Topic: BigO concept

March 24, 2020

Nan Jia

Topics

- Objectives
- Definition
- Vocabulary
- The growth of functions
- Examples

Objectives

- Big O notation is under algorithm analysis.
    - It ranks an ALGO's efficiency. The most important topic is the close relation of the code efficiency
    - It is determined by leading variables, such as $2^n > n^2$
    - General skills of calculating the Big O

Definition

If

Algorithm A requires time proportional to f(n)

Algorithm *A* is said to be **order *f*(*n*)**, which is denoted as **O(*f*(*n*))**. The function *f*(*n*) is called the algorithm's **growth-rate function**. Because the notation uses the capital letter O to denote ***order***, it is called the **Big O notation**.

Note: Definition of the order of an algorithm

Algorithm *A* is order $f(n)$ — denoted $O(f(n))$ — if constants k and n_0 exist such that *A* requires no more than $k * f(n)$ time units to solve a problem of size $n \geq n_0$.

Terms, terms, terms...

- An algorithm's execution time is related to the number of operations it requires. This is usually expressed in terms of the number, *n*, of items the algorithm must process.

```
Node<ItemType>* curPtr = headPtr;
while (curPtr != nullptr)
{
    cout << curPtr->getItem() < endl;
    curPtr = curPtr->getNext();
}  // end while
```

$\leftarrow$ *1 assignment*
$\leftarrow$ *$n + 1$ comparisons*

$\leftarrow$ *1 write*
$\leftarrow$ *1 assignment*

- The lines under while loops will run its time unit per loop.
- The program above requires time proportional to n

- Algorithm efficiency is typically a concern for large problems only. The time requirements for small problems are generally *not large enough to matter*. Thus, our analyses assume large values of n.

The growth of functions used in bigO estimates

O(1): TR is constant and independent of problem's size n

O($\log_2$(n)): solves small constant fraction first; n^2: only double TR

O(n): TR increases with the size of the problem; **linear**

O(n $\cdot$ $\log_2$(n)): divides problem and solves separately

O(n^2): mostly has two nested loops; **quadratic**

O(n^3): mostly has three nested loops; **cubic**

O(2^n): not practical; **exponential**

small problems

from slow to fast by means of time requirement(growth rate)

Data structure and ALGO examples

O(1): array, stack, queue, and matrix (access/peek)

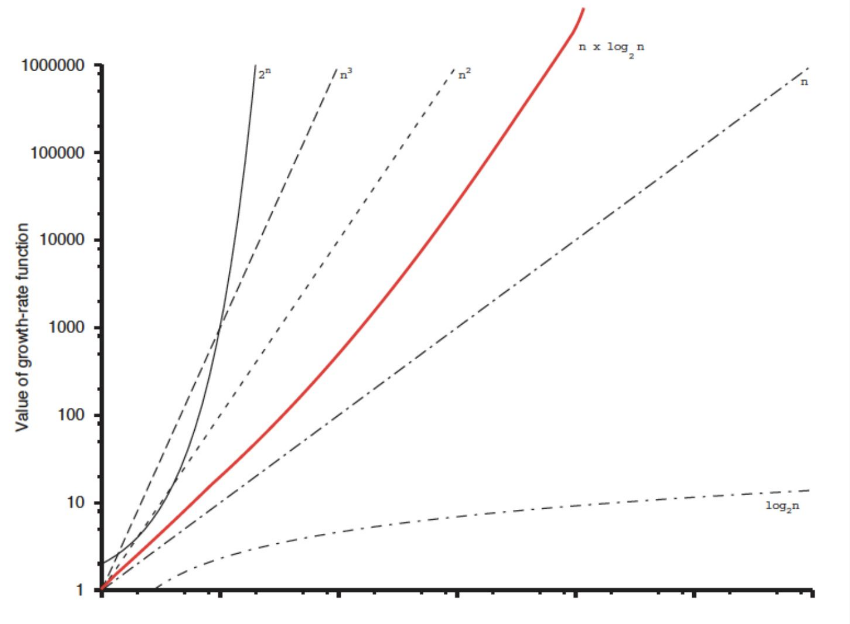
O($\log_2$(n)): recursive binary search ALGO

O(n): for loop, while, and etc.

O(n · $\log_2$(n)): quick sort and merge sort etc.

O(n^2): bubble/insertion/selection sort

but, no one ALGO is fastest in all cases



Data structure	Access /peek	Search	Insert /push	Delete /pop	Traverse
Linear					
Array	O(1)	O(n)	O(1)	O(n)	O(n)
Ordered array	O(1)	O(logn)	O(n)	O(n)	O(n)
Linked list	O(n)	O(n)	O(1)	O(n)	O(n)
Ordered linked list	O(n)	O(n)	O(n)	O(n)	O(n)
Matrix	O(1)	O(n^2)	O(1)	O(n^2)	O(n^2)
Stack	O(1)	O(n)	O(1)	O(1)	O(n)
Queue	O(1)	O(n)	O(1)	O(1)	O(n)
Non-Linear					
Tree (worst case)	O(n)	O(n)	O(n)	O(n)	O(n)
Tree (balanced)	O(logn)	O(logn)	O(logn)	O(logn)	O(n)
Binary heap	O(logn)	O(logn)	O(logn)	O(logn)	O(n)
Trie	O(n)	O(n)	O(n)	O(n)	O(n)
Graph	O(n)	O(n)	O(1)	O(n)	O(n)

COPYRIGHT © LAVIVIENPOST.COM

Algorithms and use caces	Time	Space	When to choose
Sorting			
Bubble, Insert, Selection	O(n^2)	O(1)	Simple sort
Mergesort	O(nlogn)	O(n)	Stable sort
Quicksort	O(n^2)	O(logn)	It depends
Searching			
Linear search	O(n)	O(1)	Find element in non-sorted list
Binary search	O(logn)	O(1)	Find element in sorted list
Recursion			
Factorial	O(n)	O(n)	Numbers, math
Perm of array, string	O(nxn!)	O(nxn!)	Permutation
All subset of array	O(2^n)	O(2^n)	All subset
Dynamic Programming			
Fibonacci	O(n)	O(n)	Numbers, math
Num of paths in matrix	O(n^2)	O(n^2)	Number of ways
Knapsack	O(n^2)	O(n^2)	Max, min, longest
Bits, Num & Math			
Bits	O(n)	O(1)	Find missing, odd, single nums
Decimal to binary, hex	O(n)	O(1)~O(n)	Numbers
Power of 2	O(n)	O(1)	Math

Calculation/thinking

Question 1 How many comparisons of array items do the following loops contain?

```
for (j = 1; j <= n-1; j++)
{
   i = j + 1;
   do
   {
      if (theArray[i] < theArray[j])
         swap(theArray[i], theArray[j]);
      i++;
   } while (i <= n);
}  // end for
```

$O(n^2)$

```
for (j = 1; j <= n-1; j++)
{
   i = j + 1;
   do
   {
      if (theArray[i] < theArray[j])
         swap(theArray[i], theArray[j]);
      i++;
   } while (i <= n);
}  // end for
```

n-1

1

1^{st}: (n-2) iterations
2^{nd}: (n-3) iterations
:
(n-1): 1 time

$$(n-1)\,[\,1 + (n-1)\,] \Rightarrow n^2$$

Same levels sum up, and different levels multiple. And, the final result is determined by power of n.

Sources:

- Carrano, Frank M., et al. Data Abstraction & Problem Solving with C : Walls & Mirrors. Sixth edition / Frank M. Carrano, University of Rhode Island, Timothy Henry, University of Rhode Island.. ed., Pearson, 2013.
- https://www.freecodecamp.org/news/all-you-need-to-know-about-big-o-notation-to-crack-your-next-coding-interview-9d575e7eec4/
- https://www.lavivienpost.com/top-interview-questions-and-big-o-notation-cheat-sheets/

Thank you!!!