Problem Set 3
CSE 373 Fall 2015
Due October 26 2015

Notes on Grading

• You can write “I don’t know” for any question and receive 25% credit. You can take this option for any numbered problem, but not for part of a problem. For example, you can answer 3.1 and write “I don’t know” for 3.2, but you can’t write part of the solution for 3.2 and then write “I don’t know” for the rest.

• You get a 10% bonus for typing your homework. You are encouraged to use \LaTeX. You must type your entire homework to receive the bonus. The 10% bonus does not apply to problems answered with “I don’t know.”

Important Note

All the problems on this homework ask for dynamic programs.

To answer a problem:

1. Briefly explain the recursive substructure of the problem.

2. Give the rule for filling in the dynamic programming table.

3. Give the asymptotic running time for the dynamic program.

4. Give the asymptotic space required for the dynamic program.

Then pick one of the problems and write out a complete implementation (in pseudo-code) for that algorithm.
1 Fairly splitting an array

Given an array $A$ of positive integers and an integer $k$, devise a dynamic program to split it into $k$ arrays, $A_1, \ldots, A_k$, as “fairly” as possible. Note that each $A_i$ must be a contiguous piece of $A$.

For this problem, a split would be perfectly fair if every array $A_i$ had the same sum, i.e. if for all $i$

$$\sum_j A_i[j] = C$$

where $C = \sum_j A[j]/k$.

Since a perfectly fair solution may not always be possible, your goal is to split $A$ so that

$$\max_i \left| C - \sum_j A_i[j] \right|$$

is minimized.

Solution  Note that, in the optimal solution, the arrays $A_1, \ldots, A_{k-1}$ can be chosen to be an optimally fair split of the portion of $A$ that they cover.

This gives the following recursive structure of the problem.

Let $T[i,j]$ be the cost of optimally splitting the array $A[0], \ldots, A[i-1]$ into $j$ arrays. Then

$$T[i,j] = \begin{cases} \left| \sum_{\ell=0}^{i-1} A[\ell] - C \right| & \text{if } j = 1 \\ \min_{t \in [0,i]} \max \left( T[t,j-1], \left| \sum_{\ell=t}^{i-1} A[\ell] - C \right| \right) & \text{if } j > 1 \end{cases}$$

The running time of this dynamic program is $O(n^2 k)$.

The space required is $O(n)$.
Algorithm 1 Fairly Splitting array A into contiguous pieces

1: function FairSplit(A, k)
2:   T ← new int[|A|][k + 1]
3:   for i=0 to (|A| − 1) do
4:     T[i][1] = \left|\sum_{\ell=0}^{i-1} A[\ell] - C\right|
5:   end for
6:   for j=2 to k do
7:     for i=0 to |A| − 1 do
8:       T[i][j] = \min_{t \in [0, i]} \max_{t \in [1, j-1]} \left( T[t,j-1], \left|\sum_{\ell=t}^{i-1} A[\ell] - C\right| \right)
9:     end for
10:   end for
11: return T[|A| − 1][k]
12: end function

2 Breaking a string into words

Suppose you are given a string s of letters with no spaces, and you would like to break it into English words, or at least things that look like they might be words.

You have a function q that takes a string as input and tells you a score indicating how much the string looks like an English word. Higher scores are more English-like.

Devise a dynamic program for breaking s into words w_1, \ldots, w_k, such that \sum_i q(w_i) is maximized. Note that k is not a parameter of the problem – you can break s into any number of words.

Solution If the last word is w_k, then w_1, \ldots, w_{k-1} must be an optimal decomposition of s with w_k removed.

Let T[i] be the quality of the optimal split for s_1s_2\cdots s_i. Then

\[ T[i] = \begin{cases} 
q(s_1) & \text{if } i = 1 
\max_{t \in [1, i-1]} T[t] + q(s_{t+1} \cdots s_i) & \text{if } i > 1
\end{cases} \]

The running time is O(n^2). Space is O(n).
Algorithm 2 Quality of optimal split of string $s$.

1: function OptimalSplit($s$)
2:  $T$ ← new int[$|s| + 1$]
3:  $T[1] = q(s_1)$
4:  for $i = 2$ to $|S|$ do
5:      $T[i] = \max_{\ell \in [1, i-1]} (T[\ell] + q(s_{\ell+1} \cdots s_i))$
6:  end for
7:  return $T[|s|]$
8: end function

3 Finding big squares

Show a dynamic program that, given an $n \times m$ matrix $A$ of 0s and 1s, finds the largest square region of $A$ that is all 1s.

Solution  If there is a square of size $k$ at position $(i, j)$, then there are squares of size at least $k - 1$ at positions $(i - 1, j), (i, j - 1), and (i - 1, j - 1)$.

Let $T[i, j]$ be the size of the largest square whose lower right corner is at position $(i, j)$. Then

$$T[i, j] = \begin{cases} A[i, j] & \text{if } i = 0 \text{ or } j = 0 \\ 1 + \min (T[i-1, j], T[i, j-1], T[i-1, j-1]) & \text{if } i > 0 \text{ and } j > 0 \end{cases}$$

Running time: $O(nm)$. Space: If we work by rows, we only need the previous row. Likewise if we work columnwise, so space is $O(\min(n, m))$. 


Algorithm 3 Size of the largest square of all 1s, space $O(mn)$.

1: function BiggestSquareInefficient($A, n, m$) $\triangleright$ n rows, m columns
2:   $T \leftarrow \text{new int}[n][m]$
3:   for $i=0$ to $n-1$ do
4:     $T[i][0] = A[i][0]$
5:   end for
6:   for $j=0$ to $m-1$ do
7:     $T[0][j] = A[0][j]$
8:   end for
9:   for $i=1$ to $n-1$ do
10:      for $j=1$ to $m-1$ do
11:         if $A[i][j]$ then
12:            $T[i][j] = 1 + \min(T[i-1,j], T[i,j-1], T[i-1,j-1])$
13:         else
14:            $T[i][j] = 0$
15:         end if
16:      end for
17:   end for
18:   return $\max_{i \in [0,n-1], j \in [0,m-1]} (T[i][j])$
19: end function
Algorithm 4 Size of the largest square of all 1s, space $O(\min(n,m))$

1: function BiggestSquareSpaceEfficient($A, n, m$) \triangleright n rows, m columns \triangleright
Assuming $m < n$
2: \hspace{1em} $T_{prev}$ ← new int[$m$]
3: \hspace{1em} biggestSquareSize ← 0
4: \hspace{1em} for $j=0$ to $m-1$ do
5: \hspace{2em} $T_{prev}[j] = A[0][j]$
6: \hspace{1em} end for
7: \hspace{1em} $T_{current}$ ← new int[$m$]
8: \hspace{1em} for $i=1$ to $n-1$ do
9: \hspace{2em} $T_{current}[0] = A[i][0]$
10: \hspace{2em} for $j=1$ to $m-1$ do
11: \hspace{3em} if $A[i][j]$ then
12: \hspace{4em} $T_{current}[j] =$
13: \hspace{5em} $1 + \min(T_{prev}[i-1,j], T_{prev}[i,j-1], T_{prev}[i-1,j-1])$
14: \hspace{4em} biggestSquareSize = max($T_{current}[j], biggestSquareSize$)
15: \hspace{3em} else
16: \hspace{4em} $T_{current}[j] = 0$
17: \hspace{3em} end if
18: \hspace{2em} end for
19: \hspace{1em} $T_{prev}$ ← $T_{current}$
20: end for
21: \hspace{1em} return biggestSquareSize
22: end function

Grading: half credit for finding squares of size a power of two.

4 Matched parentheses

A string $s$ of parentheses ("(" and ")") and brackets ("[" and "]") is matched if it is of one of the following forms:

- the empty string
- $(x)$, where $x$ is matched
- $[x]$, where $x$ is matched
- $xy$ where $x$ and $y$ are both matched.

Describe a dynamic program to determine whether a string $s$ is matched.
The description of the problem defines the dynamic program.

\[ P[i][j] \] stores if the sub string between \( i \) and \( j \) is matched. Then

\[
P[i][j] = \begin{cases} 
1 & \text{if } i > = j \\
\lor & \text{if } s[i] == ' ( ' \land s[j] == ')' \\
P[i+1][j-1] & \text{if } s[i] == ' [ ' \land s[j] == ']' \\
\lor \ell \in [i,j-1](P[i][i+\ell] \land P[i+\ell+1][j]) & \text{otherwise}
\end{cases}
\]

The running time is \( O(n^3) \). Space is \( O(n^2) \).

**Algorithm 5** Matching Parenthesis.

```plaintext
1: function OPTIMAL_SPLIT(s)
2: \hspace{1em} P ← new int[|s|][|s|]
3: \hspace{1em} for i=1 to |s|−1 do
4: \hspace{2em} for i=0 to |s|−1 do
5: \hspace{3em} if i > = j then P[i][j] = 1 \hfill \triangleright to take care of overlaps
6: \hspace{2em} end if
7: \hspace{1em} end for
8: \hspace{1em} end for
9: \hspace{1em} for j=1 to |s|−1 do
10: \hspace{2em} for i=0 to |s| do \hfill \triangleright sub string length
11: \hspace{3em} if (i + j) > (|s| − 1) then
12: \hspace{4em} Break out of the i loop
13: \hspace{3em} end if
14: \hspace{3em} Result ← 0
15: \hspace{3em} if (s[i] == ' ( ' \&\& (s[i + j] == ')' ) then
16: \hspace{4em} Result = Result \lor P[i+1][i+j−1]
17: \hspace{3em} else if (s[i] == ' [ ' \&\& (s[i + j] == ']' ) then
18: \hspace{4em} [i + 1][i + j − 1]
19: \hspace{3em} end if
20: \hspace{3em} Result = Result \lor (\lor \ell \in [i,i+j−1](P[i][i+\ell] \&\& P[i+\ell+1][i+j]))
21: \hspace{3em} P[i][i + j] = Result
22: \hspace{1em} end for
23: \hspace{1em} end for
24: \hspace{1em} return P[0][|s|−1]
25: end function
```
5 Correcting typos

Give a dynamic program that, given a string of parentheses and brackets, computes the edit distance to the nearest matched string.

Solution This is very similar to the previous problem. The cost of the edit distance is minimized over the three possible choices.

Let \( C[i][j] \) store the minimum cost of editing the sub string between \( i \) and \( j \) to ensure the parenthesis are all matched. Assume the cost of changing any one character into any other character is 1. Then

\[
C[i][j] = \min \begin{cases} 
0 \text{ if } i \geq j \\
C[i + 1][j - 1] + 0 & \text{if } s[i] == '(' \land s[j] == ')' \\
C[i + 1][j - 1] + 0 & \text{if } s[i] == '[' \land s[j] == ']' \\
C[i + 1][j - 1] + 1 & \text{if } s[i] == '(' \lor s[j] == ')' \\
C[i + 1][j - 1] + 1 & \text{if } s[i] == '[' \lor s[j] == ']' \\
C[i + 1][j - 1] + 2 & \text{otherwise} \\
\min_{\ell \in [1, j - i - 1]} (C[i][i + \ell] + C[i + \ell + 1][j]) & \text{for xy case}
\end{cases}
\]

The running time is \( O(n^3) \). Space is \( O(n^2) \).
Algorithm 6 Edit Distance.

1: function MinimalEditDistance(s)
2:     C ← new int[|s|][|s|]
3:     for i=1 to |s| − 1 do
4:         for i=0 to |s| − 1 do
5:             if i >= j then C[i][j] = 0  # to take care of overlaps
6:         end if
7:     end for
8:     end for
9:     for j=1 to |s| − 1 do
10:        for i=0 to |s| do  # sub string length
11:            if (i + j) > (|s| − 1) then
12:                Break out of the i loop
13:            end if
14:            parensCost ← cost of changing s[i] to ‘(’ and s[i+j] to ‘)’  # if necessary
15:            bracketsCost ← cost of changing s[i] to ‘[’ and s[i+j] to ‘]’  # if necessary
16:            xCost = P[i+1][i+j-1]
17:            xyCost = min_{ℓ∈[i,i+j−1]}(C[i][i+ℓ]&&C[i+ℓ+1][i+j])
18:            C[i][i+j] = min((parensCost + xCost), (bracketsCost + xCost), xyCost)
19:        end for
20:    end for
21:    return C[0][|s| − 1]
22: end function