

## Homework Assignment #2

Due: Feb 26

Student: Shuo Yang

1. Algorithm**Function** *FindClosePair*( $A[1 : n]$ )
$$\text{distance} := \text{ComputeCloseDistance}(A[1 : n])$$

$$(x, y) := \text{ComputeClosePair}(A[1 : n], \text{distance})$$
**Function** *ComputeCloseDistance*( $A[1 : n]$ )

Linearly scan the input array  $A[1 : n]$  from left to right, find the minimum and maximum numbers, put them into variables  $\text{min}$  and  $\text{max}$  respectively, and let

$$\text{distance} := \lfloor (\text{max} - \text{min}) / (n - 1) \rfloor.$$
**Function** *ComputeClosePair*( $A[1 : n], \text{distance}$ )

(1) Linearly scan the input array  $A$  from left to right, find the minimum and maximum numbers, put them into variables  $\text{min}$  and  $\text{max}$  respectively.

(2) Divide the  $n$  elements into  $\lfloor n/5 \rfloor$  groups of 5 elements, and  $\leq 1$  group of  $< 5$  elements.

(3) Find the median of each group using selection sort.

(4) Recursively find the median  $\text{med}$  of the  $\lfloor n/5 \rfloor$  medians found in step (3).

(5) Partition the input array  $A$  around  $\text{med}$  from step (4) into two subarrays  $A_{\text{low}}$  and  $A_{\text{high}}$  such that  $A_{\text{low}}$  contains all elements  $\leq \text{med}$  and  $A_{\text{high}}$  contains all elements  $\geq \text{med}$ .

(6) **if**  $\text{med} - \text{min} \leq \text{distance}$

**return** ( $\text{med}, \text{min}$ )

**if**  $\text{max} - \text{med} \leq \text{distance}$

**return** ( $\text{max}, \text{med}$ )

**if**  $n$  is odd

**if**  $\text{max} - \text{med} \geq \text{med} - \text{min}$

$(x, y) := \text{ComputeClosePair}(A_{\text{low}}, \text{distance})$

**else**

$(x, y) := \text{ComputeClosePair}(A_{\text{high}}, \text{distance})$

**else** //  $n$  is even

**if**  $\text{max} - \text{med} - \text{distance} \geq \text{med} - \text{min}$  //  $\text{med}$  is lower median

$(x, y) := \text{ComputeClosePair}(A_{\text{low}}, \text{distance})$

**else**

$(x, y) := \text{ComputeClosePair}(A_{\text{high}}, \text{distance})$

Correctness

**Lemma1:** Given an unsorted array  $A$  of  $n$  distinct numbers, a close pair always exists.

*Proof.* Prove by contradiction, that is, no such a close pair exists, this means for any pair  $(x, y)$  where  $x > y$ , we have:

$$x - y > \frac{1}{n-1}(\text{max} - \text{min}) \quad (1)$$

Suppose we order numbers in the array  $A$  in ascending order as a sequence:  $a_1, a_2, \dots, a_n$  where  $a_i > a_j$  if  $i > j$ . Let  $\text{distance} = \frac{1}{n-1}(\text{max} - \text{min})$ , according to the assumption, we have:

$$\begin{aligned}
a_n - a_{n-1} &> distance \\
a_{n-1} - a_{n-2} &> distance \\
&\dots \\
a_3 - a_2 &> distance \\
a_2 - a_1 &> distance
\end{aligned} \tag{2}$$

Summing the above equations together produces:

$$\begin{aligned}
a_n - a_1 &> distance \times (n - 1) \\
&= \frac{1}{n - 1} (max - min) \times (n - 1) \\
&= max - min
\end{aligned} \tag{3}$$

Since we have sorted the array  $A$  in ascending order, this means that  $a_n = max$  and  $a_1 = min$ , therefore the above equation says that  $max - min > max - min$ , this is clearly a contradiction, thus the lemma1 must be true.  $\square$

**Lemma2:** Given an unsorted array  $A$  of  $n$  distinct numbers partitioned around its median into two subarrays  $A_{low}$  and  $A_{high}$ .  $A_{low}$  contains all elements  $\leq$  median and  $A_{high}$  contains all elements  $\geq$  median. Let the maximum element be  $max$ , minimum element be  $min$  and median be  $med$ . A close pair must exist in  $A_{low}$  if  $max - med \geq med - min$  (when  $n$  is odd) or  $max - med - distance \geq med - min$  (when  $n$  is even), and in  $A_{high}$  otherwise.

*Proof.* Prove by contradiction. If  $max - med \geq med - min$  (when  $n$  is odd) or  $max - med - distance \geq med - min$  (when  $n$  is even), assume that  $A_{low}$  does not contain any close pairs. There are two cases to consider:

(a)  $n$  is odd.

In this case,  $A_{low}$  contains  $\frac{n-1}{2} + 1 = \frac{n+1}{2}$  elements, including the  $med$  itself. Applying the same method used in proving Lemma1, we sort  $A_{low}$  as:  $a_0, a_1, \dots, a_{(n+1)/2}$  in ascending order. According to the assumption, we must have,

$$\begin{aligned}
a_{\frac{n+1}{2}} - a_{\frac{n+1}{2}-1} &> distance \\
&\dots \\
a_3 - a_2 &> distance \\
a_2 - a_1 &> distance
\end{aligned} \tag{4}$$

Summing the above equations together produces:

$$\begin{aligned}
a_{\frac{n+1}{2}} - a_1 &> distance \times \left(\frac{n+1}{2} - 1\right) \\
&= \frac{1}{n-1} (max - min) \times \left(\frac{n+1}{2} - 1\right) \\
&= \frac{1}{n-1} (max - min) \times \frac{n-1}{2} \\
&= \frac{max - min}{2}
\end{aligned} \tag{5}$$

Since  $A_{low}$  is sorted in ascending order,  $a_{\frac{n+1}{2}} = med$  and  $a_1 = min$ . Thus,

$$\begin{aligned} med - min &> \frac{max - min}{2} \\ med &> \frac{max + min}{2} \end{aligned} \tag{6}$$

And because  $max - med \geq med - min$ , we have  $med \leq \frac{max+min}{2}$ . But we have just proved that  $med > \frac{max+min}{2}$ , clearly it is a contradiction. Thus, there must exist a close pair in  $A_{low}$ .

(b)  $n$  is even.

In this case, there are two medians, left median and right median. Assume that we pick the low median.  $A_{low}$  contains  $\frac{n}{2}$  elements, including the  $med$  itself. Applying the same method used in proving Lemma1, we sort  $A_{low}$  as:  $a_1, a_2, \dots, a_{n/2}$  in ascending order. According to the assumption, we must have,

$$\begin{aligned} a_{\frac{n}{2}} - a_{\frac{n}{2}-1} &> distance \\ &\dots \\ a_3 - a_2 &> distance \\ a_2 - a_1 &> distance \end{aligned} \tag{7}$$

Summing the above equations together produces:

$$\begin{aligned} a_{\frac{n}{2}} - a_1 &> distance \times \left(\frac{n}{2} - 1\right) \\ &= \frac{1}{n-1}(max - min) \times \left(\frac{n}{2} - 1\right) \\ &= \frac{max - min}{2} \times \frac{n-2}{n-1} \end{aligned} \tag{8}$$

Since  $A_{low}$  is sorted in ascending order,  $a_{\frac{n}{2}} = med$  and  $a_1 = min$ , we have:

$$\begin{aligned} med - min &> \frac{max - min}{2} \times \frac{n-2}{n-1} \\ med &> \frac{max + min}{2} - \frac{max - min}{2(n-1)} \end{aligned} \tag{9}$$

Substituting  $\frac{max-min}{n-1}$  with  $distance$ , we have:

$$med > \frac{max + min}{2} - \frac{distance}{2} \tag{10}$$

And because  $max - med - distance \geq med - min$ , we have  $med \leq \frac{max+min}{2} - \frac{distance}{2}$ . But we have just proved that  $med > \frac{max+min}{2} - \frac{distance}{2}$ , clearly it is a contradiction. Thus, there must exist a close pair in  $A_{low}$ .

The proof for the case when  $max - med < med - min$  (when  $n$  is odd) or  $max - med - distance < med - min$  (when  $n$  is even) is symmetrical.  $\square$

So combining Lemma1 and Lemma2, we can conclude that our algorithm can always find a pair of close elements.

### **Run time analysis**

The sub-routine *ComputeCloseDistance* takes  $\Theta(n)$  time. Let the total run time for *ComputeClosePair* be  $T(n)$  where  $n$  is the number of elements in the input array  $A$ . Step (1) takes  $\Theta(n)$  time, step (2) takes  $\Theta(n)$  time, step (3) takes  $\Theta(n)$  time, step (4) takes  $T(\lfloor n/5 \rfloor)$  time, step (5) takes  $\Theta(n)$  time, step (6) takes  $T(n/2)$  time because for each recursive call, we reduce the size of the input for the sub-problem into half, that is, either recurse on lower partition or higher partition. Thus, we have:

$$\begin{aligned} T(n) &= T(n/5) + T(n/2) + \Theta(n) \\ &= \Theta(n) \end{aligned} \tag{11}$$

By the master theorem, the total run time for *FindClosePair* is  $T(n) + \Theta(n) = \Theta(n)$ .

## **2. Algorithm**

The algorithm works as the following: (1) Compute the index of the median:  $m = \lfloor n/2 \rfloor$ .

(2) Find the  $m_{th}$  smallest element *median*, which is also the median of the input array.

(3) Partition the input array around *median* into lower subarray and higher array.

(4) recursively call on the lower sub-arrays, find it  $(k/2)_{th}$  quantiles, then output *median*, then recursively call on the lower sub-arrays, find it  $(k/2)_{th}$  quantiles. Return when  $k == 1$ .

### **Correctness**

The algorithm first finds the median of the input array, which is also the median of the  $k_{th}$  quantiles. Then partition the array around the median. So now we can solve two sub-problems recursively, each contains at most  $(k-1)/2$  order statistics of the original input. Thus the recursion will eventually hit the base case where  $k == 1$  and return back the  $k_{th}$  quantiles.

### **Run time analysis**

Step (1) takes constant time, step (2) takes  $\Theta(n)$  time, step (3) takes  $\Theta(n)$  time. The recursion has the depth of  $\log k$ .

Thus the recurrence equation is:

$$T(n, k) = 2T\left(\frac{n}{2}, \frac{k}{2}\right) + \Theta(n) \tag{12}$$

By the master theorem, the run time is  $\Theta(n \log k)$ .

## **3. Algorithm**

**Function** *FindSmallestInMerge*( $A[1 : m], B[1 : n], k$ )

$i := \lfloor k/2 \rfloor$

$j := \lceil k/2 \rceil$

**if**  $k == 1$  // base case

**return**  $\min(A[1], B[1])$

**if**  $A[i] > B[j]$

$s_k := \text{FindSmallestInMerge}(A[1 : i], B[j + 1 : n], i)$

**else if**  $A[i] < B[j]$

$s_k := \text{FindSmallestInMerge}(A[i + 1 : m], B[1 : j], j)$

### **Correctness**

If  $A[i] > B[j]$ , where  $i := \lfloor k/2 \rfloor$  and  $j := \lceil k/2 \rceil$ , then in the merged array, there can be at most  $k - 2$  elements that are  $< B[j]$ , that is,  $A[1 : i - 1]$  and  $B[1 : j - 1]$ . So we must have the  $k_{th}$  smallest element  $s_k > B[j]$ . On the other hand, there are at least  $k - 1$  elements that are  $< A[i]$  in the merged array, that is,  $A[1 : i - 1]$  and  $B[1 : j]$ , thus we have  $s_k \leq A[i]$ .

The above analysis shows that  $s_k$  can only appear in the subarray  $B[j + 1 : n]$  or  $A[1 : i]$ . Moreover, we have thrown out  $j$  elements that  $< s_k$ , thus, the problem is reduced to finding the  $i_{th}$  smallest element in the merged array of  $B[j + 1 : n]$  and  $A[1 : i]$ .

In the case when  $A[i] < B[j]$ , the argument is symmetric.

With the above argument, we can conclude that our algorithm can find the  $k_{th}$  smallest element in the merge of two sorted arrays.

### Run time analysis

In each recursive call, we reduce the problem into half of its original size, and other operations executes in constant time, thus the recurrence equation is:

$$T(k) = T\left(\frac{k}{2}\right) + \Theta(1) \quad (13)$$

By the master theorem, the run time is  $\Theta(\log k)$ .

#### 4. characterize the recursive structure of an optimal solution

For input string  $S = s_1 s_2 \cdots s_n$ , there are three ways an longest palindromic subsequence (LPS) could start or end:

*Case 1:* LPS starts at  $s_1$  and ends at  $s_n$ . In this situation,  $s_1 = s_n$ . The optimal solution must be the LPS of substring  $s_2 \cdots s_{n-1}$  prefixed with  $s_1$  and postfixed with  $s_n$ . If not, prefixing  $s_1$  and postfixing  $s_n$  to the LCS of  $s_2 \cdots s_{n-1}$  would yield a longer solution, which is a contradiction.

*Case 2:* LPS does not start at  $s_1$ . Then optimal solution must be LPS of  $s_2 \cdots s_n$ .

*Case 3:* LPS does not end at  $s_n$ . Then optimal solution must be LPS of  $s_1 \cdots s_{n-1}$ .

### Derive a recurrence equation for the value of an optimal solution

The recursive subproblem that arises is one of computing an LPS over a substring of the input string  $S$ . Thus the subproblem can be specified by the start and end index of the substring.

So let,

$$L(i, j) := \text{the length of LPS of substring } s_i \cdots s_j \quad (14)$$

So, based on the three cases, the recurrence equation can be describe as:

$$L(i, j) := \begin{cases} \max \begin{cases} L(i+1, j-1) + 2 & // \text{ case 1} \\ L(i+1, j) & // \text{ case 2} \\ L(i, j-1) & // \text{ case 3} \end{cases} & i \geq 1 \text{ and } j \geq 1 \text{ and } i < j \\ 1 & i = j \\ 0 & i > j \end{cases} \quad (15)$$

The solution value for the original problem is  $L(1, n)$ .

### Evaluate the recurrence bottom-up in a table

We evaluate  $L(i, j)$  in a table  $L[0 : n, 0 : n]$ . In general, the entry  $(i, j)$  depends on the three entries  $(i, j - 1)$ ,  $(i + 1, j - 1)$  and  $(i + 1, j)$ . So we fill in the table in diagonal-major order: first all the entries where  $j - i = 0$ , then all the entries where  $j - i = 1$ , etc.

**Function** *EvaluateLPS*( $S, L, n$ )

```
// fill in values at diagonal
for  $k := 1$  to  $n$ 
     $L[k, k] := 1$ 

// fill in values that rely only on diagonal values
for  $i := 1$  to  $n - 1$ 
     $j := i + 1$ 
    if  $S[i] == S[j]$ 
         $L[i, j] = 2$ 
    else
         $L[i, j] = \max(L[i, j - 1], L[i + 1, j])$ 

// fill in remaining values
for  $k := 2$  to  $n - 1$ 
    for  $i := 1$  to  $n - k$ 
         $j := i + k$ 
         $L[i, j] = \max(L[i, j - 1], L[i + 1, j], L[i + 1, j - 1])$ 
```

*Run time analysis:* First and second **for** loop both take  $\Theta(n)$  time. The third loop takes  $O(n^2)$  time. So the total run time is  $O(n^2)$ .

### Recover an optimal solution from the table of solution values

The function *RecoverLPS* recovers a LPS of input string  $S$  that starts at position  $i$ , ends at position  $j$ , given the precomputed table  $L$ .

**Function** *RecoverLPS*( $S, L, i, j$ )

```
if  $i > j$ 
    return
if  $i == j$ 
    output  $S[i]$ 
    return
if  $S[i] == S[j]$  and  $L[i, j] == L[i + 1, j - 1] + 2$  // case 1
    RecoverLCS( $S, L, i + 1, j - 1$ )
    output  $S[i]$ 
else if  $L[i, j] == L[i + 1, j]$  // case 2
    RecoverLPS( $S, L, i + 1, j$ )
else  $L[i, j] == L[i, j - 1]$  // case 3
    RecoverLPS( $S, L, i, j - 1$ )
```

*Run time analysis:* each call increments  $i$  or decrements  $j$  by 1 and spends  $\Theta(1)$  time. Since we start with  $i = 1$  and  $j = n$ , it takes total  $\Theta(n)$  time.

## 5. characterize the recursive structure of an optimal solution

Let the maximum independent set be  $MIS$ . For a tree rooted at some node  $r$ , then there are two cases for  $MIS$ : *case 1:*  $r$  is in  $MIS$ . Then  $MIS$  cannot contain the children of  $r$ .  $MIS$  must be the

sum of the optimal solutions rooted at each of the grandchildren of  $r$ , together with  $r$ .

*case 2:*  $r$  is not in  $MIS$ . Then  $MIS$  must be the sum of optimal solutions root at each of the children of  $r$ .

### Derive a recurrence equation for the value of an optimal solution

The recursive subproblem that arises is one of computing an  $MIS$  over a sub-tree of the input tree. Thus the problem can be specified by the node of the tree. Let,

$$L(v) := \text{total weights of the maximum independent set of the sub-tree rooted at } v \quad (16)$$

Our goal is to compute  $L(r)$ . So, based on the three cases, the recurrence equation can be described as:

$$L(v) := \begin{cases} \omega(v) + \sum_{\text{grandchildren of } v} L(u) \\ \max \left\{ \begin{array}{l} \omega(v) + \sum_{\text{grandchildren of } v} L(u) \\ \sum_{\text{children of } v} L(u) \end{array} \right. \\ \omega(v) \end{cases} \quad \begin{array}{l} \\ \\ v \text{ is a leaf} \end{array} \quad (17)$$

### Evaluate the recurrence bottom-up in a table

We evaluate the table  $L$  in a bottom up fashion recursively give a sub-tree rooted at  $v$ .

**Function** EvaluateMIS( $L, v$ )

**if**  $v.isLeaf$  // base case

$L(v) := \omega(v)$

**return**  $L(v)$

$sumChildren := 0$

$sumGrandchildren := 0$

**for each** child  $u$  **in**  $v.children$

$sumChildren := sumChildren + EvaluateMIS(L, u)$

**if**  $v.hasGrandchildren$

**for each** grandchild  $u$  **in**  $v.grandchildren$

$sumGrandchildren := sumGrandchildren + EvaluateMIS(L, u)$

$L(v) := \max(sumChildren, \omega(v) + sumGrandchildren)$

**return**  $L(v)$

**else** //  $v$  does not have grandchildren

$L(v) := \max(sumChildren, \omega(v))$

**return**  $L(v)$

To compute the  $MIS$  for the original input tree rooted at  $r$ , we just need to call  $EvaluteMIS(L, r)$ .

*Run time analysis:* there are total  $n$  entries to be filled in. To fill each entry  $L(v)$ , the algorithm only looks at the children and grandchildren of  $v$ . So each vertex  $u$  is accessed only for three times: when evaluating itself, evaluating its parent and when evaluating its grandparent. Since each vertex is evaluated only for a constant time, thus the total run time is  $\Theta(n)$ .

### Recover an optimal solution from the table of solution values

**Function** *RecoverMIS*( $L, v$ )

```
    sumChildren := 0
    sumGrandchildren := 0
    if  $v.isLeaf$ 
        output  $v$ 
        return
    for each child  $u$  in  $v.children$ 
        sumChildren := sumChildren +  $L(u)$ 
    if  $v.hasGrandchildren$ 
        for each grandchild  $u$  in  $v.grandchildren$ 
            sumGrandchildren := sumGrandchildren +  $L(u)$ 
        if  $L(v) == \omega(v) + \textit{sumGrandchildren}$ 
            output  $v$ 
            for each grandchild  $u$  in  $v.grandchildren$ 
                RecoverMIS( $L, u$ )
        else
            for each child  $u$  in  $v.children$ 
                RecoverMIS( $L, u$ )
    else
        if  $L(v) == \textit{sumChildren}$ 
            for each child  $u$  in  $v.children$ 
                RecoverMIS( $L, u$ )
        else
            output  $v$ 
            return
```

The procedure recovers a MIS of the sub-tree rooted at vertex  $v$ . So to recover a MIS for the original input tree rooted at  $r$ , just call *RecoverMIS*( $L, r$ ).

*Run time analysis:* similar with the analysis of the evaluation function, this time each table entry  $L(v)$  will be looked up for at most three times, that is, when looking at its grandparent, its parent and itself. There are total  $n$  entries, thus the run time is  $\Theta(n)$ .

### 6. characterize the recursive structure of an optimal solution

After having dialed a prefix of the input digits  $A$ , the two figures can be only in constant number of configurations, which is,  $\frac{12!}{(12-2)!} = 132$ . So the optimal solution for dialing  $n$  digits must be the optimal solution for dialing the  $1_{st}$  digit combining the optimal solution for dialing the remaining  $n - 1$  digits. (So the optimal solution for dialing the remaining  $n - 1$  digits must be the optimal solution for dialing the  $2_{nd}$  combining the optimal solution for dialing the remaining  $n - 2$  digits). This defines the recursive structure of an optimal solution.

### Derive a recurrence equation for the value of an optimal solution

Assume right finger's initial position is #, and left finger's initial position is \*.



Given a  $n$ -digit telephone number  $A = a_1a_2 \cdots a_n$ , lets define:

$$\begin{aligned}
& \text{distance}(i, j) = \text{Euclidean distance between digit } i \text{ and digit } j. \\
& \text{and,} \\
& \text{distance}(i, \#) = \text{Euclidean distance between digit } i \text{ and } \# \\
& \text{and,} \\
& \text{distance}(i, *) = \text{Euclidean distance between digit } i \text{ and } *
\end{aligned} \tag{18}$$

and,  $L(i, j) :=$  minimum cost to type digit  $i$  with left finger, and having right finger at digit  $j$  ( $j = 0$  if right finger is at initial position  $\#$ ),

and,  $R(i, j) :=$  minimum cost to type digit  $i$  with right finger, and having left finger at digit  $j$  ( $j = 0$  if left finger is at initial position  $*$ ).

When  $i = 0$ , both left and right fingers are at their initial positions.

So, according the recursive structure of the optimal solution, we can compute  $L(i + 1, j)$  based on  $L(i, j)$  and  $R(i, j)$ . This gives us the recurrence equation as follows:

$$L(i + 1, j) := \begin{cases} L(i, j) + \text{distance}(i, i + 1) & 0 < i < n \text{ and } 0 \leq j < n \text{ and } i > j \\ \min_{0 \leq j' < i} (R(i, j') + \text{distance}(j', i + 1)) & i = j \text{ and } 0 \leq j' < n \text{ and } i > j' \\ \text{distance}(1, *) & i = 0, j = 0 \\ 0 & i < 0, j < 0 \end{cases} \tag{19}$$

In the above equation,  $R(i, j') + \text{distance}(j', i + 1)$  means when the right finger is also at digit- $i$ , and with left finger pointing at digit- $j'$ , so in this case, we need to also consider moving left finger from  $j'$  to  $i + 1$ , which also yields a valid  $L(i + 1, j)$ . So we take the minimum of these two possible cases. Also note we consider the case when  $i = j$  separately because in this case  $L(i, j)$  is not valid because having two fingers typing one digit twice does not produce the minimum distance.

The recurrence equation for  $R(i, j)$  is similar:

$$R(i + 1, j) := \begin{cases} R(i, j) + \text{distance}(i, i + 1) & 0 < i < n \text{ and } 0 \leq j < n \text{ and } i > j \\ \min_{0 \leq j' < i} (L(i, j') + \text{distance}(j', i + 1)) & i = j \text{ and } 0 \leq j' < n \text{ and } i > j' \\ \text{distance}(1, \#) & i = 0, j = 0 \\ 0 & i < 0, j < 0 \end{cases} \tag{20}$$

### Evaluate the recurrence bottom-up in a table

Based on the recurrence equation, the following function is used to evaluate the recurrence for the table  $L[0 : n, 0 : n]$  and  $R[0 : n, 0 : n]$ .

**Function** EvaluateTFD(A, L, R)

```

 $L(1, 0) := \text{distance}(1, *)$ 
 $R(1, 0) := \text{distance}(1, \#)$ 
for  $i := 2$  to  $n$ 
  for  $j := 0$  to  $i - 2$ 
     $L(i, j) := L(i - 1, j) + \text{distance}(i - 1, i)$ 
     $R(i, j) := R(i - 1, j) + \text{distance}(i - 1, i)$ 
   $j := i - 1$ 
   $\text{minR} := \infty$ 
  for  $j' := 0$  to  $j - 1$ 
     $\text{minR} = \min(R(i - 1, j'), \text{minR})$ 
   $L(i - 1, j) := \text{minR}$ 
   $\text{minL} := \infty$ 
  for  $j' := 0$  to  $j - 1$ 
     $\text{minL} = \min(L(i - 1, j'), \text{minL})$ 
   $R(i - 1, j) := \text{minL}$ 

```

*Run time:* the run time for filling  $L$  and  $R$  takes  $O(n^2)$  time.

**Recover an optimal solution from the table of solution values**

To get the minimum distance traveled, just take the minimum one between  $L(n, x)$  and  $R(n, y)$  where  $0 \leq x, y < n$ . It takes  $\Theta(n)$  time.