Table of Contents



---

# Week 3

## Last Time

- We looked at debugging, with tools like: `help50`, to help us understand error messages when we compile a program; `eprintf`, to print messages that might be helpful; `debug50`, which will let us go through our program line by line, and see variables as they are changed.
- We also looked at cryptography and how we might implement simple ciphers.
- We learned what strings actually were, arrays of characters.
- We talked about command-line arguments, using `argc` and `argv` from the command line for our `main` function to use.

---

## Sorting

- Now that we know enough of the basics, we can start problem-solving with algorithms at a higher level.
- An array is a data structure, a way to store data in memory.
- Imagine having 7 doors, and having to find a certain number that is behind one of those doors.
- Without any information, we might open doors at random until we find the number we want.
- But if we knew the numbers behind the doors were sorted, we could look more efficiently.
- Just like in Week 0 where we got to throw half the problem away at each step, we can search in a sorted array with binary search, opening one door, looking at the value to see if it's greater than or less than the value we want, and moving to the half that might contain our number.

- Another search algorithm is linear search, moving left to right in a line. We could write pseudocode (words that look like code but is easier to understand) that looks like this:

```
for each element in array
    if element you're looking for
        return true
return false
```

  - Notice that we have `return false` outside of the `for` loop, so at the end of the loop, if we've looked at all the elements in the array but not yet found what we've been looking for, we can say we haven't found it.
    - We also indent our code to indicate the nesting of statements. `return false`, in particular, will execute after the entire `for` loop has completed, rather than after just looking at the first element, if it were indented.
- Binary search, from week 0, can be written like this in pseudocode:

```
look at middle of array
if element you're looking for
    return true
else if element is to left
    search left half of array
else if element is to right
    search right half of array
else
    return false
```

  - Now we don't have line numbers to go to, but notice `search left half of array` and `search right half array` can just refer to this entire algorithm again. We use the same method to search, but the input has gotten smaller at each step, so we'll eventually arrive at an answer. A function that calls itself again and again is called a recursive function, and this principle is important in searching and sorting.
- So how do we get to a sorted array? We have a volunteer, Allison, sort some shuffled blue books. Each time, she uses her left hand to take a book from the shuffled pile, and puts it into the correct spot in a sorted pile that she flips through with her right hand.
- If we had a deck of cards, we might first start by putting the cards into buckets by numbers or suit. And once we have all the piles, we might need to go through and sort the piles themselves.
- We can generalize this problem by representing some items with numbers:

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

- One approach is to find the smallest element, `1`, and move it to the front of the list and shift the other ones down:

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

- We can continue:

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

- But in our first step, it took a lot of work to shift the numbers to the left of `1` down. So we can just swap `1` with the first number:

```
4 2 6 8 1 3 7 5
1 2 6 8 4 3 7 5
1 2 3 8 4 6 7 5
...
```

  - Now we're doing a little less work at each step.
  - But to select the smallest element at each round, we need to look at the unsorted part of the list and look for the smallest one, to know which one to swap.
  - This algorithm is called **selection sort**, where we select the smallest element each time.

- Let's do a different version called **bubble sort**:

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

  - We move down the list from left to right and compare each pair of numbers. If they are out of order, then we swap them. The list isn't sorted yet but the highest number is now on the right, and the other numbers are slightly closer to where they should be.

- We can repeat this process over and over, until we have our sorted list. We'll know to stop if we make it through the entire list and not make any swaps.

- We'll look at one final approach, **insertion sort**. We'll take our elements one at a time and build a sorted list as we go along:

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

- Each time, we look at the next element and place it into our sorted portion of the list, even if we have to shift elements.

---

# Running Time

- For bubble sort, our algorithm might look like this:

```
repeat until no swaps
    for i from 0 to n-2
        if i'th and i+1'th elements out of order
            swap them
```

  - Recall that the element at the end of the list is the `n - 1`th since we start counting at `0`. So looking at pairs of elements, the last pair would stop at `n - 2`.

- For selection sort:

```
for i from 0 to n-1
    find smallest element between i'th and n-1'th
    swap smallest with i'th element
```

  - Now we're building a sorted list by going through the unsorted part of the list, finding the smallest element, and placing it at the end of our sorted list.

- Insertion sort:

```
for i from 1 to n-1
    call 0'th through i-1'th elements the "sorted side"
    remove i'th element
    insert it into the sorted side in order
```

  - Here we are building a sorted list by taking each element in the list, and inserting it into the correct spot of the sorted list so far.

- We can use the number of steps as a unit for measuring how efficient an algorithm is, but any unit is fine as long as we're consistent.
- For bubble sort, if we have a list with $n$ elements, we would compare $(n - 1)$ pairs in our first pass.
- And after our first pass, the largest element will have been swapped all the way to the right. So in our second pass, we'll only need $(n - 2)$ comparisons.
- So we'll have made a total of $(n - 1) + (n - 2) + \ldots + 1$ comparisons. And this one actually adds up to $n(n - 1)/2$. And that multiplies out to $(n^2 - n)/2$.
- When comparing running time, we generally just want the term with the biggest order of magnitude, since that's the only one that really matters when $n$ gets really big. And we can even get rid of the factor of 1/2.
- We can look at an example (not a proof!) to help us understand this. Imagine we had 1,000,000 numbers to sort. Then bubble sort will take $1{,}000{,}000^2/2 - 1{,}000{,}000/2$ steps, and if we multiply that out, we get 500,000,000,000 - 500,000 = 499,999,500,000. Which is awfully close to the first number.
- So when we have an expression like $(n^2 - n)/2$, we can say it is on the order of, $O(n^2)$.
- There is a more formal mathematical definition, but we'll consider it to be an upper bound on how long an algorithm might take.
- Depending on the algorithm, we might see:
  - $O(n^2)$
  - $O(n \log n)$
  - $O(n)$
  - $O(\log n)$
  - $O(1)$
    - This last one takes one step, or ten steps, or a constant number of steps regardless of the size of the problem.
- It turns out, if we wrote out the steps, bubble sort, insertion sort, and selection sort all have running time of $O(n^2)$. Even though they're all slightly different, they all involved some variation of looking through up to $n$ elements, up to $n$ times. With insertion sort, we're looking at each element just once, but as we sort we might need to shift all the elements in the list we've already sorted, which requires work.
- Finding an element in an unsorted list, with linear search, for example, would have running time of $O(n)$, since we might look at up to all $n$ elements before we find the correct one.
- Binary search would have a logarithmic running time, $O(\log n)$, since we are dividing the problem in half each time.

- And constant time algorithms, with running time $O(1)$, might include adding numbers or printing something, since in each case we can say it takes one step.
- Another symbol we might see is big Omega, **Ω**, which we can think of as the opposite of big O. Big O is the running time of the worst-case scenario (in the case of sorting, for many algorithms the worst-case scenario is a list that is completely backwards), but big Omega is the lower bound, or the best case.
- An algorithm with $\Omega(n^2)$, for example, would be selection sort. Even if the list was already sorted, we wouldn't know because we look for the smallest element in the rest of the list, one at a time, so we end up looking at about $n^2$ elements.
- Bubble sort, on the other hand, $\Omega(n)$, since we stop if we made no swaps, and after looking at all $n$ elements, we can realize that a sorted list was indeed sorted, and stop.
- But we realize that it is impossible to sort a list with $n$ elements in $\Omega(\log n)$ or $\Omega(1)$, since at the least we need to look at all $n$ elements to make sure it is sorted.
- Algorithms for search, like linear search or binary search, tend to have $\Omega(1)$ running time, since in the best case we get lucky and find our element on the first try.
- And we have yet another notation, theta, Θ, if the running time of an algorithm is the same in the worst-case (Ω) and the best-case (O).
- We take a look at this visualization (https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html) of how sorting differs between algorithms, and also this visualization (http://cglab.ca/~morin/misc/sortalg/).

---

# Sorting

- Recall that our pseudocode for finding Mike Smith in the phone book was this:

```
 0   pick up phone book
 1   open to middle of phone book
 2   look at names
 3   if Smith is among names
 4       call Mike
 5   else if Smith is earlier in book
 6       open to middle of left half of book
 7       go back to step 2
 8   else if "Smith" is later in book
 9       open to middle of right half of book
10       go back to step 2
11   else
12       quit
```

- In that version, we used `go back` to create loops in our algorithm. But we could more simply do something like this:

```
 0   pick up phone book
 1   open to middle of phone book
 2   look at names
 3   if Smith is among names
 4       call Mike
 5   else if Smith is earlier in book
 6       search for Mike in left half of book
 7
 8   else if "Smith" is later in book
 9       search for Mike in right half of book
10
11   else
12       quit
```

- Now our program is recursive, where it calls itself.
- We can write pseudocode for merge sort too:

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

- If we have fewer than 2 elements, then our list has to be sorted so we should stop.

- Now we rely on the same function to sort the halves for us, and once it sorts the halves we'll merge them together.

- We can best see this with an example:

```
4 8 6 2 1 7 5 3          // unsorted list
```

```
| 4 8 6 2 | 1 7 5 3     // sort the left half
```

```
| 4 8 | 6 2 1 7 5 3     // sort the left half of the left half
```

```
| 4 | 8 6 2 1 7 5 3     // sort the left half of the left half of the left
```

```
4 | 8 | 6 2 1 7 5 3     // sort the right half of the left half of the lef
```

```
| _ _ | 6 2 1 7 5 3     // now we merge the left half of the left half
| 4 8 |                 // use extra memory to keep our sorted list of siz
```

```
_ _ | 6 2 | 1 7 5 3     // now we go back and sort the right half of the l
4 8 | 2 6 |             // sorted right half of right half
```

- Now we can remember that our second statement earlier, "sort the left half", is wrapping up with merging its two sorted halves together:

```
_ _ | 6 2 | 1 7 5 3
4 8 | 2 6 |
2 4   6 8 |             // merged left half
```

  - To merge two sorted lists, we start at the beginning of both lists, and take whichever element is the smallest at each step.

- Now we repeat with the right half:

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

```
_ _ | _ _ | 1 7 5 3
_ _ | _ _ | 1 7 |       // sorted left half of right half
2 4   6 8 |
```

```
_ _ | _ _ | 1 7 5 3
_ _ | _ _ | 1 7 | 3 5 |  // sorted right half of right half
2 4   6 8 |
```

```
_ _ | _ _ | 1 7 5 3
_ _ | _ _ | 1 7 | 3 5 |  // sorted right half of right half
2 4   6 8 |
```

```
_ _ | _ _ | 1 7 5 3
_ _ | _ _ | _ _ | _ _ |
2 4   6 8 | 1 3   5 7    // merged right half
```

- Now we're back to the very first pass of our algorithm where we need to merge both halves, so:

```
_ _ | _ _ | 1 7 5 3
_ _ | _ _ | _ _ | _ _ |
2 4   6 8 | 1 3   5 7
1 2   3 4   5 6   7 8    // merged list
```

- It seems that there were a lot of steps, and on top of that we needed a lot of extra space to keep the new lists stored somewhere in memory.
- But we could have used the space in the original list as we went along, so we could get by with memory for just two lists.
- And with a list of 8 elements, we only needed to have 3 layers, splitting it three times.
- So with dividing the problem in half each time, it seems that we've reduced our problem to something logarithmic. And at each layer, we looked at all $n$ elements to merge them. So intuitively, we can guess that this algorithm takes $O(n \log n)$ time.
- We can even look at the pseudocode to analyze running time:

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

- The first condition takes O(1) step to return, a constant number, so T($n$) = O(1). The running time is O(1).
- But the second condition takes T($n$) = T($n$/2) + T($n$/2) + O($n$) since sorting each half requires the running time of each half, plus the time it takes to merge the two halves.
- Mathematically, this recurrence also comes out to O($n$ log $n$). But this would only be obvious if you're familiar with this subject and had the help of a textbook; no worries if not!
- Let's take a look at how this might be applied.
- `sigma0.c`:

```
#include <cs50.h>
#include <stdio.h>

int sigma(int m);

int main(void)
{
    int n;
    do
    {
        printf("Positive integer please: ");
        n = get_int();
    }
    while (n < 1);

    int answer = sigma(n);

    printf("%i\n", answer);
}

int sigma(int m)
{
    int sum = 0;
    for (int i = 1; i <= m; i++)
    {
        sum += i;
    }
    return sum;
}
```

- This program takes some integer `m` as input, and adds up all the numbers from `1` to `m`, inclusive, with a loop.

- A recursive version, `sigma1.c`, would look like this:

```
#include <cs50.h>
#include <stdio.h>

int sigma(int m);

int main(void)
{
    int n;
    do
    {
        printf("Positive integer please: ");
        n = get_int();
    }
    while (n < 1);

    int answer = sigma(n);

    printf("%i\n", answer);
}

int sigma(int m)
{
    if (m <= 0)
    {
        return 0;
    }
    else
    {
        return (m + sigma(m - 1));
    }
}
```

- The function `sigma` now calls itself, where it adds the current `m` to whatever the sum from `0` (because the function just returns `0` if `m` is `0` or less) to `m - 1`, and that gives us the sum from `0` to `m`.
- But having recursion, while it looks elegant, might not be the best decision if we have a large `m` and need to call the function over and over again, using up more memory that we would otherwise.

- Soon we'll take a look at fancier data structures, and how we might apply some of these concepts to working with them.