16 (100 PTS.) Feline indiscretion advised.

Real cats plan out their day by drawing action cards from a deck. Each time they draw a card, they must either perform the action indicated by the card, or else discard it. The possible action cards are Nap, Yawn, Eat, Stretch, and Climb. There are a few rules governing the sequences of actions they perform:

- A Nap can only be followed by a Yawn or a Stretch.
- Eat can only follow a Climb.

Other than these rules, cats are free to choose any *subsequence* of action cards from the deck of cards they are given.

For example, given the following deck of cards

$$N, C, Y, S, C, E, E$$

The subsequence N, C, E would not be compatible with the rules (since Nap was followed by something other than Yawn or Stretch. One longest acceptable subsequence would be N, Y, S, C, E.

16.A. (50 PTS.) Given a deck of n action cards, give a backtracking algorithm for computing the length of the longest sequence of actions compatible with the rules above. Describe the asymptotic running time as a function of n.

Solution:

The rules for acceptable rules are very local in nature... we can express them with a simple function $\mathsf{CanFollow}(X, Y)$, that indicates whether card Y can directly follow card X.

$$\mathsf{CanFollow}(X,Y) = \begin{cases} False & X = N, Y \notin \{Y, S\}\} \\ False & Y = E, X \neq C \\ True & otherwise \end{cases}$$

Let $A[i]$ denote the i'th card in the sequence, 0-indexed. We can give a recurrence for the solution as $\mathsf{Best}(i, X)$, for $0 \leq i \leq n$, where $X \in \{N, C, Y, S, E\}$, which denotes the length of the longest subsequence of $A[i...n-1]$ that would be acceptable if it immediately follows action X. We have the empty sequence as a base case (the only subsequence is length 0), and for the recursive cases, we have the choice to include the next value $A[i]$ or not, as long as $A[i]$ is compatible with X.

$$\begin{cases} 0 & i = n \\ \max \begin{cases} 1 + \mathsf{Best}(i+1, A[i]) \\ \mathsf{Best}(i+1, X) \end{cases} & i < n, \mathsf{CanFollow}(X, A[i]) \\ \mathsf{Best}(i+1, X)) & otherwise \end{cases}$$

The rules are perhaps ambiguous about what actions can go first — Eat can only follow a Climb, but can a cat start its day by Eating breakfast, in which case that card doesn't follow anything? Let's assume this is acceptable. Then we can read the solution directly from $\mathsf{Best}(0, C)$.

This recurrence can be evaluated directly as an algorithm - it would require $O(2^n)$ computations. Cats don't like exponential time any more than we do.

16.B. (50 PTS.) Given a deck of n action cards, give a dynamic programming algorithm for computing the length of the longest sequence of actions compatible with the rules above. Describe the asymptotic running time as a function of n. The running time and space used by your algorithm should be as small as possible.

Solution:

We can in fact reuse the same recurrence relation Best defined for our answer above.

The memoization data structure would be an $n \times 5$ array, such that $M[i, X]$ will store the solution $\mathsf{Best}(i, X)$. This would require $O(n)$ space.

Evaluating each solution, given the previous solutions already filled in, would be $O(1)$ time. There are a total of $5n$ subproblems. Hence the running time will be $O(n)$.

Each subproblem i depends on subproblems $i + 1$. Hence we should evaluate our solution in an order with decreasing values of i from n down to 0.

```
Best(A[0 ... n − 1]) :
let M be an n × 5 array
for i ← n down to 0:
  for X ← {N, C, Y, S, E}:
    if i = n: M[i, X] := 0
    else if CanFollow(X, A[i]):
      M[i, X] := max(1 + M[i + 1, A[i]], M[i + 1, X])
    else M[i, X] := M[i + 1, X]
return M[0, C]
```

17 (100 PTS.) Cutting strings.

Assume you have an oracle that can query assess the "quality" of strings: $\mathbf{q} : \Sigma^* \to \mathbb{N}$. Computing $\mathbf{q}$ using the oracle takes $O(1)$ time regardless of the size of the string.

The following is an example $\mathbf{q}$ function (please note this is just an example, your algorithms must work for *any* $\mathbf{q}$ function):

$$\mathbf{q}(CATDOG) = 10, \qquad \mathbf{q}(CAT) = 9, \qquad \mathbf{q}(DOG) = 10, \qquad \mathbf{q}(ATDO) = 15,$$

where $\mathbf{q}(\text{everything else}) = 0$. For the given input string $x[1 \dots n]$, you may invoke the $\mathbf{q}$ function by passing indexes. Thus, $\mathbf{q}(i, j)$ is a shorthand for $\mathbf{q}(x[i...j])$.

For each of the following, it is sufficient to describe how to compute the value realizing the desired optimal solution.

17.A. (20 PTS.) Give an algorithm to compute the highest quality substring of x.

For example, for the $\mathbf{q}$ function defined above, the highest quality substring of the string $CATDOG$ is 15 (because the highest quality substring is $ATDO$). Analyze the performance of your algorithm. For full credit, you need to *prove* that your algorithm achieves the best possible asymptotic efficiency.

Solution:

The algorithm simply checks all possible substrings $x[i, j]$, with $1 \le i \le j \le n$. There $\binom{n}{2} + n = n(n-1)/2 + n = n(n+1)/2$ such different substrings to try. For each substring we perform one call to $\mathbf{q}$, and return the best substring found. Overall, this takes $O(n^2)$ time.

There is no way to beat the brute force algorithm that evaluates the quality of all $\Theta(n^2)$ possible of x. To prove this, notice that any algorithm M that makes fewer than $\Omega(n^2)$ queries must leave some substrings that are not evaluated. Let $\mathbf{q}_1$ be a function that has a maximum quality q. Let x^* be one of the substrings of x that M leaves not evaluated. We can easily construct a function $\mathbf{q}_2$ that only differs at x^*, and in particular $\mathbf{q}^*_x$ is the maximum quality substring under $\mathbf{q}_2(x)$ but not under $\mathbf{q}_1$, but M must output the same answer in both cases.

17.B. (40 PTS.) A *decomposition* of string x is a sequence of **non-empty** substrings $x_1, x_2, ..., x_k$ such that $x = x_1 x_2 ... x_k$. The *quality* of such a decomposition is

$$\mathbf{Q}(x_1, \dots, x_k) = \min_i \mathbf{q}(x_i).$$

Namely, the quality is determined by the worst substring the decomposition uses.

Give a dynamic programming algorithm (and analyze its performance) to compute the decomposition that **maximizes** the quality $\mathbf{Q}$ of the decomposition (yeh, this is a bit confusing).

For example, if the string is $CATDOG$, then the best possible min-quality is 9. (CAT, DOG is a decomposition that achieves this).

Solution:

Rubric: Standard dynamic programming rubric. -5 points for an $O(n^3)$ time algorithm or an $O(n^2)$ table.

We define the subproblems such that $M(i)$ represents the best min-quality of the substring from 0 to i.

$$\mathsf{M}(i) = \max \begin{cases} \mathbf{q}(0,i) \\ \max\limits_{0 \le k < i} \{ \min\left(\mathsf{M}(k), \mathbf{q}(k+1,i)\right) \end{cases}$$

This will require a memo table of size $O(n)$, and each problem will require an $O(n)$ time to compute, hence this will take $O(n^2)$ running time. We can fill the memo table in increasing order in i.

In pseudocode:

```
MinDecomp(x) :
let M be a length-n array
M[0] := 0
for i ← 1 to n:
    M[i] := max_{0≤k<i} {min(M(k), q(k+1, i))
    M[i] := max(M[i], q(0, i))
return M[n]
```

17.C. (40 PTS.) Give a dynamic programming algorithm (and analyze its performance) to find the best possible *average* quality of the **non-empty** substrings in a decomposition (that is, the sum quality of the substrings in a decomposition divided by the number of substrings in the decomposition).

For example, continuing the above examples, the best average quality decomposition for the string *CATDOG* is 10 (the decomposition *CATDOG* achieves this).

Solution:

Rubric: Standard dynamic programming rubric. -5 points for $O(n^3)$ or larger table, or for $O(n^4)$ or worse running time.

Average needs a slightly different approach, since we need to keep track of the *number* of words as well as their total quality. We will use the index i to describe the prefix of the string up to position i, as well as an index ℓ to describe the desired number of substrings in the decomposition. $M(i,\ell)$ will denote the highest *total* quality decomposition of $x[0..i]$ that consists of exactly ℓ different substrings.

$$\mathsf{M}(i,\ell) = \begin{cases} \mathbf{q}(0,i) & \ell = 1 \\ \max\limits_{0 \le k < i} \{ \mathsf{M}(k,\ell-1) + \mathbf{q}(k+1,i) & \ell > 1 \end{cases}$$

This will require an $O(n^2)$ memo table. Each sub problem requires scanning over $O(n)$ choices of k (where to split off the last substring), hence $O(n^3)$ total time. This can be processed in increasing order of i, increasing order of ℓ, or a diagonal pattern increasing in both.

In pseudocode:

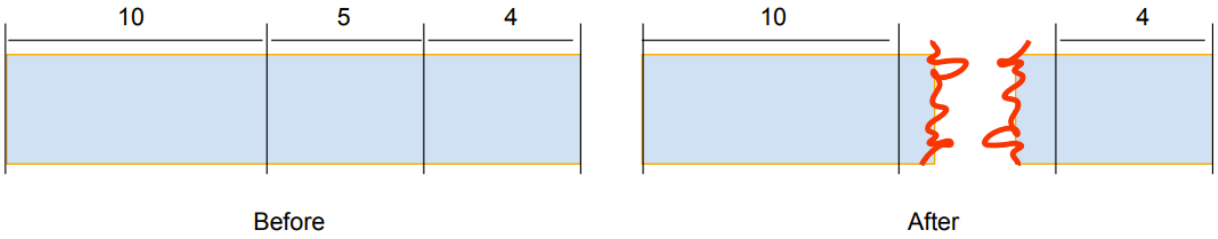
```
TotDecomp(x) :
  let M be an n × n array
  for i ← 0 to 1:
      M[i, 1] := q(0, i)
  for i ← 1 to n − 1:
      for ℓ ← 1 to n:
          M[i, ℓ] := max_{0≤k<i} {M[k, ℓ − 1] + q(k + 1, i)

return max_{ℓ>0} M[n,ℓ]/ℓ
```

Note that we are explicitly abusing notation a bit, and letting the first dimension be 0-indexed and the second dimension be 1-indexed.

18 (100 PTS.) Laser Cuttery

You are in charge of programming the laser cutter at the local Maker Space. A member of the Maker Space needs your help to cut wood for a project. You're given a piece of wood to cut into smaller pieces. The wood is already marked according to positive integer-length segments, the lengths of which are given to you as a sequence.

Your laser cutter isn't tuned all that well, such that the only way to cut a piece of wood into two smaller pieces is to burn through one of the marked segments, destroying it. For example, the following illustrates the result of cutting a piece of wood marked as $[10, 5, 4]$ into two smaller pieces $[10]$ and $[4]$.



The cost of each cut is proportional to the *total length* of the piece being cut. Thus the cut depicted above would cost \$19. (Assume after cutting into two smaller pieces, you can file the ends down for free).

It is also possible to cut off a segment at the end, so for example you could cut $[5, 4]$ into $[5]$, for a cost of \$9.

The project that hired you can only make use of pieces of wood length 5 or smaller. Anything larger is unusable. For example, pieces $[2, 3], [1], [5]$ would all be usable, but $[5, 4]$ would have to be thrown out or cut down further.

18.A. (40 PTS.) Suppose you need to produce usable wood pieces with a *total combined length* as large as possible. Cost is no object. Give an algorithm to compute the largest total combined length you could achieve. Analyze its asymptotic efficiency in terms of n, the number of marked segments in the initial piece of wood.

Solution:

Rubric: Standard dynamic programming rubric.

Let $A[i]$ be the length of the ith segment. Let's also define a helpful $\mathsf{Usable}(L)$ function that returns the length if it is usable:

$$\mathsf{Usable}(L) = \begin{cases} L & \sum L \leq 50 \\ & \textit{otherwise} \end{cases}$$

We can define a recurrence solution as follows, where $M(i)$ denotes the best total length of usable pieces we can cut from $A[i, ..., n-1]$. We have the choice to keep the entire piece, or to place the next cut at any location $i \leq k < n$.

$$M(i) = \begin{cases} 0 & i = n \\ \max \begin{cases} \mathsf{Usable}(\sum A[i..n-1]) \\ \max\limits_{i \leq k \leq \min(n-1, i+6)} \begin{cases} \mathsf{Usable}(\sum A[i..k]) + M(k+1) \end{cases} \end{cases} & \textit{otherwise} \end{cases}$$

Each sub problem apparently requires scanning up to n places to put the next cut, although if we skip the next 6 pieces before placing the next cut, we'd result in an unusable piece. Hence we really only need to scan at most 6 places to put the next cut. As a result each subproblem requires $O(1)$ to compute, an $O(n)$ time algorithm. We should evaluate this in decreasing order of i.

In pseudocode:

```
TotalLength(A) :
let M be a length n + 1 array
M[n] := 0
for i ← n − 1 down to 0:
   M[i] := Usable(∑ A[i..n − 1]) if i + 6 < n else 0
   for k ← i + 1 to min(n − 1, i + 6):
      m := Usable(∑ A[i..k − 1]) + M[k + 1]
      M[i] := max(m, M[i])
return M[0]
```

18.B. (40 PTS.) Suppose you get paid t dollars for each usable piece you produce, for some pre-specified t. Your net profits are the difference between the amount you get paid and the total cost of the cuts you make. Give an algorithm, as efficient as possible, to compute the best possible achievable net profit. Analyze its asymptotic efficiency in terms of n, the number of marked segments in the initial piece of wood.

Solution:

Rubric: Standard dynamic programming rubric.

Finding the cheapest cuts To find the best profit, we can follow a similar approach as for the best binary search tree. We will use two indexes i and j to denote each contiguous subsequence, defining the subproblems. For each problem we search over the k possible places to perform the next cut.

$$M(i,j) = \max \begin{cases} \begin{cases} \$t & \sum A[i..j] \leq 5 \\ \$0 & otherwise \end{cases} \\ \sum -A[i..j] + \max_k \{M(i,k-1) + M(k+1,j)\} \end{cases}$$

Each sub problem apparently requires scanning over $O(n)$ choices to find the best place to cut, hence the estimate is $O(n^3)$ cost.

We can store the results in an $O(n^2)$ memo table. We can fill the table in order of decreasing i, increasing j, or diagonally from $i = n-1$, $j = 1$. Total running time is $O(n^3)$.

In pseudocode:

```
BestCuts(x) :
let M be an n × n array
for i ← n − 1 down to 0:
  for j ← i + 1 to n − 1:
    M[i, j] := $t if ∑ A[i..j] ≤ 5 else 0
    Cost := ∑ A[i..j]
    for k ← i to j
      M[i, j] := max(M[i, j], M[i, k − 1] + M[k + 1, j] − Cost)
return M[0, n − 1]
```

18.C. (20 PTS.) Modify your algorithm for the previous part so that it outputs not just the lowest cost, but also the sequence of cuts necessary to achieve that cost.

Solution:

Reading the solution back from the memo table

We can modify the pseudocode of the previous solution so that we also store the location of the next cut in a table C.

```
BestCuts2(x) :
let M be an n × n array
let C be an n × n array, initialized to −1 for i ← n − 1 down to 0:
  for j ← i + 1 to n − 1:
    M[i, j] := $t if ∑ A[i..j] ≤ 5 else 0
    Cost := ∑ A[i..j]
    for k ← i to j
      p := max(M[i, j], M[i, k − 1] + M[k + 1, j] − Cost)
if p > M[i, j]
  M[i, j] := p
  C[i, j] := k
  return M[0, n − 1]
```

We can now accumulate the resulting list of cuts:

$$\mathsf{Cuts}(i,j) = \begin{cases} [] & C[i,j] = -1 \\ [C[i,j]] + \mathsf{Cuts}(i, C[i,j]-1) + \mathsf{Cuts}(C[i,j]+1, j) & \textit{otherwise} \end{cases}$$

This takes an additional $O(n)$ amount of time.