Introduction to Algorithms and Complexity

Operation Code, Kansas City, March 28, 2020 - Social Distancing Edition

Definition

Algorithm: a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of problems or to perform a computation. Algorithms are always unambiguous and are used as specifications for performing calculations, data processing, automated reasoning, and other tasks.

Informal Definition:

Algorithm: a sequence of steps to solve a problem.

Common Classes of Algorithms

Sorting: ordering an unordered list

Searching: finding an element in a data structure.

Trees: searching, traversal, insertion, removal on tree data structures.

Pattern matching and parsing

Hashing: algorithms for cryptography and efficient look up. (and others)

Graphs: algorithms on graphs to solve problems such as shortest path.

Example

Problem: We have an arbitrarily long list of unsorted numbers. We need the list sorted in order from smallest to largest.

One possible solution:

1. Iterate over the list to find the smallest element. Move that to position 1.
2. Iterate over the list from position 2 to position N to find the smallest element. Move that to position 2.
3. Repeat, incrementing the position until position equals N

Example

Let's look at some code!!!

Computational Complexity

Q: Why did the sorting algorithms perform so differently when the size of the list increased?

A: Because some of the algorithms were more efficient than the others.

Q: How can we tell how efficient an algorithms is?

A: By studying and understanding computational complexity.

Computational Complexity

Big “O” notation: expresses the upper bound (worst case) computational cost of an algorithm in terms of “n”.

Algorithm: add 2 numbers

```
function(number1, number2) {
     return number1 + number2
}
```

O(1) - constant

Algorithm: print strings in a list

```
function(list) {
     for(item in list) {
          console.log(item)
     }
}
```

O(n) - linear

Computational Complexity

Algorithm: print all items in a list concatenated to every item in the list.

```
function(list) {
     for(item1 in list) {
          for(item2 in list) {
               console.log(item1 + item2)
          }
     }
}
```

$O(n^2)$ - polynomial

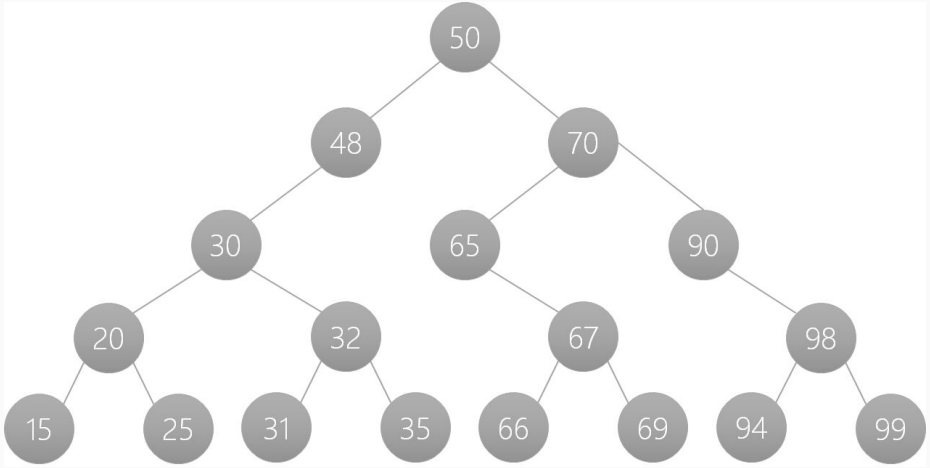
Others …

A logarithmic algorithm – O(logn)
Runtime grows logarithmically in proportion to n.

An exponential algorithm – $O(c^n)$
Runtime grows even faster than polynomial algorithm based on n.

A factorial algorithm – O(n!)
Runtime grows the fastest and becomes quickly unusable for even small values of n

Logarithms WTF? … an example



Find the largest number less than or equal to the input.

Use a “binary search tree”. All nodes to the left of any given node are less than the node. All nodes to the right of a given node are greater than the node.

Each time we traverse to a new node we cut the problem space in half.

This is non-linear, but is logarithmic (the natural logarithm - ln(n))

Back to sorting ...

Why were the Bubble Sort and Insertion Sorts so much slower than the others as n grew in size?

Bubble Sort, Insertion Sort: $O(n^2)$
- n = 10 (100 computations)
- n = 1000 (1 million computations)
- n = 100000 (10 billion computations)

Merge Sort, Quick Sort: $O(n\log_2 n)$
- n = 10 (23 computations)
- n = 1000 (6908 computations)
- n = 100000 (1151292 computations)

Radix Sort: $O(n\log_{10} n)$
- n = 10 (20 computations)
- n = 1000 (3000 computations)
- n = 100000 (500000 computations)

Choosing a Data Structure

As we saw in the binary search tree, choosing the correct data structure for a problem is important.

If we search for an element in array, the time would be O(n). With a binary search tree the time was O(logn)

But …

The time to insert a new element in an array is O(1), to insert into a binary search tree the time is O(n)

Choosing a Data Structure - Example

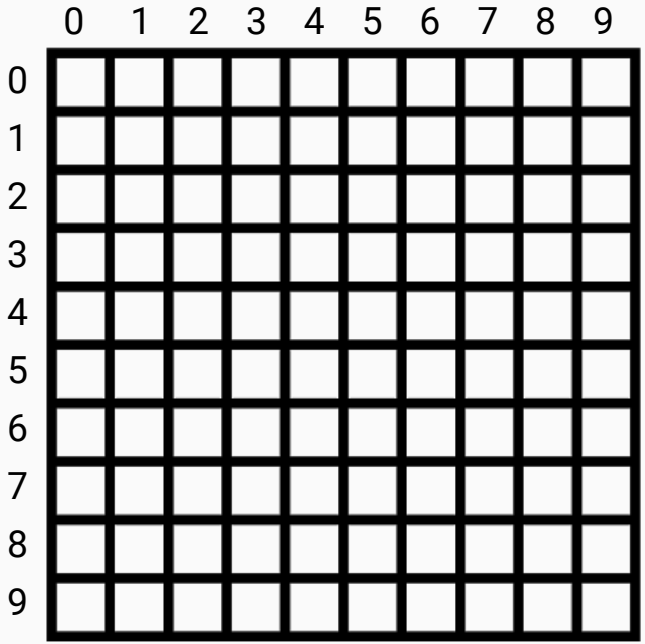
We are running an auto body shop. When a customer comes in they give us their keys. When they leave we give them their keys back.

How will we store their keys?

Option 1: Put their keys in a box

- Inserting keys: O(1)
- Finding keys: O(n)

Choosing a Data Structure - Example



Option 2: Hashing Function

When the customer drops off we ask them the last two digits of their phone number, then we put the keys in the box with the second last digit on the x axis and the last digit on the y axis.

Inserting Keys: O(1)

When the customer picks up we use the same algorithm to find the box with their keys.

Finding Keys: O(1)

Why not ask for the first two digits of their 10 digit phone number?

A Warning About Optimization

For most user facing applications the real bottleneck is human reaction time and Input / Output. (round trips to the internet, a database, etc.)

If you are sorting a list of names in a web application it doesn't really matter if you chose a bubble sort or a quick sort. (unless you are sorting A LOT of names)

Beware of the cost of premature optimization on the complexity and readability of the code.

Additional Resoures

Asymptotic Notation: https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/asymptotic-notation

Dictionary of Algorithms & Data Structures: https://xlinux.nist.gov/dads/

Algorithm Visualizations: https://visualgo.net/en