

Notes on CS6363

Hanlin He

Sunday December 11, 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
6.4.3	Applying Divide & Conquer on Max Sub-Array Sum Problem	13
6.5	Solving Recurrence	15
6.5.1	Basic Idea	15
6.5.2	Typical Proving Method	15
6.5.3	Generalizing: Recursion Tree Method	16
6.5.4	More Examples: $T(n) = \sqrt{n}T(\sqrt{n}) + n$	17
7	Dynamic Programming	19
7.1	Rod Cutting	19
7.1.1	Description of Problem	19
7.1.2	Analysis	19
7.1.3	Memoized Version	20
7.1.4	Dynamic Programming Version to Solve RodCut	20
7.2	Solving DP problem	21
7.3	Longest Increasing Subsequence (LIS)	22
7.3.1	Description of Problem	22
7.3.2	Analysis	22
7.3.3	Dynamic Programming Version to Solve LIS	23
7.4	Longest Common Subsequence	24
7.4.1	Description of Problem	24
7.4.2	Analysis	25
7.4.3	Dynamic Programming Version to Solve LCS	26
7.5	Edit Distance	27
7.5.1	Description of Problem	27
7.5.2	Analysis	28
7.5.3	Dynamic Programming Version to Solve Edit Distance Problem . . .	30
7.6	Longest Convex Subsequence	30
7.6.1	Description of Problem	30
7.6.2	Analysis	31
7.7	Professor's note on LIS/LCS/LXS/SCS	31
7.8	Summary	38
8	Greedy Algorithm	39
8.1	Greedy in a Glance	39
8.2	Greedy Class Scheduling	39
8.2.1	Description of Problem	39

8.2.2	Analysis	39
8.2.3	Proof	40
8.3	General Prove Method for Greedy Algorithm: Exchange Argument	41
8.4	Huffman Codes	41
8.4.1	Description of Problem	41
8.4.2	Definition	43
8.4.3	Analysis	43
9	Graph Algorithm	45
9.1	Basic Stuff	45
9.1.1	Fundamental Concept	45
9.1.2	Concepts Used in this Note	45
9.1.3	Graph Data Structure	46
9.2	Traversing Graph	47
9.2.1	Intuitive Approach	47
9.2.2	Generic Traverse Algorithm	48
9.2.3	Usage of Using Generic Travers Algorithm	50
9.3	Standard Graph Algorithm	51
9.4	Depth First Search	51
9.4.1	Formal Definition	51
9.4.2	Count and Label Components	52
9.4.3	Pre/Post Order Traverse of Binary Tree	53
9.4.4	Directed Graphs and Reachability	54
9.5	Directed Acyclic Graphs	55
9.5.1	Definition of DAG	55
9.5.2	Algorithm to Determine a DAG	55
9.5.3	Proof of Correctness	56
9.6	Topological Sort	57
9.7	Strong Connectivity	59
9.8	Minimum Spanning Tree (MST)	59
9.8.1	MST Definition	59
9.8.2	MST Property	60
9.8.3	MST Algorithm (General)	61
9.8.4	Prim's Algorithm	61
9.8.5	Borvka's Algorithm	63
9.8.6	Kruskal's Algorithm	65

9.9	Shortest Paths	66
9.9.1	Single Source Shortest Path (SSSP)	67
9.9.2	SSSP Algorithm	67
9.9.3	Generic SSSP algorithm	67
9.9.4	Dijkstra's Algorithm	69
9.9.5	Bellman-Ford Algorithm	70
9.9.6	Some Intuition on SSSP	71
9.10	Max Flows and Min Cuts	71
9.10.1	Flows	71
9.10.2	Cuts	72
9.10.3	The Maxflow MinCut Theorem	73
9.10.4	Ford-Fulkerson Algorithm	75
9.10.5	Application of Max Flow	76
10	NP Hardness	78

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	Recursive Solution for Maximum Sub-Array Sum Problem	13
7	Divide & Conquer Solution for Maximum Sub-Array Sum Problem	14
8	First Attempt of Solving Cutting Rod Problem	19
9	Memoized Version of Solving Cutting Rod Problem	20
10	Dynamic Programming Version of Solving Cutting Rod Problem	21
11	Store the Cutting Position in the Process	21
12	Original Algorithm for LIS Problem	23
13	Dynamic Programming Algorithm for LIS Problem	24
14	Original Algorithm for LCS Problem	26
15	Dynamic Programming Algorithm for LCS Problem	27
16	Original Algorithm for Edit Distance	29
17	Dynamic Programming Algorithm for Edit Distance Problem	30
18	Recursive Algorithm for Longest Convex Subsequence	31
19	Greedy Solution for Class Scheduling	40
20	Recursive Depth First Search Algorithm	47
21	Iterative Depth First Search Algorithm	48
22	Generic Traverse Algorithm	49
23	Remember Parent version of Generic Traverse Algorithm	50
24	Wrapper for Traverse	51
25	Formal Definition of DFS	52
26	DFS All the Component	52
27	Count and Label the Components	53
28	Pre/Post Order Based on DFS	54
29	Determine Whether a Graph is DAG	56
30	Topo-Sort from Source	58
31	Topo-Sort from Sink	58
32	Algorithm to Compute Strong Components	59
33	Prim's Algorithm	62
34	Borvka's Algorithm	64

35	Kruskal's Algorithm	66
36	Relax Operation in SSSP Algorithm	68
37	Generic SSSP Algorithm	69
38	Bellman-Ford Algorithm	70

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^{10} are the same. \square

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

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

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.

$$T(0) = 0$$

$$T(1) = 1$$

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

“Solve” the recurrence.

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

6.3 Binary Search

Input: Given a value *val*, and sorted 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 =  $\lfloor \frac{high+low}{2} \rfloor$ 
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 . . . *n*], call BIN (*val*, 1, *n*).

Let $m = high - low + 1$,

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

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

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

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

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 $maxSum(1, n - 1)$?

- $maxSum(1, n)$ 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:    $x = \text{MAXENDAT}(i, j)$ 
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  $\max\{A[j], A[j] + \text{MAXENDAT}(i, j - 1)\}$ 
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  $\max \left\{ \begin{array}{l} \text{MAXSUM}(i, mid), \\ \text{MAXSUM}(mid, j), \\ \text{MAXGOFROM}(i, mid) + \text{MAXGOFROM}(mid, j) \end{array} \right\}$ 
7: end procedure

```

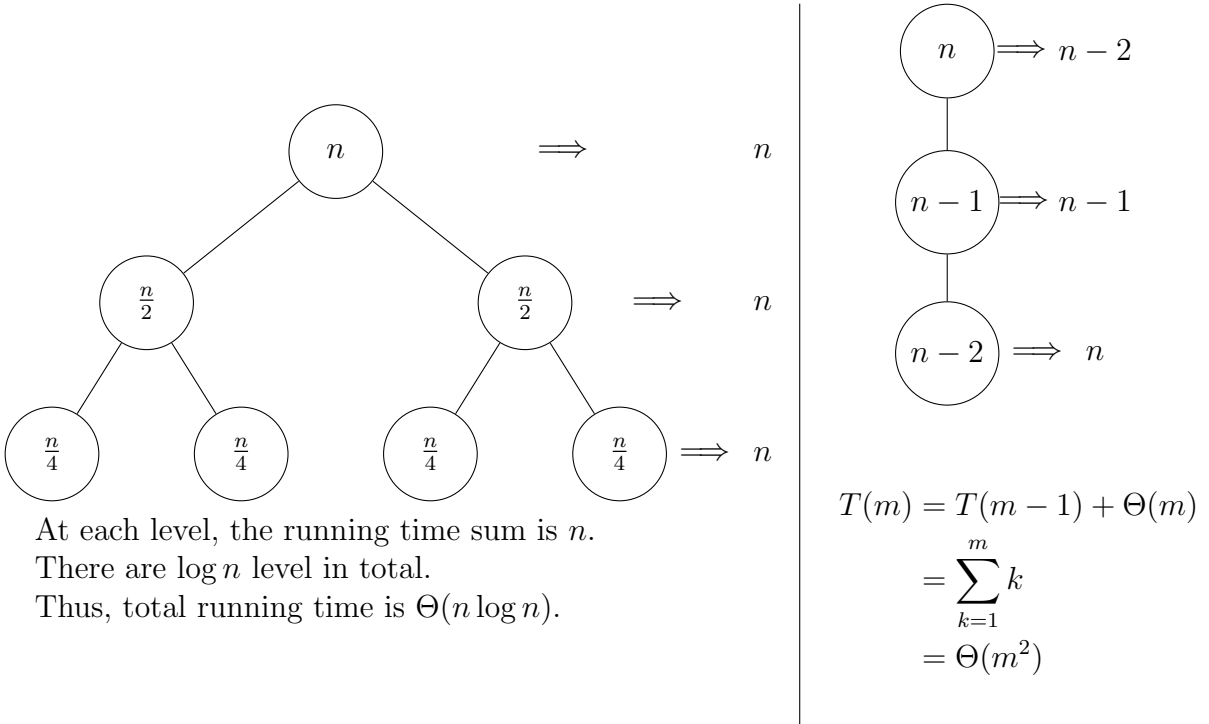
If goes cross, must include $A[mid]$ and $A[mid + 1]$.

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



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 $T(n) \leq c \times n \log n$.

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

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

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

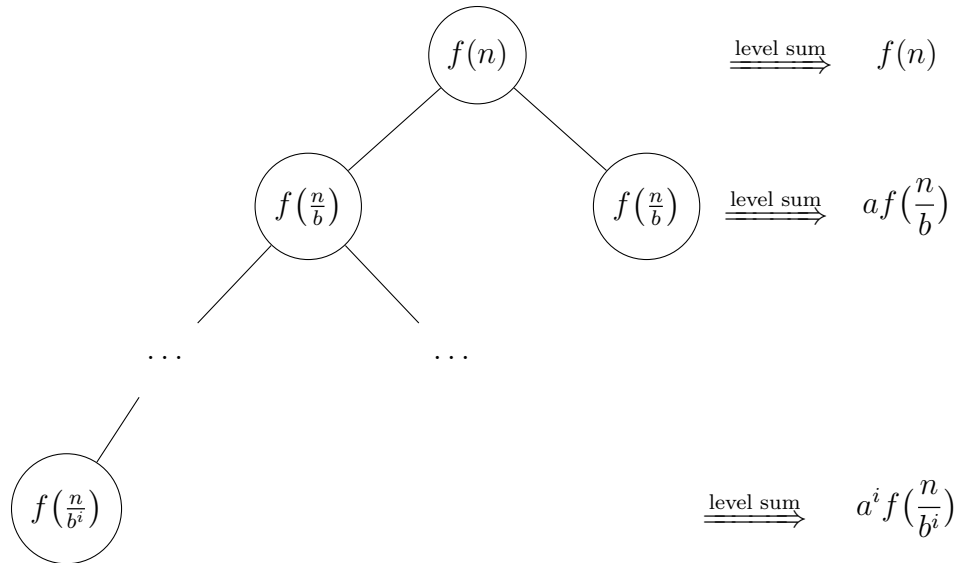
6.5.3 Generalizing: Recursion Tree Method

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

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \quad (10)$$

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

Figure 1: Generalize Recursion Tree



Observation:

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

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

In total: $T(n) = \sum_{i=1}^{\log_b n} a^i f\left(\frac{n}{b^i}\right)$, which often like a geometric series.

Look at the ratio of successive level sums:

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

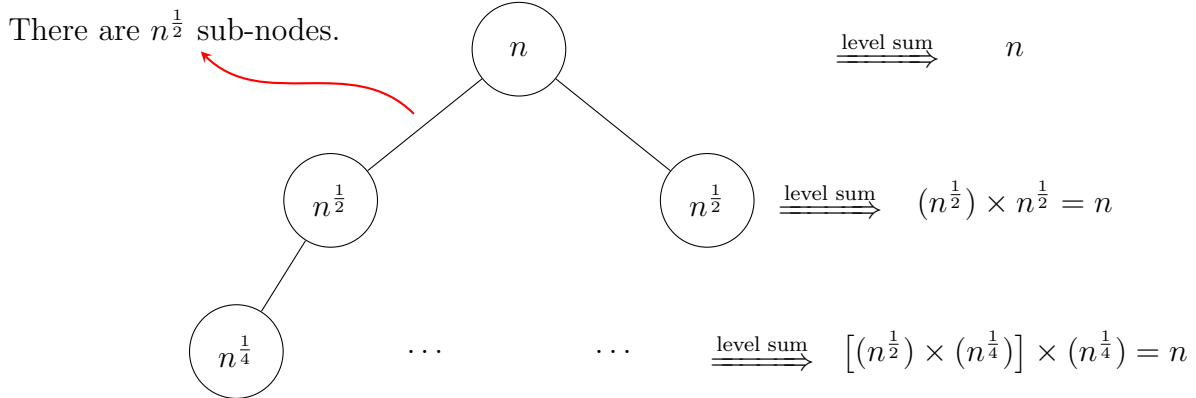
A Special Case is that, $f(n) = n^c$, i.e. $f(n)$ is polynomial.

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

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 $T(n) = \sqrt{n}T(\sqrt{n}) + n$



Observation:

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

Equal level sum, then what we need to know is the depth. So the question turns to:
How many times apply $\sqrt[n]{n}$ (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

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

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

```
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 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  $R[0\dots n]$  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:      $R[i] = q$ 
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 $C[1\dots n]$, 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, $1 \leq i_1 < i_2 < \dots < i_k < n$, 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 12 Original Algorithm for LIS Problem

```

1: procedure LIS(prev, start)  $\triangleright prev < start \triangleleft$ 
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 13 illustrates the dp version to solve LIS problem.

Algorithm 13 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 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:

$$A = \{ \text{ C G C A A T C C A G G } \}$$

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

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.

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

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

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

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 $A[curA \dots n]$ and $B[curB \dots m]$.

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

```
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 16. $\text{EDIT}(i, j) = \text{edit}(A[1 \dots i], B[1 \dots j])$.

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  $\min\{\text{EDIT}(i-1, j) + 1, \text{EDIT}(i, j-1) + 1, \text{EDIT}(i-1, j-1) + [A[i] \neq B[j]]\}$ 
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 17.

Algorithm 17 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 18 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).

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

$S[i]$ = start time of class i

$F[i]$ = 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:    $X[count] = 1$   $\triangleright X[1 \dots n] \triangleleft$ 
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 . \square

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. \square

²Can 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.

8.4 Huffman Codes

8.4.1 Description of Problem

Input: Given an alphabet Σ .

Output: A binary code for Σ maps every $x \in \Sigma$ to a unique binary string.

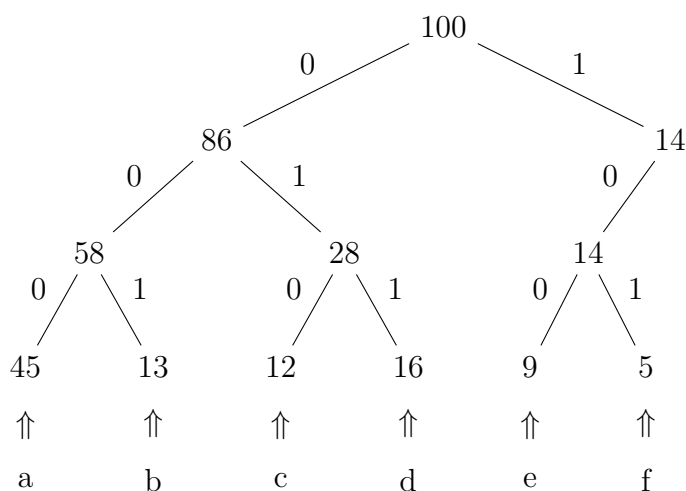
Example: Fixed length codes: Map Σ to strings all having same length.

Given Alphabet: $\Sigma = \{a, b, c, d, e, f\}$

Frequency for each character $F = \{45, 13, 12, 16, 9, 5\}$.

$a = 000$	$b = 001$	$c = 010$
$d = 011$	$e = 100$	$f = 101$

Figure 3: Binary Code Tree for Fixed Length Codes



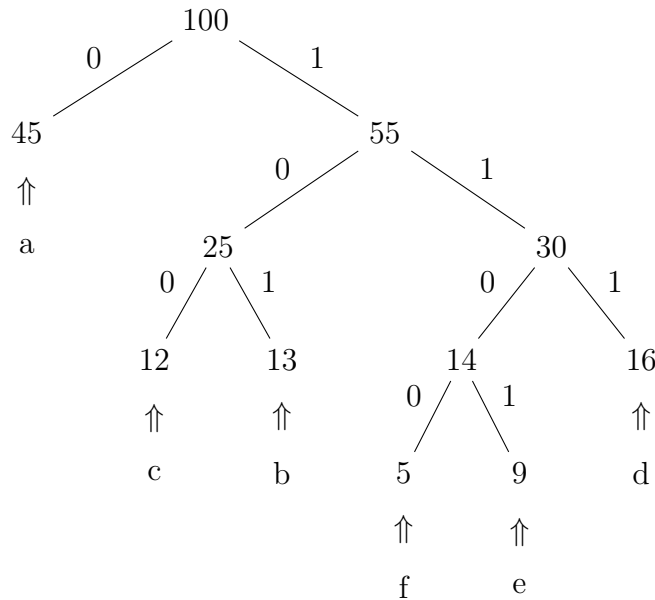
The length of coded files:

$$\begin{aligned}
 & 3 \times 45 + 3 \times 13 + 3 \times 12 + 3 \times 14 + 3 \times 16 + 3 \times 9 + 3 \times 5 \\
 &= \sum F[i] \times \text{depth}(i)
 \end{aligned}$$

What if the codes is not fixed length?

$$\begin{array}{lll}
 a = 0 & b = 101 & c = 100 \\
 d = 111 & e = 1101 & f = 1100
 \end{array}$$

Figure 4: Binary Code Tree for Not Fixed Length Codes



The length of coded files:

$$\begin{aligned}
 & 1 \times 45 + 3 \times 13 + 3 \times 12 + 3 \times 16 + 4 \times 9 + 4 \times 5 \\
 &= 224
 \end{aligned}$$

Note that, in order to decode, the codes need to be prefix-free, i.e. no code word is a prefix of another, which means characters must at leaves.

So here comes the greedy algorithm: Huffman Codes.

8.4.2 Definition

Input: Alphabet Σ of size n and the frequency array $F[1 \dots n]$, $F[i]$ is the frequency of character $\Sigma[i]$.

Output: Find a prefix free tree, s.t. $\sum F[i] \times \text{depth}(i)$ minimized.

Huffman Algorithm: Recursively merge two least frequent.

8.4.3 Analysis

Claim 8.4.1. *Huffman's Algorithm is optimal.*

Observation: Optimal code trees are full binary tree (gbt³). In other word, leaf at maximum depth has a sibling.

Lemma 8.4.2. *Let x and y be two least frequent characters. There is an optimal tree where x, y are siblings.*

Proof: Let T be an optimal tree of depth d .

By observation, there are siblings a, b at depth d . If $a = x$ and $b = y$, then proved.

Let T' be the tree obtained by swapping x and a .

Then the depth of x increased by $d - \text{depth}(x, T) = z$, and the depth of a decreased by z , so,

$$\text{Cost}(T') = \text{Cost}(T) - (F[a] - F[x]) \times z$$

where x was least frequent. Thus

$$F[a] - F[x] \geq 0 \Rightarrow \text{Cost}(T') \leq \text{Cost}(T)$$

Based on assumption, T is a optimal tree, so $\text{Cost}(T) \leq \text{Cost}(T')$.

Hence, T equals to T' \square

Theorem 8.4.3. *Huffman codes is optimal.*

Proof: Assume $F[1], F[2]$ is least frequent. Define $F[n+1] = F[1] + F[2]$.

Let T' be the Huffman tree on $F[3 \dots n+1]$, T be the Huffman tree on $F[1 \dots n]$ obtained from T' by replacing $n+1$ with 1 and 2.

³Definition is in homework 1.

For any prefix tree \overline{T} obtained from any $\overline{T'}$ on $F[3 \dots n + 1]$ by replacing $n + 1$ with 1 and 2 can prove

$$Cost(\overline{T}) = Cost(\overline{T'}) + F[1] + F[2]$$

Use Induction:

Base Case: For one or two characters, Huffman is optimal. Now want to prove Huffman tree T is optimal.

Induction step: Assume T' is an optimal tree, then $Cost(\overline{T}) = Cost(\overline{T'}) + F[1] + F[2]$, which means minimizing $Cost(\overline{T})$ is equivalent to minimizing $Cost(\overline{T'})$.

T is optimal among trees on $\Sigma[1 \dots n]$ obtained by an expanding $n + 1$ node to 1 and 2 from an optimal tree on $F[3 \dots n + 1]$.

In conclusion, Huffman is optimal. \square

Greedy Ends Here.

9 Graph Algorithm

9.1 Basic Stuff

9.1.1 Fundamental Concept

The following is the basic concept of graph.

1. A **graph** is a pair $G = (V, E)$.
2. V is set of vertices or nodes.
3. E is set of edges. An edge e in E is a pair of vertices $e = \{u, v\}$.
4. If G is undirected, e is unordered, i.e. $e = uv = vu$.
If G is directed, e is ordered, i.e. $e = u \rightarrow v$ or (u, v) .
5. Graphs represent relations between pairs of objects.

In this course, we mainly consider simple graphs, no multi-edges and no self-loop.

9.1.2 Concepts Used in this Note

The following conclude the terminology and notations used in this notes.

1. The **degree** of v is the number of adjacent edges.
2. n denotes $|v|$, m denotes $|E|$.
3. For undirected graph: $m \leq \binom{n}{2}$. For directed graph: $m \leq 2\binom{n}{2}$.
4. Sub-graph of $G = (V, E)$ is a graph $G' = (V', E')$, s.t. $V' \subseteq V$, $E' \subseteq E$.
5. A **walk** is a sequence $v_1 \dots v_l$, s.t. $v_i \in V$ and $v_i v_{i+1} \in E$.
6. A **path** is a walk where v_i distinct.
7. A walk is close, if $v_i = v_k$.
8. A cycle is a “closed path”.
9. Undirected graph connected if path between every pair $u, v \in V$.
10. If not connected, a component is a maximal connect sub-graph.

11. A **tree** is a connected “acyclic” graph.
12. A **forest** is a graph where each component is a tree.
13. A **Spanning Tree** of G is a sub-graph that is a tree and contains all vertices of G .
14. For directed graphs: directed path/cycles.
15. A graph is **Strongly Connected** if directed path from any vertex to another exists.
16. Directed acyclic graph is called a **DAG**.

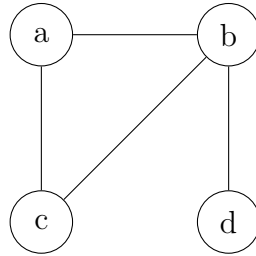
9.1.3 Graph Data Structure

There are two widely used data structure of graph:

- Adjacency Matrix: $|V| \times |V|$ matrix, where $A[i, j] = 1$ if $(i, j) \in E$.
- Adjacency List: an array of length $|V|$, where every entry in the array stores a list of neighbors.

For example:

Figure 5: Example Graph



The adjacency matrix and adjacency list for fig. 5 would be:

	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	0	0
D	0	1	0	0

Table 4: Adjacency Matrix

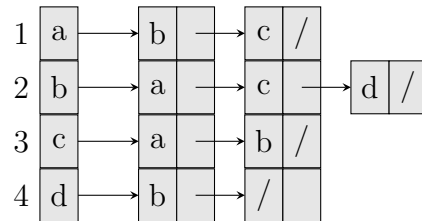


Figure 6: Adjacency List

The running time for these two data structures are shown in table 5.

Table 5: Running Time Analysis for Adjacency Matrix and Adjacency List

Time	Adjacency Matrix	Adjacency List
visit neighbor	$\mathcal{O}(1)$	$\mathcal{O}(1)$
visit all neighbors	$\mathcal{O}(n)$	$\mathcal{O}(\text{degree}(v))$
test if $uv \in E$	$\mathcal{O}(1)$	$\mathcal{O}(\text{degree}(v))$
add/delete edge	$\mathcal{O}(1)$	$\mathcal{O}(n)$
size	$\mathcal{O}(mn)$	$\mathcal{O}(m + n)$

$\mathcal{O}(1)$ with hashing.

9.2 Traversing Graph

9.2.1 Intuitive Approach

Assume G is a connected, undirected graph. Suppose we want to visit every vertex in G , we can use Depth First Search (DFS) in a recursive manner, as shown in algorithm 20.

Algorithm 20 Recursive Depth First Search Algorithm

```

1: procedure RECURSIVEDFS( $v$ )
2:   if  $v$  unmarked then
3:     Mark  $v$  ▷ has  $v$  been visited. ◁
4:     for each edge  $vw$ . do
5:       RECURSIVEDFS ( $w$ )
6:     end for
7:   end if
8: end procedure

```

Initially, call RECURSIVEDFS (s), where $s \in V$ is some start vertex.

Note that the functions was called recursively, forming an implicit stack. We can re-write as an iteration using an explicit stack, as shown in algorithm 21.

Algorithm 21 Iterative Depth First Search Algorithm

```
1: procedure ITERATIVEDFS( $s$ )
2:   PUSH ( $s$ )
3:   while stack not empty do
4:      $s = \text{POP}()$ 
5:     if  $v$  is unmarked. then
6:       Mark  $v$ 
7:       for each edge  $vw$  do
8:         if  $w$  is unmarked then      ▷ This if is optional in some cases. ◁
9:           PUSH ( $w$ )
10:        end if
11:      end for
12:    end if
13:  end while
14: end procedure
```

Observation: Traversal algorithms store candidate vertices in a “bag”, which can be any structure allowing insertion/deletion.

9.2.2 Generic Traverse Algorithm

So we can define this generic traverse algorithm as shown in algorithm 22.

Algorithm 22 Generic Traverse Algorithm

```
1: procedure TRAVERSE( $s$ )
2:   Put  $s$  in bag
3:   while bag is not empty do
4:     Take  $v$  from bag.
5:     if  $v$  is unmarked then
6:       Mark  $v$ 
7:       for each edge  $vw$  do
8:         if  $w$  is unmarked then
9:           Put  $w$  in bag.
10:        end if
11:      end for
12:    end if
13:  end while
14: end procedure
```

This step is ambiguous.

▷ This *if* is optional in some cases. ◁

In other application, this line may not be needed, as for traversing, it is needed.

Observation: This traverse algorithm clearly marks each vertex at most once. Now we want to prove it marks each vertex at least once, i.e. “visit” every vertex. To do this, we can remember each vertex’s parent in the process. The modified algorithm is shown in algorithm 23

Algorithm 23 Remember Parent version of Generic Traverse Algorithm

```
1: procedure TRAVERSE( $s$ )
2:   Put  $(\emptyset, s)$  in bag
3:   while bag is not empty do
4:     Take  $(p, v)$  from bag  $\triangleright \mathcal{O}(1)$  per round.  $\triangleleft$ 
5:     if  $v$  is unmarked then  $\triangleright \mathcal{O}(m)$  overall, regardless of inner for  $\triangleleft$ 
6:       Mark  $v$ 
7:        $Parent(v) = p$ 
8:       for each edge  $vw$  do  $\triangleright \mathcal{O}(m)$  overall.  $\triangleleft$ 
9:         if  $w$  is unmarked then
10:          Put  $(v, w)$  in bag
11:        end if
12:      end for
13:    end if
14:  end while
15: end procedure
```

Lemma 9.2.1. TRAVERSE (s) marks every vertex in connected graph exactly once.

Set of pair $(parent(v), v)$ with $parent \neq \emptyset$ defines spanning tree.

Proof: s is marked, so let $v \neq s$.

Let (s, \dots, u, v) be shortest path from s to v , use induction on path length. By induction, u is visited, which implies that when u is marked, (u, v) is put in the bag.

Call a pair $(parent(v), v)$, s.t. $parent(v) \neq \emptyset$ a parent edge. Consider the path $(v, parent(v), parent(parent(v)), \dots)$. This path eventually lead to s . If every vertex has path to s , then parent edge define connected spanning graph. \square

Running Time for Generic Traverse Algorithm

Each vertex marked exactly once. Each edge (v, w) added to bag at most once by v and at most once by w .

Inner loop executed at most $\mathcal{O}(m)$ time. Outer loop $\mathcal{O}(m)$ time. In total, $\mathcal{O}(m)$ time.

Note that, this running time analysis ignore bag operation time (consider $\mathcal{O}(1)$ for all bag operations). If bag operation takes b per operation, $\mathcal{O}(mb)$ total time is required.

For connected graph, $n = \mathcal{O}(m)$.

9.2.3 Usage of Using Generic Travers Algorithm

1. Bag is stack \implies DFS (results in a long skinny tree).

2. Bag is queue \implies BFS (results in a short fat tree).
3. If edges have weight, and bag is min priority queue on edge weights.
 \implies MST, Prim's Algorithm.
4. Bag is min priority queue on vertex distance.
 \implies Dijkstra's Algorithm, shortest path tree.

If G is disconnected, we can use a wrap functions to traverse, as shown in algorithm 24.

Algorithm 24 Wrapper for Traverse

```

1: procedure TRAVERSEALL
2:   for all vertices  $v$  do
3:     if  $v$  is unmarked then
4:       TRAVERSE ( $v$ )
5:     end if
6:   end for
7: end procedure

```

Lemma 9.2.2. TRAVERSEALL () marks every vertex in disconnected graph exactly once.
Set of pair $(parent(v), v)$ with $parent \neq \emptyset$ defines spanning forest.

9.3 Standard Graph Algorithm

Step 1 Maintain set S of visited vertices, and $T \setminus S$ of unvisited.

Step 2 Maintain tree over S .

Step 3 Repeat:

- (a) Pick vertex w in T , which is adjacent to S .
- (b) Add to S , remove from T , update tree on S .

9.4 Depth First Search

9.4.1 Formal Definition

The Formal definition of DFS is shown in algorithm 25.

Algorithm 25 Formal Definition of DFS

```
1: procedure DFS( $v$ )
2:   Mark  $v$ 
3:   for each edge  $vw$  do
4:     if  $w$  is unmarked then
5:       DFS ( $w$ )
6:     end if
7:   end for
8: end procedure
```

Observation: The procedure create a similar stack of that in algorithm 22, but it's the program's stack. When DFS (w) makes recursive call, it stores current program on program stack, i.e. when DFS (v) calls DFS (w), DFS (v) is on stack.

Given v in undirected G , DFS (v) visits all vertices in v 's component.

Algorithm 26 DFS All the Component

```
1: procedure DFSALL( $G$ )
2:   for all  $v \in V$  do
3:     if  $v$  is unmarked then
4:       DFS ( $v$ )
5:     end if
6:   end for
7: end procedure
```

9.4.2 Count and Label Components

DFS Algorithm can be modified to count and label all the components.

Algorithm 27 Count and Label the Components

```
1: procedure COUNTANDLABEL( $G$ )
2:    $count = 0$ 
3:   for all  $v \in V$  do
4:     if  $v$  is unmarked. then
5:        $count = count + 1$ 
6:       LABELCOMPONENT ( $v, count$ )
7:     end if
8:   end for
9:   return  $count$ 
10: end procedure
11: procedure LABELCOMPONENT( $v, count$ )
12:   Mark  $v$ 
13:    $component(v) = count$ 
14:   for each  $vw$  do
15:     if  $w$  is unmarked then
16:       LABELCOMPONENT ( $w, count$ )
17:     end if
18:   end for
19: end procedure
```

9.4.3 Pre/Post Order Traverse of Binary Tree

Originally, the definition are:

- Pre-order: visit, recursive call on left, recursive call on right.
- Post-order: recursive call on left, recursive call on right, visit.

Consider the problem in the view of DFS:

- Pre-order: vertices put on stack;
- Post-order: vertices taken off from stack.

A modified algorithm is shown in algorithm 28

Algorithm 28 Pre/Post Order Based on DFS

```
1: procedure DFSALL( $G$ )
2:    $clock = 0$ 
3:   for all  $v \in V$  do
4:     if  $v$  is unmarked then
5:       DFS ( $v$ )
6:     end if
7:   end for
8: end procedure
9: procedure DFS( $v$ )
10:  Mark  $v$ 
11:  PREVISIT ( $v$ )
12:  for each edge  $vw$  do
13:    if  $w$  is unmarked then
14:      PREVISIT ( $w$ )
15:    end if
16:  end for
17:  POSTVISIT ( $v$ )
18: end procedure
19: procedure PREVISIT( $v$ )
20:   $pre(v) = clock$ 
21:   $clock = clock + 1$ 
22: end procedure
23: procedure POSTVISIT( $v$ )
24:   $post(v) = clock$ 
25:   $clock = clock - 1$ 
26: end procedure
```

9.4.4 Directed Graphs and Reachability

- For undirected graph, DFS (v) explores the component of v .
- For directed graph, it is trickier.
 - For v, u in directed graph, u is reachable from v , if directed path from v to u .
 - Let REACH (v) be set of vertices reachable from v .
 - If $v \in \text{REACH}(v)$, v may or may not be in REACH (u).

9.5 Directed Acyclic Graphs

9.5.1 Definition of DAG

A **DAG** is a directed graph with no directed cycles.

- For a DAG, if $u \in \text{REACH}(v)$, then $v \notin \text{REACH}(u)$.
- A source is a vertex with no incoming edges.
- A sink is a vertex with no outgoing edges.
- Every DAG has at least one source and at least one sink.
- Checking if graph is a DAG need $\mathcal{O}(n + m)$ time using DFS.

To make concept simpler:

- Let G be a directed graph.
- G' obtained by adding source s , a directed edge from s to all vertices of G .
- G is a DAG $\iff G'$ is a DAG (if and only if).

9.5.2 Algorithm to Determine a DAG

To illustrate in a program manner, consider 3 status during the traversal.

- NEW: never been of program stack;
- ACTIVE: is currently on the stack;
- DONE: is off stack, but was previously on.

Thus, the DFS algorithm can be modified as shown in algorithm 29

Algorithm 29 Determine Whether a Graph is DAG

```
1: procedure ISACYCLIC( $G$ )
2:   Add vertex  $s$ 
3:   for all  $v \neq s$  in  $V$  do
4:     Add edge  $s \rightarrow v$ 
5:     STATUS ( $v$ ) = NEW
6:   end for
7:   return ISACYCLICDFS ( $s$ )
8: end procedure
9: procedure ISACYCLICDFS( $v$ )
10:  STATUS ( $v$ ) = ACTIVE
11:  for each edge  $v \rightarrow w$  do
12:    if STATUS ( $w$ ) = ACTIVE then
13:      return False
14:    else if STATUS ( $w$ ) = NEW then
15:      if ISACYCLICDFS ( $w$ ) = False then
16:        return False
17:      end if
18:    end if
19:  end for
20:  STATUS ( $v$ ) = DONE
21:  return True
22: end procedure
```

9.5.3 Proof of Correctness

Prove “False” implies cycle.

Suppose “False” is returned, then find edge $v \rightarrow w$, s.t. STATUS (w) = ACTIVE.

- If w is ACTIVE, then it’s on the stack and v is currently on top of stack.
- If vertex b directly on top of c on stack, then there must be an edge $c \rightarrow b$.
- Thus there must be a direct path $w \rightarrow \dots \rightarrow v$, appending $v \rightarrow w$ gives a cycle.

Prove cycle leads to “False”.

Let w be first vertex in cycle that we visit, v is vertex before w in cycle. Then, there exist a path $w \rightarrow \dots \rightarrow v \rightarrow w$.

Since w visited first. ISACYCLICDFS (w) called first in the cycle.

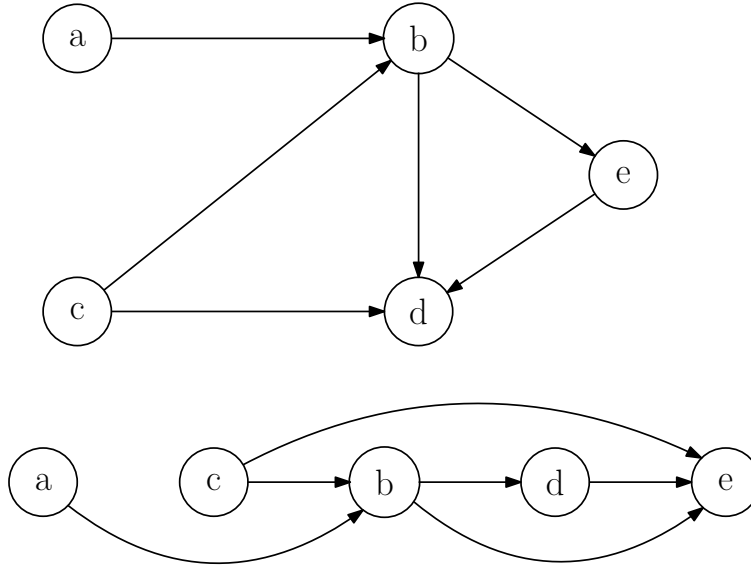
There is a directed path from w to v , so ISACYCLICDFS (v) is called while w on stack. During ISACYCLICDFS (v), $v \rightarrow w$ is looked at. Thus, result in False.

9.6 Topological Sort

Given directed G , topological ordering is a total order \prec on the vertices s.t. $u \prec v$ for every edge $u \rightarrow v$.

Informally, a topological sort is the order arranges vertices from left to right, s.t. all edges go from left to right. An example is shown in fig. 7

Figure 7: Example for Topological Sort



Observation:

The most important fact:

G has a topological ordering if and only if no cycles exist, i.e. G is a DAG.

A DAG has at least one source/sink, thus the topological sort has two implementation, one starts from sink, the other starts from source. The algorithm is shown in algorithm 31 and algorithm 30.

Algorithm 30 Topo-Sort from Source

```
1: procedure TOPOLOGICALSORT( $G$ )
2:    $n = |V|$ 
3:   for  $i = 1$  to  $n$  do
4:      $v =$  any Source
5:      $S[i] = v$ 
6:     Delete  $v$ 
7:   end for
8:   return  $S[1 \dots n]$ 
9: end procedure
```

Algorithm 31 Topo-Sort from Sink

```
1: procedure TOPOLOGICALSORT( $G$ )
2:    $n = |V|$ 
3:   for  $i = n$  to  $1$  do
4:      $v =$  any Sink
5:      $S[i] = v$ 
6:     Delete  $v$ 
7:   end for
8:   return  $S[1 \dots n]$ 
9: end procedure
```

In the algorithm, $v =$ any Sink/Source is ambiguous. Naively, the algorithm takes $\mathcal{O}(nm)$ time.

Lemma 9.6.1. *For any DAG, first vertex marked DONE by ISACYCLIC () is a sink.*

Proof: Let v be first vertex marked DONE. Suppose $v \rightarrow w$ exists.

- If w marked DONE, results in contradiction.
- If w marked ACTIVE, then there is a cycle.
- If w marked NEW, then recursively call on w . Thus w marked DONE before v .

In all three cases, v must be a sink. \square

Observation:

- ISACYCLIC (G) is modified DFS.
- First vertex marked DONE is first off stack.
- First off stack is the first post order.
- This behavior repeats.

Thus, Topological order is reverse post order traverse.

Topological ordering computed in $\mathcal{O}(n + m)$ time with DFS.

Why does linear time matter?

- Scheduling: two jobs $a \rightarrow b$ means a depends on b .
Reverse topological order is valid job scheduling.
- Dynamic Programming.
Recursive sub-problem is a vertex, edges to other sub-problems you depend on.

9.7 Strong Connectivity

u and v are in same strong component if $u \in \text{REACH}(v)$ and $v \in \text{REACH}(u)$.

Strong Component defines an equivalent relation.

- If u, v in some SC, and v, w in some SC, then u, w in same SC.
- Define a vertex partition: given two strong components S_1, S_2 , $S_1 \cap S_2 = \emptyset$, and every vertex in same SC.

Denote SSC (G) as Strong Connected Component Graph, obtained by collapsing each component to vertex, each edge from S_1 to S_2 , if and only if $\exists u \in S_1, v \in S_2$ s.t. $u \rightarrow v$ in G .

SSC (G) is DAG. In a DAG, every vertex is its own SC.

To compute SSC (G), call a component sink if sink in SSC (G). Observe that for sink components, DFS (v) for $v \in S$ explores all vertices in S . Thus, the algorithm can be defined as algorithm 32.

Algorithm 32 Algorithm to Compute Strong Components

```

1: procedure STRONGCOMPONENTS( $G$ )
2:    $count = 0$ 
3:   while  $G$  is not empty do
4:      $count = count + 1$ 
5:      $v =$  any vertex in sink component.
6:      $c = \text{DFS LABEL}(v, count)$ 
7:     Remove  $c$  from  $G$ .
8:   end while
9: end procedure

```

9.8 Minimum Spanning Tree (MST)

9.8.1 MST Definition

Input: Undirected, weighted, connected graph G . Assume there is a function $w : E \rightarrow R$

Output: Find the minimum spanning tree of G , i.e. the spanning tree T which minimizes $w(T) = \sum_{e \in T} w(e)$.

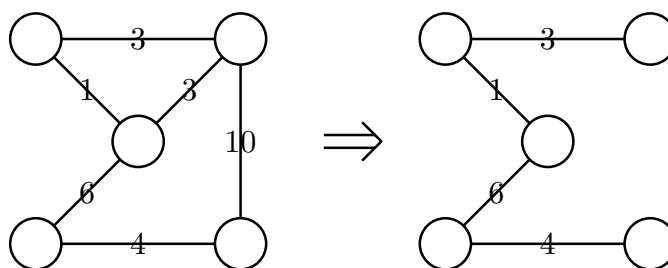
9.8.2 MST Property

Why use MST?

If vertices represent nodes in a network, and weights represent connection cost. MST is cheapest way to connect all nodes.

A example of MST is shown in fig. 8.

Figure 8: Example for MST



Observation:

Some properties are obvious:

- If all weights distinct, can prove MST is unique.
- If all weights equal, any spanning tree is a MST.

Main MST Property

A partition of V is a pair $S, R \subseteq V$, s.t. $S \cap R = \emptyset$ and $S \cup R = V$. Given a partition $S, R \subseteq V$, $\text{MWE}(S, R)$ denotes minimum weight edge $u \rightarrow v$, s.t. $u \in S, v \in R$.

- For any partition S, R , $\text{MWE}(S, R) \in \text{MST}$.
- If e is not minimum weight edge of any partition, then $e \notin \text{MST}(G)$.

Proof: First, prove that for any partition S, R , $\text{MWE}(S, R) \in \text{MST}$.

Let S, R be a partition and let $e = \text{MWE}(S, R)$.

Suppose $e = u \rightarrow v$, where $u \in S, v \in R$. Consider any spanning tree T , which does not contain e . T contains unique path from u to v . Some edges in this path has endpoint in S and another in R . Let e' be such an edge.

Remove e' from T , gives a spanning forest, with two trees. u and v must be in different trees.

Hence adding e connects two trees, producing single spanning tree $T' = T - e' + e$. $weight(e) < weight(e')$, so $weight(T') < weight(T)$. T is not MST, result in contradiction.

Then, prove that if e is not minimum weight edge of some partition, then it's not in MST.

Let T be the MST. Consider any edge $e' \in T$. e' define partition of V .

Just proved minimum weight edge across this partition must be in MST. Since e' is the only edge in T crossing this partition, it must be min weight. \square

9.8.3 MST Algorithm (General)

General idea of MST algorithm is:

Maintain a spanning forest, initially is n isolated vertices. Add an edge of MST, update forest, and repeat.

All the algorithm follows is the variation of this general idea.

9.8.4 Prim's Algorithm

Key point

- Pick vertex s . Grow MST from s , call it T .
- In each round, find the edge $e = \text{MWE}(T, V \setminus T)$. Add e to T and repeat.
- To find e , store edges adjacent to T in min priority queue.

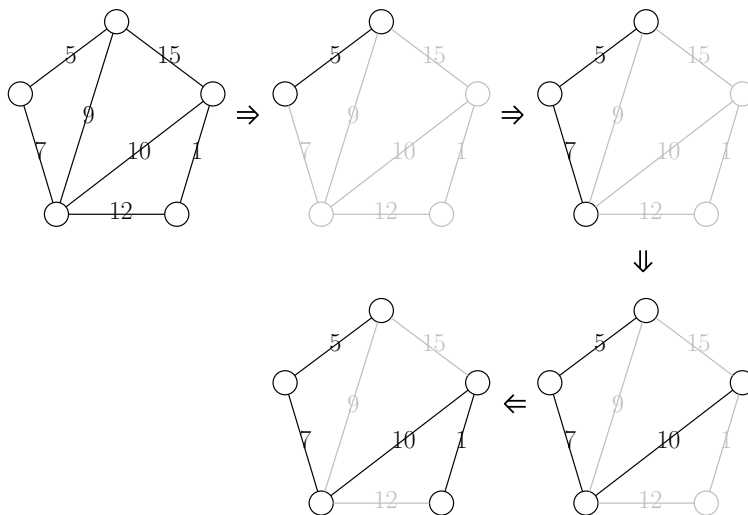
The algorithm is shown in algorithm 33

Algorithm 33 Prim's Algorithm

```
1: procedure PRIM( $s$ )
2:   Initial an empty minimum priority queue  $Q$ 
3:   Mark  $s$ 
4:   for all edges  $s \rightarrow w$  do
5:     Put  $(s, w)$  in  $Q$ 
6:   end for
7:   while  $Q$  is not empty do
8:      $(p, v) = Q.pop$ 
9:     if  $v$  is unmarked then
10:      Mark  $v$ 
11:      PARENT( $v$ ) =  $p$ 
12:      for each edge  $v \rightarrow w$  do
13:        Put  $(v, w)$  in  $Q$ 
14:      end for
15:    end if
16:  end while
17: end procedure
```

Observation: This traverse with bag = min queue. Marked mean in tree, not in queue.
An example is shown in fig. 9.

Figure 9: Example for MST



Analysis:

Each time add $\text{MWE}(T, V \setminus T)$, which we know is in MST, so correct.

Recall that for traverse, running time depend on bag operation cost. Specifically each edge is added/renewed from Q once, and this dominates runtime.

If Q implemented in binary min heap, all operations take

$$\mathcal{O}(\lg \text{heapsize}) = \mathcal{O}(\log(m)) = \mathcal{O}(\log(n))$$

time. Thus total time is $\mathcal{O}(m \log n)$

Note that the running time can be improved to $\mathcal{O}(m + n \log n)$ time using Fibonacci Heap, see Jeff's note.

9.8.5 Borvka's Algorithm

Key point

- Start with forest of singletons, $F = (v, \emptyset)$.
- Repeat until F is a single tree: For every $T \in F$, add $\text{MWE}(T, V \setminus T)$.

The algorithm is shown in algorithm 34

Algorithm 34 Borvka's Algorithm

```
1: procedure BORVKA( $V, E$ )
2:    $F = (V, \emptyset)$ 
3:    $count = \text{COUNTANDLABEL}(F)$ 
4:   while  $count > 1$  do
5:      $\text{ADDALLMWE}(E, F, count)$ 
6:      $count = \text{COUNTANDLABEL}(F)$ 
7:   end while
8:   return  $F$ 
9: end procedure
10:
11: procedure  $\text{ADDALLMWE}(E, F, count)$ 
12:   for  $i = 1$  to  $count$  do
13:      $S[i] = NULL$   $\triangleright w(null) = \infty \triangleleft$ 
14:   end for
15:   for each edge  $uv \in E$  do
16:     if  $label(u) \neq label(v)$  then
17:       if  $weight(uv) < w(S[label(u)])$  then
18:          $S[label(u)] = uv$ 
19:       end if
20:       if  $weight(uv) < w(S[label(v)])$  then
21:          $S[label(v)] = uv$ 
22:       end if
23:     end if
24:   end for
25:   for  $i = 1$  to  $count$  do
26:     if  $S[i] \neq NULL$  then
27:       add  $S[i]$  to  $F$ 
28:     end if
29:   end for
30: end procedure
```

$S[i]$ stores minimum weight edge with exactly one endpoint in tree i .

Analysis:

The algorithm is correct since add MWE $(T, V \setminus T)$ for each $T \in F$.

The running time:

- COUNTANDLABEL (F) takes $\mathcal{O}(n)$ time, since F has $\mathcal{O}(n)$ edge.
- ADDALLMWE () takes $\mathcal{O}(m)$ time, since $\mathcal{O}(m + n) = \mathcal{O}(m)$.
- While-loop takes $\mathcal{O}(m + n) = \mathcal{O}(m)$ per iteration, in total performs $\mathcal{O}(\lg n)$ iterations.

Thus, total running is $\mathcal{O}(m \log n)$.

9.8.6 Kruskal's Algorithm

Key point

- Sort edges in increasing weight order.
- Go through edges in increasing order, add edge e if its endpoints in different trees.

Correctness 1 Suppose we add e joining $T_1, T_2 \in F$, then e must be MWE ($T_1, V \setminus T_1$). If not, then there was cheaper edge e' , but in which case, e' should be add first.

The difficulty is maintaining F . For Borvka, spend $\mathcal{O}(m)$ time to update F . Hence, now consider edges one at a time leads to $\mathcal{O}(m \times m) = \mathcal{O}(m^2)$ time.

Union Find

Union Find is a data structure which can do the following:

- MAKESET (V): make set (i.e. tree in forest) with just V .
- FIND (u): return set id (some vertex in tree).
- UNION (u, v): merges sets containing u and v .

All operation of Union Find take $\mathcal{O}(\alpha(n))$ time.

The algorithm is shown in algorithm 35

Algorithm 35 Kruskal's Algorithm

```
1: procedure KRUSKAL( $s$ )
2:   sort  $E$  by increasing weight order
3:    $F = (V, \emptyset)$ 
4:   for each vertex  $v \in V$  do MAKESET ( $v$ )
5:   end for
6:   for  $i = 1$  to  $|E|$  do
7:      $uv = i$ th heaviest edge in  $E$ 
8:     if FIND( $u$ )  $\neq$  FIND( $v$ ) then
9:       UNION ( $u, v$ )
10:      add  $uv$  to  $F$ 
11:    end if
12:  end for
13:  return  $F$ 
14: end procedure
```

Analysis:

The running time:

- $\mathcal{O}(m)$ find operations.
- $\mathcal{O}(n)$ union operations.
- $\mathcal{O}(n)$ makeset operations.

So other than sorting, total time is $\mathcal{O}(m \propto (n))$.

Sorting takes $\mathcal{O}(m \log m) = \mathcal{O}(m \log n)$ time, dominating the total running.

Thus, total running time would be $\mathcal{O}(m \log n)$.

9.9 Shortest Paths

Input: Weighted directed graph G with labeled source s and target t .

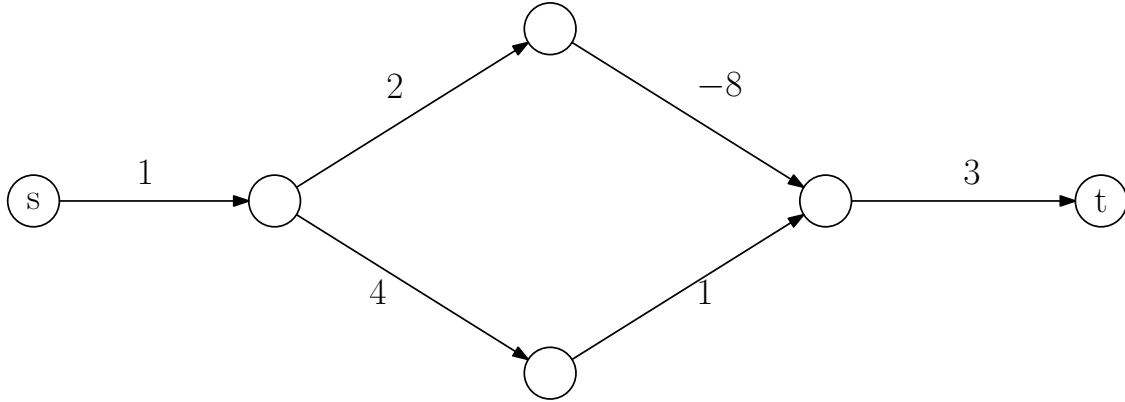
Output: Find shortest $s - t$ path, i.e. directed path P from s to t that minimized

$$w(P) = \sum_{u \rightarrow v \in P} w(u \rightarrow v)$$

Note that we can allow negative weights, but cannot be negative cycles.

A example of no SP is shown in fig. 10.

Figure 10: Example for No Shortest Path



9.9.1 Single Source Shortest Path (SSSP)

Almost every algorithm to find shortest $s - t$ path find shortest path to every vertex. We can represent all shortest paths from s using a spanning tree rooted at s .

Why a tree satisfied?

Suppose shortest $s - w$ path passes through v and shortest $s - t$ path passes through v . The subpath of a shortest path is a shortest path.

9.9.2 SSSP Algorithm

Most SSSP algorithms have same structure.

For every vertex v , store two values.

- $dist(v)$ = length of tentative shortest sv path, or ∞ if no $s - v$ path exists.
- $pred(v)$ = predecessor of v in tentative shortest $s - v$ path tree.

Predecessor pointers define tentative shortest path tree.

Initially, we know $dist(s) = 0$, $pred(s) = null$, and for every $v \neq s$, initially $dist(v) = \infty$, $pred(v) = null$ to indicate we don't know if there is $s - v$ path.

As we run algorithm $dist(v)$ values may decrease, but always upper bound tree distances.

During algorithm execution, an edge $u \rightarrow v$ is “tense” if $dist(v) + w(u \rightarrow v) < dist(v)$. If $u \rightarrow v$ tense, clearly shortest path from s to v is wrong, since $s - u$ path followed by $u \rightarrow v$ is shortest.

9.9.3 Generic SSSP algorithm

The general idea is:

Repeated find tense edge and relax it.

The relax operation is defined as algorithm 36.

Algorithm 36 Relax Operation in SSSP Algorithm

```
1: procedure RELAX( $u \rightarrow v$ )
2:    $dist(v) = dist(u) + w(u \rightarrow v)$ 
3:    $pred(v) = u$ 
4: end procedure
```

The algorithm stops when no tense edges. This can prove the following:

- 1) $dist(v) \geq$ true distance from s to v at all times.
- 2) Algorithm halts, i.e. at some finite time, no more tense.
- 3) When algorithm halts, all $dist(v)$ values correct.

Proof: For 1), prove $\forall v \in V$, $dist(v)$ is either ∞ or the length of $s - v$ walk.

Use induction on number of relaxations.

- Consider $v \in V$, if v not updated in previous relax, then holds by induction.
- If v updated, then $dist(v) = dist(u) + w(u \rightarrow v)$, and $dist(u)$ is the length of walk by induction.

For 2).

In 1) can actually prove $dist(v)$ is ∞ or length of $s - v$ path (not just walk).

There finite number of paths, and distances only decrease, thus, the algorithm must stop at some finite time.

For 3).

Suppose otherwise, and let v be vertex in true shortest path tree, s.t. $dist(v)$ wrong, and v closest to s in tree (in terms of edges).

Let s, \dots, u, v be shortest path, true distance to v is $dist(u) + w(u \rightarrow v)$. By 1), $dist(v) > truedistance$, so $dist(v) > truedistance = dist(u) + w(u \rightarrow v)$, which means tense edge still exist. So the algorithm shouldn't halts, hence result in contradiction. \square

Note that during the process, we look at a local statement, in the end result in a global statement.

So, how do we determine tense edges and what order to relax?

One way to do this is to mimic traverse. The algorithm is shown in algorithm 37.

Algorithm 37 Generic SSSP Algorithm

```
1: procedure INITSSSP( $s$ )
2:    $dist(s) = 0$ 
3:    $pred(s) = null$ 
4:   for all  $s \neq v$  do
5:      $dist(v) = \infty$ 
6:      $pred(v) = null$ 
7:   end for
8: end procedure
9: procedure GENERICSSSP( $s$ )
10:  INITSSSP ( $s$ )
11:  Put  $s$  in bag
12:  while bag not empty do
13:    Take  $u$  from bag
14:    for all  $u \rightarrow v$  do
15:      if  $u \rightarrow v$  is tense then
16:        RELAX ( $u \rightarrow v$ )
17:        Put  $v$  in bag
18:      end if
19:    end for
20:  end while
21: end procedure
```

Initially, only tense edges are those leaving s . Each time we relax an edge $u \rightarrow v$, a new edge can become tense, but those edges must have v as the origin. Hence all tense edges would be seen before bag empty.

The question is: How do we implement the bag?

If bag is stack, could run in exponential time. And here come the Dijkstra's Algorithm.

9.9.4 Dijkstra's Algorithm

Use a minimum priority queue on $dist(u)$ values to implement the bag. If no negative edge weights, we can prove vertices removed from bag in increasing order of distance from s .

It follows each vertex removed (or inserted) from bag at most once. Each time an edge relaxed, algorithm performs decrease key on priority queue. Since each vertex inserted at most once, at most one decrease key per edge. So, total number insertions/deletions is $\mathcal{O}(n)$.

Fibonacci heaps has $\mathcal{O}(1)$ decrease key time, $\mathcal{O}(\log n)$ insert/delete time. Thus in total

$$\mathcal{O}(m + n \log n)$$

For regular binary heap, every operation takes $\mathcal{O}(\log n)$ time. Thus in total

$$\mathcal{O}((n + m) \log n) = \mathcal{O}(m \log n)$$

9.9.5 Bellman-Ford Algorithm

Use a standard FIFO queue to implement the bag.

The correctness follows since **GENERICSSSP** (S) is correct.

Algorithm 38 Bellman-Ford Algorithm

```

1: procedure BELLMAN-FORD( $s$ )
2:   INITSSSP ( $s$ )
3:   loop Repeat  $|V|$  times:
4:     for every edge  $u \rightarrow v$  do
5:       if  $u \rightarrow v$  is tense then
6:         RELAX ( $u \rightarrow v$ )
7:       end if
8:     end for
9:   end loop
10:  for every edge  $u \rightarrow v$  do
11:    if  $u \rightarrow v$  is tense then
12:      return “Negative Cycle”
13:    end if
14:  end for
15: end procedure

```

In other word, the algorithm relax all tense edges and repeat.

The running time if $\mathcal{O}(mn)$.

Is the algorithm correct?

We can prove that, after i repeat phases, $dist(v)$ is length of shortest walk from s to v using at most i edges. This statement holds true regardless of negative edges.

If negative cycle exists, there is always tense edges, since it can always get shorter walk. Hence Bellman-Ford Algorithm can detect negative cycles.

9.9.6 Some Intuition on SSSP

Dijkstra's algorithm: sends out wave based on true distance.

Bellman-Ford algorithm: sends out wave based on number of edges.

9.10 Max Flows and Min Cuts

Example first:

Given a network of nodes, where each connection has some bandwidth.

1. How much data per unit time can be sent from s to t ?
2. How much bandwidth must be removed, so that s cannot reach t ?

The answer for both questions is the same! Max flow = Min cut.

9.10.1 Flows

Given a directed graph G , with source s , target t . An (s, t) -flow is a function: $f : E \rightarrow \mathbb{R}_{\geq 0}$ which satisfies following conservation at all $v \neq s, t$

$$\sum_u f(u \rightarrow v) = \sum_w f(v \rightarrow w)$$

which means flow into v and flow out of v .

The “value” of the flows, denoted by $|f|$ is the net flow leaving s :

$$|f| = \sum_w f(s \rightarrow w) - \sum_u f(u \rightarrow s)$$

To simplify notation, define

$$\Delta(f(v)) = \sum_u f(u \rightarrow v) - \sum_w f(v \rightarrow w)$$

as the total net flow out of any vertex v .

By conservation:

$$\sum_v \Delta(f(v)) = \Delta(f(s)) + \Delta(f(t)) = 0$$

since any flow leaving vertex must enter some other vertex.

Hence,

$$|f| = \Delta(f(s)) = -\Delta(f(t))$$

$|f|$ is maximized.

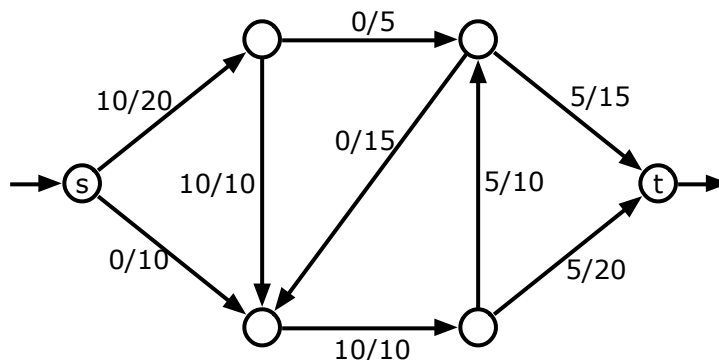


Figure 11: An (s, t) - *flow* with value 10. Each edge is labeled with its flow/capacity.

9.10.2 Cuts

An $(s, t) - cut$ is a vertex bi-partition S, T , i.e. $S \cap T = \emptyset$ and $S \cup T = V$, s.t. $s \in S$ and $t \in T$.

The capacity of a cut is the sum of capacities of edges starting in S and ending in T

$$\|S, T\| = \sum_{v \in S} \sum_{w \in T} c(v \rightarrow w)$$

The min cut problem is to *find cut whos capacity is as small as possible*.

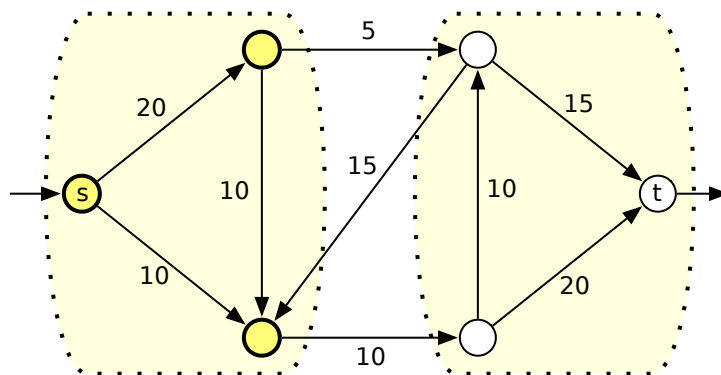


Figure 12: An (s, t) – *cut* with capacity 15. Each edge is labeled with its capacity.

Intuitively, for any feasible (s, t) -flow and any (s, t) -cut, $|f| \leq \|S, T\|$. The proof is

as follow:

$$\begin{aligned}
|f| &= \sum_w f(s \rightarrow w) - \sum_u f(u \rightarrow s) && \text{by definition} \\
&= \sum_{v \in S} \left(\sum_w f(v \rightarrow w) - \sum_u f(u \rightarrow v) \right) && \text{by the conservation constraint} \\
&= \sum_{v \in S} \left(\sum_{w \in T} f(v \rightarrow w) - \sum_{u \in T} f(u \rightarrow v) \right) && \text{removing duplicate edges} \\
&\leq \sum_{v \in S} \sum_{w \in T} f(v \rightarrow w) && \text{since } f(u \rightarrow v) \geq 0 \\
&\leq \sum_{v \in S} \sum_{w \in T} c(v \rightarrow w) && \text{since } f(u \rightarrow v) \leq c(u \rightarrow v) \\
&= \|S, T\| && \text{by definition}
\end{aligned}$$

Observation:

$|f| = \|S, T\|$ if and only if:

- f avoids every edge from T to S ;
- f saturates every edge from S to T .

If $|f| = \|S, T\|$, then f is a max flow and S, T is a min cut. A max flow saturates all min cuts simultaneously.

9.10.3 The Maxflow MinCut Theorem

Theorem 9.10.1 (The Maxflow Mincut Theorem). *In any flow network with source s and target t , the value of the maximum (s, t) – flow is equal to the capacity of the minimum (s, t) – cut.*

Proof:

To make proof easier, assume capacity function is reduced, i.e. for $v, w \in V$, either $c(u \rightarrow v) = 0$ or $c(v \rightarrow u) = 0$ (only one of the edges exist). This assumption is easy to enforce, see fig. 13.

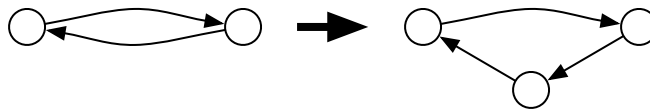


Figure 13: Enforcing the one-direction assumption

Let f be any feasible flow. Define a new capacity function $c_f : V \times V \rightarrow \mathbb{R}$, called the residual capacity, as follows:

$$c_f(u \rightarrow v) = \begin{cases} c(u \rightarrow v) - f(u \rightarrow v) & \text{if } u \rightarrow v \in E \\ f(v \rightarrow u) & \text{if } v \rightarrow u \in E \\ 0 & \text{otherwise} \end{cases} \quad (15)$$

because edge is reduced, an pair can only in one situation.

Since $f \geq 0$ and $f \leq c$, the residual capacities are always non-negative. Note that, we can have $c_f(u \rightarrow v) > 0$, for $u \rightarrow v \notin E$. Thus, define residual graph as $G_f = (V, E_f)$, where E_f is the set of edges with positive residual capacity. Notice that the residual capacities are not necessarily reduced, it can have both $c_f(u \rightarrow v) > 0$ and $c_f(v \rightarrow u) > 0$, as shown in fig. 14

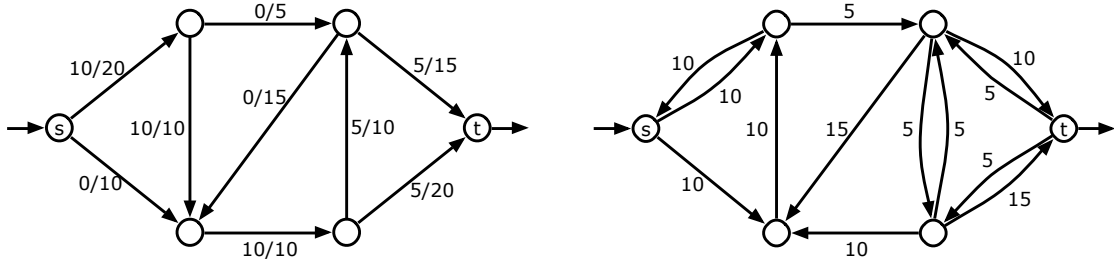


Figure 14: A flow f in a weighted graph G and the corresponding residual graph G_f .

Case 1 There is no path from s to t .

Let S be the vertices reachable from s in G_f , and let $T = V \setminus S$.

Then (S, T) is clearly a cut.

For any $u \in S$ and $v \in T$, we have

$$c_f(u \rightarrow v) = (c(u \rightarrow v) - f(u \rightarrow v)) + f(v \rightarrow u) = 0 ,$$

- If $u \rightarrow v \in E$, then $c(u \rightarrow v) = 0$, i.e. saturated.
- if $v \rightarrow u \in E$, then $f(v \rightarrow u) = 0$, i.e. avoided.

So f avoids every edge from T to S and saturates all edges from S to T .

Hence, $|f| = \|S, T\| \Rightarrow f$ is max flow, S, T is min cut.

Case 2 There is a path $s = v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_r = t$ in G_f .

Refer to such path as an augmenting path. Let $F = \min_i c_f(v_i \rightarrow v_{i+1})$ denote

the maximum amount can pass through this augmenting path. Define a new flow function $f' : E \rightarrow \mathbb{R}$ as follows:

$$f'(u \rightarrow v) = \begin{cases} f(u \rightarrow v) + F & \text{if } u \rightarrow v \text{ is in the augmenting path} \\ f(u \rightarrow v) - F & \text{if } v \rightarrow u \text{ is in the augmenting path} \\ f(u \rightarrow v) & \text{otherwise} \end{cases} \quad (16)$$

To prove that the flow f' is feasible with respect to original capacities c , we need to verify that $f' \geq 0$ and $f' \leq c$ for all edges.

Consider and edge $u \rightarrow v \in G$.

- If $u \rightarrow v$ is in augmenting path, then $f'(u \rightarrow v) > f(u \rightarrow v) \geq 0$ and

$$\begin{aligned} f'(u \rightarrow v) &= f(u \rightarrow v) + F && \text{by definition of } f' \\ &\leq f(u \rightarrow v) + c_f(u \rightarrow v) && \text{by definition of } F \\ &= f(u \rightarrow v) + c(u \rightarrow v) - f(u \rightarrow v) && \text{by definition of } c_f \\ &= c(u \rightarrow v) \end{aligned}$$

- If $v \rightarrow u$ is in the augmenting path, then $f'(u \rightarrow v) < f(u \rightarrow v) \leq c(u \rightarrow v)$, which implies that

$$\begin{aligned} f'(u \rightarrow v) &= f(u \rightarrow v) - F && \text{by definition of } f' \\ &\geq f(u \rightarrow v) - c_f(u \rightarrow v) && \text{by definition of } F \\ &= f(u \rightarrow v) - f(u \rightarrow v) && \text{by definition of } c_f \\ &= 0 \end{aligned}$$

- If $u \rightarrow v$ nor $v \rightarrow u$ is in augmenting path, nothing change.

Finally, we observe that only the first edge in augmenting path leaves s , so $|f'| = |f| + F > |f|$. Hence, f is not a max flow.

We proved: *For feasible f , let S be reachable set from s in G_f , f is max flow, if and only if $t \notin S$. If $t \notin S$, $|f| = \|S, V \setminus S\|$ and so $\|S, V \setminus S\|$ is min cut.* \square

9.10.4 Ford-Fulkerson Algorithm

Maxflow Mincut theorem proof gives an algorithm to compute max flow and hence min cut.

1. Initially set $G_f = G$;

2. Find any s, t path P in G_f ;
3. Augment f along P , update G_f and repeat until no s, t path.

The question is: how long does this algorithm takes?

Theorem 9.10.2 (Integrality Theorem). *If all capacities in a flow network are integers, then there is a maximum flow such that the flow through every edge is an integer.*

Analysis: Let f^* be max flow. Each time Ford-Fulkerson's augments, flow increases by at least 1. Thus, augmenting takes $\mathcal{O}(|E|)$ time. Total time: $\mathcal{O}(|E||f^*|)$ time.

However, Ford-Fulkerson might alternate between pushing 1 unit of flow along the augmenting path back and forth, as shown in fig. 15.

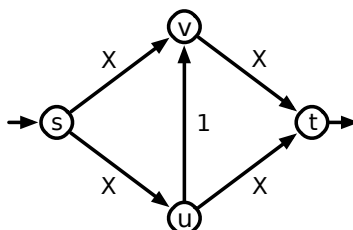


Figure 15: A bad example for the Ford-Fulkerson algorithm

To avoid this situation, we can choose (Edmonds-Karp algorithm)

- path with largest bottleneck;
- path with shortest number of edges.

For more information, see Chapter 23.6 in Jeff's notes.

In conclusion, Ford-Fulkerson's algorithm takes $\mathcal{O}(X)$ time where the maximum flow value $|f^*| = 2X$, but the input size is $\mathcal{O}(\log X)$, thus, Ford-Fulkerson is actually exponential in the input size.

9.10.5 Application of Max Flow

- Edge disjoint path (edp).

Given directed graph G , want max number of edge disjoint $s - t$ paths (note that it may share vertices). Let K be the max number.

Solution: Assign capacity 1 to all edges, compute max flow f^* , then $|f^*| = k$.

- Vertex capacities.

Also given vertex capacities, $c(v)$ for each $v \in V$, want max flow with additional requirement that total flow entering $v \leq c(v)$ (and therefore flow out of $v \leq c(v)$).

Solution: Use standard flow, but replace each v with v_{in} and v_{out} , with edge $v_{in} \rightarrow v_{out}$ of capacity $c(v)$. All incoming edges to v now to v_{in} , all outgoing edges now from v_{out} .

- Maximum Matching in Bipartite graphs.

Given an undirected graph $G = (V, E)$ is Bipartite. If V can be partitioned into two sets A, B , so that all edges are from A to B (A, B are independent set). A matching is a subset $M \subseteq E$ s.t every vertex in $G' = (V, M)$ has degree ≤ 1 (i.e. edges in M are non-adjacent). A maximum matching is matching s.t $|M|$ is maximized.

Can find max matching in bipartite G as follow:

1. orient all edges from u to W .
2. adding source s , with directed edges from all of A .
3. adding sink t , with directed edges from all of B .
4. assign capacity 1 to all edges.
5. compute max flow, saturated A to B edges are max matching.

10 NP Hardness