merge sort divides the list in half; first reorganizes the elements on the left half independently of the right half and same for the right. this is easier than sorting the whole list (1/2 the amount of work). then merge the two sorted sublists into one complete totally sorted list.

of all the elements in the sublists, there are possibly 2 elements that can go into the first position of the newly sorted list: the first elements of the sublists (since they're the smallest of their own independent lists). thus, those two are compared and whichever is smallest is put there. then increment and keep doing them for each element in the list.

Recurrence Relations and Defining Big O

merge = 1 comparison, 1 copy into list, 2++ = constant amount of work (n elements to fill in, which = n efficiency)

Finding the runtime of recursive code.

The mathematical definition of Big O.

Merge Sort Recursive

basically 1 * c, but constant doesn't matter

$$T(n) = \begin{cases} 1, & \text{for } n <= 1 \\ 2 * T(n/2) + n, & \text{otherwise} \end{cases}$$

T(100) > T(50); further, T(100) > 2 * T(50) b/c to merge sort of 100, you have to 2 merge sorts of two sizes of 50 + merging

different T(n) for different algorithms

1. write a recurrence relation that models the number of steps that the algorithm executes to solve a problem of size n (called T(n) for time), which mimics the shape of the recursive algorithm (has at least 2 cases; base + recursive)

based on number of recursive calls

Code sketch for the Merge Sort algorithm:

```
MergeSort (array){
  c      n = the size of the array
         if (n <= 1)                      //Base Case
            return done;
  c      L = left half of the array       //Size = n/2
         R = right half of the array      //Size = n/2
T(n/2)   MergeSort(L);  size of n/2       //Recursive Case
T(n/2)   MergeSort(R);                    //Recursive Case
  n      Merge(L, R); (explanation on last slide)  //Method not shown
                                            not recursive
}
```

Runtime of Recursive MergeSort

Using T(n) as shorthand for “upper bound on the number of steps it takes MergeSort to sort an array of size n”

We expect T(100) > T(50), perhaps T(100) > 2*T(50)

Because we divide the array in half and sort each half separately, T(100) must be 2*T(50) + the cost to merge the two sorted halves back together.

```
MergeSort (array){
    n = the size of the array
    if (n <= 1)
     return done;
    L = left half of the array
    R = right half of the array
    MergeSort(L);
    MergeSort(R);
    Merge(L, R);
}
```

Runtime of Recursive MergeSort

Now that we have a bit of an intuitive sense for T(n), let's break T(n) up in terms of other functions of n.

In the base case, when n <= 1, the algorithm executes a constant number of steps, so T(1) = 1 (* a const).

The Merge executes in n steps (or n * some const).

The recursive case can be summed up as:
 T(n) = T(n/2) + T(n/2) + n (+ some const)

```
              MergeSort (array){
1      ———        n = the size of the array
1      ———        if (n <= 1)                      //Base Case
1      —————          return done;
1      ———        L = left half of the array       //Size = n/2
1      ———        R = right half of the array      //Size = n/2
T(n/2) ———        MergeSort(L);                    //Recursive Case
T(n/2) ———        MergeSort(R);                    //Recursive Case
n      ———        Merge(L, R);
              }
```

Runtime of Recursive MergeSort

Base Case (when n <= 1):

T(1) = 1

Recursive Case (when n > 1):

T(n) = T(n/2) + T(n/2) + n

```
         MergeSort (array){
1      ——  n = the size of the array
1      ——  if (n <= 1)                     //Base Case
1      ——      return done;
1      ——  L = left half of the array      //Size = n/2
1      ——  R = right half of the array     //Size = n/2
T(n/2) ——  MergeSort(L);                   //Recursive Case
T(n/2) ——  MergeSort(R);                   //Recursive Case
n      ——  Merge(L, R);
         }
```

These expressions make up the **Recurrence Relation** for T(n), which is formally expressed as

$$T(n) = \begin{cases} 1 & \text{when } n = 1 \\ 2T(n/2) + n & \text{when } n > 1 \end{cases}$$

one of the two is a recursive algorithm

Practice Quiz: Recurrence relations and runtime of recursive algorithm

input is external to the algorithm, "formal parameter"

```
function helper (array A)
  s = size of A;          c
  sum = 0;

  for i = 1 to sqrt(s)   // sqrt(N) = N^0.5
     sum = sum + A[i];    c

  return sum;             c
end function helper
```

$$T(n) = \begin{cases} 1, \text{ for } n <= 5 \text{ (b/c size of A is bounded for the if, it would normally be n)} \\ 3 * T(n/6) + n^{(1/2)}, \text{ otherwise} \end{cases}$$

notice the subproblems are chopped (smaller)

how many times does it make us do the body?, the comment is that it's a polynomial; thus it's s^(1/2)

work based on its actual parameter than the formal parameter (A here != A helper)

external input (size n)

```
function solver ( array A )
  if the size of A is 5 or less
    return the largest value in A;
  else
    Let A1, A2, A3, A4, A5 and A6 be six
      contiguous "equal size" parts of A;     c

    // recursive calls
    x1 = solver (A1);
    x2 = solver (A3);
    x3 = solver (A5);

    x4 = helper (A);

    return x1 + x2 + x3 + x4;
  end if
end function solver
```

3 * T (n/6)

T(smthg); would be T(n/6) for each call b/c of above

n^(1/2)

(first do analysis of helper method, think innermost like how merge was)

a. *Give a tight upper bound for the runtime of function **helper** in terms of s, the size of its input.* answer is s^(1/2)

b. *Write a recurrence relation that models the runtime for function **solver**.*

Runtime of Recursive MergeSort

So we found that *the upper bound on the number of steps it takes MergeSort to sort an array of size n*, T(n), looks like this

$$T(n) = \begin{cases} 1 & \text{when } n = 1 \\ 2T(n/2) + n & \text{when } n > 1 \end{cases}$$

n^2

How does MergeSort compare with the quadratic SelectionSort?

we need to put them into the same format so we know if it's better/worse

Can we compute a closed-form version of T(n)?

1. is it recursive?
2. solve it ig

- Method of backwards substitution
- The Master Theorem

Runtime of Recursive MergeSort

for all n > 1

I. **$T(n) = 2T(n / 2) + n$** is true for $\forall$ $n > 1$

- We must now find what $T(n/2)$ is. For large n, n/2 is indeed > 1, so I. applies for n/2. This results in **$T(n/2) = 2T(n/2 / 2) + n/2 = 2T(n / 4) + (n / 2)$** substitute (n/2) into the original equation, so we get an equation in terms of n/2

II. Substituting this last result back into the original I. gives:

put T(n/2) we found into the original equation

- **$T(n)$** $= 2T(n / 2) + n = 2 * [2T(n / 4) + (n / 2)] + n = 4T(n / 4) + n + n =$ **$4T(n / 4) + 2n$**
- We can use the same process as before to find that for $n/4 > 1$, **$T(n / 4) = 2T(n / 8) + (n / 4)$**

III. This last result can be substituted back into II., resulting in :

- **$T(n)$** $= 4T(n / 4) + 2n = 4 * [2T(n / 8) + (n / 4)] + 2n =$ **$8T(\underbrace{n / 8}) + 3n$**

gives T(n) in terms of T(n/4)

IV. Continuing the pattern: **$T(n) = 16T(n / 16) + 4n$**

eventually becomes 1, and T (1) is just 1, so no more T (think discrete big O??)

Runtime of Recursive MergeSort

So $T(n) = \mathbf{2T(n / 2) + n}$

$= \mathbf{4T(n / 4) + 2n}$

$= \mathbf{8T(n / 8) + 3n}$

$= \mathbf{16T(n / 16) + 4n}$

… continue until the pattern is obvious and you can generalize…

V. $T(n) = 2^i * T(n / 2^i) + i * n$

what value does i need to be so that we get T(1) (which makes that value 1, or the base case)

n/(2^i) = 1
n = 2^i
logn = log(2^i)
i = logn

plug i in for what we found:

T(n) = 2^(logn) * T(n/2^(logn)) + (log n * n)
T(n) = 2^(logn) * T(1) + (logn * n)
T(n) = n * 1 + (logn * n) = n + n(logn)
~nlogn (dominant term)

since nlogn < n^2, mergesort is more efficient than selection sort for large n

Runtime of Recursive MergeSort

V. $T(n) = 2^i * T(n / 2^i) + i*n$

VI. Observe that $(n / 2^i)$ decreases until it is <= 1.

(Let's assume for simplicity's sake that $(n / 2^i) = 1$.)

At this point we will have reached the base case, $T(1) = 1$,

and we will have a closed form expression.

This happens when $\mathbf{(n / 2^i) = 1}$ or $\mathbf{n = 2^i}$ or $\mathbf{\log(n) = i}$.

Substituting this back into the generalized form in V. we get that

VII. $T(n) = 2^{\log(n)} * T(n / 2^{\log(n)}) + \log(n) * n = n*T(1) + n\log(n) = n + n\log(n)$

*This method of finding the closed form for a recurrence relation is called the method of **backwards substitution**.*

Runtime of Recursive MergeSort

So T(n) = n + n*log(n).

The dominant term in this function of n is nlog(n) so, for sufficiently large n,

the “shape” of the number of steps in MergeSort is nlog(n),

and we say that the of the **runtime** of MergeSort is nlog(n).

MergeSort is better than the quadratic SelectionSort for large inputs!

For algorithm evaluation purposes, we’ve argued that it suffices to obtain nlog(n).

Now let’s formalize this. In the slides that follow, think of

f(n) as the exact count for the number of steps in a code segment, and

g(n) as the “shape” of the number of steps.

relationship between 2 functions of n (f and g); f is the exact count for the exact number of steps and g is the shape of the exact number of steps (approx upper bound on exact # of steps)

Definition of Big O

we want the higher order term w/o the constants (g)

- Let f and g be functions of n.
- We say "**f is in big O of g**" (or "f is upper bounded by g") and write "**f ~ *O*(g)**"

if

we want the positive version of g

first quadrant

[$\exists$ a constant $c > 0$] and [$\exists$ a constant $n_0 \geq 0$] such that

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

doesn't need to hold for all values of n; the bound only holds to the right of "n n0?" (small #s, don't care, it's for large n, beyond given "n0")

upper bounded

Proof of Big O . . .

Example: Show that $\mathbf{g(n) = n^2}$ is the upper-bound of $\mathbf{f(n) = 3n^2 + 6n + 55}$, or more formally:

$$\mathbf{3n^2 + 6n + 55 \sim O(n^2)}$$

f must be upper bounded by something * n^2

For this example, we choose “c” to be 4, although it could be 3.1 or 5 or 444 or any number larger than 3.

thus, 3n^2 + 6n + 55 <= 4n^2, we ask at what "n" do they cross?

We then use the value that we chose for c to determine n_0 by setting f equal to c*g:

$$3n^2 + 6n + 55 = 4n^2 \quad \Longrightarrow \quad 4n^2 - (3n^2 + 6n + 55) = 0 \quad \Longrightarrow \quad n^2 - 6n - 55 = 0$$

$$\Longrightarrow \quad (n - 11)(n + 5) = 0 \quad \Longrightarrow \quad n = -5 \text{ and } n = 11$$

Because n_0 must be >= 0, we can choose $n_0 = 11$ when we choose c = 4.

don't have to use 11 but you can use n's higher than 11

Since $[3n^2 + 6n + 55] \leq 4 * [n^2]\ \forall\ n \geq 11$, $\mathbf{[3n^2 + 6n + 55] \sim \mathit{O}(n^2)}$.

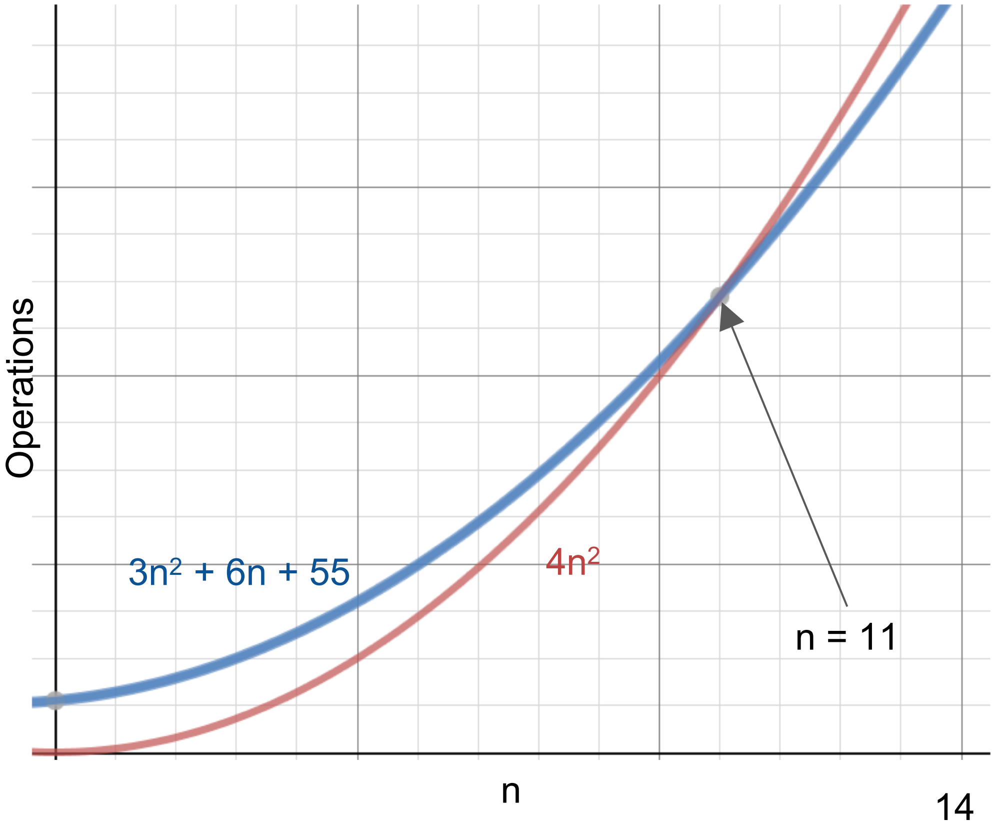
f is upperbounded by g * a constant (4); thus, for all n >= 11, so f is in the big O of g

We can blow off the lower-order terms and the constants!

Big O Example

This graph visually shows both f(n) and c*g(n), making it easy to see that c*g(n) is an upper-bound on f(n) for all values to n > 11.

Notice that the constant "c" allows us to disregard the coefficient of the higher order term and that n_0 grants forgiveness for the upper-bound failing on "finitely many" small values of n.



Big O Example

A formalized statement for the preceding example may take a form like this:

For the case when $f(n) = 3n^2 + 6n + 55$ and $g(n) = n^2$, we can indeed say:

$$3n^2 + 6n + 55 \sim O(n^2)$$

Because we can indeed pick "c" (4) and "n_0" (11) that satisfy:

$$3n^2 + 6n + 55 \leq 4n^2 \ \forall \ n \geq 11$$

To us this is useful because we've learned to estimate upper bounds (g(n)s) for exact step counts (f(n)s) and now we have a formal way to say that our upperbounds are indeed so.

Big O Example

Do not assume that any function is an upper bound on any other function just because we're allowed to choose the constants c and n_0.

Notice that for the case $f(n) = n^2$ and $g(n) = n$, it's impossible to pick c and n_0.

Proof (by contradiction):

-Suppose c and n_0 exist.

-Then $n^2 \leq c*n \ \forall \ n \geq n_0$ multiply both sides by 1/n

-Then we would have that $n \leq c \ \forall \ n \geq n_0$ as n grows to infinity, n is bounded by a constant. this is bogus because n is infinity and not limited by a constant

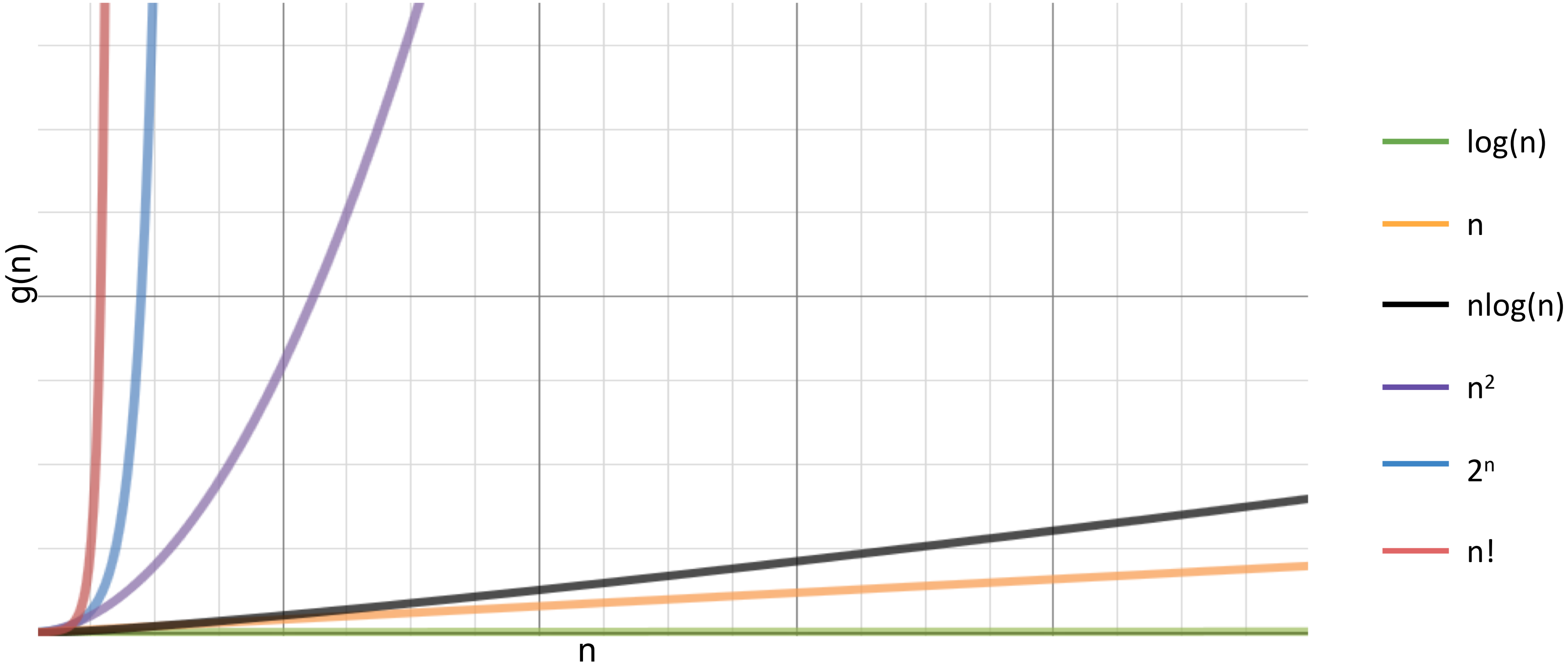
-This is nonsensical, therefore our assumption that c and n_0 exist must be wrong, and thus **n^2 is not ~ *O*(n)**.

Big O Runtimes

When comparing algorithms and their respective upper bounds, it’s important to know where a particular runtime falls in relation to another. The easiest way to determine this is to just look it up, however it’s useful to have a general idea of how some common functions relate to each other. This table shows the value of some common functions on n=128.

Big O	Operations When n = 128
$O(1)$	1
$O(\log(n))$	7
$O(n)$	128
$O(n*\log(n))$	896
$O(n^2)$	16,384
$O(n^3)$	2,097,152
$O(2^n)$	3.403×10^{38}
$O(n!)$	3.86×10^{215}

Popular Big O Functions



cousins of big O (job interview)

Definitions of Ω *and* Θ

- Let f and g be functions of n.

- We say "**f is in Big-omega of g**" (or "f is bounded below by g") and

  write "**f ~ Ω(g)**" if [∃ a constant c > 0] and [∃ a constant $n_0 \geq 0$] such that

  g is a lower bound

  $$(\forall\, n \geq n_0)\ \ (f(n) \geq c * g(n))$$

- We say "**f is in Big-theta of g**" and write "**f ~ Θ(g)**" if

  [∃ two constants c_1 and $c_2 > 0$] and [∃ a constant $n_0 \geq 0$] such that

  f is sandwiched below and above by g w/ different constants, both have same shape

  $$(\forall\, n \geq n_0)\ \ (c_1 * g(n) \leq f(n) \leq c_2 * g(n))$$