算法正确性论证 Correctness

```
Insertion-Sort(A) // sorts A[1..n]
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1..j-1]
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i+1] = A[i]
7          i = i - 1
8      A[i+1] = key
```

1.1 Loop Invariant : LI

LI: At the start of each iteration of for loop, keys originally in A[1...j-1] in sorted order.

(Basis) Initialization : LI True Initially

(Inductive) Maintenance : If LI True (IH) before an arbitrary iteration of loop it remains true before next one

Termination : LI True when loop ends.

对本题: Init : j=2 holds

Mainn: A[1,j-1] sorted : insert j-th item after k-th position if A[k] ≤ A[j] < A[k+1]
∴ sorted order is mained

Analysis Running Time

T(n) = worst-case running time on input of size n.

本题 $\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2} = \Theta(n^2)$

1.2 Asymptotic Notation

Input Size : Time gets larger as inputs do

Parameterize input : size n

O表述 : O-notation : There exist positive constant c and n_0 such that

$O(g(n)) = 0 \le f(n) \le c\,g(n)$ for all $n \ge n_0$ upper bound

Ω表述 : Ω-notation : There exist positive constant c and n_0 such that

$\Omega(g(n)) = 0 \le c(g(n)) \le f(n)$ for all $n \ge n_0$ Lower bound

Θ表述 : Θ-notation $T(n) = \Theta(g(n))$ IFF $T(n) = O(g(n)) = \Omega(g(n))$

i.e. $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ (e.g. $T(n) = 4n^2 + 2n + 12 = \Theta(n^2)$) Tight bound

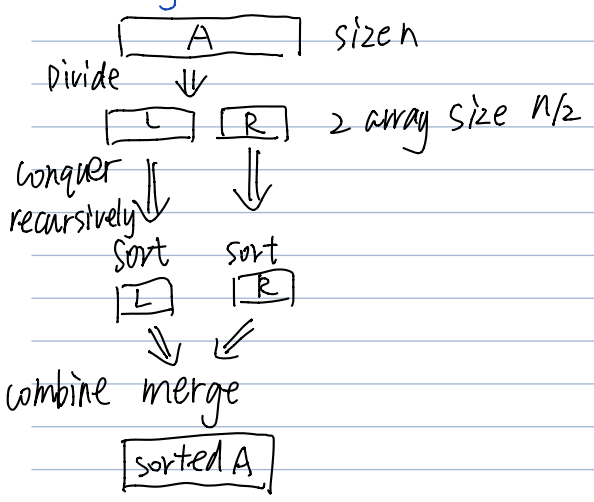
Properties : Const : $\Theta(1)$

Addition: $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$

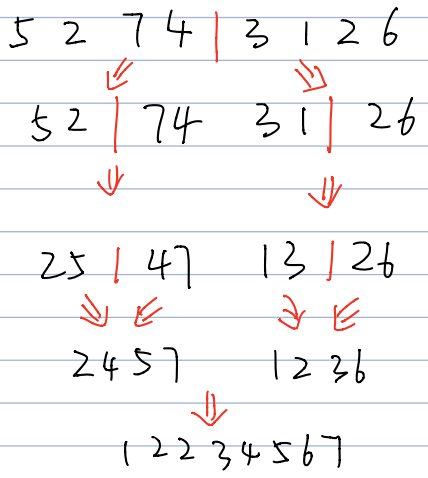
Multiplication: $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$

1.3 Divide & Conquer : Merge sort 归并排序

Merge Sort



e.g.



Merge sort 伪代码

```
Merge-sort (A, start, end)                    T(n)
01 if start < end                             Θ(1)
02    mid = ⌊(start + end)/2⌋                 Θ(1)
03    Merge-sort (A, start, mid)              T(n/2)
04    Merge-sort (A, mid+1, end)              T(n/2)
05    Merge (A, start, mid, end)              Θ(n)
```

Initial Call : Merge (A, 1, n)

∴ Merge sort = $\begin{cases} \Theta(1) & n=1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{o/w.} \end{cases}$

```
Merge (A, start, mid, end)                    Θ(n)
01  n1 = mid - start + 1
02  n2 = end - mid
03  new array L[1..n1+1]  R[1..n2+1]
04  for i = 1 to n1 :  // copy left
05        L[i] = A[start + i - 1]
06  for j = 1 to n2 :  // copy right
07        R[j] = A[mid + j]
08  L[n1+1] = ∞                  标记 sentinel
09  R[n2+1] = ∞                  // sentinel end
10  i = 1
11  j = 1
12  for k = start to end
13        if L[i] ≤ R[j]
14              A[k] = L[i]
15              i = i + 1
16        else
17              A[k] = R[j]
18              j = j + 1
```

T(n) 怎么解? (Master Theorem)

if $T(n) = aT(\lceil \frac{n}{b} \rceil) + O(n^d)$

For constant $a \geq 1$ $b \geq 1$ $d \geq 0$

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \cdot \lg n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Merge-sort = $O(n \lg n)$

1.4 Divide and conquer : Fast Multiplication of N-Digit Integer 大整数乘法

Divide :

$$X = \overbrace{|\quad X_1 \quad|\quad X_0 \quad|}^{n} = X_1 \cdot 10^{\frac{n}{2}} + X_0$$

$$Y = |\quad Y_1 \quad|\quad Y_0 \quad| = Y_1 \cdot 10^{\frac{n}{2}} + Y_0$$

$$\Big\} \; X = X_1Y_1 10^n + (X_1Y_0 + X_0Y_1)10^{\frac{n}{2}} + X_0Y_0$$

$T(n) \leq 4 \cdot T(\frac{n}{2}) + O(n) = O(n^2)$

优化： $X \cdot Y = (X_1 + X_0)(Y_1 + Y_0) \overset{n}{=} X_1Y_1 + X_0Y_1 + X_1Y_0 + X_0Y_0$

$= (X_1 + X_0)(Y_1 + Y_0) - X_1Y_1 - X_0Y_0$

```
FastMultiply(x, y)// x, y n-digit integers, assume n = 2^k
if n == 1
    use multiplication table to find z = x · y
else divide x, y in half:
    x = 10^{n/2} x_1 + x_0
    y = 10^{n/2} y_1 + y_0
    // x_1, y_1, x_0, y_0 each have n/2 digits
    P_1 = FastMultiply(x_1, y_1)   T(n/2)
    P_0 = FastMultiply(x_0, y_0)   T(n/2)
    P_2 = FastMultiply(x_1 - x_0, y_1 - y_0)   T(n/2)
    T = P_1 + P_0 - P_2
    z = 10^n P_1 + 10^{n/2} T + P_0
return z
```

$\Rightarrow 3 \cdot T(\frac{n}{2}) + O(n) = O(n^{1.585})$

Week 2.

2-1 Quick Sort

Insertion Sort { worst $\theta(n^2)$; best $\theta(n)$

Merge Sort { $O(n \lg n)$; But need $\theta(n)$ extra place

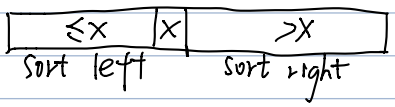
⇒ quick sort in-place sort

Moving data while dividing

Quick Sort : { Divide and conquer ; Sorts in place

1. Divide and conquer

1. Divide : Partition input array into 2 subarrays with pivot x such that elements in left $\leq x$, elements in right $> x$



2. conquer : recursively sort left, sort right

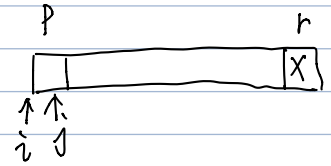
3. combine : None : B/C sort in place

Partition pseudocode

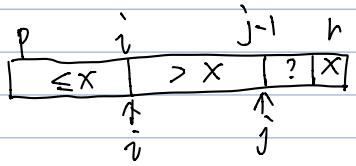
```
Partition (A,p,r)          // A[p...r]      p=1  r=n
01  x = A[r]               // pivot = A[r]
02  i = p-1                // i = last element of ≤x subarray so far
03  for j=p to r-1
04      if A[j] ≤ x
05          i = i+1
06          exchange A[i] and A[j]
07  exchange A[i+1] and A[r]
08  return i+1
```

Invariant

Initially :

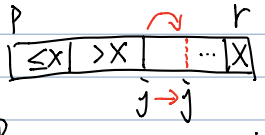


Later :

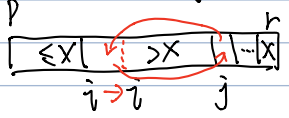


is $A[j] \leq x$?

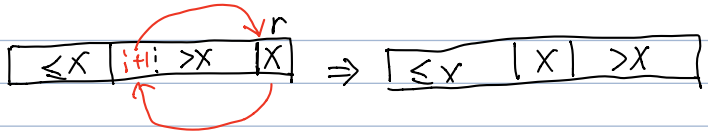
if $A[j] > x$:



if $A[j] \leq x$:



End :



Initialization: Before loop begins $A[r] =$ pivot and sub-arrays $A[p..i]$, $A[i+1..j-1]$ are empty. Conditions 1–3 hold.

Maintenance: During loop execution, if $A[j] \leq$ pivot, $A[j]$ and $A[i+1]$ are exchanged and i and j are incremented. If $A[j] >$ pivot, then only j is incremented. Conditions 1–3 hold in both cases.

Termination: When loop terminates, $j = r$, so all entries in A are partitioned into one of the 3 cases: $A[p..i] \leq$ pivot, $A[i+1..j-1] >$ pivot, $A[r] =$ pivot.

$\therefore$ Conditions 1–3 always hold.

Quick-Sort Pseudocode

```
Quicksort (p,q,r)      // A[p...r]      p=1   r=n
01. if p<r
02      q = Partition (A, p, r)
03      Quicksort (A, p, q-1)
04      Quicksort (A, q+1, r)
```

Initial Call
Quicksort (A, 1, n)

Correctness: 1st: pivot element is put in correct (sorted) position } in final sorted order
2nd: after patition no element move to the other side

Time analysis

```
Quicksort(A, p, r)                    // T(n)
1  if p < r                           // O(1)
2      q = Partition(A, p, r)         // Θ(n)
3      Quicksort(A, p, q − 1)         // T(q − 1)
4      Quicksort(A, q + 1, r)         // T(n − q)
```

Recurrence:

$$T(n) = \underbrace{T(q-1) + T(n-q)}_{\text{cost of subproblems}} + \underbrace{\Theta(n)}_{\text{Partition cost}}$$

Best case: pivot always split array into 2 sub arrays of size $n/2$

$T(n) \leq 2T(\frac{n}{2}) + \theta(n) = O(n \lg n)$

(same as merge sort)

Worst case: require $\Omega(n^2)$ comparisons $q=n$ or 1

$A[1...n] = [1, 2, \cdots, n]$ 或反向

1st pivot : n patition : $n-1$

$[1 \cdots n-1, \underbrace{[n]}_{R}]_{L}$

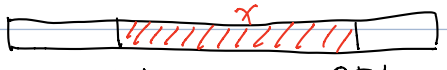
2st pivot : $n-1$ patition : $n-2$

Total $= 1 + \cdots + n-1 = \frac{n(n-1)}{2}$

$T(n) = O(n^2)$

Average Running time : close to best

⇒ call a pivot x good if it lies between



25th to 75th

一半概率

$T(n) \leq T(\frac{n}{4}) + T(\frac{3}{4}n) + \theta(n)$

$T(n) = O(n \lg n)$ (不知用这个前面公式)

CLRS 4.4-9 P.93

定理 for $0<a,b<1$ 且 $a+b=1$ $T(n) = T(an) + T(bn) + O(n)$ solves to $O(n \lg n)$

2-2 Randomized - Quicksort

Partition around random element

Randomized-partition (A, p, r)

```
01 i = random(p, r)   // pick i at random
02 exchange(A[r], A[i])
03 return partition(A, p, r)
```

CLRS p.117

Analysis: compute a bound on expected # of comparisons

Lemma: for randomized quicksort E[# comparison] $\in [1, 2n\ln n)$

Proof: input $A[1] \cdots A[n]$ let $S_1, \cdots S_n$ be sorted(A)

a pair S_i and S_j are compared at most once during algo.

Why? 1st time compare either S_i or S_j pivot. never compare again

for $i < j$ let $R_{ij} = \begin{cases} 1 & S_i S_j \text{ compared} \\ 0 & \text{o/w} \end{cases}$

Let R = total # of comparisons

$$R = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} R_{ij}$$

$$E(R) = E\left[\sum\sum R_{ij}\right] = \sum\sum \underbrace{E(R_{ij})}_{= 1 \cdot Pr(R_{ij}=1) + 0 \cdot Pr(R_{ij}=0)}$$

$$= Pr(R_{ij} = 1)$$

Note: 1st time a pivot is chosen from $S_i \cdots S_j$ it is equally likely to be any of them

$$Pr(R_{ij} = 1) = \frac{2}{j-i+1}$$

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E(R_{ij}) = 2 \cdot \sum_{i=1}^{n-1} \left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-i+1}\right) \leq 2 \sum_{j=1}^{n-1} \underbrace{\left(\frac{1}{2} + \cdots + \frac{1}{n}\right)}_{= \int_2^n \ln x \, dx}$$

$$\leq 2 \cdot n \cdot \ln n$$

2-3 Hashing

- Hashing Basic
- Chained Hash Tables – Universal Hashing
- Open Addressing – Linear Probing
  – Double Hashing

Hashing :

Dictionary Problem : maintain a dynamic set of items, each with a key, in a table.

3 operations:

- insert (item) add item
- delete (item) remove item
- search (key) return item

} O(1) per operation in expection

(Assum items have distinct keys)

① Hash table:

Give short name, # between 1~250 to each of 2^{32} IP Addresses.

Store records of 250 IP's in table of size 250. indexed by short names.

What if more than one record associated with same name? → collision

Two issue: ① How assign "short names"
② How resolve "collision"

① General Set Up:

- Universe : $U = \{1, \dots 2^w\}$ → Table $T[0 \cdots m-1]$
  2^{32} IP's — 250 Entries ($m=250$)
- Subset $S \subseteq U$ $|S| = n$
- Hash Func $h: U \to \{0, 1, \cdots, m-1\}$ store $x \in U$ at $T[h(x)]$

When $\begin{cases} x \neq y \\ h(x) = h(y) \end{cases}$ ⇒ collision (goal) ⇒ $\begin{cases} \text{rare} \\ \text{cheap} \end{cases}$

② Chain hash table

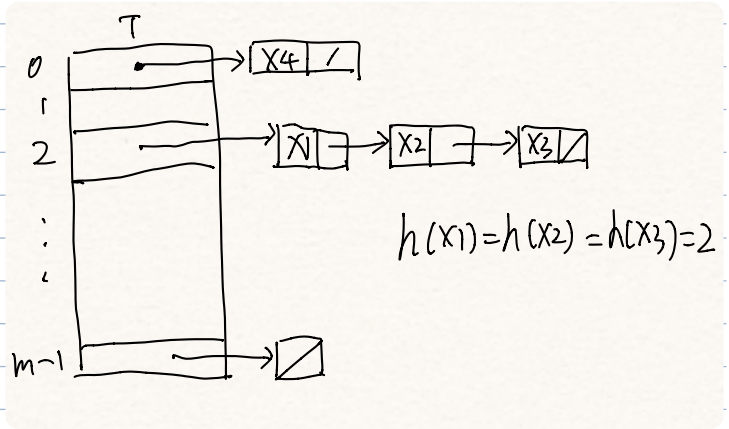
Chaining { table $T[0 \cdots m-1]$
$T[i]$ is linked list of keys with $h(x) = i$

Insert

Hash-insert (T, x) (x: key)

insert at head of list T[h(x)] O(1)

if allow duplicates O(length of list)



search

Hash-search (T, x)

search for element with key x in T[h(x)]

Time: O(length of list)

Delete

Hash-delete (T, x) also O(length)

Running time:

proportional to the length of list of element in slot h(k)

Time for search $O(1 + \ell(x))$ (compute h(x))

$\ell(x)$: # of keys such that $h(x) = h(y)$

Time analysis (hash n items into table of size m)

Worst case: all hash to same position: search $\Theta(n)$

Expected Time:

- Assumption: each key is equally likely to hash to any slot of table independent of where other key hash to
- Define: load factor: $\alpha = \frac{n}{m}$ = avg (# keys per slot) ($\frac{n}{m}$: Avg length)
- expected performance given assumption

$O(1 + E[\ell(x)])$ $\quad E[\ell(x)] = \frac{n}{m} = \alpha$

Time $= O(1+\alpha) = O(1)$ iff $\alpha = O(1)$ i.e. $m = \Omega(n)$

($1+\alpha$: compute $i = h(x)$, search list)

clearly like $m = \Omega(n)$ and $m = O(n)$

$\Rightarrow m = \Theta(n)$

e.g. $n/4 \le m \le 4n$ very nice!

How compute h(x)?

keys as natural values. e.g. string → ASCII

"CLRS" $= 67 \times 128^3 + 76 \times 128^2 + 82 \times 128^1 + 83 \times 128^0$

- Division method : $h(x) = x \bmod m$

advantage : easy O(1)
disadvantage : must avoid certain values of m power of 2

BAD : if $m = 2^p$ for integer p. then h(x) is just p least significant bits of x

h(x)就是x面p个最低有效位

e.g. CLRS = ... + 83×128^0
⇓ divide
128 : 1010011
⇑ divide
ABCS = ... + 83×128^0

- $X_1 \equiv X_2 \pmod m \Rightarrow prob(collision) = \frac{1}{m}$
- But if keys : k, 2k, 3k, if k,m have common factor d
  ⇒ we only use $\frac{1}{d}$ of table

→ Good Practice : Choose m prime, not too close to powe of 2

- universal hashing : randomize hash function

Fix in advance : A set of hash function $\mathcal{H}$

$\mathcal{H}$: - easy
- cheap

☑ At run time choose $h \in \mathcal{H}$ at random

☑ Statistical Property : universal : for all keys $x, y \in U$, $x \neq y$

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{m} \quad m: \text{table size}$$

☑ example : multiply-add hash func

Run Time
- choose prime number $p > |U|$
- choose integer $a \in \{1, \cdots, p-1\}$ randomly
- choose integer $b \in \{0, 1, \cdots, p-1\}$ randomly

$\Rightarrow h_{a,b}(x) = ((ax+b) \bmod p) \bmod m$

$\in (0, p-1)$

$\in (0, m-1)$ ∈ table index

Worst case: $x_1 \neq x_2$ $\underset{a,b}{\text{Prob}}\{h(x_1) = h(x_2)\} = \frac{1}{m}$

$E(\text{collision}) = \frac{n}{m} = \alpha$

Week 3. Dynamic Programming

Fibonacci Members

0.1.1.2.3.5.8.13.21

$F(n) = F(n-1) + F(n-2)$ if $n > 1$ $\quad F(0)=0$ $F(1)=1$ ⇒ 转移方程

Exponential in n : $F(n) = 2^{0.694n} = (1.618)^n = \infty$ 指数级

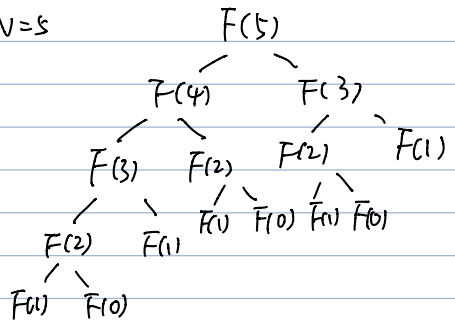
```
F(n) // 非负N
01 if n=0 return 0      O(1)
02 if n=1 return 1      O(1)
03 return F(n-1)+F(n-2) O(1)
```

T(n)= # computer steps to compute F(n)

$T(n) \le 2$ for $n \le 1$

$T(n) = T(n-1) + T(n-2) + 3$

e.g. N=5



⇒ Many computations repeated

迭代存储

```
01 if n=0 return 0
02 create f[0..n]
03 f[0]=0 f[1]=1
04 for i=2 to n
05     f[i]=f[i-1]+f[i-2]
06 return f(n)
```

O(n) ⇒ linear

Dynamic Programming (DP)

Knapsack Problem CLRS PP425-427

N items

Weight : $W_1, W_2, \dots W_n$

Value : $V_1, V_2, \dots V_n$

W : weight limit

} 正实数

Optimization Problem :

find $S \subseteq \{1, \dots, n\}$ subject to constraint : $\sum_{k \in S} W_k \le W$

↑ subject

object : $\max(\sum_{k \in S} V_k)$

(possible choices) : $2^n = C(n,0) + C(n,1) + \dots + C(n,n)$

△ exponentially

⇓ ? DP

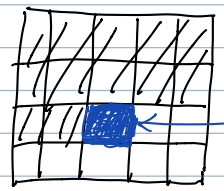
polynomial

⇒ 增加限制 ⇒ 有界特征 ⇒ W_i 和 W ⇒ 正整数

Theorem : can be solved in $O(nW)$ time

pseudo polinomial

DP Idea

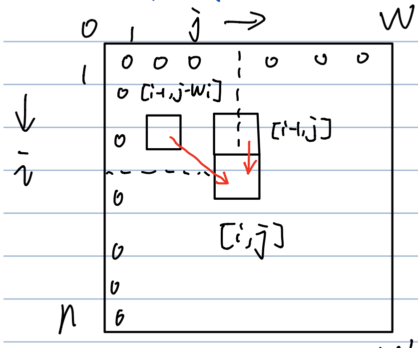


Subproblem:
a knapsack problem
with different inputs

↑ should be easy based on before result

状态转移表

For this case:



$(n+1)\cdot(W+1)$ table

i : item
j : weights

$M[i,j]$ = max <u>value</u> for $[i,j]$

$[i,j]$: in knapsack restricted to { items $1 \dots i$; sum(weight) $< j$ }

$$\Rightarrow \begin{cases} M[i,j] = \max\left\{ \sum_{k \in S} V_k : \sum_{k \in S} W_k \le j \right\} \\ S \subseteq \{1, 2, \dots, i\} \end{cases}$$

Recurrence

Base cases: $m[0,j] = 0$, $m[i,0] = 0$

$$m[i,j] = \max \begin{cases} m[i-1,j] & \text{if do not take item } i \\ m[i-1, j-W_i] + V_i & \text{if take item } i \end{cases}$$

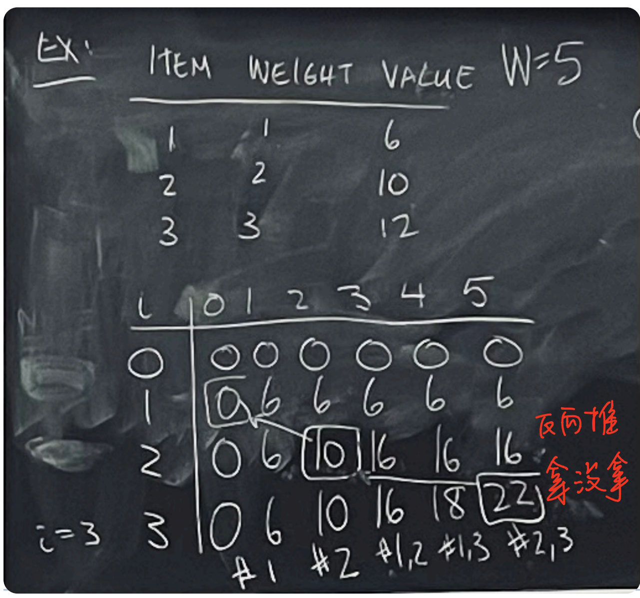
($j - W_i$: positive, 假如能放下 item i)

$O(nW) \Rightarrow$ space / time

Knapsack$(n, W[1..n], V[1..n], W)$

```
for i = 0 to n            // item        } O(n)
    m[i,0] = 0
for j = 1 to W            // weight      } O(W)
    m[0,j] = 0
for i = 1 to n                           } O(nW)
    for j = 1 to W
        if Wi > j:
            m[i,j] = m[i-1,j]   // too heavy
        else
            m[i,j] = max (m[i-1,j],
                          m[i-1,j-Wi] + Vi)
return m[n,W]
```

我们怎么知道包里放了什么?



Optional Knapsack$(M[0..n, 0..W], W[1..n], W)$

```
create array L[0..n]
k = 0          // # items in optional knapsacks
j = W          // unused capability
for i = n downto 1
    if m[i,j] > m[i-1,j]     // take i
        k = k+1
        L[k] = i
        j = j - Wi
return L
```

⇑ 算了不少没用的.
可以只算有用的吗?

Memorized-Knapsack (i, j)   // input $w[1..n]$, $v[1..n]$, W, $M[0..n, 0..W]$

全设成 -1

```
create M[0..n, 0..W]
for i = 0 to n   M[i,0] = 0
for j = 0 to W   M[0,j] = 0
if M[i,j] < 0                       // not compute
    if j < wi                       // item i too heavy
        value = Memorized-Knapsack (i-1, j)
    else
        value = max { Memorized-Knapsack (i-1, j)
                      Memorized-Knapsack (i-1, j-wi) + vi }
    M[i,j] = value   ⇐ 这一步很重要，保证了不会重复计算
return M[i,j]
```

「属于提拓展 不用掌握」

Initially call $M[n, W]$ ⇒ 也就是利用倒查 只算有用的部分

Longest Common Subsequence (LCS)   15.4 in CLRS

Problem: Give 2 sequences $X = [1..m]$ $Y = [1..n]$

Object: Find a longest subsequence common to both
↳ element must be in order but not need to be consecutive

Ex: X: A B C B D A B
Y: B D C A B A

{ BCBA, BCAB, BDAB }   length = 4
但结果不止一个

Brute Force Algorithm

$m \le n$ for every subsequence of X, check if it is a subsequence of Y, keep track of length

- worst case time: ?
  - 2^m subsequence of X to check
  - each check need $O(n)$
  - ⇒ worst $O(n2^m)$ expotential !! —DP?→ ☆Polynomial

Key Idea: LCS of X and Y can be expressed as smaller sequences

- if $x_m = y_n$, set last symbol of LCS to x_m
  find LCS of $X[1..m-1]$ $Y[1..n-1]$
- if $x_m \ne y_n$, then LCS of X, Y is { LCS $X[1..m-1]$ $Y[1..n]$; LCS $X[1..m]$ $Y[1..n-1]$ }

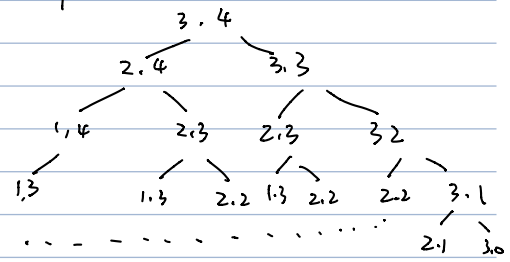
⇓ Recursive & DP

Recursive Method

```
LCS-length (X[1..m] Y[1..n])
01 if m=0 or n=0 then return 0
02 if Xm = yn
03     return 1+ LCS-length (X[1..m-1], Y[1..n-1])
04 else // Xm ≠ yn
05     return max { LCS-length (X[1..m-1], Y[1..n]),
06                  LCS-length (X[1..m], Y[1..n-1]) }
```

worst case : totally different

Ex: X: AAA
Y: BBBB



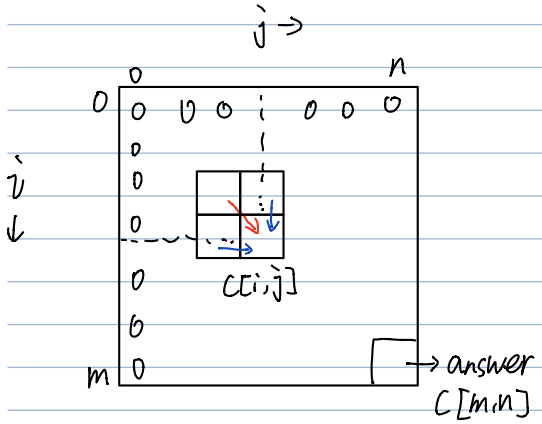
height: $O(m+n)$

time : $O(2^{O(m+n)})$

distinct problems: $(m \times n)$

DP method

- Table of LCS length
- subproblems : each recursive call of form : LCS-Length $(X[1..i]\ Y[1..j])$ $1 \le i \le m$ $1 \le j \le n$
- Table : m rows n columns $c[i,j] = LCS\text{-}Length\ (X[1..i]\ Y[1..j])$



Base Case :
$c[i,0] = 0$
$c[0,j] = 0$

$c[i,j] =$ if $x_i = y_j$ = $c[i-1, j-1] + 1$
if $x_i \neq y_j$ = $\max(c[i-1,j], c[i,j-1])$

```
LCS-length (X, Y, m, n)
01 create new table C[0..m, 0..n]
02 for i = 1 to m                    ] O(m)
03     C[i,0] = 0
04 for j = 1 to n                    ] O(n)        Time/space O(mn)
05     C[0,j] = 0
06 for i = 1 to m                    ] O(m·n)
07     j = 1 to n
08     if X[i] = Y[j]
09         C[i,j] = 1 + c[i-1, j-1]
10     else
11         C[i,j] = max (C[i-1,j], C[i,j-1])
12 return C
```

CLRS 15.2

Matrix Chain Multiplication

Parenthesization Problem: Optimal Evaluation Associative Expression

Given "chain" (sequence) of n matrices

$A_1 \cdot A_2 \cdots A_n$

where for $i=1 \cdots n$ matrix A_i has dimention $P_{i-1} \times P_i$

Object: To find the most efficient way to multiply these matrices

Given: A_1 A_2 A_3 .. A_n

Rows: r_1 r_2 r_3 ... r_n

Cols: c_1 c_2 c_3 ... c_n

Where: $c_i = r_{i+1}$ $1 \le i \le n$

不然沒法乘

兩矩陣相乘的 cost: $r_1 \cdot c_1 \cdot c_2$ ⇐

→ min (# of scalar multiplication)

EX: $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix} = \begin{pmatrix} 14 & 32 \\ 32 & 77 \end{pmatrix}$

2×3 3×2 → 2×2

12次計算

12 scalar multiplication

Fact: $12 = r_1 \times c_1 \times c_2$

EX A_1 A_2 A_3 A_4

1. $(A_1 A_2)(A_3 A_4)$
2. $(A_1 (A_2 (A_3 A_4)))$
3. $A_1((A_2 A_3) A_4)$

⋮ 共5種

Fact: $i \times j$ matrix with $j \times k$ matrix

⇒ # scalar multiplication : ijk

① $(A_1 \times A_2) \times A_3$

1×10^3 $10^3\times1$ 1×10^3

1×1

multi $= 1\times10^3\times1 + 1\times1\times10^3$

$= \boxed{2\cdot10^3}$

② $A_1 \times (A_2 \times A_3)$

$10^3\times10^3$

multi $= 10^3\times1\times10^3 + 1\times10^3\times10^3$

$= \boxed{2\cdot10^6}$

Week 4 Data Structures : Heap, Binary Search Tree (BST)

Data structures

e.g. { Linked-list : good for insert ; bad for search
Array : good for search ; bad for insert

Heap

Def: Realization of the abstract data type priority queue

priority queue : matains a set S of elements X, each associated with a numeric key k supports operations :

- insert (S, x)
- Min (S) : return element of S with smallest key
- Extract-Min (S) : return element of S with smallest key and delete it
- decrease-key (S, x, k) : decreases value of x's key to new value k

↓

Implementation of PQ:

Binary Tree : An array, represented as a binary tree (nearly complete)

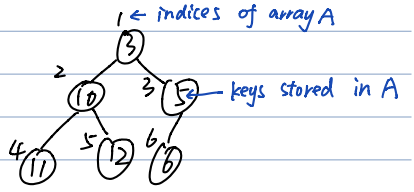
↓

arbitrary size arrays

e.g.

	1	2	3	4	5	6
A	3	10	5	11	12	6

1 ← indices of array A



Heap as tree:

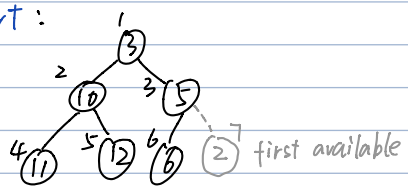
- Root: 1st element in array (i=1)
- parent[i] = $\lfloor \frac{i}{2} \rfloor$ (index of parent)
- left[i] = 2i (index of left child)
- Right[i] = 2i+1 (index of right child)

Min-heap Property (ordering Constraint)

key (parent(i)) ≤ key (i) // such as example A

Heap Operations:

- insert :



↓

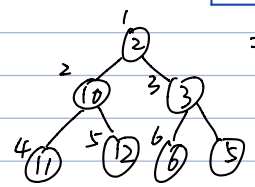
```
Min-heap-insert (A, key)
01 heapsize[A] = heapsize[A] + 1
02 i = heapsize[A]
03 while i > 1 and A[parent[i]] > key
04         A[i] = A[parent[i]]
05         i = parent[i]
06 A[i] = key
```

值2<5 swap?

2 < parent[7] = 5

key[7] ⟷ key[3]

直到 2 大于自己的 parent



comparisons:

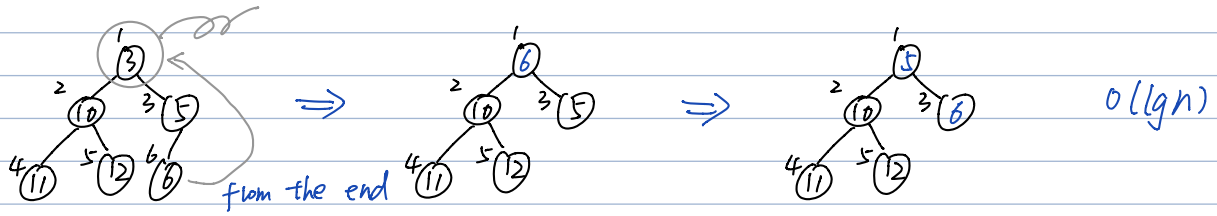
At most height of tree : O(lgn)

- **decrease-key**

similar to insert, except element already in tree $O(\lg n)$

```
decrease-key (A, key)
01 if A[i] ≤ key
02       return
03 while i > 1 and A[parent[i]] > key
04       A[i] = A[parent[i]]
05       i = parent[i]
06 A[i] = key
```

- **extract-min**



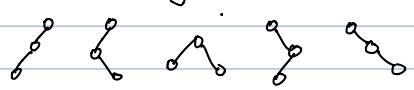
$O(\lg n)$

```
Heap-extract-min (A)
01 min = A[1]
02 A[1] = A[heapsize[A]]
03 heapsize[A] = heapsize[A] - 1
04 min-heapfy (A, 1, heapsize[A])
05 return [A]
```

```
min-heapfy (A, i, heapsize[A])
// left, right subtrees of i are min-heap
// goal: make i's subtree a min-heap
01 if left(i) ≤ heapsize[A] and A[left[i]] < A[i]
02       then smallest = left[i]   // if violation
03 else
04       smallest = i   // if no violation
05 if right(i) ≤ heapsize[A] and A[right[i]] < A[i]
06       then smallest = right[i]
07 if smallest ≠ i
08       then exchange (A[i], A[smallest])
09       min-heap (A, smallest, heapsize[A]
```

Binary Search Trees (BSTs)

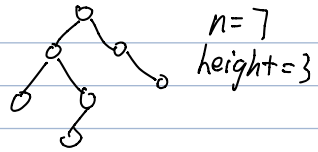
- **Def:** Maintains complete ordering of data, rooted binary tree
  - Rooted binary tree: defined on a finite set of nodes
    either (1) empty
    or (2) consists of a node called the root
    - Together with 2 rooted binary trees: left subtree, right subtree
    - order matters



- each node X contains:
  - key[x] (possibly other data)
    ← data field
  - three pointers: left[x]: point to left child

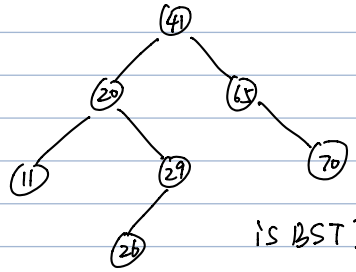
right[x] : point to right child

p[x] : point to parent



- height of tree : length (# edges) of longest simple downward path from root to leaf

- Defining property : BST properties
  - for any node x :
    - for all nodes y in left subtree of x
      - key[y] ≤ key[x]
    - for all nodes y in right subtree of x
      - key[y] ≥ key[x]



is BST? Yes!

- inorder-tree-walk (x) 先序遍历

```
01  if x ≠ nil
02      inorder-tree-walk ( left[x] )
03      print key[x]
04      inorder-tree walk ( right[x] )
```

11 -> 20 -> 26 -> 29 -> 41 -> 65 -> 70

How long take? O(n) => BIC visit and print each node once

- Tree-search (x, k) 从结点 x 开始找 k

```
01  if x == nil or k == key[x]
02      return x
03  if k < key[x]
04      return Tree-search (left[x], k)
05  else
06      return Tree-search ( right[x], k)
```

Initial call : tree-search (root[T] , k) Running time : O(height)

- Min , Max

```
Tree-min(x)
01  while left[x] ≠ nil
02      x = left[x]
03  return key[x]
```

```
Tree-max(x)
01  while right[x] ≠ nil
02      x = right[x]
03  return key[x]
```

O(height)

- Insert : dynamic : BST property must be preserved

```
Tree-insert (T, z) // to insert key v into BST T input node z (new node) with key[z] = v
                   // left[z] = nil  right[z] = nil
01  y = nil  // parent of root
02  x = root[T]
03  while x ≠ nil:
04      y = x
```

```
05        if key[z] < key[x]
06            x = left[x]
07        else
08            x = right[x]
09    p[z] = y
10    if y == nil
11        root[T] = z
12    else if key[z] < key[y]
13            left[y] = z
14    else    right[y] = z
```

O(n)

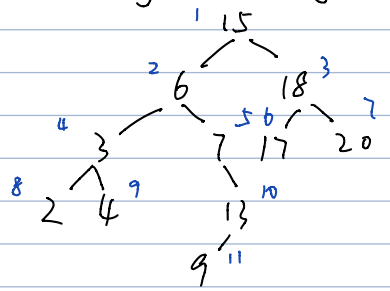
Successor

Assume all keys distinct

def: successor of node x is node y such that key[y] is the smallest key that > key[x]

```
Tree-Successor(x)
01  if right[x] ≠ nil
02      return tree-min(right(x))
03  y = p(x)
04  while y ≠ nil and x == right[y]:
05      x = y
06      y = p[y]
07  return y
```

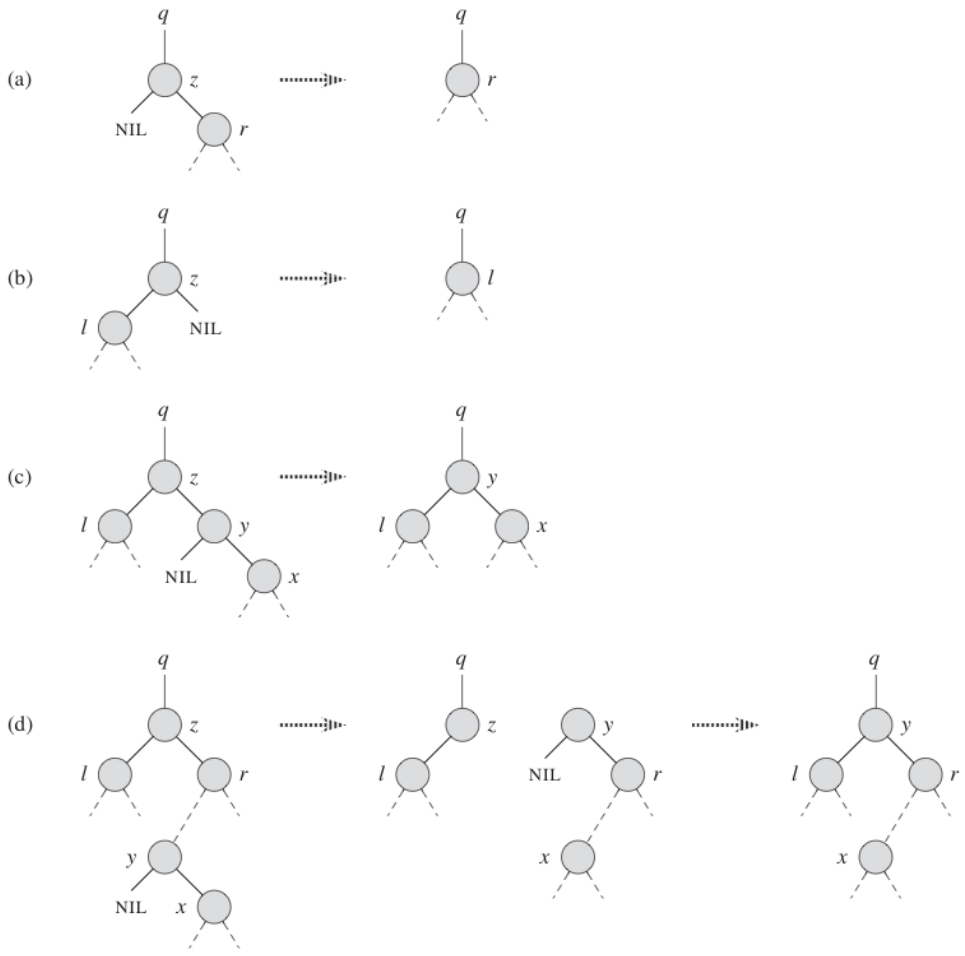
(찾고자 하는 값보다 큰 값)



Successor[Node(value=13)] = Node(value=15)

Successor[10] = 1

Delete z (찾 successor)



☆ part one:

Transplant : replace one subtree as child of its parents by another subtree

```
Transplant (T, u, v)                    // u: replaced   v: new
01  if p[u] == nil                      // u is root of T
02      root[T] = v
03  else if u = left[p[u]]              // u is left child
04      left[p[u]] = v                  // v is left child
05  else                                // u is right child
06      right[p[u]] = v                 // v is right child
07  if v ≠ nil
08      v.p = u.p
```

☆ part two:

Tree-delete-node

```
Tree-delete (T, z)
01  if left[z] == nil
02      Transplant (T, z, right[z])
03  else if right[z] == nil
04      Transplant (T, z, left[z])
05  else  y = Tree-min (Right[z])       // z has 2 children
06      if p[y] ≠ z                     // y is the successor but it not connect to z (condition d)
07          Transplant (T, y, right[y])
08          right[y] = right[z]
09          p[right[y]] = y
10      Transplant (T, z, y)
11      left[y] = left[z]
12      p[left[y]] = y
```

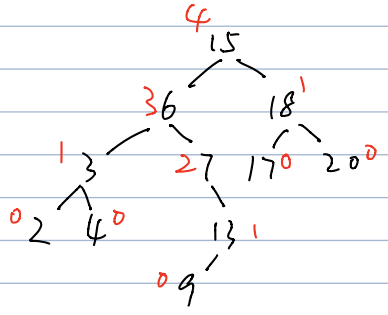
右孩子接到原位 → 接右手 → 換根 → 接左手

♥ Insert (T, z) => 加入有关 height 的信息

```
Tree-insert (T, z) // to insert key v into BST T  input node z (new node) with key[z] = v
                   // left[z] = nil   right[z] = nil
01  y = nil   // parent of root
02  x = root[T]
03  while x ≠ nil:
04      y = x
05      if key[z] < key[x]
06          x = left[x]
07      else
08          x = right[x]
09  P[z] = y
10  height[z] = 0
11  if y == nil
12      root[T] = z          // z is the only node in the tree
13  else if key[z] < key[y]
14          left[y] = z
15  else    right[y] = z
```

O(n)

Height



```
16    if left[y] == nil or right[y] == nil        // z is the only child of y
17          x = p[z]                              // 调高
18    while x ≠ nil
19          height[x] = height[x] + 1
20          x = p[x]
```

Delete (T, z) // 调整高度版 T augmented

```
    Tree-delete (T, z)
01  if left[z] == nil
02      Transplant (T, z, right[z])
03      x = p[z]
04  else if right[z] == nil
05      Transplant (T, z, left[z])
06      x = p[z]
07  else  y = Tree-min (Right[z])                 // z has 2 children
08        if p[y] ≠ z                             // y is the successor but it not connect
09            Transplant (T, y, right[y])           to z (condition (d))
10            right[y] = right[z]
11            p[right[y]] = y
12        Transplant (T, z, y)
13        left[y] = left[z]
14        p[left[y]] = y
15        y = height[z] - 1
16        x = p[y]
17  while x ≠ nil
18        height[x] = max { height[left[x]] } + 1
19        x = p[x]
```

Week 5 Graph

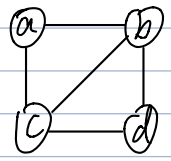
Graph: $G=(V,E)$

V: finite set of vertices $(V \neq \emptyset)$

E: set of edges (vertex pairs)

Directed / undirected

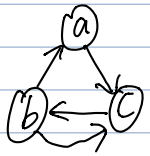
Undirected Graph:



$V=\{a,b,c,d\}$

$E=\{(a,b),(a,c),(b,d),(c,d)(b,c)\}$ Note: $(u,v)=(v,u)$

Directed Graph



$V=\{a,b,c\}$

$E=\{(b,a)(a,c),(c,b),(b,c)\}$ Note: $(u,v) \neq (v,u)$

loop:

$|V|$: number of vertices

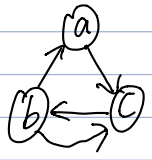
$|E|$: number of edges

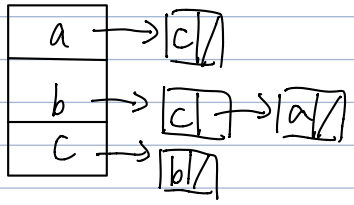
$0 \leq |E| < |V|^2$

Graph Representations:

Adjacency List

- V= set of vertices array of $|V|$ linked list
- for each vertex $u \in V$ $Adj[u]$ = list of vertices adjacent to u.





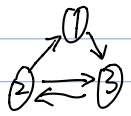
Good for sparse graph where $|E| << |V|^2$

space required: $\theta(|V|+|E|)$

Adjacency Matrix

Assume $V=\{1,2,\dots|V|\}$

Let $A=(a_{ij})=|V|\times|V|$ matrix $\Rightarrow$ a_{ij} (row, col) $\begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{o/w} \end{cases}$



$A=$

	1	2	3	
1	0	0	1	1
2	1	0	1	2
3	0	1	0	3

Good for dense graphs where $|E| \simeq |V|^2$

space = $O(V^2)$

Graph Search

Given : Graph $G=(V,E)$ Start vertex : $S \in V$

Explore : visit every vertex reachable from S

① : S is reachable

② : if u is reachable from S and $V \in adj[u]$ then v is reachable from S

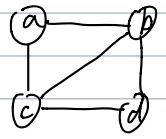
③ : if u reachable from S then there is a path from S to u

$S \to V_1 \to V_2 \to \cdots \to V_k \to u$ of length $k+1$ leading from S to u

⇒ find path algorithm

- if v discovered from u, then u parent of v
- keep track of parents $\pi[S] = nil$ S has no parent
  $\pi[v] = u$ $u \to v$
- length of path : #(Edges) in the path
  - ↳ simple : if all its vertices are distinct
    circle : a closed path $(v_0, v_1, \cdots, v_k, v_0)$

ex.



distance : $dist(u,v) = \delta(u,v)$ is the length of shortest path from u to v

Single-source Shortest paths problem

- select source vertex $S \in V$
- return : list of distances $dist(s,v)$ for all $v \in V$ $O(V+E)$
- tree : $\pi[v]$ $\pi[\pi[u]]$

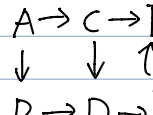
Breadth-First Search (BFS)

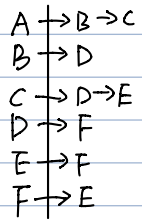
Input : $G=(V,E)$ $S \in V$ ADJ List format

Output : $d[v]$ = min # of edges in path from s to V
$\pi[v]$ = parent / predecessor of v in tree

Time : $O(V+E)$

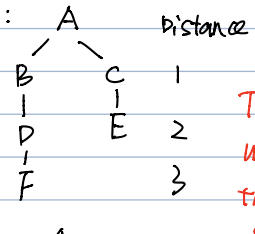
Ex: G



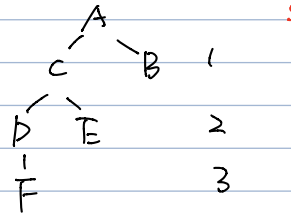


Source : A

BFS tree :



Tree is not unique, but the distance stay the same.



Label : Status of vertex { white : not yet discovered / scanned
gray : discovered / scanned
black : finished

Maintain 3 variables : color[v] : status
$d[v]$: distance
$\pi[v]$: parent

Maintain a list of gray vertices in order of discovery

when discover a vertex, add it to end of list

when finish a vertex, remove it from beginning of list

↳ FIFO → queue 佇列 { append to list $O(1)$ enqueue (Q, key)
remove from list $O(1)$ dequeue (Q, key)

↑ vertices discovered but not finished (only gray)

BFS (G, s)

```
01 for each u in V                          ]
02     color[u] = white                     ] O(V)
03     d[u] = ∞                             ]
04     π[u] = nil                           ]
05 color[s] = gray                          ]
06 d[s] = 0                                 ] O(1)
07 Q = ∅   // queue                         ]
08 enqueue(Q, s)                            ] O(1)
09 while Q ≠ ∅:                             ← O(V)
10     u = dequeue(Q)                       O(1)
11     for every vertex v in Adj[u]         ]
12         if color[v] = white              ]
13             color[v] = gray              ] O(E)
14             d[v] = d[u] + 1              ]
15             π[v] = u                     ]
16             enqueue(Q, v)                O(1)
17         color[u] = black                 O(1)
```

O(V+E)

Printed-Path (G, s, v)

```
01 if v == s                                ]
02     print "s"                            ]
03 else if π[v] == nil                      ] O(V)
04         print "no path"                  ]
05 else                                     ]
06     print path(G, s, π[v])               ]
07     print v                              ]
```

Depth-first-search (DFS)

Input: G(V,E) directed or undirected, in Adj List format
Goal: what parts of graph reachable from a given vertex
Output: d[v] : discovery time
f[v] : finish time
π[v] : parent of v on dfs tree
Time: Θ(V+E)

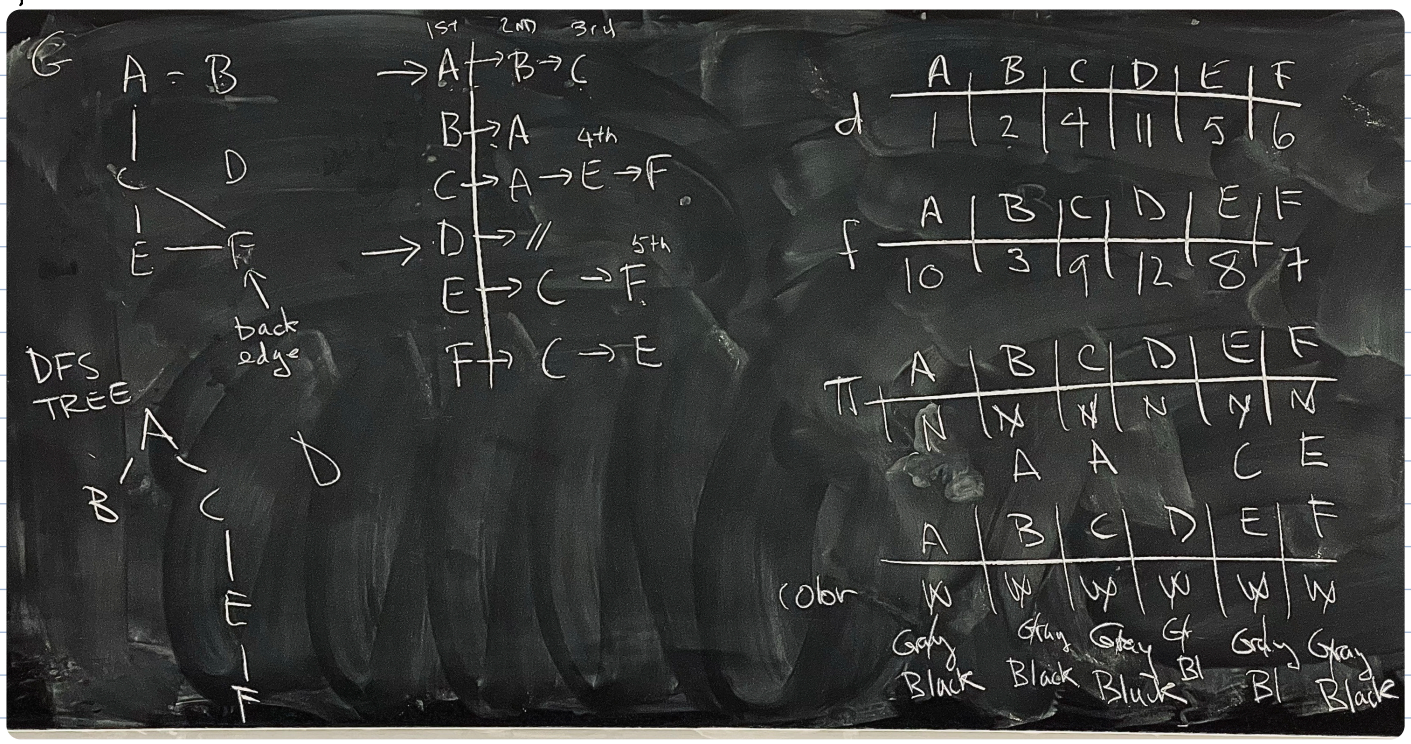
DFS (G) // Adj List Format

```
01 for each vertex u in V
02     color[u] = white
03     π[u] = nil
04 time = 0 // global variable
05 for each vertex u in V
06     if color[u] = white
07         DFS-visit(G, u)
```

DFS-visit (G, u)

```
01 time = time + 1
02 d[u] = time
03 color[u] = gray
04 for each v in Adj[u]
05     if color[v] == white
06         π[v] = u
07         DFS-visit(G, v)
08 color[u] = black
09 time = time + 1
10 f[u] = time
```

Ex for DFS process



- Not all edges in G1 are in DFS tree

Tree Edges	Back-Edges (None tree)
AB. AC. CE. EF	FC

Properties

- Tree Edge (u,v) : vertex v white when edge explored 1st time

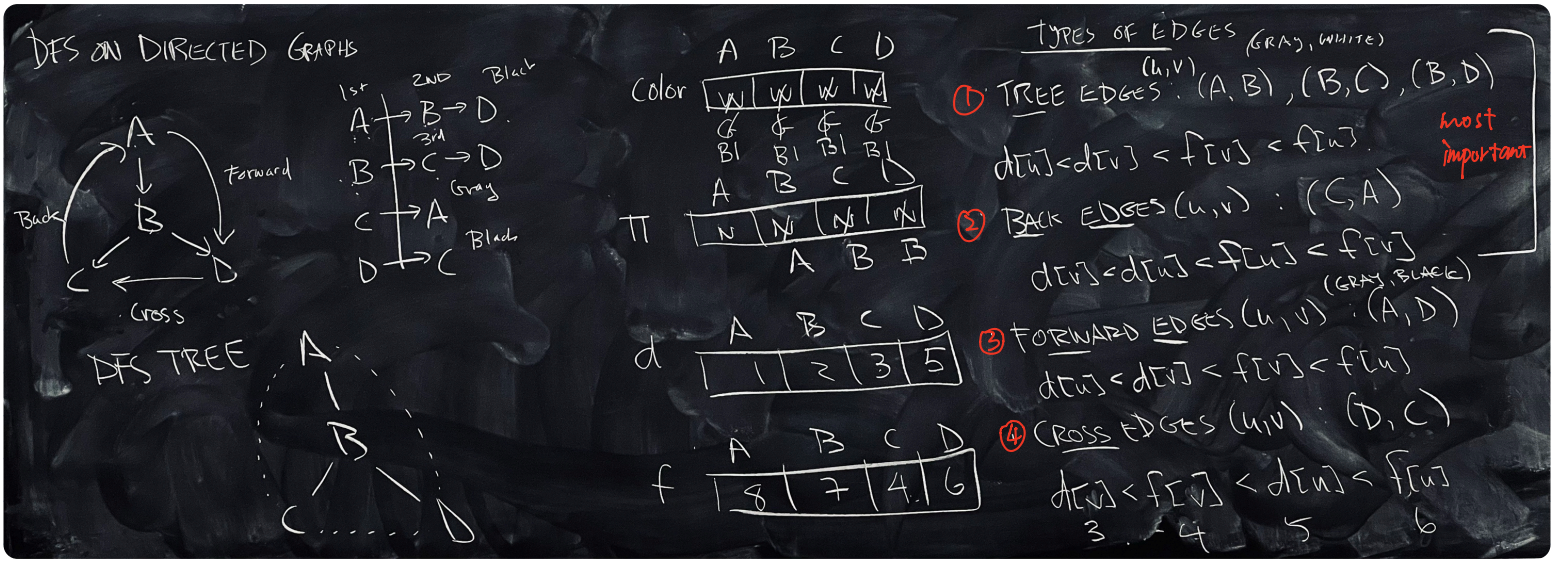
u=A, v=B

A	B	B	A
d[u] <	d[v] <	f[v] <	f[u]
1	2	3	10

- Back Edge (u,v) : vertex V gray when edge explored 1st time

u=F, v=C

C	F	F	C
d[v] <	d[u] <	f[u] <	f[v]
4	6	7	9

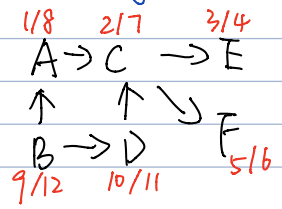


(*) tree, forward, cross $f[v] < f[u]$

(*) back $f[u] < f[v]$

Directed Acyclic Graphs (DAG)

Ex:

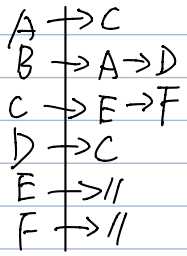


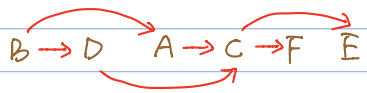
<= discover/finish

list: E F C A D B

output by finish time

reverse ⇒





topological sort :

Goal: a list of vertices such that all edges go from left to right

topological-sort (G)

01 $L = \phi$

02 DFS (G) to compute $f[v]$ $O(V+E)$

03 when vertex finish append to L

04 reverse L $O(V)$

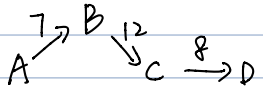
$O(V+E)$

Ex.

Week 6 Graph

Shortest path

weighted directed graph $G=(V,E,W)$ where $W: E \to \mathbb{R}$ associated with a real number to each edge



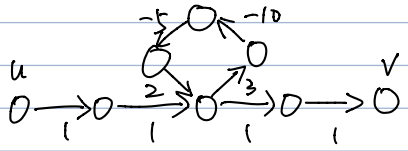
weight of path: path $p: v_1 \to v_2 \to \cdots \to v_k$. has weight $w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$

- shortest path: from u to v is a path of minimum weight from u to v
- shortest path weight: notation

$$\delta(u,v) = \min\{w(p) : p: \text{path from } u \text{ to } v\}$$

- when do shortest path not exist?
  - negative edge weights $\Rightarrow$ some shortest path may not exist

eg.



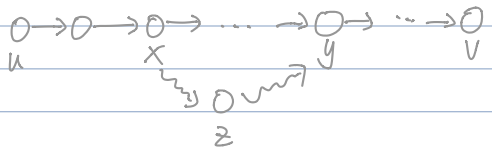
go around the circle never end

$\delta(u,v) = -\infty$

  - No path exist: $\delta(u,v) = \infty$

Claim: A subpath of a shortest path is a shortest path

Proof: suppose we have a subpath $x \to y$ of a shortest path $u \to v$



R: there is another path from $x \to z \to y$ shorter than $x \to y$

$\cdots \Rightarrow u \to v$ longer than $u \to z \to v \Rightarrow$ contradiction R is false

Dijkstra's Algorithm

- input: (G, w, s) $G=(V,E)$ directed, adj list format
  - w: weight function $w: E \to \mathbb{R}$
  - s in V: source vertex
- output: $d[v]$: distance $= \delta(s,v)$
  - $\pi[v]$: parent of v on shortest path to v from s

The unique path from s to each vertex in shortest paths tree is a shortest path

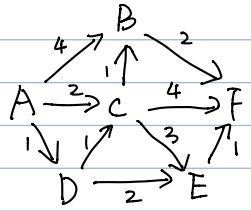
- Maintains a current cost to every vertex
  - $d[v]$: current min cost of reaching vertex V.
  - $\pi[v]$: parent of v

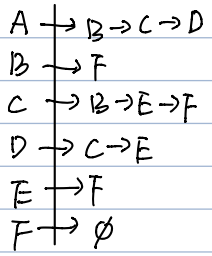
```
Dijkstra (G, W, s)
01 Initialize-single-source (G, s)
02 T = ∅        // Store the node has already find path
03 Q = V        // keyed by d[v]      Q: min-heap
04 while Q ≠ ∅:
05     u = extract-min (Q)
06     T = T ∪ {u}
07     for each v in adj[u]
08         Relax (u, v, w)
```

```
Initialize-single-source (G, s)
01 for each vertex v in V
02     d[v] = ∞
03     π[v] = nil
04 d[s] = 0
```

```
Relax (u, v, w)
01 if d[v] > d(u) + w(u,v)
02     d[v] = d(u) + w(u,v)
03     π[v] = u
```

Ex:

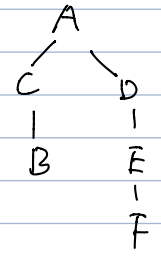




Q: ~~A~~ ~~B~~ ~~C~~ ~~D~~ ~~E~~ ~~F~~

	A	B	C	D	E	F
d	0	~~4~~ 3	~~∞~~ 2	~~∞~~ 1	~~∞~~ 3	~~∞~~ ~~5~~ 4
π	N	N ~~A~~ C	N ~~A~~ A	N A	N D	N ~~C~~ B ~~B~~ E

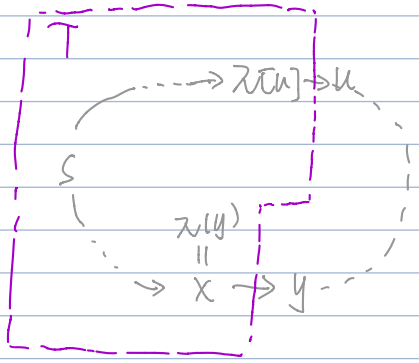
T : A D C B E F



Correctness: NTS: $d[v] = \delta(s,v)$ when Dijkstra terminate

LEMMA: when u is deleted from Q and added to T, $d[u] = \delta(s,u)$

Proof:
- Basic: at 1st iteration $u = s$ $d[u] = d[s] = 0$ and $0 = \delta(s,s)$
- Induction: Suppose u just deleted from Q.
- IH: For all vertices z previously deleted from Q $d[z] = \delta(s,z)$ NTS: $d[u] = \delta(s,u)$
- Suppose for contradiction: $d[u] \neq \delta(s,u)$, we know that for any v. when relax sets d[v] to a finite value, there is always evidence of a path of that weight. $\therefore d[v] \geq \delta(s,v)$ so if $d[u] \neq \delta(s,u)$ then $d[u] > \delta(s,u)$. thus there is a shorter path from s to u with weight $< d[u]$. call this path p.



Let y be 1st vertex on P that is not in T. (y might be u), let $x = \pi[y]$ on P. Then $x = \pi(y) \in T$, so by IH.: $d[x] = \delta(s,x)$

$\therefore d[y] \leq \delta(s,x) + w(x,y) \leq$ weight of $p < d[u]$

$\leq$? since (x,y) was relaxed when x added to T.

weight of optional path from s to u

$d[y] < d[u]$ means u is not min, contradiction!

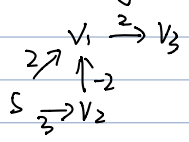
$\therefore d[u] = \delta(s,u)$

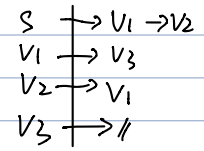
Time analysis:

Q: Data Struct	Times	min-heap	array	
insert (Q, v)	$\lvert V \rvert$	$O(\lg V)$	$O(1)$	$O(V^2)$ better for dense
extract-min (Q)	$\lvert V \rvert$	$O(\lg V)$	$O(V)$	
decrease-key (Q, v, new, key)	$O(E)$	$O(\lg V)$	$O(1)$	

with heap: $|V|O(\lg V) + |V|O(\lg V) + O(E)\cdot O(\lg V) = O((V+E)\lg V)$ better for sparce

Ex. where DIJKSTRA fails.





d	S	v_1	v_2	v_3
	0	~~∞~~	~~∞~~	~~∞~~
		②	3	4
		1		

π	S	v_1	v_2	v_3
	N	~~N~~	~~N~~	N
		S	S	v_1
			v_2	

这个时候 v_1 已经加入了 S. 但 $d[v_1]=2 \neq \delta(S,v_1)$

Bellman-Ford Algorithm

Matains: $d[v]$: current best estimate of shortest path weight from s to V
Innitially: $d[S]=0$ $d[v]=\infty$ $\forall v \neq S$
IDEA: Build a tree of shortest paths by relaxing all edges in arbitrary ordering then repeat, until edges relaxed $|V|-1$ times.

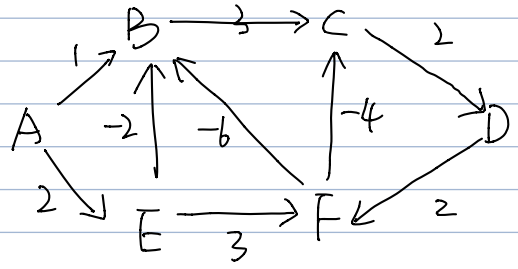
Bellman-Ford (G, W, S)

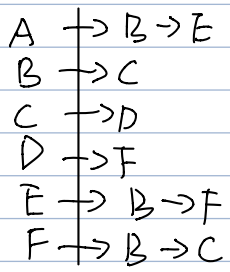
```
01 Initialize-single-source (G,s)
02 for i = 1 to |V|-1
03       for each edge (u,v) in E
04             Relax (u,v,w)
05 for each edge (u,v) in E
06       if d[v] > d[u]+w(u,v)
07             return False
08 return true
```

part1	01	O(V)
part2	02-04	O(VE)
part3	05-08	O(E)

} => O(VE)
if dense $O(V^3)$

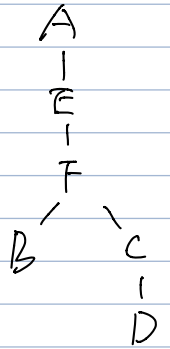
Ex





d	A	B	C	D	E	F
	0	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~
		~~1~~	4	~~6~~	~~2~~	8
		0	1	3		5
		-1		↑ second loop		

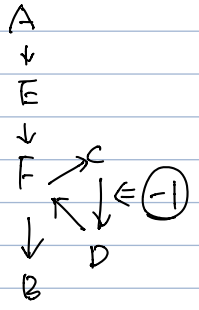
π	A	B	C	D	E	F
	N	N	N	N	N	N
		A	B	C	A	D
		E	F			E
		F				



有地 DF 探试 1. 我们有负圈 C→D→F 现在会怎样? 期末

	A	B	C	D	E	F
d	0	∞	∞	∞	∞	∞
		1	4	6	2	7
		0	1			5
		-1				
second loop		-2	0	3		4
third loop		-3	-1	2		3
		⋮	⋮	⋮		⋮
		⋮	⋮	⋮		⋮
		⋮	⋮	⋮		1

	A	B	C	D	E	F
π	N	N	N	N	N	N
		A	B	C	A	D
		E	F			E
						D



Does not converge. so the algo return false

Why this algo works?

- Invariant : in absence of neg-weight cycle d-values always either overestimate or exactly correct
  start at ∞ : only max change is by relaxing along an node

Relax (u, v, w) : $d[v] = \min\{d[v], d[u] + w(u,v)\}$ D.P.

1. it gives correct distance to v when u is the 2nd-to-last vertex in shortest path to v and d[u] correctly set.
2. it never make d[v] too small ∴ extra relax don't hurt

shortest path from s to t $s \to u_1 \to u_2 \to \cdots \to u_k \to t$

Week7 Minmun Spanning Tree

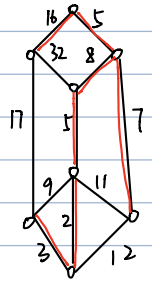
MST

Input: Connected, Undirected, $G=(V,E)$ with weight function $w: E \to \mathbb{R}$
No constraint on weight

output: A Spanning tree $T=(V,E')$ $E' \subseteq E$, of minmum weight $w(T) = \sum_{(u,v)\in T} w(u,v)$

Kruskal Algorithm

Ex.



• : take cheapest edge one by one
but cannot create cycle
2, 3, 5, 5, 7, 8, 16
weight = 46 Greedy Algorithm

Greedy Choice Property:
Locally optimal choices
lead to Global optimal Solution

⇓

kruskal's Algorithm

1. sort edges by weight
2. choose any edge with smallest weight put it in spanning tree constructing
3. successively consider remaining edges in order of increasing weight. Add edge to tree if it does not create a cycle with edges already in tree.

```
Kruskal (G, m)   // m = |E|
01 sort edges by weight : w(e1) ≤ w(e2) ≤ ··· ≤ w(em)
02 X := φ  //edge in MST
03 for i=1 to m
04     if ei does not create a cycle with edges in X
05         then add ei to X
```

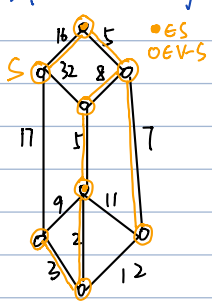
① 用 DFS
② 最简单的方法
树有 n-1 条边
若边 > n-1, 则有环

Time: $O(E \lg E)$ ⇐ $|E| \le |V|^2 \Rightarrow \lg|E| \le 2\lg|V|$
$\lg|E| = O(\lg|V|)$

⇓

$O(E \lg|V|) + O|V| + O|VE| = O(VE)$
sort union using DFS

Prim's Algorithm (Greedy)

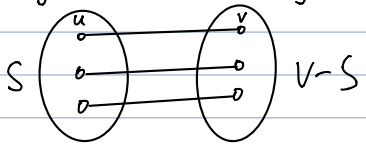


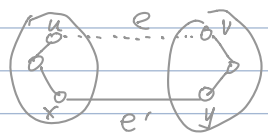
•∈S ∘∈V-S
Start from S 每次找与当前树相连所有边中的最小边

• start at a "root"
Then grow a tree from the root by selecting the least-weight edge with one vertex already in the tree

• cut property
Def: a cut in $G=(V,E)$ is any partition of vertices V into non-empty sets, S and V-S

Theorem: (cut property): For any cut (S, V-S) in a connected, weighted, undirected Graph $G=(V,E,w)$ any least-weight edge $e=(u,v)$ crossing the cut, ie $u \in S$, $v \in V-S$, is in some MST of G.





Proof: • let T be an MST of G, if T contains e. Done.
• If T not contain $e=(u,v)$, since T is a tree, there is a unique path between u, v. since $u \in S, v \in V-S$, this path must contain $e'=(x,y)$ crossing the cut. Add e, we have a cycle $\Rightarrow T \cup \{e\} - \{e'\} = T'$ T' is the ST contain e.

$w(u,v) < w(x,y)$ ∴ T' is a MST

- **cycle property**

Theorem (cycle property) let $G=(V,E)$ be a connected, undirected, weighted graph. let C be any cycle in G. let $e=(u,v)$ be a max-weight edge on C then e does not belong to some MST of G

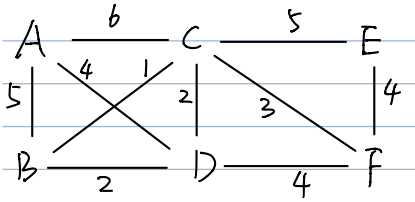
- **Prim's Algorithm** (what is relevant cut $(S, V-S)$?
  Let S be the subset of vertices in current tree T. (start with one vertex, r (root))
  Add cheapest edge e with exactly one endpoint in S.
  cut property asserts that e is in MST
  - Question: How to find cheapest edge with one endpoint in S?
  - Answer: Maintain such edges in a priority queue, delete min to determine next edge to add to T.

binary-min-heap

```
Prim(G, w, r)  // G=(V,E) adj list format, r∈V start vertex
01 for each u in V                                          O(V)
02     key[u] = ∞      // key[u] = weight of an edge connected u to a vertex in S
03     π[u] = nil
04 key[r] = 0                                               O(1)
05 Q = V  // ordered by key[u]                              |V| insert O(V)
06 while Q ≠ ∅
07     u = extract-min(Q)                                   |V| extract-min O(V)
08     color[u] = Black   // add to S
09     for each vertex v in adj[u]:
10         if color[v] = white and w(u,v) < key[u]          O(E)
11             π[v] = u                                    decrease-key
12             key[u] = w(u,v)
```

$O(V) + O(\lg V)(O(V) + O(V) + O(E))$
$= O(V) + O((V+E)\lg V)$
$= O((V+E)\lg V) = O(E \lg V)$
$|E| \geq |V|-1$

Ex:



A → B → C → D
B → A → C → D
C → A → B → D → E → F
D → A → B → C → F
E → C → F
F → C → D → E

(V-S): Q: A B C D E F
S: A D B C F E

key	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
		~~6~~ 2	~~6~~ 2	4		~~4~~
			1			
					~~3~~ 4	

π	A	B	C	D	E	F
	N	N	N	N	N	N
		A	A	A		
				D	D	~~D~~
				B		
					~~C~~ F	C

color	A	B	C	D	E	F
	W	W	W	W	W	W
	Bl					
					Bl	
		Bl				
				Bl		
						Bl

E - F - C - B - D - A

Kruskal Algorithm

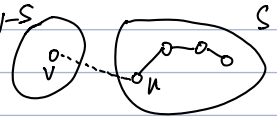
What is the relevant cut

- edges sorted in increasing order
- if adding $e=(u,v)$ to T does not create a cycle, then e is the min-weight with exactly one endpoint in S, so cut property asserts e in MST.

S: set of vertices in u's connected component

Question: How to check if adding an edge to T create a cycle?

⇒ Answer: Union find data structure



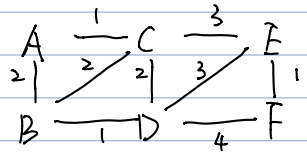
CLRS Union by rank P571

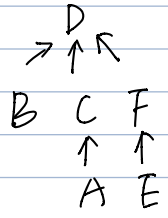
```
kruskal (G, w)
01 X = ∅        // maintain connected component (edges)
02 for each vertex u in V
03     Makeset(u)
04 Sort(E)
05 for each edge (u,v) in E
06     if Find(u) ≠ Find(e)      // if u,v in different CC's
07         X = X ∪ {(u,v)}       // add to X
08         Union(u,v)
09 return X
```

// each vertex has rank. height of subtree hanging from x

```
Makeset(x)
01 π(x) = x
02 rank(x) = 0

Find(x) // find root
01 while x ≠ π(x)
02     x = π(x)
03 return x

Union(x,y)
01 Link(Find(x), Find(y))

Link(x,y)
01 if rank(x) > rank(y)
02     π(y) = x
03 else π(x) = y
04     if rank(x) = rank(y)
05         rank(y) = rank(y) + 1
```

Ex:





② sorted edge

(A,C,1) (B,D,1) (E,F,1)
(A,B,2) (B,C,2) (C,D,2)
(C,E,3) (D,E,3) (D,F,4)

π	A	B	C	D	E	F
⓪	~~A~~	B	C	D	E	F
①	C					
②		D				
③					F	
④			D			
⑤						
⑥						
⑦						D
⑧						
⑨						

Rank	A	B	C	D	E	F
⓪	0	0	∅	∅	0	∅
①			1			
②				1		
③						1
④				2		
⑤						
⑥						
⑦						
⑧						
⑨						

⓪ ① ② ③ ④ ⑦

X = {(A,C), (B,D), (E,F), (A,B), (C,E)}

① (A,C) Link(A,C) π(A)=C rank(C)=1
② (B,D) Link(B,D) π(B)=D rank(D)=1
③ (E,F) Link(E,F) π(E)=F rank(F)=1
④ (A,B) Link(Find(A), Find(B)) = Link(C,D) π(C)=D Rank(D)=2
⑤ (B,C) find(B) = find(C) do nothing
⑥ (C,D) find(C) = find(D) do nothing
⑦ (C,E) Link(find(C), find(E)) = Link(D,F) π(F)=D
⑧ (D,E) Find(D) = Find(E) = F do nothing
⑨ (E,F) Find(E) = Find(F) = F do nothing

Complexity kruskal

Times
- |V| makeset's
- 2|E| find's
- |V-1| union

Cost
- O(1)
- proportional to height O(lgV)
- O(1)

Total cost = O(V) + O(E lgV) + O(V lgV) + O(E lgV)

= O(E lgV)

- cycle property

Theorem (cycle property) let $G=(V,E)$ be a connected, undirected, weighted graph. let C be any cycle in G. let $e=(u,v)$ be a max-weight edge on C then e does not belong to any MST of G

Proof for cycle property: Let T be a MST of G. Suppose $e=(u,v) \in T$.
Assume $e \notin C$. e' is on path from u to v in C
Remove e from C. consider: $T-\{e\}$. now we have 2 set of vertex: X and $V-X$. Because C is a cycle. $\exists e' \in C$ $e \neq e'$. and e' crosses the cut $(X, V-X)$, and is on the path from u to v. Since e is max-weight edge in C, $w(e') < w(e)$, the new MST T' has less weight than T
$\therefore w(T') < w(T) \Rightarrow$ contradiction

Week 8 Network Flow

Flow network: Directed graph with non-negative edge weights called capacities
edges: carry some measurable quantity
each edge has a given capacity: Limit amount to quantity edge carry

DEF: Flow network: a flow network: directed graph $G=(V,E)$ with a source vertex $s \in V$, a sink vertex $t \in V$ and edge capacity $c: V \times V \to \mathbb{R}_{\geq 0}$ Define $c(u,v)=0$ for any pair of vertices $(u,v) \notin E$

DEF: Flow: a flow on G is a function $f: V \times V \to \mathbb{R}$ satisfying:

capacity: $0 \leq f(u,v) \leq c(u,v) \quad \forall u,v \in V$

conservation: for every vertex $v \in V$, $v \neq s,t$, $\sum_{u \in V} f(u,v) = \sum_{u \in V} f(v,u)$

$\sum_{u \in V} f(u,v)$: flow into v; $\sum_{u \in V} f(v,u)$: flow out of v

DEF: Value: The value of a flow f $|f| = \sum_{v \in V} f(s,v)$ i.e. the sum of flow out of source

DEF: Saturated: an edge is saturated if $f(u,v) = c(u,v)$

DEF: Residual capacity: Given a flow f on a network $G=(V,E)$ for each pair of vertices u,v in V. The residual capacity $c_f(u,v)$ is defined:

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E \\ f(u,v) & \text{if } (v,u) \in E \\ 0 & \text{o/w: } (u,v)(v,u) \notin E \end{cases}$$

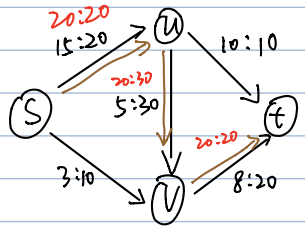
DEF: Residual Graph: let f be a flow on a network $G=(V,E)$
The residual graph $G_f = (V, E_f)$

$E_f = \{ (u,v) \in V \times V : c_f(u,v) > 0 \}$ NOTE: G_f has edges whenever $c_f(u,v) > 0$

DEF: Augmenting Path: an augmenting path P is a directed path $s \to t$ in residual graph G_f

Maximum Flow Problem : Given a network $G=(V,E,c,s,t)$ Find a flow of max value.

Ex:



$f(u,v)$

$f: c$ i.e $f(u,v)=5$

$c(u,v)=30$

IDEA:

① try Greedy

1: find an s to t directed path in G : always choose max capacity edge

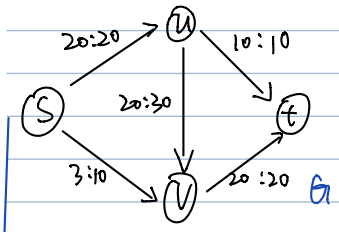
2: find the min(capacity) of edges on the path

capacity of path = $\min\{c(u,v) : (u,v) \text{ on path } p\}$

3. Assign $f(e) = c(p)$ for all edges e on path

4. repeat until no s-t unsaturated path in G

IDEA 2 : Ford-Fulkerson ALGORITHM

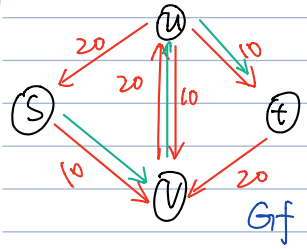


⇐ Do greedy first

G_f : edges in residual Graph

Forward edges:

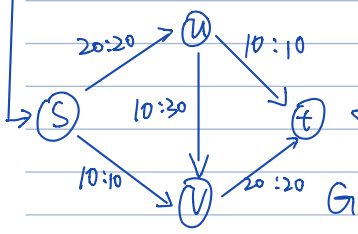
for each $(u,v) \in G$ for which $f(u,v) < c(u,v)$, create an (u,v) in G_f. Assign it capacity $c_f(u,v) = c(u,v) - f(u,v)$



Backward edges:

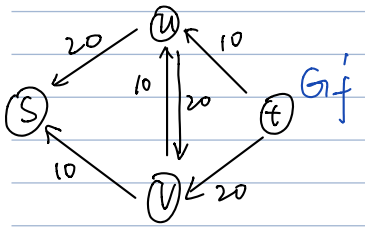
for each $(u,v) \in G$ for which $f(u,v) > 0$ create edge (v,u) in G_f. with $c_f(v,u) = f(u,v)$.

NOTE: G_f has edges whenever $c_f(u,v) > 0$



find augmenting path : $c_f(p) = 10$

Augment flow f in G. using Path p and $c_f(p)$. $f'(e) = f(e) + c_f(p)$ if e is a forward edge. $f'(e) = f(e) - c_f(p)$ if e is a backward edge

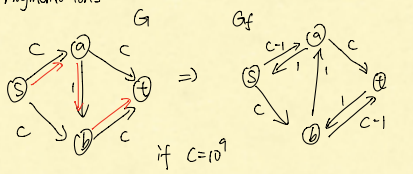


Ford-Fulkerson Algorithm

```
01  f(u,v) = 0   ∀ u,v ∈ V
02  Build residual graph Gf
03  while (there exist an augmenting path)
04        find augmenting path p
05        compute capacity cf(p) of p
06        augment flow by cf(p) along p
```

Analysis: O(E) time per augmentation

-capacities integers $\in [0,C]$ then at most $|E|C$ Augmentations



if $C=10^9$

we need $|E|C$ augmentations

Prove: Does ALG terminate having found max flow? analyze running time

Claim: if capacities are integers, then ALG terminates

Proof: $|f'| > |f|$. in fact, $|f'| = |f| + c_f(p)$ $c_f(p) > 0$

∴ flow is strictly increasing

There is only finite capacity coming out of S

By capacity constraint $f(s) \le c(s)$: S all edges (s,x), $x \in V$, $c_f(p) \ge 1$

∴ smallest the flow can increase on each iteration is 1

∴ with finite bound on $c(s)$, will come to halt and cannot increase flow anymore. □

Irrational : 2wicky

DEF: Cut : partition of vertices V into 2 subsets : $S, V-S$

DEF: S-t Cut : a cut with $s \in S$ $t \in V-S$

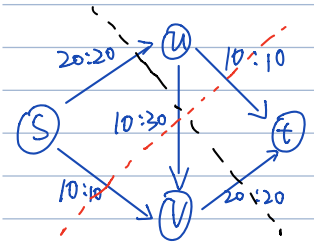
DEF: Capacity of a cut : $C(S,T) = \underbrace{\sum_{u \in S} \sum_{v \in T} c(u,v)}_{e \text{ out of } S}$

DEF: Flow across cut : $f(S,t) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{v \in T} \sum_{u \in S} f(v,u)$ Claim: $f(S,T) \leq C(S,T)$

Corollary: let (S,T) be any s-t cut and f any s-t Flow Then $|f| \leq C(S,T)$
($|f|$: value of flow; $C(S,T)$: capacity of cut)

Theorem: for any flow network G, $|f| = C(S,T)$ for some f and some cut (S,T)
Max flow = Min cut

(1) f is a max flow
(2) G_f has no augmenting path
(3) $|f| = C(S,T)$ for some cut (S,T)



$\Rightarrow S = \{s, v\}$ $C(S) = 40$

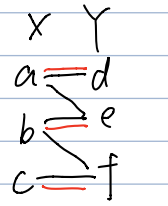
$\Leftarrow S = \{s, u\}$ $C(S) = 50$

$S = \{s\}$ $C(S) = 30$

Edmonds - Karp
1. use BFS to find augmenting path
2. use dijkstra "fastest path"

$O(|V|E|)$ Augmentations
$O(VE^2)$ Running time

Maximum Bipartite Matching
DEF: a bipartite graph (undirected) $V = X \cup Y$, $X \cap Y = \emptyset$, and every edge e in E has one end-point in X and one end point in Y



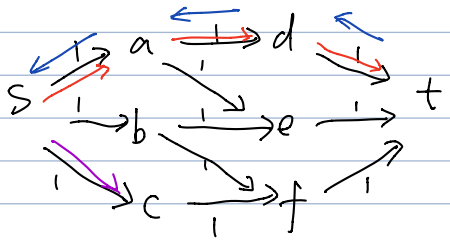
max matching
$|M| = 3$

A matching M in $G = (V,E)$ $M \subseteq E$ such that for all vertices $v \in V$, at most one edge of M is incident on V

A maximum matching is a matching M such that for any matching M', $|M| \geq |M'|$

Given: undirected bipartite graph $G = (V,E)$ $V = X \cup Y$
- construct a flow network $G' = (V', E')$ for G

- Direct all edges in G from X to Y
- Add s,t & V V' = V ∪ {s,t}
- Create new directed edges from s to each x ∈ X
  Create new directed edges from each y ∈ Y to t
- Assign unit capacity to each edge in E'



|f| = 1

residual

8 ⊤ ⇒ { 4 non-graph, 4 graph

Week 9 NP-complete Problems

- MST in Graph
- Shortest Paths in Graph
- Max Flows in Networks
- Matchings in Bipartite Graphs

Problems with running times $O(n^k)$ for some $k>0$
n: input size
efficiently solvable or tractable

Problems whose running time cannot be bounded by $O(n^k)$ for any $k>0$ are intractable

$(n!)$

1. Satisfiability (SAT)

Given a Boolean formular in CNF (i.e. conjunctions of clauses), each clause is the disjunction of several literals, where a literal is a boolean variable or its negation

E.x. $(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2) \wedge (x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$
① ② ③ ④ ⑤

Verify one solution
$O(mn)$: n: # variables
m: # clauses

A satisfying truth assignment (TVA) is an assignment of true or false to each variable so that every clause contains a literal whose value is true

SAT Problem: Given a Boolean Formula $f(x_1, x_2, \cdots x_n)$ on n variables, is there a TVA to $x_1, x_2 \cdots x_n$ that makes $f(x_1, x_2, \cdots x_n)$ = TRUE?

(Take e.x. ②③④ force ⇒ $x_1, x_2 x_3 = T$ ⇒ ① or ⑤ false) ∴ E.x. false
or $x_1 x_2 x_3 = F$

How decide in general? ⇒ Truth Table : 2^n Rows = 2^n Assignment $O(2^n)$ Exponential

Finding Solution:

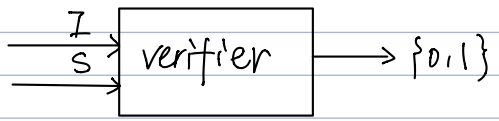
Better Method? ⇒ No currently

Special case: 2 SAT $(x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_3) \wedge (x_1 \vee x_2)$ $T \frac{T}{F} T$ ⇒ polynormial time (Advanced Algorithm)

P and NP Problem

DEF: Class of all problems that can be solved in polynomial is P

DEF: A problem is in class NP if there exists a poly-time verifying ALG that takes an instance I of X and a proposed solution S and outputs true I.F.F S is a solution to solution I. i.e. $V(S, I) = 1$



Note: Any problem in P is also in NP ∴ $P \subseteq NP$

IS $P = NP$? (if "=" ⇒ every problem in the world can be solved in poly-time)

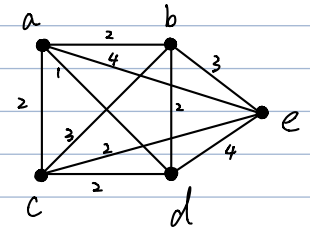
Consensus: NO ⇒ not proved

traveling salesman problem (TSP)

Given n vertices $1, \dots, n$ and all $\frac{n(n-1)}{2}$ distance between them, and a budget b
Is there a tour, i.e. a cycle that passes through each vertex exactly once of total cost $\le b$?
i.e. permutation $\tau(1), \dots, \tau(n)$ of vertices such that total distance is at most b:

$$d[\tau(1), \tau(2)] + \dots + d[\tau(n-1), \tau(n)] + d[\tau(n), \tau(1)] \le b$$

E.X.



$b = 10$. is there a tour of cost ≤ 10?
Yes: (a.d.b.e.c.a)

How decide in general? exhaustive search: try all $(n-1)!$ tours

DP ALG: $O(n^2 2^n)$: still exponential

$TSP \in NP$

MST: Given a distance (weight) matix and a bound b is there a spanning tree with total weight $\sum_{(i,j) \in T} d(i,j) \le b$? $MST \subseteq P$ $MST \subseteq NP$

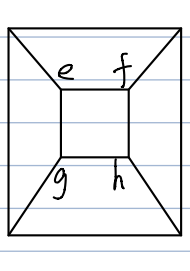
What make the huge difference? MST: spanning tree
Tsp: spanning tour $\Leftarrow$ tree that not allow to branch

Approximation ALG for TSP: MST + Matching 3/2 of optimal

3. Hamilton Cycle

A hamilton cycle in undirected graph $G = (V, E)$ is a simple cycle that goes through every vertex exactly once.

Ex. a



$\Rightarrow$ a,e,g,c,d,h,f,b,a

cycle problem: No poly time ALG know for this problem.

Euler Cycle? goes every edge exactly once.
(every vertices has even edgree)
Linear time

Table of problems

Hard	Easy (P)
SAT, 3SAT	2SAT
TSP	MST
HAM cycle	Euler cycle

NP

i.e. is $P \neq NP$?

Reductions

Reduce A to B $\quad A \overset{?}{\le_p} B$ ← know how to solve

"reduce to"

hard $A \le_p \overset{?}{B}$ ← Also hard

"reduce to"

Reduction Compose: if $A \leq_p B$ and $B \leq_p C$, Then $A \leq_p C$

DEF: A problem A is NP-complete if

1. A is in NP
2. for all problems Q in NP, $Q \leq_p A$

NP-Hard $Q_1 \leq_p Q_2 \leq_p \cdots \leq_p Q_k \leq_p A$

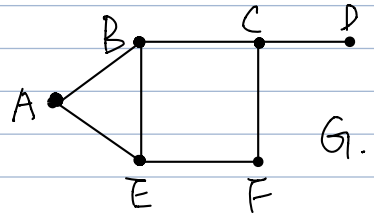
1st problem: SAT (Cook, 1971)

3 $SAT \leq_p$ IND SET $\Rightarrow$ IND SET is NP-complete

Reduction Example

3 set ≤ Independent Set

Def: an independent set in an undirected graph $G=(V,E)$ is a subset $S \subseteq V$ such that for all $x,y \in S$, $(x,y) \notin E$. No two vertices in S are joint by an edge.



G_1.

{A, D, F} is an IS in G_1.

Independent set problem: Given $G_1=(V,E)$ positive integer $k \leq |V|$, is there an ISet $S \subseteq V$ of size k?

3SAT: 3 CNF Formula to find a satisfying TVA.
must pick one literal from each clause and give it truth value
TRUE choices must be consistent

E.X. $(x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \bar{x_3} \vee \bar{x_4}) \wedge (\bar{x_1} \vee x_3 \vee x_4) \wedge (x_1 \vee \bar{x_2} \vee \bar{x_4})$

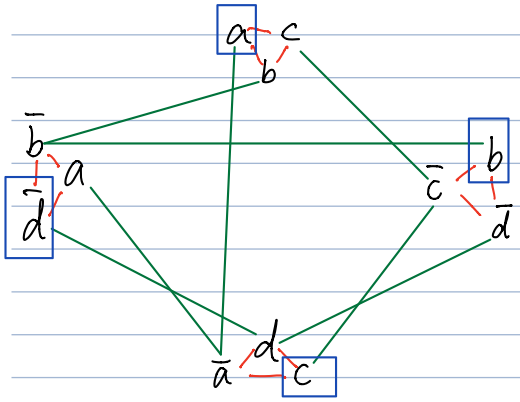
4 choices: TVA: $x_1=T, x_2=T, x_3=T, x_4=T$

$\Downarrow$

n: variables
m: clauses

Reduction: Graph: each clause becomes a triangle

E.X. Φ: $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$ $k=4$



$S = \{a, b, c, \bar{d}\}$

Claim: G contains an IS of size exactly k.
I.F.F. the original formula Φ is satisfiable