Big-O Examples

Some commonly used function classes (and typical examples) in order of growth rate

O(1)
- Constant (fixed expressions, not depending on the input size)

O(log n)
- Binary Search

O(n)
- Sequential search, finding minimum/maximum

O(n^2)
- Selection Sort

O(n log n)
- Merge Sort

O(n^3)
- Matrix Multiplication

O(2^n)
- Subset sum ("is there any subset of the elements of an array that add up to exactly p?"). Strictly speaking the subset sum algorithm is O($n2^n$) but we think of it as an "exponential time" or intractable algorithm

Note there is a spreadsheet posted in the Notes/Examples section of WebCT showing sample running times to give a sense of a) relative growth rates, and b) some problems really are intractable.

What big-O complexity means

Given two functions f(n) and g(n), we say that f is O(g) ("f is big-O of g") if f(n) is bounded by some multiple of g(n) for all large values of n. Technically, f is O(g) if you can find two constants c and n_0 such that f(n) <= cg(n) for all n > n_0. In most cases the point of doing this is to get a simple description of how a function f(n) will behave as n gets larger. For example, we may analyze an algorithm and decide that for an input of size n, it takes

$f(n) = 472n^2 – 95n + \log_{17}(2n + 152)$

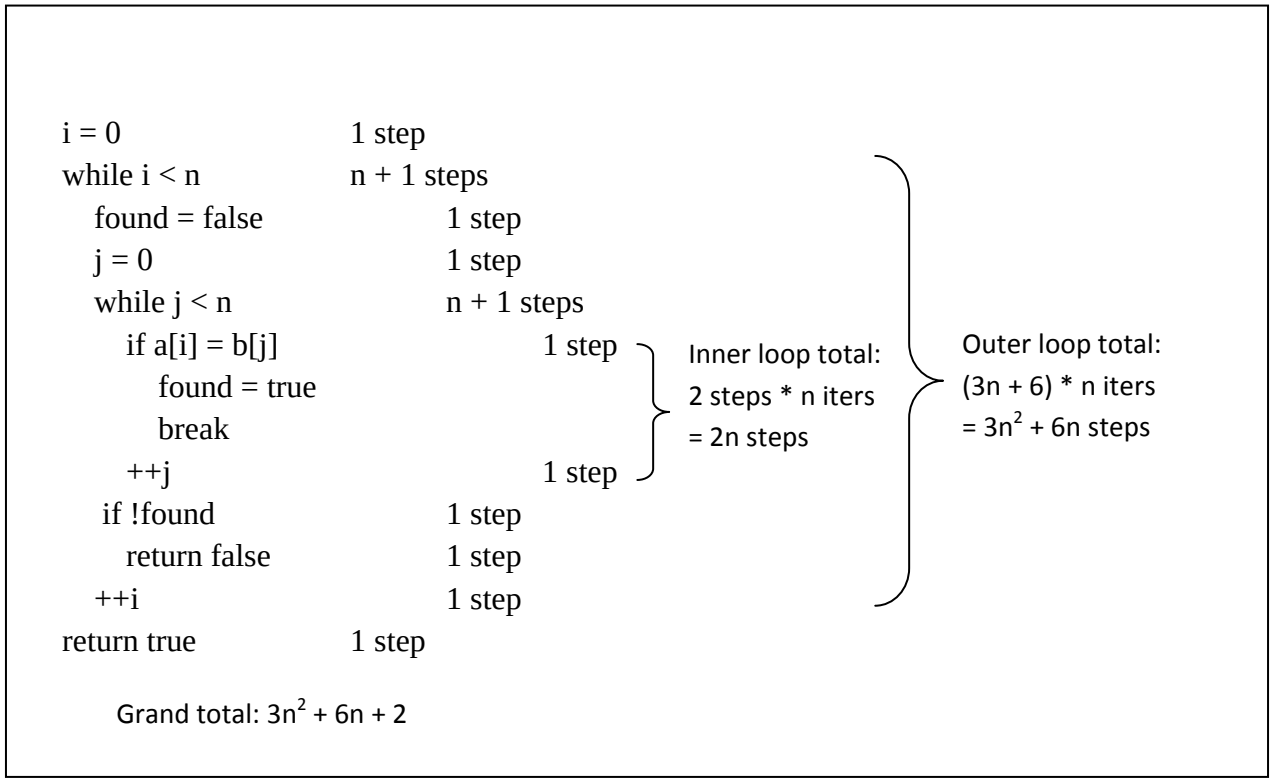
steps to execute in the worst case. But we can classify this function as O(n^2), which tells us than as we give the algorithm larger and larger inputs, the execution time will essentially be proportional to a quadratic function.

Determining big-O complexity

The basic idea is that we are counting "execution steps". Whatever the algorithm, it takes more work to process a big input than a small input, so the number of execution steps is expressed as a function of the input size. (In the examples we've seen so far, the input size is the length of an array, but that will change later on.) A "step" is not precisely defined, it is just some simple bit

of code that executes in constant time (not dependent on the input size). You can go through and literally count the steps like we did at first. Here is an example for an algorithm that, given two integer arrays a and b (without duplicates), determines whether they contain the same elements. In this case the input size is the array length, referred to as n.

Remember, we are most often interested in the *worst-case* execution time.



The algorithm takes $f(n) = 3n^2 + 6n + 2$ steps for arrays of size n, and f is $O(n^2)$.

Since all we ultimately care about is the big-O class of the function, you can see that we really didn't have to work so hard counting up the individual steps of the algorithm. Just notice that the inner loop has O(n) iterations, and it executes O(n) times, so we get O(n * n) or $O(n^2)$.

Code Examples

```
// O(n), where n = arr.length
void method1(int [] arr)
{
      int n = arr.length;
      for(int i = n - 1 ; i >= 0;  i = i - 3)
      {
            SOP (arr[i]);
      }
}
```

Remark: We're assuming that SOP (System.out.println) is O(1).

```
// O(n^2)
void method2(int [] arr)
{
	for(int i = 0; i <arr.length; i++)
	{
		for(int k = 0; k <arr.length; ++k)
		{
			SOP (Math.log(arr[i]));
		}
	}
}
```

```
// O(n^2)
void method2a(int [] arr)
{
	for(int i = 0; i <arr.length; i++)
	{
		for(int k = i; k <arr.length; k++)
		{
			SOP (arr[i]);
		}
	}
}
```

Remark: You can use the fact (from Calc 1) that 1 + 2 + ... + n-1 = n(n - 1)/2, or observe intuitively that the *average* number of iterations for inner loop must be n/2

```
// O(n^3)
void method3(int [] arr) {
	for(int i = 0; i <arr.length; i++)
	{
		method1(arr);
		method2(arr);
	}
}
```

Remark – this illustrates a couple of general principles about big-O
1) If you have an O(f(n)) operation and you do it O(g(n)) times, the execution time is O(f(n) * g(n))
2) If you have an O(f(n)) operation followed by an O(g(n)) operation, time is O(max(f(n), g(n))), for example, O(n^2 + n) is the same as O(n^2)

```
// O(n * log n)
void method4(int [] arr)
{
	for(int i = 0; i <arr.length; i++)
	{
		for(int k = arr.length - 1;  k > 0; k = k / 3 )
		{
			SOP (arr[i]);
		}
	}
}
```

Remark: A logarithm function is just the answer to the question "what's the exponent?", which just means, "how many times can I multiply?"

So here we're asking: given n, how many times can I multiply by 1/3 before it gets down to 1 ?

This is the same as asking: 3 to what power is equal to n ? $n\left(\frac{1}{3}\right)^{?} = 1 \Leftrightarrow n = 3^{?}$

In this example the answer is the log base 3 of n. 3^y = n means that y = $\log_3$ n. In general for any base b, $\log_b$ x is the exponent you'd put on b to get x as the result. E.g. 2^5 = 32 tells us that $\log_2$ 32 = 5.

It turns out for talking about big-O we don't care about the base, since all log functions are proportional to each other. For example to get the base three log on a pocket calculator, you'd use the log base *e* (natural log) and multiply by the constant $\frac{1}{\ln 3}$, that is, $\log_3 x = \frac{1}{\ln 3}\ln x$.

Basic rule: if the problem size is reduced by a constant factor at each iteration, the total number of iterations is O(log n).

(This is useful b/c for "divide-and-conquer" style algorithms like binary search or merge sort, we typically think in terms of reducing the problem size by some factor which is a fraction.)

Contrast this loop with the loop of method 1, where we are *subtracting* 3, not multiplying.

```
// O(n^2), where n is the number of elements in arr
void foo (ArrayList<Integer> arr)
{
      for(int i = 0; i < arr.size();  i++)
      {
            If(!(arr.contains(i))) SOP (arr[i]);
      }
}
```

Remark: the input doesn't have to be an array, but the notion of "input size" is only meaningful if the input consists of multiple items, and we don't yet know of any data structures besides arrays and ArrayList

Remark: How do we know about the time complexity of contains() ? See the second paragraph of ArrayList class documentation:

"The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in *amortized constant time*, that is, adding n elements requires O(n) time. **All of the other operations run in linear time** (roughly speaking)."

Recursion examples

Binary search (code on next page)

To analyze the big-O time complexity for binary search, we have to count the number of recursive calls that will be made in the worst case, that is, the maximum *depth* of the call stack.

Here, each recursive call looks at (at most) only half the array, so the max depth is the number of times we can divide an n-element array in half, i.e., the base 2 log of n. The actual work done in each frame is O(1). So this binary search algorithm is O(log n).

```
public static boolean binarySearch(int[] arr, int p)
{
  return binarySearchRec(arr, p, 0, arr.length - 1);
}

private static boolean binarySearchRec(int[] arr, int p, int first, int last)
{
  // base case
  if (first > last) return false;

  int mid = first + (last - first) / 2;
  if (p == arr[mid])
  {
    return true;
  }
  else if (p < arr[mid])
  {
    return binarySearchRec(arr, p, first, mid - 1);
  }
  else
  {
    return binarySearchRec(arr, p, mid + 1, last);
  }
}
```

Merge sort

This is a recursive algorithm that is not so simple to analyze as binary search, because there are multiple recursive calls in the method body.

```
private static void mergeSortRec(int[] arr, int first, int last)
{
  if (first >= last) return;

  int mid = (first + last) / 2;
  mergeSortRec(arr, first, mid);
  mergeSortRec(arr, mid + 1, last);

  merge(arr, first, mid, last);
}
```

It is not difficult to check that the merge operation is O(k) for two subarrays of length k/2, since at each step, one element is added to the final array.

The initial array of size n is split into two subarrays of size (roughly) n/2. After two recursive calls to mergesort, the sorted subarrays are then merged back into a single array of size n. The actual work done in each frame is just the merge operation.

Since every call divides the subarray in half, it is not hard to see that, just as with binary search, the call stack can never be more than log n frames deep. However, each call generates *two* more recursive calls, so the total number of steps in this case is much more than log n.

To count up all the steps, we group together all the work that is *ever* done in each level of the call stack. At the first level, the work is to merge two sorted subarrays of length n/2 into a single array of length n. This requires O(n) steps. At the second level, the work involves merging two subarrays of length n/4 into an array of length n/2, but by the time the algorithm finishes this is going to happen twice, so it is still O(n) steps. At the next level, subarrays of size n/8 are merged into arrays of size n/4, but eventually this will happen 4 times, so it is still O(n) steps altogether. At subsequent levels the same argument can be made: there are many smaller subarrays getting merged at various points in the execution, but the total work always adds up to O(n). (See picture.)

Ultimately there are log n levels and O(n) steps are taken at each level, for a total of O(n log n).