Dynamic Programming Problems:

1. The President of the United States is elected according to the Electoral College, where 568 votes are assigned to the candidates (by state). We assume the following simplifications:

- The candidate that wins the majority of the popular vote in a state wills all of the electoral votes of that state.
- Each of the 50 states must cast all of its electoral votes for the same candidate
- There are two candidates
- The candidate that receives more than 50% of the electoral votes wins.
- That is the winning candidate is the one who receives V >= sum(v_i)/2 + 1 electoral votes. That is one more than half the total number of electoral votes.

Let p_i be the population of state i, and v_i the number of electoral votes for state i. Your goal is to find a set of states S that minimizes the total population needed to win the Electoral College vote.

(a) The dynamic programming solution involves computing a function OPT where OPT[i, v] gives the minimum populations of a set of states from 1, 2, . . . , i such that their votes sum to exactly v. Give a recursive definition of OPT along with the base cases.

OPT[i,v] = min { OPT[i-1, v]. OPT [i-1, v-vi] + pi }

OPT[0,0] = 0, OPT[0,v] = infinity or some large number since we are minimizing above

(b) What value of OPT would contain the final solution?

OPT[n, (V/2) + 1] or OPT[50, 285] if including DC OPT[51, 285] or

The first entry in the table OPT[I, k] such that I = n, k ≥ 285 and OPT[I, k]≠∞.

(c) If there are n states and V electoral votes what is the running time (Big-O) of a dynamic programming algorithm that implements OPT?

O(nV) pseudo-polynomial

Dynamic Programming Problems:

2.

Consider the following game. A "dealer" produces a sequence $s_1 \cdots s_n$ of "cards," face up, where each card s_i has a value v_i. Then two players take turns picking a card from the sequence, but can only pick the first or the last card of the (remaining) sequence. The goal is to collect cards of largest total value. (For example, you can think of the cards as bills of different denominations.) Assume n is even.

(a) Show a sequence of cards such that it is not optimal for the first player to start by picking up the available card of larger value. That is, the natural *greedy* strategy is suboptimal.

(b) Give an $O(n^2)$ algorithm to compute an optimal strategy for the first player. Given the initial sequence, your algorithm should precompute in $O(n^2)$ time some information, and then the first player should be able to make each move optimally in $O(1)$ time by looking up the precomputed information.

Dynamic Programming Problems:

a) Consider the sequence $(2, 10, 1, 1)$. Then the best available card at first is 2. Then the second player can pick the card 10 and will win. If the first player pick the rightmost 1 first, then the first will be able to pick the 10 and will win.

b) Dynamic programming solving.

Suproblems definition. Let $\mathbf{OPT}(i, j)$ be the difference between:

- the largest total the first player can obtain,
- and the corresponding score of the second player

when playing on sequence $s_i, ..., s_j$.

$$s_1 \quad s_2 \quad \dots \quad \dots \quad \boxed{s_i \quad \dots \quad \dots \quad s_j} \quad \dots \quad \dots \quad s_n$$

Recursive formulation. At any stage of the game, there are 2 possible moves for the first player:

- either choose the first card, in which case he will gain s_i and score $-\mathbf{OPT}(i+1, j)$ in the rest of the game,
- or the last card, gaining s_j and $-\mathbf{OPT}(i, j-1)$ from the remaining cards.

Because we are interested in the largest total the first player can gain over the second, we take the maximum of these 2 values (for $i < j$):

$$\mathbf{OPT}(i, j) = \max\{s_i - \mathbf{OPT}(i+1, j), s_j - \mathbf{OPT}(i, j-1)\}$$

And the base cases are: $\forall i \in \{1, ..., n\},\ \mathbf{OPT}(i, i) = s_i$.

Pseudo-code.

```
// base cases
for i = 1...n:
  OPT[i, i] = s[i]

// main loop
for j = 1...n:
  for i = j...1:
    OPT[i, j] = max(s[i] - OPT(i+1, j), s[j] - OPT(i,j-1))

// final result
return OPT[1, n]
```

Complexity.

- We have $O(n^2)$ subproblems,
- each update takes time $O(1)$.

Therefore, the overall complexity is $O(n^2)$.