

Notes on CS6363

Hanlin He

Friday October 7, 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
6.2.1	Description of Problem	9
6.2.2	Analysis	9
6.3	Binary Search	10
6.4	Maximum Sub-array Sum	12
6.4.1	Description of Problem	12

6.4.2	Analysis	12
7	Dynamic Programming	13
7.1	Rod Cutting	13
7.1.1	Description of Problem	13
7.1.2	Analysis	13
7.1.3	Memoized Version	14
7.1.4	Dynamic Programming Version to Solve RodCut	14
7.2	Solving DP problem	15
7.3	Longest Increasing Subsequence (LIS)	16
7.3.1	Description of Problem	16
7.3.2	Analysis	16
7.3.3	Dynamic Programming Version to Solve LIS	17
7.4	Longest Common Subsequence	18
7.4.1	Description of Problem	18
7.4.2	Analysis	19
7.4.3	Dynamic Programming Version to Solve LCS	20
7.5	Edit Distance	21
7.5.1	Description of Problem	21
7.5.2	Analysis	22
7.5.3	Dynamic Programming Version to Solve Edit Distance Problem	24
7.6	Longest Convex Subsequence	24
7.6.1	Description of Problem	24
7.6.2	Analysis	25
7.7	Professor's note on LIS/LCS/LXS/SCS	25
7.8	Summary	32

List of Algorithms

1	Stupid Find Max Algorithm	2
2	Sort & Find Max Algorithm	3
3	Search & Find Max Algorithm	3
4	Recursive Hanoi	10
5	Binary Search Algorithm	11
6	First Attempt of Solving Cutting Rod Problem	13
7	Memoized Version of Solving Cutting Rod Problem	14
8	Dynamic Programming Version of Solving Cutting Rod Problem	15
9	Store the Cutting Position in the Process	15
10	Original Algorithm for LIS Problem	17
11	Dynamic Programming Algorithm for LIS Problem	18
12	Original Algorithm for LCS Problem	20
13	Dynamic Programming Algorithm for LCS Problem	21
14	Original Algorithm for Edit Distance	23
15	Dynamic Programming Algorithm for Edit Distance Problem	24
16	Recursive Algorithm for Longest Convex Subsequence	25

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 course's 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: $MAX A[i], 1 \leq i \leq n$.

2.2.1 Algorithm 1

Stupid way.

Algorithm 1 Stupid Find Max Algorithm

```
1: procedure FINDMAX
2:   for  $i = 1$  to  $n$  do
3:      $count = 0$ 
4:     for  $j = 1$  to  $n$  do
5:       if  $A[i] > A[j]$  then
6:          $count = count + 1$ 
7:       end if
8:     end for
9:     if  $count = n$  then
10:      return  $A[i]$ 
11:    end if
12:  end for
13: end procedure
```

Analysis: Worst Case, n^2 comparison.

2.2.2 Algorithm 2

Sort & Find.

Algorithm 2 Sort & Find Max Algorithm

```
1: procedure FINDMAX
2:    $\bar{A} = \text{sort}(A)$ 
3:   return  $\bar{A}[n]$ 
4: 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

```
1: procedure FINDMAX
2:    $current = 1$ 
3:   for  $i = 2$  to  $n$  do
4:     if  $A[i] > A[current]$  then
5:        $current = i$ 
6:     end if
7:   end for
8:   return  $A[current]$ 
9: 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 $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \neq \infty$, i.e. it is $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < k$, for some constant k .

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

Table 2: Definition for all Asymptotic Notation

$f(n)$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$	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. $\forall n \geq n_0$, $100 * f(n) \leq c' * g(n) = 100c * g(n)$. \square

Example:

$$C * 2^n = \Theta(2^n) \quad (1)$$

$$(C * 2)^n \neq \Theta(2^n) \quad (2)$$

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. $\forall a, b > 1$

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

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^1 are the same. \square

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

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

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 ‘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 $\sum_{i=1}^{k+1} i = \frac{(k+1)(k+2)}{2}$

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

Thus establishes the claim for $k + 1$.

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

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

6.2.1 Description of Problem

- 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?

6.2.2 Analysis

A small hint: not consider the smallest first, but the largest first.

In order to move the largest disk:

- *dst* has to be empty.

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

- *src* has only largest one.
- *tmp* has $n - 1$ disks sorted.

So we must:

1. move $n - 1$ disks from *src* to *tmp* .
2. move largest from *src* to *dst* .
3. move $n - 1$ disk from *tmp* to *dst* .

Don't know how to do.

Don't think about it!!

Don't think about how to move $n - 1$ disks, recursion fairy will do it.

Algorithm 4 Recursive Hanoi

```

procedure HANOI( $n, src, dst, tmp$ )
  if  $n > 0$  then
    HANOI ( $n - 1, src, tmp, dst$ )
    Move disk  $n$  from src to dst.
    HANOI ( $n - 1, tmp, dst, src$ )
  end if
end procedure

```

How many moves in algorithm 4 ?

Let $T(n)$ be the total moves for n disks.

$$T(0) = 0$$

$$T(1) = 1$$

$$T(n) = T(n - 1) + 1 + T(n - 1)$$

$$= 2T(n - 1) + 1$$

“Solve” the recurrence.

$$\sum_{l=1}^n 2^{l-1} = 2^n - 1$$

6.3 Binary Search

Input: Given a value *val*, and sorted array $A[1 \dots n]$.

Output: Is *val* contained in *A*?

Algorithm 5 Binary Search Algorithm

```
1: procedure BIN(val, low, high)
2:   if high < low then
3:     return Not Found
4:   end if
5:   mid =  $\lfloor \frac{\textit{high} + \textit{low}}{2} \rfloor$ 
6:   if val < A[mid] then
7:     return BIN (val, low, mid - 1)
8:   end if
9:   if val > A[mid] then
10:    return BIN (val, mid + 1, high)
11:  end if
12:  return mid
13: end procedure
```

To find whether *val* is contained in $A[1 \dots n]$, call BIN (*val*, 1, *n*).

Let $m = \textit{high} - \textit{low} + 1$,

$$T(m) \leq \max \left\{ T\left(\left\lfloor \frac{m-1}{2} \right\rfloor\right), T\left(\left\lceil \frac{m-1}{2} \right\rceil\right) \right\} + \Theta(1) \quad (6)$$

$$\leq T\left(\frac{m}{2}\right) + \Theta(1) \quad (7)$$

$$T(m) = T\left(\frac{m}{2}\right) + 1 \quad (8)$$

$$T(m) = \Theta(1) \text{ for } m = \Theta(1) \quad (9)$$

Floor and Ceilings has differences in constant time.

To think the running time in another way:

A constant size problem should has a constant solution.

$$\begin{aligned} T(m) &= T\left(\frac{m}{2}\right) + 1 \\ &= T\left(\frac{m}{2 \times 2}\right) + 1 + 1 \\ &= T\left(\frac{m}{2 \times 2 \times 2}\right) + 1 + 1 + 1 \end{aligned}$$

Counting the number of times divide by 2 to get to 1, i.e.

$$\frac{m}{2^x} = 1 \rightarrow m = 2^x \rightarrow x = \lg m$$

6.4 Maximum Sub-array Sum

6.4.1 Description of Problem

Input: Given an unsorted array $A[1 \dots n]$ of integers, including both negative and positive number.

Output: $\max_{i \leq j} \left\{ \sum_{k=i}^j A[k], 0 \right\}$.

6.4.2 Analysis

The Naive Solution

Let $W[i][j] = \sum_{k=i}^j A[k]$ for $i \leq j$. Return $\max\{0, \max\{W[i][j]\}\}$. The running time of this

brute force solution is $T(n) = \sum_{j=1}^n \sum_{i=1}^j (j - i + 1) = \Theta(n^3)$

Think Recursively!

Let $maxSum(i, j)$ be the maximum sub-array sum in $A[1 \dots n]$ or 0. The solution is $maxSum(1, n)$.

There are two cases:

Can we express $maxSum(1, n)$ in term of $maxSum(1, n - 1)$?

- todo.
- todo.

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

$$\max(0, \max(P[i] + \text{cutRod}(n - i)))$$

So, the first attempt of the algorithm could be described as algorithm 6.

Algorithm 6 First Attempt of Solving Cutting Rod Problem

```
1: procedure CUTROD( $n$ )
2:   if  $n = 0$  then                                     ▷ If the remaining rod length is 0. ◁
3:     return 0
4:   end if
5:    $q = 0$ 
6:   for  $i = 1$  to  $n$  do
7:      $q = \max(q, P[i] + \text{CUTROD}(n - i))$ 
8:   end for
9:   return  $q$ 
10: end procedure
```

Running time of algorithm 6: $T(n) = n + \sum_{i=0}^{n-i} T(i)$, which is clearly **Exponential** since there are a lot of subproblem overlap!

7.1.3 Memoized Version

algorithm 7 illustrates the memoized version of the algorithm in algorithm 6

Algorithm 7 Memoized Version of Solving Cutting Rod Problem

```
1: procedure MEMRODCUT( $n$ ) ▷ Globally define  $R[1..n]$  ◁
2:   if  $n = 0$  then
3:     return 0
4:   end if
5:   if  $R[n]$  undefined then
6:      $q = 0$ 
7:     for  $i = 1$  to  $n$  do
8:        $q = \max(q, P[i] + \text{MEMRODCUT}(n - i))$ 
9:     end for
10:     $R[n] = q$ 
11:  end if
12:  return  $R[n]$ 
13: 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 8 illustrates the dynamic programming version of the algorithm according to the memoized version algorithm 7.

Algorithm 8 Dynamic Programming Version of Solving Cutting Rod Problem

```
1: procedure DPRODCUT( $n$ )
2:   Let  $R[0\dots n]$  be an array.
3:    $R[0] = 0$ 
4:   for  $j = 1$  to  $n$  do
5:      $q = 0$ 
6:     for  $i = 0$  to  $j$  do
7:        $q = \max(q, P[i] + R[j - i])$ 
8:     end for
9:      $R[j] = q$ 
10:  end for
11:  return  $R[n]$ 
12: end procedure
```

Running time of algorithm 8: $T(n) = \mathcal{O}(n^2)$.

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 $C[1\dots n]$, and replace the inner for loop in algorithm 8 as:

Algorithm 9 Store the Cutting Position in the Process

```
1: for  $i = 0$  to  $j$  do
2:    $q = \max(q, P[i] + R[j - i])$ 
3:    $C[j] = i$ 
4: end for
5:  $R[i] = q$ 
```

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, $1 \leq i_1 < i_2 < \dots < i_k < n$, 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.

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

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

Algorithm 10 Original Algorithm for LIS Problem

```

1: procedure LIS( $prev, start$ )  $\triangleright prev < start <$ 
2:   if  $start > n$  then
3:     return 0
4:   end if
5:    $ignore = LIS(prev, start + 1)$ 
6:    $best = ignore$ 
7:   if  $A[start] > A[prev]$  then
8:      $include = 1 + LIS(start, start + 1)$ 
9:     if  $include > ignore$  then
10:       $best = include$ 
11:    end if
12:   end if
13:   return  $best$ 
14: 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 order, s.t. $B[\][start + 1]$ filled before $B[\][start]$.

7.3.3 Dynamic Programming Version to Solve LIS

algorithm 11 illustrates the dp version to solve LIS problem.

Algorithm 11 Dynamic Programming Algorithm for LIS Problem

```
1: procedure LISDP( $A[1 \dots n]$ )
2:    $A[0] = -\infty$ 
3:   init  $B[0 \dots n][1 \dots n + 1]$ 
4:   for  $i = 0$  to  $n$  do
5:      $B[i][n + 1] = 0$ 
6:   end for
7:   for  $start = n$  to  $1$  do
8:     for  $prev = start - 1$  to  $0$  do
9:       if  $A[prev] \geq A[start]$  then
10:         $B[prev][start] = B[prev][start + 1]$ 
11:      else
12:         $B[prev][start] = \max\{(B[prev][start + 1], 1 + B[start][start + 1])\}$ 
13:      end if
14:    end for
15:  end for
16:  return  $B[0][1]$ 
17: end procedure
```

The running time for algorithm 11 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.

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

and

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

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

Example:

$$\begin{aligned} A &= \{ \text{ C G C A A T C C A G G } \} \\ B &= \{ \text{ G A T T A C G A } \} \end{aligned}$$

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.

$$LCS(A[1 \dots n], B[1 \dots m]) = \max \left\{ \begin{array}{l} LCS(A[2 \dots n], B[1 \dots m]), \\ LCS(A[1 \dots n], B[2 \dots m]) \end{array} \right\} \quad (10)$$

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

$$LCS(A[1 \dots n], B[1 \dots m]) = \max \left\{ \begin{array}{l} 1 + LCS(A[2 \dots n], B[2 \dots m]), \\ LCS(A[2 \dots n], B[1 \dots m]), \\ LCS(A[1 \dots n], B[2 \dots m]) \end{array} \right\} \quad (11)$$

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 12. $LCS(curA, CurB)$ is the longest common sequence of $A[curA \dots n]$ and $B[curB \dots m]$.

Algorithm 12 Original Algorithm for LCS Problem

```
1: procedure LCS(curA, curB)
2:   if curA > n or curB > m then
3:     return 0
4:   end if
5:   ignore = max{LCS(curA + 1, CurB), LCS(CurA, CurB + 1)}
6:   best = ignore
7:   if A[curA] = B[curB] then
8:     include = 1 + LCS(curA + 1, curB + 1)
9:     if include > ignore then
10:      best = include
11:    end if
12:   end if
13:   return best
14: 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 13.

Algorithm 13 Dynamic Programming Algorithm for LCS Problem

```
1: procedure LCSDP( $A[1 \dots n], B[1 \dots m]$ )
2:   Define  $C[1 \dots n + 1][1 \dots m + 1]$ 
3:   for  $i = 0$  to  $n + 1$  do
4:      $C[i][m + 1] = 0$ 
5:   end for
6:   for  $i = 0$  to  $m + 1$  do
7:      $C[n + 1][i] = 0$ 
8:   end for
9:   for  $curA = n$  to  $1$  do
10:    for  $curB = m$  to  $1$  do
11:       $ignore = \max\{C[curA + 1][curB], C[curA][curB + 1]\}$ 
12:       $best = ignore$ 
13:      if  $A[curA] = B[curB]$  then
14:         $include = 1 + C[curA + 1][curB + 1]$ 
15:        if  $include > ignore$  then
16:           $best = include$ 
17:        end if
18:      end if
19:       $C[curA][curB] = best$ 
20:    end for
21:  end for
22:  return  $C[1][1]$ 
23: 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: $A[1 \dots m], B[1 \dots n]$

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 :

$$A = \{ \text{ F O O D } \}$$

$$B = \{ \text{ M O N E Y } \}$$

The edit process is:

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

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.

$$A = \{ \text{ F O O } \quad \text{ D } \}$$

$$B = \{ \text{ M O N E Y } \}$$

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:

$$A = \{ \text{A L G O R } \quad \text{ I } \quad \text{ T H M } \}$$

$$B = \{ \text{A L } \quad \text{ T R U I S T I C } \}$$

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;

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

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

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

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

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

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

The algorithm is described in algorithm 14. $\text{EDIT}(i, j) = \text{edit}(A[1 \dots i], B[1 \dots j])$.

Algorithm 14 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  $\min\{\text{EDIT}(i-1, j) + 1, \text{EDIT}(i, j-1) + 1, \text{EDIT}(i-1, j-1) + [A[i] \neq B[j]]\}$ 
end procedure

```

Observation:

The procedure EDIT (i, j) has the following features:

- i has m values.
- j has n values.
- store in table of size $\mathcal{O}(mn)$.

- 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 15.

Algorithm 15 Dynamic Programming Algorithm for Edit Distance Problem

```

1: procedure EDITDP( $A[i \dots m], B[i \dots n]$ )
2:   for  $j = 0$  to  $n$  do
3:      $E[0][j] = j$ 
4:   end for
5:   for  $i = 0$  to  $n$  do
6:      $E[i][0] = i$ 
7:   end for
8:   for  $i = 1$  to  $m$  do
9:     for  $j = 1$  to  $n$  do
10:      if  $A[i] = B[j]$  then
11:         $E[i][j] = \min\{E[i-1][j] + 1, E[i][j-1] + 1, E[i-1][j-1]\}$ 
12:      else
13:         $E[i][j] = \min\{E[i-1][j] + 1, E[i][j-1] + 1, E[i-1][j-1] + 1\}$ 
14:      end if
15:    end for
16:  end for
17:  return  $E[m][n]$ 
18: end procedure

```

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

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

7.6 Longest Convex Subsequence

7.6.1 Description of Problem

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

Output: Longest subsequence of indices,

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

s.t. $2 \times A[i_j] < A[i_{j-1}] + A[i_{j+1}]$ for all j .

7.6.2 Analysis

If consider $A[i]$ as current element, whether choose it or throw it depends on its next element. Thus, consider $A[i + 1]$ as current element.

Algorithm 16 Recursive Algorithm for Longest Convex Subsequence

```

1: procedure LXS(preprev, prev, cur)
2:   if cur > n then
3:     return 0
4:   end if
5:   ignore = LXS ( preprev, prev , cur + 1 )
6:   best = ignore
7:   if preprev = -1 or  $A[\textit{cur}] > 2 \times A[\textit{prev}] - A[\textit{preprev}]$  then
8:     include = 1 + LXS(prev, cur, cur + 1)
9:     if include > ignore then
10:      best = include
11:    end if
12:  end if
13:  return best
14: end procedure

```

Does not change.

Consider next element.

Call LXS $(-1, -1, 1)$ will get the longest convex sequence.

There are three parameter with n size, so the table is $\mathcal{O}(n^3)$ size.

Each entry require $\mathcal{O}(1)$ time to fill, so the total running time is $\mathcal{O}(n^3)$.

The third parameter *cur* is always strictly larger, so the outer for loop should be a decreasing for loop for *cur*. The inner two for loop is irrelevant.

7.7 Professor's note on LIS/LCS/LXS/SCS

Longest Increasing Subsequence

Globally defined array $A[1 \dots n]$, augmented s.t. $A[0] = -\infty$. Assumes $0 \leq prev < start$.

```
1: procedure LIS(prev, start)
2:   if start > n then
3:     return 0
4:   end if
5:   ignore = LIS(prev, start + 1)
6:   best = ignore
7:   if  $A[start] > A[prev]$  then
8:     include = 1 + LIS(start, start + 1)
9:     if include > ignore then
10:      best = include
11:    end if
12:   end if
13:   return best
14: end procedure
```

Claim: $LIS(prev, start)$, for $prev < start$, returns the longest increasing subsequence in $A[start \dots n]$ s.t. all elements are greater than $A[prev]$.

Proof: If $start > n$, there are no elements left in the remaining part of A , and so the algorithm correctly returns 0. Otherwise $LIS(prev, start)$ either includes $A[start]$ or not.

- if not, then $LIS(prev, start) = LIS(prev, start + 1)$.
- if so, then it must be that $A[start] > A[prev]$, and all remaining element of the LIS that come after $A[start]$ must be great than $A[start]$. Therefore, $LIS(prev, start) = LIS(start, start + 1) + 1$, where the +1 counts $A[start]$.

So if $A[start] \leq A[prev]$ the solution must be $LIS(prev, start + 1)$, which is what the algorithm returns, i.e. in this case the if statement is not executed. If $A[start] > A[prev]$ the solution is either $LIS(prev, start + 1)$ or $LIS(start, start + 1)$, whichever is bigger. Since we don't know which is bigger, our algorithm tries both and takes the max. Note in both case the problem is reduced to a subproblem on a strictly smaller array (i.e. $A[start + 1 \dots n]$), and so can be assumed to be correctly handled by induction (where the base case is handled by the initial $start > n$ conditional). \square

To compute the LIS of $A[1 \dots n]$ we call $LIS(0, 1)$ as this is the LIS in $A[1 \dots n]$ s.t. all elements $> -\infty$, which is the same as the LIS of $A[1 \dots n]$.

Applying DP: $LIS(prev, start)$ depends on two parameters, each ranging over $O(n)$ values, as they are both indices into $A[0 \dots n]$. Hence the above recursive algorithm can be turned into a DP algorithm using a 2D array, of total size $O(n^2)$. Note that $LIS(prev, start)$ only depends on $LIS(prev, start + 1)$ and $LIS(start, start + 1)$, both of which have a strictly larger value of the second parameter. Therefore this table can be filled in any order such that all $LIS(\cdot, start + 1)$ values are computed before any $LIS(\cdot, start)$ value. Namely with a decreasing for loop for the second parameter, and a second inner loop going over all values of the first parameter (in any order). Ignoring the time for computing recursive calls, the above algorithm runs in $O(1)$ time. Therefore, if processed in the right order, each table entry takes $O(1)$ time to compute and so the total running time is $O(n^2)$.

Longest Common Subsequence

Input: Character arrays $A[1 \dots n]$ and $B[1 \dots m]$. Goal: Find the length of the longest common subsequence. Specifically, find the length k of the longest strings of indices $1 \leq i_1 \leq \dots \leq i_k \leq n$ and $1 \leq j_1 \leq \dots \leq j_k \leq m$ such that for all $1 \leq l \leq k$, $A[i_l] = B[j_l]$

If A is $[C, G, C, A, A, T, C, C, A, G, G]$ and B is $[G, A, T, T, A, C, G, A]$ an LCS is G, A, T, C, A as witnessed by the index strings $2, 4, 6, 8, 9$ and $1, 2, 3, 6, 8$.

As this is a subsequence problem, similar to LCS, the focus is to figure out how to handle the very first element of A and B . We have the following.

- 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 it should be the best of either throwing out $A[1]$ or $B[1]$, i.e.
 $LCS(A[1 \dots n], B[1 \dots m]) = \max\{LCS(A[2 \dots n], B[1 \dots m]), LCS(A[1 \dots n], B[2 \dots m])\}$.
- If $A[1] = B[1]$ then we can either match or throw out so $LCS(A[1 \dots n], B[1 \dots m]) = \max\{1 + LCS(A[2 \dots n], B[2 \dots m]), LCS(A[2 \dots n], B[1 \dots m]), LCS(A[1 \dots n], B[2 \dots m])\}$.

Note that if $A[1] = B[1]$ then one can prove $LCS(A[1 \dots n], B[1 \dots m]) = 1 + LCS(A[2 \dots n], B[2 \dots m])$. While this may seem obvious, unless it is proven you cannot assume it, so we won't.

Now to turn this into a recursive algorithm, define $LCS(curA, curB)$ to be the longest common subsequence of $A[curA \dots n]$ and $B[curB \dots m]$ (i.e. $LCS(A[curA \dots n], B[curB \dots m])$). Based on the above observations we have the following.

Again assume $A[1 \dots n]$ and $B[1 \dots m]$ defined globally, and $curA, curB > 0$

```

1: procedure  $LCS(curA, curB)$ 
2:   if  $curA > n$  or  $curB > m$  then
3:     return 0
4:   end if
5:    $ignore = \max\{LCS(curA + 1, curB), LCS(curA, curB + 1)\}$ 
6:    $best = ignore$ 
7:   if  $A[curA] = B[curB]$  then
8:      $include = 1 + LIS(curA + 1, curB + 1)$ 
9:     if  $include > ignore$  then
10:       $best = include$ 
11:    end if
12:  end if
13:  return  $best$ 
14: end procedure

```

To find the longest common subsequence of $A[1 \dots n]$ and $B[1 \dots m]$ we then call $LCS(1, 1)$. The correctness follows immediately from the above (arguing the same way as for LIS).

Applying DP. $LCS(curA, curB)$ depends on two parameters, the first ranging over $O(n)$ values and the second over $O(m)$ values, since they are indices into $A[1 \dots n]$ and $B[1 \dots m]$, respectively. Hence the above recursive algorithm can be turned into a DP algorithm using a 2D array, of total size $O(mn)$. $LCS(curA, curB)$ makes at most three recursive call to $LCS(curA + 1, curB)$, $LCS(curA, curB + 1)$, and $LCS(curA + 1, curB + 1)$. In each case at least one of the two parameters increases and the other does not decrease. Therefore, the 2D array can be filled in using a pair of nested for the loops, the outer one ranging over the first parameter and starting at n and going to down 1, and the inner one ranging over the second parameter and starting at m and going down to 1. Ignoring the time for computing recursive calls, the above algorithm runs in $O(1)$ time. Therefore, if processed in the right order, each table entry takes $O(1)$ time to compute and so the total running time is $O(n^2)$.

```

1: procedure LCSDP( $A[1 \dots n], B[1 \dots m]$ )
2:   Define  $C[1 \dots n + 1][1 \dots m + 1]$ 
3:   for  $i = 1$  to  $n + 1$  do
4:      $C[i][m + 1] = 0$ 
5:   end for
6:   for  $i = 1$  to  $m + 1$  do
7:      $C[n + 1][i] = 0$ 
8:   end for
9:   for  $curA = n$  to 1 do
10:    for  $curB = m$  to 1 do
11:       $ignore = \max\{C[curA + 1][curB], C[curA][curB + 1]\}$ 
12:       $best = ignore$ 
13:      if  $A[curA] = A[curB]$  then
14:         $include = 1 + C[curA + 1][curB + 1]$ 
15:        if  $include > ignore$  then
16:           $best = include$ 
17:        end if
18:      end if
19:       $C[curA][curB] = best$ 
20:    end for
21:  end for
22:  return  $C[1][1]$ 
23: end procedure

```

Longest Convex Subsequence

Call a sequence $x_1 \dots x_m$ of numbers convex if $2x_i < x_{i-1} + x_{i+1}$ for all i . Describe an efficient algorithm to compute the length of the longest convex subsequence of an arbitrary array $A[1 \dots n]$ of integers.

In the following *preprev* may get set to the dummy value -1 , which removes the convexity requirement.

```

1: procedure LXS(preprev, prev, cur)
2:   if cur > n then
3:     return 0
4:   ignore = LXS(preprev, prev, cur + 1)
5:   best = ignore
6:   if (preprev =  $-1$ )  $\vee$  ( $A[\textit{cur}] > 2A[\textit{prev}] - A[\textit{preprev}]$ ) then
7:     include = 1 + LXS(prev, cur, cur + 1)
8:     if include > ignore then
9:       best = include
10:  return best

```

$LXS(\textit{preprev}, \textit{prev}, \textit{cur})$ computes the length of the longest convex subsequence in the array $A[\textit{cur} \dots n]$, but with the requirement that the sequence is still convex after appending $A[\textit{preprev}]A[\textit{prev}]$ to the front (unless $\textit{preprev} = -1$, in which case there is no additional requirement involving $A[\textit{preprev}]A[\textit{prev}]$). Thus the longest convex subsequence of A is given by $LXS(-1, -1, 1)$. Note that the optimal such subsequence either “includes” or “ignores” the element $A[\textit{cur}]$ and the algorithm just returns the best of these two options (where including is only allowed if convexity is satisfied).

Each parameter in the above algorithm ranges over $O(n)$ values, and hence the DP table has size $O(n^3)$. Each entry takes $O(1)$ to compute so the total run time is $O(n^3)$. Every subproblem has a strictly larger last parameter hence the table can be filled with a decreasing outermost for loop for this parameter, and two nested inner for loops with any order for the other parameters.

Shortest Common Supersequence

Let $A[1 \dots m]$ and $B[1 \dots n]$ be two arbitrary arrays. A common supersequence of A and B is another sequence that contains both A and B as subsequences. Describe an efficient algorithm to compute the length of the shortest common supersequence of A and B .

```

1: procedure SCS( $curA, curB$ )
2:   if  $curA > m$  then
3:     return  $n - curB + 1$ 
4:   if  $curB > n$  then
5:     return  $m - curA + 1$ 
6:    $different = \min\{1 + SCS(curA + 1, curB), 1 + SCS(curA, curB + 1)\}$ 
7:    $best = different$ 
8:   if  $A[curA] = B[curB]$  then
9:      $match = 1 + SCS(curA + 1, curB + 1)$ 
10:    if  $match < different$  then
11:       $best = match$ 
12:  return  $best$ 

```

Assuming we argued you can always take a match, this simplifies to:

```

1: procedure SCS( $curA, curB$ )
2:   if  $curA > m$  then
3:     return  $n - curB + 1$ 
4:   if  $curB > n$  then
5:     return  $m - curA + 1$ 
6:   if  $A[curA] = B[curB]$  then
7:     return  $1 + SCS(curA + 1, curB + 1)$ 
8:   return  $\min\{1 + SCS(curA + 1, curB), 1 + SCS(curA, curB + 1)\}$ 

```

$SCS(curA, curB)$ is the length of the shortest common supersequence of $A[curA \dots m]$ and $B[curB \dots n]$, hence the shortest common supersequence of $A[1 \dots m]$ and $B[1 \dots n]$ is $SCS(1, 1)$. Specifically the next element in the shortest common supersequence is either $A[curA]$ or $B[curB]$, and the algorithm tries both options (where if $A[curA] = B[curB]$ then the supersequence only needs one character to cover both $A[curA]$ and $B[curB]$).

Each of the two parameters range over $O(n)$ values so use a DP table of size $O(n^2)$. Each entry takes $O(1)$ time to compute, so the total time is $O(n^2)$. At least one of the two parameters is larger (and the other not smaller), hence we can fill in the table with two nested decreasing for loops going over the two parameters.

7.8 Summary

Quote from professor:

When solving the DP problem on the exam ask yourself:

“What is the minimal amount of information needed to determine whether I can take the current element as the next element in the subsequence?”

For the longest convex subsequence I need to know what the previous two elements in the subsequence were. Shortest common super-sequence was even easier, as there is no requirement imposed by earlier elements. In both problems we then try all options of taking or not taking the current element(s) and return the best result.

For increasing/decreasing in the For-Loop, look for the parameter(s) that is strictly increasing or decreasing.

- If only one is increasing/decreasing, then the outer For-Loop should be decreasing/increasing about that parameter. The other For-Loop is irrelevant (for example the inner two For-Loop in LXS).
- If both parameter is increasing/decreasing, then two For-Loop should both be decreasing/increasing about its parameter. Which For-Loop is the outer one is irrelevant (for example the two For-Loop in SCS).