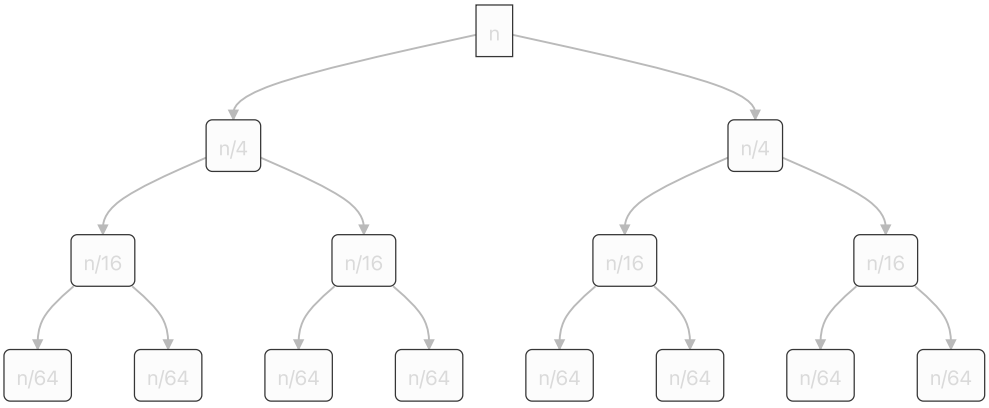
1. Draw the recursion tree for the following recurrence relation.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(\frac{n}{4}) & \text{if } n > 1 \end{cases}$$



2. Notice that the while loop in INSERTION-SORT uses a linear search to scan (backward) through the sorted subarray A[1..j-1]. Could we replace that with binary search to improve its worst-case runtime?

```
INSERTION-SORT (A)
1 for j = 2 to A.length
2     key = A[j]
3     // Insert A[j] into the sorted sequence A[1..j-1].
4     i = j - 1
5     while i > 0 and A[i] > key
6         A[i + 1] = A[i]
7         i = i - 1
8     A[i+1] = key
```

[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 0]
No, we could not. Binary search would work, and would find the correct place to insert the key in $\lg(n)$ time, but it would take n time (worst case) to actually insert it.
Because the process for insertion would essentially be the linear search described in the origional algorithm, no time would be saved, unless you had a linked list or were doing some funny buisness with low-level stuff and how things are stored.

3. Suppose we want to try another hash function for the CSUS student body that maps every student to the day they were born. (For example, if a student was born on May 23, they would be mapped to index 23 of the hash table.) How does this hash function compare to the sum of the letter values you explored in Lab 05?
   There were 429 unique hashes for all 574 students with the unicode hash. There was one hash with an overlap of 6, three with 3, and many with 1 and 2. With birthdays, there would be far more overlaps, because there are only 31 possible hashes, which is far less than 429.

4. Below is pseudocode for solving the Rod-cutting problem (without using dynamic programming). Give a loop invariant to justify the correctness of the for loop on lines 4-5 and justify it.

```
CUT-ROD(p, n)
1 if n == 0
2     return 0
3 q = -∞
4 for i = 1 to n
5     q = max(q, p[i] + CUT-ROD(p,n-i))
6 return q
```

Strictly speaking, it's not correct, because if n>0 and all of the peice sizes in p are greater than n, q will end up as -Inf when the theoretical maximum is actually 0.
If we change line 3 to `q = 0`, the algorithm is fixed.

LI: At the start of every loop (between lines 4 and 5), q is the maximum yield acheivable considering a price list `p[:i-1]`.
Initialization:
At the start, $q = 0$ and $i = 1$, so it is trivial that by not cutting the piece (since `len(p[:0])==0`) you get $0 profit.

Maintainence:
Each step, two things could happen:

1. q remains the same, which is the optimal price without cutting a section of length `i`
   or
2. it becomes `p[i] + CUT-ROD(p, n-i)`, the price gain from the newly cut section of length `i` plus the optimal price for a rod whose length is shortened by enough to fit that new piece.
   The algorithm uses `max()` to determine which of these two options is better. Because they fully represent the space of possible actions at each step, the algorithm is guarenteed to always choose the best option.

The loop then goes back up to the start after having set q for a price list of `p[:i]` and increments `i`. This means that at the start of the next loop, q is the optimal price for a price list `p[i-1]`.

Termination:
At the end of the loop, the algorithm has choosen the optimal solution for every possible decision. Once the for loop exits, `i=n`, so if you increment `i` again (by going back to the top of the loop and only executing the for loop header, not any code), `q` must be the optimal price for a price list `p[:(n+1)-1]`. This means that it, the same q from the end of the loop, was the optimal price for a price list `p[:n]`, which is the whole list. The algorithm must have solved the problem.

CUT-ROD just repeats this process, recursing downwards until it reaches its base case of `n==0`, in which case the result of 0 can be trivially verified.

5. How would you use dynamic programming to solve the following problem:
   Given an array of numbers, find a subset of values such that no two chosen values are adjacent in the array and the sum of all the values in the subset is maximized.
   Let A be the input array and G be our grid, of which G_i represents the maximum possible sum under the rules of the problem for the input array $[A_0, A_1, \ldots, A_i]$. Then:

$$G_i = \max(G_{i-1},\ G_{i-2} + A_i)$$

   For each new element we consider, there are two options: take it, or don't. But if we take it, we cannot have taken anything in the last step, so we must compare the previous elemnt to the sum of the current element and the element *two* indices earlier.
   This will always work because if nothing was taken in the previous step, then the two previous elements will be the same.
   This can be verified. Here's a human-made solution to a small problem:

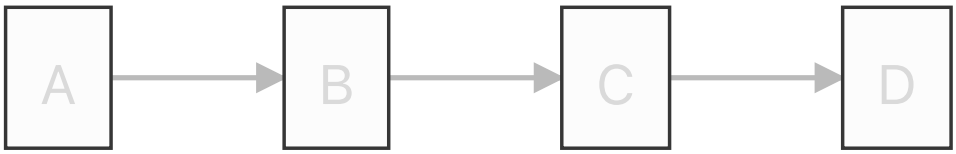
$$\begin{aligned} A &= [1 \quad 4 \quad 3 \quad 2 \quad 5 \quad 2 \quad 5 \quad 3 \quad 0 \quad 2 \quad 3 \quad 5] \\ G &= [1 \quad 4 \quad 4 \quad 6 \quad 9 \quad 9 \quad 14 \quad 14 \quad 14 \quad 16 \quad 17 \quad 21] \end{aligned}$$

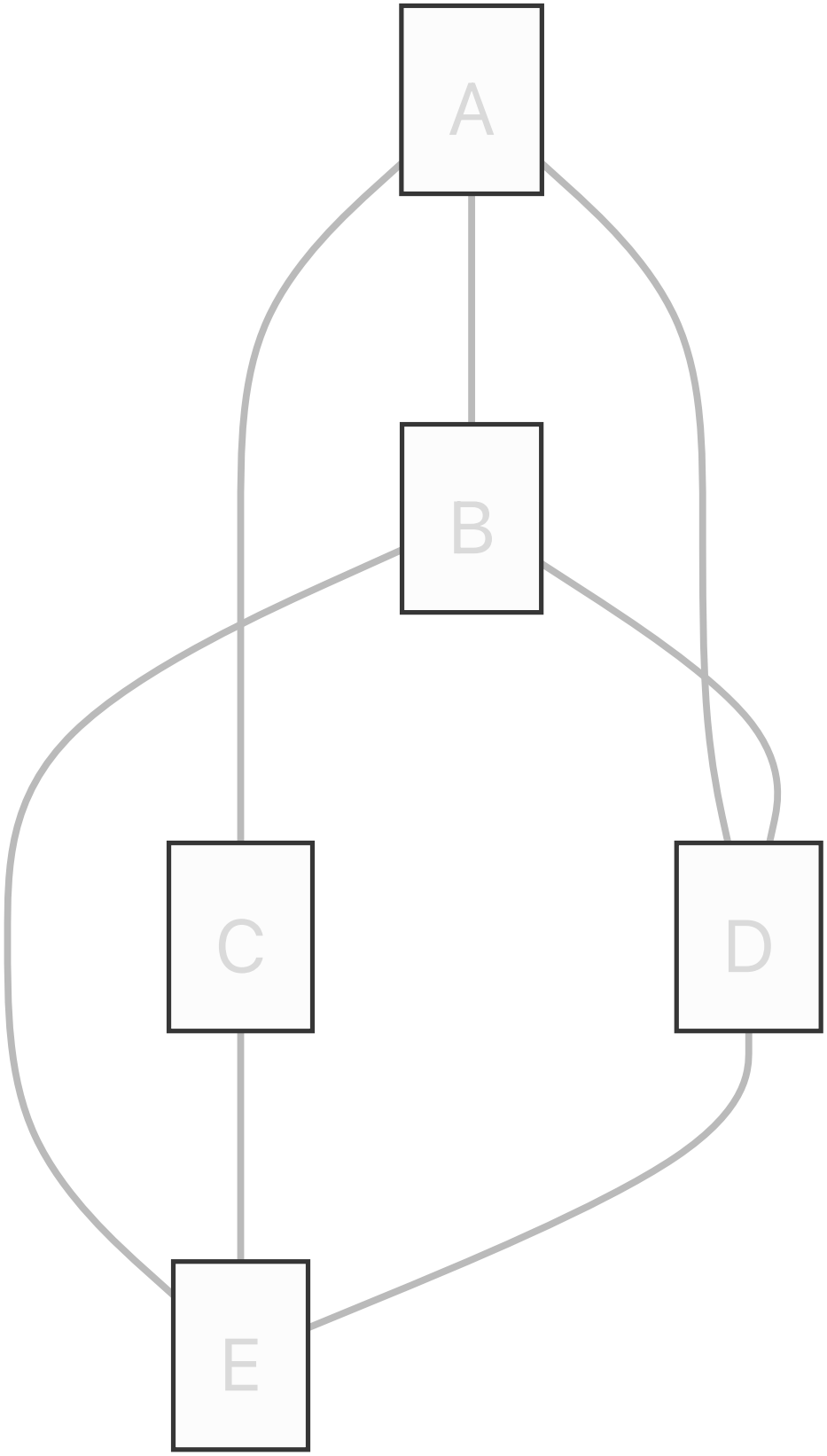
   Now we mix in a little bit of memoization (to taste), and get this:

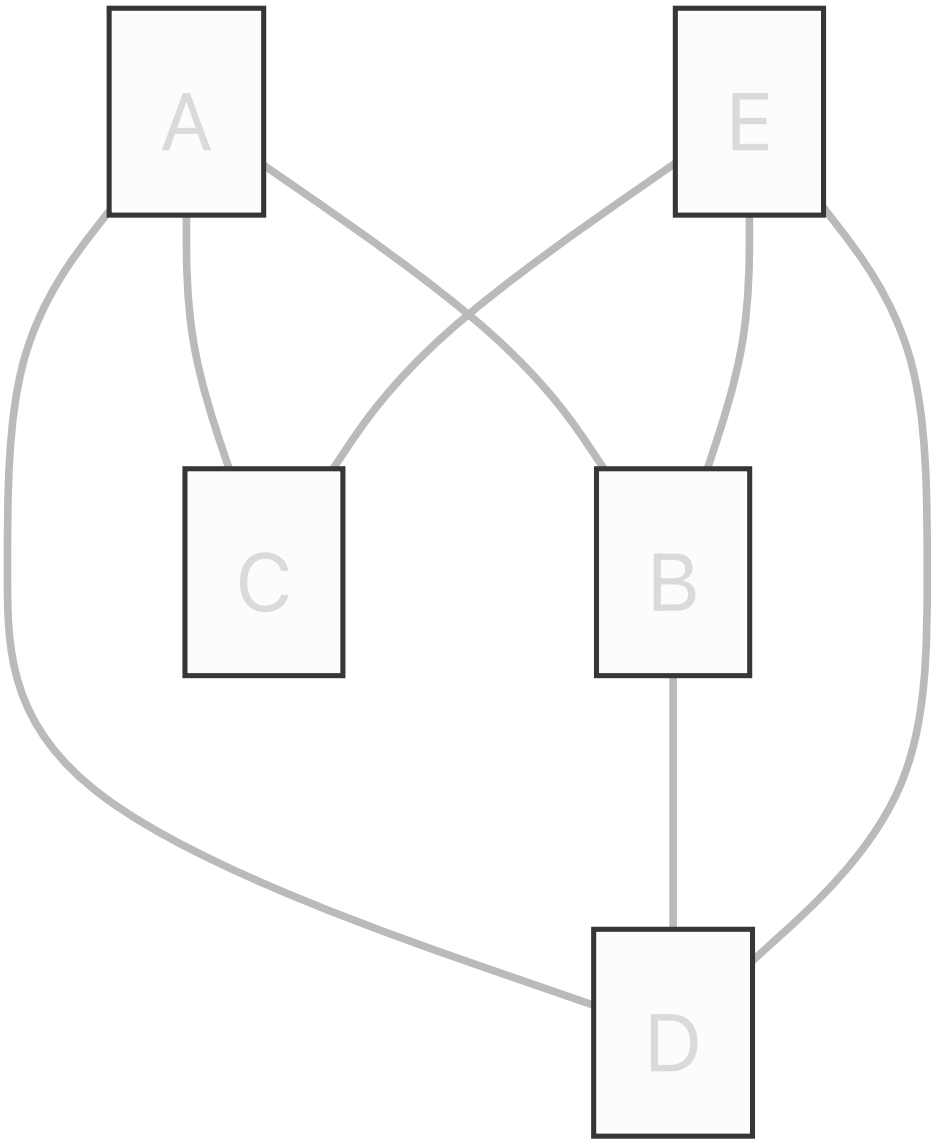
```python
def soln(A):
        # init our grid as [(G_{i-2}, memo),
        #                   (G_{i-1}, memo)]
        grid = [(0, 0b0), (0, 0b0)] # memos notated 0b0 for clarity, not neccessary

        for idx, val in enumerate(A):
                grid = (
                        grid[1], # first element of grid is second element (shift our "moving window")
                        max(
                                grid[1], # second is either previous second (G_{i-1})...
                                (grid[0][0]+val, # or the one before that plus A_i.
                                 grid[0][1]|(1<<idx)), # add the memo, (just previous memo w/ a 1 for this slot)
                            key=lambda x: x[0] # key says: compare the first element of these tuples
                        )
                )

        return grid[1][0], list(map(lambda x: bool(int(x)), # convert to boolean array
                                                            bin( # turn number into binary string representation
                                                                    grid[1][1] # the memo, as a binary number
                                                            )[-1:1:-1] # flip around and remove '0b'
                                                            .ljust(len(A), '0'))) # replace any missing leading zeros
```


A
B
C
D


A
B
C
D
E


A
E
C
B
D