Week 3, continued

This is CS50. Harvard University. Fall 2015.

Anna Whitney

Table of Contents

1. Introduction

- Today's lecture comes to you filmed in Hauser Studio in Widener Library.

- Because of the green screen in this studio, CS50's production team can put any background they want behind David (like a bunch of adorable puppies).
- Looking ahead to Problem Set 3, you'll be implementing the Game of Fifteen, which you might have played on a physical plastic puzzle as a child.
  # In the Hacker edition, you'll be tasked to not only make it possible for a user to play the Game of Fifteen, but also to implement God Mode, which solves the puzzle for the user!

2. Merge Sort

- Recall from last lecture that we had several sorting algorithms (bubble sort, selection sort, insertion sort) that all run in **O(n^2)**.
- We also showed an example of **merge sort**, which runs much faster, and which we've teased leverages the same sort of divide-and-conquer logic as our binary search method of finding Mike Smith in the phonebook.
- Thinking back to the pseudocode from week 0:

```
pick up phone book
open to middle of phone book
look at names
if "Smith" is among names
    call Mike
else if "Smith" is earlier in book
    open to middle of left half of book
    go to line 3
else if "Smith" is later in book
    open to middle of right half of book
    go to line 3
else
    give up
```

# Notice how we have an iterative, or looping, approach with lines 8 and 11, where we go back to line 3 and repeat the process again and again.

- But let’s simplify this and change the nature of our program:

```
pick up phone book
open to middle of phone book
look at names
if "Smith" is among names
    call Mike
else if "Smith" is earlier in book
    search for Mike in left half of book ❶

else if "Smith" is later in book
    search for Mike in right half of book ❷

else
    give up
```

# Now we seem to have an incomplete instruction. How do we `search for Mike` in lines 7 and 10? Well, we just use this exact same program, starting at the top. Lines 4 and 12 makes sure that the program will end. With lines 7 and 10, we have a **recursive** algorithm that "calls" itself.

- Using recursion, we can express merge sort as follows:

```
On input of n elements
```

```
if n < 2
    return
else
    sort left half of elements
    sort right half of elements
    merge sorted halves
```

- The base case - `if n < 2` - ensures we don't loop forever, because a list of 0 elements or of 1 element is trivially sorted.
- It seems like this can't possibly be enough information to sort a list, but remember that `sort left half of elements` and `sort right half of elements` are recursive calls to this same algorithm!
- Consider a list of eight elements, which we'll now envision as stored in an array:

```
---------------------------------
| 4   8   6   2   1   7   5   3 |
---------------------------------
```

- So let's run through merge sort. We first need to `sort left half of elements`, so we divide the list of 8 elements in half:

```
---------------------------------
| 4   8   6   2 | 1   7   5   3 |
---------------------------------
```

- In order to sort the left half, we start over from the beginning of the algorithm, and again proceed to the `else` block, which means we are sorting the left half of the left half:

```
---------------------------------
| 4   8 | 6   2 | 1   7   5   3 |
---------------------------------
```

- We call the algorithm again, but this time we have just one element, 4, so the base case is called - `[4]`, this list of size 1, is already sorted. We move on to 8, which is also sorted by itself:

```
---------------------------------
| 4 | 8 | 6   2 | 1   7   5   3 |
---------------------------------

---------------------------------
```

```
| 4 | 8 | 6   2 | 1   7   5   3 |
---------------------------------
```

- Now the left half of our list `[4,8]` is sorted and the right half is sorted, so we need to merge the two halves.

  # First, we allocate some additional memory, big enough to fit the secondary list.

  ```
  ---------------------------------
  | 4 | 8 | 6   2 | 1   7   5   3 |
  ---------------------------------
  ---------
  |       |
  ---------

  ---------------------------------
  |   | 8 | 6   2 | 1   7   5   3 |
  ---------------------------------
  ---------
  | 4     | // bring the 4 forward
  ---------

  ---------------------------------
  |   |   | 6   2 | 1   7   5   3 |
  ---------------------------------
  ---------
  | 4   8 | // and then the 8
  ---------

  ---------------------------------
  | 4   8 | 6   2 | 1   7   5   3 | // now the left half of the left half
   is sorted
  ---------------------------------
  ```

- The next step is to mentally rewind and remember that we need to sort the right half of the left half, since we just sorted the left half of the left half. So the same process takes place; `[6]` and `[2]` are already sorted, so we merge these two lists:

```
---------------------------------
| 4   8 | 6 | 2 | 1   7   5   3 |
---------------------------------
        ---------
        |       | // allocate some more memory
```

```
        ---------

---------------------------------
| 4   8 | 6 |   | 1   7   5   3 |
---------------------------------
        ---------
        | 2     | // bring down the 2 first
        ---------

---------------------------------
| 4   8 |   |   | 1   7   5   3 |
---------------------------------
        ---------
        | 2   6 | // then the 6
        ---------

---------------------------------
| 4   8 | 2   6 | 1   7   5   3 |
---------------------------------
-----------------
|               | // now we merge the 2 halves of 2 elements each
-----------------

---------------------------------
|       |       | 1   7   5   3 |
---------------------------------
-----------------
| 2   4   6   8 | // bring down the 2, then 4, then 6, then 8
-----------------
```

- We return to the right half and quickly repeat the same process (and again remember that we need to bring merged elements forward before putting them back in the list):

```
---------------------------------
| 2   4   6   8 | 1   7   5   3 | // sort right half
---------------------------------

---------------------------------
| 2   4   6   8 | 1   7 | 5   3 | // sort left half of right half
---------------------------------

---------------------------------
| 2   4   6   8 | 1 | 7 | 5   3 | // 1 is sorted
```

```
---------------------------------

---------------------------------
| 2   4   6   8 | 1 | 7 | 5   3 | // 7 is sorted
---------------------------------
                ---------
                |       | // more memory
                ---------

---------------------------------
| 2   4   6   8 |   |   | 5   3 |
---------------------------------
                ---------
                | 1   7 | // bring down 1, then 7
                ---------

---------------------------------
| 2   4   6   8 | 1   7 | 5 | 3 | // 5 and 3 are sorted
---------------------------------
                        ---------
                        |       | // more memory
                        ---------

---------------------------------
| 2   4   6   8 | 1   7 |   |   |
---------------------------------
                        ---------
                        | 3   5 | // bring down 3, then 5
                        ---------

---------------------------------
| 2   4   6   8 | 1   7 | 3   5 | // merge halves of right half
---------------------------------
                -----------------
                |               |
                -----------------

---------------------------------
| 2   4   6   8 |       |       |
---------------------------------
                -----------------
                | 1   3   5   7 | // bring down 1, then 3, then 5, then 7
                ----------------
```

- By recursively calling this algorithm, we’ve sorted both halves, and we can merge them just as we’ve done within each half:

```
---------------------------------
| 2   4   6   8 | 1   3   5   7 | // merge left and right half
---------------------------------
---------------------------------
|                               |
---------------------------------

---------------------------------
| 2   4   6   8 |     3   5   7 | // 1 is smallest, so we bring it down
---------------------------------
---------------------------------
| 1                             |
---------------------------------

---------------------------------
|     4   6   8 |     3   5   7 | // now 2
---------------------------------
---------------------------------
| 1   2                         |
---------------------------------

---------------------------------
|     4   6   8 |         5   7 | // then 3
---------------------------------
---------------------------------
| 1   2   3                     |
---------------------------------

---------------------------------
|               |               | // and so on, taking smaller option
 until entire list is sorted
---------------------------------
---------------------------------
| 1   2   3   4   5   6   7   8 |
---------------------------------
```

- If we look back at the structure we’ve moved through, we can see the divide-and-conquer nature of this algorithm:

```
---------------------------------
|   |   |   |   |   |   |   |   |
---------------------------------

---------------------------------
|       |       |       |       |
---------------------------------

---------------------------------
|               |               |
---------------------------------

---------------------------------
|                               |
---------------------------------
```

- Recalling the logarithmic running time of our divide-and-conquer search algorithm (binary search), we can see that there's something of a logarithmic nature to merge sort, although it's not just log *n*.
- Note that when we talk about log *n* in this class, we're referring to $\log_2 n$, not $\log_{10} n$ or ln *n* as you may be familiar with from math classes (although when we discuss big-O logarithmic running time, these are all equivalent, because logs of different bases can be interconverted with a constant and constants are irrelevant in big-O).
- We divided our input in half log(*n*) times - with 8 numbers, this is $\log_2$ (8), or 3 divisions.
- Each time we split the list, we had to merge *n* elements together (touching all 8 elements at each level to order the sublists correctly).
- With log *n* stages, and *n* steps of work in each stage, merge sort has a total of $O(n \log n)$ running time.
- Recall that bubble sort, insertion sort, selection sort and so on all run in $O(n^2)$ time, so just as our O(log *n*) search algorithm was much faster than linear, or O(*n*), search, merge sort is much faster than our previous sort algorithms.
- Let's look back at our algorithm and express some of this more formally:

```
On input of n elements
    if n < 2
        return
    else
```

```
        sort left half of elements
        sort right half of elements
        merge sorted halves
```

- Our base case, `if n < 2, return`, runs in constant time, or O(1) - some fixed number of steps regardless of the value of *n*.
- This means we can say that *T*(*n*) (the running time of our algorithm given an input of size *n*) is equal to O(1) if *n* < 2.

• What about in the `else` case? How long does each of those steps take?

```
On input of n elements
    if n < 2
        return
    else
        sort left half of elements
        sort right half of elements
        merge sorted halves
```

- If the running time of sorting the whole list is some *T*(*n*), then sorting the left half runs in *T*(*n*/2), as does the right half.
- Merging the sorted halves takes O(*n*), because we have to touch every element as we merge.
- Combining these, when *n* is greater than or equal to 2, we have the following:

```
T(n) = T(n/2) + T(n/2) + O(n)
```

- This formula, if calculated out to a closed form, gives O(*n* log *n*) - you're not responsible for the precise mathematical details.

• In this video[1], then-Senator Obama is asked what would be the most efficient way to sort a million 32-bit integers, and responds that "bubble sort would be the wrong way to go."

3. Compiling

• Recall our simple C program from previous weeks:

[1] https://www.youtube.com/watch?v=k4RRi_ntQc8

```
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
}
```

- We previously said that when we compile this program, our source code is turned by the compiler into object code, composed of 0s and 1s that the computer can understand.
- Turns out there are some intermediate steps! Our code is first turned into **assembly code**, composed of simple machine instructions that interact with memory directly.
- So, our process now looks as follows: **source code** (e.g., `hello.c`) is **compiled** into **assembly code** (`hello.s`), which is then **assembled** into **object code** (`hello.o`).
- `clang`, like many compilers, performs all these steps together, and typically does not output the intermediate files.
- When we call a function declared in a library, like `printf` in `stdio.h`, we have to pull in the actual code of that function from a file `stdio.c`.
  - # This code has already been compiled and assembled into an object code-like format.
  - # After our code has been compiled and assembled into object code, the object code for any libraries we're using has to be **linked** with our object code to create an **executable file**.
- So the full process consists of the source code `hello.c` being compiled into assembly code, `hello.s`, which is then assembled into object code, `hello.o`, which is then linked with `stdio.o` (or other object code-like library files, depending on which header files you've included with `#include` in your source code) to produce an executable that you can run as `./hello`.

4. Bitwise Operators

- Thus far, we've dealt with data of type `char`, `int`, `float` and so on, but all of these data types are at least 8 bits - we haven't been manipulating individual bits (`0` or `1`).
- C does let us access and manipulate individual bits using specific syntax:

```
&      |      ^      ~      <<      >>
```

- `&` is bitwise AND (not to be confused with `&&`, logical AND), which gives `1` if both of its arguments are `1`.
  - `0 & 0` is equal to `0`.
  - `0 & 1` is equal to `0`.
  - `1 & 0` is equal to `0`.
  - `1 & 1` is equal to `1`.
- `|` is bitwise OR (not to be confused with `||`, logical OR), which gives `1` if at least one of its arguments is `1`.
  - `0 | 0` is equal to `0`.
  - `0 | 1` is equal to `1`.
  - `1 | 0` is equal to `1`.
  - `1 | 1` is equal to `1`.
- `^` is bitwise XOR, or **exclusive or**, which gives `1` if exactly one of its arguments is `1`.
  - `0 ^ 0` is equal to `0`.
  - `0 ^ 1` is equal to `1`.
  - `1 ^ 0` is equal to `1`.
  - `1 ^ 1` is equal to `0`.
- `~` is bitwise NOT, which is a **unary operator** - unlike the above, it operates on only one argument, rather than a pair. It flips the given bit:
  - `~0` is equal to `1`.
  - `~1` is equal to `0`.
- `<<` is the **left shift** operator.
  - `1 << 7` is equal to `10000000` - the `1` is shifted to the left 7 places, and the places to the right are padded with \`0\`s.
  - We'll see some more clever uses of this operator as well!