Ravi Daryabhai

Quiz 1 Review

High Level

- Need to solve large problems n with constant-sized code, **correctly** and **efficiently**
- Analyzing running time: **How to count?**
  - **Asymptotics**
  - **Recurrences** (substitution, tree method, Master Theorem)
  - **Model of computation**: Word-RAM, Comparison
- How to solve an algorithms problem
  - Reduce to a problem you know how to solve
    - Use an algorithm you know (e.g. **sort**)
    - Use a data structure you know (e.g. **search**)
  - Design a new recursive algorithm (harder, mostly in 6.046)
    - Brute Force
    - Decrease & Conquer
    - Divide & Conquer (like merge sort)
    - Dynamic Programming (later in 6.006!)
    - Greedy/Incremental

Algorithm: Sorting

Reduce your problem to a problem you already know how to solve using known algorithms. You should know **how** each of these sorting algorithms are implemented, as well as be able to **choose** the right algorithm for a given task.

Algorithm	Time $O(\cdot)$	In-place?	Stable?	Comments
Insertion Sort	n^2	Y	Y	$O(nk)$ for k-proximate
Selection Sort	n^2	Y	N	$O(n)$ swaps
Merge Sort	$n \log n$	N	Y	stable, optimal comparison
AVL Sort	$n \log n$	N	Y	good if also need dynamic
Heap Sort	$n \log n$	Y	N	low space, optimal comparison
Counting Sort	$n + u$	N	Y	$O(n)$ when $u = O(n)$
Radix Sort	$n + n \log_n u$	N	Y	$O(n)$ when $u = O(n^c)$

Data Structures

Reduce your problem to using a data structure storing a set of items, supporting certain search and dynamic operations efficiently. You should know **how** each of these data structures implement the operations they support, as well as be able to **choose** the right data structure for a given task.

Sequence data structures support **extrinsic** operations that maintain, query, and modify an externally imposed order on items.

Sequence Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic		
	`build(X)`	`get_at(i)` `set_at(i,x)`	`insert_first(x)` `delete_first()`	`insert_last(x)` `delete_last()`	`insert_at(i,x)` `delete_at(i)`
Array	n	1	n	n	n
Linked List	n	n	1	n	n
Dynamic Array	n	1	n	$1_{(a)}$	n
Sequence AVL	n	$\log n$	$\log n$	$\log n$	$\log n$

Set data structures support **intrinsic** operations that maintain, query, and modify a set of items based on what the items are, i.e., based on the **unique key** associated with each item.

Set Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic	Order	
	`build(X)`	`find(k)`	`insert(x)` `delete(k)`	`find_min()` `find_max()`	`find_prev(k)` `find_next(k)`
Array	n	n	n	n	n
Sorted Array	$n \log n$	$\log n$	n	1	$\log n$
Direct Access	u	1	1	u	u
Hash Table	$n_{(e)}$	$1_{(e)}$	$1_{(a)(e)}$	n	n
Set AVL	$n \log n$	$\log n$	$\log n$	$\log n$	$\log n$

Priority Queues support a limited number of Set operations.

Priority Queue Data Structure	Operations $O(\cdot)$			
	`build(X)`	`insert(x)`	`delete_max()`	`find_max()`
Dynamic Array	n	$1_{(a)}$	n	n
Sorted Dyn. Array	$n \log n$	n	$1_{(a)}$	1
Set AVL	$n \log n$	$\log n$	$\log n$	$\log n$
Binary Heap	n	$\log n_{(a)}$	$\log n_{(a)}$	1

Problem Solving

Testing Strategies

- Read every problem first, rank them in the order of your confidence
- For most problems, you can receive $\geq 50\%$ of points in two sentences or less
- Probably better to do half the problems well than all the problems poorly

Types of problems

Type	Internals	Externals	Tests understanding of:
Mechanical	Y	N	how core material works
Reduction	N	Y	how to apply core material
Modification	Y	Y	how to adapt core material (augmentation, divide & conquer, amortization, etc.)

Questions to ask:

- Is this a Mechanical, Reduction, or Modification type problem?
- Is this problem about data structures? sorting? both?
- If data structures, do you need to support **Sequence** ops? **Set** ops? both?
- If stuck, is there an easy way to get a correct but inefficient algorithm?

Question yourself if you are:

- Trying to compute decimals, rationals, or real numbers
- Using Radix sort for every answer
- Augmenting a binary tree with something other than a subtree property

Data Structures Problems

- First solve using Sorting or Set/Sequence interfaces, choose algorithm/data structure after
- Describe all data structure(s) used (including **what data they store**) and their invariants
- Implement **every operation** we ask for in terms of your data structures
- Separate and label parts of your solution!

Yes, but don't forget O(1)ac to remove name from hash table!

Problem 1. Restaurant Lineup (S19 Q1)

Popular restaurant Criminal Seafood does not take reservations, but maintains a wait list where customers who have been on the wait list longer are seated earlier. Sometimes customers decide to eat somewhere else, so the restaurant must remove them from the wait list. Assume each customer has a different name, and no two customers are added to the wait list at the exact same time. Design a database to help Criminal Seafood maintain its wait list supporting the following operations, each in $O(1)$ time. State whether each operation running time is worst-case, amortized, and/or expected.

`build()`	initialize an empty database
`add_name(`x`)`	add name x to the back of the wait list
`remove_name(`x`)`	remove name x from the wait list
`seat()`	remove and return the name of the customer from the front of the wait list

Problem 2. Rainy Research (S19 Q1)

Mether Wan is a scientist who studies global rainfall. Mether often receives data measurements from a large set of deployed sensors. Each collected data measurement is a triple of integers (r, ℓ, t), where r is a positive amount of rainfall measured at latitude ℓ at time t. The **peak rainfall** at latitude ℓ **since** time t is the maximum rainfall of any measurement at latitude ℓ measured at a time greater than or equal to t (or zero if no such measurement exists). Describe a database that can store Mether's sensor data and support the following operations, each in worst-case $O(\log n)$ time where n is the number of measurements in the database at the time of the operation.

`build()`	initialize an empty database
`record_data(`r, ℓ, t`)`	add a rainfall measurement r at latitude ℓ at time t
`peak_rainfall(`ℓ, t`)`	return the peak rainfall at latitude ℓ **since** time t

(1.) Use a doubly-linked list to model the line (i.e., each node holds name of customer) and a hash table mapping names to nodes. `build` takes O(1) time to initialize empty versions of these data structures, `add_name(x)`/`seat` are O(1) operations on a doubly-linked list, and `remove` name is O(1)e to look up name in dictionary and O(1) to remove associated node in the doubly-linked list.

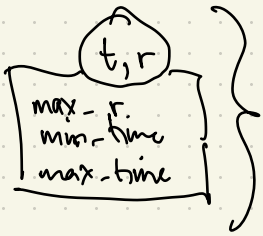
Don't need to keep min/max time!

6.006 Introduction to Algorithms
Spring 2020

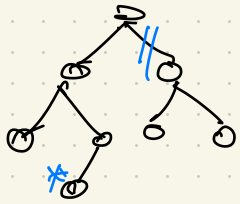
(2) We can keep a dictionary keyed on latitude l, whose values are pointers to set AVL trees keyed on time t. These trees may be augmented such that each node carries information about max rainfall, ~~min time~~, and max ~~time~~ among all nodes in subtree rooted at a given node (subtree augmentation since each value can be calculated in $O(1)$ time from child nodes for all nodes).

`build` takes $O(1)$ time to initialize these data structures, and `record_data` takes $O(1)$ ae to create entry in the dictionary and $O(\log n)$ to perform insertion into the set AVL tree (no additional cost to update augmentations). `peak_rainfall` takes $O(1)$ time to locate the proper tree, and $O(\log n)$ to return the peak rainfall since a given time (enabled by the subtree augmentations discussed earlier; see next page for detailed description).

[②, CONT.]



info contained at each node in set AVL tree keyed on time t



CASE 1: $t^* = A.t \Rightarrow$ update peak_since_t_star = max(A.r, A.right_child.max_r) (Return)

CASE 2: $t^* > A.t \Rightarrow$ search right

CASE 3: $t^* < A.t \Rightarrow$ search left, but update

peak_since_t_star =

max(peak_since_t_star,

max(A.r, A.right_child.max_r))

If at leaf B:

CASE 1: $t^* \leq B.t \Rightarrow$ update peak_since_t_star = max(peak_since_t_star, B.r) (Return)

CASE 2: $t^* > B.t \Rightarrow$ (Return) peak_since_t_star

$t^* = 3.5$

e.g.

t r

max_r

(min_time, max_time)