Notes on CS6363

Hanlin He

Wednesday November 2, 2016

Contents

List of Algorithms

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: $MAX A[i], 1 <= i <= n$.

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

```
1: procedure FindMax
2:     $\overline{A} = sort(A)$
3:     return $\overline{A}[n]$
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 $\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty$, 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}). □

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. $\forall a, b > 1$

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

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

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.

[1]Reduction 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.

$$\begin{aligned} T(0) &= 0 \\ T(1) &= 1 \\ T(n) &= T(n-1) + 1 + T(n-1) \\ &= 2T(n-1) + 1 \end{aligned}$$

"Solve" the recurrence.

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

6.3 Binary Search

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

Output: Is *val* contained in A?

Algorithm 5 Binary Search Algorithm

```
procedure BIN($val, low, high$)
    if $high < low$ then
        return Not Found
    end if
    $mid = \left\lfloor \frac{high+low}{2} \right\rfloor$
    if $val < A[mid]$ then
        return BIN ($val, low, mid - 1$)
    end if
    if $val > A[mid]$ then
        return BIN ($val, mid + 1, high$)
    end if
    return $mid$
end procedure
```

To find whether val is contained in $A[1 \dots n]$, call BIN $(val, 1, n)$.
Let $m = high - low + 1$,

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

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

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

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

Floor and Ceilings has differences in constant time.

A constant size problem should has a constant solution.

To think the running time in another way:

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

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

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

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: $\max\limits_{i \le j} \left\{ \sum\limits_{k=i}^{j} A[k], 0 \right\}$.

6.4.2 Analysis

The Naive Solution

Let $W[i][j] = \sum\limits_{k=i}^{j} A[k]$ for $i \le j$. Return $\max\{0, \max\{W[i][j]\}\}$. The running time of this brute force solution is$T(n) = \sum\limits_{j=1}^{n} \sum\limits_{i=1}^{j} (j - i + 1) = \Theta(n^3)$

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 $maxSum(1, n-1)$?

- $maxSum(1, n)$ does not include $A[n]$

$$maxSum(1, n) = maxSum(1, n-1)$$

- $maxSum(1, n)$ does include $A[n]$

$$maxSum(1, n) = maxEndAt(1, n)$$

$maxEndAt(i, j)$ is the maximum sub-array sum in $A[i, j]$ restricted to include $A[j]$.
Here comes the recursive version algorithm to solve this problem, showing in .

Algorithm 6 Recursive Solution for Maximum Sub-Array Sum Problem

```
1: procedure maxSum($i, j$)
2:     if $j < i$ then
3:         return 0
4:     end if
5:     $x =$ maxEndAt $(i, j)$
6:     return $\max\{x, \text{maxSum}(i, j-1)\}$
7: end procedure
8: procedure maxEndAt($i, j$)
9:     if $j < i$ then
10:        return 0
11:    end if
12:    return $\max\{A[n], A[n] + \text{maxEndAt}(i, j-1)\}$
13: end procedure
```

The running time for maxEndAt (i, j) is $T(n) = T(n-1) + \Theta(1) = \Theta(n)$. Accordingly, the running time for maxSum (i, j) is

$$\begin{aligned} T(m) &= T(m-1) + \Theta(m) \\ &= T(m-1) + m \\ &= \sum_{k=1}^{m} k \\ &= \Theta(m^2) \end{aligned}$$

Can we do better?

Yes, use Divide and Conquer.

6.4.3 Applying Divide & Conquer on Max Sub-Array Sum Problem

The key point of D&C is try to get problem size down quickly.

Algorithm 7 Divide & Conquer Solution for Maximum Sub-Array Sum Problem

```
1: procedure DCMaxSum($i, j$)
2:     if $j < i$ then
3:         return 0
4:     end if
5:     $mid = \lfloor \frac{i+j}{2} \rfloor$
6:     return $\max\{\text{maxSum}(i, mid), \text{maxSum}(mid, j), \text{maxGoFrom}(i, mid) + \text{maxGoFrom}(mid, j)\}$
7: end procedure
```

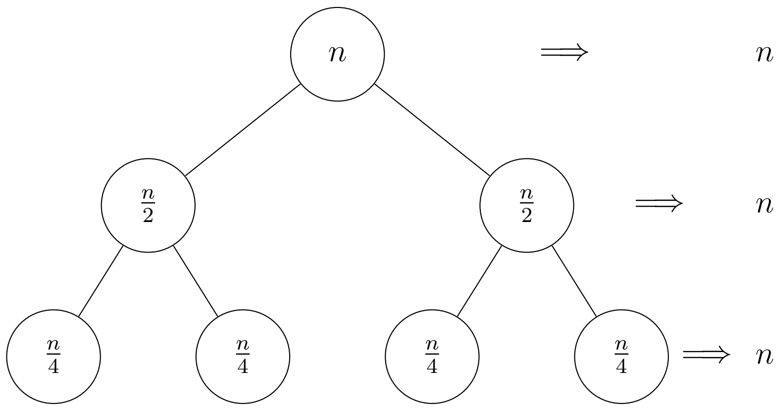
If goes cross, must include $A[mid]$ and $A[mid+1]$.

The running time for algorithm 7 is

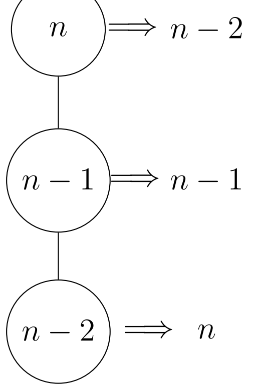
$$T(m) = 2T\left(\frac{m}{2}\right) + m$$

in which $m = j - i + 1$.

Table 3: Comparison Between Previous Two Method



At each level, the running time sum is n.
There are $\log n$ level in total.
Thus, total running time is $\Theta(n \log n)$.


n $\Longrightarrow$ $n-2$
$n-1$ $\Longrightarrow$ $n-1$
$n-2$ $\Longrightarrow$ n

$$\begin{aligned} T(m) &= T(m-1) + \Theta(m) \\ &= \sum_{k=1}^{m} k \\ &= \Theta(m^2) \end{aligned}$$

Can we do better?

The divide & conquer solution constantly calculate the maxEndAt (i, j). Memoizing the result leads to the dynamic programming solution as shown in next section.

6.5 Solving Recurrence

6.5.1 Basic Idea

Fail-safe method for any recurrence:

Guess solution and prove correct with induction.

The subtle point is: need to make the right guess!

Example: Tower of Hanoi, $T(n) = 2T(n-1) + 1$, $T(1) = 1$.

Claim: $T(n) = 2^n - 1$

Proof:

Base Case: $T(1) = 2^1 - 1 = 1$, true.

Induction step: Assume $T(m) = 2^m - 1$, for $m < n$.

$$T(n) = 2T(n-1) + 1 = 2 \times (2^{n-1} - 1) + 1 = 2^n - 1$$

Hold true for n.

Conclusion: $T(n) = 2^n - 1$. $\square$

6.5.2 Typical Proving Method

Example: maxSubArraySum, $T(n) = 2T(n/2) + \Theta(n)$.

Claim: $T(n) = \Theta(n \log n)$

Proof: Still using induction, only show the induction phase below.

Replace original statement with:

$$T(n) = 2T(n/2) + n$$

First, prove $T(n) \le c \times n \log n$.

$$\begin{aligned} T(n) &= 2T\big(\frac{n}{2}\big) + n \\ &\le 2 \times c \times \frac{n}{2} \times \log\big(\frac{n}{2}\big) + n \\ &= c \times n \times \log(n-1) + n \\ &= c \times n \times \log n + n(1-c) \\ &= n \log n \text{ for } c = 1 \end{aligned}$$

Then, prove $T(n) \ge c \times n \log n$. The same as previous one. □

Note: The constant in $T(n) = \mathcal{O}(n \log n)$ muse be specified. And in $T(n) = c \times n \log n - bn$, sometimes bn need to guess.

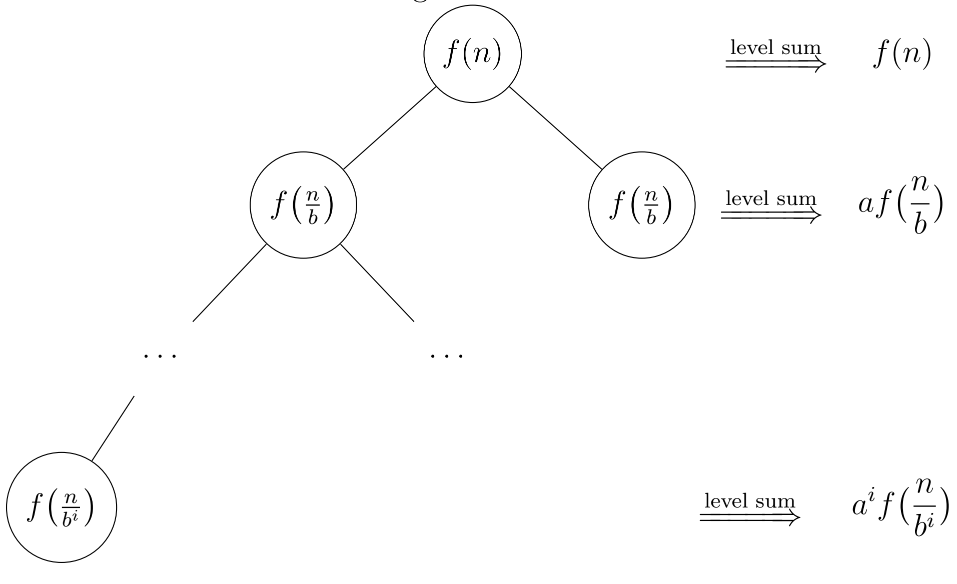
6.5.3 Generalizing: Recursion Tree Method

Generalize the equation $T(n) = T(n/2) + n$ to

$$T(n) = aT\big(\frac{n}{b}\big) + f(n) \tag{10}$$

where $a, b \ge 1$. The recursion tree for eq. (10) is shown in fig. 1.

Figure 1: Generalize Recursion Tree



Observation:

- Level sum: $a^i f\big(\frac{n}{b^i}\big)$

- Depth: $\log_b n$
- Number of Leaves: $a^{\log_b n}$

In total: $T(n) = \sum_{i=1}^{\log_b n} a^i f\big(\frac{n}{b^i}\big)$, which often like a geometric series.
Look at the ratio of successive level sums:

$$\frac{a^{i+1} f(\frac{n}{b^{i+1}})}{a^i f(\frac{n}{b^i})} = a \times \frac{f(\frac{n}{b^{i+1}})}{f(\frac{n}{b^{i+1}})} = \begin{cases} > 1, \text{ then sum determined by the number of leaves.} \\ < 1, \text{ then sum determined by root.} \\ = 1, \text{ then sum determined by the depth or root.} \end{cases} \tag{11}$$

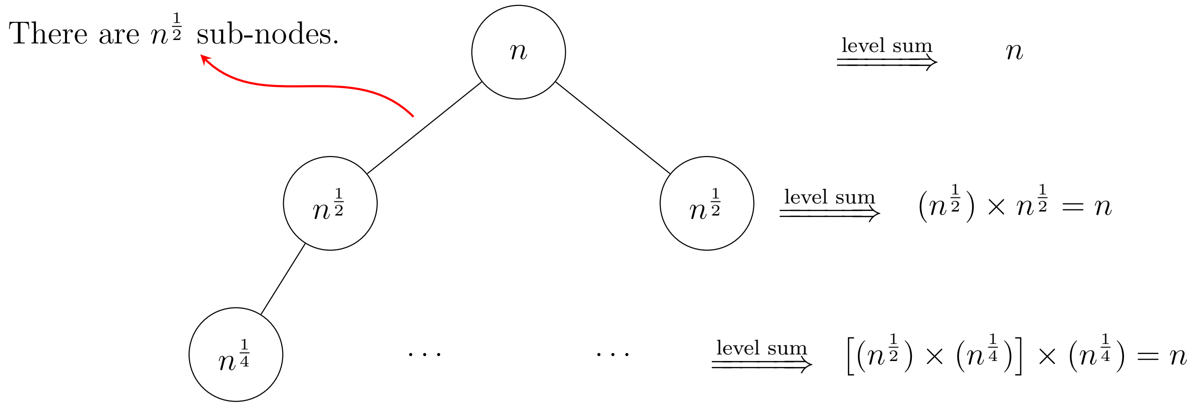
A Special Case is that, $f(n) = n^c$, i.e. $f(n)$ is polynomial.

$$a \times \frac{f(\frac{n}{b^{i+1}})}{f(\frac{n}{b^{i+1}})} = a \times \frac{(\frac{n}{b^{i+1}})^c}{(\frac{n}{b^i})^c} = \frac{a}{b^c} = \begin{cases} > 1, \text{ if } a > b^c, \text{ then } T(n) = a^{\log_b n} = n^{\log_b a} \\ < 1, \text{ if } a < b^c, \text{ then } T(n) = n^c \\ = 1, \text{ if } a = b^c, \text{ then } T(n) = n^c \log_b n \end{cases} \tag{12}$$

The eq. (12) is called the Master Theorem.

6.5.4 More Examples: $T(n) = \sqrt{n}T(\sqrt{n}) + n$

Figure 2: Recursion Tree for $T(n) = \sqrt{n}T(\sqrt{n}) + n$


There are $n^{\frac{1}{2}}$ sub-nodes.
n
level sum
n
$n^{\frac{1}{2}}$
$n^{\frac{1}{2}}$
level sum
$(n^{\frac{1}{2}}) \times n^{\frac{1}{2}} = n$
$n^{\frac{1}{4}}$
$\cdots$
$\cdots$
level sum
$\left[(n^{\frac{1}{2}}) \times (n^{\frac{1}{4}})\right] \times (n^{\frac{1}{4}}) = n$

Observation:

$$\text{Level sums: } \left(\prod_{i=1}^{l} n^{\frac{1}{2^i}}\right) \times \frac{1}{2^i} = n^{\left(\sum_{i=1}^{l} \frac{1}{2^i}\right) + \frac{1}{2^l}} = n^1$$

Equal level sum, then what we need to know is the depth. So the question turns to: How many times apply $\sqrt{n}$ (root function) until get to constant size?

$$
\begin{aligned}
& n^{\frac{1}{2^i}} = c \\
& \Rightarrow \frac{1}{2^i} \log n = \log c \\
& \Rightarrow \log n = 2^i \log c \\
& \Rightarrow \log \log n = i \log 2 + \log \log c \\
& \Rightarrow i = \Theta(\log \log n)
\end{aligned}
$$

In total, the running time would be: $T(n) = \Theta(n \log \log n)$.

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

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

So, the first attempt of the algorithm could be described as algorithm 8.

Algorithm 8 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 8:$T(n) = n + \sum_{i=0}^{n-i} T(i)$, which is clearly **Exponential** since there are a lot of subproblem overlap!

7.1.3 Memoized Version

algorithm 9 illustrates the memoized version of the algorithm in algorithm 8

Algorithm 9 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
            $q = \max(q, P[i] + \text{MemRodCut}(n - i))$
        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 10 illustrates the dynamic programming version of the algorithm according to the memoized version algorithm 9.

Algorithm 10 Dynamic Programming Version of Solving Cutting Rod Problem

```
1: procedure DPRodCut(n)
2:     Let $R[0...n]$ 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:         $R[i] = q$
10:    end for
11:    return $R[n]$
12: end procedure
```

Running time of algorithm 10: $T(n) = \mathcal{O}(n^2)$.

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 10 as:

Algorithm 11 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, $1 \leq i_1 < i_2 < ... < i_k < n$, 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 12 Original Algorithm for LIS Problem

```
1: procedure LIS(prev, start)                         ▷ $prev < start$ ◁
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 ordeer, s.t. $B[\,][start + 1]$ filled before $B[\,][start]$.

7.3.3 Dynamic Programming Version to Solve LIS

algorithm 13 illustrates the dp version to solve LIS problem.

Algorithm 13 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 13 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:

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

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.

$$LCS(A[1 \ldots n], B[1 \ldots m]) = \max \left\{ \begin{array}{l} LCS(A[2 \ldots n], B[1 \ldots m]), \\ LCS(A[1 \ldots n], B[2 \ldots m]) \end{array} \right\} \tag{13}$$

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

$$LCS(A[1 \ldots n], B[1 \ldots m]) = \max \left\{ \begin{array}{l} 1 + LCS(A[2 \ldots n], B[2 \ldots n]), \\ LCS(A[2 \ldots n], B[1 \ldots m]), \\ LCS(A[1 \ldots n], B[2 \ldots m]) \end{array} \right\} \tag{14}$$

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

Algorithm 14 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 15.

Algorithm 15 Dynamic Programming Algorithm for LCS Problem

```
procedure LCSDP($A[1 \ldots n], B[1 \ldots m]$)
    Define $C[1 \ldots n+1][1 \ldots m+1]$
    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.

```
A = { F O O   D }
B = { 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 = {A L G O R   I   T H M }
B = {A L   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;

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

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

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

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

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

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

The algorithm is described in algorithm 16. $\textsc{Edit}(i, j) = edit(A[1 \ldots i], B[1 \ldots j])$.

Algorithm 16 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 $\min\{\textsc{Edit}(i-1, j) + 1, \textsc{Edit}(i, j-1) + 1, \textsc{Edit}(i-1, j-1) + [A[i] \neq B[j]]\}$
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 17.

Algorithm 17 Dynamic Programming Algorithm for Edit Distance Problem

```
1: procedure EDITDP($A[i \ldots m], B[i \ldots n]$)
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 $A[1 \ldots n]$ of integers.

Output: Longest subsequence of indices,

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

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 18 Recursive Algorithm for Longest Convex Subsequence

```
procedure LXS($preprev, prev, cur$)
    if $cur > n$ then
        return 0
    end if                                   Does not change.
    $ignore$ =LXS ( $\underline{preprev, prev}$ , $\underline{cur + 1}$ )     Consider next element.
    $best = ignore$
    if $preprev = -1$ or $A[cur] > 2 \times A[prev] - A[preprev]$ then
        $include = 1 + \text{LXS}(prev, cur, cur + 1)$
        if $include > ignore$ then
            $best = include$
        end if
    end if
    return $best$
end procedure
```

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

```
procedure LIS($prev, start$)
    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
```

Claim: $LIS(prev, start)$, for $prev < start$, returns the longest increasing subsequence in $A[start \ldots n]$ 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. $A[start + 1 \ldots n]$), 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 $A[1 \ldots n]$ we call $LIS(0, 1)$ as this is the LIS in $A[1 \ldots n]$ s.t. all elements $> -\infty$, which is the same as the LIS of $A[1 \ldots n]$.

Applying DP: $LIS(prev, start)$ depends on two parameters, each ranging over $O(n)$ values, as they are both indices into $A[0 \ldots n]$. 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 large 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 $1 \leq i_1 \leq \dots i_k \leq n$ and $1 \leq j_1 \leq \dots j_k \leq m$ 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 $curA, curB > 0$

```
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 $A[curA] = A[curB]$ then
        $include = 1 + LIS(curA + 1, curB + 1)$
        if $include > ignore$ then
            $best = include$
        end if
    end if
    return $best$
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 $A[1 \ldots n]$ and $B[1 \ldots m]$, 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 $O(n^2)$.

```
procedure LCSDP($A[1 \ldots n], B[1 \ldots m]$)
    Define $C[1 \ldots n+1][1 \ldots m+1]$
    for $i = 1$ to $n + 1$ do
        $C[i][m + 1] = 0$
    end for
    for $i = 1$ 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 $A[curA] = A[curB]$ 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
```

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.

```
procedure LXS($preprev, prev, cur$)
    if $cur > n$ then
        return 0
    $ignore = LXS(preprev, prev, cur + 1)$
    $best = ignore$
    if $(preprev = -1) \vee (A[cur] > 2A[prev] - A[preprev])$ then
        $include = 1 + LXS(prev, cur, cur + 1)$
        if $include > ignore$ then
            $best = include$
    return $best$
```

$LXS(preprev, prev, cur)$ computes the length of the longest convex subsequence in the array $A[cur \dots n]$, but with the requirement that the sequence is still convex after appending $A[preprev]A[prev]$ to the front (unless $preprev = -1$, in which case there is no additional requirement involving $A[preprev]A[prev]$). 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 $A[cur]$ 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.

```
procedure SCS($curA, curB$)
    if $curA > m$ then
        return $n - curB + 1$
    if $curB > n$ then
        return $m - curA + 1$
    $different = \min\{1 + SCS(curA + 1, curB), 1 + SCS(curA, curB + 1)\}$
    $best = different$
    if $A[curA] = B[curB]$ then
        $match = 1 + SCS(curA + 1, curB + 1)$
        if $match < different$ then
            $best = match$
    return $best$
```

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

```
procedure SCS($curA, curB$)
    if $curA > m$ then
        return $n - curB + 1$
    if $curB > n$ then
        return $m - curA + 1$
    if $A[curA] = B[curB]$ then
        return $1 + SCS(curA + 1, curB + 1)$
    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).

8 Greedy Algorithm

8.1 Greedy in a Glance

Greedy Algorithm is repeatedly taking locally optimal step in hopes reaching global optimal solution.

Often Greedy is WRONG!

So we should assume greedy is wrong, unless proven otherwise.

However, if we can prove greedy works in some circumstance, it would be a simple and fast solution in most cases.

8.2 Greedy Class Scheduling

8.2.1 Description of Problem

Target: On given day of week, we want to take as many classes as possible. Constraint: We cannot take two classes whose times overlap.

Input: Given $S[1\dots n]$ and $F[1\dots n]$, which

$$S[i] = \text{start time of class} i$$

$$F[i] = \text{finish time of class} i$$

Output: Select largest subset $X \subseteq \{1\dots n\}$, s.t. for $i \neq j \in X$ either $S[i] > F[j]$ or $S[j] > F[i]$.

8.2.2 Analysis

Recursive approach: consider class 1.

$$B_4 = \{i | 2 \leq i \leq n \text{ and } F[i] < S[1]\}$$

$$L_8 = \{i | 2 \leq i \leq n \text{ and } S[i] > F[1]\}$$

DP approach: $\mathcal{O}(n)$ running time.

But we can do better with greedy: Pick class which finishes first.

In other word, the algorithm is:

> Scan classes in increasing order of finishing time. Each time encounter non-conflicting class, select it.

as described in algorithm 19.

Algorithm 19 Greedy Solution for Class Scheduling

```
1: procedure GreedySchedule($S[1 \dots n]$, $F[1 \dots n]$)
2:     Sort $F$ and permute $S$ to match.
3:     $count = 1$
4:     $X[count] = 1$                                    ▷ $X[1 \dots n]$ ◁
5:     for $i = 2$ to $n$ do
6:         if $S[i] > F[X[count]]$ then
7:             $count = count + 1$
8:             $X[count] = i$
9:         end if
10:    end for
11:    return $X[1 \dots count]$
12: end procedure
```

Running time: $\Theta(n \log n) \rightarrow$ sorting running time.

8.2.3 Proof

Note that there may be many optimal solutions. And Greedy produces unique solution. So we cannot argue any optimal solution looks like greedy.

Lemma 8.2.1. *At least one optimal solution include class that finishes first.*

Proof: Let f be class that finishes first, X be an optimal set of classes. If $f \in X$ then lemma proven. Otherwise, $f \notin X$. Let g be class in X that finishes first. Since f finishes before g, f cannot conflict with any class in $X \setminus \{g\}$

Thus $X' = X \cup \{f\} \setminus \{g\}$ is a valid solution with max size and contain f. □

Theorem 8.2.2. *Greedy schedule is optimal.*[2]

Proof: Let f be class that finishes first. L is set that starts after f finishes. Best schedule containing f is optimal. This best schedule must be optimal on L. L is strictly smaller set of classes, so can apply induction. □

[2]Can be proved using induction

8.3 General Prove Method for Greedy Algorithm: Exchange Argument

1. Assume there is a optimal solution other than greedy algorithm.
2. Fine the “first” difference between it and greedy algorithm.
3. Argue can exchange optimal choice for greedy one, without degrading solution value.