Lecture 3: Big O and Recursion

built on 2019/05/01 at 10:56:50

This lecture continues the discussion on performance analysis. We'll also look at how to extend it to recursive programs.

1 Big-O: What is it and why that matters?

We have seen formal definitions of Big-O and how to calculate the Big-O of a program. But Big-O isn't only used in formal contexts. In normal conversations, people talk about the performance of a program in terms of its Big-O. Most often, the meaning of Big-O in such a conversation is simply:

> The program being discussed takes approximately that many steps, ignoring constants, as a function of the input size.

At this casual level, intuitions can be formed about the nature of Big-O. For example:

- If we run an $O(n)$ program followed by another $O(n)$ program, what's the total number of steps in Big-O? **Ans:** Since the first program takes about n steps and the second program takes about n steps, we must be doing about $2n$ steps—but that's still about n steps if we ignore constants. Hence, it's $O(n)$.
- If our program performs about n subtasks, each costing about $O(\log n)$, what's the overall running time? **Ans:** Remember that when we say each subtask costs $O(\log n)$, we really mean it costs about $\log n$ steps. Therefore, n subtasks altogether takes about $n \times \log n$ steps—that is, $O(n \log n)$.

At this point, it is tempting to think that Big O is almost useless: after all, a program that takes $10000000n$ steps and a program that takes $1.5n$ steps are rated as equally good—$O(n)$.

In reality, good programmers won't try to jam a gigantic constant overhead into their programs, so we're talking about reasonably small constants. So then, Big-O allows us to talk about relative performance:

- First, by knowing how fast a program is on input of size n, knowing the Big-O, we can predict how fast it will be on input of size, for example, $2n$. This is a pretty useful thing.
- Furthermore, assuming that the constants are small, we can compare how two algorithms behave by comparing their Big-O's.

2 More Example

2.1 Example I: What's This?

Last time, we saw a number of basic examples. Let's try a different flavor this time. We'll just show you an algorithm, without attempting to explain what it tries to solve:

```java
public static int[] foo(int array[]) {
  int[] ans = new int[array.length];
  for (int i=0;i<array.length;i++) {
    int[] copy = Arrays.copyOfRange(array, 0, i+1);
    Arrays.sort(copy);
    ans[i]=copy[i];
  }
  return ans;
}
```

We now analyze the running time of our `foo` algorithm. Unlike any of the previous examples, this algorithm has only one loop but it makes a number of function calls.

Let $n =$`array.length`. Once again, we know that Line 9 (`return ans`) costs us just a constant. We also know that Line 3 (allocating a new array of length n) costs us $O(n)$. The bulk of the cost comes from the for-loop, and it is not clear a priori how to analyze it. Some questions we need to answer:

1. How much does it cost per iteration (seems like it will vary)?
2. How much is each of these function calls `Arrays.copyofRange` and `Arrays.sort`?

Let's look at the cost of the functions that we call first. By looking up the documentation, we find out that

- On input an array `a` of length n, `Arrays.sort(a)` takes $O(n \log n)$ time.
- `Arrays.copyOfRange(a, from, until)` takes $O(z)$ where $z =$ `until` $-$ `from`.

Hence, in summary, in iteration i:

- the call to `copyOfRange` costs $O(i)$ unit of work.
- the call to `sort` is made on an array of size i so costs $O(i \log i)$ work.
- `ans[i] = copy[i]` is a constant number of primitive operations, hence $O(1)$.

That is, iteration i costs us $O(i) + O(i \log i) + O(1) = O(i \log i)$—more formally, $ci \log i$ for some constant c.

This means, in all, we're looking at:

$$O\Big(n \log n + (n-1)\log(n-1) + (n-2)\log(n-2) + \cdots + 2\log 2 + 1\log 1\Big) = O(n^2 \log n).$$

2.2 Example II: Maximum Contiguous Subsequence Sum (mcss)

Given a sequence of (possibly negative) numbers $a_1, a_2, ..., a_n$, find the contiguous subsequence with the greatest sum. If it helps to spell this out mathematically, we wish to

$$\text{Find } 1 \leqslant i \leqslant j \leqslant n \text{ to maximize } \sum_{k=i}^{j} a_k.$$

- What do we mean by a contiguous subsequence? Say we have as input `3 2 5 7 8 9 10 12`. Any sublist of that list is a subsequence. It's gotta be contiguous, i.e., we can't skip a number. That is, `3 2 5` is okay, but we can't do 3 skiping 2, then 5 7 8.
- What do we mean by the greatest sum? Let's answer this in a few steps.
  - If the numbers are all positive, say 1 2 3 4 5 6 7 8. The more, the merrier. We might as well take the whole thing.
  - It's different when the input sequence contains negative numbers. E.g., $-2, 11, -4, 13, -5, -2$. The answer now is not obvious, but the answer really is 20 $(11, -4, 13, -5)$.

We want to write a function `mcss(a)` that takes a sequence `a` and returns the sum of the contiguous subsequence that has the greatest sum.

In the next few minutes, we'll develop a simple algorithm for solving mcss. It will not be fast but it will be easy to implement. We're going to apply two techniques here, both applicable in a variety of contexts:

- brute force (try all the possibilities)
- modularity

We'll start by examining a seemingly unrelated problem that we already know how to solve: how to find the maximum value from a list of values. If you recall, one can write code that looks something like:

```
maxSoFar = -infty (i.e. something so tiny)
for each value in the list:
   if this value > maxSoFar:
      set maxSoFar to this value.
report maxSoFar
```

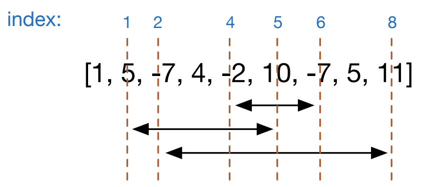
This is how we solve the max-element problem by trying all possibilities. As it turns out, we can do the same for mcss.

Our first goal is to *enumerate all possible subsequences*, so that we can compute the sum for each of them and find the maximum sum. This brings us to the second goal: *compute the sum for any sequence*.

For the first goal, a moment's thought reveals that we can describe any contiguous subsequence by

- the index of the starting point; and
- the index of the ending point

To illustrate this observation, let's look at an example:



Therefore, we could enumerate all subsequences as follows:

```
for (int start=0;start<n;start++) {
  for (int stop=start;stop<n;stop++) {
    // do what you may with start..stop
  }
}
```

Equally simple is the second goal: write a simple function that computes the sum of a given subsequence:

```
static int sumSubseq(int[] a, int from, int until) {
  int sum=0;
  for (int i=from;i<until;i++) sum += a[i];
  return sum;
}
```

Once we have these two pieces of the puzzle, it's not hard to put them together to solve the mcss problem:

```
static int mcss1(int[] a) {
    int maxSum = Integer.MIN_VALUE;
    for (int i=0;i<n;i++) {
        for (int j=start;j<n;j++) {
            thisSum = sumSubseq(a, i, j+1);
            if (thisSum > maxSum)
                maxSum = thisSum;
        }
    }
    return maxSum
}
```

What's the running time of this algorithm? Although this looks like a normal two nested-loop program, there is a benign-looking line that packs a lot of punch: `thisSum = sumSubseq(a, i, j+1)`.

How long does `sumSubseq(a, i, j+1)` take?. Let's take a moment to analyze this function:

> The function loops over i = `from`, ..., `until - 1`, and per iteration, it does constant work. This is a total of `until` − `from` iterations. Setting up `sum = 0` and return take constant work. The total cost is therefore proportion to `until` − `from` or $O(m)$, where m = `until` − `from`.

Now we wish to proceed like before, first deriving the cost for each i and summing them up for all i's. Let n denote the length of a. To analyze the cost for $i = 0$, we'll focus on the j loop. Several things are clear:

- j ranges from 0 to $n-1$, inclusive.
- For each j, the amount of work performed is about $j-i+1 = j+1$, then a constant amount from the if clause.
- Thus, the number of steps for $i = 0$ is
$$1+2+3+\cdots+n = \frac{n(n+1)}{2}.$$
by the summation formula.

For $i = 1$, j goes from 1 to $n-1$ and for each j, the work done is $j-1+1 = j$, so it is $1+2+3+\cdots+(n-1) = (n-1)(n-2)/2$.

Following this pattern, for a general i, the value of j ranges from i to $n-1$. For each j, the amount of work performed is about $j-i+1$, plus a constant. That is, for this i, the work done is $1+2+3+\cdots+(n-i) = (n-i)(n-i+1)/2$.

Hence, the total cost is

$$\begin{array}{rrcr}
\text{Cost for when } i=0: & n(n+1)/2 & \leqslant & n^2 \\
\text{Cost for when } i=1: & (n-1)(n)/2 & \leqslant & (n-1)^2 \\
\text{Cost for when } i=2: & (n-2)(n-1)/2 & \leqslant & (n-2)^2 \\
\vdots & \vdots & & \vdots \quad + \\
\text{Cost for when } i=n-2: & (n-[n-2])(n-[n-3])/2 & \leqslant & 2^2 \\
\text{Cost for when } i=n-1: & (n-[n-1])(n-[n-2])/2 & \leqslant & 1^2 \\
\hline
 & & & 1^2+2^2+\cdots+n^2
\end{array}$$

But we know that
$$1^2+2^2+3^2+\cdots+n^2 = n(n+1)(2n+1)/6,$$
so this is $O(n^3)$.

Room for Improvement Do you see any room for improvements? We wanted a faster algorithm. To do this, we need some more thoughts.

(Hint: There's a nontrivial amount of redundancy. For a start value, consider how `thisSum` *changes as* `stop` *goes from say some value t to $t+1$.)*

In fact, can you come up with an $O(n)$ algorithm by yourself? There's a simple algorithm that runs in $O(n)$ that is pretty intuitive.

3 Running Time Analysis of Recursive Algorithms

How can we determine the running time of a recursive algorithm? So far, we have talked about counting the number of steps taken by an algorithm and use that as a prediction for the algorithm's running time. However, when an algorithm is recursive, it can be daunting to attempt to count the number of steps directly. For this, we'll use a tool called *recurrence relations*.

You have seen some recurrece relations already. For example, the Fibonacci recurrence $F(0) = 1, F(1) = 1, F(n) = F(n-1) + F(n-2)$; or $S(n) = S(n-1) + n$. These are simply recursive functions: *functions that call themselves.*

What we're going to do is:

> Given a function that we want to analyze, we'll write another recursive function—parallel to the function being analyzed—to determine the running time.

3.1 Example: Raising A Number To A Positive Integral Power

Say we wish to implement a function `pow`(b, w) which takes as input a non-zero real number $b \neq 0$ and a non-negative integer $w \geqslant 0$. The function is to output the number b^w. Here's the code we'd perhaps write using recursion:

```python
def pow(b, w):
  if w==0: return 1
  else: # this means w > 0
     return pow(b, w-1)*b
```

```java
int pow(int b, int w) {
  if (w==0) return 1;
  else return pow(b, w-1)*b;
}
```

Why this code is correct will be the subject of discussion of another lecture. For now, let us focus on the running time of this program.

The function takes two parameters as input, but we'll see soon that only w factors into the running time.

Assuming that foresight, we'll measure the problem size in w, so we write a recurrence for

$T(w)$ = how long it takes to run $\texttt{pow}(b, w)$ for any b.

The function $T(w)$ is a recursive function that takes w, our problem size, and returns the number of steps (i.e., how long it takes) to run $\texttt{pow}(b, w)$ for any b.

To write $T(w)$, we'll draw parallel to the code we wrote for pow. Once we examine the code, it is clear that there are two cases: Either we're in the small-instance case ($w = 0$) or we're in the recursive case (i.e., $w > 0$).

It is easy to see that when $w = 0$, we perform constant work (we test if w is 0, then we return 1), so $T(0)$ is $O(1)$.

The case where $w > 0$ is more interesting: if $w > 0$, we carry out the following steps:

(i) we recursively call ourselves with $w - 1$; and
(ii) we multiply that result with b.

Therefore,

$$T(w) = \boxed{\text{cost of (i)}} + \boxed{\text{cost of (ii)}}$$

To determine the complexity of the recursive call, the crucial question to ask ourselves is, *how large is the problem instance we're giving the recursive call?* Answering this question in our case is easy: we're measuring the size in w and we're calling pow with $w - 1$. Hence, the cost of (i) is $T(w - 1)$. And the cost of (ii) is constant, so we have

$$\begin{aligned} T(w) &= \underbrace{\text{cost of (i)}}_{=T(w-1)} + \underbrace{\text{cost of (ii)}}_{=O(1)} \\ &= T(w-1) + O(1), \end{aligned}$$

which, by the look-up table, solves to $T(w) \in O(w)$.

In conclusion: Our choice of size measure works (remember we picked w). We have written a recursive function for computing b^w, $w \geqslant 0$, with running time $O(w)$.

3.2 Example II: Computing the Sum of Numbers

The second example deals with adding together numbers in an array. This is what you might write, recursively:

```java
int sum(int[] A) {
    if (A.length==0) return 0;
    else if (A.length==1) return A[0];
    else {
        m = A.length/2;
        return sum(Arrays.copyOfRange(A,0,m)) +
            sum(Arrays.copyOfRange(A,m,A.length));
    }
}
```

Abstractly, in the code above, the following steps are performed:

1. Derive a copy of A with only elements in range 0 and m (exclusive).
2. Run `sum` recursively on this new array.
3. Derive a copy of A with only elements in range $m + 1$ and the end of the list.
4. Run `sum` recursively on this new array.

3.3 Analysis

What's the running time? To find out, we'll write a recurrence. Let $T(n)$ be the time to run `sum` on an array of length n. Immediately, we have $T(0) = O(1)$. And for $n > 0$, we know that there are two sources of running time: one due to recursive calls and the other due to basic operations, such as arithmetic and array slicing. The former is obvious: $2T(n/2)$ because we make two calls to `sum`, each on a problem of size $n/2$. The latter, as was shown before, is $O(n)$ because $m \approx n/2$ so each of `A[:m]` (or `Arrays.copyOfRange(A,0,m)`) and `A[m]` (or `copyOfRange(A,m+1,A.length)`) is $O(m) = O(n)$. Hence, we have $T(n) = 2T(n/2) + O(n)$, which solves to $T(n) \in O(n \log n)$.

This is, however, not quite what we wanted. The most naïve implementation we had runs in $O(n)$. How can we fix this so that it runs as least as fast as the nonrecursive program? We'll look at this soon.

4 Common Recurrences

We list a couple common recurrences, assuming $T(0)$ and $T(1)$ are constant:

- $T(n) = T(n/2) + O(1)$ solves to $O(\log n)$.
- $T(n) = T(n/2) + O(n)$ solves to $O(n)$.
- $T(n) = 2T(n/2) + O(1)$ solves to $O(n)$.
- $T(n) = 2T(n/2) + O(\log n)$ solves to $O(n)$.
- $T(n) = 2T(n/2) + O(n)$ solves to $O(n \log n)$.
- $T(n) = 2T(n/2) + O(n^2)$ solves to $O(n^2)$.
- $T(n) = T(n-1) + O(1)$ solves to $O(n)$.
- $T(n) = T(n-1) + O(n)$ solves to $O(n^2)$.

Discrete Math teaches you how to solve them yourself.