Outline

- insertion sort
- algorithm runtime analysis
- principles of recursion
- recursive factorial
- binary search
- using recursions to print the digits of integers

Insertion Sort

- in selection sort, find the smallest element, and put it in the next position
- in insertion sort, take the next element, and put it in the right place
- trivia: card players often use insertion sort to arrange their decks
- the sub-array at the beginning of the array is already sorted, just as in selection sort
- in selection sort, the next element e is always taken from the next index in the array
- elements in the sorted part of the array that are greater than e are shifted up one position, to free the position where element e will be stored

  see animations at https://en.wikipedia.org/wiki/Insertion_sort
- in-class exercise (in groups of 2 or 3): write code to implement insertion sort

Introduction to Algorithm Analysis: 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

Introduction to Algorithm Analysis: analysis

- **for each algorithm, we will have a worst-case runtime, best-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 to keep things simple, all constant-time operations are assumed to take the same amount of time**
  - e.g. this analysis assumes that addition, multiplication, swapping elements, and comparisons, each take the same time

Introduction to Algorithm Analysis: 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
  - the runtime is always the worst-case runtime, unless best case or average case is explicitly stated
- **If the worst case of an algorithm takes time proportional to the size of the input, it is O(n), with n the size of the input**
  - in the worst case, finding out whether an unsorted array of integers contains a specific number 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**
  - unless we are specifically looking for the best case or average case

Runtime Analysis of Selection Sort

- **selection sort: the inner loop does a linear search to find the minimum element in elements i..n-1**
  - the linear search takes linear time, O(n-i)
- **the linear search is repeated n times, so the entire algorithm takes time $O(n^2+ni) = O(n^2)$, quadratic time**
- **this is an approximation: n-i gets smaller as i gets larger**
- **runtime analysis uses these approximations: a more accurate formula would be $(n^2+n)/2 = n^2/2 + n/2$, but O() analysis only considers the largest term, in this case $O(n^2)$, and ignores factors such as ½**

Computing Approximate Logarithms

- **given x = 2^y, then y is the logarithm (base 2) of x**
- **given x = 10^y, y is the logarithm (base 10) of x**
- **if $\log_{10} x = y$, then $\log_{10}(10x) = y + 1$**
  - that is, for every additional digit in a number, the $\log_{10}$ increases by 1
  - $\log_2$ increases by 1 for every additional bit in a number
- **a rough estimate of the log of a number is then just the number of digits (or bits) in the number:**

$$6 < \log_{10} 1{,}234{,}567 < 7$$

$$10 < \log_2 110\ 0011\ 1011 < 11$$

- **in ICS 211 we will assume that the log of a number is the same as the number of its digits**

Principles of recursion

- **many problems have:**
  - an obvious solution when they are small, and
  - a way to make a big problem into a smaller problem
- **if this is the case, we can solve any size problem!**

- **for example, the factorial function n! is defined to be 1 for n=1, and n * (n-1)! for larger n**
- **so when n is 1, the factorial is 1**
- **when n is greater than one the factorial is n * the factorial of n-1**
- **and when n is greater than 1, we can make the problem smaller by computing (n-1)!, then multiply the result by n**

Example of recursion

```
static int factorial(int n) {
  if (n == 1) {     // base case
    return 1;
  } else {          // recursive case
    return n * factorial (n - 1);
  }
}
```

Fun facts about recursion

- **every loop can be replaced by recursion**
  - not every recursion can be replaced by a loop (unless there is an additional data structure)
- **some things are much simpler to compute recursively:**

```
static int ackermann(int m, int n) {
    if (m == 0) return n + 1;
    if (n == 0) return ackermann(m - 1, 1);
    return ackermann(m - 1, ackermann(m, n - 1));
}
```

- **sometimes recursion is the simplest way of thinking of problems**
- **in-class exercise: trace an execution of ackermann(2, 1)**

Recursive thinking: searching for a value in an array

- can I break up a problem into one or more similar and smaller problems?
- example: find a value in a sorted array
- one solution is linear search: search through all the numbers with while or for loops
- this is an iterative (not a recursive) solution
- linear search doesn't even need the array to be sorted
- can we do better?

Recursive thinking: binary search

- **we can do better than linear search when the array is already sorted**
- **when the array only has 1 element, I can compare what I am looking for (x) to the one element (e), and will immediately know if x is in the array**
- **when the array has more than one element, I look at the value e in the middle of the sorted array:**
  - if x.equals(e), then x is in the array
  - if x < e, then if x is in the sorted array, it is to the left of e
  - otherwise x > e, and if x is in the array, it is to the right of e
- **we use two indices to keep track of which part of the array may still have the value x**
  - first and last, or start and end
  - define the sub-array that may still have x

Recursive implementation: binary search

```
public static boolean search(char[] a, char x,
                             int first, int last) {
  int middle = (first + last) / 2;
  if (x == a[middle]) return true;   // found
  if (first >= last)  return false;  // not found
  if (x < a[middle])  return search(a, x, first, middle - 1);
                      return search(a, x, middle + 1, last);
}
```

- in any given call, x can only be in the sub-array a[first]..a[last]
- different versions of binary search might return the index (or -1 for not found), others the value that was found (or null for not found)

Principles of recursion

- **many problems have:**
  - an obvious solution when they are small (**base case**), and
  - a way to make a big problem into a smaller problem (**recursive case**)
- **if this is the case, we can solve any size problem!**
- **The base case immediately returns the answer we can figure out right away**
  - there may be multiple base cases, for example in binary search one base case is where we find the value, the other base case is where we didn't find the value and the sub-array has size 1
- **The recursive case(s) recursively call the method with parameters such that the problem becomes smaller**
- **for example in binary search, the sub-array is at most half the size of the original array**
- **in factorial, the number is one less than the original number**

Proving that a recursive method terminates

- **prove that every recursive case gets closer to a base case**
- **for example, in binary search:**
  - the base cases are:
    - when the value is found, or
    - first >= last
  - on each recursive call, either first moves towards last, or last moves towards first, by at least one: because middle is not less than first, middle + 1 is bigger than first, and because middle is not greater than last, middle - 1 is less than last
- **in general, to be sure our code terminates we must prove that the recursive case approaches the base case**
- **for factorial, a correctness depends on whether the base case is (n <= 1) or (n == 0)**
- **if the base case is n==0, when n < 0 the recursion never terminates!**

Proving that recursive factorial terminates

- n factorial, written !n, is n * (n – 1) * (n – 2) * … * 2 * 1
- another way of writing this is to say !n = n * !(n – 1)
- this second formulation is recursive, and gives us an intuitive recursive implementation:

```
public int factorial(int n) {
  if (n <= 1) return 1;
  return n * factorial(n – 1);
}
```

- if n <= 1, the call terminates immediately
- otherwise, the parameter to the recursive call is always closer to the base case (closer than n)

Example of recursion: printing integers

- **print an integer in binary (base 2) or in any other base (base 8, base 10)**
- **must print the most significant bit or digit first**
  - but modulo only gives us access the least significant bit or digit
- **very easy to do recursively:**
  - first, recursively print the number divided by the base
  - then print the last bit or last digit

Example of recursion: printing integers

- **very easy to print integers recursively:**

```
private static void printInt(int toPrint, int base) {
    if (toPrint < base) {            /* first (or only) bit/digit */
        System.out.print(toPrint);
    } else {
        /* print the bits/digits before this one */
        printInt(toPrint / base, base);
        /* print last bit or digit */
        System.out.print(toPrint % base);
    }
}
```

- **this code reaches the base case (toPrint < base) before printing anything**
- **once the base case is reached, the remaining bits/digits are printed after the inner call completes**

Tracing printInt

to print an integer in base 10:

`printInt(5432, 10)` calls

`printInt(543, 10)`, which calls

`printInt(54, 10)`, which calls

`printInt(5, 10)`. Here the code goes through the base case, prints 5 and returns

`printInt(54, 10)` now prints 54 % 10, which is 4

`printInt(543, 10)` can now print 543 % 10, i.e. 3

`printInt(5432, 10)` finally prints 5432 % 10, which is 2

- **the result is that 5432 is printed**

```
private static void
 printInt(int toPrint, int base) {
  if (toPrint < base) { // first digit
    System.out.print(toPrint);
  } else {  // recursive case
    printInt(toPrint / base, base);
    System.out.print(toPrint % base);
```

The Java system stack

- **a stack is a data structure that can grow and shrink as needed**
- **at any given moment during execution, the Java stack holds information about what executions are suspended:**
  - what calls have been made, but not completed
- **for recursion, the stack must have:**
  - return address
  - parameters
  - local variables
- **the stack also holds exception handlers, i.e. information about `try/catch` blocks**
  - throwing an exception means popping method information (*frames*) from the stack until we find a matching `catch`
- **the Java system stack is not visible to the Java programmer**

In-class exercise

- write a recursive method
- to print integers
- printing a comma every 3 digits
- for example, 1,234,567,890