CS 240 - S23
Data Structures and Data Management
Full Course Notes

Written from Lecture Slides

I wrote notes in class before midterms, and used lecture slides afterwards. Hope these help a little!

Josiah Plett

unspecified base: 2

CS 240 Spring 2023 MC 2034 (1)

(1) Initial Definitions + Overview

Problems (input + output)

- sorting: given numbers, put them in order.
- structured search: given data with keys, search by key.
- unstructured search: given text search by string.
- instance: one particular input to the problem.
- algorithm: step-by-step process to mod input (not same as program, a specific implementation)
- recursive: algorithm that uses itself.
- solving: finishes in finite time, returning correct answer.
- run time: the number of computing steps.
- n: the size of the input (use size(I))
- worst case: even for worst possible
- big O: ignore constant factors in the runtime bound

(2) Asymptotic Analysis

Algorithms will be presented using pseudo-code and analyzed using order notation.

- For a problem Π, we can have several algorithms.
- For an algorithm A solving Π, several programs.

Solving Problems

1. Design algorithm A that solves Π. (alg design)
2. Assess correctness and efficiency of A. (alg analysis)
3. If acceptable, implement A. (write program)

Efficiency
1. Running time
2. Auxiliary space
3. Size(Input)

Run Time
- (A) # of primitive operations
- (B) Growth rate (complexity)
- (C) idealized computer
- (D) RAM → constant time

Order Notation

★ $f(n) \in O(g(n))$ if there exists $c>0$ and $n_0>0$ s.t. $|f(n)| \le c|g(n)|$ for all $n \ge n_0$.

1. Build g out of $n \ge ? \to \#\# \le ? \to \# \le g(n)$
2. Build f out of $\#\# \le ? \to f \le \#$
3. Conclude $f \le \# \le g(n)$ for $n_0 = ?$

Ω-Notation (asymptotic lower bound)

★ $f(n) \in \Omega(g(n))$ if there exists $c>0$ and $n_0 \ge 0$ s.t. $c|g(n)| \le |f(n)|$ for all $n \ge n_0$.

Θ-Notation (asymptotic tight bound)

★ $c_1|g(n)| \le |f(n)| \le c_2|g(n)|$ for all $n \ge n_0$.

o (ω) - Notation (asymptotically strictly smaller)

★ if for all $c>0$, there exists $n_0 \ge 0$ s.t. $|f(n)| \le c|g(n)|$ for all $n \ge n_0$.

(3) Algebra, Techniques

Transitivity: $f(n) \in O(g(n)) \wedge g(n) \in O(h(n)) \to f(n) \in O(h(n))$

Maximums: $f(n) + g(n) \in \Theta(\max\{f(n), g(n)\})$ (if $f(n)>0$, $g(n)>0$)

Techniques

Limit

Suppose $f(n)>0$ & $g(n)>0$, $n \ge n_0$.

Assume $L = \lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists.

Then:

$$f(n) \in \begin{cases} o(g(n)) & \text{if } L=0 \\ \Theta(g(n)) & \text{if } 0<L<\infty \\ \omega(g(n)) & \text{if } L=\infty \end{cases}$$

Polynomials

If $f(n)$ is polynomial of degree $d \ge 0$, then $f(n) \in \Theta(n^d)$.

Sine

Just use first principles I guess.

Growth Rates

- If $f(n) \in \Theta(g(n))$, then growth rates of $f(n)$ & $g(n)$ are $\boxed{\text{same}}$.
- If $f(n) \in o(g(n))$, then growth rate of $f(n)$ is $\boxed{\text{less than}}$ $g(n)$.
- If $f(n) \in \omega(g(n))$, then growth rate of $f(n)$ is $\boxed{\text{greater than}}$ $g(n)$.

- $\Theta(1)$ - Constant
- $\Theta(\log n)$ - Logarithmic
- $\Theta(n)$ - Linear
- $\Theta(n \log n)$ - Linearithmic
- $\Theta(n \log^k n)$ - Quasi-linear
- $\Theta(n^2)$ - Quadratic
- $\Theta(n^3)$ - Cubic
- $\Theta(2^n)$ - Exponential

Relationships

$f(n) \in \Theta(g) \iff g(n) \in \Theta(f)$ etc.

(4) Algorithm Complexity

worst/best case: $T_A(n) = \max^{(\text{best})}\{T_A(I) : \text{Size}(I)=n\}$

average case: $T_A(n)^{\text{Avg}} = \frac{1}{|\{I : \text{Size}(I)=n\}|} \sum_{\{I:\text{Size}(I)=n\}} T_A(I)$

Recurrence Relations

exact recurrence: constants, not orders

sloppy recurrence: floor/ceiling removed

Merge sort example:

$$T(n) = \begin{cases} T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \underbrace{\Theta(n)}_{cn} & \text{if } n>1 \\ \underbrace{\Theta(1)}_{c} & \text{if } n=1 \end{cases}$$

(ABUSE)

$\boxed{n = 2^k}$ for analysis

RELATIONS

$T(n) = T(n/2) + \Theta(1)$	$T(n) \in \Theta(\log n)$
$T(n) = 2T(n/2) + \Theta(n)$	$T(n) \in \Theta(n \log n)$
$T(n) = 2T(n/2) + \Theta(\log n)$	$T(n) \in \Theta(n)$
$T(n) = T(cn) + \Theta(n)$ ($0<c<1$)	$T(n) \in \Theta(n)$
$T(n) = 2T(n/4) + \Theta(1)$	$T(n) \in \Theta(\sqrt{n})$
$T(n) = T(\sqrt{n}) + \Theta(\sqrt{n})$	$T(n) \in \Theta(\sqrt{n})$
$T(n) = T(\sqrt{n}) + \Theta(1)$	$T(n) \in \Theta(\log\log n)$

(4) Cont'd: Priority Queues

ADT: describes information and operations (interface)
realization: specifies data structure and algorithms
data structure: how information is stored
algorithms: how the operations are performed

STACK	QUEUE	PRIORITY QUEUE
• push()	• enqueue()	• insert() ← tagged w/ priority
• pop()	• dequeue()	• deleteMax()
• size()	• size()	"priority" AKA "key"
• isEmpty()	• isEmpty()	
• top()	• front()	
▫ LIFO	▫ FIFO	

Empty tree height: -1

(5) Priority Queue Realization

Unsorted Arrays: insert: $O(1)$, deleteMax: $O(n)$
PQ-sort → selection(n^2)

Sorted Arrays: insert: $O(n)$, deleteMax: $O(1)$
PQ-sort → insertion(n^2)

Binary Heap

Heap: (1) All levels of heap are full, except last is left-justified.
(2) For any node i, the key of the parent j is $j \geq i$.
("max-oriented binary heap")

Heap height: $\Theta(\log n)$
heap.insert: $O(\log n)$
heap.deleteMax: $O(\log n)$
PQ-sort → $(n \log n)$

Heaps in Arrays

- Store root in $A[0]$, and read left → right then top → down.

Left Child of i is $\boxed{2i+1}$
Right Child of i is $\boxed{2i+2}$
Parent of i is $\boxed{\lfloor \frac{i-1}{2} \rfloor}$

eg: Given n items (A), build heap.
Using fixups, we get $\Theta(n \log n)$
Using fixdowns, we get $\Theta(n)$

(6) Heapify, Heapsort

Heapify ← $O(n)$ — **Heapsort**

Heapify:
(1) Fix parent relative to children
bottom-to-top, left-to-right
(2) call fixdown

Heapsort:
- swap with last unused.
  - fix down.
- To sort in literal order.

Complexity

PQ-Sort w/ binary heaps $\boxed{O(n \log n)}$

(7) Runtime Analysis (Avg) & Randomized

$A = [12, 8, 0, 10, 4]$
$\pi = \langle 2, 4, 1, 3, 0 \rangle$
$\pi^{-1} = \langle 4, 2, 0, 3, 1 \rangle$

Only Permutations Matter

Avg Runtime

$$T^{avg}(n) = \frac{\sum_{I: size(I)=n} T(I)}{\# \text{ of size of } n} = \frac{\sum_{I \in I_N} T(I)}{|I_N|}$$

(8) Avg Runtime Randomized Algs

EG

$$T^{avg}(n) = \frac{1}{n!} \sum_{\pi \in \Pi_n} T(\pi) = \frac{1}{|\Pi_n|} \left(\sum_{\pi \in \Pi_n : \pi \text{ good}} T(\pi) + \sum_{\pi \in \Pi_n : \pi \text{ bad}} T(\pi) \right)$$

Average Runtime Strats: (1) Apply the def (2) Break up the summation (3) Insert (recursive) r.t. formulae (4) Simplify the sums

$T(I, R)$ is the runtime of a randomized algorithm A for an instance I and the sequence of random choices R.

Expected Running Time for I

$$T^{exp}(I) = E[T(I,R)] = \sum_R T(I,R) \cdot Pr(R)$$

"probability"

Expected Running Time for A

$$T^{exp}(n) := \max_{I \in I_n} T^{exp}(I)$$

(PDF) Module 3 Slides 26-35

QuickSelect

choose-pivot(A): return index p in A (semi-hard-coded)
partition(A, p): rearrange A and return pivot-index i s.t.
- the pivot-value v is in $A[i]$
- all items left of i are $\leq v$
- all items right of i are $\geq v$

A: | $\leq v$ | v | $\geq v$ |

(9) Randomized

QuickSelect Random

choose-pivot(A): random pivot
$T^{exp}(A) \in O(n)$

QuickSort (randomized)

$T^{worst}(n) \in \Omega(n^2)$
$T^{best}(n) \in O(n \log n)$
$T^{avg}(n) \in O(n \log n)$
$T^{exp}_{(rand)}(n) \in O(n \log n)$

Best Sort in practice

- Randomize the pivot
- While loop, not recursion
- Stop recursing when $n \leq 10$. Run InsertionSort then!
- If duplicates: $\boxed{\leq v} \boxed{=v} \boxed{\geq v}$
- Pass range of indices, not A.
- Keep explicit stack (no recursion!)

CS 240 (2)

(10) (11) Radix Sort

<u>radix-R</u>: {0, ..., R-1} are all the possible values in input.

<u>Stable</u>: equal keys return in the same order.

Bucket Sort
Everyone's Mom Sort
<u>Time</u>: $O(n+R)$
<u>Space</u>: $\Theta(n+R)$

Count Sort
Bucket sort but counting # of items in each bucket instead of moving.

LSD-Radix-Sort
1: <u>for</u> d ← m down to 1
<u>do</u> bucket-sort(A,d)

<u>Time</u>: $\Theta(m(n+R))$
<u>Space</u>: $\Theta(n+R)$

MSD-Radix Sort
Group by first digit, then next, recursively.
<u>Time</u>: $O(Rn)$
<u>Space</u>: $\Theta(n+R) + \underbrace{\Theta(m)}_{\text{stack}}$
m = # of digits

MSDRS(A, n, l=0, r=n-1, d=1)

```
■ l, r, range A[l, ..., r] to sort
■ d is digit we are sorting
· if (l<r) {
· bucket-sort(A, n, l, r, d)
· if (d < m) {
    // find sub-arrays w/same d^th digit and recurse
    int L = l
    while (L < r) {
        int q = L
        while (q<r & A[q+1]=A[L]) {      (only d^th digit)
            q++
        }
        MSDRS(A, n, L, q, d+1)
        L = q+1
    }
· }
```

Biased Search Requests

Key	A	B	C	D	E
access-frequency	2	8	1	10	5
access-probability	$\frac{2}{26}$	$\frac{8}{26}$	$\frac{1}{26}$	$\frac{10}{26}$	$\frac{5}{26}$

} Static

<u>Dynamic</u>: move forward if recently accessed?

Expected Access-Cost:
Expected # of comparisons needed to find random key.

"competitive analysis" is <u>online</u> (vs) <u>offline</u>.

Self-Adjusting Lists/Arrays
<u>Move-to-front heuristic (lists)</u>: exactly what you think.
<u>Move-to-front heuristic (arrays)</u>: same thing, but make the "front" of your unsorted array the end (where you insert).
<u>Transpose heuristic</u>: if accessed, move up by 1.

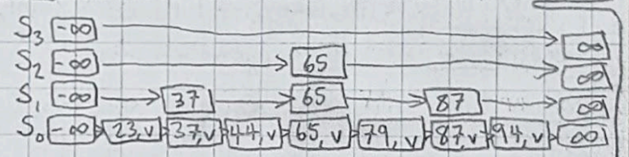
(12) (13) (14) (15) Stuff

BSTs. → insert, delete
AVLs. → restructure

Randomized BST
· <u>Expected</u> height: $O(\log n)$
↳ as long as no delete();

★ At this point I totally stopped going to lectures :)

Skip Lists
DEF A heirarchy S of ordered lists S_i (aka "levels" "layers") s.t.:
1. S_i contains keys $-\infty$ & ∞, "sentinels"
2. S_0 contains key-value pairs of S in non-decreasing order.
3. Each list is subsequence of prev: $S_0 \supseteq S_1 \supseteq \dots \supseteq S_h$
4. S_h contains only the sentinels.



getPredecessors(k)
eg to get 79, we do:
$S_3[0] \to S_2[0] \to S_2[1] \to S_1[2] \to S_1[3]$
$79 = S_0[5] \leftarrow S_0[4]$
result: $\{S_3[0], S_2[1], S_1[2], S_0[4]\}$

search(k)
basically getPredecessors(k), then check ∃.

insert(k, v)
insert, then <u>randomly</u> choose stack height.

delete(k)
remove it from all S_i (after finding it)

Special-Key Dictionaries
<u>Interpolation Search</u>: faster than binary search when values are numerical!
↳ Guess the approximate location! Instead of (for k=100) $\frac{110}{40} > 75$, do $\frac{110}{40} > \sim 100$

<u>Lemma 6.2</u>: If the keys are uniformly distributed, interpolation search is $\Theta(\log \log n)$ in average case

more on next page

Dictionary for Words

\$ → end-of-word character.

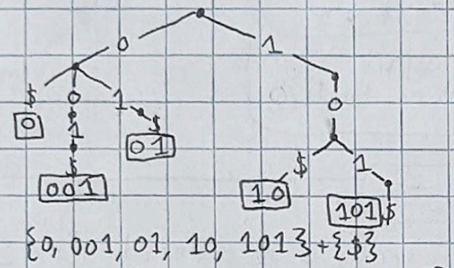
strcmp(w_1, w_2) → $\{-1, 0, +1\}$

Binary Tries

We assume:
- elements are $\Sigma^* = \{0,1\}^*$
- no word is a prefix of another.

Second is true if we attach \$, or all words are the same length.

Binary Trie → Ternary Tree



$\{0, 001, 01, 10, 101\} + \{\$\}$

All operations have runtime $O(|w|)$.

Variations

① No leaf Labels

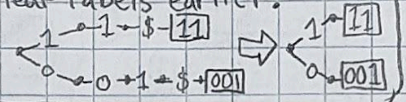
Don't store words at end.

② Allow Proper Prefixes

Tree becomes binary, holding value of whether in dictionary.

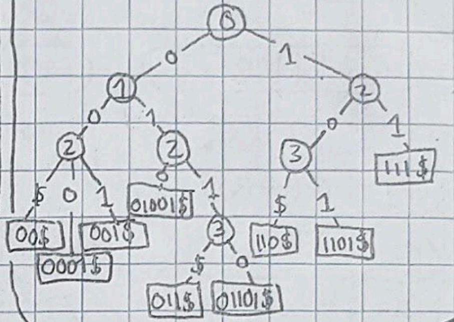
③ Pruned tries

If more traversals won't be needed, just store the leaf labels earlier!



Compressed Tries

Like pruned trees on steroids. Each node stores the lower limit on the length of its "sub-words."



HASHING

$h(k) = k \bmod M$

Direct Addressing: The key k goes directly to the map.
(If k is known bounded $0 \le k \le M$, everything is $\Theta(1)$.)
→ Can't be used if: • keys aren't ints • ? smth else

Hash Collision: when a new key maps to a location already taken.

Solving Hash Collisions

① **Chaining:** each slot is a bucket. Use unsorted linked list w/MTF. (move to front)
- → average bucket size is $\frac{n}{M} = \alpha$ (load factor). Insert is still $\Theta(1)$ but search and delete are $\Theta(1+\alpha)$.
- → Re-hashing: when α gets too big, double the hash table size.

② **Probe Sequencing:** Take an extra parameter for probing.
Search & insert are normal but delete needs to not leave holes,
- → So: delete: → move later items back; → mark slot as deleted, not NIL.
- → Linear Probing: $h(k,i) = (h(k)+i) \bmod M$

③ **Double Hashing:** open addressing (meaning look for new address when collide) with a second hash function. $h_0(k)$ & $h_1(k)$.
- → We need $h_1(k) \neq 0$, and $h_1(k)$ relative prime w/M for all k.
- → $h(k,i) = (h_0(k) + i \cdot h_1(k)) \bmod M$

④ **Cuckoo Hashing:** take h_0, h_1, use T_0, T_1 tables. Item with key k can only be at $T_0[h_0(k)]$ or $T_1[h_1(k)]$, so we do booting if there's a collision! Rehash everything if too many boots.
- → load factor $\alpha = n/(|T_0|+|T_1|)$.
- → for insertion expected runtime to be constant we need $\alpha < \frac{1}{2}$.

Hashing Assumptions

- We know the universe of keys
- Have hash function $U \to \{0,1,\dots,M-1\}$
- Dictionary is in array T, $|T| = M$.

Uniform Hashing Assumption:
Any hash function is equally likely.
Then: $P(h(k)=i) = \frac{1}{M}$ for any k and i.

Independent hash functions:
$h_1(k)$ and $h_2(k)$ should be independent otherwise, StUpId.

Multiplication Method:
$h(k) = \lfloor M \cdot (kA - \lfloor kA \rfloor) \rfloor$, $0 < A < 1$

Carter-Wegman's Universal Hashing

All keys $\{0,1,\dots,p-1\}$ for a (big) prime p.
Choose $M < p$. Choose random a, b, $a \neq b$, in key range.

$h(k) = ((ak+b) \bmod p) \bmod M$

Keys collide with probability at most $\frac{1}{M}$.

CS 240 (3)

Range Searches

S = output size
I = interval (x, x')

- If unsorted, runtime $\Omega(n)$ since we have to search each individually.

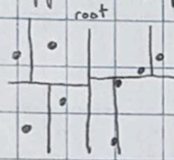
RangeSearch implementation:
↳ binary search to find the two limits, i, i'
↳ report $A[i+1, \dots, i'-1]$ and $A[i]$ $A[i']$ if in range!

Multidimensional (d) Range Search

Given a query rectangle, return all points in it!
Assumption: No two x- or y-coors are the same.

Kd-Trees

Suppose n points... use splitting lines at halfway:



- assume first split is vertical.
- height is $O(\log(n))$
- maintain height by occasionally rebuilding.
- Rangesearch complexity: $O(s + Q(n))$ — s: output size, $Q(n)$: # of "boundary" nodes visited

Range Tree

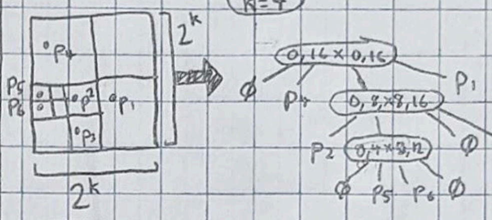
Literally just a balanced binary search tree for x-coordinates that stores a balanced binary search tree for y-coordinates.
Each node in the main tree is a point and also another tree, that holds the same children.
Space: $O(n \log n)$

Quad Trees

Bounding Box: $R = [0, 2^k) \times [0, 2^k)$.

2	1
3	4

↳ make by picking smallest k that works w/data.



search, insert, & delete are intuitive

Quadtree Range Search

Get given a rectangle, do expected stuff!
spread factor: $\beta(S) = \dfrac{\text{sidelength of } R(\text{egion})}{\text{min distance between points}}$
height: $h = O(\log(\beta(S)))$
building complexity: $O(hn)$

Module (9) Pattern Matching

Definitions

Pattern Matching: find a string (pattern) in a lot of text. — returns first occurrance
$T[0 \dots n-1]$: The text (haystack)
$P[0 \dots m-1]$: The pattern (needle)
If P doesn't occur in T, return FAIL.
Substring: Exactly what you think. eg. $T[4 \dots 7]$
Prefix / Suffix: substring at start / end.
Guess or Shift: position i s.t P might start at $T[i]$.
Check: position j, $0 \leq j < m$, to check $T[i+j] = P[j]$

Use DFA to read the letters linearly

Algorithms (for pattern matching)

Karp-Rabin: Compute fingerprint (hashing) for each guess.
↳ if fingerprint different from P's, no need to check.
↳ key insight: update fingerprints in constant time.
This allows runtime to just be $O(m+n)$, worst: $O(mn)$.

Boyer-Moore: Better pattern matching on english.
(1) Reverse order (where) (2) Bad char jumps
(3) Good suffix jumps ((2) & (3) are both "move as far as can").
Implement w/a last-occurrence array. Example: — $O(m+\Sigma)$ construction

0	1	2	3	4
p	a	p	e	r

→

char	p	a	e	r	others
$L[\cdot]$	2	1	3	4	-1

(2) ★ To shift after guess at i_{old}, $\boxed{i_{new} = i_{old} + (m-1) - \min\{L[\text{char}], j_{old} - 1\}}$

(3) Good suffix is like "good" character but in groups (no details).

some real aminals lol

these what you gonna look like if you keep not showering during exam season
take care of yourself