CS2040C Tut 2

Sorting (Part 2)

ADT

Mini Experiment

Yes we start with Q2 first.

Q2 Answer (for checking)

Input type → ↓ Algorithm	Random	Sorted Ascending	Sorted Descending	Nearly Sorted Ascending	Nearly Sorted Descending
(Opt) Bubble Sort	$O(N^2)$	$O(N)$ - best	$O(N^2)$	$O(N)$	$O(N^2)$
(Min) Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$
Insertion Sort	$O(N^2)$	$O(N)$ - best	$O(N^2)$	$O(N)$	$O(N^2)$
Merge Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
Quick Sort	$O(N \log N)$	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$
(Rand) Quick Sort	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$
Counting Sort	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$
Radix Sort	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$

Huhh??

Q: If counting/radix sort has a '*better time complexity*',
why don't we always just use these sorts?

Merge Sort:	O(N log N)
(Rand) Quick Sort:	O(N log N)
Counting Sort:	O(N)
Radix Sort:	O(N)

Ahh

Time complexity only denotes how runtime is *related* to **input size**. (i.e. value of **N**)

Radix sort is proportional to **number of digits** of the values *as well*, i.e. O(dN).

Counting sort requires time **and memory** proportional to the **maximum value** in the array *as well*.

Constant time differences

Between same time complexities, there are also differences in constant time.

In real world, *benchmarking* and understanding your data is important.

- Select the best algorithm for your use case.
- Justify why

Can I haz sort?

Sort an array of **N** numbers, between 0 and **K** inclusive.

Merge Sort / Counting Sort / Radix Sort

1. $N = 10^7$, $K = 31$ (Days of the month)
2. $N = 10^{15}$, $K = 10^{12}$ (Big data, *memory issue*?)
3. $N = 10^6$, $K = 10^{18}$ (Generic)

Can I haz sort? (Answers)

Sort an array of **N** numbers, between 0 and **K** inclusive.

Merge Sort / Counting Sort / Radix Sort

1. $N = 10^7$, $K = 31$ — **Counting Sort**
2. $N = 10^{15}$, $K = 10^{12}$ — **Radix Sort**
3. $N = 10^6$, $K = 10^{18}$ — **Merge Sort**

Challenge

Find out what sorting algorithm C++ STL **sort** uses *now*.

What sorting algorithm does C++ STL **stable_sort** use?

What about other languages?

Java, Python?

Why do you think they implemented it this way?

Sorting (Part 2)

Applications

Mini Experiment

Quick Select

Applications of Sorted Array

5. Set intersection/union between two sorted arrays **A** and **B**.
6. Finding a target pair **x** and **y** such that **x+y** equals to a target **z**, etc. (in the same array)

Brute Force - 5

For each number **x** in array A, loop through array B to see if it exists in array B.

Do the inverse for union.

Append to array **C** accordingly (union/intersection).

Complexity: O(**NM**)

Edge cases:

Duplicates in array A

Brute Force - 6

For each number **x** in array, loop through to find **z-x**.

Complexity: $O(\mathbf{N}^2)$

Binary Search - 5

Observation

Array A and B are sorted!

Instead of looping through for **x**, we can binary search instead.

Set intersection: O(N log M)**

Set union: O(N log M + M log N)

Binary Search - 6

Array is sorted!

Instead of looping through for **z - x**, we can binary search instead.

Complexity: O(N log N)

Better?

Is that the best we can do?

Observation

The value that we are binary searching for (**x**), always **increases*** (App. 5) or **decreases*** (App. 6).

*Technically non-decreasing/non-increasing

Better?

Observation

The value that we are binary searching for, always **increases*** (App. 5) or **decreases*** (App. 6).

This implies that the result of the binary search (index) will always be **increasing*** (App. 5) or **decreasing*** (App. 6).

2 pointer method

An *improved* brute force

For each number **x** in array A, loop through array B to see if it exists in array B.

Since B is sorted,

1. We can prune once it exceeds **x**.

2 pointer method

An *improved* brute force

We observed that **x** is *increasing**.

Since B is sorted,

1. We can prune once it exceeds **x**.
2. We can start from the index of the previous **x**.

2 pointer method

1	2	4	5	7	9	10	12

Search for **x** = 3.

2 pointer method

1	2	4	5	7	9	10	12

Search for **x** = 3. Does not exist.

2 pointer method

1	2	4	5	7	9	10	12

Search for **x** = 3. Does not exist.

Search for **x** = 5. Continue from current pointer position

2 pointer method

1	2	4	5	7	9	10	12

Search for **x** = 3. Does not exist.

Search for **x** = 5. Found.

Notice we don't reset the pointer.

Why can we do this?

2 pointer method

1	2	4	5	7	9	10	12

3	5	6	7	9	10	11	12

Search for **x** = 3.

Search for **x** = 5.

Search for **x** = 6.

2 pointer method

We can use this technique to solve Application 5 & 6.

Since the provided array is already sorted,

Application 5: **O(N + M)**

Application 6: **O(N)**

Will be demonstrated in lab 2.

Quick Select

http://en.cppreference.com/w/cpp/algorithm/nth_element

Summary

1. Randomly pick a number as **pivot**
2. Compute its rank
    a. If k == rank, we are done
    b. If k < rank, our target is to the left
    c. If k > rank, our target is to the right
3. Repeat step 1-2 but limiting the pivot to possible ranges

Expected **O(N)** [Explanation in CS3230]

Discussion

1. What is the best case for this Partition algorithm?
   a. Non-randomized?
   b. Randomized?

2. What happens if we do not randomize the pivot of partition?

Discussion

1. What is the best case for this Partition algorithm?
   a. Non-randomized?
   b. Randomized?

Both O(N),

if we ‘luckily’ selected the answer on the first try.

Discussion

2. What happens if we do not randomize the pivot of partition?

Easy to hit near-worse case behaviour.

(Unless the array itself is randomized)

ADT

Introduction to ADT

List ADT

Abstract Data Type (ADT)

An *abstract* data type that is defined by the **operations** you can perform on it.

Implementations of the same ADT can vary

- Some are better than others in different ways

Abstract Data Type (ADT)

Since ADTs are defined by *operations*:

- Implementation can be changed without affecting **functionality** of existing code
  - STL Libraries
- Usually implemented in OOP fashion
  - **Encapsulation**

Common List Array ADT operations

get(i): Gets the *i*-th element from the front. (0-indexed)

search(v): Return the first index which contains *v*, or returns -1/NULL (to indicate failure).

insert(i, v): Insert element *v* at index *i*.

remove(i): Remove the element at index *i*.

PS1 again

Continuous ~~Torture~~ Median

PS1 Tips

PS1B

Is your array to sort *special*?

- Eg: *almost* sorted? Only last number is not sorted?

Do you need **O(N log N)** to sort this *special* array?

- Refer to Tut 02 Qn 2
- No, I don’t mean counting/radix sort :<

PS1C Tip

How will the *sorted array* look like?

1, 1, 1, ..., 1, 3, 3, 3, ..., 3, 5, 5, 5, ..., 5

[0 or more 1s][0 or more 3s][0 or more 5s]

We can use some *counters* to track how many of each number there is.

How do you check what number is at index **K**?

Questions?