Asymptotic Complexity

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 = $a * n^3 + b * n^2 + c * n + d$ (for some a,b,c,d)

Asymptotic complexity: O(n^3)

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.

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,

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

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 $c = 6$:

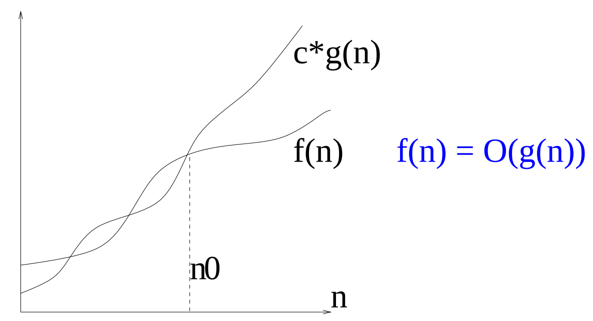
$f(n) = n + 5 \leq 6 * n$ for all $n > 0$.

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

Choose $c = 6$:

$f(n) = 17n \leq 6 * 3n^2$ for all $n > 0$.

A graphical view of big-O notation



To prove that $f(n) = O(g(n))$, find an $n0$ and c such that $f(n) \leq c * g(n)$ for all $n > n0$.

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.

Complexity	1 second	1 minute	1hour
n	1000	60,000	3,600,000
n log n	140	4893	200,000
n^2	31	244	1897
$3n^2$	18	144	1096
n^3	10	39	153
2^n	9	15	21

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.

Example: selection sort

```
  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

```
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;
```

Number of times while loop executes depends not only on size of input array a but on the values in it and the value of v.

Big-O complexity: worst-case scenario

For linear search, worst-case scenario is that all array elements are examined.

So complexity of linear search = O(n) where n is size of input array.

Analysis of binary search is a little more difficult.

```
  public static boolean binarySearch(Comparable[] a, Object v) {
    ......
    while (left <= right){ <-- how many times does this loop execute?
        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;
  }
```

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,

```
c(1) = 1  c(2) = 2
c(n) = 1 + c(ceiling((n-1)/2))
```

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

Note: any time you have a procedure whose complexity is less than $O(n)$, it means intuitively that procedure does not examine all of the input.

Analysis of merge-sort:

```
 public static Comparable[] mergeSort(Comparable[] A, int low, int high) {
   if (low < high - 1) //at least three elements
     {int mid = (low + high)/2;
      Comparable[] A1 = mergeSort(A, low, mid); <-- comparisons in method
      Comparable[] A2 = mergeSort(A, mid +1, high); <-- comparisons in method
      return merge(A1,A2);} <-- comparisons in method
   ....
```

```
c(1) = 0 c(2) = 1
c(n) = 2c(n/2) + n
```

It can be shown that c(n) = $O(nlog_2(n))$.

Analysis of quicksort: tricky

```
 public static void quickSort(Comparable[] A, int l, int h) {
    if (l < h)
        {int p = partition(A,l+1,h,A[l]); <--- comparisons
             //move pivot into its final resting place
             //swap A[p-1] and A[l]
             Comparable temp = A[p-1];
     A[p-1] = A[l];
     A[l] = temp;
               quickSort(A,l,p-1); <-- comparisons
               quickSort(A,p,h);}} <-- comparisons
```

Incorrect attempt:

```
c(1) = 1 c(2) = 1
c(n) = (n-1)    + 2c(n/2)
       -----      ------
     partition   sorting the two partitioned arrays
```

Why is this wrong?

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

```
fib(n) = fib(n-1) + fib(n-2)
fib(1) = 1
fib(2) = 1
```

Here is a recursive program:

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

Iterative Fibonnacci Code

fib (n) = fib(n-1) + fib(n-2) | n > 2

```
dad = 1
granddad = 1
current = 1;
for (i = 3; i <= n; i++) {
   granddad = dad;
   dad   = current;
   current = dad +granddad;
  }
printf("answer is " + current);
```

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

=> Time complexity of algorithm = O(n).

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!

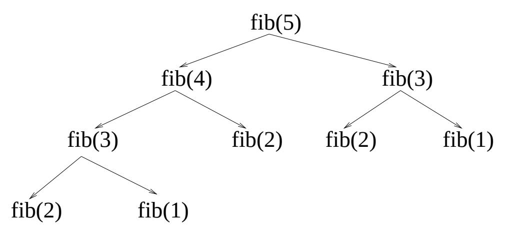
So actual recurrence relation is

```
c(1) = 1 c(2) = 1
c(n) = (n-1)   + c(n-1)
       -----     ------
     partition   sorting the two partitioned arrays
```

It can be shown that c(n) $= O(n^2)$

On the average (not worst-case), quick-sort runs in $n * log_2(n)$ 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.



c(n) = c(n-1) + c(n-2) + 2

c(2) = 1   c(1) = 1

It can be shown that $c(n) = O(2^n)$. Cost of computing value is exponential in the size of the input!

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

```
Recurrence relation      Closed-form       Example
-------------------      -----------       -------
c(1) = 1                 c(n) = O(n)       Linear search
c(n) = 1 + c(n-1)

c(1) = 1
c(n) = n + c(n-1)        c(n) = O(n^2)     Quicksort

c(1) = 1
c(n) = 1 + c(n/2)        c(n) = O(log(n))  Binary search

(1) = 1
c(n) = n + c(n/2)        c(n) = O(n)

c(1) = 1
c(n) = n + 2c(n/2)       c(n) = O(n*log(n)) Mergesort


c(1) = c(2) = 1                            Fibonacci
c(n) = c(n-1)+c(n-2)+1   c(n) = O(2^n)
```