CMSC131

Searching and Sorting

https://www.youtube.com/watch?v=ZZuD6iUe3Pc

When was the last time you searched a physical phone book?

- 20% A. In the past year.
- 20% B. Between 1 and 5 years.
- 20% C. Between 6 and 10 years.
- 20% D. More than 10 years.
- 20% E. Never.

Response Counter

Pair and Share

Discuss with your neighbor any strategy you could use when doing things like searching for a name in a physical phone book or searching for a page number in a physical text book.

Searching

If we know ***nothing*** about the way information in a list is organized, then searching that list for a specific item could require looking at every item in the list.

- If we find it we can of course stop searching, but it could be in the last position we check in the worst case.
- If it's not in the list, we won't know that until we've looked at every item.

Is there a way to organize a list that would help us search quicker?

Searching a sorted list

One of the advantages of having a sorted list is that searching it can be quicker!

- On an unordered list we would just start at the first position and look at each value one by one from the first to the last position until we found it or discovered it wasn't there.
- How could we take advantage of a list being sorted to improve our search approach (assuming the list is sorted by the same field that we would use to search for something)?

Concept: BinarySearch

With an **unordered list** we would likely just start to search from the first position, checking to see if it had what we wanted to find and if not moving on to the next position.

- Each check we performed that didn't find the item would only remove 1 candidate.

However, with a **sorted list** if we looked at the item in the middle position and it wasn't what we were looking for, we could then either eliminate all the items to the left or to the right of it (so roughly half).

- Any items to the left of a position would have to be less than it and to the right would be greater.

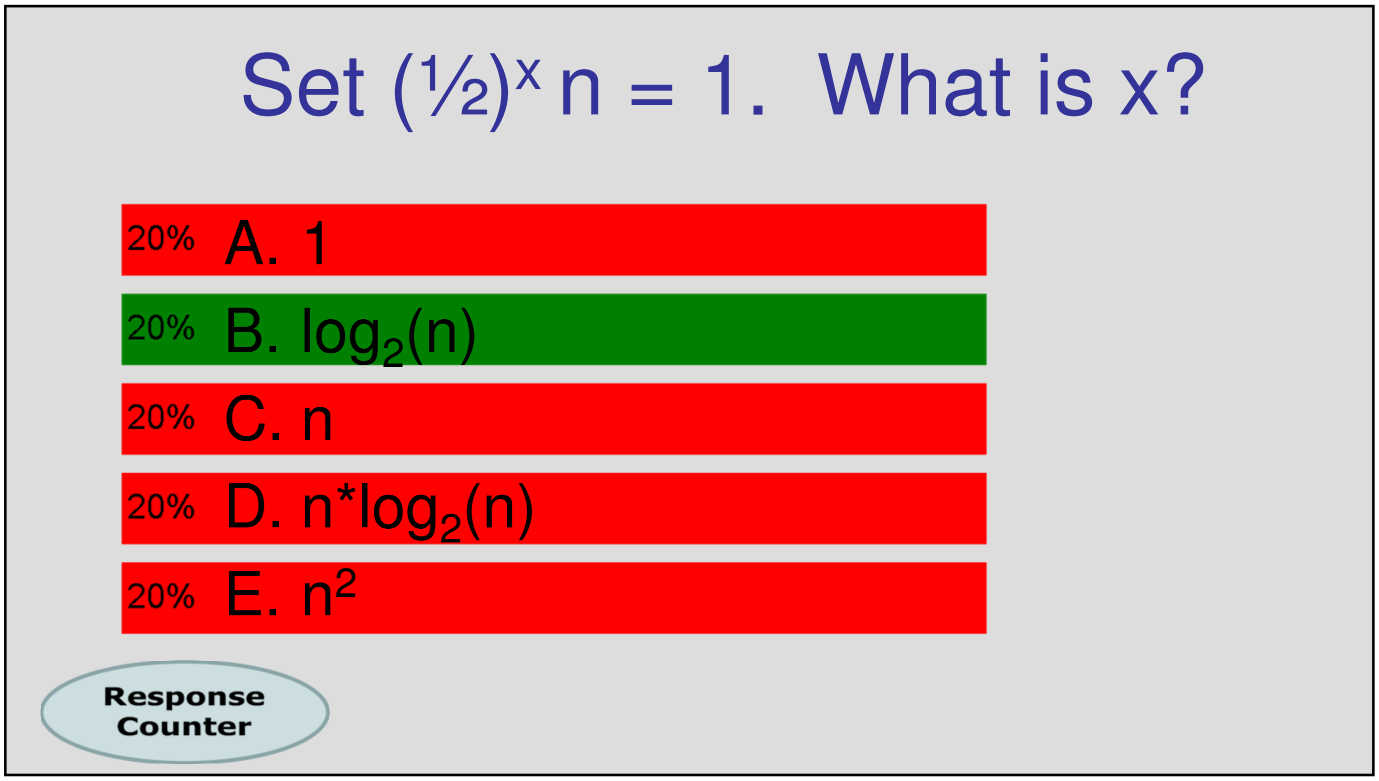
Code: BinarySearch

```
int searchRangeLeft=0, searchRangeRight=values.length-1;
int searchPos;
boolean found=false;
while (!found && searchRangeLeft<=searchRangeRight) {
  searchPos = (searchRangeLeft+searchRangeRight)/2;
  if (values[searchPos]==findVal) {
    found=true;
  } else if (values[searchPos]<findVal) {
    searchRangeLeft=searchPos+1; //Eliminate left half
  } else { //we know values[searchPos]>findVal
    searchRangeRight=searchPos-1; //Eliminate right half
  }
} //When the loop ends, found tells us the answer
  //If the answer is true, searchPos tells us where
```

Efficiency: BinarySearch

Even in the worst case scenario, every time we look at an item, we have to eliminate half of the remaining ones as candidates.

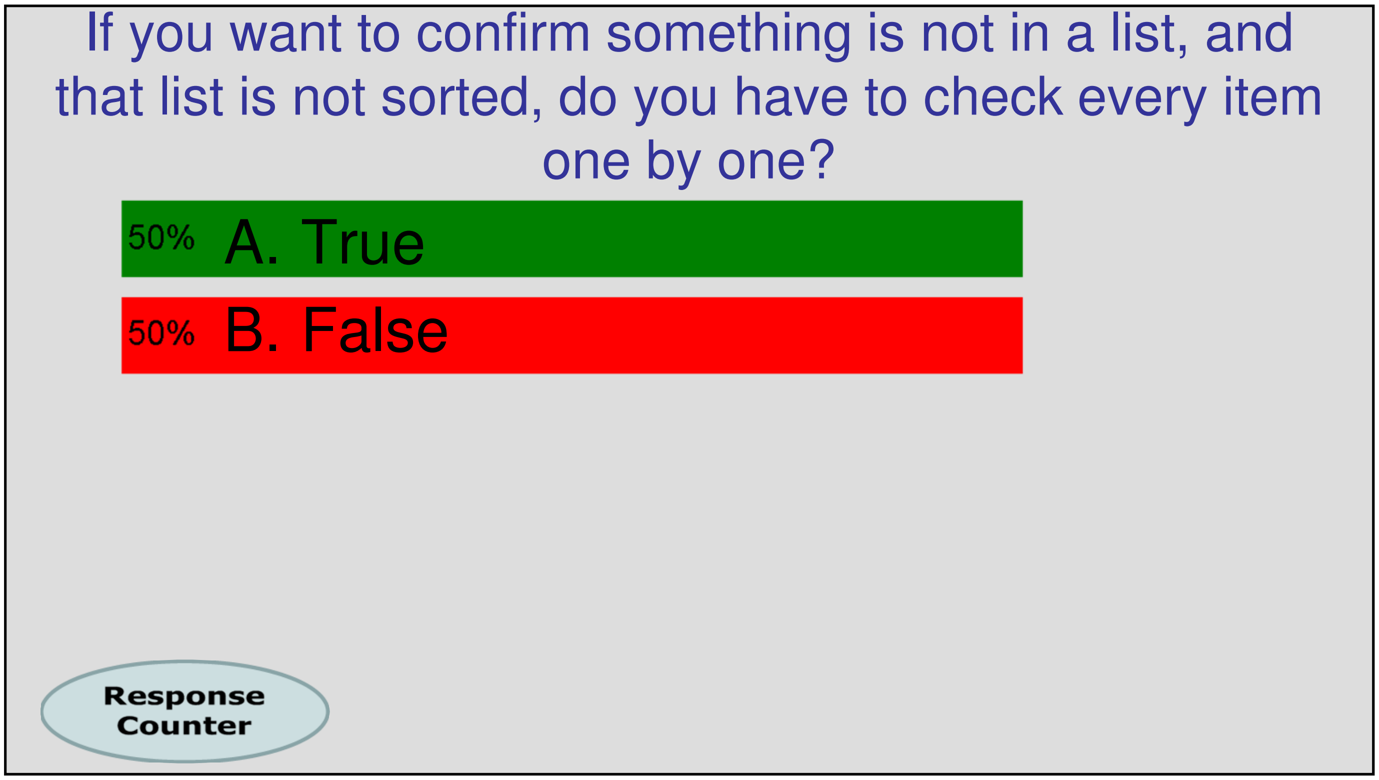
- How many times can you take a half of a half of a half… of n items until you have just one item left?
- Set $(½)^x$ n = 1 and solve for x and you have your answer!



Set $(½)^x n = 1$. What is x?

- 🟥 20% A. 1
- 🟩 20% B. $\log_2(n)$
- 🟥 20% C. n
- 🟥 20% D. $n*\log_2(n)$
- 🟥 20% E. n^2

Response Counter



If you want to confirm something is not in a list, and that list is not sorted, do you have to check every item one by one?

- 🟩 50% A. True
- 🟥 50% B. False

Response Counter

What is a natural way to sort?

Some Real-World Sorting Examples:

- stacks of exams (to make it easier to enter them in a grade book and return in class)
- a hand of playing cards (want to be able to plan your strategy)
- a deck of playing cards (maybe we want to make sure no cards are missing)
- a case of collector cards (so you can buy a case of packs make full sets)
- other examples?

Would any/all of the ways we approach these work as computer algorithms?

Sorting

We've seen the idea of sorting the items in a list based on some specific field with the **`ArrayList`** and the sort method it provides. The **`Arrays`** class in Java also provides the ability to sort the contents of an array.

However, let's look at some of the standard algorithms for implementing our own sorting methods, assuming we have a **compareTo** method available to us…

A simple sort?

As a starting point, let’s narrow our abilities to being able to compare adjacent values and swap them if they are in the wrong order.

https://www.youtube.com/watch?v=lyZQPjUT5B4

An algorithm that supports this (**BubbleSort**) uses pair-wise adjacent comparisons to allow larger values to “bubble towards the top” of a list.

- How could we write a loop that would do a single pass through an array, comparing adjacent values and swapping them if they aren’t in ascending order?
- How could we do this repeatedly and know when to stop?

Comparing neighbors

Assuming that we have an array **values** that stores primitive numeric values and an integer **n** that stores the length of that array, the following would do a single pass comparing adjacent values and swapping them if they are not in ascending order.

```
for (int inner=0; inner<n-1; inner++) {
  if (values[inner]>values[inner+1]) {
    temp = values[inner];
    values[inner]=values[inner+1];
    values[inner+1]=temp;
  }
}
```

How many times?

The number of times we might have to repeat that central code depends on the data. If it's already properly ordered, we really only need to do a single pass and notice that nothing needed to be swapped…

If we do a pass and something needs to be swapped at any point, we might need yet another pass to properly order things based on the new arrangement of values.

- Is it possible that this happens infinitely many times? No! We'll discuss this more soon…

BubbleSort

```
int temp, n=values.length;
boolean anySwaps = true;
while (anySwaps) {
  anySwaps = false;
  for (int inner=0; inner<n-1; inner++) {
    if (values[inner]>values[inner+1]) {
      temp = values[inner];
      values[inner]=values[inner+1];
      values[inner+1]=temp;
      anySwaps = true;
    }
  }
}
```

Why can’t it be infinite?

We could (and in a later course might) prove that if we do a pass through a list of values comparing and swapping neighbors if needed that we end that pass with the largest of the values in the last position.

We can take advantage of this and have our inner loop stop “early” to reflect our logical knowledge of what sub-part of the list still needs to be considered.

Slightly more efficient BubbleSort

```
int temp, unknown=values.length;
boolean anySwaps = true;
while (anySwaps) {
  anySwaps = false;
  for (int inner=0; inner<unknown-1; inner++) {
    if (values[inner]>values[inner+1]) {
      temp = values[inner];
      values[inner]=values[inner+1];
      values[inner+1]=temp;
      anySwaps = true;
    }
  }
  unknown--;
}
```

Efficiency of BubbleSort

In the best case (the list was already in the correct order) the algorithm only makes n-1 comparisons.

In the worst case (the list was in reverse of the correct order) the algorithm will require n outer iterations. The inner loop, however, will get smaller each time.

- (n-1)+(n-2)+…+(n-n) = Sum of the values from 1 to n-1 = $(n^2-n)/2$ = Big-O(n^2)

What algorithm would ***you*** use?

Assume you had some number of playing cards in a row in front of you but they were face down. You probably wouldn't BubbleSort them. What would you do given the following:

- You want to sort them in ascending face value.
- You can only turn at most two cards face up at a time and after you move or don't move them, you have to turn them back face down.
- You can't remember any face values in this example, so if a card isn't face up you do not know its value.

SelectionSort

The basic idea here is to find the largest item in a list, swap it into the last position, then proceed to do that again on the sub-list that does not contain the last position.

- The idea is simple.
- The proof it is correct would be simple.
- The coding is simple.
- The execution time in terms of comparisons as the size of the list grows is also Big-O(n^2).

Code: SelectionSort

```
n = values.length;
for (int posToFill=0; posToFill<n-1; posToFill++) {
  int minSoFarPos = posToFill;
  for (int lookAt=posToFill+1; lookAt<n; lookAt++) {
    if (values[lookAt] < values[minSoFarPos]) {
      minSoFarPos = lookAt;
    }
  }

  if(minSoFarPos != posToFill) {
    int temp = values[posToFill];
    values[posToFill]=values[minSoFarPos];
    values[minSoFarPos]=temp;
  }
}
```

InsertionSort

The basic idea here is to think of the first element of the list as an already-sorted list of size one, and to then take an element from the unsorted part and insert it into the correct position of the already-sorted part of the list.

- The idea and proof is still fairly simple.
- The coding is also still fairly simple.
- The execution time in terms of comparisons as the size of the list grows is also Big-O(n^2) but in practice is still better than Bubble and Selection (about half the number of comparisons).

Code: InsertionSort

```
n = values.length;
for (int posOfVal=1; posOfVal<n; posOfVal++) {
  int temp = values[posOfVal];
  int lookingAt = posOfVal-1;
  while (lookingAt>=0 && values[lookingAt]>temp) {
    values[lookingAt+1]=values[lookingAt];
    lookingAt--;
  }
  values[lookingAt+1] = temp;
}
```

Visualizations of Sorting

There are many different sorting algorithms.

- Some use more of different types of resources.
- Some have more predictable runtimes than others.
- Some are more amenable to parallelization.
- Some are easier to prove things about.
- Some only work on certain types of information

You can see (and hear) some visualizations of common comparison-based sorting algorithms.

www.toptal.com/developers/sorting-algorithms

www.youtube.com/watch?v=kPRA0W1kECg or www.youtube.com/watch?v=14oa9QBT5Js

www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Proving your code works?

How would you confirm that a sorting algorithm you wrote works in all cases?

- While not a formal proof, one thing that you could do is write a JUnit test that generates every possible ordering of n values, runs the sorting algorithm on each, and confirms that the list was correctly sorted.
- In CMSC351 one thing that might be covered is how to formally prove that an algorithm will always give the correct results. Of course when you implement it, you might still want to JUnit test your code…

Faster Sorts

We saw with the binary search, that a “divide and conquer” approach helped speed up a search, assuming we had an ordered list.

Could this type of approach inspire a different sorting algorithm that is faster than what we’ve seen...

Copyright © 2016-2019: Evan Golub