Data Structures Introduction

Phil Tayco

Slide version 1.5

28-Aug-2021

Introduction

Why are we here?

- Programs are created to make our lives easier
- The more efficient the program, the better they perform to serve us
- Previous classes focus on how to create programs. Here, we find ways to measure their performance and analyze how to make them more efficient

Introduction

What is efficient?

- At first, it seems fast results means efficient programming. Is that so?
- Does fewer lines of code for your program mean efficient as well?
- There are many factors to consider to measure efficiency besides speed and lines of code
- To understand them, lets look at a typical user function: a simple search for a record

Introduction

Best case/Worst case

- How long would it take to find an item in an unordered list?
- At best, you get it on the first try and at worst, you go through the entire list (particularly if the search item does not exist in the list)
- In an unordered list, checking an item in a specific location is arbitrary - it doesn’t matter which element you select first
- Can the situation be modified to improve the search time?

Introduction

Yes! Sort the list!

- Sorting the records significantly improves the search using the binary search algorithm
- Look in the middle of the list. If you found it, great. Otherwise, look in the middle of the section of the list where the record should be
- In a list of 1000 unsorted records, worst case is 1000 checks. If sorted, worst case is 11!

Introduction

Sorting the list, though…

- This process takes additional time to perform
- Begs the question: Is the process of sorting and then searching faster than searching an unsorted list?
- It turns out as the size of the list gets larger, the fastest sort takes longer than just searching the unsorted list
- This implies searching an unsorted list is better but if you sorted the list beforehand, the binary search speed is significant

Introduction

It doesn’t just depend on performance…

- We could sort the data and save it. Pre-sorting the records significantly reduces search performance time, but requires memory space and time to store the indexed records
- Next, if records are added or deleted, the sorted records needs to maintain their order to keep the search time optimal
- Is there a configuration and algorithm that is ideal in supporting all of these factors?

Introduction

That’s the goal!

- **In this class we will look at different structures and algorithms that provide the best measures of efficiency based on the factors of performance time, storage space and data maintenance in different situations**

Measuring Efficiency

When you design a solution…

- What do you use to measure how effective it is?
- Do you consider how it will do in other situations? (capacity and maintenance…)
- We address the performance factor using a notation that can be consistently and systematically applied – this is known as "Big O"

Big O Notation

O, the magnitude…

- Often, algorithms are applied on data sets (a list of records, coordinates on a map, genetic sequences…)
- An algorithm will perform a certain way that we can categorize on a set amount of data
- This categorization helps us see how that logic of the algorithm performs as the size of the data set increases

Big O Notation

This shows that number of lines of code is not an indicator of algorithm performance:

```
int visitCount = 0;
for (int loc = 0; loc < coffeeShops.length; loc++)
   if (coffeeShops[loc].visited == false)
   {
       coffeeShops[loc].visited = true;
       visitCount++;
   }
```

- Number of lines of code to measure an algorithm is not useful. This has 5 lines of code, but will vary in performance based on the size of the coffeeShops array (the more coffee shops, the longer it will take)

Big O Notation

What is the real unit of measure?

- Algorithms will use many kinds of operations. Some operations take more time or memory than others
  - Function call (power(x, y);)
  - Conditional expression (x > y)
  - Assignment (z = 5)
  - Mathematical operation (area = length * width)
- When analyzing an algorithm, we first want to identify the unit of measure by selecting an operation considered to be significant

Big O Notation

Significant Operations

- Often this is a comparison operation or set of assignment operations (like a swap, which is a 3 assignment operations)
- Question: In the code example, how many comparison operations are performed?
- Answer: It depends on the size of the array (2 * coffeeShops.length)

Big O Notation

So we reduce the total count of key ops?

- Not quite yet. Fine tuning the algorithm to reduce the number of significant operations that take place can be done later – we first want to categorize the performance
- Big O measures performance from the perspective of "as the list gets larger"
- We also usually start with looking at worst case scenarios the data set can be

Big O Notation

So how did that code example measure?

- The worst case scenario of the data set is that all coffee shops are initially unvisited
- By using the comparison operation as the unit of measure, we see that the number of compares is proportional to the size of the list
- Given a list of size n, the number of compares for the algorithm in the worst case is 2 * n
- We generalize this performance into a category by saying the Big O is “O(n)” – this means as the list size grows, the number of comparisons proportionally rises linearly

Big O Notation

Big O Types

- There are four major ways to categorize the Big O performance of an algorithm in this class
- To see what they are, consider the program example which is essentially recording a count of the number of unvisited coffee shops
- Suppose we also want to record that count in a database
- There are many ways to do this, some more effective than others. Big O provides a standard notation to categorize it

Big O Notation

Algorithm 1

- Start at coffee shop 1. If it has already been visited, go to the next coffee shop. Repeat until all coffee shops are considered visited
- If the current shop has not been visited, stop the visiting process (i.e. exit the loop)
- Add 1 to the coffee shop count
- Log on to the database and update the coffee shop count record
- Return to the first coffee shop and repeat the visiting process (i.e. go back to the first step of this algorithm)

Big O Notation

Code for Algorithm 1

```
int visitCount = 0;
int loc;
while(true)
{
   for (loc = 0; loc < coffeeShops.length; loc++)
        if (coffeeShops[loc].visited == false)
        {
               coffeeShops[loc].visited = true;
               break;
        }
   updateDatabase(++visitCount);
   if (loc == coffeeShops.length)
        break;
}
```

Big O Notation

Algorithm 2

- Visit all coffee shops starting at shop 1
- If the current shop has not been visited, mark it as visited and add 1 to the coffee shop count
- After all coffee shops have been examined, log on to the database and update the coffee shop count record

Big O Notation

Code for Algorithm 2

```
int visitCount = 0;
for (int loc = 0; loc < coffeeShops.length; loc++)
      if (coffeeShops[loc].visited == false)
      {
            coffeeShops[loc].visited = true;
            visitCount++;
      }
updateDatabase(visitCount);
```

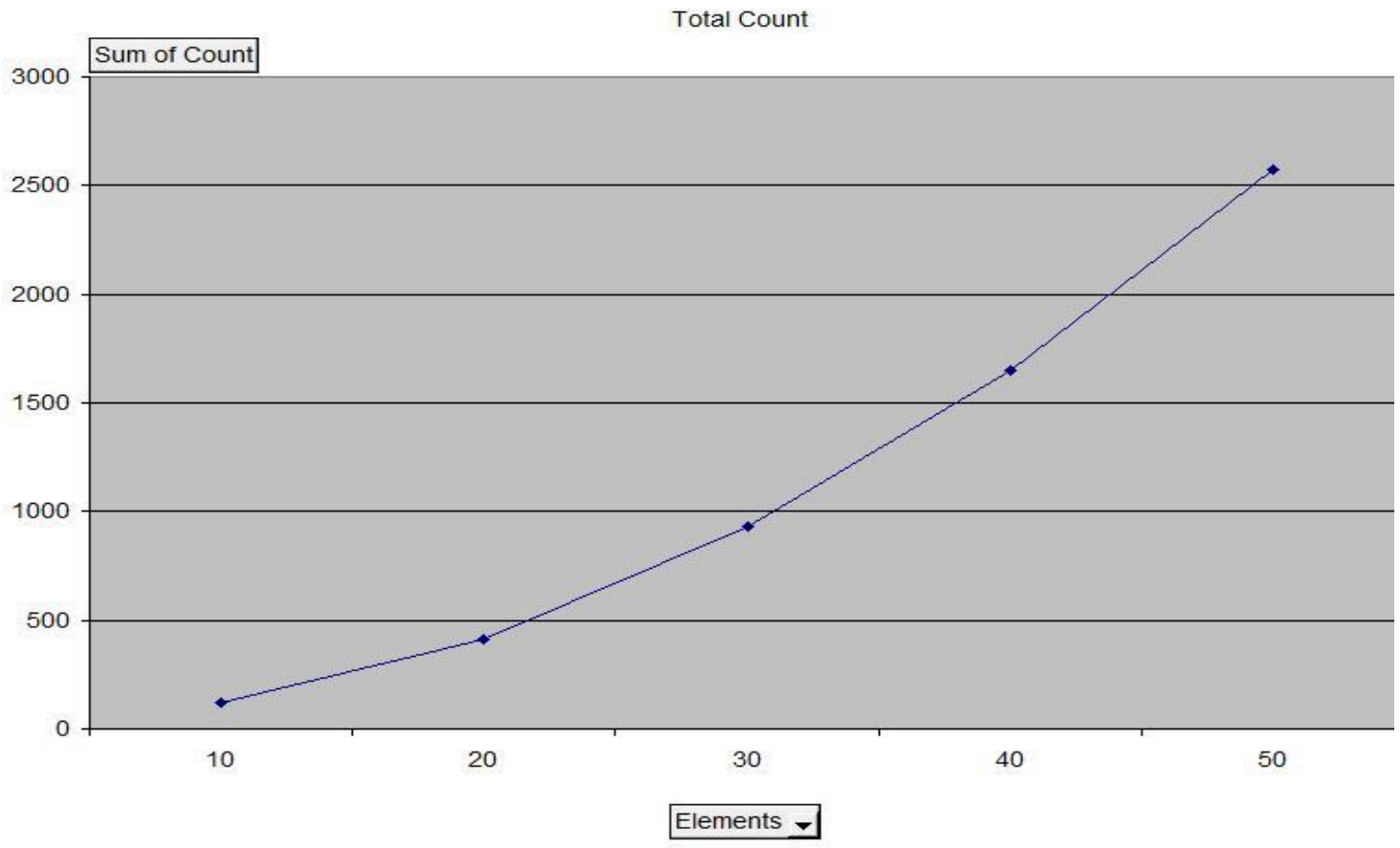
Big O Notation

Analysis

- It’s intuitively clear that the second algorithm is more efficient than the first, but let’s use Big O to formally confirm this
- We must first determine an operation type. Usually, this is the most expensive operation to consider
- The worst case scenario in this case is if all coffee shops were unvisited – we’ll apply this scenario when analyzing both algorithms
- Using the comparison operation and the worst case scenario, the counts for algorithm 1 are:
  - 10 shops: 3 + 5 + 7 + … + 19 + 21 = 120
  - 20 shops : 3 + 5 + 7 + … + 39 + 41 = 410
  - 30 shops : 3 + 5 + 7 + … + 59 + 61 = 930

Big O Notation

Algorithm 1 Plot



Big O Notation

Exponential growth

- Notice with this graph that as the number of elements in the list increases, the count of operations grows exponentially
- A list of n elements will have something to the effect of $(n^2 + C)$ comparison counts
- The exact formula can be derived but what matters more at this point is the **rate of growth**
- Big O categorizes this exponential growth as $O(n^2)$

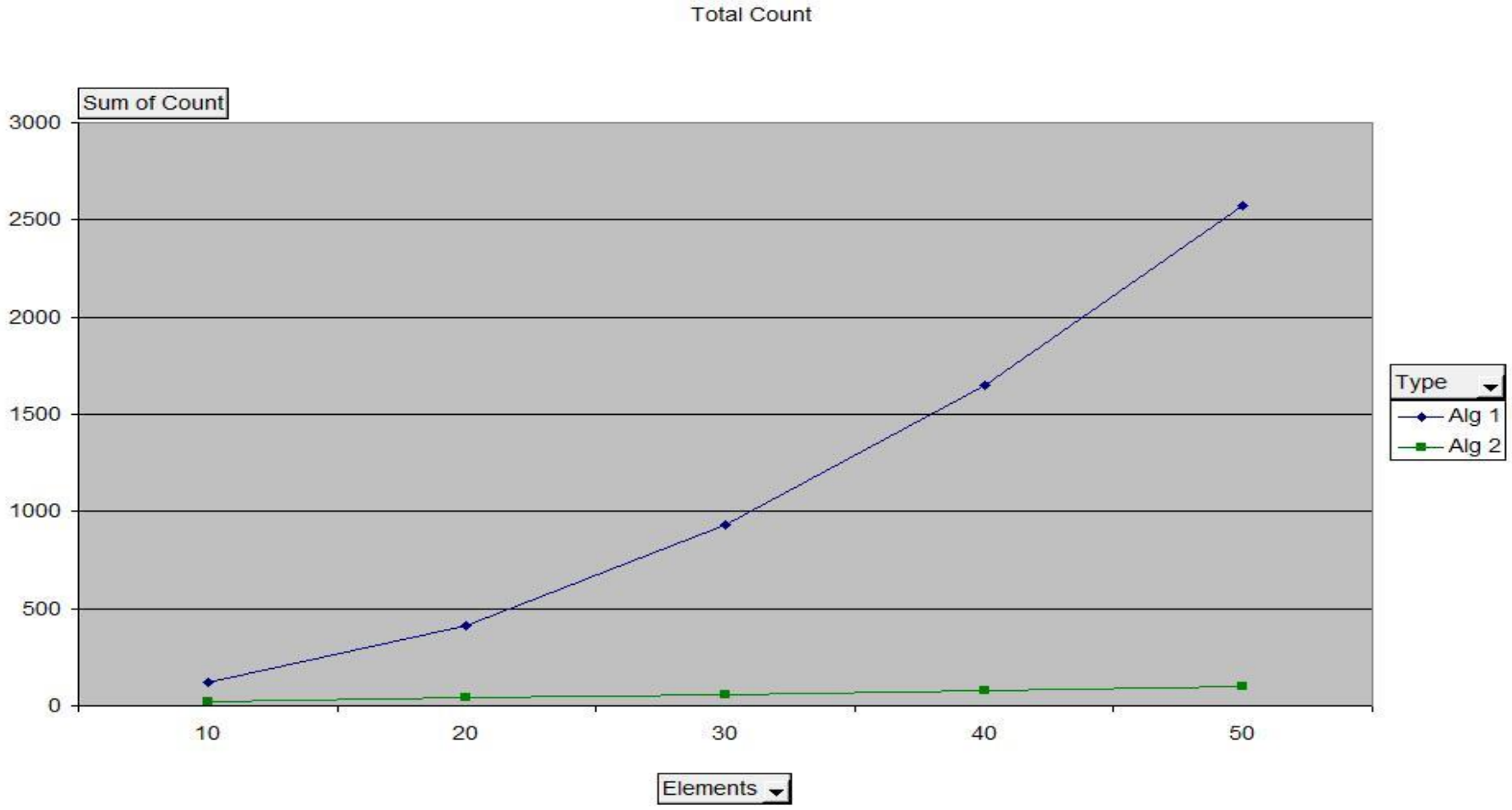
Big O Notation

What about algorithm 2?

- Using the same operation and worst case scenario, the counts for algorithm 2:
  - 10 elements: 2 * 10 = 20
  - 20 elements: 2 * 20 = 40
  - 30 elements: 2 * 30 = 60
  - n elements: 2 * n
- This count is significantly smaller than algorithm 1

Big O Notation

Algorithm 1 and 2 Plots



Big O Notation

Further analysis

- The rate of growth in relation to size n is linear. We capture this linear growth as O(n)
- Comparing between orders makes the actual counts and formulas less significant
  - O(n + 1000) is still better than $O(n^2)$ because as n increases, the counts for linear growth will eventually be significantly better than exponential growth
- Question 1: What are the Big Os of the 2 algorithms if the operation to consider is calling the database?
- Question 2: Do the Big Os change if we consider best case scenario (i.e. if all coffee shops were already visited)?

Big O Notation

The 4 main Big O groups

- From worst to best:
  - Exponential: $O(n^2)$
  - Linear: O(n)
  - Logarithmic: O(log n)
  - Constant: O(1)
- Logarithmic we will see more later. This plot line has a flatter growth rate than linear
- Constant is ideal as it indicates that no matter how large the list becomes, the number of operations performed is the same
- Some algorithms at lower values of n will have better counts than the Big O suggests. Remember that the measure is not for all values of n, but to show you performance as n increases

Big O Notation

Algorithm analysis procedure

- Identify the operation type to use for your unit of measure
- Identify the scenario(s) you want to examine (worst case, best case and/or average)
- Examine the algorithm performance focusing on that unit of measure and how its value changes as the data set the algorithm is applied to gets larger
- Determine its Big O and repeat the process with other algorithms using the same operation type and scenario

Practice

What is the Big-O category of the following algorithms when using the comparison operator as the unit of measure?

- Worst case search for an element in an unsorted list
- Worst case search for the smallest value in a sorted list
- Finding the average value of numbers in a sorted list

Study hints:

- Try coding the algorithm and doing a count of how many times the key operator repeats
- Remember you are looking at Big-O categories, not actual counts
- Look up some of the code you've written for other classes. Analyze them in terms of Big-O

Data Structures

Now that we know how to measure performance, what types of algorithms do we tend to measure?

- When all is said and done, all programs tend to focus on performing four main functions
  - Search: Finding a record of significance
  - Insert: Adding new data to the record set
  - Update: Performing a search and making a change to that record in the set
  - Delete: Removing data from the record set

Data Structures

What will we do in this class?

- We will look at data structures and evaluate them based on the performance of these function types while also considering storage and maintenance factors
- We will find that certain structures work with different situations. Our goal is not only to memorize them, but more importantly understand the evaluation process to make us better programmers and improve our eye towards efficiency