Assignment 9

Benjamin Jakubowski

July 28, 2016

1. Rod cutting with cost to cut

Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c. The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. We give a dynamic-programming algorithm to solve this modified problem.

function ROD_CUTTING_WITH_CUT_COST(p, n, c)
  let $r[0\,..\,n]$ be a new array
  $r[0] = 0$ ▷ No cut, so no cut cost incurred
  for $j = 1$ to n **do**
    $q = -\infty$
    for $i = 1$ to j **do**
      if $i < j$ **then**
        $q = \max(q, p[i] + r[j - i] - c)$ ▷ $-c$ since one additional cut
      else ▷ Then $i = j$, so no additional cut made
        $q = \max(q, p[j])$ ▷ Since $p[j] = p[i] + r[j - i]$
    $r[j] = q$
    return r[n]

2. $O(n^2)$-time algorithm for increasing subsequence

We give an $O(n^2)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers. The basic idea behind the algorithm is to find the LCS shared by A and the array A', where A' is the sorted array of unique values in A. Note we need to ensure the values in A' are unique, otherwise we would potentially find the longest monotonically non-decreasing subsequence in A.

1: **function** INCREASING_SUBSEQUENCE(A)
2: $A' = \text{sorted}(A)$ ▷ $O(n \lg n)$
3: $A' =$`delete_duplicates`(A) ▷ $O(n)$. Needed for *monotonically increasing*
4: `LCS`(A, A') ▷ $O(n^2)$ by CRLS page 394
5: `Print-LCS`(A, A') ▷ $O(n)$ by CRLS page 395

This is an $O(n^2)$-time algorithm since it is dominated by the `LCS` step on line 4.

3. $O(n \lg n)$-TIME ALGORITHM FOR INCREASING SUBSEQUENCE

We give an $O(n \lg n)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers (stored in array $A = [a_1, \cdots, a_n]$). Before presenting the algorithm, we give a brief outline:

- The algorithm does a single pass through the n elements of A.
- Each element is inserted into a Red Black tree C, unless a duplicate value has already been inserted into the tree.
- For the j^{th} element a_j:
  - The algorithm then computes and saves the following satellite data:
    - The longest monotonically increasing subsequence in $A[1 : j-1]$, inclusive, to which we can append a_j and maintain the property "monotonically increasing".
    - The length of the resulting subsequence
  - Next, the algorithm trims the tree to maintain a key invariant, namely:

    An in-order traversal of C produces an ordering of the j keys $A = [a_1, \cdots, a_j]$

    $$k_1 < k_2 < \cdots < k_j$$

    such that

    1. $k_1.length < k_2.length < \cdots < k_j.length$

    and

    2. $k_j.length =$ Maximum length of monotonically increasing subsequence

- Finally, the algorithm returns $k.max().subsequence$, a maximum-length monotonically increasing subsequence of A.

We present the detailed algorithm before arguing that it is (i) correct, and (ii) runs in $O(n \lg n)$ time.

```
1: function LINEARITHMIC_MAX_LENGTH_INCREASING_SUBSEQUENCE(A)
2:     let C be a new Red-Black Tree
3:     for i in 1 . . n do
4:         if a_i not already found in C then
5:             C.insert(a_i)
6:             pred = C.predecessor(a_i)
7:             if pred is Null then
8:                 a_i.subsequence = a_i
```

```
9:              a_i.length = 1
10:         else
11:             a_i.subsequence = pred.subsequence.append(a_i)
12:             a_i.length = pred.length + 1
13:         succ = C.successor(a_i)
14:         while succ not Null and not a_i.length < succ.length do
15:             C.delete(succ)
16:             succ = C.successor(a_i)
17:     return C.maximum().subsequence
```

To show the algorithm is correct, we must show the loop invariant:

An in-order traversal of C produces an ordering of the j keys $A = [a_1, \cdots, a_j]$

$$k_1 < k_2 < \cdots < k_j$$

such that

1. $k_1.length < k_2.length < \cdots < k_j.length$

and

2. $k_j.length =$ Maximum length of monotonically increasing subsequences of $A[1\,..\,j]$

actually holds.

Initialization The invariant trivially holds after inserting a_1.
Maintenance Now assume the invariant holds prior to re-entering the `for` loop with $i = j$. Then there are two cases:

- a_j already in C: Then the loop doesn't modify the tree. To show this maintains the loop invariant, note it clearly maintains invariant condition 1 (which held prior to re-entering the loop). Moreover, note prior to entering the loop with $i = j$

  $$\begin{aligned} & C.\text{predecessor}(a_j).\text{length} < a_j.\text{length} \\ \implies \quad & C.\text{predecessor}(a_j).\text{length} + 1 \leq a_j.\text{length} \end{aligned}$$

  Thus if we were to modify the tree by changing the subsequence ending with a_j, it would not increase the length of this a_j-terminated subsequence; hence, the maximum-length subsequence of of $A[1\,..\,j-1]$ is still maximal, and the invariant is maintained by not modifying the tree.

- a_j is not already in C: Then the loop inserts a_j into the tree- say at position m in the ordering of the keys in the tree:

  $$k_1 < \cdots < k_m = a_j < \cdots < k_\ell$$

  But then

- By construction lines 7-12 ensure $k_1.length < \cdots < k_m.length = a_j.length$.
- The while loop (lines 14-16) clearly maintains loop invariant for $k_m = a_j < \cdots < k_\ell$, and ensures $k_\ell.length$ is maximal.

Thus, in either case the invariant is maintained.
Termination After exiting the loop, $k_{\max}$.subsequence is a longest monotonically increasing subsequence in A. Thus, the subsequence associated with the largest element in the RedBlack tree is a maximum-length monotonically increasing subsequence of A.

To analyze the runtime, we present the cost of each step in the algorithm. Note lines 4-13 the costs are presented in aggregate (over the loop), while lines 14-16 are not.

Line number(s)	Cost
3: Initialize red-black tree	$O(n)$
4: Check to see if a_i in tree	$nO(\lg n)$
5: Insert a_i in tree	$nO(\lg n)$
6: Find predecessor	$nO(\lg n)$
7-12: Compute satellite data	$nO(1)$
13: Find successor	$nO(1)$
14-16: Trim tree	$O(n)$, since there are at most $n-1$ deletions from the tree
17: Return maximum subsequence	$O(\lg n)$

Thus, overall run-time is $O(n \lg n)$.

4. Dynamic programming solution to neat printing problem

Consider the problem of neatly printing a paragraph with a monospaced font (all characters having the same width) on a printer. The input text is a sequence of n words of lengths $l_1, l_2, \cdots, l_n$, measured in characters. We want to print this paragraph neatly on a number of lines that hold a maximum of M characters each. Our criterion of "neatness" is as follows. If a given line contains words i through j, where $i \leq j$, and we leave exactly one space between words, the number of extra space characters at the end of the line is $M - j + i - \sum_{k=i}^{j} l_k$, which must be nonnegative so that the words fit on the line. We wish to minimize the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the ends of lines. We give a dynamic-programming algorithm to print a paragraph of n words neatly on a printer, and analyze the running time and space requirements of our algorithm.

Our basic strategy is as follows

- First, we compute the the cost of printing a line with words $l_i, \cdots, l_j$ for every pair i, j where $i \leq j$. We structure this line cost computation to handle the following edge cases:

- If the length of the line would exceed the maximum number of characters M, then this line should not be part of our optimal solution. Since we aim to minimize our cost function, we can encode this constraint by setting the cost of this candidate line to ∞.
- If $j = n$, then this is the last line in the paragraph. Since this line incurs no cost, we will set the cost to 0.
- Otherwise, the cost is the cube of the number of extra space characters at the end of the line.

- Once we've found these line costs, we proceed to compute the minimum cost of printing the first j words, starting with $j = 0$ and incrementing the word count until we've found the minimum for $j = n$. Note this step in the computation exploits optimal substructure of the problem.
- While determining the minimum costs, we additionally store the optimal line break locations. This will allow us to determine the optimal "neat" printing, and yield our final solution.

With this in mind, we proceed by presenting the algorithm. To make it more readable, we break out several helper functions. Similarly, to ease notation, we assume the words, maximum line length M, and number of words n are global.

```
function COMPUTE_CANDIDATE_LINE_COST(i, j)    ▷ i is first word in line, j is last
    extra_space = (M − j + i − Σ_{k=i}^{j} l_k)
    if extra_space < 0 then
        return ∞
    else if j = n then
        return 0
    else
        return (extra_space)^3

function CONSTRUCT_LINE_COST_TABLE()
    let line_cost[1 . . n, 1 . . n] be a new array
    for i in 1 to n do
        for j in i to n do
            line_cost[i, j] = compute_candidate_line_cost(i, j)
    return line_cost
```

Now we're almost ready to present the main algorithm. Before we do, let min_print_cost$[j]$ be the minimum cost to print the first j words. Then since the problem has optimal substructure, we can define this cost recursively:

$$\text{min\_print\_cost}[j] = \begin{cases} 0 \text{ if } j = 0 & \triangleright \text{ Base case} \\ \min_{1 \le i \le j} (\text{min\_print\_cost}[i-1] + \text{line\_cost}[i,j]) & \text{otherwise} \end{cases}$$

With this recursive definition in mind, we present the main algorithm, which neatly prints the n words:

```
function NEATLY_PRINT_WORDS()                    ▷ Again assume words are global
    line_cost[1 . . n, 1 . . n] = construct_line_cost_table()
    let breaks[0 . . n] be a new array (to store line break points)
    let min_print_cost[0 . . n] be a new array (to store minimum costs)
    min_print_cost[0] = 0
    for j in 1 to n do
        min_cost = ∞
        for i in 1 to j do
            cost_at_i = min_print_cost[i − 1] + line_cost[i, j]
            if cost_at_i < min_cost then
                min_cost = cost_at_i
                breaks[j] = i
    print_lines(breaks)
```

Note to print the lines we call one final helper function, print_lines, which takes an array of computed optimal line breaks and prints the corresponding lines in correct order (first to last):

```
function PRINT_LINES(breaks)
    let lines_to_print be a new list      ▷ We will insert lines to print at front of list
    last_word = n                                                  ▷ Last unprinted word
    next_break = breaks[last_word]        ▷ Next place to break with minimum cost
    while last_word != next_break do                ▷ Then not on first line to print
        lines_to_print.insert(" ".join(words from (next_break + 1) to last_word))
        last_word = next_break
        next_break = breaks[last_word]
    for line in lines_to_print do
        print line
```

Now we analyze the running time and space requirements of the algorithm. First, let's consider the space requirements:

Object	Space required
line_cost	$O(n^2)$
min_print_cost	$O(n)$
breaks	$O(n)$
line_to_print	$O(n)$

Hence the overall space required is $O(n^2)$. Now let's consider the time requirements.

Object	Time required
`compute_candidate_line_cost`	$O(1)$
`construct_line_cost_table`	$O(n^2)$
`neatly_print_words`	$O(n^2)^*$
`print_lines`	$O(n)$

*(Dominated by the double `for` loop and `construct_line_cost_table`)

5. Dynamic programming solution to edit distance

a. Algorithm

Given two sequences $x[1 \,..\, m]$ and $y[1 \,..\, n]$ and set of transformation-operation costs, the edit distance from x to y is the cost of the least expensive operation sequence that transforms x to y. We describe a dynamic-programming algorithm that finds the edit distance from x to y and prints an optimal operation sequence, and analyze the running time and space requirements of our algorithm.

First, note we use an array z to hold intermediate results, and indices i and j into x and z respectively. Additionally, we consider the following transformation operations, with costs specified in such a way that minimizing the objective eliminates any invalid edits:

- **Copy** a character from x to z by setting $z[j] = x[i]$ and then incrementing both i and j. This operation examines $x[i]$, and has cost

$$cost(Copy, i, j) = \begin{cases} \infty \text{ if } x[i] \neq y[j] \\ c \text{ otherwise (i.e. a constant copy cost)} \end{cases}$$

  We can implement this cost function as

  function COST(copy, i, j)
  if $1 \leq i \leq m, 1 \leq j \leq n$, and $x[i] = y[j]$ **then**
  return c
  else
  return ∞

- **Replace** a character from x by another character c, by setting $z[j] = c$, and then incrementing both i and j. This operation examines $x[i]$, and has cost

$$cost(Replace, i, j) = r \text{ (a constant replace cost- note we assume } c = y[j])$$

  We can implement this cost function as

  function COST(replace, i, j)
  if $1 \leq i \leq m$ and $1 \leq j \leq n$ **then**
  return c
  else
  return ∞

- **Delete** a character from x by incrementing i but leaving j alone. This operation examines $x[i]$, and has cost

$$cost(Delete, i, j) = d \text{ (a constant delete cost)}$$

  We can implement this cost function as

function COST(delete, i, j)
    if $1 \leq i \leq m$ **then**
        return d
    else
        return ∞

- **Insert** the character c into z by setting $z[j] = c$ and then incrementing j, but leaving i alone. This operation examines no characters of x, and has cost

$$cost(Insert, i, j) = s \text{ (a constant insert cost)}$$

We can implement this cost function as

function COST(insert, i, j)
    if $1 \leq j \leq n$ **then**
        return s
    else
        return ∞

- **Twiddle** (i.e., exchange) the next two characters by copying them from x to z but in the opposite order; we do so by setting $z[j] = x[i+1]$ and $z[j+1] = x[i]$ and then setting $i = i+2$ and $j = j+2$. This operation examines $x[i]$ and $x[i+1]$ and has cost

$$cost(Twiddle, i, j) = \begin{cases} \infty \text{ if } x[i] \neq y[j+1] \text{ or } x[i+1] \neq y[j] \\ t \text{ otherwise (i.e. a constant twiddle cost)} \end{cases}$$

We can implement this cost function as

function COST(twiddle, i, j)
    if x[i] = y[j+1], x[i+1] = y[j+1], $2 \leq i+1 \leq m$ and $2 \leq j+1 \leq n$ **then**
        return t
    else
        return ∞

- **Kill** the remainder of x by setting $i = m+1$. This operation examines all characters in x that have not yet been examined. This operation, if performed, must be the final operation- thus it has cost

$$cost(Kill, i, j) = \begin{cases} \infty \text{ if } j \neq n+1 \qquad \text{(haven't finished editing x to y)} \\ k \text{ otherwise (i.e. a constant kill cost)} \end{cases}$$

We can implement this cost function as

function COST(kill, i, j)

if j = n+1 and $1 \leq i \leq$ m **then**
    return k
else
    return ∞

With those cost functions specified, we now exploit the optimal substructure of this problem. Specifically, let

$$(x, z_0) \overset{E_1}{\to} (x, z_1) \overset{E_2}{\to} \cdots \overset{E_{k-1}}{\to} (x, z_{k-1}) \overset{E_k}{\to} (x, z_k = y)$$

be an optimal (i.e. cost-minimizing) sequence of edits from x to y. Then note

$$(x, z_0) \overset{E_1}{\to} (x, z_1) \overset{E_2}{\to} \cdots \overset{E_{k-1}}{\to} (x, z_{k-1})$$

is itself a cost-minimizing sequence of edits from x to z_{k-1} (by the obvious cut and paste argument). Moreover, note z_{k-1} is some prefix of y.

Now let $C[i, j]$ be the minimum edit cost to increment the pointers to the values i, j. Then (since we initialize with $i = 1$ and $j = 1$), we have $C[1, 1] = 0$. Additionally, note editing is complete when $j = n + 1$ and $i = m + 1$. Thus, we have

$$C[i,j] = \begin{cases} 0 \text{ if } i,j = 1 \\ \infty \text{ if } i \text{ or } j \leq 0 \qquad \triangleright \text{ To handle edge cases} \\ \min \begin{cases} C[i-1,j-1] + cost(Copy, i-1, j-1) \\ C[i-1,j-1] + cost(Replace, i-1, j-1) \\ C[i-1,j] + cost(Delete, i-1, j) \\ C[i,j-1] + cost(Insert, i, j) \\ C[i-2,j-2] + cost(Twiddle, i-2, j-2) \\ \text{if } j = n+1 \texttt{ and } i = m+1: \\ \qquad C[i-k,j] + cost(kill, i-k, j) \text{ for each } k \in \{1, \cdots, i-1\} \end{cases} \end{cases}$$

Now that we've defined the cost of each operation for each $(i, j) \in \{1, \cdots, m\} \times \{1, \cdots, n\}$, and have a recursive definition of $C[i, j]$, we are ready to implement the dynamic programming algorithm to find the minimum edit distance. Our approach is as follows

- First, we will compute $(m + 1) \times (n + 1)$ tables with costs for each operation for each pair of possible index values $(i, j) \in \{1, \cdots, m + 1\} \times \{1, \cdots, n + 1\}$.
- Second, we will compute minimum edit distance for each prefix subsequence $C[i, j]$, storing the operations used in a separate table $Op[i, j]$.
- Finally, we will follow the operation chain from $C[m + 1, n + 1]$ to complete the algorithm and print an optimal operation sequence.

Note throughout we assume x, y, z, m, C, and Op and n are global (or at least in visible), just to have cleaner notation and avoid having to pass these variables as function arguments. Additionally, we break out a number of helper functions to clarify the main algorithm `edit_distance`. First, we present the main algorithm:

Main algorithm `edit_distance`

function EDIT_DISTANCE()
`make_operation_cost_tables()`
Initialize $m+1 \times n+1$ array Op to store optimal operations.
Initialize $(m+3) \times (n+3)$ array C ▷ Need extra rows/cols for ∞ edge-cases
for $j = -1$ to $n+1$ **do**
C[-1, j] = ∞
C[0, j] = ∞
for $i = 1$ to $m+1$ **do**
C[i, -1] = ∞
C[i, 0] = ∞
C[1,1] = 0
Op[1,1] = Null
for $i = 1$ to $m+1$ **do** ▷ Now fill in tables
for $j = 1$ to $n+1$ **do**
if $i \neq 1$ and $j \neq 1$ **then**
C[i,j], Op[i,j] = `get_min_and_op`(i, j)
print "Edit distance between x and $y = C[m+1, n+1]$".
`follow_and_print_edit_path()`

Next, we present three helper procedures- `make_operation_cost_tables`, `get_min_and_op`, and `follow_and_print_edit_path`.

function MAKE_OPERATION_COST_TABLES()
Initialize $m+1 \times n+1$ arrays copy_cost, replace_cost, delete_cost, insert_cost, twiddle_cost, and kill_cost
for i in 1 to $m+1$ **do**
for j in 1 to $n+1$ **do**
copy_cost$[i, j]$ = cost(Copy i, j)
replace_cost$[i, j]$ = cost(Replace i, j)
delete_cost$[i, j]$ = cost(Delete i, j)
insert_cost$[i, j]$ = cost(Insert i, j)
twiddle_cost[i,j] = cost(Twiddle i, j)
kill_cost[i,j] = cost(Kill i, j)

function GET_MIN_AND_OP(i, j) ▷ Assume arrays C, S in parent scope
q = ∞
if $C[i-1, j-1] + cost(Copy, i-1, j-1) < q$ **then**

```
            q = C[i − 1, j − 1] + cost(Copy, i − 1, j − 1)
            best_op = Copy
            parent = (i − 1, j − 1)
        if C[i − 1, j − 1] + cost(Replace, i − 1, j − 1) < q then
            q = C[i − 1, j − 1] + cost(Replace, i − 1, j − 1)
            best_op = Replace
            parent = (i − 1, j − 1)
        if C[i − 1, j] + cost(Delete, i − 1, j) < q then
            q = C[i − 1, j] + cost(Delete, i − 1, j)
            best_op = Delete
            parent = (i − 1, j)
        if C[i, j − 1] + cost(Insert, i, j − 1) < q then
            q = C[i, j − 1] + cost(Insert, i, j − 1)
            best_op = Insert
            parent = (i, j − 1)
        if C[i − 2, j − 2] + cost(Twiddle, i − 2, j − 2) < q then
            q = C[i − 2, j − 2] + cost(Twiddle, i − 2, j − 2)
            best_op = Copy
            parent = (i − 2, j − 2)
        if j = n + 1 and i = m + 1 then
            for k = 1 to i − 1 do
                if C[i − k, j] + cost(Kill, i − k + 1, j) < q then
                    q = C[i − k, j] + cost(Kill, i − k + 1, j)
                    best_op = Kill
                    parent = (i − k, j)
        C[i, j] = q
        S[i, j] = [best_op, parent]          ▷ Save best_op and parent as nested array in S
    function FOLLOW_AND_PRINT_EDIT_PATH()
        Initialize edit_path as a list to store the edit path
        parent = (m+1, n + 1)
        while parent ≠ (1, 1) do
            edit_path.insert(S[parent][0])          ▷ best_op first element of S[parent]
            parent = S[parent][1])                  ▷ parents seconds element of S[parent]
        step_num = 1
        next_step = edit_path.head
        for step in edit_path do
            print step
```

Now that we've presented the algorithm, plus all necessary helper functions/procedures, we analyze the running time and space requirements.
First, space:

Object	Space required
`copy_cost`	$O(mn)$
`replace_cost`	$O(mn)$
`delete_cost`	$O(mn)$
`insert_cost`	$O(mn)$
`twiddle_cost`	$O(mn)$
`kill_cost`	$O(mn)$
`S`	$O(mn)$
`Op`	$O(mn)$

Next, time:

Procedure/statement	Time cost
`make_operation_cost_tables`	$O(mn)$
`get_min_cost_and_op`	$O(1)$, but called $O(mn)$ times
`follow_and_print_edit_path`	$O(m+n)$ (since at most $O(m+n)$ operations in paths)

Thus, the main algorithm `edit_distance` is $O(mn)$.

B. DNA SEQUENCE SIMILARITY

Finally, we explain how to cast the problem of finding an optimal DNA alignment as an edit distance problem using a subset of the transformation operations copy, replace, delete, insert, twiddle, and kill. First recall (from CLRS) that we align two sequences x and y consists of inserting spaces at arbitrary locations in the two sequences (including at either end) so that the resulting sequences x' and y' have the same length but do not have a space in the same position (i.e., for no position j are both $x'[j]$ and $y'[j]$ spaces). Then we assign a scores to each position. Position j receives a score as follows:

- $+1$ if $x'[j] = y'[j]$ and neither is a space
- -1 if $x'[j] \neq y'[j]$ and neither is a space
- -2 if either $x'[j]$ or $y'[j]$ is a space

Finally, we sum the scores at each position to obtain the score for the alignment.
We can cast this as an edit distance problem quite simply. Specifically,

- We view the problem is editing x to y.
- Next we drop the operations kill and twiddle. Note if we wanted to use an out-of-the-box implementation of the edit distance algorithm (that takes costs as arguments), we could simply set the cost of these operations to ∞.
- Next, we note that inserting a space in x (such that $x'[j]$ is a space) is equivalent to inserting $y[j]$ in for $z[j]$. Based on this observation, we set the insert cost to 2 (the cost of $x'[j]$ being a space).

- Similarly, we note that inserting a space in y (such that $y'[j]$ is a space) is equivalent to deleting $x[i]$. Based on this observation, we set the delete cost to 2 (the cost of $y'[j]$ being a space).
- Next, we note $x'[j] = y'[j]$ is equivalent to a copy operation. Since this indicates a positive match between the sequences x and y, we set the cost of the copy operation to -1 (such that a copy operations decreases the overall cost $C[i, j]$).
- Finally, we note $x'[j] \neq y'[j]$ is equivalent to a replace operation. Thus we set the cost of the replace operation to 1.

With these values set for operation costs, we can reduce sequence alignment to edit distance.