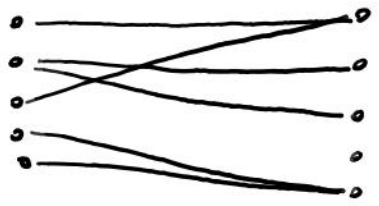
INTRODUCTION TO ALGORITHMS

LECTURE 1: THIS COURSE IS ABOUT COMPUTATIONAL PROBLEMS. SOLVING THEM, COMMUNICATING SOLUTION, FINDING A SHARED LANGUAGE FOR CORRECTNESS & EFFICIENCY

HOW DO WE PROVE WHAT WE'RE DOING IS NOT JUST CORRECT BUT THE BEST APPROACH

WHAT'S A PROBLEM.? WHAT'S AN ALGORITHM

① A PROBLEM IS A BINARY RELATION FROM INPUT TO (CORRECT) OUTPUT



WHEN DEFINING A PROBLEM WE DON'T DEFINE THE WHOLE MAP. WE ESTABLISH A TEST FOR CORRECTNESS - A PREDICATE.

WE'RE LOOKING FOR GENERAL PROBLEMS WITH ARBITRARILY-SIZED INPUTS. AND WE WANT ALGORITHMS TO SOLVE THESE PROBLEMS.

ALGORITHMS - FIXED SIZED PROCEDURE THAT GIVEN A PROBLEM INPUT MAPS TO CORRECT OUTPUT. AS A FUNCTION: $f: I \to O$

FOR A GROUP OF STUDENTS, HOW DO WE FIND OUT IF THEY SHARE A BIRTHDAY?

- maintain record
- interview students in some order
  - check if birthday in record
    - if true, return pair
    - else add new student to record
- return false

HOW DO WE KNOW THIS IS CORRECT? HOW DO WE KNOW IT'S GOOD? RECURSION / INDUCTION.

INDUCTIVE PROOF: SHOW THAT AT CONCLUSION, RETURNS PAIR OR FALSE CORRECTLY

INDUCTIVE HYPOTHESIS: IF FIRST k STUDENTS CONTAIN MATCH; ALGORITHM RETURNS A MATCH BEFORE INTERVIEWING STUDENT $k+1$ ← INDUCT ON k

BASE CASE: $k=0$ ✓

ASSUME HYPOTHESIS TRUE FOR $k=k'$ { IF MATCH $\in k$; ALREADY RETURNED / ELSE IF MATCH $\in k'+1$ ALG $k'+1$ CHECKS BY BRUTE FORCE

NEEDS TERMINATION CRITERIA TOO!

NOW WE NEED TO LOOK AT EFFICIENCY

HOW FAST DOES A PARTICULAR ALGORITHM RUN COMPARED TO OTHER ALGORITHMS? NEEDS TO BE SUBSTRATE-AGNOSTIC.

LET'S ASSUME EACH COMPUTATIONAL STEP TAKES A FIXED AMMOUNT OF TIME. THEN WE CAN COUNT STEPS FOR RUNNING THE ALGORITHM WITH A GIVEN INPUT.

DON'T MEASURE TIME, COUNT OPERATIONS - ASYMPTOTIC ANALYSIS. EXPECT PERFORMANCE TO DEPEND ON SIZE OF INPUT (n)

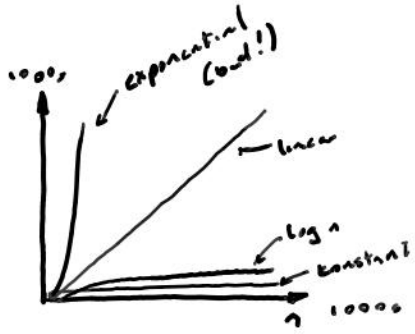
BIG O NOTATION - CORRESPONDS TO UPPER BOUND?

Ω " " " LOWER "

Θ " - " TO BOTH!

SOME COMMON FUNCTIONS TO RELATE AN ALGORITHM'S INPUT SIZE TO PERFORMANCE.

- $O(1)$ CONSTANT TIME
- $O(\lg n)$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$
- $O(n^c)$ POLYNOMIAL
- $O(2^n)$ EXPONENTIAL (PRETTY BAD)



HOW DO WE MEASURE THESE THINGS? DEFINE A MODEL OF COMPUTATION FOR WHAT OUR MACHINE IS ALLOWED TO DO IN FIXED TIME

WORD-RAM

~ RANDOM MEMORY ACCESS IN CONSTANT TIME

WE HAVE AN ADDRESS FOR EVERY 8 BITS OF MEMORY (BYTES); CPU WHEN GIVEN THIS ADDRESS CAN PULL A CHUNK OF MEMORY STARTING AT THAT BYTE & PERFORM OPERATIONS. THIS IS A "WORD"! CURRENT WORD SIZE IS 64 bits. WORD SIZE DETERMINES HOW MUCH MEMORY YOU CAN ACCESS! $2^{32} \rightarrow$ 4GB. $2^{64} \rightarrow$ 20 exabytes

WHAT ARE THE OPERATIONS A CPU CAN PERFORM?

- integer arithmetic
- logical operations
- read/write to memories
- bitwise operations

} CONSTANT TIME

CPUs OPERATE ON SET # OF WORDS AT A TIME. GENERALLY TWO WORDS IN MEMORY W/ OPERATIONS PRODUCING A THIRD. READING INFORMATION TAKES LINEAR TIME - WHICH MEANS THE SHAPE OF OUR DATA MATTERS

DATA STRUCTURES

STORE LARGE AMMOUNT OF DATA & SUPPORT VARIOUS OPERATIONS. HOW DO WE DO THIS· FAST!

STRATEGIES FOR SOLVING PROBLEM:

- REDUCE TO PREVIOUS PROBLEM (① (SEARCH/SORT/SHORTEST PATH) ②)
- DESIGN RECURSIVE ALGORITHM
  - BRUTE FORCE
  - DECREASE & CONQUER
  - DIVIDE & CONQUER
  - DYNAMIC PROGRAMMING
  - GREEDY / INCREMENTAL

① DATA STRUCTURES

② GRAPHS

LECTURE 2: DATA STRUCTURES & DYNAMIC ARRAYS

- SEQUENCE INTERFACE & DATA STRUCTURES
  - LINKED LIST
  - DYNAMIC ARRAYS
  - AMORTIZATION
- SET INTERFACE

INTERFACE (API/ADT) VS. DATA STRUCTURE

- specification	- representation
- what data you can store	- how to store data
- what operations are supported & what they mean	- algorithms to support operations
- problem statement	- solution

TWO MAIN INTERFACES (AND SPECIAL CASES)

- SET
- SEQUENCE

} WE WANT TO STORE N ITEMS. WE CARE ABOUT VALUE & WHERE THEY ARE

[5, 2, 9, 7]

TWO MAIN APPROACHES

- ARRAYS
- POINTERS

STATIC SEQUENCE INTERFACE MAINTAINS...

ITEMS $x_0, x_1, \ldots, x_{n-1}$ SUBJECT TO:

$O(n)$ → - build() : make new data store for items in x

$O(1)$ → - len() : return n

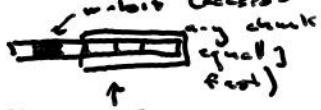
$O(1)$ → DYNAMIC ACCESS TO ANY ITEM
- get_at(i) : return x_i (index i)
- set_at(i, y) : set x_i to y

$O(n)$ → - iter_seq() : output $x_0, x_1, \ldots, x_{n-1}$ in sequence order

SOLUTION TO THIS INTERFACE PROBLEM: STATIC ARRAY

KEY: word RAM model

Memory: array of w-bit words



"array": consecutive chunk of memory

=> array[i] = memory[address(array) + i]

STORING ARRAYS STATICALLY ASSURES ACCESS TO GET/SET OPERATIONS IN CONSTANT TIME $O(1)$

MEMORY ALLOCATION MODEL: allocate array of size n in $\Theta(n)$ time

=> space = $O(\text{time})$

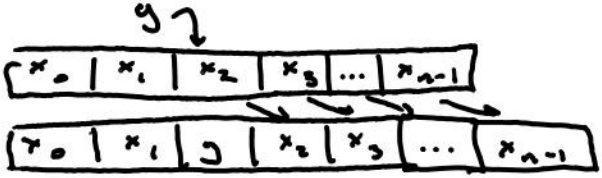
ALL THIS ASSUMES $w \geq \log(n)$. word size has to grow with n! because it needs to address them.

NOW WE GET TO MORE INTERESTING SEQUENCES...

DYNAMIC SEQUENCE INTERFACE

STATIC SEQUENCE, PLUS

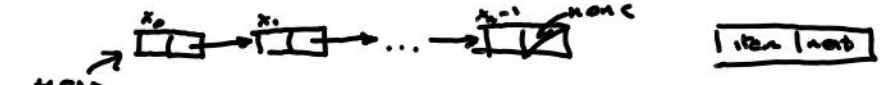
- insert_at(i, y) - make y the new x_i, shift i...n-1 up by 1
- delete_at(i) - delete x_i, shift i+1...n-1 down by 1



INSERT

- insert/delete_first/last are special cases. last changes no other indices; first changes all indices
- get first/last() } WHY DO WE NEED THESE? ALGORITHM EFFICIENCY IN SPECIAL CASES!
- set first/last(y)

SOLUTION: LINKED LISTS?



POINTER BASED, ALLOWS INSERT/DELETE IN CONSTANT TIME + O(i) TRAVERSAL STEPS

DYNAMIC SEQUENCE OPERATIONS

static	linked list
$A[i] = x_i$	
insert/delete $\Theta(n)$ time	insert/delete_first $O(1)$
shifting = copying	get/set follows all pointers $O(i)$

STATIC ARRAYS GOOD FOR READ/WRITE, LINKED LISTS GOOD FOR INSERT & DELETE

CAN WE FIND A STRUCTURE WITH BENEFITS OF BOTH?

DYNAMIC ARRAYS (lists in Python)

- relax the constraint that Array size = n
- enforce size = $\Theta(n)$ & $\geq n$
- maintain $A[i] = x_i$

- insert_last now takes constant time (unless length = size)
  when n = size we make array larger / copy to new memory segment of $C \cdot$ size
- n insert_last() from empty array

resize at $n = 1, 2, 4, 8, \ldots, 16, n$

resize cost = $\Theta\left(\sum_{i=1}^{\log n} 2^i\right) = \Theta\left(2^{(\log n + 1)} - 1\right)$ ①

geometric series dominated by the last term

① CONSTANT "AMORTISED" TIME - TOTAL COST OF k LENGTH RESIZES IS $O(k)$

AMORTIZATION

OPERATION TAKES $T(n)$ AMORTIZED TIME IF ANY k OPERATIONS TAKE AT MOST $k \cdot T(n)$ TIME

OPERATION, WORST CASE $O(\cdot)$

	static	dynamic		
	get/set	1st insert/del	LAST insert/del	@ i insert/del
array	1	n	n	n
linked list	n	1	n	n
dynamic array	1	n	1 (D)	n

(D) amortized

DYNAMIC ARRAYS OBV MEM GREEDY. PROBABLY SOME INTERESTING HISTORY HERE.

PROBLEM SET #1

1-1. FOR EACH OF THE FOLLOWING SETS OF FIVE FUNCTIONS, ORDER THEM SO THAT IF f_a APPEARS BEFORE f_b IN YOUR SEQUENCE, THEN $f_a = O(f_b)$. IF $f_a = O(f_b)$ & $f_b = O(f_a)$, INDICATE THIS BY ENCLOSING f_a & f_b W A SET WITH CURLY BRACES.

e.g. $f_1 = n$ $f_2 = \sqrt{n}$ $f_3 = n + \sqrt{n}$

would be $(f_2, \{f_1, f_3\})$ or $(f_2, \{f_3, f_1\})$

a) $f_1 = \log(n^n)$, $f_2 = (\log n)^n$, $f_3 = \log(n^{\text{COOL}})$, $f_4 = (\log n)^{\text{COOL}}$, $f_5 = \log\log(\text{COOL } n)$

$= f_5, f_3, f_4, f_1, f_2$ ✓

b) $f_1 = 2^n$, $f_2 = 600^n$, $f_3 = 2^{600^n}$, $f_4 = 600^{2^n}$, $f_5 = 600^{n^2}$

$= f_1, f_2, f_5, f_4, f_3$

c) $f_1 = n^n$, $f_2 = \binom{n}{n-6}$ ①, $f_3 = (6n)!$, $f_4 = \binom{n}{n/6}$ ②, $f_5 = n^6$

① $n!/(n-6)!(n-(n-6))! \Rightarrow n!/(n-6)!(6!) \Rightarrow \frac{n \times (n-1) \times \ldots \times (n-5)}{6!}$

leading term $\Rightarrow n^6/6!$

$\therefore f_2 \approx f_5$

② $n!/(n/6)!(5n/6)! \Rightarrow$ something bigger than $f_2 \doteq f_5$:)

(probably worth solving later; gap in knowledge!)

STIRLING!

$= \{f_2, f_6\}, f_4, f_1, f_3$

③

d) $f_1 = n^{n+4} + n!$, $f_2 = n^{7\sqrt{n}}$, $f_3 = 4^{3n \log n}$, $f_4 = 7^{n^2}$, $f_5 = n^{12 + 1/n}$

③ $4^{3n \log n} = \left(n^{\log 4}\right)^{3n} = n^{6n}$ $\therefore f_3 > f_1$

$\therefore f_5, f_2, f_1, f_3, f_4$

1.-2) GIVEN A DATA STRUCTURE D THAT SUPPORTS SEQUENCE OPERATIONS:

- D.build(x) in $O(n)$
- D.insert_at(i, x) & D.delete_at(i) in $O(\log n)$

WHERE n IS THE NUMBER OF ITEMS STORED IN D AT TIME OF OPERATION, DESCRIBE ALGORITHMS TO IMPLEMENT THE FOLLOWING HIGHER-LEVEL OPERATIONS IN TERMS OF LOWER-LEVEL ONES. EACH SHOULD RUN IN $O(k \log n)$ TIME

a) REVERSE(D, i, k): REVERSE in D THE ORDER OF THE k ITEMS STARTING AT i

- IF $k < 2$, RETURN
- DELETE $D[i+k] \Rightarrow$ "x"
- INSERT x @ D[i]
- DELETE NEW $D[i+1] \Rightarrow$ "x'"
- INSERT x' @ D[i+k]

} $O(\log n)$

- REPEAT FLOOR k/2 TIMES $O(k)$

∴ TOTAL TIME $\Rightarrow O(k \log n)$

b) move(D, i, k, j): MOVE THE ITEM IN P STARTING @ INDEX i. IN ORDER, TO BE IN FRONT OF ITEM @ x j. ASSUME i ≤ j < i+k is FALSE

- RETURN IF k < 1 .
- DELETE D[i] => "x"
- INSERT x @ D[j+n]

} O(log n)

YOU MISSED THIS!

- ITERATE FOR n=0 => n=k-1 O(k)

YOU'RE LEANING ON FOR/NEXT LOOPS RATHER THAN RECURSION; BE CAREFUL!

GIVEN SOLUTION:

```
def move(D,i,k,j):
    if k<1:
        return
    x = D.delete_at(i)
    if j>i:
        j=j-1
    D.insert_at(j,x)
    j=j+1
    if i>j:
        i=i+1
    move(D,i,k,j)
```

1-3. A STUDENT KEEPS COURSE NOTES ON PAGES IN A 3-RING BINDER. IF SHE HAS N PAGES OF NOTES, THE FIRST PAGE IS AT INDEX 0 & THE LAST IS AT INDEX n-1.
SHE OFTEN REORDERS HER PAGES; TO ASSIST SHE HAS TWO BOOKMARKS A & B

DESCRIBE A DATABASE TO KEEP TRACK OF PAGES IN THE BINDER, SUPPORTING THE FOLLOWING OPERATIONS, WHERE n IS THE NUMBER OF PAGES IN THE BINDER @ TIME OF OPERATION. ASSUME BOOKMARKS ARE PLACED PRIOR TO SHIFT/MOVE OPERATIONS, AND THAT $A_i < B_i$

build(x) - init in $O(|x|)$ time

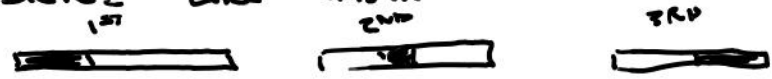
place_mark(i, m) - place bookmark between pages i & i+1 in $O(n)$

read_page(i) - return page in $O(1)$

shift_mark(m, d) - take bookmark @ i and move it in front of page i+d $(d \in \{-1, 1\}$, $O(1)$

move_page(m) - take page in front of bookmark $m \in \{A, B\}$ and move it in front of other

1-3. cont) MOVING PAGES @ MARKS IN CONSTANT TIME IS OUR BIG CONSTRAINT. WE CAN SET UP 3 DYNAMIC ARRAYS OF SIZE N EACH, AND USE BOOKMARKS TO DENOTE PARTITIONS. SO BUILD IS LINEAR TIME OVER 3 PARTITIONS, READ IS CONSTANT B/C YOU KEEP BOOKMARK METADATA TO MAP INDEX TO PARTITION, AND ORGANISE STORAGE LIKE THIS...



THIS ALLOWS CONSTANT TIME PUSH/POP OPERATIONS W/ CONSTANT TIME READ SINCE OFFSETS ARE KNOWN

1-4. DOUBLY LINKED LISTS: 

CIRCULAR POINTERS. FOR THIS PROBLEM, ASSUME NO LENGTH METADATA

a) ALGORITHMS FOR PUSH/POP W CONSTANT TIME

- INSERT FIRST
  - ADD ITEM w/ POINTER TO PREVIOUS first, and last
  - UPDATE POINTER in OLD FIRST TO new first

etc etc

1-4/ cont b) given two nodes x_1 and x_2 from a doubly linked list L, where x_1 occurs before x_2, describe a constant-time algorithm to remove all nodes between x_1 & x_2 and return as a new doubly linked list.

HOW DO YOU DO THIS W/O LINEAR TIME TRAVERSAL?! OTHERWISE IT'D BE UPDATING POINTERS & ADDING HEAD TAIL META, PAYING ATTENTION IN CASE YOU NEED TO UPDATE LOCATION OF ORIGINAL META.

DOES "given" MEAN I HAVE THEM IN CODE?!?

c) GIVEN node x from doubly linked list L_1 & second doubly linked list L_2, describe algorithm to splice L_2 into L_1 @ x

- update next pointer of L_2 tail to x next
- update x next pointer to head of L_2
- update L_2 meta to be sad :)

d) see REPO; PROBLEM SET 1

SIDEQUEST: STIRLING'S APPROXIMATION

$$n! \approx \sqrt{(2\pi n)} \cdot (n/e)^n$$

or (more useful form in asymptotic analysis)

$$\log(n!) \approx n\log(n) - n\log(e) + O(\log(n)) = n\log(n) - \Theta(n)$$

WHAT THIS IS SAYING IS THAT $N!$ GROWS LIKE N^N, UP TO LOWER-ORDER CORRECTIONS, SO WHEN LOOKING AT FACTORIAL TIME IN $O(n)$ ANALYSIS THE DOMINANT BEHAVIOR IS CAPTURED BY COMPARING AGAINST N^N

APPLY TO 1-1 <> $f_4 = \binom{n}{n/6} = \frac{n!}{(n/6)!(5n/6)!}$ (D; ①, ②, ③)

① $n\log(n) - n\log e + O\log n$

② $\frac{1}{6}(n\log(n/6) - n\log e) + O\log n/e$

③ $\frac{5}{6}(n\log(5n/6) - n\log e) + O\log 5n/6$

LOTS OF SIMPLIFICATION LATER...

$$\log\binom{N}{N/6} = nH(1/6) + O\log n$$

where $H(p)$ ↓ $-p\log(p) - (1-p)\log(1-p)$ BINARY ENTROPY FUNCTION, FIXED CONSTANT

$\therefore \binom{n}{n/6} \Rightarrow 2^{**}(\Theta(n))$ ← exponential, but below 2^n

LECTURE 3: SETS AND SORTING

NOTE TO SELF: THINK IN RECURSION

INTERFACE: COLLECTION OF OPERATIONS (e.g. sequence & set)

DATA STRUCTURE: WAY TO STORE DATA THAT SUPPORTS SET OF OPS.

THE DISTINCTION BETWEEN THESE IS IMPORTANT — WHEN CHOOSING AN ALGORITHM: MAKE SURE INTERFACE IS CORRECT FOR THE PROBLEM WE'RE CONCERNED WITH & CHOOSING A DATA STRUCTURE WHOSE EFFICIENCIES & MEM USE ALIGN WITH PRIORITIES.

SET - BIG PILE OF THINGS

CONTAINER	build(A)	given iterable A, build sequence from items in A
	len()	return # of stored items
STATIC	find(k)	return stored item w/ key k
DYNAMIC	insert(x)	add x to set. (replace item w/ key x.key if one already exists
	delete(k)	remove & return stored item w/ key k ①
ORDER	iter_ord()	return stored items one by one in key order
	find_min()	return stored item w/ smallest key
	find_max()	" " largest "
	find_next(k)	return stored item with smallest key larger than k
	find_prev(k)	return stored item with largest key smaller than k

① RETURNING IS INTERESTING!

THE ABOVE IS AN INTERFACE. IT'S A SPEC, NOT A DATA STRUCTURE. WE NEED TO LOOK AT DATA STRUCTURES THAT IMPLEMENT THIS.

IMPLEMENTATION #1: UNORDERED ARRAY

$[x_1 \; x_2 \; x_3 \; x_4 \ldots x_n]$ is this efficient or useful?

finding key item requires ordering or traversal! $O(n)$

so does build, insert, etc. - all the specced ops are $O(n)$

IMPLEMENTATION #2: ORDERED ARRAY; ORGANISED BY KEY

obviously this frontloads build time, $O(n \log n)$ but it will make lookup easier. find-min & max become "free", find-prev & find-next move to $O(\log n)$. insert/delete still linear.

binary search ($\log n$)

so this is a tradeoff @ architecture level!

LET'S FOCUS ON THE BUILD - SORTING!

VOCAB: DESTRUCTIVE: OVERWRITES INPUT ARRAY

IN-PLACE: USES $O(1)$ EXTRA SPACE

INPUT: ARRAY of n KEYS (A)

OUTPUT: SORTED ARRAY (B)

SIMPLE SORTING ALGORITHM: "PERMUTATION SORT"

```
def permutation_sort(A):
    #sort A
    for B in permutations(A):
        if is_sorted(B):
            return B
```

- this lists every possible permutation of A and returns the one which is sorted.

lower bound notation → $\Omega(n!)$ to list perms

$\Theta(n)$ to check sorted condition

∴ total time: $\Omega(n! \cdot n)$ ← very inefficient

SELECTION SORT

8 2 4 9 3 — take largest # and move to end

8 2 4 3 | 9 — then do the same for shortened list

etc.

THEORETICAL IMPLEMENTATION

① FOUND BIGGEST ITEM w/ INDEX ≤ i

② SWAP

③ SORT 1, ..., i-1

SELECTION SORT IMPLEMENTATION, CONT

①
```
def prefix_max(A, i):
    # return index of maximum in A[:i+1]
    if i > 0:
        j = prefix_max(A, i-1)
        if A[i] < A[j]:
            return j
    return i
```

SHOW BY INDUCTION THAT THIS WORKS

base case: i = 0 — 1 element

then biggest element from 0, ..., i

a) either @ element i or

b) index < i

COMPUTATION TIME: $S(n) = S(n-1) + \Theta(1) \Rightarrow S(n) \overset{!}{=} cn$

$cn \overset{?}{=} c(n-1) + \Theta(1)$

$c = \Theta(1)$ ✓

NB: ABOVE USES "SUBSTITUTION METHOD"

IMPLEMENT SELECTION SORT, CONT

```
def selection_sort(A, i = None):
    # sort A[:i+1]
    if i is None: i = len(A)-1
    if i > 0:
        j = prefix_max(A, i)        # find
        A[i], A[j] = A[j], A[i]     # swap
        selection_sort(A, i-1)      # recurse
```

$T(n) = T(n-1) + \Theta(n) \overset{?}{=} \Theta(n^2)$

$T(n) \overset{?}{=} cn^2$

$cn^2 \overset{?}{=} c(n-1)^2 + \Theta(n) = cn^2 - 2cn + c + \Theta(n)$

$\Theta(n) = 2cn - c$ ✓

MERGE SORT! HOW WE GET TO $n\log n$

7, 1, 5, 6, 2, 4, 9, 3 – each number is sorted in array of length 1

↓

[7, 1], [5, 6], [2, 4], [9, 3]

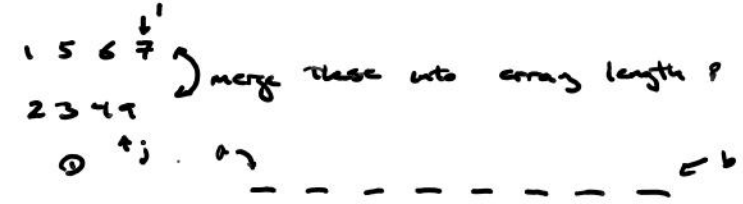
↓

[1, 7 | 5 6], [2, 4 | 9, 3]

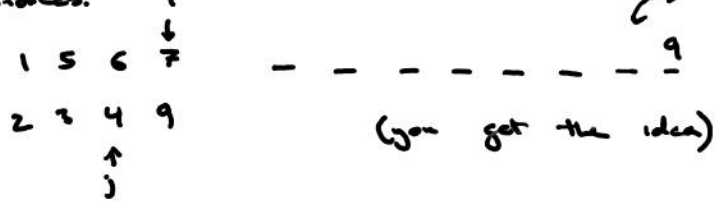
WE KNOW BOTH SIDES OF THE LINE ARE SORTED!

MERGE SORT (cont)

TWO-FINGER SORT

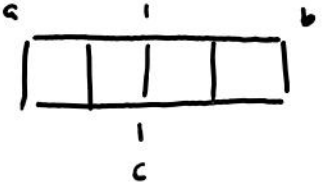


① index moves when activated, always choose largest value of two indices.



(you get the idea)

```
def merge_sort(A, a=0, b=None):
    # sort A[a:b]
    if b is None: b = len(A)
    if 1 < b-a:
        c = (a+b+1) // 2        # grabs middle of array
        merge_sort(A, a, c)     # orders to the left
        merge_sort(A, c, b)     # orders to the right
        L, R = A[a:c], A[c:b]   # merge results
        merge(L, R, A, len(L), len(R), a, b)
```



MERGE SORT (cont)

```
def merge(L, R, A, i, j, a, b)
    # merge sorted L[:i] and R[:j] into A[a:b]
    if a < b:
        # check if top or bottom pointing @ biggest
        if (j <= 0) or (i > 0 and L[i-1] > R[j-1]):
            A[b-1] = L[i-1]  # put item in top pointer @ target
            i = i-1          # move top pointer
        else:
            A[b-1] = R[j-1]  # put item in bottom pointer @ target
            j = j-1          # move bottom pointer
        merge(L, R, A, i, j, a, b-1)  # recurse
                                      # shifting target index down
```

$S(n) = S(n-1) + \Theta(1)$

$S(n) = \Theta(n)$ ← merge runtime

what is sort?

$T(1) = \Theta(1)$

$T(n) = 2T(\frac{n}{2}) + \Theta(n) \Rightarrow T(n) \stackrel{?}{=} \Theta(n \log n)$

$cn \log n \stackrel{?}{=} 2c\frac{n}{2}\log\frac{n}{2} + \Theta(n)$

$= cn(\log n - \log 2) + \Theta(n)$

$\Theta(n) = cn \log 2$ ✓

SIDE QUEST: SUBSTITUTION FOR COMPUTATIONAL TIME

METHOD: GUESS A CLOSED FORM

VERIFY GUESS SATISFIES RECURRANCE W/

DIRECT SUBSTITUTION

NOTE: GUESS ISN'T DERIVED FROM RECURRANCE; PATTERN-MATCHED FROM INTUITION OR EXPERIENCE.

LECTURE 4: HASHING

(1) PROVE YOU CAN'T FIND(k) FASTER THAN $O(\log n)$

(2) SHOW HOW TO FIND(k) "

(THIS IS BECAUSE WE'LL RELAY A CONSTRAINT, NOT BECAUSE WE'LL BREAK MATH)

FIND KEY: SEE IF ITEM IN STASH HAS PROPERTY THAT MATCHES KEY.

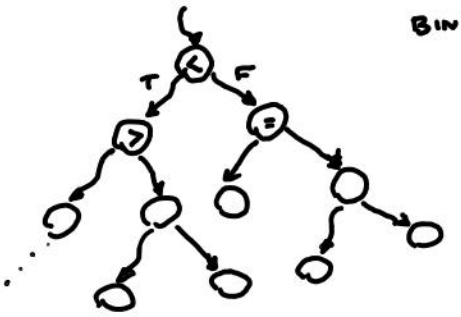
(1) COMPARISON MODEL - ITEMS ARE BLACK BOXES.

GIVEN TWO ITEMS WE CAN "weigh" THEM

$=, <, >, \geq, \leq, \neq$

OUTPUTS: TRUE OR FALSE BINARY

ALGORITHMS IN COMPARISON MODEL: DECISION TREES



BINARY TREE - NODES ARE COMPARISONS AND LEAVES ARE OUTPUTS.

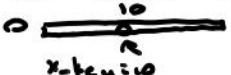
FOR FIND WE NEED AT LEAST $n+1$ LEAVES (n ITEMS; FALSE)

FOR $n+1$ LEAVES WE'LL NEED TO CHECK LONGEST ROOT-TO-LEAF PATH. THEREFORE WE WANT TREE TO HAVE AS FEW LAYERS AS POSSIBLE. MINIMUM HEIGHT OF A BINARY TREE: $\log n!$

BALANCED BINARY TREE SHOWS BEST POSSIBLE FIND(K) IS $O(\log n)$ ... IN THE COMPARISON MODEL

CAN WE DO BETTER? <u>YES</u>.

REMEMBER RANDOM-ACCESS MEMORY? CONSTANT-TIME LOOKUPS? TAKE ITEM W/ KEY 10 AND STORE IN ARRAY AT INDEX #10! DIRECT-ACCESS ARRAY. THEN WE CAN JUST CHECK Array[i] AND IF SOMETHING IS THERE WE CAN RETURN IT

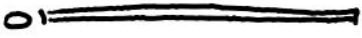


- RETRIEVE ITEM IN CONSTANT TIME
- INSERT/DELETE

PROBLEMS

- WE CAN'T STORE MULTIPLE ITEMS WITH THE SAME KEY
- WE DON'T KNOW HOW HIGH THE NUMBERS GO - INSTANTIATING VERY LARGE ARRAY
  $u \Rightarrow$ largest key might be $\gg n$
- REQUIRES INTEGER KEY
- LIMITATION OF WORD-RAM MODEL FORCES $u < 2^w$ (OR THE WORD CAN'T MAP WHOLE ARRAY)

MITIGATION: ALLOCATE LESS SPACE TO DIRECT ACCESS ARRAY → MOVE FROM LENGTH u TO ~ LENGTH n



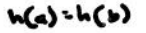
COMPRESSED SPACE → 

$m = \Theta(n)$ h: map $\{0, \ldots, u-1\} \Rightarrow \{0, \ldots, m-1\}$

"HASH TABLE"

ITEMS MUST COLLIDE IN COMPRESSION! BEST FUNCTION WILL EVENLY DISTRIBUTE KEYS ACROSS COMPRESSED SPACE

SOLUTIONS: CHOOSE $m > n$ AND FORCE "OPEN ADDRESSING" (HARD TO ANALYZE)

① 

STORE POINTER IN HASH TABLE LOCATION


LINKED LIST, DYNAMIC ARRAY, ETC.

↳ POINTS TO NEW DATA STRUCTURE WITH SMALL SIZE

CHAINING

OUR HASH FUNCTION NEEDS TO ENSURE CHAIN SIZE IS SMALL. HOW DO WE PICK A GOOD ONE?

POSSIBILITY #1: DIVISION HASH FUNCTION

$h(k) = k \bmod m$

wants well-distributed keys in u

sometimes very bad performance

BUT OUR HASH FUNCTION DOESN'T NEED TO BE DETERMINISTIC. IT IS IMPOSSIBLE TO PICK A <u>FIXED</u> HASH FUNCTION WITH A SET RUNNING TIME INDEPENDENT OF INPUT. SO WE SELECT OUR HASH FUNCTION <u>DYNAMICALLY</u>. AND WE WEAKEN OUR NOTION OF "CONSTANT TIME"

(REALLY A FAMILY)

UNIVERSAL HASH FUNCTION

$$\text{hash}(k) = (((ak+b) \bmod p) \bmod m)$$

$$\mathcal{H}(p,m) = \{h_{ab}(k) \mid a,b \in \{0,\dots,p-1\} \text{ & } a \neq 0\}$$

p: prime # larger than u

a and b picked randomly - hard for users to give bad data :)

$$\Pr_{h \in \mathcal{H}}\{h(k_i) = h(k_j)\} \leq \frac{1}{m} \quad \forall k_i \neq k_j, \in \{0 \dots u-1\}$$

distributes well over array length m

$\therefore$ chain length expected to be constant.

INDICATOR RANDOM VARIABLE

X_{ij} over choice $h \in \mathcal{H}$

$X_{ij} = 1$ if $h(k_i) = h(k_j)$; 0 otherwise

($h(k_i) = h(k_j)$: collision)

SIZE OF CHAIN @ $m[i] = X_i = \sum_{j=0}^{n-1} X_{ij}$

$$\mathop{\mathbb{E}}_{h \in \mathcal{H}}\{X_i\} = \mathop{\mathbb{E}}_{h \in \mathcal{H}}\left\{\sum_j X_{ij}\right\} = \left(\sum_{j \neq i} \mathop{\mathbb{E}}\{X_{ij}\}\right) + 1$$

linearity of expectation: the expected value of the sum of random variables is always equal to the sum of their individual expected values

$$\textcircled{1} \sum_{j \neq i} \frac{1}{m} + 1 = 1 + \frac{n-1}{m}$$

↖ CHAIN LENGTH CONSTANT

HAVE TO REBUILD WHOLE TABLE IF $m < n$ - prob amortised time.

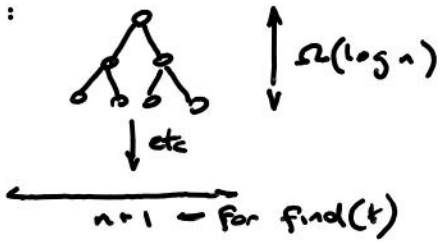
find(k) then becomes:

- follow hash function for k (constant time)
  - m, a, b, p stored in set metadata
- comparison lookup in chain (constant time)
  - ↑ linear scan of $O\left(1 + \frac{n-1}{m}\right)$

∴ choosing m to be $\Theta(n)$ gives $\frac{n-1}{m}$ as constant, so we're in constant time. since the other parts of alg are also constant time, find(k) = O(1) (insert/delete also)

LECTURE 5: LINEAR SORTING

WE'VE DONE $n \log n$ MERGE SORT – BUT CAN WE DO BETTER? NOT IN COMPARISON MODEL FOR THE EXACT SAME REASON find(k) is $\log n$ TIME – DECISION TREE:



SORTING REQUIRES $n!$ OUTPUTS. $\therefore$ RUNTIME IS $\Omega(\log n!)$ AT LOWER BOUND. THERE ARE TWO PATHS TO TAKE A LOG HERE. STERLING'S APPROX. IS ONE, BUT THE OTHER IS..

$$\underbrace{(n)(n-1)(n-2)\dots(1)}_{\text{HALF THE TERMS } \geq n/2}$$

$\therefore n! \geq \left(\frac{n}{2}\right)^{\left(\frac{n}{2}\right)} \Rightarrow \Omega(n \log n)$ LOWER BOUND FOR SORT!

HOW DO WE SORT OUTSIDE OF COMPARISON MODEL

IDEA: PUT ITEMS BY KEY INTO DIRECT ACCESS ARRAY. THEN RETURN THE ARRAY IN ORDER!

YES: THIS IS DIRECT ACCESS ARRAY SORT.

DAA SORT 

(CAVEAT: ALL KEYS HAVE TO BE UNIQUE)

- MAKE ARRAY $(O(u))$
- STORE ITEM x IN INDEX $x.key$ $(O(n))$
- TRAVERSE DA & RETURN ITEMS SEEN IN ORDER $(O(u))$

TOTAL TIME $\Rightarrow O(n+u)$ LINEAR TIME SORTING!
WHEN $u = O(n)$

WHAT IF WE HAD $u < n^2$? TIME = $O(n^2)$, WHICH IS NOT IDEAL. BUT WHAT IF WE WERE STORING KEYS AS FACTORS AND RECOVER k COMPUTATIONALLY?

$a = k // n$ $O(1)$
$b = k \% n$ $O(1)$
} FOR INTEGERS

$k = an + b$

EXAMPLE: $[17, 3, 24, 22, 12]$ $n=5$, $u=24$

$[(3,2), (0,3), (4,4), (4,2), (2,2)]$

HOW DO WE SORT TUPLES?

TUPLE SORT:

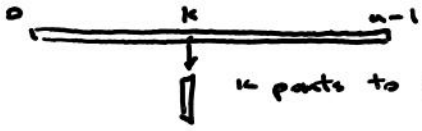
[32, 03, 44, 42, 22]

IF WE SORT BY a $\overset{1}{\in}$ THEN b WE LOSE ALL OF THE STRUCTURE GIVEN BY a. SORTING BY b THEN a WORKS BUT ONLY FOR SORTING ALGS. THAT PRESERVE ORDER: STABLE SORTING ①

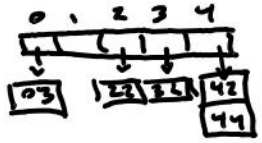
① OBVIOUSLY THERE ARE WAYS TO GET AROUND THIS BUT IT WILL TAKE MORE TIME THAN DOING TWO PASSES OF TUPLE SORT!

BUT WE CAN'T USE DAA SORT WHERE: 42 $\overset{1}{\in}$ 44 HAVE SAME 'k in a

SOLUTION: COUNTING SORT



w/ OUR TUPLES:



COMES IN ORDER, BUT WE SORTED BY b FIRST :)

$O(n+u')$

$u' = n$

$n^3 \Rightarrow (a, b, c)$ THEN TUPLE SORT w/ INCREASING PRIORITY.

RADIX SORT: BREAK UP INTEGERS MAX SIZE n INTO A BASE-n TUPLE. # DIGITS: $\log_n U$

TUPLE SORT ON DIGITS W/ COUNTING SORT

TIME: $n * n \log_n n$, WHICH IS LINEAR FOR $n \log n^c$ (CONSTANT c)

WE'VE NOW SORTED IN LINEAR TIME FOR ITEMS IN POLYNOMIAL SPACE

PROBLEM SET 2

1) SOLVING RECURRANCES:

DERIVE SOLUTIONS TO THE FOLLOWING. A SOLUTION SHOULD INCLUDE THE TIGHTEST UPPER & LOWER BOUND THAT THE RECURRANCE WILL ALLOW:

a) $T(n) = 4T\left(\frac{n}{2}\right) + O(n)$

$c = \log_2 4 = 2$; $\underline{O(n^2) \rightarrow O(n)}$ CASE 1

b) $T(n) = 3T\left(\frac{n}{\sqrt{2}}\right) + O(n^4)$

$c = \log_{\sqrt{2}} 3 = 2\log 3$; $O(n^{2\log 3}) \ll \underline{O(n^4)}$ CASE 3

c) $T(n) = 2T\left(\frac{n}{2}\right) + 5n\log n$

$c = 1$; $O(n) \underline{\leq} 5n\log n$ (CASE 2, $k = 1$)

$\Rightarrow T(n) = O(n^c (\log n)^{k+1})$

$= O(n \log^2 n)$

1) cont d) $T(n) = T(n-2) + \Theta(n)$

$$kn^2 \stackrel{?}{=} T(n-2) + \Theta(n)$$

$$= k(n-2)^2 + cn$$

$$= kn^2 - 4kn + 4k + cn$$

$$= kn^2 + 4k$$

$\checkmark$

2) SORTING:

FOR EACH OF THE SCENARIOS BELOW, CHOOSE A SORTING ALGORITHM FROM SELECTION SORT, MERGE SORT OR INSERTION SORT. JUSTIFY YOUR SELECTION

a) SUPPOSE YOU ARE GIVEN A DATA STRUCTURE D MAINTAINING ENTRIES IN ORDER ON N ITEMS. GET_AT TAKES $\Theta(1)$ TIME & SET_AT IS $\Omega(n \log n)$ TIME.

- MERGE SORT DOESN'T SORT IN PLACE
- SELECTION SORT PERFORMS FEWER SET_AT OPERATIONS THAN INSERTION SORT

∴ PICK SELECTION SORT

b) SUPPOSE YOU HAVE A STATIC ARRAY CONTAINING POINTERS TO n COMPARABLE OBJECTS PAIRS OF THESE OBJECTS TAKE $\Theta(\log n)$ TIME TO COMPARE

2) b cont) MERGE SORT IS BEST OF THESE IN TERMS OF MINIMISING COMPARISONS, WHICH WE PROVED IN LECTURE 3

c) SUPPOSE YOU HAVE A SORTED ARRAY A CONTAINING n INTEGERS, EACH FITTING INTO A SINGLE MACHINE WORD. NOW SUPPOSE SOMEONE PERFORMS $\log \log n$ SWAPS BETWEEN PAIRS OF ADJACENT ITEMS IN A SO A IS NO LONGER SORTED.

- SOME LEFTOVER STRUCTURE MAY MAKE INSERTION OR SELECTION SORT RUN FASTER.
- GIVEN INSERTION SORT SWAPS 1 AT A TIME, IT SHOULD RUN IN $O(n+s)$ TIME WHERE s IS # SWAPS, AND BE LINEAR IN THESE CIRCUMSTANCES

3) TTCARD IS SEARCHING FOR A FRIEND ON AN ISLAND WHICH RUNS NORTH-SOUTH FOR n KM. THE ISLAND IS SINKING FROM THE NORTH AND SOUTH ENDS. TTCARD CAN TELEPORT IN $O(1)$ TIME AND ALWAYS TELL IF HIS FRIEND IS TO THE NORTH OR SOUTH

3 cont) DESCRIBE AN ALGORITHM BY WHICH TCARD CAN FIND HIS FRIEND IN $O(\log k)$ TIME.

- START BY VISITING POINT 1 & $n-1$
- THEN MOVE INLAND BY 2^i EACH TIME
  i.e. 2, $n-2$, 4, $n-4$, etc
- KEEP TRACK OF FRIEND'S N/S DIRECTION. OR CROSSOVER OCCURS @ N/2
- WHEN IT FLIPS, WE HAVE A BOUNDED SEARCH AREA - SIZE $\approx 2^i$
- THIS IS A TREE W/ n LEAVES BEING REPEATEDLY SLICED IN TWO; $O(\log n)$ TIME

4) MIKBOOKTUBE.tv IS A SERVICE THAT LETS VIEWERS CHAT WHILE WATCHING SOMEONE PLAY VIDEO GAMES. EACH VIEWER HAS A UNIQUE ID; VIEWERS CAN SEE THE LAST K MESSAGES. BANNING A USER ALSO DELETES THEIR MESSAGES.

DESCRIBE A DATABASE FOR MIKBOOKTUBE.tv SUPPORTING THE FOLLOWING OPERATIONS:

build(v)	$O(n \log n)$
send(v, m)	$O(\log n)$
recent(k)	$O(k)$ ← find-last & linear traverse!
ban(v)	$O(n_v + \log n)$ doubly linked list

DOUBLE-LINKED LIST FOR TOP LEVEL STORAGE ALLOWS $O(n)$ BUILD TIME REQUIREMENT TO LINK USER w/ POSTS SUGGESTS NEED TO ALSO MAINTAIN SORTED USER LIST AS ARRAY w/ POINTER TO MESSAGES, WHICH ALSO POINT TO LINKED LIST

∴ BUILD CONSTRAINED BY USER LIST SORT @ $n \log n$

∴ SEND CONSTRAINED BY FIND IN SORTED ARRAY $\log n$.

∴ RECENT: IS A TRAVERSE FROM LAST NODE OF LINKED LIST @ $O(k)$

∴ BAN v IS A $\log n$ SEARCH IN A SORTED ARRAY PLUS LINEAR TRAVERSE THROUGH MESSAGES, WORST-CASE $O(n_v + \log n)$

5). TIM THE BEAVER IS ARRANGING TIM TALKS, A LECTURE SERIES THAT ALLOWS ANYONE TO SCHEDULE A TIME TO TALK PUBLICLY. A TALK REQUEST IS A TUPLE (s,t), WHERE s & t ARE POSITIVE INTEGERS DENOTING THE START & END TIMES OF TALKS, WITH $s < t$

5 cont) TIM MUST MAKE ROOM RESERVATIONS TO HOLD TALKS. A ROOM BOOKING IS A TRIPLE (k, s, t) WHERE k CORRESPONDS TO THE REQUESTED ROOM. TWO BOOKINGS ARE DISJOINT IF $t_1 \leq s_2$ OR $t_2 \leq s_1$ & ADJACENT IF EITHER $t_1 = s_2$ OR $t_2 = s_1$. A BOOKING SCHEDULE IS AN ORDERED TUPLE OF ROOMS WHERE:

- EVERY PAIR OF ROOM BOOKINGS FROM THE SCHEDULE ARE DISJOINT
- ROOM BOOKINGS APPEAR WITH INCREASING STARTING TIME IN THE SEQUENCE
- EVERY ADJACENT PAIR OF ROOM BOOKINGS RECEIVES A DIFFENT ROOM #.

a) GIVEN TWO BOOKING SCHEDULES $B_1 \& B_2$, WHERE $n = |B_1| + |B_2|$ & $B_1 \& B_2$ ARE THE RESPECTIVE SCHEDELES OF TALK REQUESTS $R_1 \& R_2$, DESCRIBE AN $O(n)$-TIME ALGORITHM TO COMPLETE A BOOKING SCHEDULE B FOR $R = R_1 \cup R_2$

WE NEED TO MERGE TWO BOOKING SCHEDULES, WE HAVE AN ARBITRARY # OF ROOMS W/ WHICH TO DO SO.

END STATE

5) a cont) $T = 0 \longrightarrow t(\max(B_1, B_2))$

B' is our new schedule.

goal: satisfy booking constraints as we walk up to the end state. maintain counters for B_1 & B_2

when both schedules have talks remaining

- neither booking is active; advance T to next (earliest) booking
- next booking overlaps T but does not overlap anything from the other schedule - append (k, s, t) to B'; increment T to t; update relevant counter
- next booking overlaps T and booking after (s_i) starts before t. append (k, s, t) to B'; increment T to s_i, update relevant counter.
- next talks starting simultaneously - one can be (k, s, t); other must be (k', s, t) since rooms can't overlap. increment T to $\min(t_1, t_2)$; increment both counters
- we're out of bookings from B_1 or B_2. append other schedule to B' and terminate

- this executes in linear time, something below $t(\max(B_1, B_2))$

5) b) GIVEN A SET R OF n TALK REQUESTS, DESCRIBE $O(n \log n)$ - TIME ALGORITHM TO RETURN BOOKING SCHEDULE THAT SATISFIES R

- WE'VE ALREADY DONE THIS FOR B_1 & B_2;

$\log n$ TIME SUGGESTS SPLITTING. SO $R/2 \to R_1$ & R_2 etc and use alg. above. BASE CASE IS $||R|| = 1$; $R = \{s,t\} \Rightarrow (1, s, t)$

TIME: $O(n) + 2T(n/2)$, WHICH WE DID IN 1) c) $\Rightarrow O(n \log n)$

c) SEE REPO