Dynamic Programming:

Computing Fibonacci Numbers

Algorithmic Paradigms

- Greedy: Build up a global solution incrementally, myopically by optimizing some local criterion.

- Divide-and-conquer: Break up a problem into disjoint (non-overlapping) sub-problems, solve the sub-problems recursively, and then combine their solutions to form solution to the original problem. Brand-new subproblems are generated at each step of the recursion.

- Dynamic programming: Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems. Typically, same subproblems are generated repeatedly when a recursive algorithm is run.

Dynamic Programming History

- Bellman. [1950s] Pioneered the systematic study of dynamic programming.

- Etymology.
  - Dynamic programming = planning over time.
  - Secretary of Defense was hostile to mathematical research.
  - Bellman sought an impressive name to avoid confrontation.

Reference: Bellman, R. E. *Eye of the Hurricane, An Autobiography.*

Dynamic Programming Applications

- Areas.
  - Bioinformatics.
  - Control theory.
  - Information theory.
  - Operations research.
  - Computer science: theory, graphics, AI, compilers, systems, ….

Properties of a Problem that can be Solved with Dynamic Programming

- Simple Subproblems
  - We should be able to break the original problem to smaller subproblems that have the same structure
- Optimal Substructure of the Problems
  - The solution to the problem must be a composition of subproblem solutions
- Subproblem Overlap
  - Optimal subproblems to unrelated problems can contain subproblems in common
- The Number of Distinct Subproblems is Small
  - The total number of distinct subproblems is a polynomial in the input size

Computing Fibonacci Numbers

- Fibonacci numbers:
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$ for $n > 1$

  Sequence is 0, 1, 1, 2, 3, 5, 8, 13, …

- Obvious recursive algorithm (Sometimes can be inefficient):

  Fib(n):

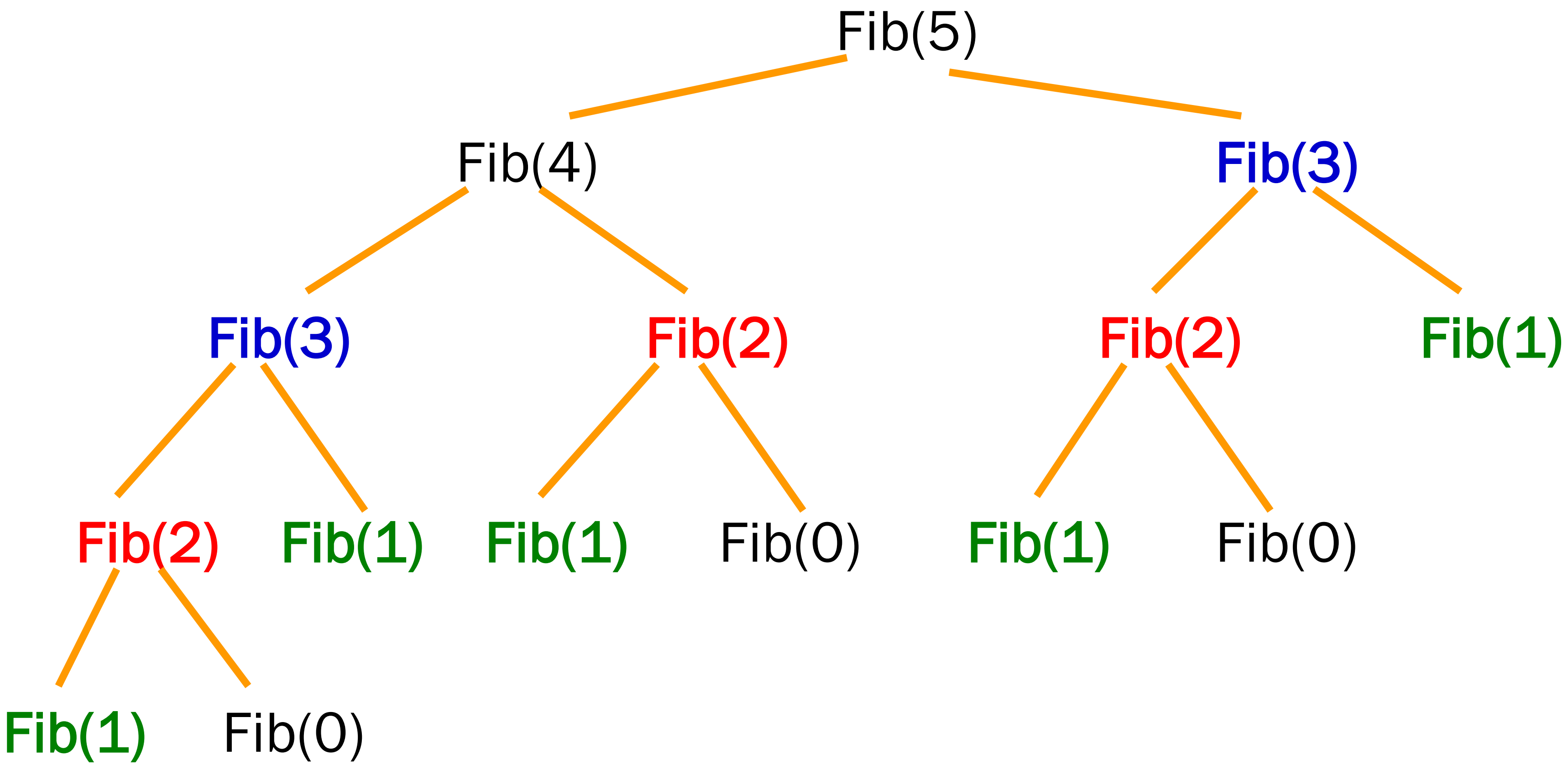
  if n = 0 or 1 then

  return n

  else

  return (Fib(n – 1) + Fib(n – 2))

Recursion Tree for Fib(5)



How Many Recursive Calls?

- If all leaves had the same depth, then there would be about 2^n recursive calls.
- But this is over-counting.
- However with more careful counting it can be shown that it is $\Omega((1.6)^n)$
- Still exponential!

- Wasteful approach - repeat work unnecessarily
  - Fib(2) is computed three times
- Instead, compute Fib(2) once, store result in a table, and access it when needed

More Efficient Recursive Algorithm

- F[0] := 0; F[1] := 1; F[n] := Fib(n);

called memoization

- <u>Fib(n):</u>

  if n = 0 or 1 then return F[n]

  if F[n – 1] = NIL then F[n – 1] := Fib(n – 1)

  if F[n – 2] = NIL then F[n – 2] := Fib(n – 2)

  return (F[n – 1] + F[n – 2])

- computes each F[i] only once, store result in a table, and access it when needed.

Example of Memoized Fib

	F
0	0
1	1
2	NIL 1
3	NIL 2
4	NIL 3
5	NIL 5

Fib(5) — returns 3+2 = 5, fills in F[5] with 5

Fib(4) — returns 2+1 = 3, fills in F[4] with 3

Fib(3) — returns 1+1 = 2, fills in F[3] with 2

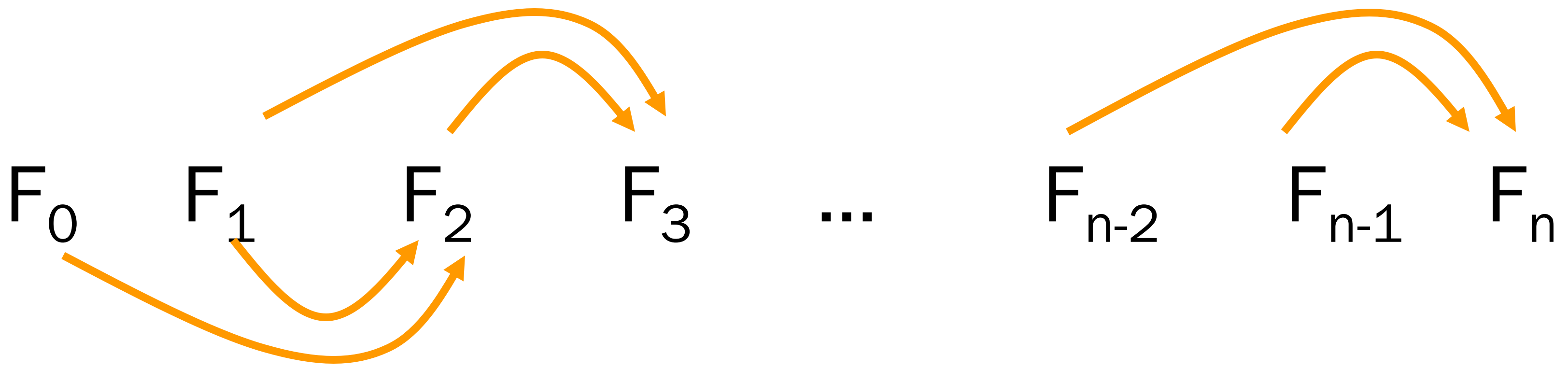
Fib(2) — returns 0+1 = 1, fills in F[2] with 1

Get Rid of the Recursion

- Recursion adds overhead
  - extra time for function calls
  - extra space to store information on the runtime stack about each currently active function call
- Avoid the recursion overhead by filling in the table entries bottom up, instead of top down.

Subproblem Dependencies

- Figure out which subproblems rely on which other subproblems
- Example:



Order for Computing Subproblems

- Then figure out an order for computing the subproblems that respects the dependencies:
  - when you are solving a subproblem, you have already solved all the subproblems on which it depends
- Example: Just solve them in the order

$F_0, F_1, F_2, F_3, \ \ldots$

called Dynamic Programming

DP Solution for Fibonacci

- <u>Fib(*n*):</u>

  F[0] := 0; F[1] := 1;

  for i := 2 to *n* do

  - F[i] := F[i – 1] + F[i – 2]

  return F[*n*]

time reduced from exponential to linear!

- Can perform application-specific optimizations
  - e.g., save space by only keeping last two numbers computed