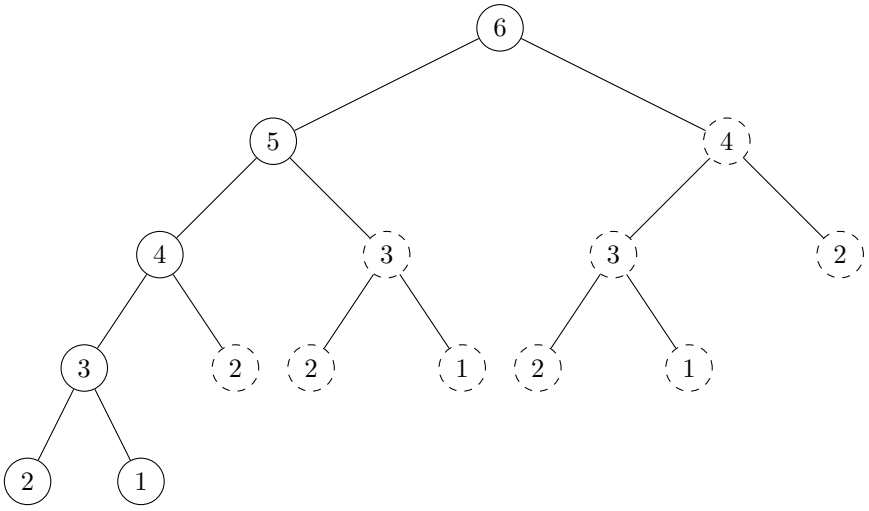
Problem 1.



A number k in a node means that `Fib(k)` is called. Dashed nodes are repetitive work. The runtime can be described by the recurrence $T(n) = T(n-1) + T(n-2) + \Theta(1)$. This is very similar to the Fibonacci sequence itself and it is in fact $T(n) \in \Theta(F(n)) = \Theta(\Phi^n)$ where $\Phi = (1+\sqrt{5})/2$. See pages 774 to 776 in the book for more details. ($O(2^n)$ was acceptable as well, though it is not tight)

Problem 2. Computing the Fibonacci numbers with dynamic programming means that we store the lower numbers in the sequence so we don't have to recompute them repeatedly as in the solution to problem 1.

```
FibDP(n)
   let F[1..n] be a new array
   F[1] = 1
   F[2] = 1
   for i = 3 to n
      F[i] = F[i-1] + F[i-2]
   return F[n]
```

Note that this can also be implemented as memoized recursion. The runtime in either case is $\Theta(n)$, assuming that arbitrarily large numbers can be added in constant time.

Problem 3.

1. The number of multiplications needed to compute the product of a chain of matrices depends on the parenthesization, that is, the order in which we compute the product of pairs of matrices. The matrix chain multiplication problem is to find the parenthesization that yields the lowest number of multiplications necessary to compute the product of a given matrix chain. (Note that the order of the matrices in the chain must not be changed, since matrix multiplication is not generally commutative!)
2. $(A_1A_2)A_3$ and $A_1(A_2A_3)$
3. $7 * 50 * 10 + 7 * 10 * 100 = 10500$ versus $50 * 10 * 100 + 7 * 50 * 100 = 85000$
4. For a chain of n matrices, there are $\Omega(4^n/n^{3/2})$ possible parenthesizations. For each one, it takes $\Theta(n)$ time to compute the required number of multiplications. So the overall runtime for this "brute force" approach would be in $\Omega((4^n/n^{3/2}) * n) = \Omega(4^n/n^{1/2})$.
5. $\Theta(n^3)$
6. Subproblems are subchains from A_i to A_j, so we need two dimensions to reference them.
7. It solves subproblems in the order of the length of the chains, that is, the number of consecutive matrices in the subchains.

Problem 4. $r = [2, 4, 7, 9, 11, 14, 16, 18]$; optimal cut: two pieces of length 1 and two pieces of length 3. Details:

$r[1] = p[1]$
$r[2] = \max(p[1] + r[1], p[2]) = \max(2 + 2, 3) = 4$
$r[3] = \max(p[1] + r[2], p[2] + r[1], p[3]) = \max(2 + 4, 3 + 2, 7) = 7$
$r[4] = \max(p[1] + r[3], p[2] + r[2], p[3] + r[1], p[4]) = \max(2 + 7, 4 + 4, 7 + 2, 8) = 9$
...

Problem 5. The greedy choice is to always use the coin with the highest value (largest denomination) that is at most as high as the remaining value to return. In the example given in the problem description, we pick coin 6 (100) for $v = 289$ and again for $v = 189$. Then the remaining value is 89 and we pick coin 5 (50), the highest value below 90. That leaves 39 and we pick coin 4 (25), leaving 14 which leads us to pick coin 3 (10). Then we have 4 left, which we satisfy with 4 times coin 1. For U.S. coin denominations, the greedy algorithm is optimal.

The runtime of the greedy algorithm is in $O(n + v)$, assuming the list of denominations is sorted, $O(n \log n + v)$ otherwise. To see this, consider two extremes. If v is low, the runtime is dominated by the iteration over the list of coins to find one that is lower. On the other hand, if v is large compared to the denominations of the coins, we might have to pick $\Theta(v)$ coins in change. We can improve this runtime to $O(n)$ by computing how many times the largest coin "fits" into the remaining value and subtracting it that many times instead of picking coins one at a time. That way the algorithm will consider each coin at most once, so it is $O(n)$ or $O(n \log n)$ if the denominations are not sorted.

For general denominations, the greedy algorithm can be arbitrarily worse than the optimum. For instance, for $A = [1, 10, 14]$ and $v = 20$, the optimal solution is $[0, 2, 0]$ but greedy would return $[6, 0, 1]$, 5 more coins than necessary.

Problem 6. The problem is very similar to the rod-cutting problem, differing mainly in the fact that we are trying to minimize a value (the number of coins) instead of maximizing (the value of rod pieces) and that each "piece" (coin) counts the same towards the value we are trying to minimize (as opposed to different values of rod pieces). So we solve the problem by solving it for increasing values i from 1 to v. For each subproblem we have to consider all coins whose denomination is less than or equal to the current i. The following pseudocode implements this approach. It assumes that A is sorted. Otherwise sort it first.

```
Change(A, v)
   let Counts[0..n], Choices[1..n], and B[1..n] be new arrays
   Counts[0] = 0
   for i = 1 to v
      j = 1
      best = ∞
      while j ≤ A.length and A[j] ≤ i
         if best > 1 + c[i - A[j]]
            best = 1 + c[i - A[j]]
            Choices[i] = j
         j += 1
   for i = 1 to n
      B[i] = 0
   k = n
   while k > 0
      B[Choices[k]] += 1
      k = n - A[Choices[k]]
   return B
```

We need to solve v subproblems and for each one we might have to consider all n coins. So the worst-case runtime is in $O(vn)$.

Problem 7. There are $\Theta(n^2)$ subproblems that need to be solved, the activity selection for all subsets S_{ij} of the activity set, with $0 \leq i < j \leq n+1$. For a given pair i, j, the solution can be found with the recurrence

$c[i,j] = \max_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\}$ (if $S_{ij} \neq \emptyset$, else 0; see page 416 in the book). S_{ij} contains those activities that start after a_i finishes and end before a_j starts. Since starting times need to be less than finishing times and the activities are sorted by ascending finishing times, only activities with an index greater than i and less than j can possibly be in S_{ij}. For each one of those $j-i-1$ candidates it can be verified in constant time whether it is in S_{ij} and then it can be determined in constant time whether a valid candidate produces a new maximum with the recurrence. In summary, $\Theta(n^2)$ subproblems need to be solved (in order of increasing length, i.e., difference between i and j) and each one can be solved in $O(n)$ ($j - i - 1 \leq n$), so the runtime is $O(n^3)$.

The following code prints an optimal selection given d and initial values $i = 0$, $j = n + 1$:

```
PrintOpt(d, i, j)
   print(d[i][j])
   PrintOpt(d, i, d[i][j])
   PrintOpt(d, d[i][j], j)
```

Note that $d[i][j]$ can be empty in which case `print(d[i][j])` prints nothing.

If the activities have values, the recurrence becomes: $c[i,j] = \max_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + val[k]\}$. Everything else stays the same.