Outline

- algorithm runtime analysis
- big-O notation
- figuring out big-O
- figuring out big-O for sorting algorithms
- big-O analysis in practice

Algorithm Analysis Review: motivation

- **selection sort, bubble sort, and insertion sort all sort an array**
- **which one of these is better?**
  - assuming the implementations already exist
- **all of these algorithms sort in place, so they all use the same amount of memory space**
- **so the only difference may be in the time they take**
- **but the time depends on both the computing system and the data**
  - for example, bubble sort is very fast on data that is already sorted, or if sorting data stored on magnetic tape

Algorithm Analysis Review: methods

- **for each algorithm, we will have a best-case time, a worst-case time, and sometimes an average-case time**
- **the time taken for sorting depends on the size of the array: sorting 2n elements takes longer than sorting n elements**
- **because computing systems vary (and for simplicity), all constant-time operations are assumed to take the same amount of time**
  - e.g. addition, multiplication, swapping elements, comparisons, each take constant time

Algorithm Analysis Review: notation

- **If, even in the worst case, an algorithm takes constant time, we say it is O(1)**
  - for example, the time to access an element of an array is constant, no matter the size of the array
- **If the worst case of an algorithm takes time proportional to the size of the input (conventionally, n), it is O(n)**
  - in the worst case, searching for a specific number in unsorted array of integers takes time proportional to the size of the array
- **If an algorithm in the worst case takes time proportional to the square of n, we say it is $O(n^2)$**
- **We will also see O(log n), O(n log n), O(√n), and so on**
- **O() always looks at the worst case as n gets very large**

Big-O Examples

- **in general, doing a fixed number of things is O(1)**
- **an algorithm to multiply all the odd numbers from 3..n takes time O(n). Here, n is the input to the algorithm and also the "size" of the problem**
- **searching through an array of n items in sequence is also O(n). Here, n is the size of the problem but the input to the algorithm are the array and the element to search for (the value n is not the input)**
- **dividing something into two nearly equal parts can only be done O(log n) times, logarithmic**
  - e.g. searching through a list of n items in sorted order, such as in a phone book, can be O(log n)
- **going through every element of an input is O(n), linear**
- **comparing every element of an input to every other element is O(n^2), quadratic**
- **trying every combination of n inputs is O(2^n), exponential**

Common Functions used with Big-O

- **common f(n) include:**
  - O(1) or constant time
  - O(log n) or logarithmic time:

    since $\log_2 n = k1 * \ln n = k2 * \log_{10} n$, any logarithm will do
  - O(n) or linear time
  - O(n log n), where a (worst-case) logarithmic operation is repeated at most n times
  - $O(n^2)$ or quadratic time
    - finding all pairs of matching items in an unsorted list of n items is $O(n^2)$
  - $O(n^3)$ or cubic
  - $O(2^n)$ or exponential
- ***hint: $\log_{10}(n)$ is approximately the number of digits in n: $\log_{10}(1000)$ is 3***

  $\log_2(n)$ is approximately the number of bits in n: $\log_2$ of 256 (binary 1 0000 0000) is 8
- **in-class exercise (everyone together): what is the value of f(n) for each of the above when n = 10? when n = 100?**

Big-O Growth with n

- O(1) means: changing n -- run time is unchanged. For example, array access time is independent of array size
- O(log n) means: doubling n -- the run time increases by a constant factor
- O(n) means: doubling n -- the run time doubles. For example, looking at every element of an array takes time proportional to the array size
- O(n log n) grows faster than O(n), but not nearly as fast as O(n^2)
- O(n^2) means: doubling n -- the run time increases by a factor of four
- O(2^n) means: increasing n by just 1 -- the run time doubles (or is multiplied by some constant factor)

Figuring out Big-O

- suppose the number of operations required for an algorithm is

  $2^n + n^3 + n \log n + 13$
- as n increases, the 2^n term will grow the fastest: $k * n^3 < 2^n$ for sufficiently large n, no matter how large k is
- ignoring slower-growing terms, this algorithm is $O(2^n)$
- constant factors can likewise be ignored: O(1) = O(5) = O(17), usually written as O(1). Similarly, O(n/3) = O(n)
- sometimes the problem size is expressed as a combination of numbers, for example the time to search a room of size m by n is proportional to m * n, and search algorithms take time O(m * n)
- big O always reflects the worst case
  - unless we specify “average case” or “best case” big-O

Example: Analyzing Bubble Sort

- **Sort an array of size n**
- **The outer loop repeats until the array is sorted**
  - best case: only one outer loop (if the array is already sorted)
  - worst case: in each outer loop, the smallest element moves one position towards the front of the array. If the smallest element was at the end of the array, the algorithm requires n outer loops
- **The inner loop compares all pairs in the array, so always takes linear time, O(n)**
- **n inner loops for each of n outer loops means bubble sort is $O(n * n) = O(n^2)$**
- **There is an optimization of bubble sort which considers that on the i-th outer loop, the last i elements of the array are already sorted, so the inner loop can stop at index n – i**
- **In-class exercise (initially in groups of up to 4, then in-class discussion with the entire class):**
  - what is the big-O runtime of bubble sort if the i-th inner loop stops at index n – i?

Friedrich Gauss: summing successive integers

- **Friedrich Gauss is a famous mathematician, with contributions in many different fields including statistics (a Gaussian curve)**
- **One formula for which he is well-known is very simple, and, the story goes, he figured it out in school**
- **what is the sum of all the integers from 1 to n?**
  - $1 + 2 = 3$
  - $1 + 2 + 3 = 6$
  - $1 + 2 + 3 + 4 = 10$
  - $1 + 2 + 3 + . . . + n = ?$
- **hint: consider how the result relates to the value $(n + 1) * n = n^2 + n$**
- **So if an outer loop has n iterations, and an inner loop has i iterations (or n – i iterations), the number of iterations of the inner loop is $1 + 2 + 3 + . . . + n$, and so $O(n^2 + n) = O(n^2)$**
- **This information is useful in analyzing the optimized bubble sort, as well as selection sort and insertion sort**

Big-O analysis in practice: 1/3

```
n = 100000;  // one hundred thousand
startTimer();
for (int i = 0; i < n; i++) {
    count++;
}
stopTimerAndPrint("example 1", n);
```

- how much longer than the first loop does the next loop take?

```
n = 10000000;  // ten million
startTimer();
for (int i = 0; i < n; i++) {
    count++;
}
stopTimerAndPrint("example 1", n);
```

Big-O analysis in practice: 2/3

```
n = 100000;  // one hundred thousand
startTimer();
for (int i = 0; i < n; i++) {
    count++;
}
stopTimerAndPrint("example 1", n);
```

- **how much longer than the first loop does the next loop take?**

```
n = 100000000; // one hundred million
startTimer();
for (int i = 0; i < n; i++) {
    count++;
}
stopTimerAndPrint("example 1", n);
```

Big-O analysis in practice: 3/3

```
n = 1000;
startTimer();
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    count++;
  }
}
stopTimerAndPrint("example 2", n);
n = 10000;
startTimer();
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    count++;
  }
}
stopTimerAndPrint("example 2", n);
```

- **how much longer than the first loop does the second loop take?**