SoC 2022 : Competitive Coding

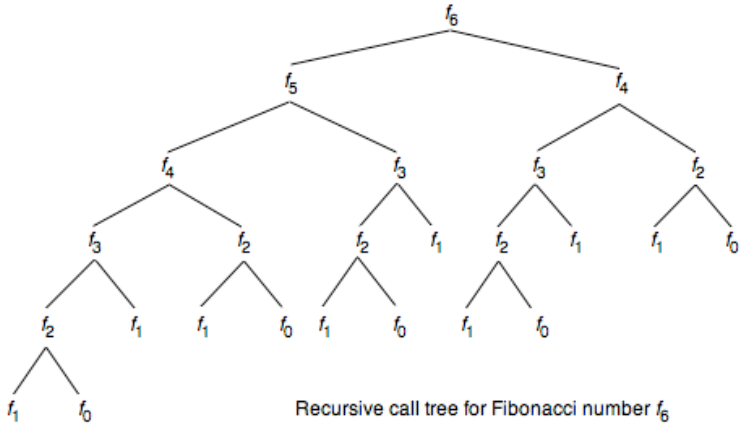
Week-5

Dynamic Programming

Contents

Memoization and Tabulation

- These are different ways to store the values so that the solutions of subproblems can be reused.



Recursive call tree for Fibonacci number f_6

- https://www.cse.unsw.edu.au/~billw/dictionaries/prolog/memoisation.html
- Can see same thing being recomputed many times, i.e., solutions to subproblems **not** being reused. Time complexity: T(n)=T(n-1)+T(n-2) ⇒ 2*T(n-2)≤T(n)≤2*T(n-1) ⇒ $2^{\frac{n}{2}} \leq T(n) \leq 2^n$.

- **Memoization ("Top-down") :**
  Use a memo (array/vector/**map**) to store calculated values.
  We write the procedure **recursively** in a natural manner, but modified to save the result of each subproblem. [Top-down, in the sense that fib(5) calls fib(4), etc.].
  The procedure now first checks to see whether it has previously solved this subproblem. If so, it returns the saved value, saving further computation at this level; if not, the procedure computes the value in the usual manner.

- **Tabulation ("Bottom-up") :**
  Use a table (array/vector) to store values.
  We start by solving the subproblems, smallest first. [Bottom-up, in the sense that we go from the base cases to the question in hand].
  When solving a particular subproblem, we have **already solved all** [in contrast to the recursive approach] of the smaller subproblems its solution depends upon, and we have saved their solutions.
  We solve each subproblem only once, and when we first see it, we have already solved all of its prerequisite subproblems.

  Example: Fibonacci sequence (recursion tree above).
  See memo-table.cpp. Time complexity: O(n). Additional space: O(n), but not really required in the bottom-up approach [since at any time, just the previous two values are required].

  Bottom-up approach usually has less overhead, since it doesn't involve making recursive calls. We will normally use bottom-up (tabular). Can also use map/unordered_map for memoization.

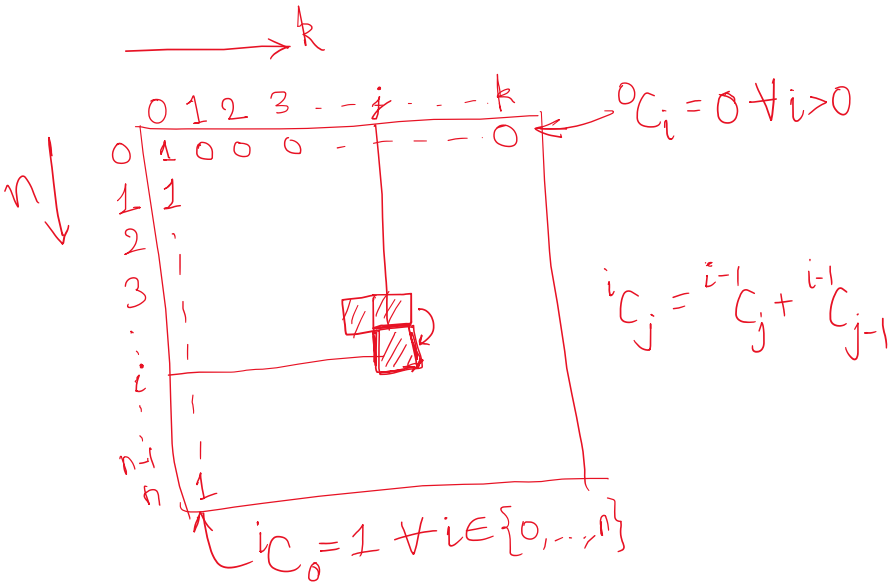
Dynamic Programming

1. Fibonacci Sequence

Consider the case of Fibonacci sequence. Every recursive call is for an element in the set $\{Fib(0), \dots, Fib(n)\}$. This set has size O(n), while the number of recursive calls made by the naïve algorithm is O(2^n). This implies that each $Fib(i)$ is called multiple times.
We optimized by not doing these repetitive computations, by remembering what we computed.

2. Binomial Coefficients

First, we need to think of a recursive relation: $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$ and the base cases. This forms the basis, and then we optimize. Can see that initial workflow is same as in recursive implementation.
Again, note that the only calls made are in the set $\{\binom{0}{0}, \dots, \binom{n}{k}\}$. This set has size O(n*k) [far less than the exponential number of calls that we would have made otherwise].
Tabular approach: use a table dp[n+1][k+1].



Note that we first filled up the base cases. Then, can see at (i,j), on how the DP table will fill up. This is an iterative procedure.
Lastly, we return dp[n][k].

See code: binomial.cpp.
Table for 6C3:

```
6 3
1       0       0       0
1       1       0       0
1       2       1       0
1       3       3       1
1       4       6       4
1       5       10      10
1       6       15      20
20
```

Note that in the process we have computed ${}^{i}C_{j}$ for all i∈{0,...,n} and for all j∈{0,...,k}.

DP is generally used for finding the optimal solution(s) [e.g., minimize some cost, or maximize profit, etc.].
Two properties of a problem that can suggest DP:

1. **Optimal substructure**: A given problem has an optimal substructure if the optimal solution of this problem can be obtained by using optimal solutions of its subproblems. [Will see this shortly.]
2. **Overlapping Problems**: Saw this property above. Fibonacci: the only subproblems were in a set of size O(n). Similar for binomial coefficients problem.
   Overlapping in the sense that the same subproblems arise multiple times. For example, fib(4) is a subproblem both for fib(5) and fib(6).
   To optimize for this, we use memoization/tabulation.
   In contrast, problems like binary search do not have overlapping subproblems [searching in one half, or in the other half, do not have anything in common].

3. Subset Sum

Q. Given a set S of non-negative integers $a_1, \dots, a_n$, and a target value B, determine if there is a subset of the given numbers with sum equal to B. [Each number can be taken at most once.]

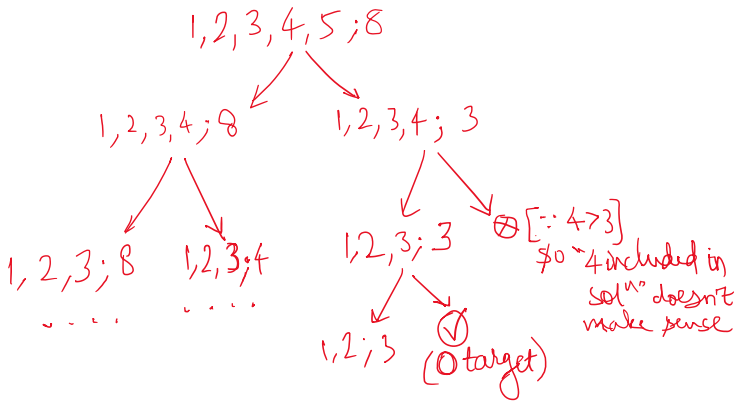
Naïve algo: Iterate over all subsets of S, and check if numbers in some subset add up to B.
Time complexity: O(2^n * some polynomial of n) [2^n subsets, and need to traverse each].

To apply DP, we need to think about subproblems. Need to formulate a recurrence, and base cases.

Can think as follows:
In a given solution "Sol" (i.e., a set whose elements add up to B), there can be 2 cases: a_n is present in the solution, or a_n isn't present in the solution.

1. a_n is present. Then, the same solution "Sol" will work for a target value of $B - a_n$, the set of numbers now being $a_1, \dots, a_{n-1}$.
2. a_n is not present in "Sol". Then, the same solution will work for target B, and set $a_1, \dots, a_{n-1}$ [as a_n contributes nothing to the sum].

   Example:



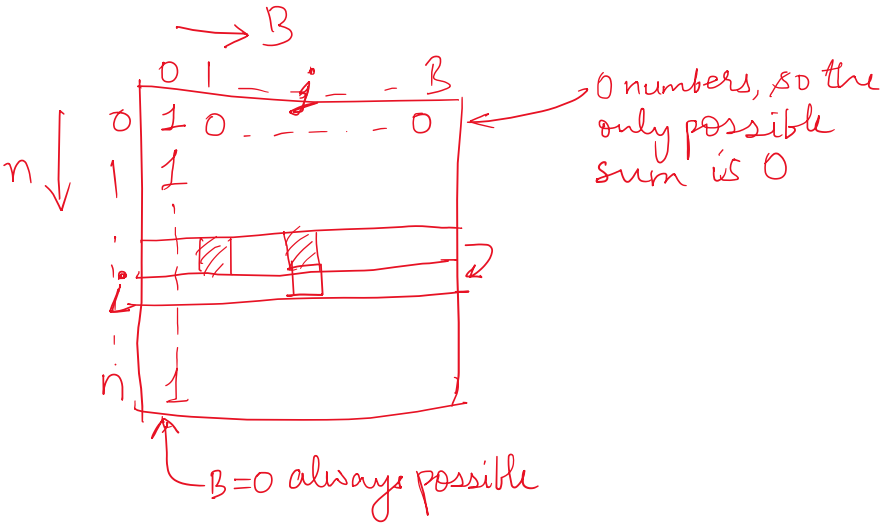
Based on the above observation, the recurrence is:

$$dp(n, B) = dp(n-1, B-a_n) \;||\; dp(n-1, B)$$
$$\boldsymbol{dp(i, j) = dp(i-1, j-a_i) \;||\; dp(i-1, j)}$$

Here i is the current number of elements we are considering, and j is the target value. Note that $B - a_n$ $(j - a_i)$ must be non-negative [otherwise, of course no solution exists, since a_i's are all non-negative].

Naïve recursive implementation: from the recursion tree, can see exponential [or, can think as T(n)=2*T(n-1)+O(1)].
But, there are many overlapping/repetitive subproblems: the only values that n and target can take are in the sets {0,…,n} and {0,…,B}, giving at most O(n*B) subproblems.
Can see the recursive substructure here: solution for (i,j) being formed from smaller instances.

Using tabulation, we get a time complexity of O(n*B).



Time and space complexity: O(n*B).

Note that in the nested for loops (i,j), at any instant, we only need the (i-1)th row for dp[i][j]. So, can do with O(B) space only [just "2 rows"].

4. 0-1 Knapsack problem

Q. Given weights and prices of n items ($w_1, \ldots, w_n$ and $p_1, \ldots, p_n$ (all non-negative)), put these items in a knapsack (bag) of capacity W to get the maximum [i.e., optimal] total value in the knapsack. [0-1, meaning each item can be taken at most once (0 or 1 times).]

As before, to get a recurrence, can think to include/not include the last item in the solution. Given an **optimal** solution (i.e., sum of prices of items is maximum, given sum of weights ≤ W)

1. n-th item is included in this optimal solution, then the same solution is optimal for bag of capacity $W - w_n$, and items 1,…,n-1.
2. n-th item not included in optimal solution, then same solution is optimal for bag with capacity W and items 1,…,n-1.

Recurrence:

$$dp(n, W) = \max\{dp(n-1, W-w_n) + p_n, dp(n-1, W)\}$$

Where 1st term corresponds to case-1, and 2nd to case-2. "max" has been used, since we need to maximize the profit.

In terms of i items and capacity j, we have $dp(i, j)$ = maximum profit that can be achieved using the first i items, and a bag of capacity j.

$$dp(i, j) = \max\{dp(i-1, j-w_i) + p_i, dp(i-1, j)\}$$

Note that the first term will be there only if $j - w_i \geq 0$.

Again, we use tabulation: just create a DP matrix, fill in base cases, and use nested for loops to fill the matrix. Inside the for loops, just need to use the above recurrence.

See knapsack.cpp [space optimized (O(W)) version too].

In this process, we have solved subproblems for all i∈{0,…,n} and for all j∈{0,…,W}.

5. Unbounded Knapsack

Again, there are n items [0,…,n-1]. Any item can be taken any number of times. Capacity of bag is W.

Define the dp state dp[i] to be the maximum price that can be achieved using a bag of capacity i [removed earlier i (no. of items) from the dp state, since the 0-1 restriction is gone].

Recurrence:

$$dp(i) = \max_{j\in\{0,\ldots,n-1\}} \left(dp(i), dp\left(i - w_j\right) + p_j \; if \; i - w_j \geq 0\right)$$

max is taken over all j.

[Note that any item can be taken any number of times, so need to consider each item every time; can't do it like the previous problem.]

6. Longest Increasing Subsequence

Given an array of size n, find the length of the longest subsequence such that all elements of the subsequence are sorted in increasing order. For example, the length of LIS for {10, 22, 9, 33, 21, 50, 41, 60, 80} is 6 [{10, 22, 33, 50, 60, 80}].

O(2^n * poly(n)) solution is to consider every subsequence.

Let dp[i] be the longest increasing subsequence that can be achieved till index i.
Initially, all dp[i]=1 (the number a[i] itself forms an "increasing subsequence".)
Then,

$$dp[i] = \max_{0\leq j<i \text{ and } a[j]\leq a[i]} dp[j] + 1$$
$$(= 1 \; if \; no \; such \; j \; exists)$$

[If "strictly increasing" is required, then change to a[j]<a[i].]

Example: array elements are 10, 22, 9, 33, 21, 50, 41, 60, 80. Consider 41 (i=6). We iterate from 0 to 6, taking max of dp[] over indices that have element ≤ 41, i.e., j=0,1,2,3,4. Max of dp[j] over these j is 3. So, dp[6]=3+1=4 [the subsequence is {10,22,33,41}].

Time complexity: $O(n^2)$; Auxiliary space: O(n) [dp array].
Code: LIS.cpp.

Similarly, we have Longest Decreasing Subsequence.

$$dp[i] = \max_{0 \le j < i \text{ and } a[j] \ge a[i]} dp[j] + 1$$
$$(= 1 \; if \; no \; such \; j \; exists)$$

[Difference from LIS recurrence marked in red.]

7. Largest Sum Contiguous Subarray and Submatrix

Given an array of size n, find the sum of the **contiguous** subarray that has the largest sum.
Example: array is {-2, -3, 4, -1, -2, 1, 5, -3}, then answer is 4+(-1)+(-2)+1+5 = 7.

One way is to check all contiguous subarrays, computing sums for each.

Can use DP as follows: [note that first step is to get a recurrence]
Let dp[i] be the largest sum till index i, **and the subarray ending at i**. So, if ((sum obtained till i-1) + a[i]) is more than a[i], then dp[i] = this sum, else dp[i]=a[i].

$$dp[i] = \max\{dp[i-1] + a[i], a[i]\}$$

Time: O(n).
Can do it with O(1) space, since only the just previous dp value is being used.

Variation of the above problem: Maximum sum rectangle in a 2D matrix.

1	2	-1	-4	-20
-8	-3	4	2	1
3	8	10	1	3
-4	-1	1	7	-6

Method is: iterate over columns, keeping two indices (say, i and j). Compute the sum of all columns from i to j to get a 1D array. Apply above problem to this 1D array to get the max sum contiguous subarray.

Let matrix be $n \times m$.

```
Curr_sum = 0                    // max sum submatrix has size 0 initially
For i = 0 to m-1:
        Initialize temp_1D_array of size n with all 0s.
        For j = i to m-1:
                Add j^th column to temp_1D_array.
```

```
Apply above algo to get max sum cont. subarray. Returns max sum.
Temp_sum = sum returned on previous line    // answer this time
If Temp_ans > Curr_ans:
        Curr_ans = Temp_ans
```

In the above image, we'll get the max answer when i=1, j=3. The 1D array we'll get is: -3,3,19,7. The earlier algo returns 29.
Time complexity: O(m^2*n).

8. Longest Common Subsequence

Aka LCS.
Given 2 arrays of sizes n and m, find the longest subsequence [not necessarily contiguous] that both have in common.

If dp[i][j] is the optimal answer till indices i and j in the 2 arrays [i.e., the length of the longest common subsequence in a[0 to i] and b[0 to j]], then:
Again, think by the last numbers:
1,2,3,4,5
2,3,5,7

1. a[i]==b[j], then dp[i][j]=1+dp[i-1][j-1]
2. a[i]≠b[j], then dp[i][j] = max(dp[i-1][j], dp[i][j-1])

9. Matrix Parenthesization Problem

Define the cost of multiplying 2 (compatible) matrices A ($m \times n$) and B ($n \times p$) be m.n.p. Cost can be thought of as the number of operations, etc.

Given multiplication-compatible matrices $A_1, \dots, A_n$, we need to find a parenthesization such that cost is minimized.
Example: A_1, A_2, A_3 with sizes $10 \times 100, 100 \times 5, 5 \times 50$.
1. $(A_1A_2)A_3$ – cost is 5000+2500 = 7500
2. $A_1(A_2A_3)$ – cost is 25000+50000 = 75000

Clearly, 1 is better.

Optimal Substructure: consider $(A_1 \dots A_m)(A_{m+1} \dots A_n)$, then the answer depends on the optimal answers of the 2 subproblems.
Overlapping Subproblems:

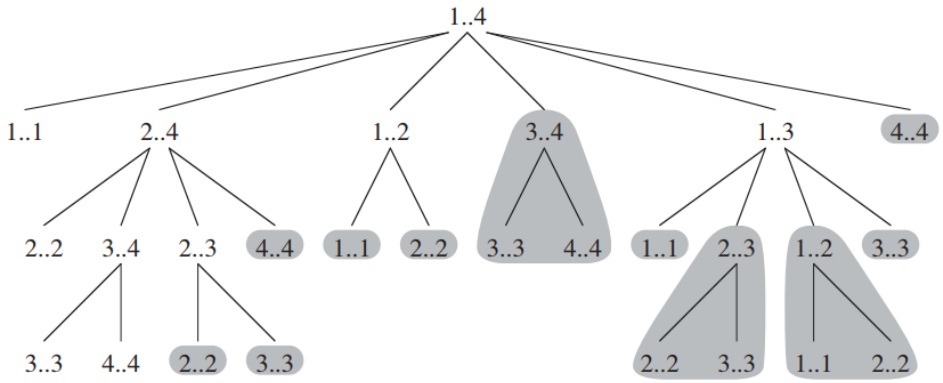


Figure 15.7 The recursion tree for the computation of RECURSIVE-MATRIX-CHAIN(p, 1, 4). Each node contains the parameters i and j. The computations performed in a shaded subtree are replaced by a single table lookup in MEMOIZED-MATRIX-CHAIN.

(i..j) is the problem to parenthesize $A_i \dots A_j$. Can see repetitive computations [e.g., 3..4, 2..3, 1..2].

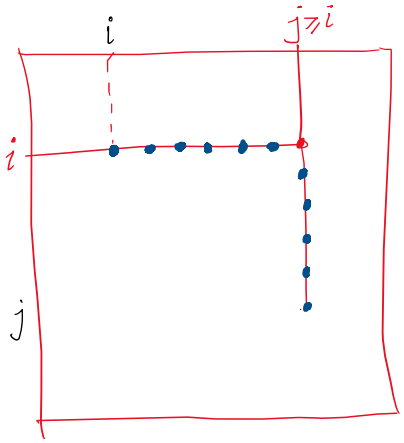
To get a recurrence:
We observed above that if the product $A_i \dots A_j$ is split between k and $k+1$, then optimal parenthesizations for $A_i \dots A_k$ and $A_{k+1} \dots A_j$ are required.
How to choose this k? Iterate from i to j-1.
Define dp[i][j] to be the minimum cost to multiply $A_i \dots A_j$.
[We will obtain the "actual parenthesization" from this "minimizing cost" process itself.]
Base cases: dp[i][j]=0 if i==j.
Let A_t have dimensions $p[t] \times p[t+1]$.

$$dp[i][j] = \min_{k=i \text{ to } j-1} dp[i][k] + dp[k+1][j] + p[i]p[k+1]p[j+1]$$

Where the last term is the cost to multiply the matrix $[A_i \dots A_k]$ with the matrix $[A_{k+1} \dots A_j]$.
[Note that we first try to get a recurrence relation.]

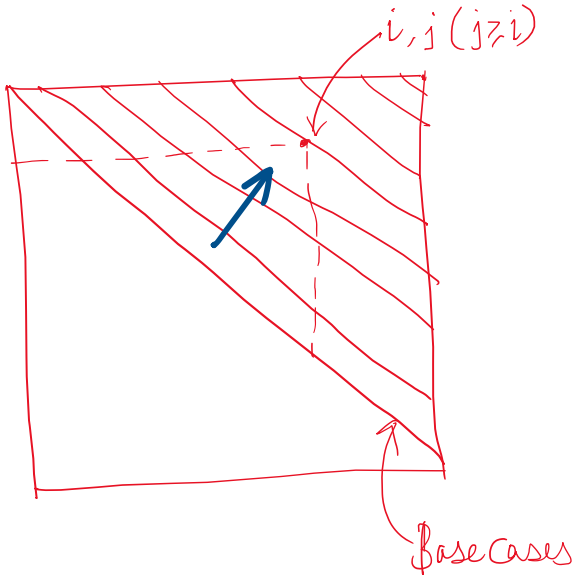
Implementation:
Note from the recurrence:

- We use dp[i][j] with j≥i only.
- For dp[i][j], we need dp[something > i][j] (2nd term). So, filling it row-wise (as before) won't work.
- For dp[i][j], we need dp[i][i],...,dp[i][j-1], and dp[i+1][j],...,dp[j][j].
  Marking these in the dp matrix:



Moreover, our base cases are: dp[i][i]=0 ∀ i∈{1,...,n}.

So, we fill the matrix diagonally.



Can see from the 2 diagrams that all required dp[][] values will be computed by the time we reach dp[i][j].

Moving along these lines is same as saying (j-i = constant). Also note that j-i+1 is nothing but the number of matrices in $A_i \dots A_j$. So, same length segments' dp values getting filled in the same iteration.

Thus, the loops will be like:

```
For len = 2 to n:                    // len is length (number of matrices in A_i ... A_j)
                                     // based on above observation
        For i = 1 to n-len+1:
                j = i+len-1
                dp[i][j] = ...
```

Answer: dp[1][n] (considered 1-indexing here).

The above code minimizes the cost.
To get the parenthesization itself, we store the value of k that minimized the dp[i][j] RHS, as this k marks the division of $A_i \dots A_j$ into (...)(...).

Code: matrix-paren.cpp.

10. Palindrome Partitioning Problem

Q. Given a string s, determine the minimum number of partitions required to get every substring to be a palindrome.

Examples:
abaccdc → aba|c|cdc (2 partitioning pipes required)
abcde → a|b|c|d|e (4)

Can see optimal substructure just as before: If s → s1|s2, then need optimal answers for s1 and s2.
Note the similarity to the previous problem: $A_i \dots A_j \rightarrow A_i \dots A_k | A_{k+1} \dots A_j$ and here $s \rightarrow s_1 | s_2$. In both problems, need to iterate over possible partition positions [minimizing some "cost" in both].

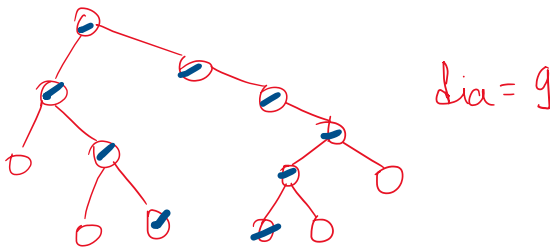
Can use similar recurrence as above. Filling of dp matrix needs to be done in a similar way (along the leading diagonal's direction, length = 2 to n, etc.).

DP on Trees

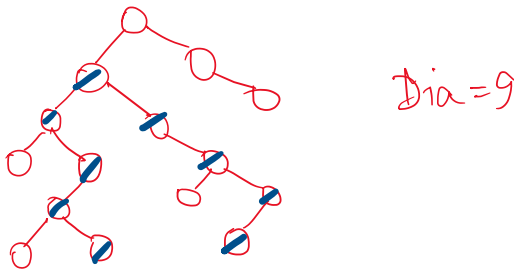
Recall that a tree is a connected graph, that has no cycles. A tree with n nodes has n-1 edges.
Useful to "root" the tree and recurse on its subtrees.

1. Diameter of Tree

Given a rooted tree with n nodes, calculate the longest path [in terms of no. of nodes] between any two nodes (also known as diameter).

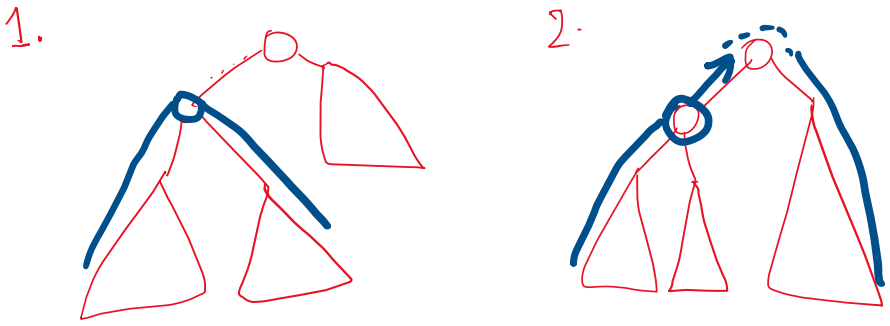


Doesn't necessarily pass through the root:



[There can of course be > 2 children.]

First, need to think of a recurrence.
For any node, there are 2 possibilities [note that we are considering the subtree rooted at this node] :



Case 1: It doesn't "pass on" the diameter to its parent.
Case 2: It passes on the diameter to its parent [doesn't act like a root itself].

We maintain 2 DP arrays for this. Call them dp1 and dp2. dp1[i] is the max length achievable from case-1 (i.e., case-1 *at node i*). dp2[i] is the max length from case-2 at i.

$$dp_1[i]\ (diameter\ passes\ through\ this, and\ doesn't\ pass\ through\ its\ parent)$$
$$= 1 + \left(2\ greatest\ values\ in\ the\ set\ \{dp_2[j]\}_{j\ is\ a\ child\ of\ i}\right)$$

Note that we're using dp2 in the second term, since we want the children of i to pass their values to their parent, which is i.

$$dp_2[i] = 1 + \left(greatest\ value\ in\ the\ set\ \{dp_2[j]\}_{j\ is\ a\ child\ of\ i}\right)$$

Using these recurrences, we can fill up the dp arrays. [Base cases: leaves have dp values = 1.]
The final answer will be *max_element(dp1,dp1+n).

Observe that we need DP values for all children, before processing the parent. This tree traversal of "processing all children, and then the root" is called a "post-order traversal" [post meaning that current node is processed *after* its children].
[We also have pre-order traversal, and in-order traversal. Applications: Polish notation, and Expression-evaluation, respectively.]
We can simply use a **recursive** approach like DFS for any of these traversals: for post, we process the node after the recursive calls to its children, etc.

Code: tree-dia.cpp. Time and space complexity: O(n).

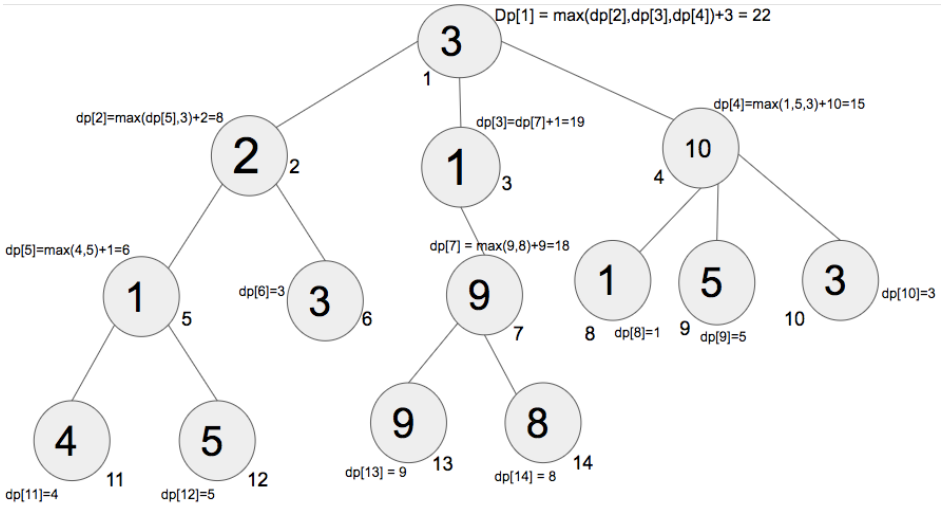
2. Maximum Sum without revisiting

Given a rooted tree with n nodes, calculate the maximum sum of the node values from root to any of the nodes without revisiting any node.

Define dp[i] to be the maximum sum that can be obtained from the subtree rooted at i. In the diagram below, our final answer will be dp[1], since our tree is rooted at 1.
Following the above point, we have:

$$dp[i] = val[i] + \max_{\text{j is a child of i}}(dp[j])$$

Can put 0 too, in the max function, in case of negative values.

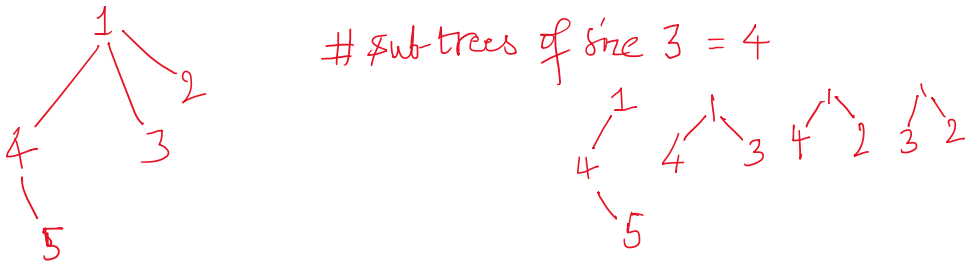


https://www.geeksforgeeks.org/dynamic-programming-trees-set-1/

3. Size K sub-trees

Given a tree T of n nodes and an integer K, find number of different sub-trees of size less than or equal to K.

Here a "sub-tree" of size k is defined as any set of k nodes, that are connected.
Different from the rooted-trees' subtrees we've been discussing till now. But as before, we will try to get a recurrence based on the earlier concept of rooted trees.



Consider the tree to be rooted at some vertex.
To build a recurrence, consider a vertex v. Its children nodes are $v_1, \dots, v_m$.

Define dp[v][k] to be the number of sub-trees of size k that can be obtained by necessarily "including v", and using any of the nodes "below" it [i.e., any of its descendents].
$v_1, \dots, v_m$ are the children of v.
Then,

$$dp[v][k] = \sum_{all\ sets\ \{a_i\}\ that\ add\ up\ to\ (k-1)} dp[v_1][a_1] * dp[v_2][a_2] * \dots * dp[v_m][a_m]$$

Example:
Want to find dp[1][3] in the above diagram:
Since we're including "1" necessarily, so the rest of the nodes in such sub-trees must be 3-1=2 (hence the k-1). This can be obtained by

1. 2 from the subtree rooted at 4, 0 from others [i.e., $a_1 = 2, a_2 = 0, a_3 = 0$]
2. 1 from the subtree rooted at 4, 1 from that at 3 [i.e., $a_1 = 1, a_2 = 1, a_3 = 0$]
   And so on.

The summation represents these cases.

But iterating over such subsets will not have a good time complexity [no. of non-negative integral solutions to $a_1 + \dots + a_m = k - 1$ is ~ (k+m-2)C(m-1)].

However, we can see that the possible "calls" are far less:
For any node v, the only calls can be dp[v][0]…dp[v][k].
Thus, we need to fill up a DP matrix of size $n \times (k + 1)$.

To fill this optimally, note that $dp[v_1][a_1] * dp[v_2][a_2] * \dots * dp[v_m][a_m] = (\dots) * dp[v_m][a_m]$ over the entire summation is $dp[v][k - a_m] * dp[v_m][a_m]$. That is, "v" included in the subset with count $k - a_m$, and then v_m contributes a_m nodes.

Rewriting this:

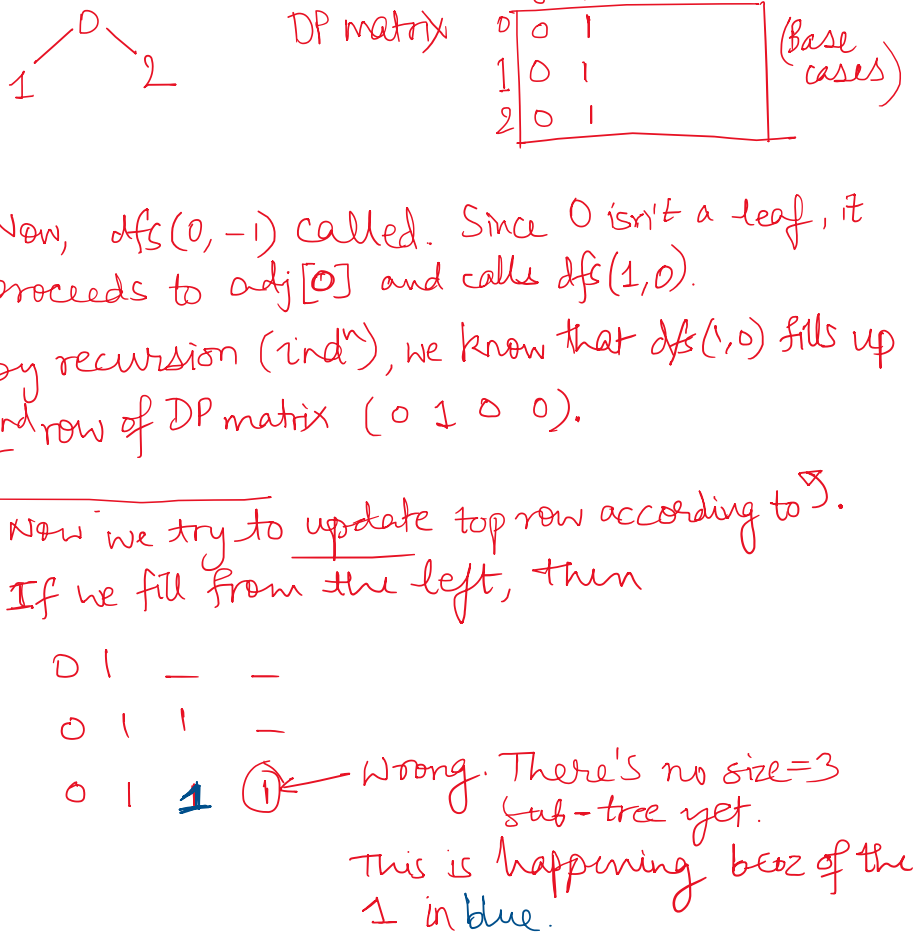
$$dp[v][i + j] = dp'[v][i] * dp[v_m][j]$$

Where v_m has not been considered in $dp'[v][i]$.

Where v_m is some child, contributing j nodes. As for the summation, it builds up as we keep on updating dp[v] row for each of its children.

See code: k-sub-tree.cpp. Time complexity: $O(n*K^2)$.

There are some subtleties involved in filling up the DP matrix:



To remedy this, we just fill this row from the back.

```
For k1=k to 0:
        For k2=k1 to 0:
                dp[v][k1]+=dp[v][k1-k2]*dp[i][k2]    // i is the child who has just been
                                                     // recursed
                                                     // [by dfs(i,v)]
```

(This ensures the dp' thing.)

Problems

First 6 problems from CSES problemset (https://cses.fi/problemset/list/).