Notes on CS6363

Hanlin He

Tuesday October 4, 2016

Contents

1 Syllabus **1**

2 Basic **2**
2.1 What is an algorithm? 2
2.2 Input & Output 2
2.2.1 Algorithm 1 2
2.2.2 Algorithm 2 3
2.2.3 Algorithm 3 3
2.3 Can we do better? 3

3 Asymptotic Notation – big "O" notation **4**
3.1 Growth of Functions 4
3.2 big "O" notation 4
3.3 Asymptotic Relation's feature 4
3.4 Properties of $\log(n)$ 5
3.5 Something More 6

4 Series **7**
4.1 Some Definition 7
4.2 Some Theorem 7

5 Induction **7**
5.1 When to use? 7
5.2 Definition 8
5.3 Example 8
5.3.1 Good Induction: 8
5.3.2 Bad Induction: Prove all horses are the same color 8

6 Recursion (Divide & Conquer) **9**
6.1 Definition of Recursion: a Powerful type of reduction 9
6.2 Tower of Hanoi 9

7 Dynamic Programming **10**
7.1 Rod Cutting 10
7.1.1 Description of Problem 10
7.1.2 Analysis 10

7.1.3 Memoized Version . . . 11
7.1.4 Dynamic Programming Version to Solve RodCut . . . 11
7.2 Solving DP problem . . . 12
7.3 Longest Increasing Subsequence (LIS) . . . 12
7.3.1 Description of Problem . . . 12
7.3.2 Analysis . . . 13
7.3.3 Dynamic Programming Version to Solve LIS . . . 14
7.4 Longest Common Subsequence . . . 14
7.4.1 Description of Problem . . . 14
7.4.2 Analysis . . . 15
7.4.3 Dynamic Programming Version to Solve LCS . . . 16
7.5 Edit Distance . . . 17
7.5.1 Description of Problem . . . 17
7.5.2 Analysis . . . 18
7.5.3 Dynamic Programming Version to Solve Edit Distance Problem . . . 20

1 Syllabus

1. Asymptotic notation, recurrence.
2. Divide and Conquer.
3. Dynamic Programming.
4. Greedy Algorithm.
5. Graph Algorithm.
6. NPC

2 Basic

2.1 What is an algorithm?

Unambiguous, mechanically executable sequence of elementary operations.

There are certain types of algorithm:

Traditional (This courses main focus.)	Modern algorithm research
Deterministic	Randomized
Exact	Approximate
Off-line	On-line
Sequential	Parallel

2.2 Input & Output

View algorithm as a function with well defined inputs mapping to specific outputs. For example:

Input: $A[1...n]$ // Positive real number, distinct.

Output: $MAXA[i], 1 <= i <= n$.

2.2.1 Algorithm 1

Stupid way.

Algorithm 1 Stupid Find Max Algorithm

```
procedure FindMax
    for i = 1 to n do
        count = 0
        for j = 1 to n do
            if A[i] > A[j] then
                count = count + 1
            end if
        end for
        if count = n then
            return A[i]
        end if
    end for
end procedure
```

Analysis: Worst Case, n^2 comparison.

2.2.2 Algorithm 2

Sort & Find.

Algorithm 2 Sort & Find Max Algorithm

```
procedure FindMax
    Ā = sort(A)
    return Ā[n]
end procedure
```

Analysis: Worst Case, sorting takes $c\ n \log n$ time.

2.2.3 Algorithm 3

Dynamically store the biggest one.

Algorithm 3 Search & Find Max Algorithm

```
procedure FindMax
    current = 1
    for i = 2 to n do
        if A[i] > A[current] then
            current = i
        end if
    end for
    return A[current]
end procedure
```

2.3 Can we do better?

It depends on the operations allowed. For example the dropping the curtain and find the first appearing one.

3 Asymptotic Notation – big "O" notation

3.1 Growth of Functions

The growth of function in table 1 increase downwards.

Table 1: Function List

$\log_{10} n$	binary search
n	input
n^2	pairs
$10^{10}n^{10}$	
$1.000.1^n$	
2^n	Binary string of length n
$n!$	Permutation

Let $f(n)$, $g(n)$ be function.

3.2 big "O" notation

Definition 3.2.1. $f(n) = \mathcal{O}(g(n))$, if $\exists n_0 \in \mathbb{N}$, $c \in \mathbb{R}^+$, s.t. $\forall n \geq n_0$, $f(n) \leq c * g(n)$, and $\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty$, i.e. it is $\lim_{n\to\infty} \frac{f(n)}{g(n)} < k$, for some constant k.

Table 2 shows the basic definition of all the asymptotic notations.

Table 2: Definition for all Asymptotic Notation

$f(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)}$	relation
$\mathcal{O}(g(n))$	$\neq \infty$	$\leq$
$\Omega(g(n))$	$\neq \infty$	$\leq$
$\Theta(g(n))$	$= k > 0$	$=$
$o(g(n))$	$= 0$	$<$
$\omega(g(n))$	$= \infty$	$>$

3.3 Asymptotic Relation's feature

Theorem 3.3.1. *Multiplying by positive constant does NOT change asymptotic relations. i.e. if $f(n) = \mathcal{O}(g(n))$, then $100 * f(n) = \mathcal{O}(g(n))$.*

Proof: $f(n) = \mathcal{O}(g(n)) \Rightarrow \exists n_0 \exists c, \forall n \geq n_0, f(n) \leq c * g(n)$,

then, $\exists n_0 \exists c'$, s.t. $\forall n \geq n_0$, $100 * f(n) \leq c' * g(n) = 100c * g(n)$. □

Example:

$$C * 2^n = \Theta(2^n) \tag{1}$$
$$(C * 2)^n \neq \Theta(2^n) \tag{2}$$

Claim 3.3.2. *Show:* $2n\log(n) - 10n = \Theta(n\log(n))$

Proof: First show: $2n\log(n) - 10n = \mathcal{O}(n\log(n))$

For $n_0 = 1$, $c = 2$

$$2n\log(n) - 10n \leq 2n\log(n)$$

Now show: $2n\log(n) - 10n = \Omega(n\log(n))$

For $n_0 = 2^10$, $c = 1$,

$$\begin{aligned} 2n\log(n) - 10n &\geq n\log(n) + n\log(2^{10}) - 10n \\ &= n\log(n) + 10n - 10n \\ &= n\log(n) \end{aligned}$$

$n_0 = 1$ ($n_0 = 2^{10}$) means n is at least 1 (or 2^{10}). $\square$

Corollary 3.3.3. $\mathcal{O}(1)$ *means* ***Any Constant***.

Attention: *Asymptotic notation has limit. It is not applicable for all scenarios.*

3.4 Properties of $\log(n)$

Definition 3.4.1. $n = C^{\log_c n}$, $c > 1$, $\lg n = \log_2 n$, $\ln n = \log_e n$.

Corollary 3.4.2. $\forall a, b > 1$

$$\begin{aligned} \log_b(n) &= \frac{\log_a(n)}{\log_a(b)} \\ \log_b(n) &= \Theta(\log_a(n)) \end{aligned} \tag{3}$$

Corollary 3.4.3. $\forall a, b \in \mathbb{R}$

$$
\begin{aligned}
\log(a^n) &= n * \log(a) \\
\log(a * b) &= \log(a) + \log(b)) \\
a^{\log(b)} &= b^{\log(a)}
\end{aligned} \tag{4}
$$

Note: $\lg(n)$ is to n as n is to 2^n.

3.5 Something More

Theorem 3.5.1. *Let $f(n)$ be a polynomial function, then* $\log(f(n)) = \Theta(\log(n))$.

Proof: The asymptotic result of n^2 and n^10 are the same. $\square$

Definition 3.5.2. $\log^*(n) = o(\log\log\log\log\log(n)) = \alpha$.

Example: $\lg^*(2^{2^{2^{2^2}}}) = 5$.

4 Series

4.1 Some Definition

Definition 4.1.1. Harmonic Series:

$$\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log(n))$$

Definition 4.1.2. Geometric Series:

$$\sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1} = \begin{cases} \Theta(x^n) & \text{if } \forall x > 1, \\ \Theta(1) & \text{if } \forall x < 1, \\ \Theta(n) & \text{if } \forall x = 1. \end{cases}$$

Definition 4.1.3. Arithmetic Series:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \Theta(n^2)$$

4.2 Some Theorem

Suppose I want to know if $f(n) = o(g(n))$.

Theorem 4.2.1. *If* $\log(f(n)) = o(\log(g(n))$*, then* $f(n) = o(g(n))$.

Example: Let $f(n) = n^3$, $g(n) = 2^n$. Then $\log(f(n)) = \log(n^3) = 3\log(n)$, $\log(g(n)) = \log(2^n) = n$.

$$\text{i.e. } \log(f(n)) < \log(g(n) \Rightarrow f(n) < g(n)$$

Note that this theorem stands for 'o', NOT TRUE for '$\mathcal{O}$'.

Example: $\log(n^3) = \mathcal{O}(\log(n^2))$, but $n^3 \neq \mathcal{O}(n^2)$.

5 Induction

5.1 When to use?

Prove statement for all $n \in \mathbb{N}$, s.t. $n \geq n_0$.

5.2 Definition

Basically, induction has two parts:

1. Base case(s) – Sometimes there are more than one base cases.

   Prove statement for some n. – Often $n_0 = 0$ or 1.

2. Induction Hypothesis

   Assume statement hold true for all $m \leq n$.

   Prove the hypothesis implies that it hold true for $n + 1$.

Note that the process may be different from previous, which just hypothesize $n - 1$ is true and prove for n.

5.3 Example

5.3.1 Good Induction:

Claim: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Proof: We are required to prove $\forall n > 0,\ \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

Base Case: $n = 1$, $\sum_{i=1}^{1} i = 1 = \frac{1\times(1+1)}{2}$. Hence the claim holds true for $n = 1$.

Induction step: Let $k > 1$ be an arbitrary natural number.

Let us assume the induction hypothesis: for every $k < n$, assume $\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$. We will prove $\sum_{i=1}^{k+1} i = \frac{(k+1)(k+2)}{2}$

$$\sum_{i=1}^{k+1} i = \left(\sum_{i=1}^{k} i\right) + (n+1) = \frac{k(k+1)}{2} + \frac{2(k+1)}{2} = \frac{(k+1)(k+2)}{2} \tag{5}$$

Thus establishes the claim for $k + 1$.

Conclusion: By the principle of mathematical induction, the claim holds for all n. □

5.3.2 Bad Induction: Prove all horses are the same color

The process is omitted. The key point is that: if the base case is not true for induction hypothesis, the induction will not be solid.

6 Recursion (Divide & Conquer)

- Recursion is like Induction's twin brother, whereas induction is similar to movie filmed, and recursion is similar to movie backward.
- Recursion design may be most important course topic.
- Recursion is a type of reduction. [1]

6.1 Definition of Recursion: a Powerful type of reduction

1. if problem size very small (think $\mathcal{O}(1)$), just solve it.
2. reduce to one or more small instances of some problem.

Question: How are the smaller (but not $\mathcal{O}(1)$ size) problem solved?
Not your problem! Handled by the recursion fairy.

6.2 Tower of Hanoi

- 3 pegs, which hold n distinct sized disks.
- initially *tmp*, *dst* empty and *src* has all disks sorted.
- 3 rules:
    1. larger cannot be placed on smaller.
    2. only one disks can move at a time.
    3. move all disks to *dst*.

Question: How long until the world end?

Solution

[1] Reduction is to solve problem A using a black box for B. Typically B is smaller.

7 Dynamic Programming

7.1 Rod Cutting

7.1.1 Description of Problem

- steel rod of length n, where n is some integer.
- $P[1...n]$, where $P[i]$ is market price for rod of length i.

Question:

Suppose you can cut rod to any integer length for free. How much money can you made?

7.1.2 Analysis

- Consider leftmost cut of optimal solution.

  cut can be at positions $1...n$.

  If leftmost cut at i, then you get $P[i]$ for leftmost piece and then optimally sell remaining $n-i$ length rod.

- Don't know where to make first cut, so try them all and find

$$\max\left(0, \max(P[i] + cutRod(n-i))\right)$$

So, the first attempt of the algorithm could be described as algorithm 4.

Algorithm 4 First Attempt of Solving Cutting Rod Problem

procedure CUTROD(n)
    if $n = 0$ **then** ▷ `If the remaining rod length is 0.` ◁
        return 0
    end if
    $q = 0$
    for $i = 1$ to n **do**
        $q = \max\left(q, P[i] + \text{CUTROD}(n-i)\right)$
    end for
    return q
end procedure

Running time of algorithm 4:$T(n) = n + \sum_{i=0}^{n-i} T(i)$, which is clearly **Exponential** since there are a lot of subproblem overlap!

7.1.3 Memoized Version

algorithm 5 illustrates the memoized version of the algorithm in algorithm 4

Algorithm 5 Memoized Version of Solving Cutting Rod Problem

```
procedure MemRodCut(n)                    ▷ Globally define R[1..n] ◁
    if n = 0 then
        return 0
    end if
    if R[n] undefined then
        q = 0
        for i = 1 to n do
            q = max (q, P[i] + MemRodCut(n − i))
        end for
        R[n] = q
    end if
    return R[n]
end procedure
```

Note that $R[1...n]$ *is filled in form left to right.* It means we can store the result and use it later, which brings us to the dynamic programming version of the algorithm.

7.1.4 Dynamic Programming Version to Solve RodCut

algorithm 6 illustrates the dynamic programming version of the algorithm according to the memoized version algorithm 5.

Algorithm 6 Dynamic Programming Version of Solving Cutting Rod Problem

```
procedure DPRodCut(n)
    Let R[0...n] be an array.
    R[0] = 0
    for j = 1 to n do
        q = 0
        for i = 0 to j do
            q = max(q, P[i] + R[j − i])
        end for
        R[i] = q
    end for
    return R[n]
end procedure
```

Running time of algorithm 6: $T(n) = \mathcal{O}(n^2)$.

Note that the process only computes the total number. If we are to know how to cut, we can store the cutting position during the progress.

Define $C[1...n]$, and replace the inner for loop in algorithm 6 as:

Algorithm 7 Store the Cutting Position in the Process

```
for i = 0 to j do
    q = max(q, P[i] + R[j − i])
    C[j] = i
end for
R[i] = q
```

The for loop does the following:

- $C[j]$ stores last leftmost cut length for rod of length j.
- $C[n]$ says where to make first

Thus $C[n - C[n]]$ tells the second cut.

7.2 Solving DP problem

According to previous examples, we can summarize the general method to solve DP problem.

1. Write recursive solution, explain why the solution is correct.
2. Identify all subproblems considered.
3. Described how to store subproblems.
4. Find order to evaluate subproblems, s.t. subproblems you depend on evaluated ***before*** current subproblem.
5. Running Time: time to fill an entry X size table.
6. Write DP/Memoized algorithm.

7.3 Longest Increasing Subsequence (LIS)

7.3.1 Description of Problem

Input: Array $A[1...n]$ of integers.

Output: Longest subsequence of indices, $1 \leq i_1 < i_2 < ... < i_k < n$, s.t. $A[i_j] < A[i_{j+1}$ for all j.

Warning: Subarray is "contiguous". So what is a subsequence?

- if $n = 0$, the onl subsequence is empty sequence.
- otherwise, a subsequence is either
  1. a subsequence of $A[2...n] or$,
  2. $A[1]$ followed by the subsequence of $A[2...n]$.

7.3.2 Analysis

Suggest recursive strategy for any array subsequence problem.

- if empty, do nothing.
- otherwise figure out whether to take $A[1]$ and let recursion fairy handle $A[2...n]$.

However, the definition of the subsequence is not fully recursive as stated, causing handling $A[2...n]$ depends on whether take $A[1]$.

To fix it, define LIS subsequence with all elements greater than some value as follow.

- LIS(prev, start) be the LIS in $A[start, n]$, s.t. all elements greater then $A[prev]$.
- Augment A s.t $A[0] = -\infty$, then LIS of $A[1...n]$ is LIS(0, 1).

Note that the idea of adding a $A[0]$ maybe useful in many scenarios.

Algorithm 8 Original Algorithm for LIS Problem

```
procedure LIS(prev, start)                    ▷ prev < start ◁
    if start > n then
        return 0
    end if
    ignore = LIS(prev, start + 1)
    best = ignore
    if A[start] > A[prev] then
        include = 1 + LIS(start, start + 1)
        if include > ignore then
            best = include
        end if
    end if
    return best
end procedure
```

LIS $(prev, start)$ is the length of longest increasing subsequence in $A[start \dots n]$, s.t. all elements greater than $A[prev]$.

Observation:

The procedure LIS $(prev, start)$ has the following features:

- LIS $(prev, start)$ depends on LIS $(prev, start + 1)$
- So need 2D table $B[0 \dots n][1 \dots n+1]$, each entry takes $\mathcal{O}(1)$ time to fill in, and can fill in any ordeer, s.t. $B[\ ][start+1]$ filled before $B[\ ][start]$.

7.3.3 Dynamic Programming Version to Solve LIS

algorithm 9 illustrates the dp version to solve LIS problem.

Algorithm 9 Dynamic Programming Algorithm for LIS Problem

procedure LISDP($A[1 \dots n]$)
    $A[0] = -\infty$
    init $B[0 \dots n][1 \dots n+1]$
    for $i = 0$ to n **do**
        $B[i][n+1] = 0$
    end for
    for $start = n$ to 1 **do**
        for $prev = start - 1$ to 0 **do**
            if $A[prev] \geq A[start]$ **then**
                $B[prev][start] = B[prev][start+1]$
            else
                $B[prev][start] = \max\{(B[prev][start+1], 1 + B[start][start+1]\}$
            end if
        end for
    end for
    return $B[0][1]$
end procedure

The running time for algorithm 9 is the time to fill the table, i.e. $\mathcal{O}(n^2)$.

7.4 Longest Common Subsequence

7.4.1 Description of Problem

Input: Character arrays: $A[1 \dots n]$, $B[1 \dots m]$.

Output: Find length of longest common subsequence, i.e. find length k of longest pair of strings of indices.

$$1 \leq i_1 < i_2 < \ldots < i_k \leq n$$

and

$$1 \leq j_1 < j_2 < \ldots < j_k \leq m$$

s.t. for all $1 \leq l \leq k$, $A[i_l] = B[j_l]$.

Example:

$$A = \{ \texttt{C G C A A T C C A G G} \}$$
$$B = \{ \texttt{G A T T A C G A} \}$$

LCS = G A T C A.

The sequence is {2,4,6,8,9} and {1,2,3,6,8}. Note that the sequence may not be unique.

7.4.2 Analysis

Handle first element of A and B:

- if A or B is empty, return 0;
- if $A[1] \neq B[1]$, then $A[1]$ and $B[1]$ cannot both be used. So should be the best solution from throwing out $A[1]$ or $B[1]$, i.e.

$$LCS(A[1\ldots n], B[1\ldots m]) = \max \left\{ \begin{array}{l} LCS(A[2\ldots n], B[1\ldots m]), \\ LCS(A[1\ldots n], B[2\ldots m]) \end{array} \right\} \tag{6}$$

- if $A[1] = B[1]$, then can either match or throw both of them out, i.e

$$LCS(A[1\ldots n], B[1\ldots m]) = \max \left\{ \begin{array}{l} 1 + LCS(A[2\ldots n], B[2\ldots n]), \\ LCS(A[2\ldots n], B[1\ldots m]), \\ LCS(A[1\ldots n], B[2\ldots m]) \end{array} \right\} \tag{7}$$

Note that if $A[1] = B[1]$, why consider throw them out?

Though without proof, the option is right, but generally, the option should be considered.

The algorithm is described in algorithm 10. LCS $(curA, CurB)$ is the longest common sequence of $A[curA \ldots n]$ and $B[curB \ldots m]$.

Algorithm 10 Original Algorithm for LCS Problem

```
procedure LCS(curA, curB)
    if curA > n or curB > m then
        return 0
    end if
    ignore = max{LCS(curA + 1, CurB), LCS(CurA, CurB + 1)}
    best = ignore
    if A[curA] = B[curB] then
        include = 1 + LCS(curA + 1, curB + 1)
        if include > ignore then
            best = include
        end if
    end if
    return best
end procedure
```

To find LCS of $A[1 \dots n]$, $B[1 \dots m]$, call LCS $(1, 1)$.

Observation:

The procedure LCS $(curA, curB)$ has the following features:

- LCS $(curA, curB)$ depends on two parameters, need a 2D array of total size $\mathcal{O}(nm)$.
- LCS $(curA, curB)$ makes 3 recursive calls. All recursive calls have at least one parameter strictly larger, and none smaller.
- The table can be filled in two nested decreasing for loop.

7.4.3 Dynamic Programming Version to Solve LCS

The DP algorithm to solve LCS problem is showed in algorithm 11.

Algorithm 11 Dynamic Programming Algorithm for LCS Problem

```
procedure LCSDP(A[1...n], B[1...m])
    Define C[1...n+1][1...m+1]
    for i = 0 to n+1 do
        C[i][m+1] = 0
    end for
    for i = 0 to m+1 do
        C[n+1][i] = 0
    end for
    for curA = n to 1 do
        for curB = m to 1 do
            ignore = max{C[curA+1][CurB], C[curA][curB+1]}
            best = ignore
            if A[curA] = B[curB] then
                include = 1 + C[curA+1][curB+1]
                if include > ignore then
                    best = include
                end if
            end if
            C[curA][curB] = best
        end for
    end for
    return C[1][1]
end procedure
```

The operation to fill the table takes $\mathcal{O}(1)$ time, ignoring recursive calls.

So, total time is $\mathcal{O}(n \times m \times 1) = \mathcal{O}(nm)$.

7.5 Edit Distance

7.5.1 Description of Problem

Input: Character arrays: $A[1 \ldots m]$, $B[1 \ldots n]$

Output: Edit distance between A and B, which is the minimum number of characters insertion, deletion and substitution to turn A into B.

Example: For character arrays A and B:

$$A = \{ \texttt{F O O D} \}$$
$$B = \{ \texttt{M O N E Y} \}$$

The edit process is:

$$
\begin{aligned}
\texttt{F O O D} &\rightarrow \texttt{M O O D} \\
&\rightarrow \texttt{M O N D} \\
&\rightarrow \texttt{M O N E D} \\
&\rightarrow \texttt{M O N E Y}
\end{aligned}
$$

7.5.2 Analysis

Consider a better way to display edits:

Place A above B, put a gap in A for each insertion, gap in B for each deletion, i.e.

$$
\begin{aligned}
A &= \{\ \texttt{F O O \ \ D}\ \} \\
B &= \{\ \texttt{M O N E Y}\ \}
\end{aligned}
$$

Then, the edit distance is the number of columns where characters don't match (in an optional alignment).

Let $edit(A, B)$ denotes the edit distance from A to B. Note that $edit(A, B) = edit(B, A)$.

Another example:

$$
\begin{aligned}
A &= \{\texttt{A L G O R \ \ I \ \ T H M}\ \} \\
B &= \{\texttt{A L \ \ T R U I S T I C}\ \}
\end{aligned}
$$

If remove last column from an optimal alignment, the remaining must represent the shortest edit sequence for remaining substrings.

So now we can recursively define edit distance $edit(A, B)$:

- if $m = 0$, i.e. A is empty, then $edit(A, B) = n$;
- if $n = 0$, i.e. B is empty, then $edit(A, B) = m$;
- otherwise, $m, n \geq 1$, then look at last column in optimal alignment. There are three cases:

  a) insertion, i.e. top row empty, which means:
     - All of A remains to left;

– All but last character of B to left;

$$edit(A[1 \dots m], B[1 \dots n]) = edit(A[1 \dots m], B[1 \dots n-1]) + 1$$

b) deletion, i.e. bottom row empty, which means, like insertion:

$$edit(A[1 \dots m], B[1 \dots n]) = edit(A[1 \dots m-1], B[1 \dots n]) + 1$$

c) substitution, i.e. both rows non-empty, which means:

$$edit(A[1 \dots m], B[1 \dots n]) = edit(A[1 \dots m-1], B[1 \dots n-1]) + [A[m] \neq B[n]]$$

where the condition $[A[m] \neq B[n]]$ is 1 if true and 0 if false.

The algorithm is described in algorithm 12. $\textsc{Edit}(i,j) = edit(A[1 \dots i], B[1 \dots j])$.

Algorithm 12 Original Algorithm for Edit Distance

```
procedure EDIT(i, j)
    if i = 0 then
        return j
    end if
    if j = 0 then
        return i
    end if
    return min{EDIT(i − 1, j) + 1, EDIT(i, j − 1) + 1, EDIT(i − 1, j − 1) + [A[i] ≠ B[j]]}
end procedure
```

Observation:

The procedure $\textsc{Edit}$ (i,j) has the following features:

- i has m values.
- j has n values.
- store in table of size $\mathcal{O}(mn)$.
- $\textsc{Edit}$ (i,j) depends on three sub-problems, in each case either i or j smaller (and never larger).
- can fill in table with two nested increasing for loop.
- constant time per entry, so $\mathcal{O}(mn)$ overall.

7.5.3 Dynamic Programming Version to Solve Edit Distance Problem

The DP algorithm to solve Edit Distance problem is showed in algorithm 13.

Algorithm 13 Dynamic Programming Algorithm for Edit Distance Problem

```
procedure EditDP(A[i...m], B[i...n])
    for j = 0 to n do
        E[0][j] = j
    end for
    for i = 0 to n do
        E[i][0] = i
    end for
    for i = 1 to m do
        for j = 1 to n do
            if A[i] = B[j] then
                E[i][j] = min{E[i-1][j] + 1, E[i][j-1] + 1, E[i-1][j-1]}
            else
                E[i][j] = min{E[i-1][j] + 1, E[i][j-1] + 1, E[i-1][j-1] + 1}
            end if
        end for
    end for
    return E[m][n]
end procedure
```

The operation to fill the table takes $\mathcal{O}(1)$ time, ignoring recursive calls.

So, total time is obviously $\mathcal{O}(mn)$.