CS50 Section. Week 4. 9/20/16.

Tuesdays 4:00-5:30pm, CGIS S-040

> Nicholas Boucher nboucher@college.harvard.edu

Important links

- Sorting algorithm visualizations:
  - Bubble Sort
  - Insertion Sort
  - Merge Sort
- C language reference: https://reference.cs50.net/
- This week's material on Study50: Study50
- CS50 Discuss: https://cs50.harvard.edu/discuss
- CS50 Style Guide: https://manual.cs50.net/style/

Section Agenda

1. Suggestions from Last Pset
2. Review of Previous Concepts
3. Searching
4. Sorting
5. Recursion

Suggestions from Last Pset

Use Style50

Most of the general issues that I saw on psets could have been caught by style50. Everyone seems to be doing a good job of running check50, but do be sure to run style50 and correct any issues before submitting your code.

Comments

In the last pset, many people either entirely lacked or used significantly fewer comments than is expected. Be sure to document what you are doing in your code using comments - you should not just be explaining *what* but also *why* your code is the way it is.

Also, do not forget to include the required comment stubs at the top of each file and before each function you write (except perhaps main). The following is excerpted from the Style Guide:

Comments at the Top of Each File

```
/**
 * hello.c
 *
 * David J. Malan
 * malan@harvard.edu
 *
 * Says hello to the world.
 */
```

Comments Before Each function

```
/**
 * Returns n^2 (n squared).
 */
int square(int n)
{
    return n * n;
}
```

Magic Numbers

Magic numbers are any numbers which are "hardcoded" into your code. As a general rule, we try to avoid these hardcoded constants at all costs, since it:

- makes code harder to read and understand
- makes it more difficult and error-prone to change these values later

Instead of using magic numbers, we should instead use #define statements at the top of each pertinent file.

Spacing

It is important that your code stylistically conforms to the normal conventions of C. One recurring issue in the past pset was improper spacing between variables, costants, and operators. There should always be a space on either side of an operator. For example, adding one to the variable x should look like this:

```
x = x + 1;
```

NOT this:

```
x = x+1;
```

Debug50

Debug50 is a great way to locate issues within your program. It is a graphical debugger that will allow you to see the current value of variables at any point in the program. Use debug50 by placing breakpoints on the relevant lines of code (where you think the bug might be) and executing `debug50 <program name>`.

Review of Previous Concepts

Modulo

Modulo is a basic mathematical operator in C which calculates the "remainder" from division. It is denoted with the % symbol. For example, `7 % 3 = 1`, since `7 / 3 = 2 **remainder 1**`.

ASCII

ASCII, or "American Standard Code for Information Interchange", is a standard that equates human-readable sysmbols (such as the alphabet)

to numbers which can be represented in binary. C stores `char`s as ASCII `int`s under the hood, which is why you can do arithmetic on `char`s.

Functions

Functions are a way to *factor out* some of your code into named, reusable chunks. Functions can take an arbitrary numner of arguments (zero included), which allow you to pass information into them. Functions also have the ability to return a single up to one value.

Functions are created using the following syntatic form:

```
<return type> <name>(<argument type> <argument name>, ...)
{
    <function code here>
}
```

Function prototypes

Say you want to use a function besides `main`. It's not enough to just write the function; C reads top-to-bottom so you'll need to tell C some information about the function ahead of time.

```
#include <stdio.h>

// function prototype
// <return type> <name>(<argument type>, ...)
// this is the same as in the function definition, except
// you don't specify the argument names
float months_to_years(int);
```

```c
int main(void)
{
    // prints 1.50
    // (the "%.2f" means to print a float with
    // 2 decimal places)
    printf("%.2f\n", months_to_years(18));
}

// function definition
//<return type> <name>(<argument type> <argument name>,...)
// here's where you *actually* implement the function
float months_to_years(int months)
{
    return years / 12.0;
}
```

By the way, if you were actually writing this code, you wouldn't include any of these comments. They're just for illustration here

Command Line Arguments

Values can be passed into a program from the command line. These are interpreted as arguments to the `main` function. Traditionally, we have written `main` like this:

```c
int main(void) {
    <code here>
}
```

If we want to be able to accept values from the command line, however, we must change the structure of `main` to look like this:

```c
int main(int argc, string argv[])
```

```
{
    <code here>
}
```

`argc` is the argument count - the number of arguments passed to the program. `argc` is always greater than or equal to 1, since the name of the program is counted as the first argument.

`argv` is the argument vector - the actual values passed into the program. It is an array of strings, so it should be accessed using the array indexing syntax (i.e. `argv[n]`, where n is the index of the element you are trying to access). `argv[0]` is always the name of the program being executed.

Arrays

Arrays let you store a bunch of related pieces of data in a clean, organized way.

Here's the primary way to create an array:

```
// <datatype> <name>[<size>];
string my_classes[4];
// you don't have to fill in all the elements of the array
my_classes[0] = "CS50";
my_classes[1] = "Ec 10";
my_classes[2] = "Expos 20";
my_classes[3] = "Math 55";
```

You can use this shorthand if you know all the elements of the array ahead of time. Notice that C will infer the array size for you, so you don't need to write it!

```
// <datatype> <name>[] = {elements};
int important_years[] = {1636, 1776, 2015};
```

You can access and change the elements of an array like this:

```
// prints "This is CS50." (without quotes)
printf("This is %s.\n", my_classes[0]);

// prints 1787
important_years[1] = 1787;
printf("%i\n", important_years[1]);
```

Strings

Strings of text are really just arrays of characters. You can work with them just like normal arrays, for the most part:

```
// prints "u" (without quotes)
string band = "Young the Giant";
char third_letter = band[2];
printf("%c\n", third_letter);
```

Note that strings are immutable - that is, you cannot change a string once it has been created. For instance, the following code will cause an error.

```
band[2] = 'u';
```

Do keep in mind that *replacing* a string with a complete new string is not the same as modifying an existing string, which is why we can do this:

```
band = "Green Day";
```

Arrays, the General Case

So far, we have given examples of arrays using `int[]` and `char[]` (read as `string`). Know, however, that arrays can hold any single data type, so constructs like `string[]` and `float[]` are perfectly valid.

Asymptotic Notation (O, Ω)

We use specific notation to describe the theoretical *running time* of a given algorithm. `O` typically refers to the *Upper Bound* of the running time and `Ω` (Omega) typically refers to the `Lower Bound` of the running time. You can think of these as best case and worst case running time - or *time complexity* - scenarios.

As we introduce new algorithms, we will practice determining the Upper and Lower runtime complexity bounds.

Searching

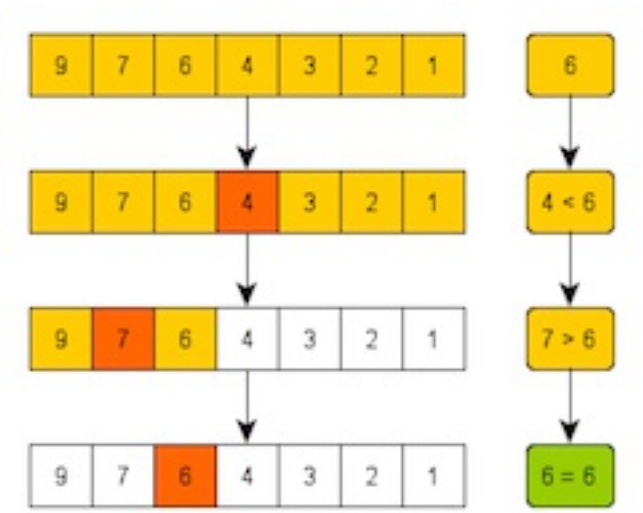
Linear Search

Linear Search is the most basic type of search. It involves simply examing each element in an array in the order that they appear and comparing it to the value we are looking for.

Excercise: Implement a linear search function for searching arrays of `int`s.

Binary search

Binary search finds a value in a *sorted* array. You cut the problem in half at each stage by eliminating the half of the array that couldn't possibly contain the element you're looking for.




```
while there are still elements to search:
    find middle element
    if middle element = what we're looking for:
        return true
    if middle element < what we're looking for:
        look in lower half
    if middle element > what we're looking for:
        look in upper half

return false (give up)
```

Think back to David's first lecture containing the phone book example.

Sorting

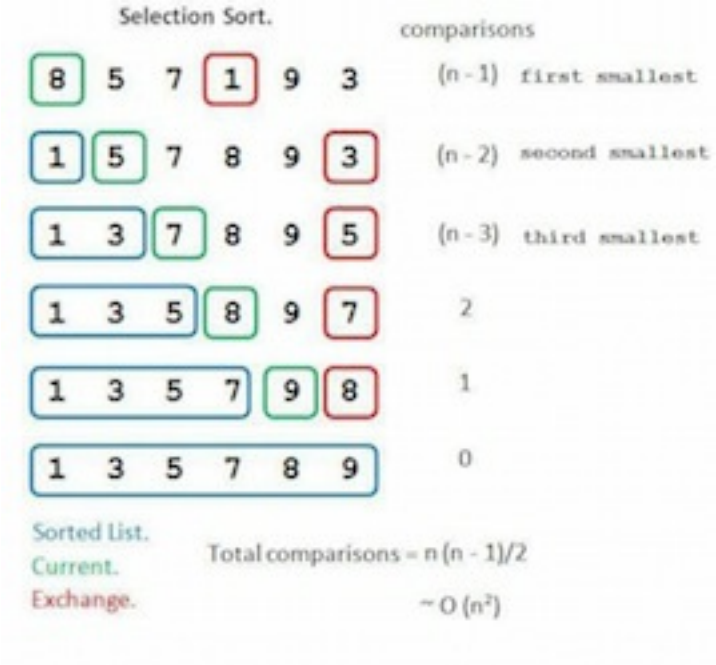
Bubble sort

In a sorted array, every element will be smaller than the one to its right, so this sort swaps two neighboring elements if they're out of order. This way, the biggest elements end up *bubbling* to the top.


```
for each element X in the array:
    for each remaining element Y in array:
        swap X and Y if X > Y
```

Selection sort

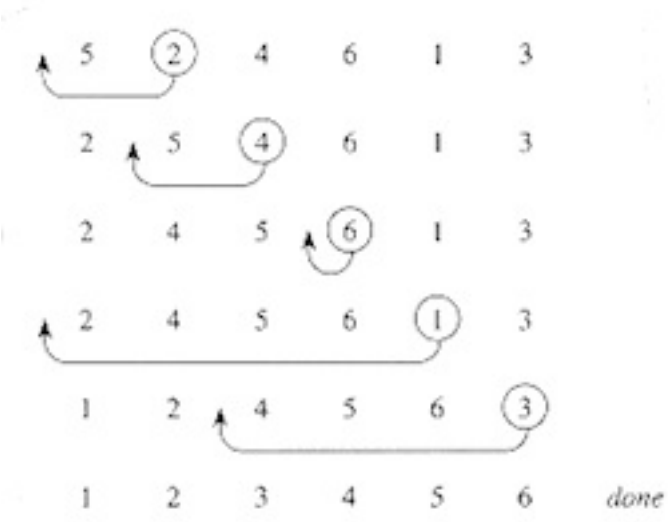
This sort *selects* the smallest elements remaining in the array and swaps them so they're in the front.




```
for each element in the array:
    find the smallest element in the rest of the array
    swap that element with the current element
```

Insertion sort

This sort sorts the array as it sees it, *inserting* each element in its proper place in the array.

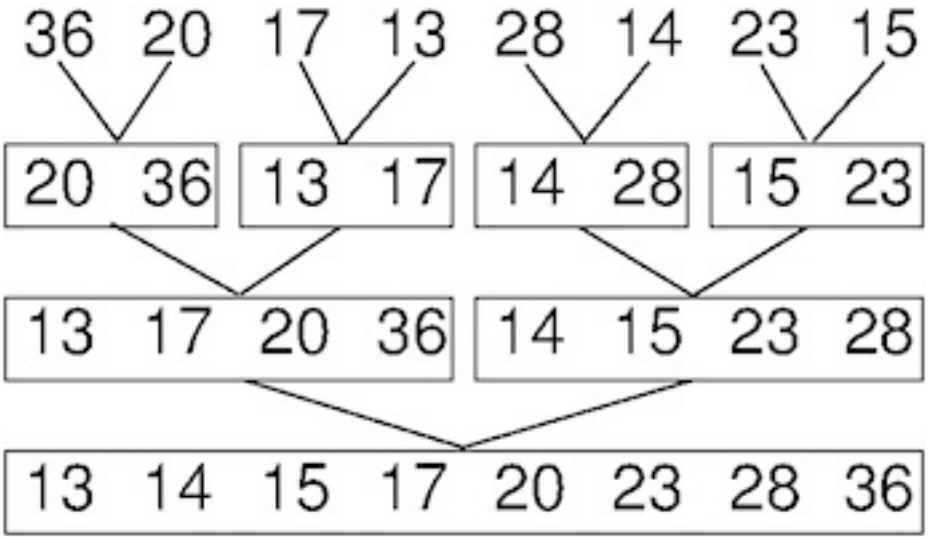



```
for each element in the array:
    look at the element X directly to our right
```

```
    shift all elements on our left to the
          right if they're > X
    put X in the vacated spot
```

Merge sort

This sort recursively sorts its sub-arrays and then *merges* them together to produce one larger sorted sub-array.




```
function sort:
    sort left half
    sort right half
    merge two halves

function merge:
    create temporary array
    while elements in left & right sub-arrays:
        compare first elements of two sub-arrays;
            place smaller one in temp array
    dump any remaining elements in longer sub-array
            in temp array
    copy temp array onto original array
```

Algorithm cheatsheet

Name	What it does	Worst-case	Best-case
Linear search	Finds an element in a list by searching left-to-right	O(n)	Ω(1)
Binary search	Finds an element in a sorted list using divide-and-conquer	O(log n)	Ω(1)
Bubble sort	Sorts a list by bubbling biggest elements to end	O(n^2)	Ω(n)
Selection sort	Sorts a list by moving smallest elements to front	O(n^2)	Ω(n^2)
Insertion sort	Sorts a list by moving elements to properly sorted place	O(n^2)	Ω(n)
Merge sort	Recursively sorts a list by partitioning and merging	O(n log n)	Ω(n log n)

More at http://www.bigocheatsheet.com/.

Recursion

Recursive code is code in which a function makes a call to itself. It is sort of the *Inception* (excuse my now outdated pop culture references) of code. Recursion can be a difficult concept to wrap your head around when you first see it.

Recursive algorithms need to have at least one *base case* (in which the function does not call itself) and at least one *recursive case* (in which the function does call itself).

Some programming problems are simply easiest/cleanest to implement recursively. Beyond that, recursive code is beautiful once you get used to seeing it.

Here is an example of implementing the Fibonacci sequence in C using recursion:

```
int Fibonacci(int n)
{
   if ( n == 0 )
      return 0;
   else if ( n == 1 )
      return 1;
   else
      return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```