Data Structues 1.1

Garrett Giles

July 16, 2022

What is a Data Structure?

A data structure (DS) is a way of organizing data so that it can be used effectively.

Why Data Structures?

They are essential ingreatients in creating fast and powerful algorithms.
They help to manage and organize data.
They make code cleaner and easier to understand.

Abstract Data Types vs. Data Structures.

Abstract Data Type

An abstract data type (ADT) is an abstraction of a data structure which provides only the interface to which a data structure must adhere to.
The interface does not give any specific details about how something should be implemented or in what programming language.

Table 1: Examples

Abstraction (ADT)	**Implementation (DS)**
List	Dynamic Array Linked List
Queue	Linked List based, Array Based Queue, Stack based Queue
Map	Tree Map, Hash Map / Hash Table
Vehicle	Golf Cart, Bicycle, Smart Car

Computational Complexity Analysis

Complexity Analysis

As programmers, we often find outselves asking the same two questions over and over again:
How much time does this algorithm need to finish?
How much space does this algorithm need for its computation?

Big-O Notation

Big-O Notation gives and upper bound of the complexity in the worst case, helping to quantify performance as the input size becomes arbitrarily large.

n - The size of the input

Table 2: Complexities ordered in from smallest to largest.

Term	Notation
Constant Time	$O(1)$
Logarithmic Time	$O(log(n))$
Linear Time	$O(n)$
Linearithmic Time	$O(nlog(n))$
Quadric Time	On^2
Cubic Time	$O(n^3)$
Exponential Time	$O(b^n), b > 1$
Factorial Time	$O(n!)$

Big-O Properties

$$\mathbf{O(n + c) = O(n)}$$

$$\mathbf{O(cn) = O(n), c > 0}$$

Let f be a function that describes the running times of a particular algorithm for an input size of n:

$$\mathbf{f(n) = 7log(n)^3 + 15n^2 + 2n^3 + 8}$$

$$\mathbf{O(f(n)) = O(n^3)}$$

Big-O Examples

The following run in constant time: O(1)

Algorithm 1: The following run in constant time: O(1)

$a := 1$
$b := 2$
$c := a + 5 * b$
$i := 0$
while $i < 11$ **do**
 | $i = i + 1$
end while

Algorithm 2: The following run in linear time: $O(n)$

$i := 0$
while $i < n$ **do**
  i = i + 1 ; // $f(n) = n$
  // $O(f(n)) = O(n)$
end while

Algorithm 3: The following run in linear time: $O(n)$

$i := 0$
while $i < n$ **do**
  $i = i + 3$; // $f(n) = n/3$
  // $O(f(n)) = O(n)$
end while

Both of the following run in quadratic time. The first may be ovvious since n work done n times is $n * n = O(n^2)$, but what about the second one?

Algorithm 4: $f(n) = n * n = n^2, O(f(n)) = O(n^2)$

for $i := 0; i < n; i = i + 1$ **do**
  for $j := 0; j < n; j = j + 1$ **do**
  end for
end for

Algorithm 5: Here, 0 in the second loop is changed into **i**

for $i := 0; i < n; i = i + 1$ **do**
  for $j := i$; $j < n$; $j = j + 1$ **do**
  end for
end for

For a moment just focus on the second loop. Since i goes form [0,n) the amount of looping done is directly determined by what i is. Remark that if $\mathbf{i = 0}$, we do n work, if $i = 1$, we do $n - 1$ work, if $i = 2$, we do $n - 2$ work, etc…
So the question then becomes what is: $(n)+(n-1)+(n-2)+(n-3)+\cdots+3+2+1$? Remakably this turns out to be $n(n + 1)/2$, so $O(n(n + 1)/2) = O(n^2/2 + n/2) = O(n^2)$
Suppose we have a sorted array and we want to find the index of a particular value in the array, if it exists. What is the time complexity of the following algorithmm?

Algorithm 6: Answer: $O(log2(n)) = O(log(n))$

```
low := 0
high := n-1
while low <= high do
    mid := (low + high) / 2
    if array[mid] == value then
        return mid
    else if array[mid] < value then
        low = mid + 1
    else if array[mid] > value then
        high = mid - 1
    return -1                                   ▷ value not found
end while
```

Table 3: Big-O common examples

Finding all the subsets of a set	$O(2^n)$
Finding all permutations of a string	$O(n!)$
Sorting using mergsort	$O(nlog(n))$
Iterating over all the cells in a matrix of size n by m	$O(nm)$