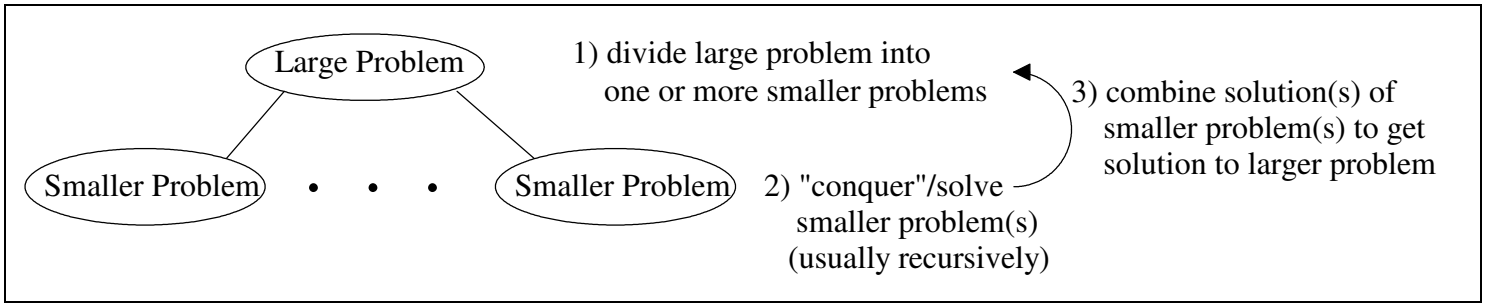
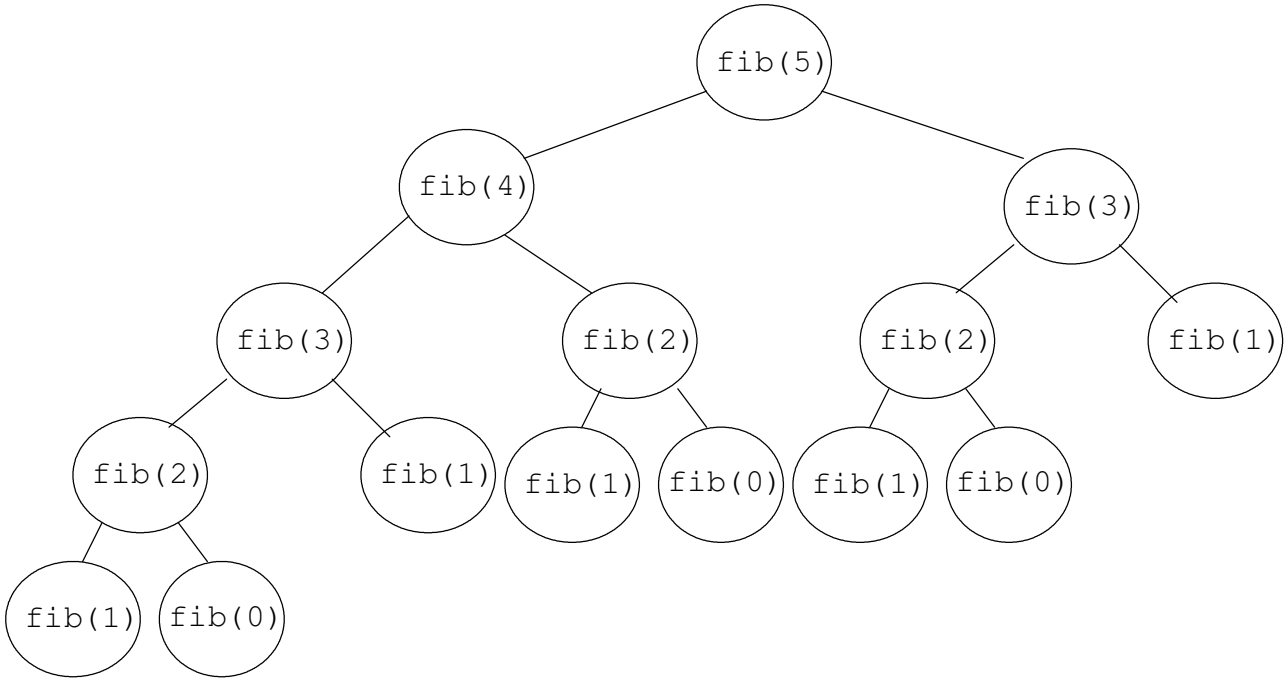
0) In "divide-and-conquer" algorithms: (e.g., recursive fibonacci, binary search, merge sort, ...):




Recursive fibonacci was extremely slow because we re-solved smaller problems many times.




What did we do to make the algorithm fast?

Consider the coin-change problem: Given a set of coin types and an amount of change to be returned, determine the **fewest number** of coins for this amount of change.

1) What "greedy" algorithm would you use to solve this problem with US coin types of {1, 5, 10, 25, 50} and a change amount of 29-cents?

2) Do you get the correct solution if you use this algorithm for coin types of {1, 5, 10, 12, 25, 50} and a change amount of 29-cents?

3) One way to solve this problem in general is to use a divide-and-conquer algorithm. Recall the idea of **Divide-and-Conquer** algorithms.
Solve a problem by:

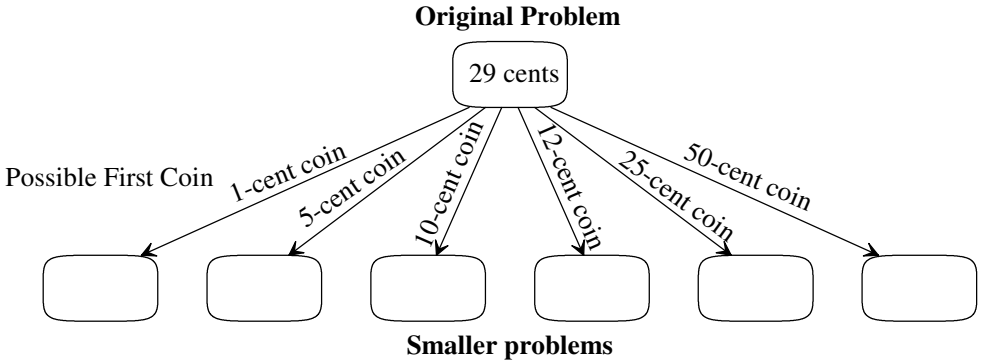
- dividing it into smaller problem(s) of the same kind
- solving the smaller problem(s) recursively
- use the solution(s) to the smaller problem(s) to solve the original problem

a) For the coin-change problem, what determines the size of the problem?

b) How could we divide the coin-change problem for 29-cents into smaller problems?

c) If we knew the solution to these smaller problems, how would be able to solve the original problem?

4) After we give back the first coin, which smaller amounts of change do we have?

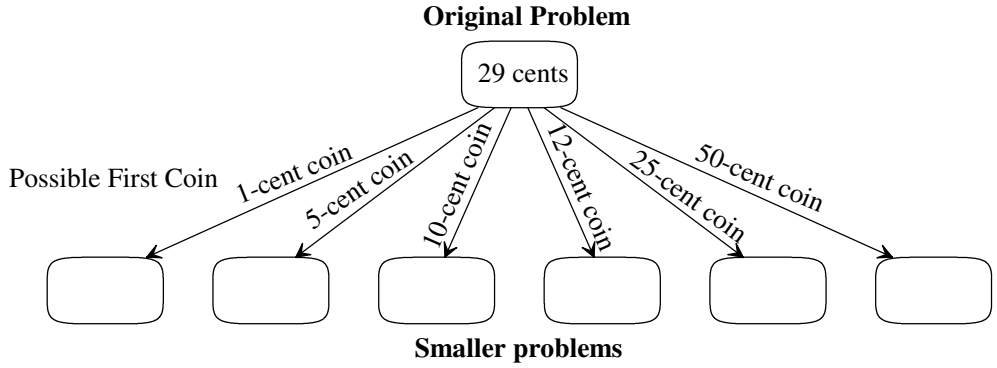



5) If we knew the fewest number of coins needed for each possible smaller problem, then how could determine the fewest number of coins needed for the original problem?

6) Complete a recursive relationship for the fewest number of coins.

$$\text{FewestCoins(change)} = \begin{cases} \min_{\text{coin} \in \text{CoinSet}}( \text{FewestCoins}( \quad ) ) + & \text{if change} \notin \text{CoinSet} \\ 1 & \text{if change} \in \text{CoinSet} \end{cases}$$

7) Complete a couple levels of the recursion tree for 29-cents change using coin types {1, 5, 10, 12, 25, 50}.




8) For coins of {1, 5, 10, 12, 25, 50}, typical timings using a recursive (backtracking) program:

| Change Amount | Run-Time (seconds) | Number of Call-Tree Nodes |
|---|---|---|
| 200 | 0.92 | 236,583 |
| 300 | 33.23 | 8,617,265 |
| 320 | 64.12 | 16,676,454 |
| 340 | 116.8 | 30,370,729 |

Why the exponential growth in run-time?

9) As with Fibonacci, the coin-change problem can benefit from dynamic program since it was slow due to solving the same problems over-and-over again. Recall the general idea of dynamic programming:

- Solve smaller problems before larger ones
- store their answers (so you never need to recompute them)
- look-up answers to smaller problems when solving larger subproblems

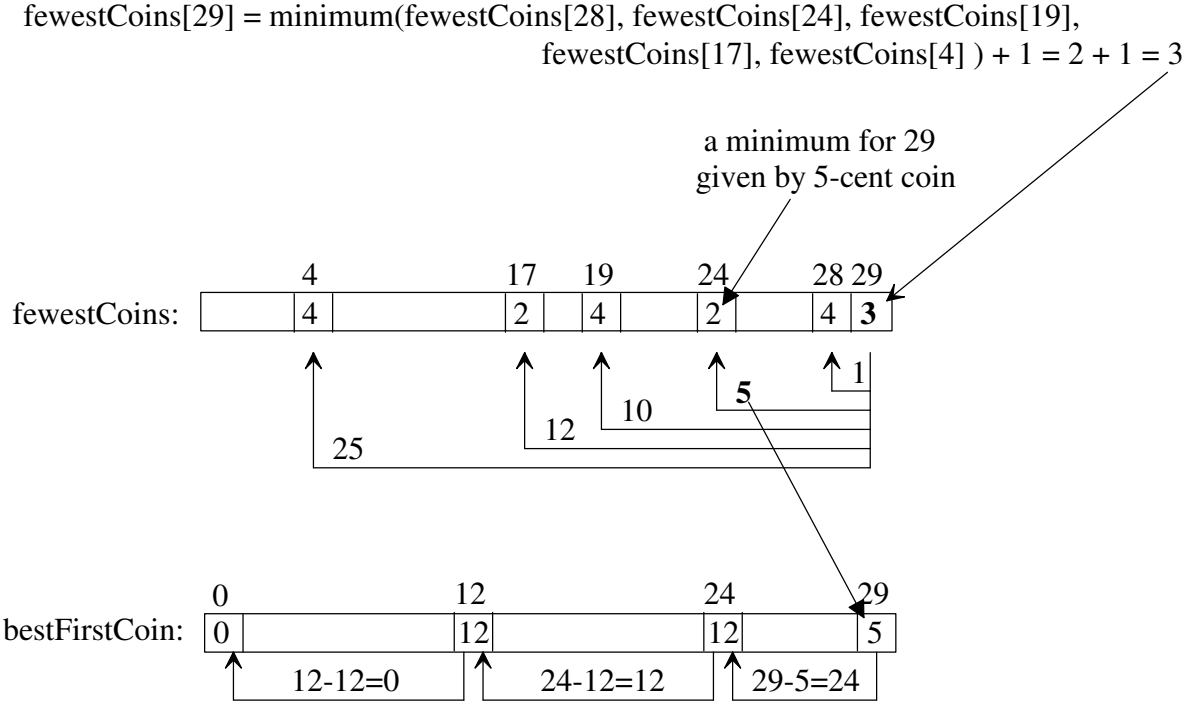
a) How do we solve the coin-change problem using dynamic programming?

Dynamic Programming Coin-change Algorithm:

I. Fills an array fewestCoins from 0 to the amount of change. An element of fewestCoins stores the fewest number of coins necessary for the amount of change corresponding to its index value.

For 29-cents using the set of coin types {1, 5, 10, 12, 25, 50}, the dynamic programming algorithm would have previously calculated the fewestCoins for the change amounts of 0, 1, 2, ..., up to 28 cents.

II. If we record the best, first coin to return for each change amount (found in the "minimum" calculation) in an array bestFirstCoin, then we can easily recover the actual coin types to return.




Extract the coins in the solution for 29-cents from bestFirstCoin[29], bestFirstCoin[24], and bestFirstCoin[12]

b) Extend the lists through 32-cents.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| fewestCoins: | 0 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 1 | 2 | 3 | 2 | 3 | 2 | 3 | 4 | 2 | 3 | 2 | 3 | 2 | 1 | 2 | 3 | 4 | 3 | | | |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| bestFirstCoin: | 0 | 1 | 1 | 1 | 1 | 5 | 1 | 1 | 1 | 1 | 10 | 1 | 12 | 1 | 1 | 5 | 1 | 5 | 1 | 1 | 10 | 1 | 10 | 1 | 12 | 25 | 1 | 1 | 1 | 5 | | | |

c) What coins are in the solution for 32-cents?

10) In Mathematics the factorial function is usually written as n!. For example, 5! = 5 x 4 x 3 x 2 x 1. You've probably seen it implemented as a recursive function, called factorial(n) using the recursive definition:

$n! = n * (n - 1)!$ for n ≥ 1, and
$0! = 1$ for n = 0

You have probably used the binomial coefficient formula:

$$C(n,k) = \frac{n!}{k!(n-k)!}$$

to calculate the number of combinations of "n choose k," i.e., the number of ways to choose k objects from n objects. For example, the number of unique 5-card hands from a standard 52-card deck is C(52, 5).

One problem with using the above binomial coefficient formula directly in most languages is that n! grows very fast and overflows an integer representation before you can do the division to bring the value back to a value that can be represented. When calculating the number of unique 5-card hands from a standard 52-card deck (e.g., C(52, 5)) for example, the value of

52! = 80,658,175,170,943,878,571,660,636,856,403,766,975,289,505,440,883,277,824,000,000,000,000

is much, much bigger than can fit into a 64-bit integer representation.

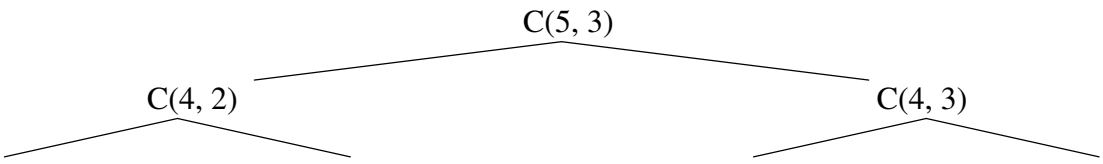
Fortunately, another way to view C(52, 5) is recursively by splitting the problem into two smaller problem by focusing on the hands containing a specific card, say the ace of clubs, and those that do not. For those hands that do contain the ace of clubs, we need to choose 4 more cards from the remaining 51 cards, i.e., $C(51, 4)$. For those hands that do not contain the ace of clubs, we need to choose 5 cards from the remaining 51 cards, i.e., $C(51, 5)$. Therefore, $C(52, 5) = C(51, 4) + C(51, 5)$.

a) Write the recursive definition for the binomial coefficient.

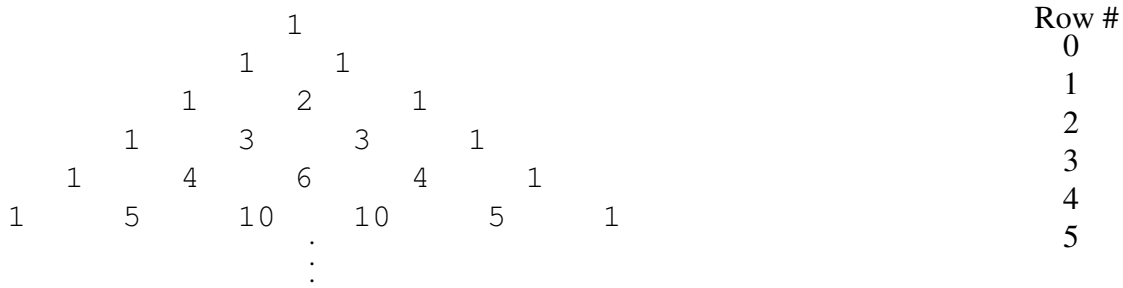
$C(n, k) =$ for $1 \le k \le (n - 1)$, and

$C(n, k) =$ for $k = 0$ or $k = n$

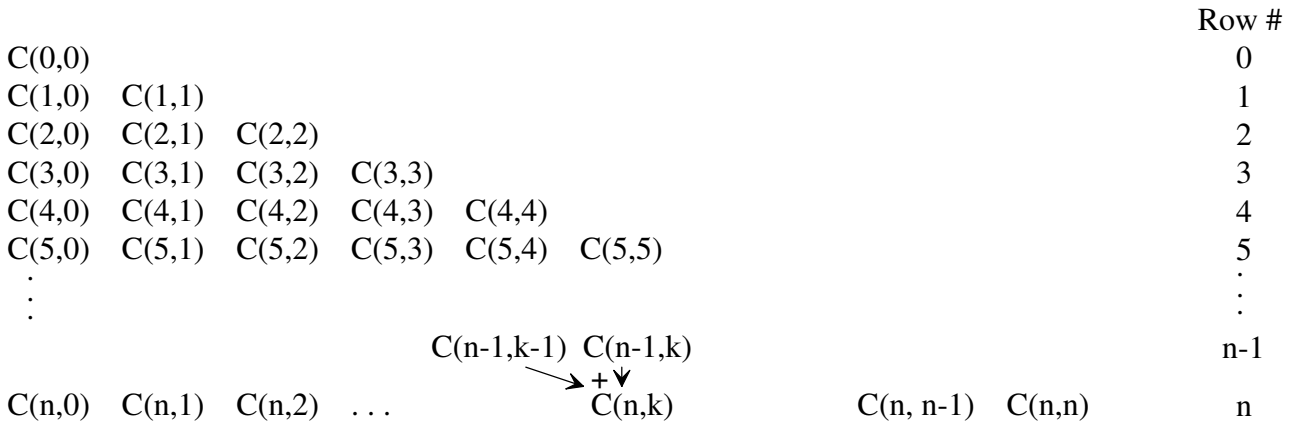
b) As you might guess, implementing the recursive "divide-and-conquer" binomial coefficient function using its recursive definition is slow due to redundant calculations performed due to the recursive calls. Complete the call tree for C(5, 3) = 10 is:



Pascal's triangle (named for the 17th-century French mathematician Blaise Pascal, and for whom the programming language Pascal was also named) is a "dynamic programming" approach to calculating binomial coefficients. It is general written with numeric values in the form:




Recall that dynamic programming solutions eliminate the redundancy of divide-and-conquer algorithms by calculating the solutions to smaller problems first, storing their answers, and looking up their answers if later needed instead of recalculating it. Abstractly, Pascal's triangle relates to the binomial coefficient as in:




c) Write psuedo-code for the "dynamic programming" binomial coefficient function using a two-dimensional array and loops (no recursion needed).

11. Dynamic-programming solutions trade-off space for storing smaller-problem solutions vs. improved execution time. What is the space-complexity of our above algorithm?

12. Often we can optimize (reduce) the amount of storage needed in dynamic programming by not storing solutions to all smaller problems, but just the ones we need again.
a) In binomial coefficient, to calculate the next row of solutions what smaller solutions do we need?

b) How would you modify the previous algorithm to reduce the space-complexity?

c) What is the space-complexity of the modified algorithm?