We can see that time required to search/sort grows with size of input. How do space/time needs of program grow with input size?

Let us focus on execution time. Space analysis is similar.

Execution time: count number of operations as function of input size

- Basic operation: arithmetic/logical operation counts as 1 operation
- Assignment: counts as 1 operation (operation count of righthand side expression is determined separately)
- Loop: number of operations/iteration * number of loop iterations
- Method invocation: number of operations executed between when a method is invoked and when invocation returns

Asymptotic complexity:
In most cases, we are only interested in the most significant (fastest-growing) term in the expression for execution time as a function of input size.

Asymptotic complexity:

- express required number of operations as a function of input size
- drop all terms except leading term, and ignore constant multiplier

Example: f(x) = 13*n + 8
f(x) = O(n)

Example: matrix multiplication
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      C[i][j] = C[i][j] + A[i][k] + B[k][j];

Problem size: n
Each execution of innermost assignment statement does 2 floating-point operations, 3 loads, 1 store, and some integer operations to index into the arrays.

Statement is executed \( n^3 \) times.

So total number of operations = \( 2n^3 + 3n^2 + n^2 + d \) (for some \( a,b,c,d \))

Asymptotic complexity: \( O(n^3) \)
Formal definition of $O()$ notation:

Let $f(n)$ and $g(n)$ be functions. We say that $f(n)$ is of order $g(n)$, written $O(g(n))$ if there is a constant $c > 0$ such that for all but a finite number of positive values of $n$,

$$f(n) \leq c \cdot g(n)$$

In other words, $g(n)$ sooner or later overtakes $f(n)$ as $n$ gets large.

Example: $f(n) = n + 5, g(n) = n$. We show that $f(n) = O(g(n))$.

Choose $c = 6$:

$$f(n) = n + 5 \leq 6 \cdot n \text{ for all } n > 0.$$ 

Example: $f(n) = 17n, g(n) = 3n^2$. We show that $f(n) = O(g(n))$.

Choose $c = 6$:

$$f(n) = 17n \leq 6 \cdot 3n^2 \text{ for all } n > 0.$$ 

Asymptotic complexity gives an idea of how rapidly space/time requirements grow as problem size grows.

Suppose we have a computing device that can execute 1000 operations per second. Here is the size of the problem that can be solved in a second, a minute and an hour by algorithms of different asymptotic complexity.

<table>
<thead>
<tr>
<th>Complexity</th>
<th>1 second</th>
<th>1 minute</th>
<th>1 hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>1000</td>
<td>60,000</td>
<td>3,600,000</td>
</tr>
<tr>
<td>$n \log n$</td>
<td>140</td>
<td>4893</td>
<td>200,000</td>
</tr>
<tr>
<td>$n^2$</td>
<td>31</td>
<td>244</td>
<td>1897</td>
</tr>
<tr>
<td>$3n^2$</td>
<td>18</td>
<td>144</td>
<td>1096</td>
</tr>
<tr>
<td>$n^3$</td>
<td>10</td>
<td>39</td>
<td>153</td>
</tr>
<tr>
<td>$2^n$</td>
<td>9</td>
<td>15</td>
<td>21</td>
</tr>
</tbody>
</table>

Subtlety: operation count might depend not only on size of input but also on the value of the input (look at linear search or binary search). For big-O determination, use worst-case scenario.

A graphical view of big-O notation:

To prove that $f(n) = O(g(n))$, find an $n_0$ and $c$ such that $f(n) \leq c \cdot g(n)$ for all $n > n_0$. 

\[ f(n) = O(g(n)) \]
Example: selection sort

```java
public static void selectionSort(Comparable[] a) { <!-- array of size n
   for (int i = 0; i < a.length; i++) { <!-- n iterations
      int MinPos = i;
      for (int j = i+1; j < a.length; j++) { <!-- n-i-1 iterations
         if (a[j].compareTo(a[MinPos]) < 0) <!-- comparison
            MinPos = j;
      }
      Comparable temp = a[i];
      a[i] = a[MinPos];
      a[MinPos] = temp;}
   }

   Total number of comparisons = (n-1) + (n-2) + ... + 1 = n(n-1)/2
   Complexity: O(n^2)
```

Example: linear search

```java
public static boolean linearSearch(Comparable[] a, Object v) {
   int i = 0;
   while (i < a.length) <!-- How many times does this loop execute???
      if (a[i].compareTo(v) == 0) return true; <!-- comparison
      else i++;
   return false;
```

Analysis of binary search is a little more difficult.

```java
public static boolean binarySearch(Comparable[] a, Object v) {
   .......
   middle = (left + right)/2;
   int test = a[middle].compareTo(v); <!-- comparison
   if (test < 0) left = middle+1;
   else
      if (test == 0) {
         return true;
      }
   else right = middle-1;
   //if we reach here, we didn’t find the object
   return false;
```

For searching and sorting algorithms, you can usually determine big-O complexity by counting comparisons.

Reason: you usually end up doing some fixed number of arithmetic/logical operations per comparison.
Number of iterations of while loop depends on values in array and
value of \( v \).

OK, let’s make worst case estimates: if array is of size \( n \), what is
the worst case number of iterations you make?

Easy to see that if size of array is \( n \), first iteration cuts the size of
the interval you need to look at to at most ceiling((n-1)/2). So if
\( c(n) \) is worst-case number of comparisons,

\[
\begin{align*}
c(1) &= 0 \\
c(2) &= 1 \\
c(n) &= 2c(n/2) + n
\end{align*}
\]

It can be shown that \( c(n) = O(n \log_2(n)) \).

Analysis of merge-sort:

\[
\begin{align*}
public \text{static} \text{Comparable}[] \text{mergeSort}(&\text{Comparable}[] \text{A, int low, int high}) \{
    \text{if} \ (\text{low} < \text{high} - 1) \ /\!\!/ \text{at least three elements}
    \hspace{1cm} \{ \text{int mid} = (\text{low} + \text{high})/2; \\
    \hspace{1cm} \text{Comparable}[] \text{A1} = \text{mergeSort}(\text{A, low, mid}); \quad \text{<- comparisons in method} \\
    \hspace{1cm} \text{Comparable}[] \text{A2} = \text{mergeSort}(\text{A, mid +1, high}); \quad \text{<- comparisons in method} \\
    \hspace{1cm} \text{return merge(}{\text{A1, A2}); \quad \text{<- comparisons in method}
    \hspace{1cm} . . . \hspace{1cm} . . .
    \}\hspace{1cm} \}
\end{align*}
\]

\[
\begin{align*}
c(1) &= 0 \\
c(2) &= 1 \\
c(n) &= 2c(n/2) + n
\end{align*}
\]

It can be shown that \( c(n) = O(n \log_2(n)) \).

Analysis of quicksort: tricky

\[
\begin{align*}
public \text{static} \text{void quickSort}(&\text{Comparable}[] \text{A, int l, int h}) \{
    \text{if} \ (l < h)
    \hspace{1cm} \{ \text{int p = partition(A,l+1,h,A[l]); \quad \text{<- comparisons} \\
    \hspace{1cm} \text{//move pivot into its final resting place} \\
    \hspace{1cm} \text{//swap A[p-1] and A[l]} \\
    \hspace{1cm} \text{Comparable temp = A[p-1];} \\
    \hspace{1cm} \text{A[p-1] = A[l];} \\
    \hspace{1cm} \text{A[l] = temp;} \\
    \hspace{1cm} \text{quickSort(A,1,p-1); \quad \text{<- comparisons} \\
    \hspace{1cm} \text{quickSort(A,p,h);}) \quad \text{<- comparisons}
    \hspace{1cm} . . . \hspace{1cm} . . .
    \}\hspace{1cm} \}
\end{align*}
\]

Incorrect attempt:

\[
\begin{align*}
c(1) &= 1 \\
c(2) &= 1 \\
c(n) &= (n-1) + 2c(n/2)
\end{align*}
\]

\[
\begin{align*}
\text{partition sorting the two partitioned arrays}
\end{align*}
\]
Remember: big-O is worst-case complexity.

Worst-case for quicksort: one of the partitioned array is empty, and the other has \((n-1)\) elements!

So actual recurrence relation is

\[
\begin{align*}
c(1) &= 1 \\
c(2) &= 1 \\
c(n) &= (n-1) + c(n-1)
\end{align*}
\]

partition sorting the two partitioned arrays

It can be shown that \(c(n) = \mathcal{O}(n^2)\)

On the average (not worst-case), quick-sort runs in \(n \cdot \log_2(n)\) time, which is why it is usually preferred in practice.

One approach to avoiding worst-case behavior: pick pivot carefully so it partitions array in half. Many heuristics for doing this, but none of them can guarantee that worst-case behavior will not show up.

Programs for the same problem can vary enormously in asymptotic efficiency.

\[
\begin{align*}
fib(n) &= fib(n-1) + fib(n-2) \\
fib(1) &= 1 \\
fib(2) &= 1
\end{align*}
\]

Here is a recursive program:

```c
static int fib(int n) {
    if (n <= 2) return 1;
    else return fib(n-1) + fib(n-2);
}
```

Here is a recursive program:

\[
\begin{align*}
fib(n) &= fib(n-1) + fib(n-2) \mid n \geq 2
\end{align*}
\]

Number of times loop is executed is bounded by \(n\).
Each iteration does some constant amount of work.

\[\Rightarrow\text{ Time complexity of algorithm } = \mathcal{O}(n).\]
In CS 211, you are expected to know the complexity of the algorithms we discuss in class.

You are also expected to know how to determine informally the asymptotic complexity (in closed-form) of toy recursive programs similar to merge-sort or binary search.

### Cheat Sheet for closed-form expressions

<table>
<thead>
<tr>
<th>Recurrence relation</th>
<th>Closed-form</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c(1) = 1$</td>
<td>$c(n) = D(n)$</td>
<td>Linear search</td>
</tr>
<tr>
<td>$c(n) = 1 + c(\frac{n}{2})$</td>
<td>$c(n) = D(\log(n))$</td>
<td>Binary search</td>
</tr>
<tr>
<td>$c(1) = 1$</td>
<td>$c(n) = D(\log(n))$</td>
<td>Binary search</td>
</tr>
<tr>
<td>$c(1) = 1$</td>
<td>$c(1) = c(2)$</td>
<td>Fibonacci</td>
</tr>
<tr>
<td>$c(n) = 2c(\frac{n}{2})$</td>
<td>$c(n) = D(\omega(n))$</td>
<td>Mergesort</td>
</tr>
<tr>
<td>$c(1) = 1$</td>
<td>$c(1) = c(2)$</td>
<td>Fibonacci</td>
</tr>
<tr>
<td>$c(n) = c(n-1)+c(n-2)+1$</td>
<td>$c(n) = D(2^n)$</td>
<td>Fibonacci</td>
</tr>
</tbody>
</table>