CSCI 3104 Spring 2018

Problem Set 7

Merola, Michael

06/04/1998

Problem Set 7

In [5]:
```
import pandas as pd
import numpy as np
import matplotlib.pylab as plt
%matplotlib inline

namedata_path = 'https://www2.census.gov/topics/genealogy/1990surnames/dist.all.last'
```

Problem 1

(45 pts) Recall that the string alignment problem takes as input two strings x and y, composed of symbols $x_i, y_j \in \Sigma$, for a fixed symbol set Σ, and returns a minimal-cost set of edit operations for transforming the string x into string y.

Let x contain n_x symbols, let y contain n_y symbols, and let the set of edit operations be those defined in the lecture notes (substitution, insertion, deletion, and transposition).

Let the cost of indel be 1, the cost of swap be 13 (plus the cost of the two sub ops), and the cost of sub be 12, except when $x_i = y_j$, which is a "no-op" and has cost 0.

In this problem, we will implement and apply three functions.

(i) alignStrings(x,y) takes as input two ASCII strings x and y, and runs a dynamic programming algorithm to return the cost matrix S, which contains the optimal costs for all the subproblems for aligning these two strings

(ii) extractAlignment(S,x,y) takes as input an optimal cost matrix S, strings x; y, and returns a vector a that represents an optimal sequence of edit operations to convert x into y. This optimal sequence is recovered by finding a path on the implicit DAG of decisions made by alignStrings to obtain the value S[nx; ny], starting from S[0; 0].

(iii) commonSubstrings(x,L,a) which takes as input the ASCII string x, an integer $1 \leq L \leq nx$, and an optimal sequence a of edits to x, which would transform x into y. This function returns each of the substrings of length at least L in x that aligns exactly, via a run of no-ops, to a substring in y.

(a) From scratch, implement the functions alignStrings, extractAlignment, and commonSubstrings. You may not use any library functions that make their implementation trivial. Within your implementation of extractAlignment, ties must be broken uniformly at random. Submit (i) a paragraph for each function that explains how you implemented it (describe how it works and how it uses its data structures), and (ii) your code implementation, with code comments. Hint: test your code by reproducing the APE / STEP and the EXPONENTIAL / POLYNOMIAL examples in the lecture notes (to do this exactly, you'll need to use unit costs instead of the ones given above)

I Don't Know

(b) Using asymptotic analysis, determine the running time of the call commonSubstrings(x, L, extractAlignment(alignStrings(x,y), x,y)) Justify your answer.

I Don't Know

(c) (15 pts extra credit) Describe an algorithm for counting the number of optimal alignments, given an optimal cost matrix S. Prove that your algorithm is correct, and give is asymptotic running time. Hint: Convert this problem into a form that allows us to apply an algorithm we've already seen.

I Don't Know

(d) String alignment algorithms can be used to detect changes between different versions of the same document (as in version control systems) or to detect verbatim copying between different documents (as in plagiarism detection systems). The two data string files for PS7 (see class Moodle) contain actual documents recently released by two independent organizations. Use your functions from (1a) to align the text of these two documents. Present the results of your analysis, including a reporting of all the substrings in x of length L = 9 or more that could have been taken from y, and briefly comment on whether these documents could be reasonably considered original works, under CU's academic honesty policy.

I Don't Know

Problem 2

(20 pts) Ron and Hermione are having a competition to see who can compute the nth Pell number P_n more quickly, without resorting to magic. Recall that the nth Pell number is defined as $P_n = 2P_{n-1} + P_{n-2}$ for $n > 1$ with base cases $P_0 = 0$ and $P_1 = 1$. Ron opens with the classic recursive algorithm:

In [ ]:
```
Pell(n):

    # BASE CASE
    if (n == 0):
        return 0
    elif (n == 1):
        return 1

    # ALGO
    else:
        return 2*Pell(n-1) + Pell(n-2)
```

(a) Hermione counters with a dynamic programming approach that "memoizes" (a.k.a. memorizes) the intermediate Pell numbers by storing them in an array $P[n]$. She claims this allows an algorithm to compute larger Pell numbers more quickly, and writes down the following algorithm.

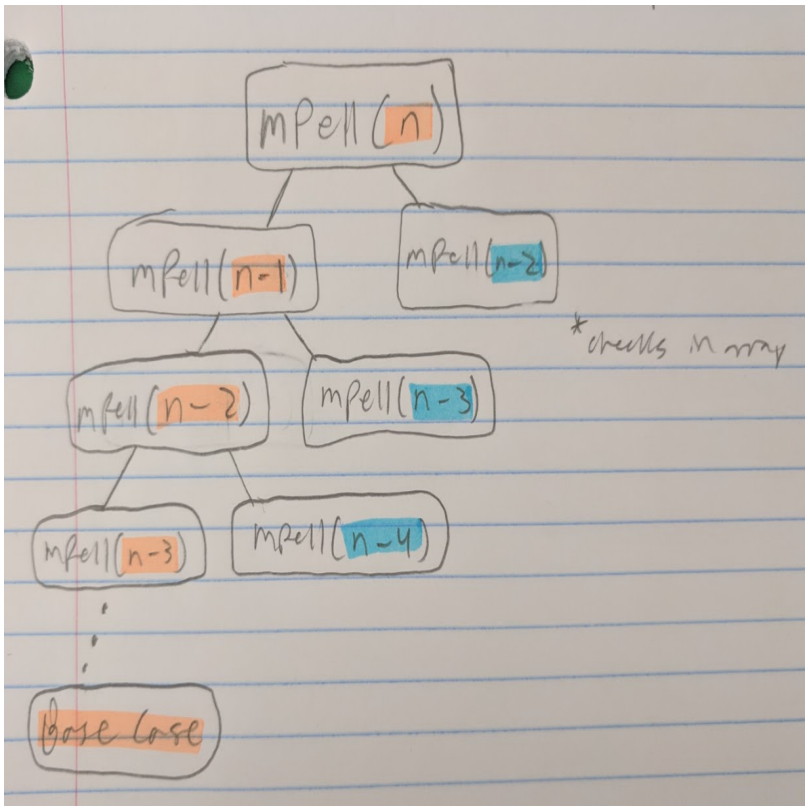
In [ ]:
```
MemPell(n):

    # BASE CASE
    if (n == 0):
        return 0
    else if (n == 1):
        return 1

    # ALGO
    else:
        if (P[n] == undefined):
            P[n] = 2*MemPell(n-1) + MemPell(n-2)
        return P[n]
```

(i) Describe the behavior of MemPell(n) in terms of a traversal of a computation tree. Describe how the array P is filled.

MemPell calculates the Pell number at n using a recursive computation tree. The root of the tree starts with MemPell(n) and branches each time on the left with MemPell(n-1) and on the right with MemPell(n-2). At each branch, the Pell number at n is calculated recursively and continues until the tree reaches the base case. Then each value is calculated as the tree returns through the branches. However, each value is only calculated once because the algorithm checks if there is no value at "n", and stores the Pell value at *n* in the array. Because of this check, the computation tree does not need to recomputate redundant values and stays efficient.



(ii) Determine the asymptotic running time of MemPell. Prove your claim is correct by induction on the contents of the array.

According to the computation tree for MemPell, the algorithm calculates each value from the base case to *n* only **once**. Because the algo does *n* operations over a span of *log(n)* recursions, its asymptoticrunning time is $O(n)$.

(b) Ron then claims that he can beat Hermione's dynamic programming algorithm in both time and space with another dynamic programming algorithm, which eliminates the recursion completely and instead builds up directly to the final solution by filling the P array in order. Ron's new algorithm is

In [ ]:
```
def DynPell(n) :
    P = []
    # BASE CASE
    P[0] = 0
    P[1] = 1

    # ALGO
    for i in range(2, n) :
        P[i] = 2*P[i-1] + P[i-2]

    return P[n]
```

Determine the time and space usage of DynPell(n). Justify your answers and compare them to the answers in part (2a).

$\sum_{i=2}^{n} P[i] = 2 * P[i-1] + P[i-2]$

so the **time complexity** for DynPell(n) is $O(n)$

Because the algorithm stores all calculatations in an array of size **n**...

space complexity for DynPell(n) is $O(n)$

DynPell is faster than MemPell because it avoids using recursion to solve the Pell calculations, but both have the same space complexity because they dynamically store calculations.

(c) With a gleam in her eye, Hermione tells Ron that she can do everything he can do better: she can compute the nth Pell number even faster because intermediate results do not need to be stored. Over Ron's pathetic cries, Hermione says

In [ ]:
```
FasterPell(n) :
    a = 0
    b = 1
    for i in range(2, n):
        c = 2*a + b
        a = b
        b = c

    return a
```

Ron giggles and says that Hermione has a bug in her algorithm. Determine the error, give its correction, and then determine the time and space usage of FasterPell(n). Justify your claims.

Bug : $c = 2 * a + b$; **Correct** : $c = 2 * b + a$

Bug : return a ; **Correct** : return c

b is equivalent to $n - 1$ and a is equivalent to $n - 2$, so a&b need to be swapped for the algo to calculate Pell correctly. You also need to return c instead of a because c is the most recent calculation.

The algo runs thru n operations in the for loop, so its **time complexity** is $O(n)$

FasterPell() only stores the past two calculations to operate, so its **space complexity** is atomic, $O(1)$

(d) In a table, list each of the four algorithms as columns and for each give its asymptotic time and space requirements, along with the implied or explicit data structures that each requires. Briefly discuss how these different approaches compare, and where the improvements come from. (Hint: what data structure do all recursive algorithms implicitly use?)

In [3]:
```
Algorithm  | Time Complexity | Space Complexity | Structure
-------------------------------------------------------------------------
Pell       | O(2^n)        | O(n)             | Full Binary Tree
MemPell    | O(n)            | O(n)             | Unbalanced Tree, Array
DynPell    | O(n)            | O(n)             | Array
FasterPell | O(n)            | O(1)             | none
```

```
  File "<ipython-input-3-f260df21cd24>", line 2
    Algorithm  | Time Complexity | Space Complexity | Structure
                                 ^
SyntaxError: invalid syntax
```

(e) Implement FasterPell and then compute P_n where n is the four-digit number representing your MMDD birthday, and report the first five digits of P_n. Now, assuming that it takes one nanosecond per operation, estimate the number of years required to compute P_n using Ron's classic recursive algorithm and compare that to the clock time required to compute P_n using FasterPell.

In [8]:
```
def FasterPell(n) :
    a = 0
    b = 1
    for i in range(2, n):
        c = 2*b + a
        a = b
        b = c

    return c

birthday = 604
print(FasterPell(birthday))
print()
print("23024")
```

```
23024662091417682014145684577059889616286131373750941920226309696425781
88927993388351016205920533467389337346791657937002980551306332353087984
90800928141280027925124576247588183007048570885674034909966955351388178
874130370044651005

23024
```

FasterPell Clocktime:

- 5 atomic operations, so 5 ns per loop
- loop runs 602 times

5 * 602 = 3010 ns

ClassicPell Clocktime:

3 atomic operations, so 3 ns per recursion

2^{602} = recursive calls

$(2^{602} * 3) * 1e-9 = t_s$ ...time in seconds

$t_s/31,536,000$ ...seconds to days

$= 1.5789^{165}$ years

The clocktime for fasterPell is significantly smaller than for classicPell. The biggest drawback to the recursive algorithm is its ineffecient computation binary tree.

Sources

People

Krish Dholakiya

Gustav Solis

George Allison

Selena Quintilla

Eric Weng