,
  - how can we maximize the revenue?
  - what is the best(optimal) way of cutting up rod into shorter ones?

Learning by example: Rod cutting

- According to this price table:

length i	1	2	3	4	5	6	7	8	9	10	inches
price p_i	1	5	8	9	10	17	17	20	24	30	dollars

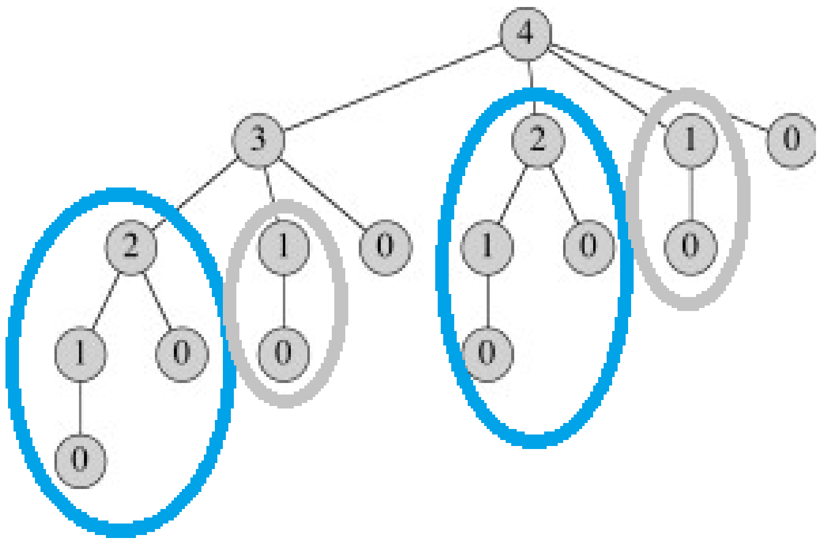
- Consider the case where the length of rod, n = 4 inches.
  - There are 2^{n-1} different ways to cut up a rod of length n.
  - If n=4 then $2^{4-1} = 2^3 = 8$ possible solutions



- And, the best(optimal) solution to maximize revenue is to cut rod into two pieces of 2 inches.

<u>Optimal Substructure</u>

- To solve the original problem of size n, we solve smaller problems of same type.



- This problem exhibits *<u>optimal substructure</u>*: The best solution to the problem can be constructed from best solutions to its subproblems.

Recursive solution to Rod-cutting problem

- Recursive (brute-force) solution to rod-cutting problem:

```
CUT-ROD(p, n)
    if n == 0
        return 0
    q = -∞
    for i = 1 to n
        q = max(q, p[i] + CUT-ROD(p, n - i))
    return q
```

- This Cut-Road is so inefficient, because
    - it calls itself recursively many times, with the same parameter values.
    - It solves the same subproblems repeatedly.
- The running time-of CutRod is exponential. $O(2^{n-1})$
    - $2^4 = 16$
    - $2^5 = 32$
    - ....
    - $2^{30} = 1,073,741,824$

Implementation of CutRod

<u>Apply Dynamic Programming to CutRod</u>

- Convert <u>*CutRod*</u> into an efficient algorithm using Dynamic Programming method.
- The dynamic-programming method works as follows:
  - We observed that the recursive function is inefficient, because it solves the same problems repeatedly.
  - We solve each subproblem only <u>**once**</u>, and save its solution in the memory. (e.g.in an array, or hashMap)
  - If we need this subproblem's solution again later, we can just look it up from memory, instead of recomputing it.
- Dynamic-programming uses additional memory to save computation-time. An example of ***time-memory trade-off***.
- **Savings:** An exponential-time solution may be transformed into a polynomial-time solution. If n = 30
  - Recursive function takes: $2^{n-1} = 2^{30} = 1,073,741,824$
  - Dynamic programming takes: $n(n+1)/2 = 30(30+1) = 930$

How to implement Dynamic-Programming

- There are two ways to implement a dynamic-programming approach:
  - 1-) Top-down with memoization:
    - Write the recursive function in a natural manner
    - Modify it to save the result of each subproblem(usually in a hashMap
    - The function now first checks whether it has previously solved this subproblem.
      - If solved, return the saved value, saving further computation-time
      - If not solved, compute the value in a usual manner.
  - Recursive procedure has been memoized; it "remembers" what results it has computed previously
  - 2-) Bottom-up method:

How to implement Dynamic-Programming

- There are two ways to implement a dynamic-programming approach:
  - 1-) Top-down with memoization:
  - 2-) Bottom-up method:
    - Solving any particular subproblem depends on solving "smaller" subproblems.
    - We sort the subproblems by size, and solve them in order, smallest first.
    - Again, we solve each subproblem only once, and save its solutions in memory. (e.g. array, hashMap, hashTable etc.)
    - When solving a subproblem, we have already solved all of the smaller subproblems its solution depends on. And we don't recompute it, we just look it up from the memory.

"Dynamic Programming"?

- The name of this design technique, "Dynamic Programming", is intimidating.

- But in reality it is a very simple technique.

- You only need to store the solutions in a "dynamic" table.

    The size of the "table"(array) may not be known in advance.

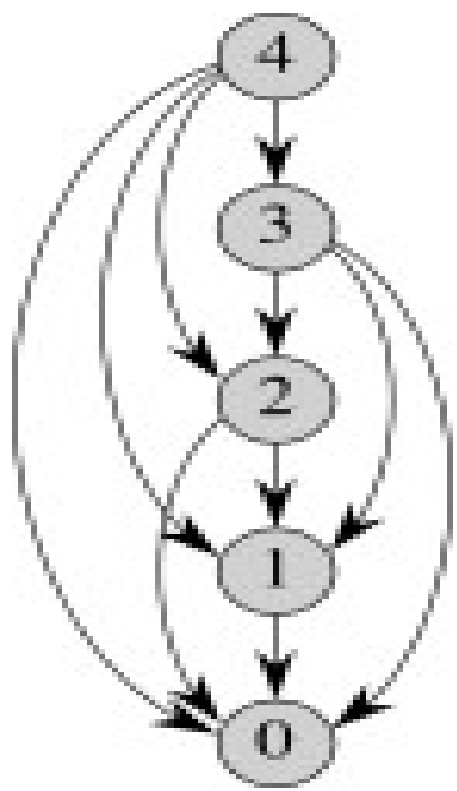
Which approach is better?

- Which approach is better:
  - 1-) Top-down with memoization:
  - 2-) Bottom-up method:

- Both of them have the same asymptotic running-time.
- But, in practice bottom-up method outperforms the top-down with memoization. Because;
  - Bottom-up method has no overhead for recursion.
  - And, less overhead for maintaining the table in memory.
- We usually use bottom-up method for real problems.

Subproblem graphs

- When we think about a dynamic programming problem
  - We should understand the set of subproblems involved, and
  - How subproblems depend on one another.
- We should be able to draw the subproblem graph for the problem which shows these information:



Reconstructing a solution

- Our dynamic programming solution to the rod-cutting problem returns the optimal solution
  - maximum revenue for a given length
- But it does not return an actual solution(a list of piece sizes). Such as, cut the whole rod into following two pieces
  - 2 inch
  - 3 inch.
- We can easily modify our bottom-up solution to return both
  - The maximum revenue, and
  - The list of piece sizes for max revenue.
- We only need an additional array to keep the piece sizes.

Elements of Dynamic Programming

- When we should look for a dynamic programming solution to a problem?

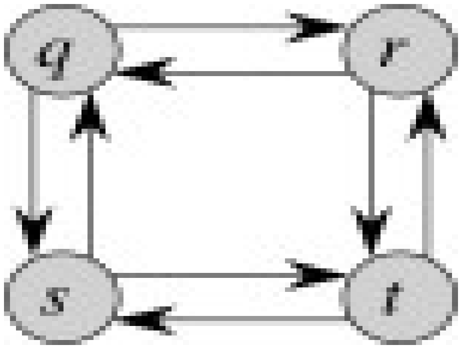
- This algorithm is so slow. Its running time is exponential. $O(2^n)$. I can not run this program for large n values.

- You can not apply Dynamic Programming for all problems with exponential running time.

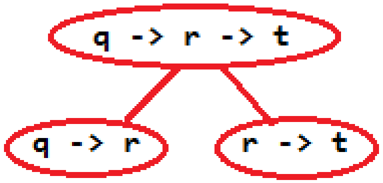
- In order to apply dynamic-programming, the optimization problem must have these two key characteristics:
  - **Optimal substructure**:
  - **Overlapping subproblems**:

Elements of Dynamic Programming

- Two key characteristics:
  - **1-) Optimal substructure**: A problem exhibits optimal substructure if an optimal solution to the problem contains within its optimal solutions to subproblems.

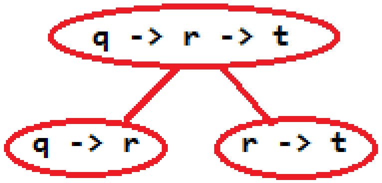


Shortest path: The shortest path from q to t is SP(q->r->t)



- SP(q->r->t) = SP(q->r) + SP(r->t)
  2 = 1 + 1

Longest path: The longest path from q to t is LP(q->r->t) (no cycles)



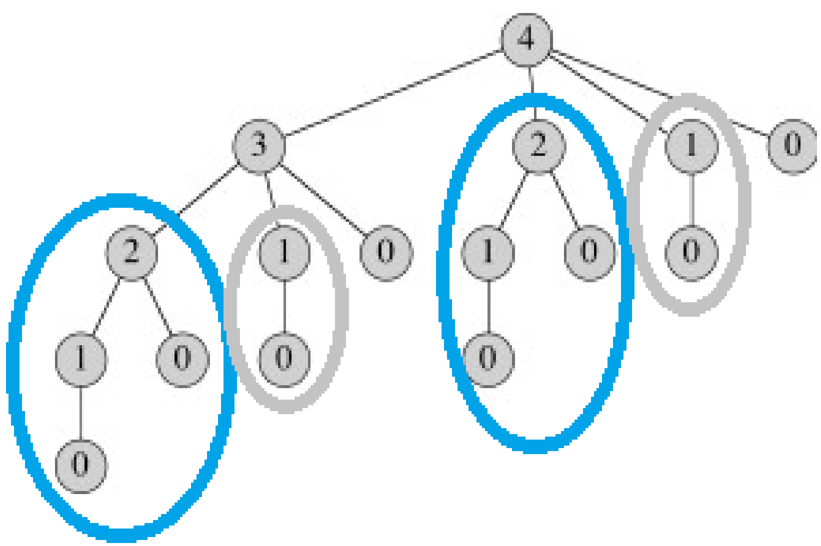
- LP(q->r->t) ≠ LP(q->r) + LP(r->t)
  2 ≠ 3 + 3

Elements of Dynamic Programming

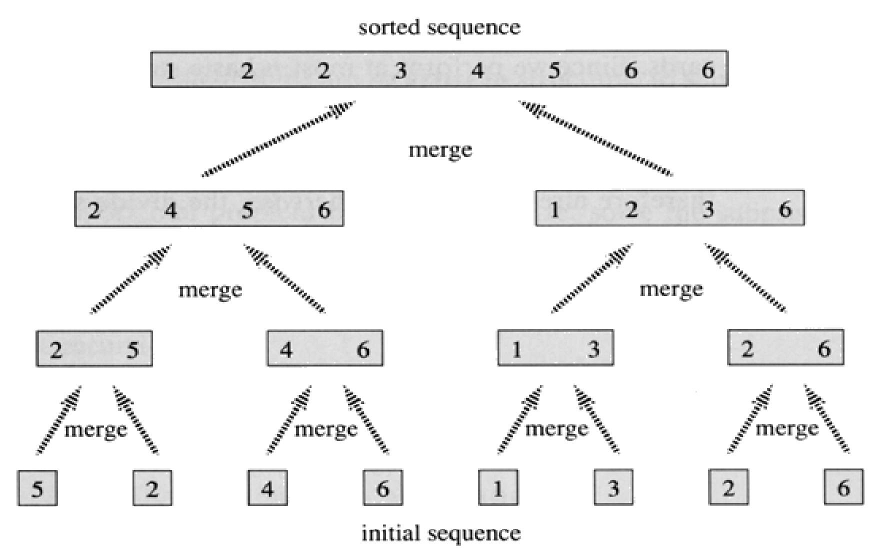
- Two key characteristics:
  - **1-) Optimal substructure**:
  - **2-) Overlapping subproblems**: When a recursive algorithm revisits the same subproblems repeatedly, we say that the optimization problem has overlapping subproblems.

Hint: Check out the smallest subproblem.

Rod-cutting problem:
Exhibits overlapping subproblems



MergeSort problem:
No overlapping subproblems



Outline of the lecture

- Overview of the Dynamic Programming
- Learning by example
  - The case of rod-cutting
  - Demo of the rod-cutting problem
- Elements of the Dynamic Programming
- Other examples:
  - Matrix-chain multiplication
  - Longest common subsequence

Example 2: Matrix-chain multiplication

- Let's refresh our memory about matrix multiplication:
- If we multiply a 2×3 matrix with a 3×1 matrix, the product matrix is 2×1

$$\begin{array}{ccccc} 2\times 3 & & 3\times 1 & & 2\times 1 \\ \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \end{bmatrix} & \times & \begin{bmatrix} t_{11} \\ t_{21} \\ t_{31} \end{bmatrix} & = & \begin{bmatrix} M_{11} \\ M_{21} \end{bmatrix} \end{array}$$

Here is how we get M_{11} and M_{22} in the product.

$$\mathbf{M_{11} = r_{11} \times t_{11} + r_{12} \times t_{21} + r_{13} \times t_{31}}$$

$$\mathbf{M_{12} = r_{21} \times t_{11} + r_{22} \times t_{21} + r_{23} \times t_{31}}$$

We are interested in the number of scalar multiplications.

In this case, the number of scalar multiplications is 2 x3x1 = 6.

Image:http://www.mathwarehouse.com/algebra/matrix/multiply-matrix.php

Example 2: Matrix-chain multiplication

- Let's assume we have 3 matrices:
  - A with dimension 10 x 100
  - B with dimension 100 x 5
  - C with dimension 5 x 50
- We have two options for the matrix chain multiplication.
  - (A x B) x C = (10 . 100 . 5) + (10 . 5 . 50) = 7500 scalar multiplications.
  - A x (B x C) = (10 . 100 . 50) (100 . 5 . 50) = 75000 scalar multiplication
- Both options have the same result, but the number of scalar multiplications are different. The first approach is 10 times faster than the second one.
- --
- *Problem definition:* Given a chain of n matrices, fully paranthesize the product $A_1 A_2 .....A_n$ in a way that minimizes the number of scalar multiplications.
- In this problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices.

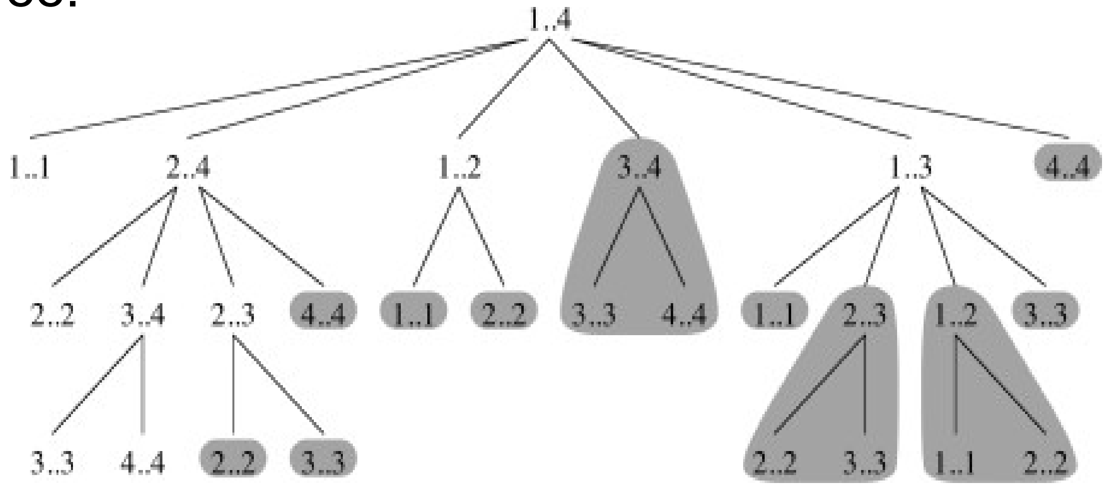
Example 2: Matrix-chain multiplication

- Applying dynamic-programming:
  - Optimal substructure: Does the best solution to the problem contains within the best solutions to subproblems?
    - If the best solution to the matrix-chain multiplication of A, B, C, and D is
      - ((A . B) . C) . D
    - Then the best solution to the subproblem A, B, and C must be

      (A . B) . C *it can not be A . (B . C)*

  (!!!!You can not find optimal substructure in solutions other than the optimal one.)

  - Overlapping subproblems: Write a recursive procedure to check each way of parenthesizing the product. And, observe that, a recursive algorithm solves the same subproblems in different branches of the recursion tree.



Example 3: Longest Common Subsequence (LCS)

- Biological applications often need to compare DNA of two different organisms, and try to determine how "similar" they are.
  - S1 = ACCGGTCGAGTGCGCCGGAAGCCGGCCGA
  - S2 = GTCGTTCGGAATGCCTTGCCGTTGCTCTGTA
- One way to find similariy is to find common subsequence between two strings.
- The difference between substring and subsequence:
  - Substrings are consecutive parts of a string. But subsequences need not to be consecutive.
  - Longest common substring of
    - ABCBDAB and BDCABA is

      AB
  - Longest common subsequence of
    - ABCBDAB and BDCABA is

      BCBA

Longest Common Subsequence(LCS)

- Problem: Given two sequences X = (x_1, x_2, ... x_m) and Y = (y_1, y_2, ... y_n), find the maximum-length common subsequence of X and Y.
- It can be solved using a recursive algorithm. But it is not better than brute-force which results in exponential running-time.
- Applying dynamic-programming with bottom-up approach: Fill the table with the following recurrence function.

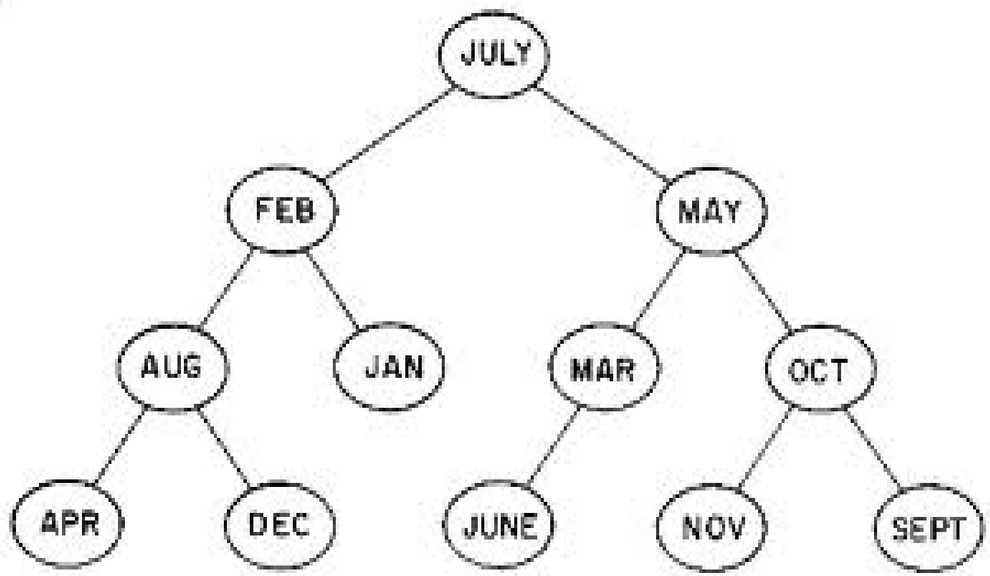
$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

	j	0	1	2	3	4	5	6
i		y_j	*B*	*D*	*C*	*A*	*B*	*A*
0	x_i	0	0	0	0	0	0	0
1	*A*	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	*B*	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	*C*	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	*B*	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	*D*	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	*A*	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	*B*	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

- Optimal substructure: Does the best solution to the problem contains within the best solutions to subproblems?
- Overlapping subproblems: If we write recursive function, we can observe it.

Example 4: Optimal binary search trees

- Suppose that, we are designing a program to translate text from English to French. For each occurence of English word, we need to lookup its French equivalent.
- We could perform these lookup operations by building a binary search tree with n English words as keys, and their French equivalent as satellite data.



- We could ensure an O(lg n) search time for each word, using a balanced binary tree.

Image: http://serghei.net

Example 4: Optimal binary search trees

- Words appear with different frequencies. A frequently used word such as "the", "and" may appear far from the root, while a rarely used word such as "machicolation" appears near the root.

- Such an organization would slow down the total time spent fo translation, since the number of nodes visited depends on the depth of the node containing the key.

- We want frequent words to be placed near the root, rare words to be placed far from the root.

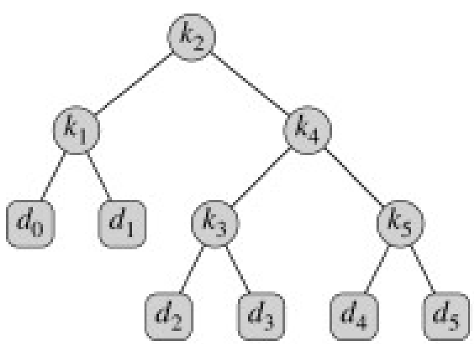
Example 4: Optimal binary search trees

- Optimization Problem: Given that we know how often each word occurs, how do we organize a binary search tree to minimize the number of nodes visited in all searches?

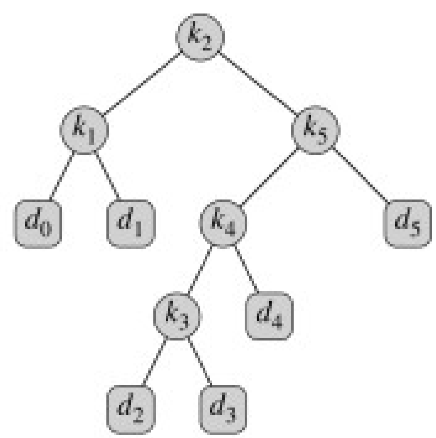
i	0	1	2	3	4	5
Pi		0.15	0.10	0.05	0.10	0.20
Qi	0.05	0.10	0.05	0.05	0.05	0.10

- Minimize the expected cost

ExpectedCost = 1+ ∑depth(ki) * pi + ∑depth(di) * qi



Binary Search Tree
(a)



Optimal Binary Search Tree
(b)

Take-home message

- If you will remember just one thing from this lecture. Here it is:
  - If there are overlapping subproblems, consider applying dynamic programming.