CSCI 3104 Spring 2018 Problem Set 7
Cole Schlisner (03/10)

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.*

```
alignStrings(x,y) :  // x,y are ASCII strings
  S = table of length nx by ny // for memoizing the subproblem costs
  initialize S      // fill in the basecases
  for i = 1 to nx {
    for j = 1 to ny {
      S[i,j] = cost(i,j) // optimal cost for x[0..i] and y[0..j]
  }}
  return S
```

(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 $S[n_x, n_y]$, starting from $S[0, 0]$.*

```
extractAlignment(S,x,y) : // S is an optimal cost matrix from alignStrings
  initialize a            // empty vector of edit operations
  [i,j] = [nx,ny]         // initialize the search for a path to S[0,0]
```

```
while i > 0 or j > 0
  a[i] = determineOptimalOp(S,i,j,x,y) // what was an optimal choice?
  [i,j] = updateIndices(S,i,j,a) // move to next position
}
return a
```

When storing the sequence of edit operations in `a`*, use a special symbol to denote no-ops.*

(iii) *`commonSubstrings(x,L,a)` which takes as input the ASCII string x, an integer $1 \leq L \leq n_x$, 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, youll need to use unit costs instead of the ones given above).

alignStrings: This implementation is essentially the same algorithm as the pseudocode provided, except instead of using a separate function cost(i,j) to determine the optimal subproblem cost it just stores the subproblem costs in an array and picks the minimum. The specific subproblems stored in the cost array (added with their respective costs) are S[i-2][j-2] (Swap), S[i-1][j-1] (Sub), S[i-1][j] (Delete), S[i][j-1] (Insert). S[i][j], the current subproblem is set to the minimum of these. The algorithm first initializes S with the base cases (first column and first row), and then computes each row sequentially using the previous entries in the table.

python 3.6:

```
def alignStrings(x,y):
  costS = 12
  costT = 13
  costID = 1

  # Y on top, X on side as in example
  S = [[0 for j in range(len(y)+1)] for i in range(len(x)+1)]

  # base cases = cost of aligning empty strings
  for j in range(len(y)+1):
    S[0][j] = j
  for i in range(len(x)+1):
    S[i][0] = i

  for i in range(1,len(x)+1):
    for j in range(1,len(y)+1):
      cost = []

      # Swap -- i-2,j-2
      if j > 1 and i > 1:
        cost.append(S[i-2][j-2] + costT
              + (0 if (x[i-1] == y[j-2]) else 1)
              + (0 if (y[j-1] == x[i-2]) else 1))

      # Sub -- i-1,j-1
      cost.append(S[i-1][j-1] + (costS if x[i-1] != y[j-1] else 0))

      # insert -- i,j-1
      cost.append(S[i][j-1] + costID)

      # delete -- i-1,j
      cost.append(S[i-1][j] + costID)

      S[i][j] = min(cost)
  return S
```

extractAlignment: This implementation works in largely the same way that alignStrings works – by looking at each relevant subproblem and picking the minimum. The differences are that (a) this is done in reverse and (b) instead of setting S[i][j] to this minimum value, we record which subproblem the minimum came from (using the *ops* array) and then set the coordinates i and j to the coordinates of the subproblem. Specifically, this algorithm records the indicies of all subproblems with the minimum value within the cost array and picks one at random. This allows the algorithm to break ties evenly if there are two or more subproblems that yield the same optimal solution. My implementation also records the indicies of the chosen subproblems in an array (*optimalpath*) so that the path of the extracted alignment can be visualized easily.
python 3.6 (next page):

```
import random
def extractAlignment(S,x,y):
  costS = 12
  costT = 13
  costID = 1
  optimaledits = []
  optimalpath = [[0 for j in range(len(y)+1)] for i in range(len(x)+1)]
  i = len(x)
  j = len(y)
  while i > 0 or j > 0:
    optimalpath[i][j] = 1
    # build cost array of tuples = (symbol, cost)
    cost = []
    cost.append(("s", S[i-1][j-1] + (costS if x[i-1] != y[j-1] else 0)))
    cost.append(("i", S[i][j-1] + costID))
    cost.append(("d", S[i-1][j] + costID))
    if j > 1 and i > 1:
      cost.append(("t", S[i-2][j-2] + costT
              + (0 if (x[i-1] == y[j-2]) else 1)
              + (0 if (y[j-1] == x[i-2]) else 1)))
    # the current optimal value is the minimum of cost
    optval = S[i][j]

    # choose one of the edits at random that have the same cost
    # as the current optimal value
    opt = random.choice([c[0] for c in cost if c[1] == optval])

    # if the characters in both strings were the same (and it was a sub),
    # then insert "|" (no-op) at the beginning of the list
    if (opt == "s" and x[i-1] == y[j-1]):
      optimaledits.insert(0, "|")

    # otherwise insert the operation
    else: optimaledits.insert(0,opt)

    # update the indicies to the selected subproblem
    if opt == "s":
      i -= 1
      j -= 1
    elif opt == "i":
      j -= 1
    elif opt == "d":
      i -= 1
    elif opt == "t":
      optimaledits.insert(0,opt)
      i -= 2
      j -= 2
  return optimaledits, optimalpath
```

commonSubstrings: This implementation of commonSubstrings() simply scans through the list of edit operations *a*, and looks for the no-op character "|". Once this is found it adds the corresponding character in X (which is the same in Y) to the last substring in an array of common substrings. For each sequential "|" in *a*, the same thing is done, until an edit operation in *a* is found which is not a no-op ("|"), then the algorithm adds a new empty string to the list of common substrings. Now, if another no-op sequence is found, the characters will be added to this new empty string. The algorithm will add all common substrings to the array, and then return a filtering of this array by length L.
python 3.6:

```
def commonSubstrings(x,L,a):
  substr = [""]
  for i in range(len(a)):
    if a[i] == "|":
      substr[-1] += x[i]
    else:
      substr.append("")
  return [s for s in substr if len(s) >= L]
```

(b) *Using asymptotic analysis, determine the running time of the call*

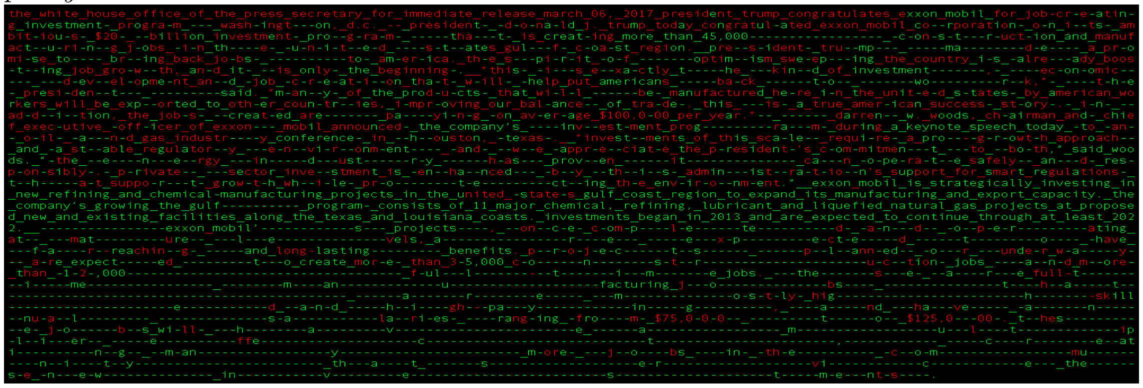
```
commonSubstrings(x, L, extractAlignment( alignStrings(x,y), x,y ) )
```

Justify your answer.

- The first function called within this call is *alignStrings* – the running time of alignStrings is $O(n_x n_y)$, as each entry in the (n_x x n_y) table is calculated in constant time (the process of picking the minimum of the previous optimal solutions). The cost of the setup before these calculations is $\Theta(n_x n_y) + \Theta(n_x) + \Theta(n_y)$ and thus does not contribute to the total running time.
- The second function call is *extractAlignment* – this takes $O(n_y + n_x)$ time in the worst case, which corresponds to deleting the input string X and inserting the target string Y. In terms of the cost table, this would require the algorithm to visit (and do a constant time operation on) each entry along the perimeter of the table (i.e. the longest path from S[i,j] to S[0,0]). In the best case, the input string and the target string are the same and *extractAlignment* takes $O(n_y)$ – the cost of visiting each entry along the diagonal (a series of no-ops).
- The final function call is *commonSubstrings* – this will take $\Theta(n_a)$, as it is a simple loop through the edit operations array (plus another loop through the created substrings array to filter by length).

Thus the total running time of the call would be $O(n_x n_y) + O(n_x + n_y)) = O(n_x n_y)$

(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.*



```
the_white_house_office_of_the_press_secretary_for_immediate_release_march_06,_2017_president_trump_congratulates_exxon_mobil_for_job-cr-e-atin-
g_investment-_progra-m_---_wash-ingt---on,_d.c._--_president-_-d-o-na-ld_j._trump_today_congratul-ated_exxon_mobil_co--rporation-_o-n_i--ts-_am
bit-iou-s-_$20-_--billion_investment-_pro--g-ra-m-_-----tha---t-_is_creat-ing_more_than_45,000-----------------_-c-on-s-t---r-uct-ion_and_manuf
act--u-ri-n--g_j-obs_-i-n_th----e-_-u-n-i-t--e-d_----s-t--ates_gul---f-_c-oa-st_region.__pre--s-ident-_tru--mp---_-----ma-------d-e----_a_pr-o
mi-se_to----_br--ing_back_jo-bs-_----------to-_am-er-ica._th-e_s---pi-r-it_-o-f_-------optim--ism_swe-ep---ing_the_country_i-s_-alre---ady_boos
-t--ing_job_gro-w--th,_an-d_it-_--is_only-_the_beginning-.__“this-_-i---s_e--xa-ctly_t-----he_---kin--d_of_investment-------,-_----ec-on-omic--
---_---d-ev--el-opme-nt_an--d_-job_-c-r-e-at-i--on_tha-t-_w-ill-_-help_put_americans_-----ba-ck_------t-o--_-------wo---------r--k,”-_----t-h-e
-_presi-den--t---_---------said._“m-an--y-_of_the_prod-u-cts-_that_wi-l-l_------be-_manufactured_he-re_i-n_the_unit-e-d_s-tates-_by_american_wo
rkers_will_be_exp--orted_to_oth-er_coun-tr--ies,_i-mpr-oving_our_bal-ance--_of_tra-de-._this_---is-_a_true_amer-ican_success-_st-ory-.-_i-n-_--
ad-d--i--tion,_the_job-s-_--creat-ed_are-------_pa---yi-n-g_-on_av-er-age_$100,0-00_per_year.”--_------_darren--_w._woods,_ch-airman_and-_chie
f_exec-utive_-off-icer_of_exxon---_mobil_announced-_the_company’s_----inv--est-ment_prog----------ra---m-_during_a_keynote_speech_today_-to_-an-
-_o-il-_-a---n-d_gas_industr----y_conference-_in_--h--ouston,_-texas-.__“invest--ments_of_this_sca-le--_requi-re-_a_pro----g-r-owt-h_approach--
_and_-a_st--able_regulator--y_---e-n--vi-r--onm-ent----_-and-_--w--e_-appr-e--ciat-e_the_p-resident-’s_c-om-mitmen---t_---to_--bo-th,”_said_woo
ds._“-the_--e----n---e--rgy_---in-----d---ust------r-y_-----h-as---_prov--en_------it--------------_ca---n_-o-pe-ra-t--e_safely--_an---d-_res-
p-on-sibly-.-_p-rivate--_---sector_inve--stment_is_-en--ha--nced---_-b--y_--th--i--s-_admin---ist--ra-t-io--n’s_support_for_smart_regulations-_
t--h-----a-t_suppo-r---t-_grow-t-h_wh--i-le-_pr-o--------t-e---------ct--ing_th-e_env-ir-o--nm-ent.”__exxon_mobil_is_strategically_investing_in
_new_refining_and_chemical-manufacturing_projects_in_the_united_-state-s_gulf_coast_region_to_expand_its_manufacturing_and_export_capacity._the
_company’s_growing_the_gulf----------_program-_consists_of_11_major_chemical,_refining,_lubricant_and_liquefied_natural_gas_projects_at_propose
d_new_and_existing_facilities_along_the_texas_and_louisiana_coasts._investments_began_in_2013_and_are_expected_to_continue_through_at_least_202
2.__-----------------exxon_mobil’------------s---_projects----,_--on--c-e-_c-om-p----l-e-----te----------d-_-a-n---d--_-o--p-e-r---------ating_
at--_---mat---------ure--_---l---e-------------vels,_a-----------------r----e---_-----e---x-p---------e-ct-e-----d_------t---------o----_-have_
---f-a----r--reachin--g-_-----and_long-lasting---------_benefits._p--r-o-j-e-c-----t--s------_------p--l--ann-ed--_-o---r_---unde-r_w-a-----y-
--_a-re_expect-----ed_---------t---o_create_mor-e-_than_3-5,000_c-o------n-------s-t--r-------------------------u-c--tion-_jobs_----a--n-d_m--ore-
_than_-1-2-,000-------------------------------------_f-ul---l---------t------i--m-------e_jobs._---the-------s---e-_--a----r---e_full-t-------
--i-----me---------------_--------m----an-------------u---------------------facturing_j---o--------------bs----_-----------------t---h--a-----t--
------------------------------------_------a-------r--------e-----_---m--------------o-s-t-ly-_hig--------------------------h--------skill
----------------------e-----------d_--a-n-d_----h--i---gh--pa---y------------in----g----------,_----a-----nd_---ha---ve-----_-a-------n-----
--nu-a--l---------_--------------s-a------------la--ri-es-_----rang-ing_-fro----m-_$75,0-0-0--_----------t------o--_$125,0---00-._t--hes---------
--e-_j-o------b--s_wi-ll-_---h--------a---------v--------------------e_------a---------------------_m---------------------u---l----t-------------ip
-l--i---er--_-----e----------ffe----------------------c---------------------------t-----------------------------,-----------_----c----r---------e--at
i----------n--g--_--m-an------------------y-----------------------------_m-ore-_---j--o----bs_----in-_-th-e-----------_-c--o-m------------------mu------
----n--i----t--y--------------------------_th---a----t_-------s------------------------------e-r-----------------------vi-------------c-------------e---_the------
s-e_-n---e-w---------------_in------------v---------e----------------s-----------s-----------------------t-----m-e---nt-s----.
```

This is the input string (x.txt) formatted so that green characters are those which were kept via no-ops, and the red characters are characters that were deleted. Dashes represent insertions, and underscores represent spaces. It is immediately apparent that a large portion of the text was identical, however here are the common substrings of length at least 9:

- ”manufacturing ”
- ” gulf coast ”
- ”the company's ”
- ” a keynote speech today ”
- ”conference”
- ”,” said woods. ””
- ”.” exxon”
- ”mobil is strategically investing in new refining and chemical-manufacturing projects in the u”
- ” gulf coast region to expand its manufacturing and export capacity. the companys growing the gulf”
- ” consists of 11 major chemical, refining, lubricant and liquefied natural gas projects at proposed new and existing facilities along the texas and louisiana coasts. investments began in 2013 and are expected to continue through at least 2022. ”
- ” to create ”
- ”facturing ”

This would classify as a violation of the honor policy, as the news article quoted the press release without citing the source.

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:

```
Pell(n):
  if n == 0 { return 0 }
  else if n == 1 { return 1 }
  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.* [1]

```
MemPell(n) {
  if n == 0 { return 0 } else if n == 1 { return 1 }
  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.*

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

i. If MemPell(n-1) is the left subtree, and MemPell(n-2) is the right, then the computation would look like a post-order traversal because it computes all of the (n-1) terms down to the base case, and then computes each corresponding (n-2) term before returning. The array is filled sequentially from the beginning to the end starting from the base cases.

ii. Because of the post-order traversal, each P[n] for n $\in$ [2..n] is calculated before anything else in the tree, which memoizes all pell numbers $\leq$ n. Thus there is no computation left to do for the rest of the tree. Thus the running time must be O(n).

[1] Ron briefly whines about Hermione's `L[n]=undefined` trick ("an unallocated array!"), but she point out that `MemLuc(n)` can simply be wrapped within a second function that first allocates an array of size n, initializes each entry to undefined, and then calls `MemLuc(n)` as given.

(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* [2] *is*

```
DynPell(n) :
  P[0] = 0, P[1] = 1
  for i = 2 to 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).

T(n) = O(n), because the loop is executed n-2 times, each time doing a constant-time operation.
S(n) = O(n), as the array is exactly n elements.
This is exactly the same time and space cost as Hermoine's algorithm. The array is even filled in in the same order.

(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*

```
FasterPell(n) :
  a = 0, b = 1
  for i = 2 to n
    c = 2 * a + b
    a = b
    b = c
  end
  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.

error: c = 2 * a + b
correction: c = 2 * b + a
T(n)= O(n) - this is a simple loop doing three O(1) operations n-2 times
S(n)= O(1) - only three numbers are stored at any given time.

[2] Ron is now using Hermione's undefined array trick; assume he also uses her solution of wrapping this function within another that correctly allocates the array.

(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?)*

	Pell(n)	MemPell(n)	DynPell(n)	FasterPell(n)
Running Time	O(2^n)	O(n)	O(n)	O(n)
Space	O(1)	O(n)	O(n)	O(1)
Data Structs	Tree	Array/Tree	Array	None

MemPell improves upon Pell by memoizing results in the computation tree - all of the pell numbers are computed along the left subtree. This is an improvement in time at the cost of space. DynPell does not improve upon MemPell in time or space. FasterPell is an improvement on DynPell/MemPell because it uses the same method of calculation without memoizing the intermediate results - this is like only using the last 3 rows in the napsack problem.

(e) *(5 pts extra credit) 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.*

First 5 digits of $P_{0828} = P_{828}$: 14916

Number of years for good ol' ronny boy's algorithm: 2^{828}ns = 5.672*10^{232} years (according to Wolfram alpha)

Real time according to unix: 0m0.240s

References

1. CLRS