CS50 Section 3
Somewhere in Between

Annaleah Ernst, TF

The Agenda…

- Quick recap of last week (and expectations)
- Asymptotic Notation (O, Ω)
- Searches
  - Binary Search
- Sorts
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Merge Sort
- Recursion

Recap & Rules

Last week…

- Debugging
  - `help50`
  - `eprintf`
  - `debug50`
  - Duck Debugging
- Arrays
- Functions
  - Scope
- Command Line Arguments
- ASCII

Style and Design

- Comments
- Descriptive Variable names
- No magic numbers!!!
  - #define <NAME> <value>

What’s going on here?

```
if (p[i] >= 65 && p[i] <= 90)
{
    p[i] = ((p[i] - 65 + k) % 26) + 65;
}
else if (p[i] >= 97 && p[i] <= 122)
{
    p[i] = ((p[i] - 97 + k) % 26) + 97;
}
```

- No comments
- Undescriptive variable names
- Magic numbers
- Why do we care?
  - Harder to debug
  - Harder to understand
  - Harder to update

A better implementation:

```
// only encrypt letters, not other chars (e.g., digits)
if (plaintext[i] >= 'A' && plaintext[i] <= 'Z')
{
    plaintext[i] = ((plaintext[i] - 'A' + key) % NUM_LETTERS) + 'A';
}
else if (plaintext[i] >= 'a' && plaintext[i] <= 'z')
{
    plaintext[i] = ((plaintext[i] - 'a' + key) % NUM_LETTERS) + 'a';
}
```

- Comment at top describing the code
- Descriptive variable names
- No magic numbers
- It’s very clear what is going on here

When do we worry (what do we do)

- Is there a magic number here?

```c
for (int i = 0, n = strlen(text); i < n; i++)
```

  - No – starting a counter at 0 makes intuitive sense
- What about here?

```c
for (int i = 5, n = strlen(text); i < n; i++)
```

  - Yes – what does this 5 signify? I have no idea.
- Two solutions:
  - Use variables if the value will change
  - Use #define <NAME> <value> for constants
    - This goes at the top of the file right after you #includes, eg
    - `#define LEN_ALPHA 26`

Asymptotic notation – "Big O"

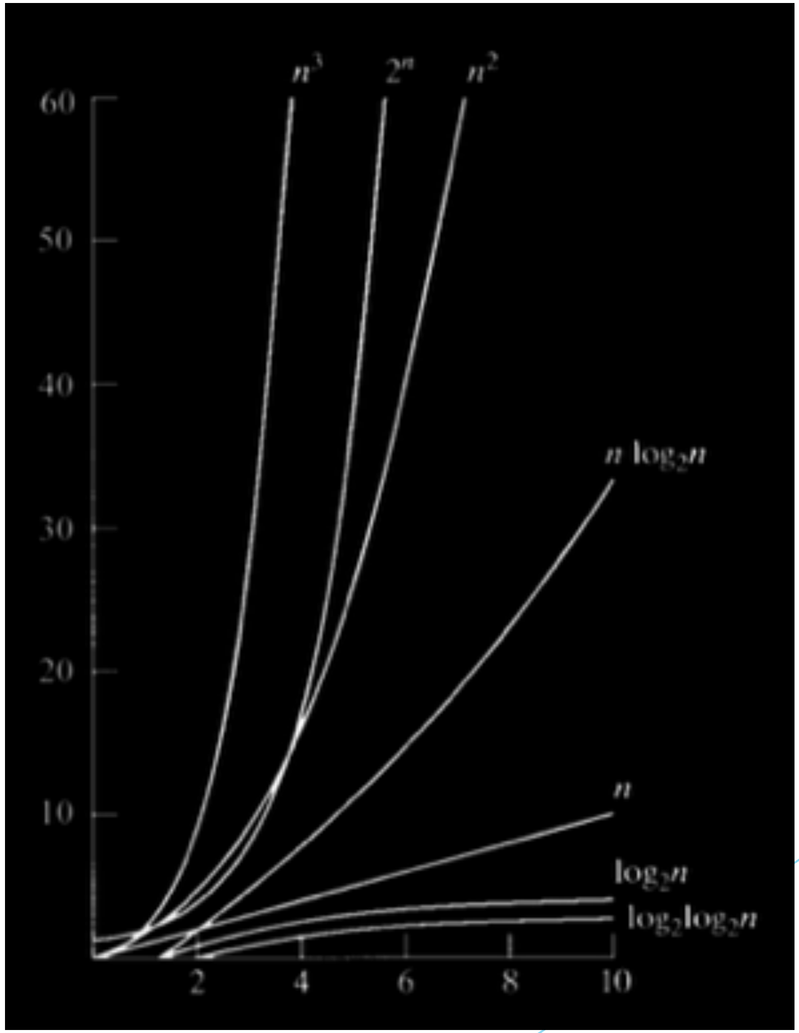
- Describes upper bound on program's runtime
  - We usually use it to describe running time with worst case inputs
- The method:
  - Suppress low order terms and constant factors
    - This is because the limit as n (ie, number of elements to work with) gets very large, the only term that means anything will be the largest order
- Some examples to think about:
  - What's the worst case run time for finding an element in a list of length n?
  - What's the worst case for (naively) sorting a list?
  - What would their representative Big O be?
    - O(n), O(n(^2)

Asymptotic Notation - Ω

- Ω describes the lower bound
  - We usually use it to describe best cases
- Some examples to think about:
  - What’s the best case run time for finding an element in a list of length n?
  - What’s the best case for sorting a list (and making sure it’s sorted)?
  - What would their representative Ω be?
    - $\Omega(1)$, $\Omega(n)$

More on Asymptotics

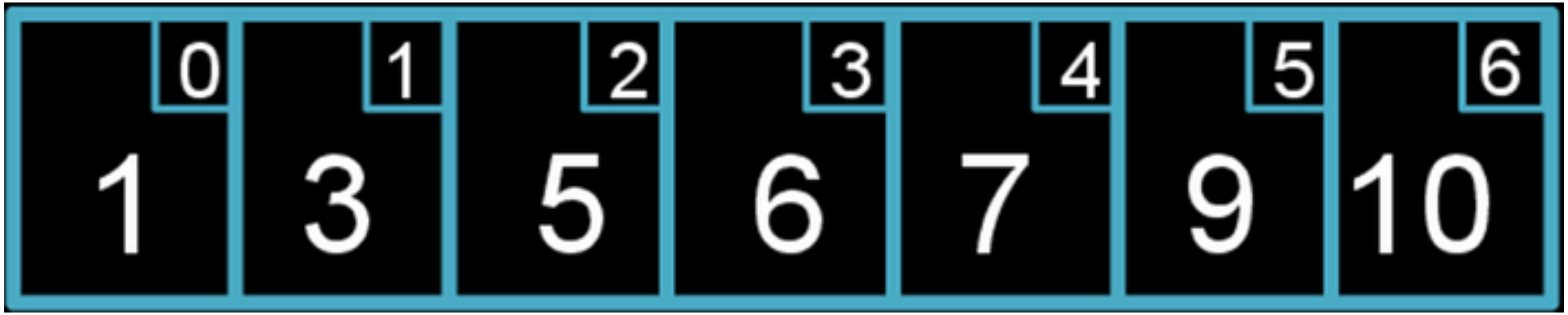
- Asymptotic runtime gets more and more important as n goes to infinity
- Rate of growth gets really significant



Searching

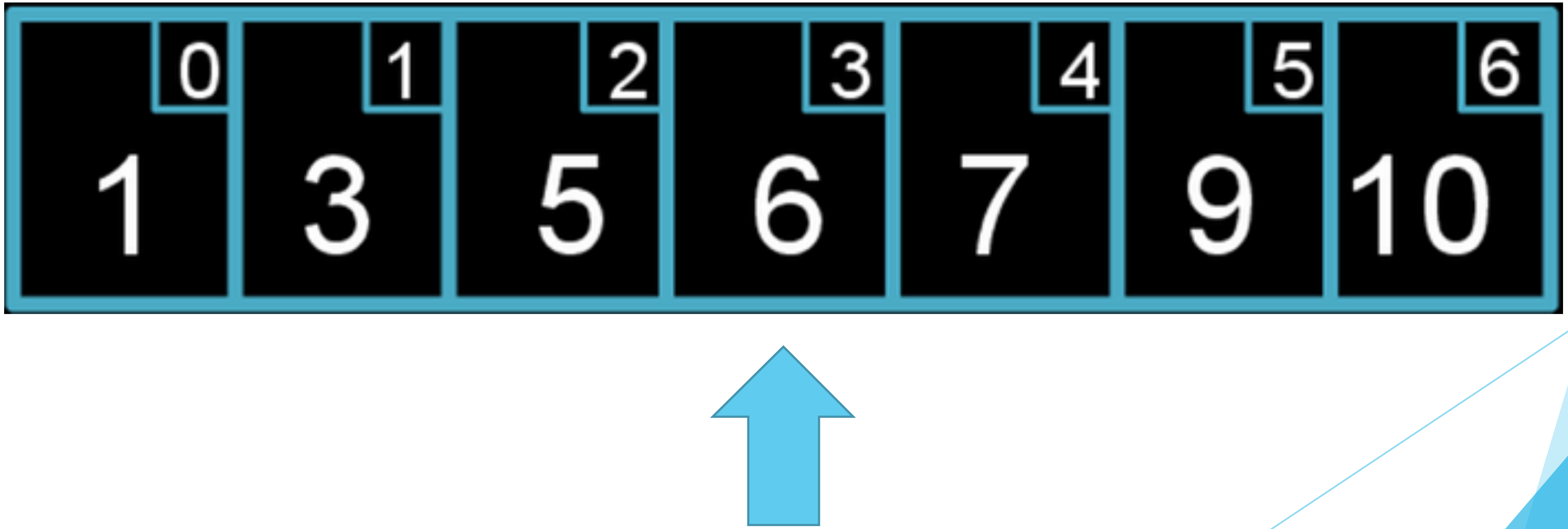
- Linear search – just look at each element
  - Advantages: doesn’t require array to be sorted
  - Disadvantage: it’s suuuuper slow
- Binary Search
  - Works for sorted arrays
  - Check middle of the array
  - If it’s equal, we’re done
  - If middle is higher, repeat process on lower half of the list
  - If middle is lower, repeat on upper half of the list

Is 7 in the array?



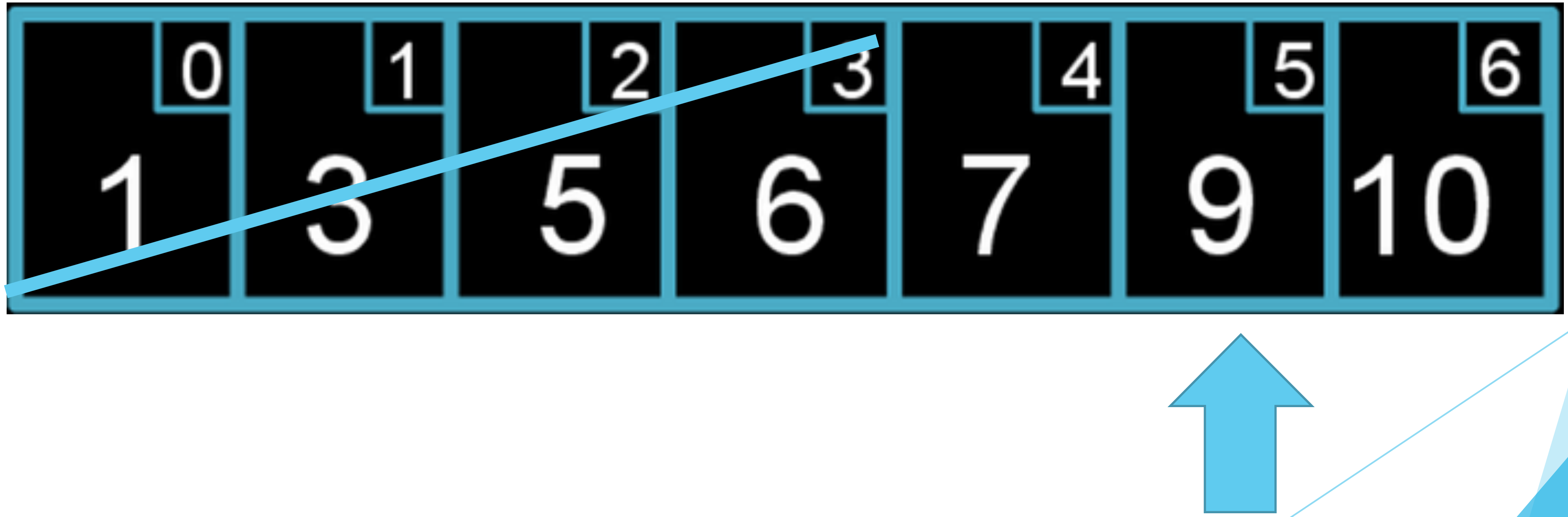
Finding 7

- Is array[3] == 7?
- Is array[3] < 7?
- Is array[3] > 7?



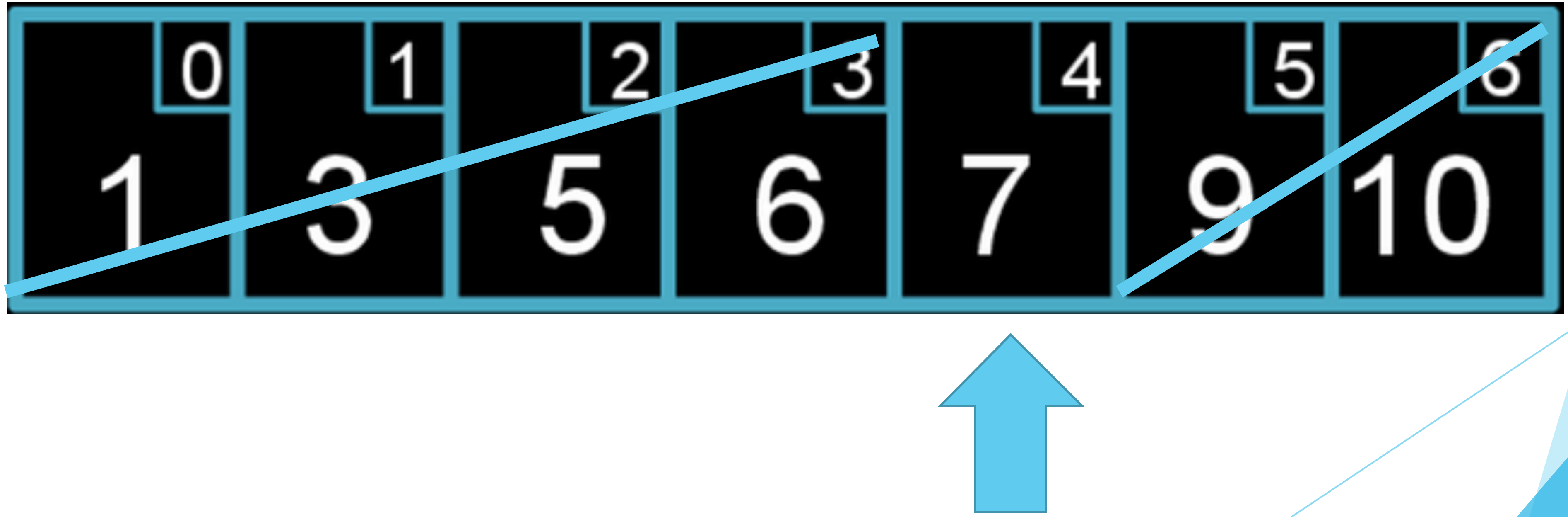
Finding 7

- Is array[5] == 7?
- Is array[5] < 7?
- Is array[5] > 7?



Finding 7

- Is array[4] == 7?
- Is array[4] < 7?
- Is array[4] > 7?



Your Turn - Binary Search

- Implement the iterative version of binary search using the following declaration:
- ```
  bool binary_search(int value, int values[], int n);
  ```
  - `value`: the number we're searching for
  - `values`: the (sorted) integer array we are searching though
  - `n`: the length of `values`
- No need to write main, just the loop of binary search
  ```

Your Turn – Binary Search

- Fill in the contents of the while loop for binary search

```
bool binary_search(int value, int values[], int n)
{
    // Set values for the top and the bottom of the search
    int lower = 0;
    int upper = n - 1;

    // Binary search
    while (lower <= upper)
    {
        // your code here
    }

    return false;
}
```

Binary Search

- What is the asymptotic runtime of binary search? (worst case/best case)
  - $O(\log(n))$
  - $\Omega(1)$

Sorts

- Bubble
- Selection
- Insertion
- Merge

Bubble sort

- Works on array of size n by…
  - Iterating over unsorted part of the array
  - Swapping adjacent items that are out of place
    - Large elements “bubble” to the top
- Move higher elements generally to the right and lower elements generally to the left
- Psuedocode
  - Set swap counter to a non-zero value
  - Repeat swap counter is 0:
    - Reset swap counter to 0
    - Look at each adjacent pair
      - If two adjacent elements are not in order, swap them

Bubble Sort

- See pdf for example

Bubble Sort

- Worst Case: array in reverse order
  - bubble each of the n elements all the way across the array
  - only one gets to destination per pass, so we must do this n times
  - What's the O runtime for Bubble sort?
    - $O(n^2)$
- Best case: already sorted
  - We make no swaps on the first pass
  - What's the Ω runtime for Bubble sort?
    - $\Omega(n)$

Your Turn!

- Implement the inner loop of bubble sort
- (Swap adjacent elements if out of order)

```
void bubble_sort(int array[], int n)
{
    // cycle through array
    for(int k = 0, outer_max = n - 1; k < outer_max; k++)
    {
        // optimize; check if there are no swaps
        int swaps = 0;

        // swap adjacent elements if out of order
        for(int i = 0, inner_max = outer_max - 1; i < inner_max; i++)
        {
            // you code here
        }

        if (!swaps)
            break;
    }
    printIntArray(array, n);
}
```

Selection Sort

- Works on array of size n by…
  - Removing the smallest element in unsorted part of array
  - Placing at the head
- Psuedocode:
  - Repeat until no unsorted elements remain
    - Search unsorted part of the data to find the smallest value
    - Swap the smallest found value with the first element of the unsorted part

Selection Sort

- See pdf for example

Selection Sort

- Worst Case
  - iterat over each of the n elements in the array to find the smallest unsorted elt
  - only one gets to destination per pass, so we must do this n times
  - What's the O runtime for Selection sort?
    - O(n^2)
- Best case
  - Exactly the same!
  - What's the Ω runtime for Bubble sort?
    - Ω(n^2)
- Since best case and worst case are the same…
  - **Θ(n^2)**

Your turn!

- Follow the pseudocode to make selection sort

```
void selection_sort(int array[], int size)
{
    // iterate over array
    for(int i = 0; i < size - 1; i++)
    {
        // smallest element and its position in sorted array

        // unsorted part of array
        for(int k = i + 1; k < size; k++)
        {
            // find the next smallest element

        }

        // swap current smallest element with element in current index

    }
    printIntArray(array, size);
}
```

Insertion Sort

- Build your sorted array in place shifting elements as necessary
- Pseudocode
  - Call the first element of the array sorted
  - Repeat until all elements are sorted:
    - Look at the next unsorted element
    - Insert into sorted portion by shifting requisite elements

Insertion Sort

- See pdf for example

Insertion Sort

- Worst Case: array in reverse order
  - we have to shift n elements n positions each time we insert
  - What's the O runtime for Insertion sort?
    - $O(n^2)$
- Best case: already sorted
  - Simply keep moving the line between sorted and unsorted as we examine each elt
  - What's the Ω runtime for Bubble sort?
    - $\Omega(n)$

Your turn

- Explain to your neighbor the difference between the three sorts. What are possible tradeoffs?

Recursion

- Divide and conquer!
- A function that calls itself from within itself
- Often used to “elegantly” solve a problem – it’s very pretty
- Two cases for recursive functions:
  - Base case – when triggered, this ends the recursive calls to the function and start returning
  - Recursive case – we call the function again on a smaller subset of the problem
- EX: factorial

Merge Sort

- Easiest to implement with recursion!
- Runs in O(nlogn)
- Power comes from divide and conquer approach
- Only has to do n comparisons log(n) times!
  - The size of the arrays being compared is halved every recursive call

Merge Sort

- See pdf for example

Asymptotic Runtimes of Sorts

	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort
O	n^2	n^2	n^2	$n \log n$
Ω	n	n^2	n	$n \log n$
Θ		n^2		$n \log n$

GDB (the GNU Debugger)

- Works on executable files, allows you to step through your code
- Common commands:
  - `gdb ./<program name>`
  - `break <function name>`, `break <line number>`
    - Set breakpoint at beginning of program: `break main`
  - `run <command line args>`
- Common commands, cont
  - `next, n`
  - `step, s`
  - `list`
  - `print, p`
  - `info locals`
  - `continue, c`
  - `disable`
  - `quit, q`