Data Structures/Asymptotic Notation

Asymptotic Notation

Introduction

There is no single data structure that offers optimal performance in every case. In order to choose the best structure for a particular task, we need to be able to judge how long a particular solution will take to run. Or, more accurately, you need to be able to judge how long two solutions will take to run, and choose the better of the two. We don't need to know how many minutes and seconds they will take, but we do need some way to compare algorithms against one another.

Asymptotic complexity is a way of expressing the *main component* of the cost of an algorithm, using idealized (not comparable) units of computational work. Consider, for example, the algorithm for sorting a deck of cards, which proceeds by repeatedly searching through the deck for the lowest card. The asymptotic complexity of this algorithm is the *square* of the number of cards in the deck. This quadratic behavior is the main term in the complexity formula, it says, e.g., if you double the size of the deck, then the work is roughly quadrupled.

The exact formula for the cost is more complex, and it contains more details than we need to understand the essential complexity of the algorithm. With our deck of cards, in the worst case, the deck would start out reverse-sorted, so our scans would have to go all the way to the end. The first scan would involve scanning   cards, the next would take   , etc. So the cost formula is   . Generally, letting   be the number of cards, the formula is   , which equals   . But the   term dominates the expression, and this is what is key for comparing algorithm costs. (This is in fact an *expensive* algorithm; the best sorting algorithms run in sub-quadratic time.)

Asymptotically speaking, in the limit as   tends towards infinity,   gets closer and closer to the pure quadratic function   . And what difference does the constant factor of   make, at this level of abstraction? So the behavior is said to be   .

Now let us consider how we would go about *comparing* the complexity of two algorithms. Let   be the cost, in the worst case, of one algorithm, expressed as a function of the input size   , and   be the cost function for the other algorithm. E.g., for sorting algorithms,   and   would be the maximum number of steps that the algorithms would take on a list of   items. If, for all values of   ,   is less than or equal to   , then the algorithm with complexity function   is strictly faster. But, generally speaking, our concern for computational cost is for the cases with large inputs; so the comparison of   and   for small values of   is less significant than the "long term" comparison of   and   , for   larger than some threshold.

Note that we have been speaking about *bounds* on the performance of algorithms, rather than giving exact speeds. The actual number of steps required to sort our deck of cards (with our naive quadratic algorithm) will depend upon the order in which the cards begin. The actual time to perform each of our steps will depend upon our processor speed, the condition of our processor cache, etc., etc. It's all very complicated in the concrete details, and moreover not relevant to the essence of the algorithm.

The O Notation

Definition

The   (pronounced *big-oh*) is the formal method of expressing the upper bound of an algorithm's running time. It's a measure of the longest amount of time it could possibly take for the algorithm to complete. We can assume that it represents the "worst case scenario" of a program.

More formally, for non-negative functions,   and   , if there exists an integer   and a constant   such that for all integers   ,   , then   is big O of   . This is denoted as   . If graphed,   serves as an upper bound to the curve you are analyzing,   .

Note that if   can take on finite values only (as it should happen normally) then this definition implies that there exists some constant   (potentially larger than   ) such that for all values of   ,   . An appropriate value for   is the maximum of   and   .

Theory Examples

So, let's take an example of Big-O. Say that   , and   . Can we find a constant   , so that   ? The number   works here, giving us   . For any number   greater than   , this will still work. Since we're trying to generalize this for large values of   , and small values   aren't that important, we can say that   is generally faster than   ; that is,   is bound by   , and will always be less than it.

It could then be said that   runs in   time: "f-of-n runs in Big-O of n-squared time".

To find the upper bound - the Big-O time - assuming we know that   is equal to (exactly)   , we can take a few shortcuts. For example, we can remove all constants from the runtime; eventually, at some value of   , they become irrelevant. This makes   . Also, for convenience of comparison, we remove constant multipliers; in this case, the   . This makes   . It could also be said that   runs in   time; that lets us put a tighter (closer) upper bound onto the estimate.

Practical Examples

       : printing a list of   items to the screen, looking at each item once.

", taking a list of items, cutting it in half repeatedly until there's only one item left.

: taking a list of n items, and comparing every item to every other item.

Big-Omega Notation

For non-negative functions,   and   , if there exists an integer   and a constant   such that for all integers   ,   , then   is   . This is denoted as
.

This is almost the same definition as Big Oh, except that   , this makes   a lower bound function, instead of an upper bound function. It describes the **best that can happen** for a given data size.

Theta Notation

For non-negative functions,   and   ,   is theta of   if and only if   and   . This is denoted as   .

This is basically saying that the function,   is bounded both from the top and bottom by the same function,   .

The theta notation is denoted by Q.

Little-O Notation

For non-negative functions,   and   ,   is little o of   if and only if   , but   . This is denoted as   .

This represents a loose bounding version of Big O.   bounds from the top, but it does not bound the bottom.

Little Omega Notation

For non-negative functions,   and   ,   is little omega of   if and only if   , but   . This is denoted as   .

Much like Little Oh, this is the equivalent for Big Omega.   is a loose lower boundary of the function   ; it bounds from the bottom, but not from the top.

How asymptotic notation relates to analyzing complexity

Temporal comparison is not the only issue in algorithms. There are space issues as well. Generally, a trade off between time and space is noticed in algorithms. Asymptotic notation empowers you to make that trade off. If you think of the amount of time and space your algorithm uses as a function of your data over time or space (time and space are usually analyzed separately), you can analyze how the time and space is handled when you introduce more data to your program.

This is important in data structures because you want a structure that behaves efficiently as you increase the amount of data it handles. Keep in mind though that algorithms that are efficient with large amounts of data are not always simple and efficient for small amounts of data. So if you know you are working with only a small amount of data and you have concerns for speed and code space, a trade off can be made for a function that does not behave well for large amounts of data.

A few examples of asymptotic notation

Generally, we use asymptotic notation as a convenient way to examine what can happen in a function in the worst case or in the best case. For example, if you want to write a function that searches through an array of numbers and returns the smallest one:

```
function find-min(array a[1..n])
  let j :=
  for i := 1 to n:
    j := min(j, a[i])
  repeat
  return j
end
```

Regardless of how big or small the array is, every time we run **find-min**, we have to initialize the *i* and *j* integer variables and return *j* at the end. Therefore, we can just think of those parts of the function as constant and ignore them.

So, how can we use asymptotic notation to discuss the **find-min** function? If we search through an array with 87 elements, then the *for* loop iterates 87 times, even if the very first element we hit turns out to be the minimum. Likewise, for   elements, the for loop iterates   times. Therefore we say the function runs in time   .

What about this function:

```
function find-min-plus-max(array a[1..n])
  // First, find the smallest element in the array
  let j :=    ;
  for i := 1 to n:
    j := min(j, a[i])
  repeat
  let minim := j

  // Now, find the biggest element, add it to the smallest and
  j :=    ;
  for i := 1 to n:
    j := max(j, a[i])
  repeat
  let maxim := j

  // return the sum of the two
  return minim + maxim;
end
```

What's the running time for **find-min-plus-max**? There are two *for* loops, that each iterate   times, so the running time is clearly   . Because   is a constant, we throw it away and write the running time as   . Why can you do this? If you recall the definition of Big-O notation, the function whose bound you're testing can be multiplied by some constant. If   , we can see that if   , then the Big-O condition holds. Thus   . This rule is general for the various asymptotic notations.

Data Structures

[Introduction](#) - [Asymptotic Notation](#) - [Arrays](#) - [List Structures & Iterators](#)
[Stacks & Queues](#) - [Trees](#) - [Min & Max Heaps](#) - [Graphs](#)
[Hash Tables](#) - [Sets](#) - [Tradeoffs](#)

Retrieved from "https://en.wikibooks.org/w/index.php?title=Data_Structures/Asymptotic_Notation&oldid=3440026"

This page was last edited on 2 July 2018, at 18:01.

Text is available under the Creative Commons Attribution-ShareAlike License.; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy.