CMSC 441 Project 1: Dynamic Programming

Alex McCaslin, Brendan Waters

Table of Contents

Algorithm Description:

Given an array of letters and the first and last indices of a sub-array, this algorithm will determine an optimal line folding to match the most valid pairs within the sub-array.

Description of the important variables throughout the algorithm:

- result - The maximum number of pairs that can be matched on letters[i...j]
- resultList - The pairs in the maximal folding
- n - Refers to the length of the line
- i - Refers to the beginning of the sub-array
- j - Refers to the end of the sub-array

Line-Folding Algorithm:

Looking at Code Example I below and starting with the first line in the code, OPT(i, j) we see that whenever the recursive function is called, the function takes in the first indice of the sub-array and the last indice of the sub-array. We start out by setting the result to 0. We later take the maximum over all sub-results to get the final result. At line 3, the for loop starts at j-5, since elements paired together must have a distance greater than 4 between them. Looping down from j-5 to i we attempt to pair j with all possible elements of this sub-array to find the optimal solution.

Inside the for-loop on line 4, we check whether or not the letters at position t and j in the array are a valid match. If the letters are not a combination of {H, G} or {W,T} then (t, j) is not a pair in the maximal folding and thus OPT(i, j) with respect to t is equal to OPT(i, j-1). Thus this problem is reduced to a subproblem of size (j - i) - 1. If the letters are a valid pair, then (t, j) is a pair in the maximal folding S with respect to t and thus OPT(i, j)=OPT(i, t-1)+OPT(t+1, j-1)+1. Then similarly we have reduced this problem into two smaller problems of size (n-2) / 2. After checking whether or not the pair is valid and after finding the sub-results, we check if the sub-result is higher than the current result. This allows us to find the most optimal solution for all possible pairs (t, j).

Code Example I - Algorithm Without Maximal Line Folding Pairs:

```
OPT(i, j)

    result = 0

    #No t within 5 of j can pair with j, so we skip these
    for t=j-5 down to i

        #If the pair doesn't match, then j is not in the solution
        If not validPair(letters[t], letters[j])
            subResult = OPT(i, j - 1)
        #If the pair does match, it is in solution and other pairs are found in subproblems
        else
            subResult = OPT(i, t-1) + OPT(t + 1, j - 1) + 1

        #Finds the maximum result over all solutions
        If subResult > result
            result = subResult


    Return result
```

Code Example II below, is very similar to Code Example I, but this one additionally returns the pairs of the optimal line folding. Python allows us to return multiple variables from a function, so we use this to our advantage and return the list of pairs found within the current subproblem OPT(i, j). This allows us to find both the result, and the set of pairs it took to achieve this result. For example, on lines (6, 8, 9, 10) we now have variables named subresultList, subresultList1, and subresultList2. These variables store the results from the smaller subproblems and if (t, j) is a pair we add it to the list.

Code Example II - Algorithm With Maximal Line Folding Pairs:

```
OPT(i, j)

    result = 0
    resultList = []

    #No t within 5 of j can pair with j, so we skip these
    For t = j-5 down to i

        #If the pair doesn't match, then j is not in the solution
        if not validPair(letters[t], letters[j]):
                    subresult, subresultList = OPT(i, j-1)

        #If the pair does match, it is in solution and other pairs are found in subproblems
        else:
                    subresult1, subresultList1 = OPT(i, t-1)
                    subresult2, subresultList2 = OPT(t+1, j-1)

                    subresult =  subresult1 + subresult2 + 1
                    subresultList = subresultList1 + subresultList2 + [(t, j)]

        if subresult > result:
                    result = subresult
                    resultList = subresultList

    return result, resultList
```

Lastly, we have Code Example III below. This is the final code produced that takes the result total from Code Example I, but instead uses memoization in the form of a matrix in order to prevent redundant recursive calculations. With a non-memoized version, our calculation times for an array of just 20 letters was taking minutes to compute. Now, on lines (5, 7, 9, 11, 13, 14, 16, and 18) we store the results of OPT(i, j) and if we ever come across another recursive call asking for OPT(i, j) we use that instead of making a new recursive call.

Code Example III - Memoized Algorithm without Maximal Line Folding Pairs

```
#Global array has solutions[i][j]=OPT(i, j). It is n x n and has all elements initialized to -1
def OPT(i, j):
        result = 0

        #No t within 5 of j can pair with j, so we skip these
        for t from j-5 down to i:

                #If the pair doesn't match, send in the initial array minus the last index
                if not validPair(letters[t], letters[j]):

                                if solutions[i][j-1] == -1:
                                        subresult = OPT(i, j-1)
                                        solutions[i][j-1] = subresult
                                else:
                                        subresult = solutions[i][j-1]

                #If the pair does match, get the results of the two sub-arrays
                else:
                                if solutions[i][t-1] == -1:
                                        subresult1 = OPT(i, t-1)
                                        solutions[i][t-1] = subresult1
                                else:
                                        subresult1 = solutions[i][t-1]

                                if solutions[t+1][j-1] == -1:
                                        subresult2 = OPT(t+1, j-1)
                                        solutions[t+1][j-1] = subresult2
                                else:
                                        subresult2 = solutions[t+1][j-1]

                                subresult =  subresult1 + subresult2 + 1

                if subresult > result:
                                result = subresult

    return result
```

Proof the Problem has Optimal Substructure:

Let |S| be the number of pairs in a maximal folding S of line A from i to j
In other words, |S| = OPT(i, j)

If $j <= i + 4$ then $OPT(i, j) = 0$

If not, let T iterate down over the values $i <= t < j - 4$ with t being some particular value of T

Let S_t be a solution which has t in some pair. Then $|S_t| = OPT_t(i, j)$

Case 1:

- A[t] and A[j] are not a valid pair
- Then $(t, j) \notin S_t$
- Then t is paired with some $m < j$
- Then if j is paired with some n we have $n < t < m < j$
- Then since $n < t$, a later iteration of T will reach this pair possibility and we need not consider it here
- Then we can say j belongs to no pair in the current S_t
- Thus $OPT_t(i, j) = OPT_t(i, j-1)$

Case 2:

- A[t] and A[j] are a valid pair
- Then $(t, j) \in S_t$
- Then for some other pair $(m, n) \in S_t$ we have either $m < n < t < j$ or $t < m < n < j$
- So we have just found 1 pair and know other possible pairs are in A[i… t-1] or A[t+1...j-1]
- Thus $OPT_t(i, j) = OPT_t(i, t-1) + OPT_t(t+1, j-1) + 1$

Thus since we are iterating over all valid values of T we have

$$|S| = \max \{ |S_t| : t \in T \}$$

Then since |S| is given by some $|S_t|$ which is solved by solving smaller problems of the same type, this problem has an optimal substructure.

Runtime Analysis of the Non-Memoized Algorithm:

The runtime of Code Example I is given by

$$\sum_{1}^{n-4} (T(n-1) \; or \; 2T(\tfrac{n-2}{2})) \; + \; O(1)$$

which is equal to

$$(n-4)(T(n-1) \; or \; 2T(\tfrac{n-2}{2})) \; + \; O(1)$$

Since each recursive call contains a loop of length n-4 which makes either one recursive call of size n-1 or two recursive calls of size $\frac{n-2}{2}$ based on if a pair has been found or not.

Case 1:

- $T(n) = (n-4) * T(n-1) + O(1)$
- Then we can construct a recursion tree with the value 1 at each node with depth n and width of the lowest level is (n-4)!
- Then we can make an educated guess that T(n) is bounded above by n*(n-4)!

Case 2:

- $T(n) = (n-4) * 2T(\frac{n-2}{2}) + O(1)$
- Then we can construct a recursion tree with the value 1 at each node with depth logn and the width of the lowest level is $2^{log(n)}(n-4)! \; = \; n(n-4)!$
- Then we can make an educated guess that T(n) is bounded above by $nlog(n) * (n-4)!$

And since $n * (n-4)! \; < \; nlog(n) * (n-4)! \; < \; nlog(n) * n!$

In either case T(n) is bounded above by $nlog(n) * n!$

Proof by Substitution:

Assume T(m) = $O(mlog(m) * m!)$ for some $m < n$

Then $T(m) <= cmlog(m) * m!$

Case 1:

- Let $m \; = \; n-1$
- Then $T(n-1) <= c(n-1)log(n-1) * (n-1)!$
- Subtracting a lower order term to save us work later and making a tighter bound we get
- $T(n-1) <= c(n-1)log(n-1) * (n-1)! - 1$
- Then substituting this into T(n) we get
- $T(n) = (n-4)T(n-1) + 1$
- $<= (n-4)(c(n-1)log(n-1) * (n-1)! - 1) + 1$
- $= \; c(n-4)log(n-1) * (n-1)(n-1)!$
- $<= cnlog(n) * n!$
- Thus for case 1 $T(n) = O(nlog(n) * n!)$

Case 2:

- Let $m = \frac{n-2}{2}$
- Then $T(\frac{n-2}{2}) <= c(\frac{n-2}{2})log(\frac{n-2}{2})(\frac{n-2}{2})!$
- Subtracting a lower order term to save us work later and making a tighter bound we get
- $T(\frac{n-2}{2}) <= c(\frac{n-2}{2})log(\frac{n-2}{2})(\frac{n-2}{2})! - 1$
- Then substituting this into T(n) we get
- $T(n) = (n-4)2T(\frac{n-2}{2}) + 1$
- $<= (n-4)(2c((\frac{n-2}{2})log((\frac{n-2}{2})((\frac{n-2}{2})! - 1) + 1$
- $= c(n-4)log(n) * (n-2)(\frac{n-2}{2})!$
- $<= cnlog(n) * n!$
- Then for case 2 $T(n) = O(nlog(n) * n!)$

Runtime Analysis of the Non-Memoized Algorithm:

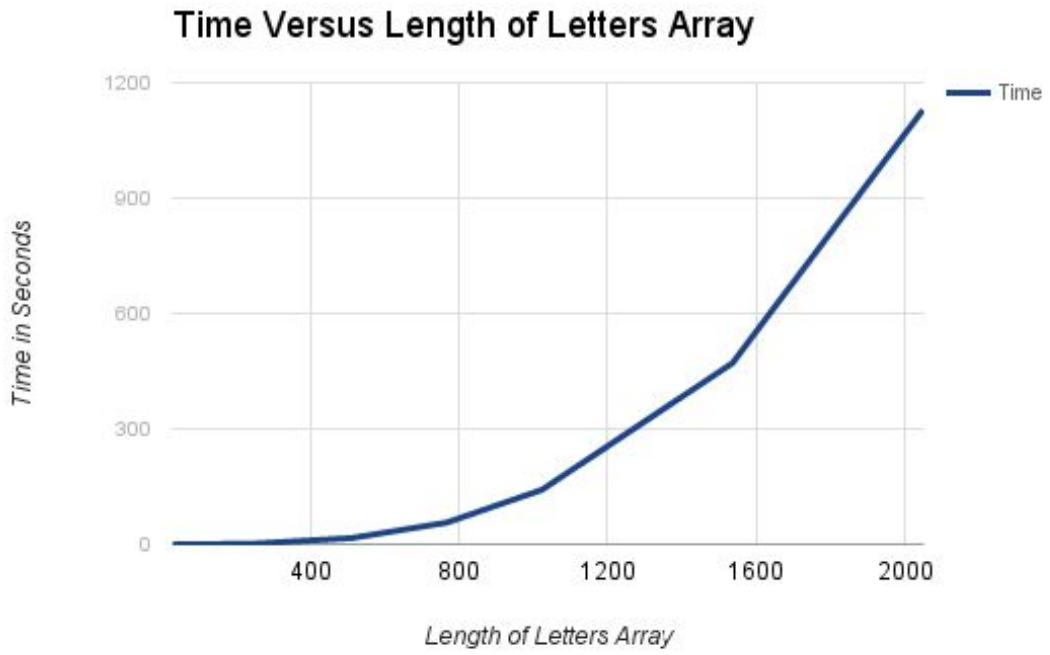
The runtime of Code Example III (the memoized version), on the other hand, is O(n^3)

To fill in the n x n matrix of solutions to subproblems OPT(i, j) we need at most n^2 recursive calls now that each particular OPT(i, j) need only be evaluated a single time.
Since each recursive call contains a loop of length n-4 then we have
$T(n) <= c(n-4) * n^2 <= cn^3$
Therefore T(n)=O(n^3)

This matches with our empirical runtime data of Code Example 3 given below

Time in Seconds Versus N Letters for Memoized Algorithm

N	32	64	128	256	512	768	1024	1536	2048
Time	.00239	.02569	.19793	1.8775	15.986	56.293	141.54	471.01	1129.0



Plugging in these values into a least-squares best fit, we get a cubic result of
$(1.3476 * 10^{-8})x^3 - (7.86 * 10^{-6})x^2 + (.0021)x - (.2743)$