Dynamic Programming

Solutions -- do not distribute!

A. Short Answer Questions

Give *brief* answers to the following in the cell below.

1. Based on the timing numbers from the `# TIMING INFO FOR PART A` cell above, approximately how much slower is $ways(n)$ than $ways(n-1)$? (i.e. what is $\frac{time(ways(n))}{time(ways(n-1))}$, roughly?)
2. Why is this so slow? What calculations do we compute repeatedly? Hint: consider the following diagram.
   DP diagram
3. **Food for thought (not graded):** Assuming that $time(ways(n)) = time(ways(n-1)) + time(ways(n-2))$, what is $\lim_{n\to\infty} \frac{time(ways(n))}{time(ways(n-1))}$?

A. Answers

Solutions -- do not distribute!

1. ~ 1.6 times slower
2. By breaking the problem down recursively as a binary tree of calculations, we have to call `naive_ways()` 2^n times before getting to the bottom of the tree. Many of the same `naive_ways()` calls are repeated. On the given binary tree diagram, f(n-2) is at least called two times and f(n-3) is at least called three times. We can expect to see more of such repetitions with f(n-4), f(n-5) and f(n-6).
3. This is the Golden Ratio (https://en.wikipedia.org/wiki/Golden_ratio)!

Apply Dynamic Programming

Let's see if we can compute the results more cleverly, by keeping a cache (or table) of intermediate results that we can re-use.

First answer the following questions, then use what you gleaned to examine the code.

B. Short Answer Questions

1. What are the values of A, B, and C in this table?

n	ways(n)
1	1
2	2
3	3
4	5
5	8
6	13
7	A
8	B
9	C

2. To compute these values, did you look at $n = 4$ or earlier?
3. What is the minimum number of values you need to keep as you fill the table from top to bottom, while maintaining the DP property of not recomputing any values?

B. Answers

Solutions -- do not distribute!

- 1 A = 21, B = 34, C = 55
- 2 No we only need to look at n = 5 or later.
- 3 We only need to keep the last 2 values.

C. Short Answer Questions

1. In the inner loop, why do we only need to try each location of a single cut (at `j`)? Why don't we need to try cutting the bar into three or more pieces?
2. Answer the question posed in the code: Why do we not need to optimally split both sides of `j`? In other words, why can we get away with only taking the optimal split (`profit_on_left_most`) on the left and just `score()` the right?
3. Where are the cuts you should make for n=9? Be sure to explain your work. *(Hint: Add a "print" to the code to see where you end up updating the profit made. Leave this code in when you submit.)*

C. Answers

Solutions -- do not distribute!

1. To the left of the cut `j`, the optimal finer cuts of interval [0,j] has already been cached while we loop through smaller values of `k` in the previous outer loop.

2. To the right of the cut `j`, the optimal finer cuts of interval [j,k] will be accounted for in future iterations of the inner loop. Let's say `k=10` and in our current inner loop `j=6`, the optimal finer cuts for left interval [0,6] has already been cached from earlier calculations, the optimal finer cuts for right interval [6,10] will be covered a few loops later when `j=10` for our left interval [0,10]. Using dynamic programming on this problem, we only need to care about building the finer cuts solutions from left to right of the bar.

3. We need to make cuts at n = 1, 4.

Bookkeeping

Great! We know we can make over a hundred dollars cutting up our length 9 bar!

Unfortunately, while we computed the revenue available by cutting the bar, we didn't actually track the cuts we need to make in order to earn it!

D. Coding Exercise

No coding solutions for part (d)

Finish modifying the code (in the cell below) to keep track of the cuts you made in order to achieve the optimal revenue.

i.e. `potential_profit[4]` might be `(39.5, [1, 2])`.

In [6]:
```
# DP cache for the function below.
def best_cuts_with_trace(n, score):
    """Determine the optimal revenue possible by (optionally) cutting a bar
       of length n.

    Args:
      n: the length of the pipe
      score: a function that accepts "left" and "right and gives you the score
             (revenue) received for the segment of pipe extending the interval
             from [left, right)

    Returns:
      The best profit to be had for a pipe of length n.
    """

    profit_on_left_most = []
    for k in range(n + 1):
        # What if we don't split the k-length bar at all?
        profit_on_left_most.append((score(0, k), []))  # We did this part for
 you:  no cuts, so [].

        # Maybe split it up.  Pick a split location at "j", optimally split th
e left
        # hand side but keep the right hand side whole.
        # (Answer below why this doesn't skip any options.)
        for j in range(k):
            pass  # Placeholder so python doesn't complain about having no bod
y.
            # YOUR CODE HERE
            # In the previous cell, this block looked like this:
            # potential_profit = profit_on_left_most[j] + score(j, k)
            # if potential_profit > profit_on_left_most[k]:
            #     profit_on_left_most[k] = potential_profit
            #
            # Hint:  When adding "j" to the list of cuts already required, us
e:
            # original_list + [j], not original_list.append(j).  You don't wan
t to edit that original
            # list (since then it'll be wrong in its original place).



            # END YOUR CODE HERE

    return profit_on_left_most[n]


result = best_cuts_with_trace(9, pipe_profit)
# You may end up with [4, 1] (or if you edit a lot more code than you need to,
 other equivalent sets of cuts).
# At the very least, make sure you return them sorted as the code below relies
 on it.
# You should NOT end up with a cut at 0 or a cut at 9.  That's the end of the
 bar already - no need to cut again!
assert result == (100.5, [1, 4])
```

```
          assert result[1] == sorted(result[1])
          result
Out[6]:   (100.5, [1, 4])
```

We'll see bookkeeping like this throughout the rest of the course. For example, we'll want to know the optimal way to tag words in a sentence with their parts of speech. The optimization will be over some likelihood of a particular assignment (rather than revenue). This optimization is only useful to us however if we have a way to know what sequence of part of speech tags gave us that score, so we'll have to do this same kind of bookkeeping.

Pipe cutting is segmentation

The next cell implements a very light wrapper around your best_cuts_with_trace function (it just takes the cut indexes your function returns and turns them into text to pretty-print). It also implements a simple unigram language model (much simpler than what you will build in later assignments!).

Don't worry about the details here, this is just a fun coda to the assignment! Figuring out these details will be the work of the rest of the course.

Feel free to experiment with some sentences. See if you can find at least one that breaks it, yielding a sub-optimal segmentation. If the dynamic programming algorithm is exact (finds the highest scoring split), how can it produce a bad solution?

```
 In [9]:  segment('helloworldhowareyou')

 Out[9]:  (-46.80790451590498, ['hello', 'world', 'how', 'are', 'you'])

In [10]:  segment('downbythebay')

Out[10]:  (-31.397834251805303, ['down', 'by', 'the', 'bay'])

In [11]:  segment('wikipediaisareallystrongresourceontheinternet')

Out[11]:  (-70.82840309210602,
           ['wikipedia',
            'is',
            'a',
            'really',
            'strong',
            'resource',
            'on',
            'the',
            'internet'])
```

Congratulations!

You're done with Dynamic Programming.

There is a completely optional section that shouldn't take very long, if you're keen to learn about edit distance. We won't delve into much detail about this anywhere else in the class.

(Optional) String edit distance

Another classic DP problem in the NLP space - but not one we otherwise will talk about in the course is the idea of "edit distance (https://en.wikipedia.org/wiki/Levenshtein_distance)". It's a way of measuring how many "edits" to one string you need to make in order to turn it into another.

We've provided two implementations below for you to play with.

1. **levenshtein_cache:** The "cache everything in a dict" approach is first. The keys are coordinates into a table that is len(str1) x len(str2) in size.
2. **levenshtein_explicit:** Similar to the version of ways(n) that only keeps the previous two values at hand, the explicit ordering approach only keeps the immediately previous row of the table while building the next. Setting the verbose flag to this version prints each row of the table out as it computes it.

E. (Optional) Short Answer Questions:

Give brief answers to the following in the cell below.

1. Let `n = len(str1)` and `m = len(str2)`. In terms of `n` and `m`, what is the size of the DP table (cache) for computing Levenshtein distance? *Hint: how many valid keys are there? Do we use all of them?*

2. Based on your answer to 1., what is the running time (in Big-O notation) of the edit distance algorithm? *Hint: it takes $O(1)$ work at each step, assuming we have the needed cache entries.*

3. Consider transpositions (as mentioned in section 6.6 of the async), such as `xy` -> `yx`. How can we compose a transposition from insertions, deletions, and substitutions? What is the edit distance between `wxyz` and `wyxz`?

4. Suppose we wanted to handle transpositions directly, rather than allowing our algorithm to compose them from other operations. (This might be useful if we want to score them differently.) If we have for the other operations:

   ```
   _ed(i - 1, j) + 1  # insertion
   _ed(i, j - 1) + 1  # deletion
   _ed(i - 1, j - 1) + substitution  # substitution, free if letters match
   ```

   what line would we add (calling `_ed`) to handle a transposition? (You may want to define a variable `transposition_match` to check that a transposition makes sense at the current position.) Based on your answer to 1. and 2., does this change the Big-O runtime of the algorithm?

E. Answers (optional)

1. *Your answer here!*
2. *Your answer here!*
3. *Your answer here!*
4. *Your answer here!*

```
In [14]: # Substitution.
         levenshtein_explicit('abc', 'dbc', verbose=True)

         range(0, 4)
         [1, 1, 2, 3]
         [2, 2, 1, 2]
         [3, 3, 2, 1]

Out[14]: 1
```

In [15]:
```
# Deletion.
levenshtein_explicit('abc', 'ac')
```

Out[15]:
```
1
```

In [16]:
```
# Insertion.
levenshtein_explicit('ac', 'abc')
```

Out[16]:
```
1
```

In [17]:
```
# All of the above.
levenshtein_cache('kitten', 'sitting')
```

Out[17]:
```
3
```

In [18]:
```
# Fun!
levenshtein_cache('w266 class', 'with 6 classic tricks')
```

Out[18]:
```
13
```