Jin Wei CS2040S Notes T20 R04

$\log(n!) = O(n \log n)$

Sterling approx: $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

unstable, then stable

sort	best	worst	avg	in-place	stable
bubble	n sorted	n^2 reverse	n^2	✓	✓
selection	–	–	n^2	✓	×
insertion	n	n^2	n^2	✓	✓
merge	–	–	nlogn	×	✓
quick	nlogn	n^2	nlogn	✓	×
counting	–	–	n	×	✓
radix	–	–	mn (m = # bits)	×	✓
heap	–	–	nlogn	✓	×
heapify	–	–	n	✓	×

Invariants

Bubble: biggest i sorted at end
Selection: smallest i sorted at front
Insertion: first i sorted (rest no change)
merge: groups of 2^k then 2^{k-1} then unsorted

Quick: $\boxed{\le p \mid p \mid > p}$ or $\boxed{< p \mid = p \mid ip \mid > p}$ ip = in-progress

1 : 9 ratio split still $O(n \log n)$

For 3-way partitioning:
$O(n \log k)$ num of distinct keys

$[1, 2, \cdots, n, 0]$ slow for bubble, fast for insert

Order Statistics / Quick Selection

Find k^{th} smallest in unsorted array.

1. Pick random pivot
2. Partition around pivot
3. Then pivot index is known.
4. Recurse on left or right if $i < k$ or $i > k$ respectively.

$O(n)$ for random pivot ~9:10 split
$O(n^2)$ worst case, only 1 elem removed each loop

Trees

number of
- insertion orders: $n!$
- shapes: $\sim 4^n$ by Catalan numbers

$n! > 4^n \Rightarrow$ by PHP, order of insertions do not result in unique shape.

height = max level of any vertex [null = -1, leaf = 0]

DFS
→ use stack

- Pre-order (n, l, r)
- In-order (l, n, r)
- Post-order (l, r, n)

$O(n)$, every node visited ≤ 1.

BFS
→ use queue

Root, adjacent, next level.

Balance / Height-balanced

Balanced $\Leftrightarrow h = O(\log n)$
$\left[h < \frac{\log n}{\log \phi} \Rightarrow h < 1.44 \log n\right]$

Node v is height-balanced
$\Leftrightarrow |v.l.h - v.r.h| \le 1$

Binary tree is height-balanced
$\Leftrightarrow$ every node is height-balanced

Height-balanced $\Rightarrow$ balanced
Balanced $\not\Rightarrow$ height-balanced

$n_h \ge 1 + n_{h-1} + n_{h-2}$ left & right children
$n_h \ge 2n_{h-2}$
$n_h \ge 2^{h/2}$
$n_h \le \sum_{i=1}^{h} 2^i = 2^{h+1} - 2$

BST

n.left $\le$ n < n.right
[recursively for grandchildren]

Search / Insert: $O(h)$

≤ 2 rotations [≤ 1 node]

Remove / Delete: $O(h)$

- if leaf, just remove
- if one child, swap w/ child & remove
- if two children, swap w/ succ & remove
  → succ guaranteed to have ≤ 1 child

$\le O(h)$ rotations (leaf to root)

Successor Finding: $O(h)$

if x not in tree, search(x) finds either pred or succ.
BUT, if x is max, root will be returned.

Rotations: $O(1)$

Rotations always have inversions
Right rotation requires left child
Left rotation requires right child

Heavy

left-heavy: v.l.h > v.r.h
right-heavy: v.l.h < v.r.h
v unbalanced & left-heavy
1. v.left balanced, right-rotate(v)
2. v.left left-heavy, right-rotate(v)
3. v.left right-heavy, left-rotate(v.left) & right-rotate(v)

(other side similar)

Tries
→ Strings & bits

Search: $O(L)$
Space: $O(nL)$ + overhead
→ possible to compress (unchain only if needed)

(a,b)-tree & B-tree

B-tree == (b, 2b)-tree
1. Every node has [a,b] children
2. Root has ≥ 2 children
3. All leaf at same level
4. Keys are sorted.
5. Bottom-up building
6. #keys = #child − 1

$O(h)$ but h very small so $O(1)$

$h = [\log_b n, \log_a n + 1]$

Dynamic Order Statistics

Store weight of subtree
Time complexity not affected
select & rank in $O(h)$

Augmenting Data Structures

→ use properties that only depend on subtree/subset to update easily

kd-tree

Alternate splitting x, y, z, ...

Search: $O(h)$

Construction: $O(n \log n)$

1. findMedian(points) based on x
2. Partition around median
3. Recurse with next dimension on both halves (now children)

Amortised Analysis

Operation has amortised cost of $T(n)$ if for every integer k, cost of k operations $\le kT(n)$

Interval Search Trees

Sorted by left endpoint

leftEnd, rightEnd, $\underbrace{\text{left, right}}_{\text{child}}$, maxEnd (subtree)

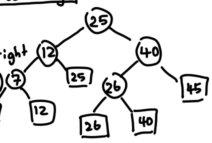
$O(h)$ search

If search goes left & no interval, guaranteed no interval in right.
If search goes right, guaranteed not in left.

$\forall$ intervals,
$O(k \log n)$ where k = result.size()
→ there exists a $O(k + \log n)$ solution but more complicated

Orthogonal Range Searching

leafs store value,
sorted from left → right
key = max(left leafs)



Split node is highest node lo < node.val $\le$ hi

$O(k + \log n)$ → k is result.size()
$O(n \log n)$ to build
$O(n)$ space complexity

2D / n-D Range Queries

1. Build a x-tree
2. For each node, build y-tree

$O(k + \log^2 n)$ query
$O(n \log n)$ to build
$O(n \log n)$ space complexity

2D

→ cannot maintain balance

Minimum / Maximum Coordinates $O(\sqrt{n})$

If balanced for 2D:
Even: $T(n) = T(n/2) + O(1)$
Odd: $T(n) = 2T(n/2) + O(1)$
Overall: $T(n) = 2T(n/4) + O(1)$
$T(n) = O(\sqrt{n})$

Hashing

Dictionary is ordered symbol table

If use array, need 2^n slots, n = max bit length (bad!)

Hash Collisions

2 distinct keys k_1 & k_2 collide if $h(k_1) = h(k_2)$
– always have collisions $\because U >> m$ by PHP

Solutions
1. Choose new hash function (naive) – will collide again
2. Chaining
3. Open addressing

Probability
← assuming equal chance

$X(i,j) = \begin{cases} 1 & \text{if item } i \text{ in bucket } j \\ 0 & \text{otherwise} \end{cases}$

$P(X(i,j) == 1) = \frac{1}{m}$

$E(X(i,j)) = \frac{1}{m}$

By linearity of expectation, for n items, $\frac{n}{m}$ items/bucket

$\log_b a = c \Leftrightarrow b^c = a$

Recurrence Relations

$T(n-1) + O(1)$	$O(n)$
$T(n-1) + O(n)$	$O(n^2)$
$T(n-1) + O(\log n)$	$O(n \log n)$
$T(n-1) + O(n^2)$	$O(n^3)$
exp: $2T(n-1) + O(1)$	$O(2^n)$
exp: $3T(n-1) + O(1)$	$O(3^n)$
exp: $2T(n-1) + O(n)$	$O(n2^n)$
$T(n-100) + O(1)$	$O(\frac{n}{100}) = O(n)$
$T(n-1) + O(\sqrt{n})$	$O(n\sqrt{n})$
$T(\sqrt{n}) + O(\log n)$	$O(\log n)$

$O(2^{2n^2 + 4n + 7}) = O(2^{2n^2 + 4n})$

Hash Chaining
← uniform hashing assumption

Put all colliding items into a linkedlist.

Insert: $O(1 + h)$
→ just insert at front
but if there are duplicates, have to sacrifice either search/insert

$\Theta(\log n / \log\log n)$ for n items
↑ proof beyond CS2040S

Search / Remove

Expected: $O(h + \frac{n}{m}) = O(h)$
Worst: $O(h + n)$

Space ← bad for cache ← constantly allocate new memory

Total: $O(m + n)$
Table size: $O(m)$
Linked list size: $O(n)$

Table Size

Optimal: $m = \Theta(n)$
too small: too many collisions
too big: wasted space

Growing & Shrinking

$O(m_1 + m_2 + n)$ $\because$ recompute hashes

		resize	n items	avg
grow	constant	$O(n)$	$O(n^2)$	$O(n)$
grow	doubling	$O(n)$	$O(n)$	$O(1)$
grow	square	$O(n^2)$	$O(n^2)$	$O(n)$ and waste space
shrink	halving	$O(n)$	$O(n)$	$O(1)$

Optimal: half when $\frac{3}{4}$ empty, double if full
→ amortised $O(1)$ insert & delete

Open Addressing – find another bucket!

hash fn returns SEQUENCE
stronger uniform hashing assumption
→ each n! sequence same probability
for linear probing: cluster size of $O(\log n)$

expected cost: $\le \frac{1}{1-\alpha}$ $(\alpha = \frac{n}{m} < 1)$ by GP
$n < m$

insert
$h(k,0) \to h(k,1) \to \cdots$
if empty, insert
if deleted, overwrite

search
probe until found or empty
skip deleted

delete
mark as deleted

double hashing
$h(k,i) = f(k) + i \times g(k) \bmod m$
if $g(k)$ relatively prime to m, $h(k,i)$ is good fn

tabulation hashing T is some 2D [x,N] table
for j in 0..N:
hash ^= T[key[j]][j]

quadratic probing
0, 1, 4, 9, 16, 25, 36, 49, ...

$\alpha < 0.5$, m is prime
→ can find empty slot

Calculus

AP: $\sum_{x=0}^{n} a + xd$
$S_n = \frac{n}{2}(2a + (n-1)d)$
$= \frac{n}{2}(a_1 + a_n)$

GP: $\sum_{x=0}^{n} ar^x$
$S_n = \frac{a_1(1-r^n)}{1-r}$, $r \ne 0$
$S_\infty = \frac{a}{1-r}$, $|r| < 1$ converges

$n^n >> n! >> x^n >> n^x >> \sqrt{n} >> \log^i n >> \log n >> \log\log n$

$T(n) = aT(n-b) + f(n)$, $a > 0$, $b > 0$, $f(n) = \Theta(n^k)$, $k \ge 0$
case $a = 1$: $\Theta(n^{k+1}) = \Theta(n \cdot f(n))$
case $a > 1$: $\Theta(n^k a^{n/b}) = \Theta(a^{n/b} \cdot f(n))$
case $a < 1$: $\Theta(n^k) = \Theta(f(n))$

$T(\frac{n}{2}) + O(1)$	$O(\log n)$	$T(\sqrt{n}) + O(\log n)$	$O(\log n)$
$2T(\frac{n}{2}) + O(1)$	$O(n)$	$\sum_{i=1}^{n} \frac{n}{i}$	$O(n \log n)$
$T(\frac{n}{2}) + O(n)$	$O(n)$	$T(\frac{n}{2}) + O(n^2)$	$O(n^2)$
$2T(\frac{n}{2}) + O(n)$	$O(n \log n)$		
$2T(\frac{n}{2}) + O(n \log n)$	$O(n \log^2 n)$	$2T(\frac{n}{2}) + O(n^2)$	$O(n^2)$

$T(n) = aT(\frac{n}{b}) + f(n)$, $a \ge 1$, $b > 1$, $f(n) = \Theta(n^k \log^p n)$
case $\log_b a > k$: $\Theta(n^{\log_b a})$
case $\log_b a = k$: $p > -1$: $\Theta(n^k \log^{p+1} n)$; $p = -1$: $\Theta(n^k \log\log n)$; $p < -1$: $\Theta(n^k)$
case $\log_b a < k$: $p \ge 0$: $\Theta(n^k \log^p n)$; $p < 0$: $\Theta(n^k)$

Graphs

(simple) path: set of edges connecting 2 nodes (no nodes repeated)

cycle: path w/ src == dest

bipartite {
- tree: no cycle ($E = V-1$)
- forest: ≥ 1 disconnected trees

sparse: $E = O(V)$

dense: $E = \Theta(V^2)$

	deg	diameter (longest shortest path)
star	n-1	2
clique/complete graph	n-1	1
line (is bipartite)	2	n-1
cycle (even is bipartite)	2	$\frac{n}{2}$ or $\frac{n-1}{2}$ (even / odd)
bipartite	–	n-1
AVL BST	2	log n

	space	find an adj.	enumerate adj.	isAdj
adj. list	O(V+E)	fast	fast	slow
adj. matrix	$O(V^2)$	slow	slow	fast

Graph Searching

BFS & DFS ← unweighted SSSP

$O(V+E)$ - list

$O(V^2)$ - matrix

- parent edges form tree (use defn of tree)

Topological Ordering → on DAG

1. sequential total ordering (NOT unique)
2. no bidirectional edge (antisymmetric)
3. no cycle

pre-order DFS on tree = topo order

reverse post-order DFS

same as DFS

kahn's algorithm

repeat:
1. $S = \{x$ has no incoming edge $\mid x \in V\}$
2. add all in S to result
3. remove edges connected to $x \in S$
4. remove all in S from V

O(V+E)

Δ triangle inequality

$\delta(S,C) \leq \delta(S,A) + \delta(A,C)$

keep reducing estimate & estimate ≥ actual

Bellman-Ford

→ do this V-1 times (can terminate early) [alt: queue-based variant exists]

after i iterations, i hop estimate on shortest path is correct!

O(EV)

Dijkstra's

CANNOT find longest unless non-negative after conversion

works for -ve edge {
proof of ∃ of correct way to relax
1. find shortest path tree
2. relax tree edges in BFS
3. relax the other edges in any order

→ after pop d, afterwards always ≥ d

min pq of nodes → relax all edges from v

→ use log to avoid multiplication

$\log(a_1 a_2 \cdots a_n) = \log a_1 + \log a_2 + \cdots$

$\log\left(\frac{1}{a_1 a_2 \cdots a_n}\right) = -\log a_1 + -\log a_2 + \cdots$

if $0 < x < 1$, $-\log x > 0$, can Dijkstra

after j iters, j closest is correct

PQ	insert	deleteMin	decreaseKey	Dijkstra
Array	1	V	1	$O(V^2)$
AVL	logV	logV	logV	O(E logV)
d-way	$d\log_d V$	$d\log_d V$	$\log_d V$	$O(E \log_{E/V} V)$
fib	1	logV	1	O(E+V logV)

SSSP

unweighted	BFS
no -ve cycle	Bellman-Ford
no -ve weight	Dijkstra
DAG	Topo-sort + relax
longest DAG	negate then SSSP_DAG

Strongly Connected Component (SCC)

∀ v,w ∈ SCC, path exists from v→w & w→v.

Graph of multiple SCC is acyclic if SCCs are connected acyclic-ly.

DAG has V SCC (∵ single node is SCC)

weights

∀ e ∈ E, c ∈ ℝ

e + c → shortest path CHANGE

e × c → shortest path SAME

Heap

sorted ⇒ heap, heap ⇏ sorted

[1, 2, …, n] root: 1, parent: $\lfloor i/2 \rfloor$, left: 2i, right: 2i+1

Full binary tree: every level is full (full ⇒ complete) (2^n array filled)

Complete ": 1. every level except leaf is full 2. leaf node far left (array no gaps)

get min/max k out of n: O(n log k)

max heap tree

1. root ≥ children (not nephew) ← local only
2. complete binary tree

- child ≤ root ≤ parent
- largest always root
- 2^{nd} largest always root's child
- $h = \lfloor \log n \rfloor$

bubble up/down raises/lowers level of invariant violation

insert : O(log n)

1. insert in far left of leaf
2. bubble up ← pointer or position

extractMax : O(log n)

1. swap root & last elem
2. remove last elem (the max)
3. bubble down (picks larger side)

(in general, raise priority to ∞, extractMax)

heapify : O(n)

1. start w/ complete tree
2. iterate from last to first index ← can skip leaf
   1. if heap (only check children), good!
   2. else bubble down

after i iterations, last i elem are heap

cost $= \sum_{j=1}^{n} \log\left(\frac{n}{j+1}+1\right) = n + \log\frac{n^n}{\cdots} + \cdots$

$= n + \log\frac{n^n}{n!} \simeq n + \log\frac{n^n}{\frac{n}{e}^n}$ by sterling $= \cdots = O(n)$

Union-Find

← Binomial Tree is an application of UF

$B_n = \text{root} + B_0 + B_1 + \cdots + B_{n-1} = B_{n-1} + B_{n-1}$

Quick-find

int [] id → connected ⟺ same id

find: O(1)

union: O(n)

Quick-Union

int[] parent pointer (NOT binary) → connected ⟺ same tree

find: O(h), union: O(h) } h = n possible (BAD!)

Weighted-Union

- pick taller tree as new root
- h only increase when size doubles

h = O(log n)

tree of height k has size ≥ 2^k → use induction

find/union: O(log n)

Path Compression

- set parent pointer of every node to its root
- do this only when traversing up

find/union: O(log n) [first op is O(n)]

WU + PC : very flat tree

m op on n items: [first op is O(log n)] O(n + m α(m,n))

find/union: O(α(m,n)) ← almost linear

MST

→ for weighted, undirected graphs

ST: acyclic subset of edges that connect all nodes

1. no cycles
2. split MST → 2 MSTs
3. cycle property (red): for every cycle, max edge NOT in MST.
4. cut property (blue): for every cut, min edge IS in MST.

(3, 4: proof by contradiction ↓ add 1 more edge, then remove)

∀ v ∈ V, min outgoing edge ⇒ in MST (∵ cut)

→ divide & conquer no work unless along median.

∃ in a MST (not all) ⟺ not max in any cycle

∵ suppose it is not max in cycle, can always avoid removing it

Prim's Algorithm

→ use min PQ

1. S = {start}
2. add min distance (∵ cut) & decrease Priority to edge weight
3. repeat 2 & 3

each node add/extract once

each edge decreaseKey

E ≥ V ∵ connected

O(E log V) if AVL pq

Kruskal's

1. sort edges small → big
2. iterate & add to result if no form cycle

- terminate after V-1 edges added
- use UF to union/find if form cycle

```
if (!uf.find(src, dest)) {
    result.add(e);
    uf.union(src, dest);
}
```

- each added edge crosses a cut

O(E log V) ∵ sorting > find/union O(E α(E))

Boruvka's

→ parallelisable

1. V connected components
2. repeat until 1 CC [O(log V) iterations]
   1. for each CC, take min outgoing edge
   2. add them
   3. merge these CC

overall: O(E log V)

Variants of MSTs

A. all edges same weight

→ run DFS/BFS (any ST is min)

O(E) if connected

B. all edge weights [1, 10]

Kruskal

1. use array of size 10 to sort (linked list)
2. 2^{nd} part same

sorting: O(E) (counting sort)

overall: O(α(E,E))

prim

1. insert/remove from PQ: O(V)
2. decreaseKey: O(E)

overall: O(V+E) = O(E)

C. DAG w/ 1 root (w/o incoming edges)

1. ∀ v ∈ V, add min incoming edge
2. done

∵ acyclic & V-1 edge ⇒ tree

D. directed graph

NP-hard

Stuff about MST

A. re-weighing edges is OK (adding/multiplying both OK) → only relative weights matter

B. MaxST: negate edges OR run kruskal from big → small

C. smallest max edge
1. get MST
2. BFS/DFS from A to B (will contain answer)

APSP Floyd-Warshall

S[v,w,P] be shortest path from v→w only passing through v∈P

$P_0 = \{\}$ $P_1 = \{1\}$ … $P_n = \{1, 2, \ldots, n\}$

$S[v,w,P_{i+1}] = \min(S[v,w,P_i],\ S[v,i+1,P_i] + S[i+1,w,P_i])$

$O(V^3)$ better than naive Dijkstra ($V^3 \log V$) when dense ∵ V × E logV

$O(V^2)$ space to store parent pointers to get path

```
for k in nodes:
    for each pair of nodes:
        use k as shortcut
```

DP

- optimal substructure → can construct problem from smaller problems
- overlapping subproblems → differ from d&c

recipe

1. identify optimal substructure
2. define subproblems ★
3. solve
4. done

longest increasing subsequence

prefix ver.

S[i] = max(all prev) + 1

$O(n^2)$

bin search ver.

- tails array storing smallest tail of lengths 0..n-1
- is non-decreasing → monotonic

```
for x in arr:
    bin search for position to add in tails
return tails.size_used
```

prize-collecting

1. if +ve weight cycles → ∞
2. negate edge weights & Bellman-Ford

lazy prize-collecting

→ in ≤ k steps: graph ver O(kE + kV)
1. model as DAG
2. make k copies
3. super node to every v ∈ G_1
4. DAG_SSSP from super node

→ in = k steps: DP $O(kV^2)$

P[v,k] = max prize at v in = k steps

= max { P(w,k-1) + weight(v,w) | w ∈ v.neighbour }

P[v,0] = 0

longest common subsequence

x(j) = x[0..j]

LCS(A(i), B(i)) = A(i) == B(i)

? LCS(A(i-1), B(i-1)) + 1

: max(LCS(A(i-1), B(i)), LCS(A(i), B(i-1)))

min vertex cover on tree

set of nodes that touch every edge

S[v,0] = size of VC of subtree, if v covered

S[v,1] = " , if v not covered

S[leaf, 0] = 0

S[leaf, 1] = 1

$S[v,0] = S[w_1,1] + S[w_2,1] + \cdots$ ← all child covered

$S[v,1] = 1 + \max(S[w_1,0], S[w_1,1]) + \cdots$

→ 2V subproblems O(V) time

misc.

- can combine u v if weight(u,v) = 0
- k copies of G for k states