Describe Algorithms

Characterizing Running Times

If running time of an alg bounded by some linear function of n. above and below.

$\Theta(n)$.

If we have 2 functions $f(n)$ and $g(n)$, we say that $f(n)$ is $\Theta(g(n))$ if $f(n)$ is within a constant factor of $g(n)$ for sufficiently large n.

$f(n) = \frac{n^2}{4} + 100n + 50$. $\Theta(n^2)$.

Running time for worst case $\Theta(n)$.
best case $\Theta(1)$.

$O(n)$ indicates running time never worse than n. (or constant times of n).

$f(n) = O(g(n))$ if once n becomes sufficiently large. $f(n)$ bounded by some constant times of $g(n)$.

If $f(n)$ is never better than $g(n) \Rightarrow \Omega(g(n))$.

$\Omega(n)$ is a lower bond.

Complexity Analysis

Analyse Algorithm based on number of operations performed.

$T(n) = n^2 + n$. $n^2 + n \leq 2n^2$ $n^2 - n \geq 0$ $n(n-1) \geq 0$

$T(n) \leq cf(n)$ for $n \geq m$. $c = 2$. $m = 1$ $n \geq 1$.

Big-O notation → intended to indicate efficiency for large values of n.

Constant of Proportionality

c is only important when 2 algs have the same $O(f(n))$

Constructing $T(n)$.

count number of operations made.

Classes of Algos.

f.	
1	const.
$\log n$.	logarithmic
n	linear
$n \log n$.	log linear.
n^2	quadratic
n^3	cubic.
a^n.	exponential.

ADT Abstract Data Types

Data items. – sequence of binary digits

binary sequence – could be char / int / real.

Data Type. – distinguishes between types of data

Programming language provided datatypes (primitives)

: ① simple DT. integer / real

② complex DT. objects / lists.

Abstraction.

① Procedural / functional sqrt(x).

② Data abstraction String.

Lvls of Abstraction.

Software-implemented Big Ints
High lvl log Instructions
ASM log Instructions

Hardware. Int.

ADT is a program-defined data type that specifies a set of data values and a collection of well-defined operations

Interact with ADT through a interface / set of pre-defined operations

ADT Examples List. / Set.

Bag ADT

|

Implementation with List / Dictionary / Map

Queue ADT

Implement a queue with circular Array

Good: add elements without moving the elements

Bad: maximum capacity unknown, might override.

→ usually implemented with specified max size

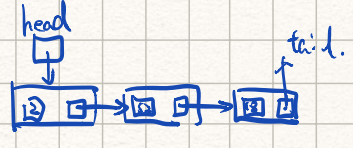
Linked lists

disadvantages of array / lists

size of array fixed.

Insertion / deletion time consuming

Singly linked list.



doubly linked list.

each node has prev & succ.

Circularly linked list.

tail <-> head. allows complete traversal from any point.

Using tail reference. - Appending.

Sometimes we can have two external references.

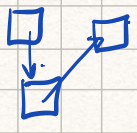
head / tail.

Append. add node first
tail set to last node. (new)

Remove.
tail set to prev node.
delete node

Inserting into

An unsorted list (placement not important)
Sorted list.
insert in middle.



Sorting.

Binary Search.

Assume sorted array

A[l...r]
find $\frac{l+r}{2}$.
if $A[\frac{l+r}{2}] = x$ stop.
else search $A[l, \frac{l+r}{2}-1] - [\frac{l+r}{2}+1, r]$

Selection sort

for i: 0 to n-1
smallest set to : (current unsorted left most)
for j: i+1 to n.
if (smallest.val > a[j].val) smallest = j (keep updated of the smallest index)

swap(i, smallest)

inner loop runs (n-i) times

$T(n) = n + (n-1) + \cdots + 1$.

$= \frac{(n+1)\cdot n}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$

running time $\Theta(n^2) \Rightarrow O(n^2)$
$\Omega(n^2)$

Insertion sort

subarray A[0..i-1] contains sorted. elements in i positions of the array

to determine the index of A[i].

go from A[i-1] to A[i], shift elements by one to the right.

for i = 1 to n.-1.

key = i.val. j = i-1

while j ≥ 0 and. j.val > key

j+1.val = j.val. move to right.

j--;

j+1. val = key.

Best case:

j goes zero change every time.

happens when already sorted

i loops n-1 times $\Theta(n)$.

Worst case:

j goes to end every time.

reverse ly sorted.

$1+2+\cdots+(n-1) = \frac{n(n-1)}{2}$ $\Theta(n^2)$.

S.t. Insertion sort $O(n^2)$ $\Omega(n)$

Merge sort.

Divide A[0..n] is divided to $A[0, \frac{n}{2}]$, $A[\frac{n}{2}+1, n]$ $\Theta(1)$ time

Conquer / Combine | Merging two sorted arrays using a new array.

$\Theta(n)$ time for each combination.

total of $\lg n$ times of combination.

$T(n) = 2T(\frac{n}{2}) + \Theta(n)$.

$T(n) = \Theta(n \lg n)$ best and worst case

Quicksort.

divide-and-conquer.

qsort works in place.

running time $\Theta(n^2)$

avg. case running time $\Theta(n \lg n)$

Divide.

pick an element called pivot.

re-arrange the array according to the pivot.

qsort (array-left) and array-right.

don't need to combine / sorted in place

Linkedlist sorting. insertion sort / bubble sort.

Bubble sort inefficient. $\Theta(n^2)$ compare every two elements

Selection / Insertion sort $\Theta(n^2)$.

qsort $O(n^2)$ $\Theta(n \lg n)$ (av.g.)

mergesort $\Theta(n \lg n)$ extra memory / not in place

Radix sort.

have 10 bins (or less bin according to radix)

distribute keys in ones column and collect them from bin 0-9.

tens

:

depending on how many digits

Assume max has d-digit, array nets with n radix.

$\Theta(d(n+n))$ given, d, n constants.

take from bins. put in bins.

$\Theta(n)$

Quicksort

performance effected by the choice of pivot.

Worst case $\Theta(n^2)$: always choose the element that makes the partition completely unbalanced

Best case $\Theta(n \lg n)$ fairly balanced choice of pivot

Cost of Qsort.

$$T(n) = (T(q) + T(n-q-1)) + \Theta(n)$$

For worst case $T(n)_{max} = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n)$

Assume. $T(n) = O(n^2) \le cn^2$.

$$T(n)_{max} = \max(\underbrace{cq^2 + c(n-q-1)^2}_{\Downarrow}) + \Theta(n)$$

$$0 \le q \le n-1$$

$$\frac{\partial f}{\partial q} = \frac{c\partial(q^2 + n^2 + q^2 + 1 - 2nq - 2n + 2q)}{\partial q}$$

$$\frac{\partial f}{\partial q} = 4q - 2n + 2. \ge 0 \quad \text{is local minima.}$$

$$\frac{\partial^2 f}{\partial q} = 4 > 0$$

$q = \frac{n-1}{2}$ (min)

$q = 0$ $q = n-1$ (max)

$f(0) = (n-1)^2$.

$$T(n) \le ((n-1)^2 + \Theta(n) = cn^2 - 2cn + c + \Theta(n) \le cn^2.$$

$$T(n) \in O(n^2)$$

$$T(n) \ge c\frac{1}{2}(n-1)^2 + \Theta(n) = c\frac{(n^2+1)}{2} \in \Omega(n^2)$$

$$\therefore T(n) \in \Theta(n^2).$$

Best case analysis.

$$T(n) = 2T(\tfrac{n}{2}) + \Theta(n)$$

master theorem. $T(n) = aT(n/b) + f(n)$

if $f(n) = \Theta(n^{\log_b a})$ $T(n) = \Theta(n^{\log_b a} \log n)$

$a = 2. b = 2.$ $f(n) = \Theta(n).$

$\therefore T(n) = \Theta(n \log n)$

Avg. analysis

probability of split. $\uparrow$ $\frac{1}{n}$

$$C_n = n + \frac{1}{n}[(C_0 + C_{n-1}) + (C_1 + C_{n-2}) + \cdots + (C_{n-1} + C_0)]$$

$$= n + \frac{2}{n}(C_0 + \cdots + C_{n-1}).$$

$\Downarrow$

$nC_n = n + 2(C_0 + \cdots + C_{n-1})$.

$(n-1)C_{n-1} = (n-1)^2 + 2(C_0 + \cdots + C_{n-2})$.

$nC_n - (n-1)C_{n-1} = 2(n-1) + 2C_{n-1}$.

$$C_n = \frac{2n-1}{n} + \frac{(n+1)C_{n-1}}{n}$$

$$\frac{C_n}{n+1} = \frac{2}{n+1} - \frac{1}{n(n+1)} + \frac{C_{n-1}}{n} \le \frac{2}{n+1} + \frac{C_{n-1}}{n}$$

$$\frac{C_n}{n+1} \le \frac{2}{n+1} + \frac{C_{n-1}}{n}$$

$$\le \frac{C_{n-2}}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

$$\le \cdots \le \frac{C_1}{2} + 2\left(\frac{1}{n+1} + \frac{1}{n} + \cdots + \frac{1}{3}\right)$$

$$= \frac{C_1}{2} + 2\sum_{i=2}^{n} \frac{1}{i+1} \le 2\sum_{i=1}^{n} \frac{1}{n} \approx 2\log_e(n)$$

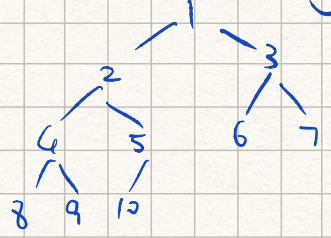
$$= 2\ln(n)$$

$$C_n \le 2(n+1)\frac{\log_2 n}{\log_2 e} = 1.39(n\log_2 n + \log_2 n) = O(n\log n)$$

Heapsort

Heap

Binary Heap is a nearly complete binary tree.



We can use an array to represent a heap

Swim | sink.

Swap element to $\frac{n}{2}$ until they are not smaller than the element above

Swap element downwards to $n*2$ / $n*2+1$ until it is smaller than $2n$ / $2n+1$.

popping/pushing to the heap = priority queue.

Heapsort.

1. Heap construction

Proceed from right to left using sink.

make subheaps.

if two children are heaps.

calling sink make a heap

starts from $\frac{n}{2}$.

At most $2n$ compares and n swaps $O(n)$

2. sort down.

remove top element from heap

re-construct i.e. $O(\lg n)$

repeat n times

$O(n \lg n)$

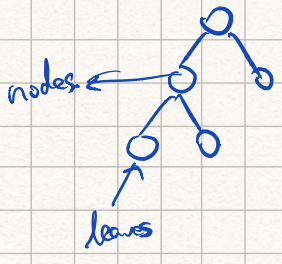
Heapsort – bad cache performance

– short implementation.

Heap. – use for priority queues

BST

Binary Trees



Interior nodes nodes with at least one child.

depth: distance (hwls) from root.

size: number of nodes in tree

every node has a key.
right
left.

BST min. go always to the left.

max – " – to right.

Successor keys. (the smallest key greater than this key)

if there is a right then return min(right(x))

otherwise. go up $y = \text{parent}(x)$

while y ≠ null and x = right(y)
x = y
y = parent(y)
return y

find a y where x is an element of its left subtree

BST Delete

if z has no children just remove it.

if z has one child, connect child to z parent.

if z has two children, find out minimum value in its right subtree. (Since it's a min value have at most one children (right))

replace root's key value with that value.

and delete the original node.

BST Worst case.

given a sorted array → chain

So we should keep a tree balanced.

2-3-4 Tree.

2-node contains 1 key (if not a leaf 2 children)

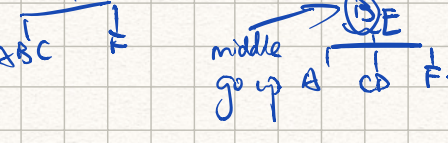
3-node contains 2 key 3.

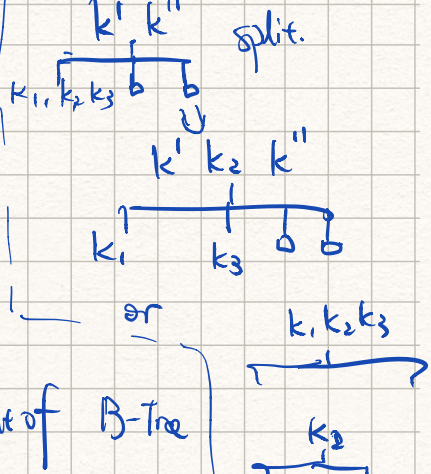
4-node 3 4

Insertion

→ 2-3-node node if a 3-4 node based on (i<j<k) (i<j)

→ 4-node ← insert D.





B-Tree.

Balanced BST

Each node is called a page.

Works well on disks / storage devices

Nodes may have many children.

number of disk access is required : the height of B-Tree

Order of a B-tree = n bound for keys per node/page. k_1 k_3

Each page has max $2n$ keys ≥ 3-4 tree $n=2$.
min n keys basic type of B-tree

Each page either a leaf or $m+1$ children
$\Uparrow$
number of keys on that page.

If there are too many keys in one page (node)
binary search., otherwise liner search.

k_1 k_2 — k_m
p_0 p_1 p_2 — p_m.

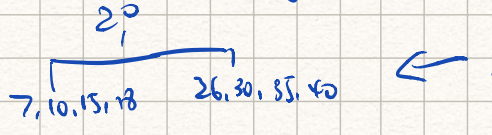
if $x<k_1 \rightarrow p_0$.
$k_i<x<k_{i+1} \rightarrow p_i$
$k_m<x \rightarrow p_m$

Each insertion is at leaf.

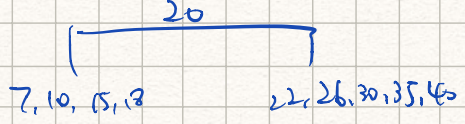
If page is not full its ok.

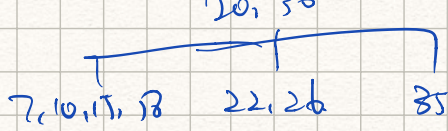
If page is full., there should be $2n+1$ keys.

keys → in order.
middle key take up to parent.
keys left of middle key to left.
right — — right



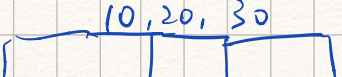
← 22.





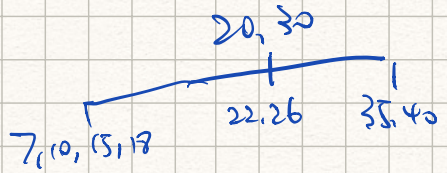
← 8

78, (10), 15, 18

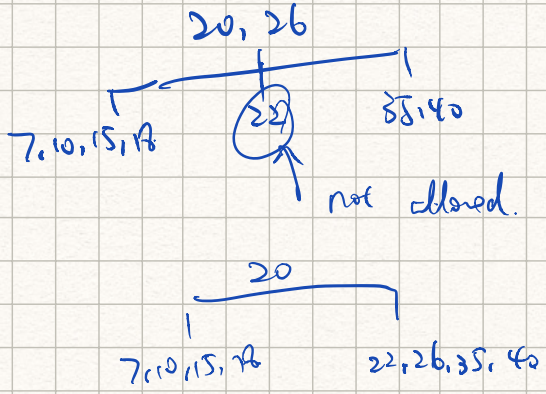


7,8 15,18 22,26 35,40

Deletion



delete 30, use 26, 35 from leaves



B-Tree scalable.

B-tree with 1000 keys per page.

branching factor of 1001.

lvl. 1. 1 page and 1000 keys.

lvl. 2. 1001 page and 1001000 keys

B-Tree trade offs

Smaller page → higher tree, more pages to consider when searching.

load/unload is quicker.

search in page is faster

larger page → lower tree, less pages

load/unload is slower.

search in page is slower.

B-Tree Worst case.

Number of page accessed when searching. = height = $\log_n N$ ← tot. keys in tree.

↑ page size.

Search per page $\simeq \log_2(n)$ if binary search (extra stack size)

worst case $\simeq$ height. × Search per page

$= \log_n N \cdot \log_2 N$

Hash tables

Many occassions where quick access to info is needed.

Compiler → symbol table.

Hashing offers good time-space compromise

If no space limit. / use memory address

If no time limit. use sequential search.

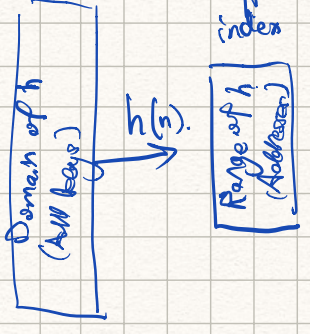
Hashing.

keys $k_1 \dots k_n$ and hash table size of m.

$1 \le h(k) \le m$.

$h(k)$ in table $1 \dots m$ as the page

↑ index to store stuff about k.



Hash Collision

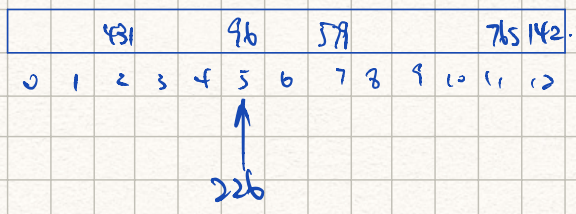
We may have $k_1 \neq k_2$ $h(k_1) = h(k_2)$

if m is sufficiently large
h is good choice. collision should be rare

Access in O(1) time.

Example. $h(x) = x \bmod 13$.

431 → 2 96 → 5 142 → 12 579 → 7.
765 → 11

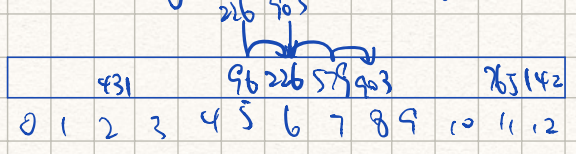


collision when adding 226

1. Linear Probing

Examine table in sequential order.

goes to right find 0 available



Add 903. (find 226)

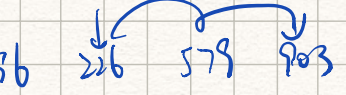
simple solution.

collision at s → s+1 → s+2.
if n is occupied. try 0

$x = (x+1) \bmod m$

Searching in liner probe.

search for 903.



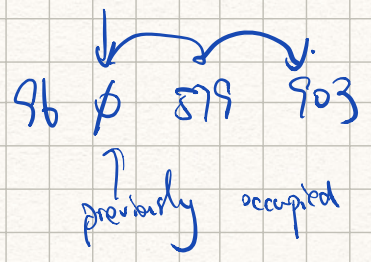
should be in slot 6 (might be probed)

go right / right find 903

delete 226.

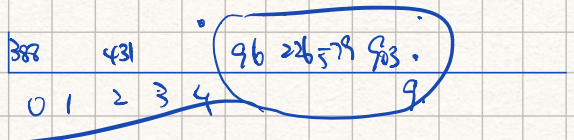
not set to null. (if null search for 903 would fail)

set to special empty char.



clustering

Providing more keys to hash tables → collisions are more likely.



9 is much higher in probability to be occupied.

→ primary clustering. since it occurs around original hash position

reduce primary clustering by

Modified linear prob.

quadratic probing.

double hashing.

Modified Linear Probing.

add more gap when probing.

reduce clustering → more even division.

Quadratic Probing.

false original hash index and adds successive of arbitrary quadratic

polynomial until a vacant is found.

+1 +3 +5 +7.

Chaining

An alternative to linear and quadratic probing

If a collision happens at S, give it a linked list.

All items hashed to the same location are in the list.

Reduced Collision / Regular search in linked list.

Expandable data structure.

Hash vs Binary tree.

Hash is preferred to binary search trees.

Simple
faster. for large table.

BST. no limit on insertions (dynamic).

B-Tree. Hash tables are.
fast to construct. / size limited / performance fluctuation

However.
B-Trees slow to construct (used when key type doesn't change?

Expandable predictable performance.

Recursion.

· Function calls and stack.

When make a function call

we also need to know where to return.

done by storing the return address on the stack.

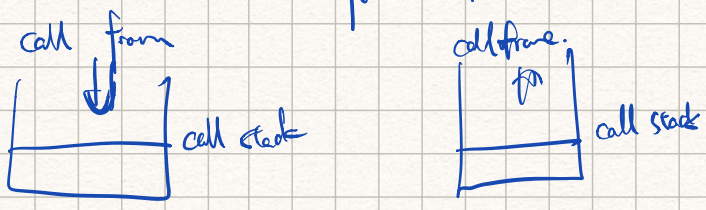
System stack is part of the run-time organisation

Every call adds a call frame / activation record.

return address and local variables

new function called.

function returns



Every recursion adds a call frame.

too deep ⇒ stack overflow.

Recursion leads to simpler code.

Example tower of hanoi

```
hanoi (lvl from mid to)
    if lvl = 1.
        print (lvl, from to).
    else
        hanoi (lvl-1. from, to, mid)
        hanoi (lvl, from mid to)
        hanoi (lvl-1, mid, from to)
```

Recursion 3 rules

1. Base case

   A case where things return.

2. Subproblems

   recursively descent to base case.

3. Recursive calls should not address overlapping subproblems

Tail Recursion.

f(x) has a last step recursive call f(y)

no need for recursion / set x=y loop around.

Dynamic Programming

1. Avoids repetition / solve each subproblem only once
2. Recursively define the value of an optimal solution
3. Compute value of an optimal bottom-up
4. Construct an optimal solution.

Example Sequence comparison (edit distance)

$a_1\ a_2 \cdots a_i$ two strings with length i, j respectively
$b_1 \cdots b_j$

$$d(a^i, b^j) = \min \begin{cases} d(a^{i-1}, b^j) \\ d(a^{i-1}, b^{j-1}) \\ d(a^i, b^{j-1}) \end{cases}$$

Matrix Multiplication with DP

if multiply 3 matrices

$A_1\ A_2\ A_3$

$(A_1\ A_2)\ A_3$ calculation effort is different from $A_1\ (A_2\ A_3)$

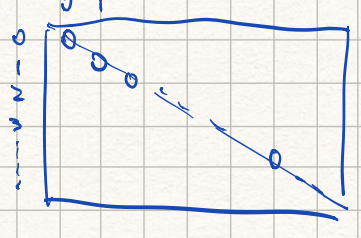
Given a chain of n matrices $(A_1 \cdots A_n)$

each with a dimension of $p_{i-1} \times p_i$

ABCD gets split to A BCD
AB CD
ABC D

sub problems that overlaps.

Such that base case



on digonal A_i itself (no multiplication needed)

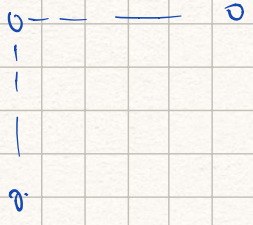
loop though length. loop though i 1 ~ size-len +1

j = length + i - 1

let k be the break
k from i to j-1
update matrix &(bracket info)
print bracket recursively downwards

Longest Common Subsequence.

Base Case



for any len[x][y]. if x.char = y.char
LCS len[x-1][y-1] +1
else max(len[x-1][y], len[x][y-1])

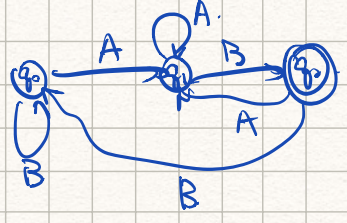
then len[x][y] is what we want.

back tracking.

if x[i-1] == y[j-1].
revert to i-- j--
lcs += x[i-1]
len_lcs--
otherwise see which one is bigger
len[i-1][j] or len[i][j-1]
revert accordingly

String matching with automaton.

Automata that matches AB



Build a finite automata that matches string.

record state / state transition.

1. Need to get next state from current state for all chars
2. For a state S, compute the longest prefix of P that is also P[0, S-1]|c.

Graph Algorithms

$G = (N, E)$ ← set of edges. order pairs (u, v)

↑ set of nodes / vertices

directed / undirected

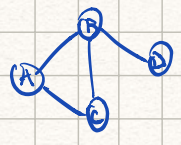
graph → adjacency matrix

2D array. good for dense graphs

adjacency lists (linked list.

A → B C

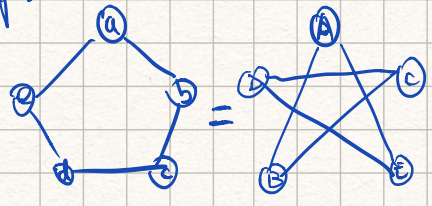
B → A C D



Matrix: removing/adding O(1) time.
query O(1)

more space (sparse graph)

isomorphic graphs



isomorphic.

a) same number of vertices
b) same — — edges
c) one-to-correspondence.

Directed Acyclic Graphs 有向无环图

有向树 ⊆ 有向无环图

DFS

traversal of graph → minimum spanning tree

detect cycles in a graph.

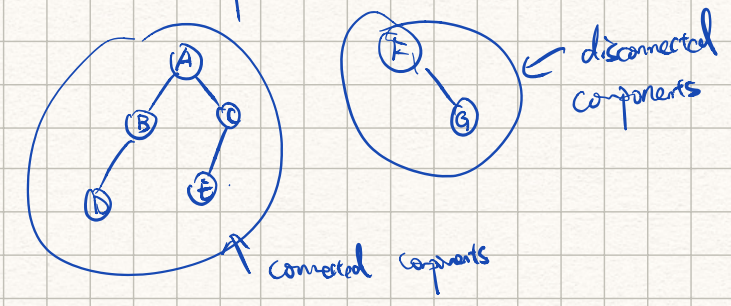
path finding

topological sorting

Strongly connected components of a graph.

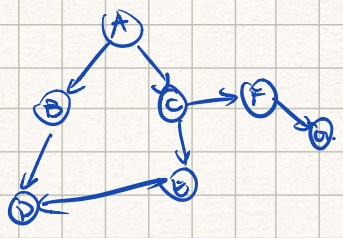
a directed graph. ∃ a path ∀ vertex$_1$, vertex$_2$.

Connected components.



Topological sort. on DAG.

if (u, v) then u is in front of v.



A, B, C, E, F, D, G. Find a node without incoming edges and remove

Remove A, Remove B, - - -.

PERT Chart

Program Evaluation and Review Technique

DAG - Example of PERT chart.

Edges - Time/effort required.

Critical path - sum of task times gives a minimum possible time

Start at one vertex with in-degree 0.

finish at —— —— out-degree 0

Shortest path. (in DAG)

dijkstra's algo

1. set shortest[v] = ∞ for all v except s
2. set Q to contain all vertices
3. while Q not empty.

find vertex u in Q with the lowest shortest value.

remove it from Q.

for each v adjacent to u

RELAX(u, v)

Relax (u, v). $O(n^2)$.

if shortest[u] + (u,v) < shortest v.
shortest[v] = shortest[u] + (u,v)
pred[v] = u

if we have negative weights

Bellman Ford algo

set shortest[v] to ∞ for each v except s.
shortest[s] = 0. pred[v] = NULL.
for i = 1 ~ n-1. O(n·e)
for each edge (i, v).
Relax (i, v)

Floyd Wershall. algo

any subpath of a shortpath is a shortest path.

FW algo: Assume vertices numbered 1~n.
shortest[u, v, x] is the weight of a shortest path from u to v. in which goes through 1~x.

1. Let shortest and pred be n × n × (n+1) arrays
2. For each u and v from 1 ~ n
A. set shortest[u, v, 0] = w_uv. ← is edge(u,v)'s length or ∞
B. if Edge(u,v) pred[u, v, 0] = u. or pred[u, v, 0] = NULL
3. for x = 1 ~ n
for u = 1 ~ n
for v = 1 ~ n.
if shortest[u, v, x-1] > shortest[u, x, x-1] + shortest[x, v, x-1]
set shortest[u, v, x] = shortest[u, x, x-1] + shortest[x, v, x-1]
pred[u, v, x] = pred[x, v, x-1]
otherwise shortest[u, v, x] = shortest[u, v, x-1].

pred[u,v,x] := pred[u,v,x-1]

Articulation point.

A vertex in a undirected graph is an articulation point. if removing it disconnects the graph.
(useful when designing reliable networks)
for disconnected graph. articulation point is a vertex removing which connected components

find articulation point

remove vertex one by one.

use DFS to see if removing v changes connected components increase

Carry out DFS in G.

Let T be the dfs tree.

prenum be the number assigned.

Postorder T.

lowest[v] is the min of

prenum[v].
prenum[w] (v,w) is in G but not T.
lowest[x] for child x in T.

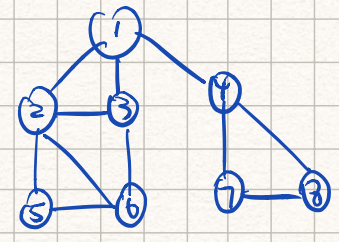
Articulation:

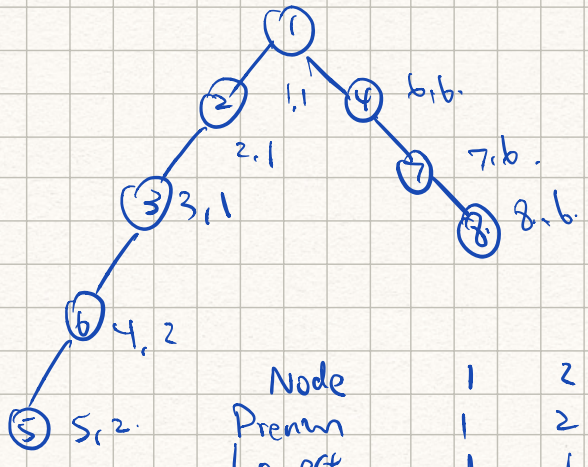
the root of T is an articulation iff it has more than one child.

node v is an articulation if

v has a child x lowest(x) < prenum[v]

Example





Node	1	2	3	4	5	6	7	8
Prenum	1	2	3	6	5	4	7	9
Lowest	1	1	1	6	2	2	6	6

A directed graph is strongly connected if a path exist for every pair $u \to v$ and $v \to u$.

We want to get strongly connected sub-components

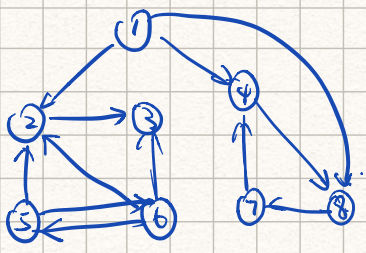
num=0 — global.

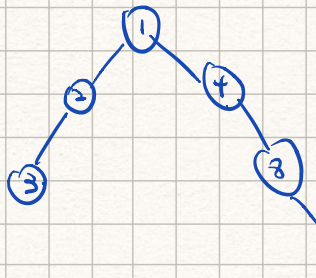
```
Search (G)
    for v ∈ N.
        mark[v] = not vis.
    for each v ∈ N.
        if mark[v] ≠ vis.
            DFS (v)

DFS (v)
    mark[v] = vis
    for each (v, w)
        if mark[w] ≠ not vis
            DFS(w)
```

num++.

postnum[i] = num

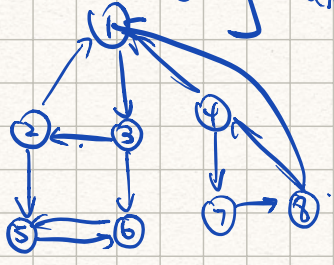
Example.



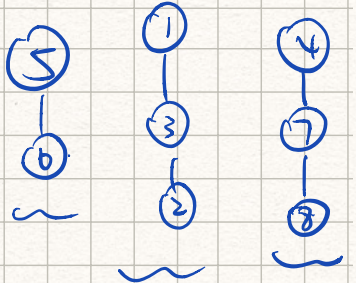


1	2	3	4	5	6	7	8
6	2	1	5	8	7	3	4

Construct G^* every direction is reversed.



do search from the highest postnum. 5.
Second highest ---.

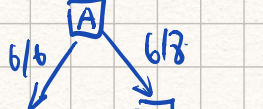


G, G^* have same strongly connected components
Highest postnum is the root of a search.
So starting search for G^* is forcing us to visit same nodes in reversing order.

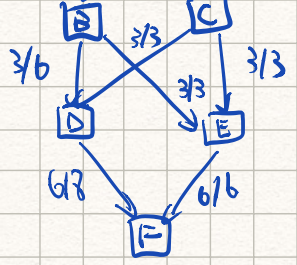
Applications of graphs

optimising network flow

Network flow.



a normal path is a seq. of forward traversals.



ACEF.

a super path is a seq of forward and backward traversals

ACDBEF

backwards

Ford-Fulkerson Method.

Part A.

for any normal path, if there is spare capacity by x in each of ocs, increase flow by x

Part B.

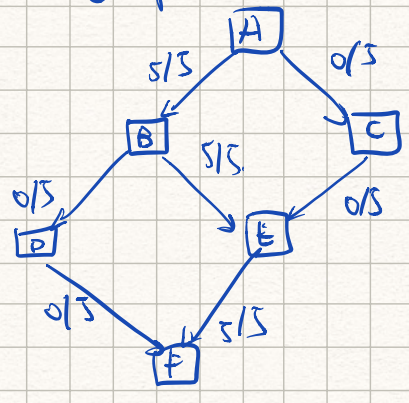
for any super path.

If forward traversal of an oc has at least x unused capacity

AND back traversal has a current flow x.

THEN flow increa by x by adding x to forward traversal AND subtracting x of backward ocs

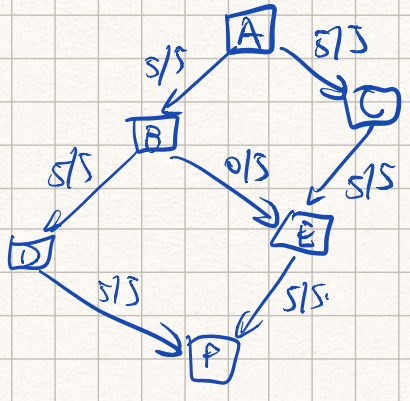
Example



ACEBDF

Then EB -5

AC, CE, BD, DF +5



Compression.

Compression algo $\begin{cases} \text{lossless} \\ \text{lossy} \end{cases}$ (image / video and audio)

Compression — reduces storage
reduce bandwidth

Run-length encoding

0---01--1 0---0 1---1

15 7 7 11.

15, 7, 7, 11.

encode as 15, 7, 7, 11.

11110111011111011

len 40 → 16 0.4 ⇒ 40% Compression ratio

4 bits each for the length.

if n bits, on average runs should be longer than n.

Huffman coding

Assume char occur at different freq.

Shorter codes to chars that occur more frequently

ABRACADABRA

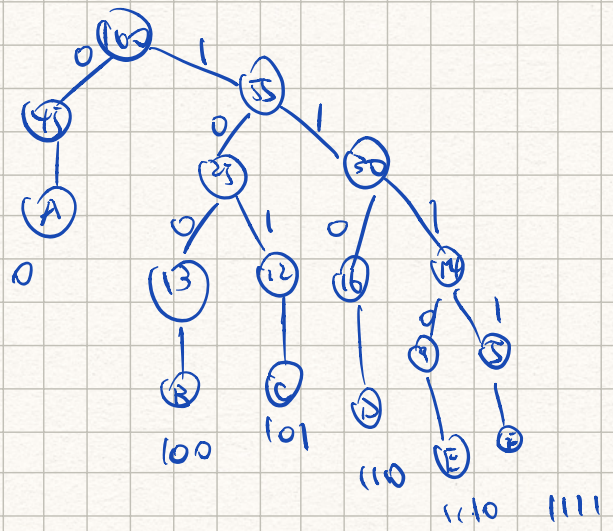
A(5) 0
B(2) 1
C(1) 01
D(1) 10
R(2) 00

01000101001

we need delimiters
otherwise not understandable

Therefore no character code can be prefix of another code.

	A	B	C	D	E	F
Freq	45	13	12	16	9	5
Fixed length	000	001	010	011	100	101
Varied length	0	101	100	111	1101	1100



1. Build an Encoding.
2. Encode the Bitstream
3. Transmit encoding Alex with encoded stream.
4. Read the code
5. decode to Bitstream

mapp objects with least freq.

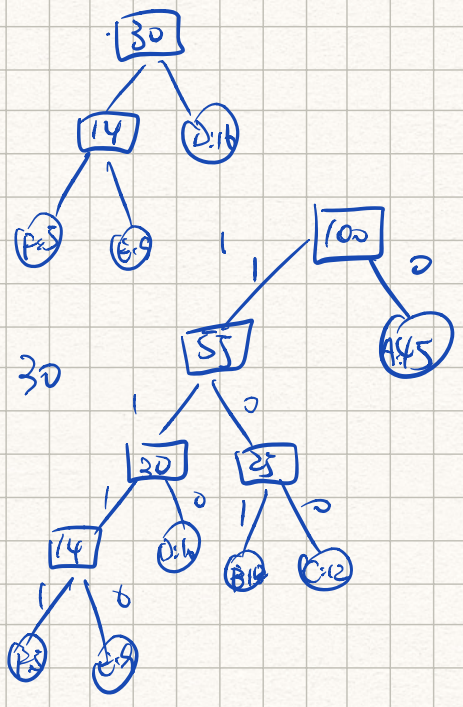
use min-que to store objects then pop, pop

A:45 F:5 E:9 C:12 B:13 D:16

F:5 E:9 ⇒

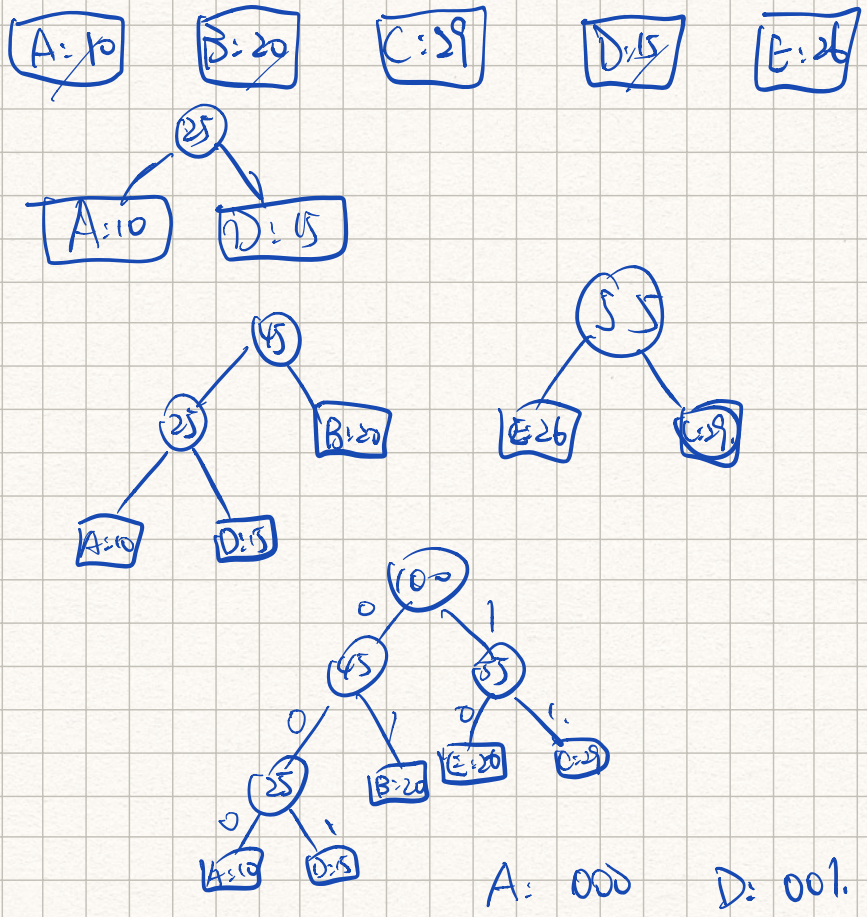
C:12 B:13

FE 14 D:16



BC:25 FED: 30

Exercise.



A: 000 D: 001.

B: 01 E: 01

C: 11.

Greedy algo ≥ Huffman encoding

↓

makes locally optimum choice

hope ⇒ global optimum solution
not the case sometimes!

Burrows Wheeler transformation