Comparing Function Growth Rates

Discussion

To compare the efficiency of two algorithms, we must find a mathematical function that defines the amount of time or memory the algorithm requires, which depends upon some variable n. The variable n is, in some way, is a measure of the input(s) to the algorithm, and is often the size of a data structure. Once we have calculated such functions for two algorithms, we need to be able to compare the growth rates of those two functions to determine which of the algorithms is more efficient.

To compare functions, we must define an ordering. Although total orderings are often preferable, a partial ordering is sufficient. The simplest way to partially order functions would be as follows:

$$f \leq g \Leftrightarrow \forall x \;\; f(x) \leq g(x)$$

The problem with such an ordering is that many function pairs are not comparable, for example x^2 and x^3 cannot be compared because x^3 is greater than x^2 for positive numbers but less for negative ones.

Of course, with the functions that define algorithm performance, we are never interested in their behavior for negative values. Moreover, we also are not interested in their behavior for small values. By using Big-O to define our ordering, we can achieve an ordering in which more function pairs are comparable and a comparison that considers only the behavior for large values of n.

Definition: For any function g: $\mathbf{N} \rightarrow \mathbf{R}^+$, O(g) is the set of all functions f: $\mathbf{N} \rightarrow \mathbf{R}^+$ such that for some real constant c and some natural number n_0, $f(n) \leq c\, g(n)$ for all $n \geq n_0$.

The ordering defined using Big-O is the following:

$$f \leq g \Leftrightarrow \; f \in \mathrm{O}(g)$$

Being able to prove such relationships using the definition of Big-O is important because it ensures that we understand the kind of ordering that Big-O defines.

Because Big-O proofs themselves are somewhat cumbersome, once we understand such proofs, it is simpler to prove such relationships by calculating their relative limits.

Sample Problem

Suppose we have

$f(n) = 8n^2 + 7n + 6$ and

$g(n) = n^3 + 10n + 1$.

Prove that $f \in O(g)$ using the definition of Big-O and also by calculating the limit of the ratio of the two functions by applying L'Hôpital's rule.

Solution

The definition of Big-O gives us the freedom to choose two constants. The first constant n_0 is the point after which the inequality specified in the definition must hold. Although we are free to choose this value as large as we like, it is often easiest to just choose 1, which rules out the behavior for negative values. In choosing the multiplicative constant, we pick one so that once it is multiplied by all the coefficients of g, each of the resulting coefficients will be greater than a corresponding coefficient in f. Choosing c as 8 accomplishes that goal.

To satisfy the definition, we must show:

$$\forall n \geq n_0 \;\; f(n) \leq c\, g(n).$$

We know that each of the following inequalities is true for $n \geq 1$:

$8n^2 \leq 8 \cdot n^3$ because $n^2 \leq n^3$ for $n \geq 1$
$7n \leq 8 \cdot 10n$ because $7 \leq 80$
$6 \leq 8 \cdot 1$ because $6 \leq 8$

By adding these inequalities, we get the following:

$$8n^2 + 7n + 6 \leq 8 \cdot (n^3 + 10n + 1) \text{ for } n \geq 1$$

which is the inequality that we needed to satisfy the definition.

Calculating the limit of the ratios of two functions is an alternate method for proving that one function grows no faster than another. The ratio whose limit we must calculate in this case is the following:

$$\lim_{n \to \infty} \frac{8n^2 + 7n + 6}{n^3 + 10n + 1}$$

We can apply L'Hôpital's rule to simplify calculating this limit. Its application involves taking the first derivative of both numerator and denominator, which gives the following limit:

$$\lim_{n \to \infty} \frac{16n + 7}{3n^2 + 10}$$

A second application gives the following limit, which clearly goes to 0:

$$\lim_{n \to \infty} \frac{16}{6n} = 0$$

Showing that the limit of the ratio of these two functions is some finite constant establishes that $f \in O(g)$.

Ordering Functions According to Their Growth Rate

Discussion

The ability to recognize whether one function grows faster than another is critical to algorithm analysis. Although we can use the techniques discussed in the previous sample problem to compare the growth rates of two functions, there are some general rules that can help.

The slowest growing functions are those that don't grow at all, which are the constant functions. Next are the logarithmic functions. All logarithmic functions belong to the same Big-θ class because logarithms of different bases differ only by a constant factor. Next are the roots. The larger roots grow slower so cube root grows slower than square root. The opposite is true among polynomials. The exponential functions grow fastest of all. Unlike logarithms of different bases do not belong to the same Big-θ class.

Sample Problem

Rank the following functions from lowest asymptotic order to highest. List any two or more that are of the same order on the same line.

- $5\,n$
- $n^4 + 9n^2 + 1$
- $n^2 \log_2 n$
- 4^n
- 156
- $10n^3 + 18n$
- $\log_3 n$
- $n^4 + 5n^2 + 1$
- 3^n
- $10n + 7$
- $\sqrt{n}$

Solution

- 156
- $\log_3 n$
- $\sqrt{n}$
- $5\,n, 10n + 7$
- $n^2 \log_2 n$
- $10n^3 + 18n$
- $n^4 + 9n^2 + 1, n^4 + 5n^2 + 1$
- 3^n
- 4^n

Calculating the Efficiency of Recursive Algorithms Using Recursion Trees

Discussion

When an algorithm involves one or more recursive methods, each of whose individual growth rates is constant, we can use recursion trees to determine the efficiency of the algorithm. We draw a recursion tree by using a node to represent one activation of a recursive method. When we draw such trees, we generally choose some small value of *n* as the value given to the activation at the root of the tree.

Drawing the recursion tree helps us to determine formulas for the number of nodes in the tree as a function of *n* and the height of the tree as a function of *n*. The number of nodes in the tree is proportional to the execution time of the algorithm, and the height of the tree is proportional to the amount of memory on the compiler's run-time stack that will be used.

To ensure that our formulas are correct, we use mathematical induction to prove their correctness.

Keep in mind that when a single activation of a recursive method requires more than constant time, this technique cannot be used.

Sample Problem

Draw the recursion tree when $n = 4$ for the following pair of recursive methods:

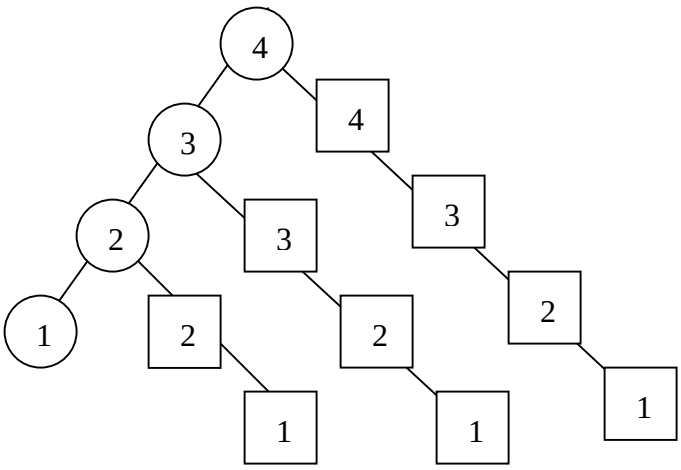
```
int sequence(int n)
{
  if (n == 1)
    return 1;
  return sequence(n-1) + progression(n);
}

int progression(int n)
{
  if (n == 1)
    return 1;
  return progression(n-1) + 3;
}
```

- Determine a formula that counts the numbers of nodes in the recursion tree. Prove the formula is correct by mathematical induction.
- What is the Big-Θ for execution time?
- Determine a formula that expresses the height of the tree.
- What is the Big-Θ for memory?
- Write an iterative solution for this same problem and compare its efficiency with this recursive solution.

Solution

Shown below in Figure 1 is the recursion tree, when $n = 4$. Note that in this tree, the circular nodes represent activations of the method `sequence` and square nodes represent activations of the method `progression`. The number inside the node represents the value of n for that particular activation.

Figure 1
Recursion Tree for `sequence` and `progression` Methods



The formula for the number of nodes in the tree is

$$t(n) = \frac{n^2 + 3n - 2}{2}.$$

To prove it is correct, we use mathematical induction.

First we consider the base case, where n is 1. In that case the diagram will consist of a single circular node. So we must verify that our formula yields that value when 1 is substituted for n:

$$t(1) = \frac{1^2 + 3 \cdot 1 - 2}{2} = 1$$

Next we prove the inductive case. Assuming that the formula is true for $k = n - 1$, we must show it is true for $k = n$:

$t(n) = t(n - 1) + 1 + n$ — From the diagram, we note that each time we increase the value of n, we add 1 circular node and n square nodes.

$$= \frac{(n-1)^2 + 3(n-1) - 2}{2} + 1 + n$$ By the inductive assumption

$$= \frac{n^2 - 2n + 1 + 3n - 3 - 2 + 2 + 2n}{2}$$ By algebraic expansion

$$= \frac{n^2 + 3n - 2}{2}$$ By algebraic simplification

The above result is what we needed to show, consequently we have shown it is true for all $n \geq 1$. Now that we have proven our formula correct, we can conclude that the execution time efficiency is $\Theta(n^2)$ because $t \in \Theta(n^2)$.

From the diagram, it is easy to see that the height of the tree is the length of the right-most branch, so $h(n) = n + 1$. Consequently, the efficiency of memory utilization is $\Theta(n)$ because $h \in \Theta(n)$.

Examining the recursion tree, we note that there are numerous activations of `progression` for the same value of n, so we should not be surprised that this problem can be solved with the more efficient iterative algorithm shown below:

```
int sequence(int n)
{
  int sum = 0;
  for (int i = 1; i <= n; i++)
    sum += 3 * n - 2;
  return sum;
}
```

This solution is $\Theta(n)$ for execution time and is $\Theta(1)$ for memory, which is better than the recursive solution for both measures. In fact, we can even derive a formula for this sequence that would enable us to solve it in constant time.

Calculating the Efficiency of Recursive Algorithms Using the Little Master Theorem

Discussion

Although there is no general technique for solving all recurrence equations, there are solutions for certain kinds of equations. One kind of recurrence equations that is of particular interest arises from divide-and-conquer algorithms. The Little Master Theorem provides us a general technique for solving such equations.

Theorem: For recurrence equations of the form $t(n) = b\ t(\frac{n}{c}) + f(n)$, consider the row sums of the nonrecursive cost for corresponding recursion tree. We have the following three cases:

1. If the row-sums form an increasing geometric sum starting at the root, then $t \in \Theta(n^E)$, where E is the **critical exponent** defined as $E = \frac{\lg(b)}{\lg(c)}$.

2. If the row-sums are constant, then $t \in \Theta(f(n) \log(n))$

3. If the row-sums form a decreasing geometric sequence, then $t \in \Theta(\log(n))$

These cases essentially are based on whether all levels in the tree contribute equally to the overall cost or whether the top level, the root node, predominates or the bottom level does.

Sample Problem

Given the following divide-and-conquer algorithm for finding the largest value in an array of integers:

```
int divide(int[] array, int first, int last)
{
  if (first == last)
    return array[first];
  else if (first + 1 == last)
    return max(array[first], array[last]);
  else
  {
    int mid = (first + last) / 2;
    return max(divide(array, first, mid),
      divide(array, mid + 1, last));
  }
}

int max(int left, int right)
{
  if (left > right)
    return left;
  return right;
}
```

- Write the recurrence equation and initial conditions that define the required execution time as a function of *n*.
- Determine the critical exponent for the recurrence equation.
- Apply the Little Master Theorem to solve that equation.
- Explain whether this algorithm optimal.

Solution

This algorithm divides the array in half, so the recurrence equation that defines the number of calls to `max` that are made is the following:

$$t(n) = t\left(\left\lceil \frac{n}{2} \right\rceil\right) + t\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 \approx 2t\left(\frac{n}{2}\right) + 1$$

The use of floor and ceiling reflect the fact that when *n* is odd, the array will be split into unequal halves. The addition of 1 indicates that in its recursive case, `divide` results in one call to `max`.

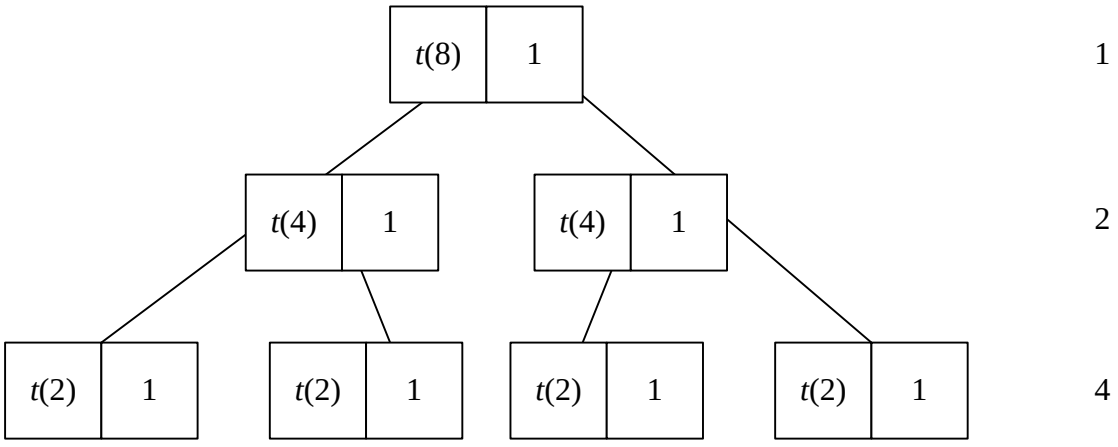
Next, we must define the initial conditions. Because, the recursive method has two base cases, we need to define two initial conditions. They are the following:

$t(2) = 1$
$t(1) = 0$

These cases reflect the fact that no calls to `max` are required when the subarray has only one element, and one call is needed when it has two elements.

We can apply the Little Master Theorem because the recurrence equation is of the correct form. To decide which of the three cases applies, we must examine the recursion tree, which is shown in Figure 2 below:

Figure 2
Recursion Tree for the `divide` Method



We notice that the row sums shown on the right side of the diagram form an increasing geometric sequence, so case 1 of the theorem applies, which means that the nodes at the bottom level of the tree predominate.

From the equation, we note that the branching factor *b* is 2, and the cutting factor *c* is also 2, so we calculate the critical exponent as follows:

$$E = \frac{\lg(b)}{\lg(c)} = \frac{\lg(2)}{\lg(2)} = 1$$

Because part 1 of the Little Master Theorem applies, we can conclude the following:

$$t(n) \in \Theta(n^E) = \Theta(n)$$

Finally, to assess whether this algorithm is optimal, we must determine the lower bound. Clearly, this operation cannot be accomplished without examining every element of the array. If we failed to examine any one of them, that one could have been the largest, so the lower bound is $\Theta(n)$. Consequently, this algorithm is optimal.