"Big O" Notation

Formal definition: \( f(n) \) is \( O(g(n)) \) if there exist constants \( c > 0 \)
and \( N \geq 0 \) such that for all \( n \geq N \), \( f(n) \leq c \cdot g(n) \).

Intuitively, \( f(n) \) is \( O(g(n)) \) means that \( f(n) \) grows
like \( g(n) \) or slower.

Prove that \( n^2 + n \) is \( O(n^2) \)

Formal definition: \( f(n) \) is \( O(g(n)) \) if there exist constants \( c > 0 \)
and \( N \geq 0 \) such that for all \( n \geq N \), \( f(n) \leq c \cdot g(n) \).

Let \( f(n) \) and \( g(n) \) be two functions.
\( f(n) \geq 0 \) and \( g(n) \geq 0 \).

We showed that \( n+6 \) is \( O(n) \).

In fact, you can change the 6 to any constant \( c \) you want
and show that \( n+c \) is \( O(n) \).

An algorithm that executes \( O(n) \) steps on input of size \( n \)
is called a linear algorithm.

The difference between executing 1,000,000 steps and 1,000,006 is
insignificant.

Oft-used execution orders

In the same way, we can prove these kinds of things:

1. \( \log(n) + 20 \) is \( O(\log(n)) \) (logarithmic)
2. \( n + \log(n) \) is \( O(n) \) (linear)
3. \( n/2 \) and \( 3n \) are \( O(n) \)
4. \( n \cdot \log(n) + n \) is \( O(n \log(n)) \)
5. \( n^2 + 2n + 6 \) is \( O(n^2) \) (quadratic)
6. \( n^3 + n^2 \) is \( O(n^3) \) (cubic)
7. \( 2^n + 5n \) is \( O(2^n) \) (exponential)
Understand? Then use informally

1. \( \log(n) + 20 \) is \( O(\log(n)) \) (logarithmic)
2. \( n + \log(n) \) is \( O(n) \) (linear)
3. \( n/2 \) and \( 3*n \) are \( O(n) \)
4. \( n^2 + 2*n + 6 \) is \( O(n^2) \) (quadratic)
5. \( n^3 + n^2 \) is \( O(n^3) \) (cubic)
6. \( \log(n) + 20 \) is \( O(\log(n)) \) (logarithmic)

Once you fully understand the concept, you can use it informally. Example:

An algorithm executes \( (7*n + 6)/3 + \log(n) \) steps. It's obviously linear, i.e. \( O(n) \)

Some Notes on \( O() \)

• Why don’t logarithm bases matter?
  – For constants \( x, y \): \( O(\log_x n) = O(\log_y y \log_x n) \)
  – Since \( \log_y y \) is a constant, \( O(\log_x n) = O(\log_y n) \)

• Usually: \( O(f(n)) \times O(g(n)) = O(f(n) \times g(n)) \)
  – Such as if something that takes \( g(n) \) time for each of \( f(n) \) repetitions... (loop within a loop)

• Usually: \( O(f(n)) + O(g(n)) = O(\max(f(n), g(n))) \)
  – “\( \max \)” is whatever’s dominant as \( n \) approaches infinity
  – Example: \( O((n^2-n)/2) = O(1/2)n^2 + (-1/2)n) = O((1/2)n^2) \)
  = \( O(n^2) \)

ANALYZING AN ALGORITHM

Runtime

```
public static void mS(Comparable[] b, int h, int k) {
  if (h >= k) return;
  int e = (h+k)/2;
  mS(b, h, e);
  mS(b, e+1, k);
  merge(b, h, e, k);
}
```

Use \( T(n) \) for the number of array element comparisons that \( mS \) makes on an array of size \( n \)

Recursion: \( T(n) = 2 \times T(n/2) + \text{comparisons made in merge} \)

Simplify calculations: assume \( n \) is a power of 2
/** Sort b[h..k]. Pre: b[h..e] and b[e+1..k] are already sorted.*/
public static void merge (Comparable b[], int h, int e, int k) {
    Comparable[] c= copy(b, h, e);
    int i= h; int j= e+1; int m= 0;
    for (i= h; i!= k+1; i++) {
        if (j <= k && (m > e - h || b[j].compareTo(c[m]) <= 0)) {
            b[i]= b[j]; j= j+1;
        } else {
            b[i]= c[m]; m= m+1;
        }
    }
}

Runtime
We show how to do an analysis, assuming n is a power of 2 (just to simplify the calculations)

Use T(n) for number of array element comparisons to mergesort an array segment of size n

public static void mS(Comparable[] b, int h, int k) {
    if (h >= k) return;
    int e= (h+k)/2;
    mS(b, h, e);
    mS(b, e+1, k);
    merge(b, h, e, k);
}

Thus: T(n) < 2 T(n/2) + n, with T(0) = 0, T(1) = 0

Proof by recursion tree of T(n) = n lg n
T(n) = 2*T(n/2) + n, for n > 1, a power of 2, and T(1) = 0

MergeSort vs QuickSort
• Covered QuickSort in Lecture
• MergeSort requires extra space in memory
  — The way we’ve coded it, it needs that extra array
  — QuickSort is an “in place” or “in situ” algorithm. No extra array. But it does require space for stack frame for recursive calls. Naïve algorithm: O(n), but can make O(log n)
• Both have “average case” O(n lg n) runtime
  — MergeSort always has O(n lg n) runtime
  — QuickSort has “worst case” O(n^2) runtime
  • Let’s prove it!
Quicksort

• Pick some "pivot" value in the array
  
  Partition the array:
  – Finish with the pivot value at some index j
  – everything to the left of j ≤ the pivot
  – everything to the right of j ≥ the pivot
• Run QuickSort on b[h..j-1] and b[j+1..k]

Runtime of Quicksort

• Base case: array segment of 0 or 1 elements takes no comparisons
  
  \[ T(0) = T(1) = 0 \]
• Recursion:
  – partitioning an array segment of \( n \) elements takes \( n \) comparisons to some pivot
  – Partition creates length \( m \) and \( r \) segments (where \( m + r = n - 1 \))
  
  \[ T(n) = n + T(m) + T(r) \]

Worst Case Runtime of Quicksort

• When \( T(n) = n + T(n-1) + T(0) \)
  
  \[ \text{Hypothesis: } T(n) = (n^2 - n)/2 \]
  
  \[ \text{Base Case: } T(1) = (1^2 - 1)/2 = 0 \]
  
  \[ \text{Inductive Hypothesis: } \]
  
  Assume \( T(k) = (k^2 - k)/2 \)
  
  \[ T(k+1) = k + (k^2 - k)/2 + 0 = (k+1)^2/2 = ((k+1)^2 - (k+1))/2 \]
  
  \[ \text{Therefore, for all } n \geq 1: } \]
  
  \[ T(n) = (n^2 - n)/2 = O(n^2) \]

Worst Case Space of Quicksort

You can see that in the worst case, the depth of recursion is \( O(n) \). Since each recursive call involves creating a new stack frame, which takes space, in the worst case, Quicksort takes space \( O(n) \).

That is not good!

To get around this, rewrite QuickSort so that it is iterative but it sorts the smaller of two segments recursively. It is easy to do. The implementation in the java class that is on the website shows this.