Dynamic Programming Algorithm-design Technique.

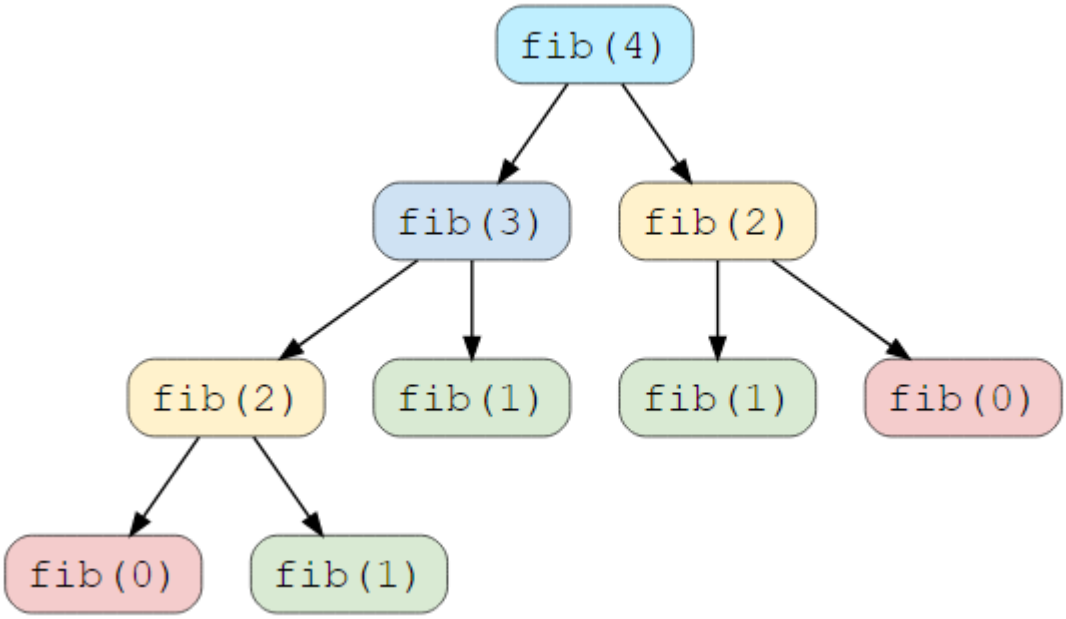
- Similar to the divide-and-conquer technique in that it can be applied to a wide variety of different problems.
- There are few algorithmic techniques that can take problems that seem to require exponential time and produce polynomial-time algorithms to solve them. Dynamic programming is one such technique.
- Algorithms that result from applications of the dynamic programming technique are usually quite simple

Definition

- Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.
- Optimization problems
- Often these subproblems overlap in some way

Fibonacci Sequence

- Defined as: The n^{th} Fibonacci value is a piecewise recursive function :
  $$fib(n) = \begin{cases} 0 & , n = 0 \\ 1 & , n = 1 \\ f(n-1) + f(n-2) & , otherwise \end{cases}$$
- Subproblems are smaller versions of the original problem.
- Any problem has overlapping sub-problems if finding its solution involves solving the same subproblem multiple times.
- Take the example of the Fibonacci numbers; to find the `fib(4)`, we need to break it down into the following sub-problems:



Memoization

- Solve the bigger problem by recursively finding the solution to smaller sub-problems.
- Whenever we solve a sub-problem, we cache its result so that we don't end up solving it again
  - Just return the cached result
- This technique is known as memoization

0/1 Knapsack

Problem

- Have a sack with a maximum weight capacity.
- All of the items will exceed the sack's weight capacity.
- Given a list of items with their respective weight and value, maximize the total value of items contained in the sack without exceeding its weight capacity.
- Called 0/1 because the items themselves are not divisible.
- Either select an item ot not
- The set $\{x_0, x_1, x_2, \dots\}$ represents the solution to the problem where x_i represents if $item_i$ was placed in the sack.

- A 1 in the x_0 position means that $item_1$ was placed in the sack, 0 means it wasn't
- For example: given the solution set, $\{0, 1, 1, 0\}$, to a problem with 4 items shows that item 1 and 2 were put in the sack but items 0 and 3 were not.

Further setup

let V be the maximized value solution table
Let W represent the weight limit of the knapsack
let w represent the subproblem weights
let i represent an item
let v_i = the value of item i
let w_i = the weight of item i

We create a table of rows (items) and all possible weight limits from 0 to W
the cells of the table hold the sub problem's maximized value.
then:

$$V[i, w] = \begin{cases} max(V[i-1][w], V[i-1][w-w_i] + v_i) & , when w_i <= w \\ v[i-1][w] & , otherwise \end{cases}$$

Problem 1:
$i = \{0, 1, 2, 3\}$
$v = \{1, 2, 5, 6\}$
$w = \{2, 3, 4, 5\}$

Problem 2:
$i = \{0, 1, 2, 3\}$
$v = \{6, 5, 7, 3\}$
$w = \{5, 3, 4, 2\}$

Longest Common Subsequence

- Defined as the longest subsequence that is common to all the given sequences for which the elements of the subsequence are not required to occupy consecutive positions within the original sequences.

- Example: $\Sigma = \{A, C, G, T\}$

```
let m = |S| and n = |T|  and m' = |S'| and n' = |T'|
S = {G, C, T, A, A, C, T}
T = {A, C, G, T, A, C}
```

Recursive Approach

```
LCS(sequence S, sequence T) RETURN longest common subsequence
START
        let m = SIZEOF(S)
        let n = SIZEOF(T)
        let S' = substring(S, m - 1)
        let T' = substring(T, n - 1)

        IF(m == 0 or n == 0) THEN
                RETURN 0

        IF(S[m-1] == T[n-1]) THEN
                RETURN 1 + LCS(S', T')

        RETURN MAX( LCS(S, T'), LCS(S', T) )
END
```

Dynamic Programming Approach

- Create an (m + 1) x (n + 1) table
  - This table represents the decomposition of the problems into subproblems
  - The rows represent the sequence of symbols of S prefixed by the empty string
  - The columns represent the sequence of symbols of T prefixed by the empty string

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