CS 21 LECTURE #8

Program #4:

high: 100

mean: 88

median: 100

Midterm:

high: 148/160 (x2)

Mean: 121/160

median: 128/160

Program #5 moved back 24 hrs!

Issues with Median Values:

i^{th} median value in n values

$i = 0$

$i = size - 1$ } Min, Max value → $O(n)$ (Min or Max)

hack

Min & Max:

One loop vs two loops

loop:

compare A[i] vs max

compare A[i] vs min } $2n$ comparisons. ⇒ too many ⇒ $\frac{3n}{2}$ compares.

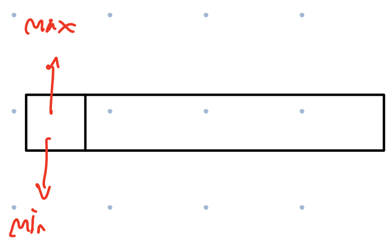
Faster way?

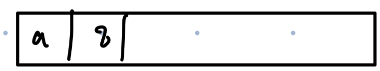
take elements two at a time

- compare against eachother

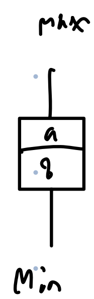
  compare bigger to max

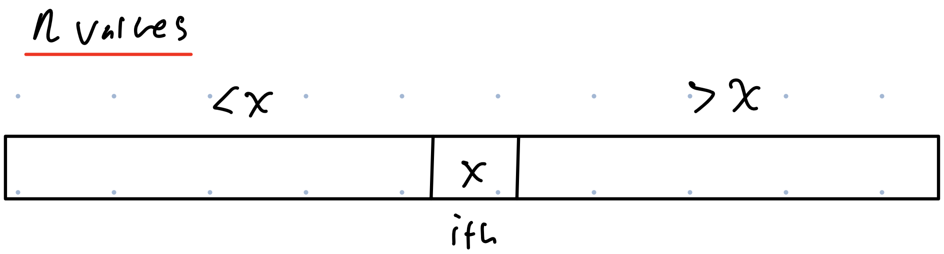
  compare smaller to Min





a compare b Max





The Selection Problem...

- Related to the sorting problem...

Finds the nth largest item...

Randomized_select → $O(n)$ expected
→ $O(n^2)$ worst case

- Based on quicksort
- Randomized selection of pivot

(probably good enough... dont really need MO3)

left, right, find ith smallest

```
Randomized_Select(A, p, r, i) {
    if (p==r) {
        return A[p];
    }

    q = Randomize_partition(A,p,r);
    k = q-p+1;

    if (i==k) {
        return A[q];
    }
    if (i<k) {
```

← Partition w/ random selection (same as used in quicksort)

```
        return Randomized_select(A,p,r,i);
    }

    return Randomize_select(A,p,r, i-k);

}
```

Data Compression.

Huffman Codes — Whats the optimal way to encode things & how do we find it???

Data representation:

fixed length encoding vs Variable length encoding

fixed length encoding: for example, all ascii values are made up of 8 bits!

Variable length encoding: Think of morse code in which frequent letters have short codes & less common letters have longer codes!

Huffman Code Algorithm = A "greedy algorithm" constructs optimal prefix code.

"greedy algorithm" → obtains an optimal solution by making a series of choices at each point in the algorithm. The choice that seems best is chosen.

Optimal Substructure.

1 ~ 7 in binary, impossible 7 so both these codes CAN NOT BE

111 ~ } to distinguish   ξ CHOSEN

Prefix codes:

└ No code is the prefix of another code.

Process:

- Build a tree
  0 = left, 1 = right
- Optimal = full binary tree
- Alphabet = has size m should have m leaves and $m-1$ internal nodes
- Assume S is a set of m characters each w/ a frequency $f(x)$.
- Use a priority queue queued on frequency (least)

The Algorithm:

- Store characters in priority queue (buildheap = $O(n)$)
- $O(n)$ • repeat $n-1$ times:
  - Make new node t
  - $O(\log(n))$ • left = extractMin()
  - $O(\log(n))$ • right = extractMin()
  - Attach left & right to t

• Attach left & right to t

• $f(t) = f(left) + f(right)$

$O(\log(n))$ • insert(t)

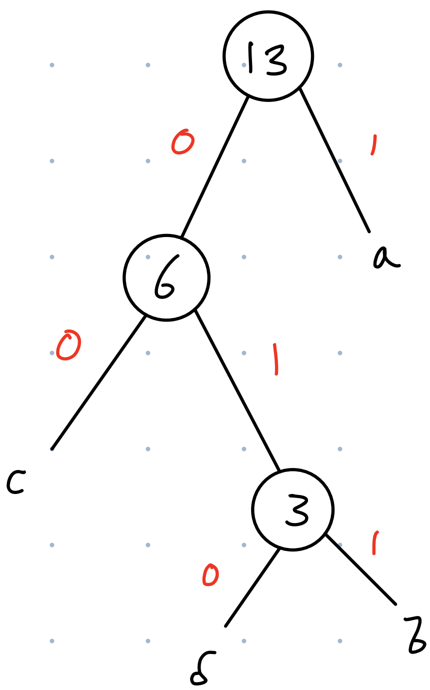
abaccaadabcaa ⇒ a = 7 b = 2 c = 3 d = 1

fixed length:

00 a
01 b
10 c
11 d

$13 \cdot 2 = 26$ bits

what algorithm does:



Now:

a 1
b 011
c 00
d 010

you can now write all of these codes in sequence with NO AMBIGUITY. Awesome!

EX

1	1	1	011	00	010
a	a	a	b	c	d

$7 \cdot 1 + 2 \cdot 3 + 3 \cdot 2 + 1 \cdot 3$

$= 22$ bits

SMALLER NOW!

Complexity

$O(1)$ $O(n \log n)$ $O(n)$ $O(\log n)$ $O(n^2)$ $O(n^3)$

can all problems be solved in polynomial time? NO

Halting Problem:

"Given a program & an input to that program, determine if that program will eventually stop with that input"

Something not solvable by ANY computer Regardless of the time it's given:

Claim: solves halting problem

Step 1: bool doesItStop (program p, input i) {

```
    if ( code here ) {
        return true;
    else
        return false;
    }
}
```

Step 2:

```
bool StopsOnSelf (program p) {
    return doesItStop (p, p);
}
```

Step 3:

```
bool wTF (program p) {
    if ( stopsOnSelf (p) ) {
        while (true);
        return false;
    }
    return true;
}
```

Step 4:

run wTF on itself

2 possibilities:

(1.) Runs forever
(2.) Stops & returns true

Both are contradicting to eachother. Therefore this program can never work.

$\boxed{\text{Unsolvable Problems}}$ (even with infinite time)

e.g. halting problem

called "NP"

$(P \neq NP)$

"P" vs "NP"

P → Polynomial

NP → Non-deterministic Polynomial

- Problems whose status is unknown
- AKA problems for which no polynomial time algorithm has been found.
- BUT if "someone hands you a solution" you can verify correctness in polynomial time

Then how do we solve it?

- approximation

The Bin Packing Problem:

Given a set of n objects with sizes between 0 & 1, pack all of the objects in the fewest number of unit sized bins (1.0)

Optimal solution is NP-Hard

so not practical / done / to able

BUT approximate solutions are possible

First Fit $O(n \log(n))$

Solution is $\sim \frac{17}{10}$ (about 2:1 of best solution)

Takes each obj & places it in the first bin that can accomodate it. If no bin can accomodate it, you get a new bin.