CSE 3318

Week of 06/05/2023

Instructor : Donna French

Code Formatting

Formatting will count as 10% of the grade for any code you write in this class – Coding Assignments or OLQs.

- Indention and alignment
  - Code blocks should be indented at least 3 spaces and not more than 5 spaces
    - If tabs are used, always use tabs and set tab size to be 3-5 spaces
    - If spaces are used, always use spaces and always use the same number of them
    - Curly braces { } should align vertically and be on their own line

```
A
{
   B;
   C
   {
      D;
   }
}
```

Code Formatting

Code formatting has several benefits

- allows quick readability – it is easier/faster to understand the gross structure of the code without in depth examination
- allows for less reliance on the editor to match up braces and code blocks
- creates readable code that is easier for someone other than the student to read – for example, when the student is asking the instructor or TAs for assistance
- allows for easier grading of code – both the instructor and student benefit – code that is easier to grade is less likely to be marked as incorrect
- gives the students the experience of apply a given formatting standard which they will likely encounter as a professional programmer

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

new 1

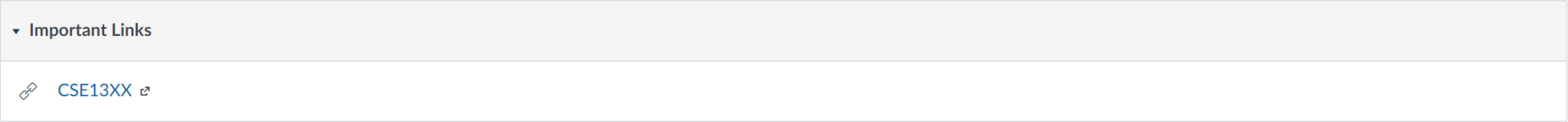
```c
#include <stdio.h>int main(void){
int X = 1;int Y = 1;int Z = 0;int a = 1;
if (X && Z){a /= 3;}else
if (X && Y){a *= 4;}else
{a -= 4;}printf("%d", a);return 0;}
```

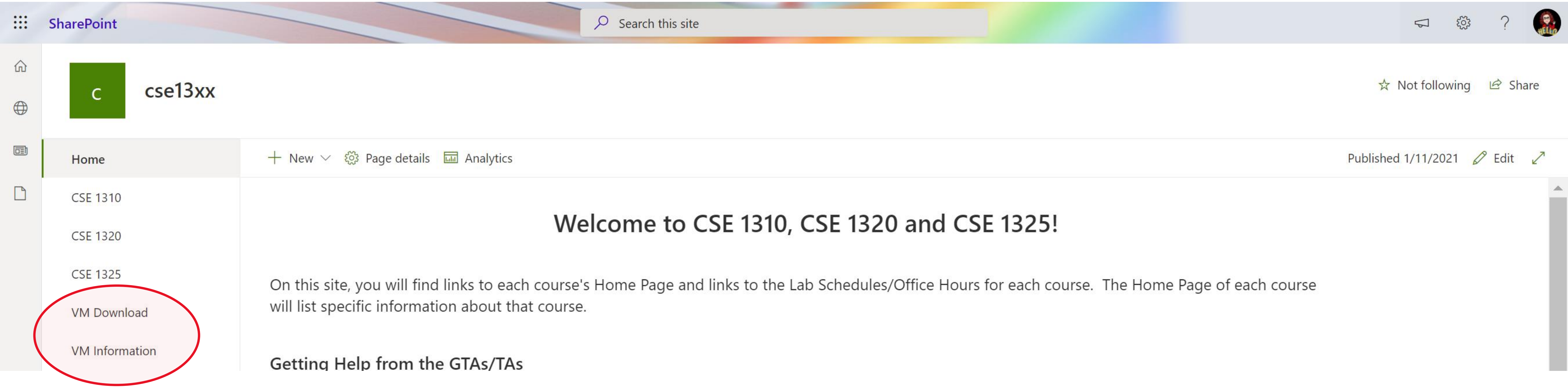
Tools Needed for this Class

- Text editor or IDE that recognizes the C language (syntax highlighting)
- File Transfer program (FileZilla)
- Terminal emulator (PuTTY or SSH)
- Oracle VM or Visual Studio Code with an Ubuntu Terminal

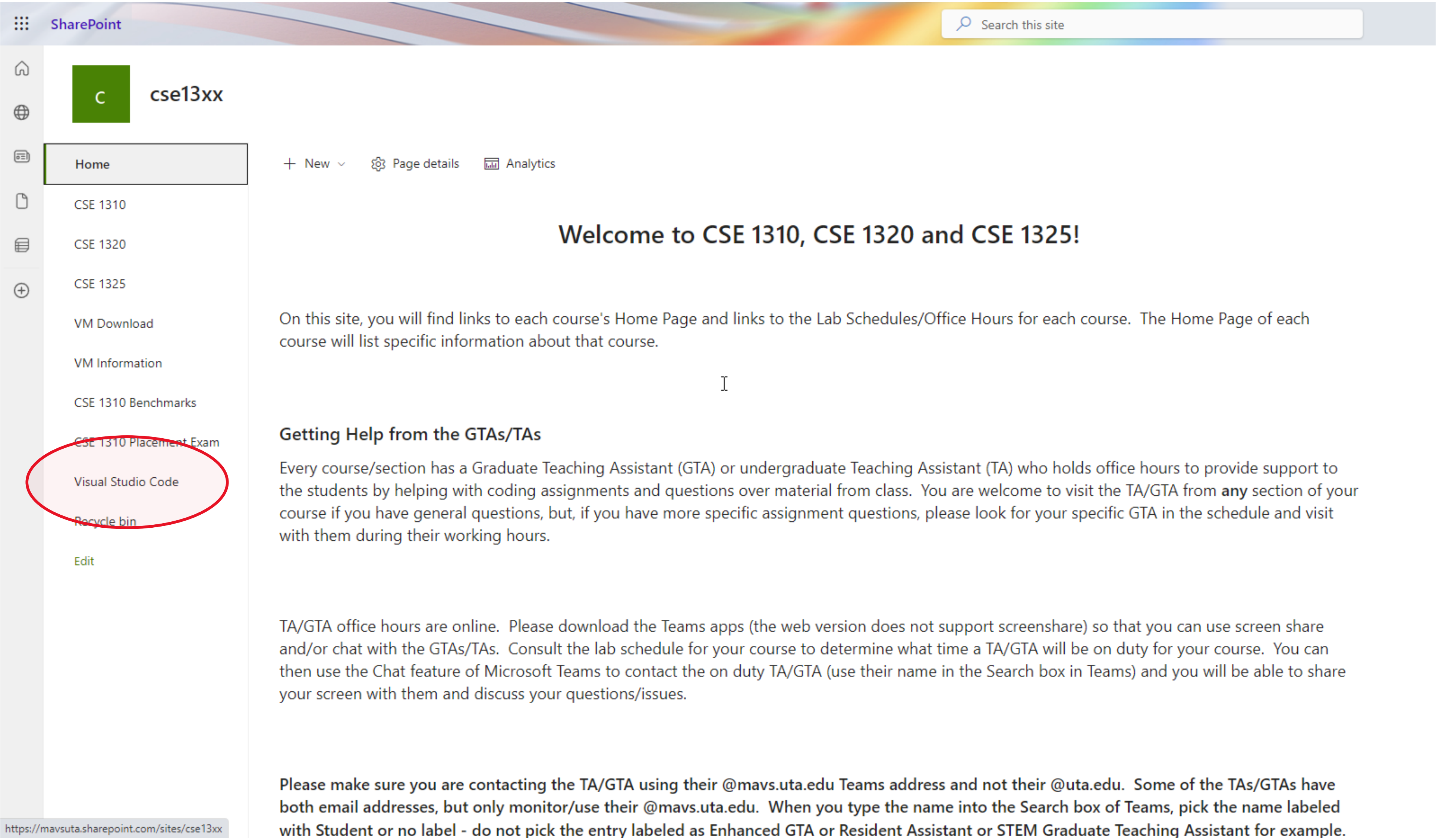
We will be using **BOTH** Omega and PCs to compile our C programs.

Installing the VM





Installing Visual Studio Code



The Role of Algorithms in Computing

Informal definition

An algorithm is any well-defined computational procedure that takes some value, or set of values, as **input** and produces some value, or set of values, as **output**.

An algorithm is a sequence of computational steps that transform the **input** to the **output**.

The Role of Algorithms in Computing

An algorithm is also a tool for solving a well-specified computational problem.

The statement of the problem specifies in general terms the desired input/output relationship.

The algorithm describes a specific computational procedure for achieving that input/output relationship.

The Role of Algorithms in Computing

Informal definition of problem :

Sort a sequence of numbers into nondecreasing order.

Formal definition of problem :

Input : A sequence of *n* numbers $\langle a_1, a_2,\ldots,an\rangle$.

Output : A permutation (reordering) $\langle a'1, a'2,\ldots a'_n\rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

The Role of Algorithms in Computing

Input : A sequence of *n* numbers $\langle a_1, a_2, \ldots, an \rangle$.

Output : A permutation (reordering) $\langle a'1, a'2, \ldots a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Given the input sequence of

$$\langle 31, 41, 59, 26, 41, 58 \rangle$$

our sorting algorithm would return the output as

$$\langle 26, 31, 41, 41, 58, 59 \rangle$$

The Role of Algorithms in Computing

This input sequence

$$\langle 41, 31, 58, 26, 41, 59 \rangle$$

or this input sequence

$$\langle 31, 41, 59, 26, 41, 58 \rangle$$

or this input sequence

$$\langle 58, 59, 41, 41, 26, 31 \rangle$$

would all result in the same output

$$\langle 26, 31, 41, 41, 58, 59 \rangle$$

Each of these are an ***instance*** of the sorting problem.

An ***instance of a problem*** consists of the input needed to compute a solution to the problem.

The Role of Algorithms in Computing

Sorting is a fundamental operation in computer science.

Which sorting algorithm is best for a given application depends on many factors…

- the number of items to be sorted
- the extent to which the items are already somewhat sorted
- possible restrictions on the item values
- the architecture of the computer
- type of storage device (memory, disk, tape…)

The Role of Algorithms in Computing

So what makes an algorithm "correct"?

An algorithm is called **correct** if, for every input instance, it halts with the correct output.

A correct algorithm solves the given computational problem.

An incorrect algorithm might not halt at all on some input instances or it might halt with an incorrect answer.

The Role of Algorithms in Computing

What kinds of problems are solved by algorithms?

Lots and lots and lots of them!!!

- Sorting
- Internet – managing and manipulating large volumes of data
- Internet – routing data
- Internet – searching
- Cryptography and digital signatures
- Allocation of resources
- GPS routing

The Role of Algorithms in Computing

What kinds of problems are solved by algorithms?

Lots and lots and lots of them!!!

- Given a person's song history and a large song database, find a song they might like hearing next.
- Analyze thousands of casual pictures taken by hundreds of different cameras to track an individual
- Given a grocery list and the layout of a store, find the quickest way to collect all items.

The Role of Algorithms in Computing

Two characteristics common to many algorithmic problems

1. Have many candidate solutions
   a. most of which do not solve the problem
   b. finding a solution that does or is the best can be very challenging

2. Have practical applications
   a. You want your GPS to not only find a route but find the best route – which may not be the shortest because of other factors

The Role of Algorithms in Computing

We are going to use various data structures to store the data our algorithms will be manipulating.

A data structure is a way to store and organize data in order to facilitate access and modifications.

No single data structure works well for all purposes.

We will learn the strengths and limitations of several.

The Role of Algorithms in Computing

Other than speed, what other measures of efficiency might one use in a real-world setting?

Resources

Memory space

For example, using recursion may be the most efficient approach timewise but does the machine running the algorithm have the resources/memory space to execute the recursion? Remember all of those function execution environments?

The Role of Algorithms in Computing

Select a data structure that you have seen previously and discuss its strengths and limitations.

Arrays vs Linked Lists

Do you want a car that goes fast or a car that has an expandable number of seats?

Arrays require contiguous memory but allow for random access.

Linked lists do not require contiguous memory but are limited to sequential access.

An array's size is set when it is created – a linked list can grow.

The Role of Algorithms in Computing

Come up with a real-world problem in which only the best solution will do. Then come up with one in which a solution that is "approximately" the best is good enough.

Launching a probe to Mars

vs

Driving to Grandma's house for Sunday dinner

The Role of Algorithms in Computing

Why study algorithms?

Computers are fast but not infinitely fast.

Memory may not be expensive, but it is not free.

Computing time is a bounded resource and so is space in memory.

We need to use these resources efficiently.

The Role of Algorithms in Computing

Efficiency

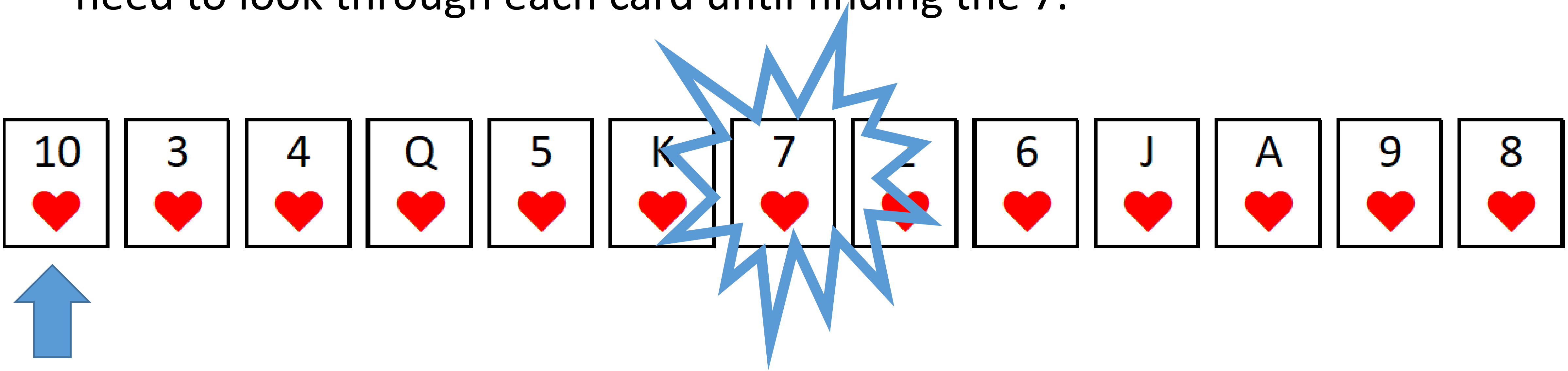
Different algorithms devised to solve the same problem often differ dramatically in their efficiency.

These differences can be much more significant than differences due to hardware and software.

Linear Search Algorithm

A **linear search** or **sequential search** is a method for finding an element within a list. It sequentially checks each element of the list until a match is found or the whole list has been searched.

For example, if you have a set of cards and need to find the 7, you will need to look through each card until finding the 7.

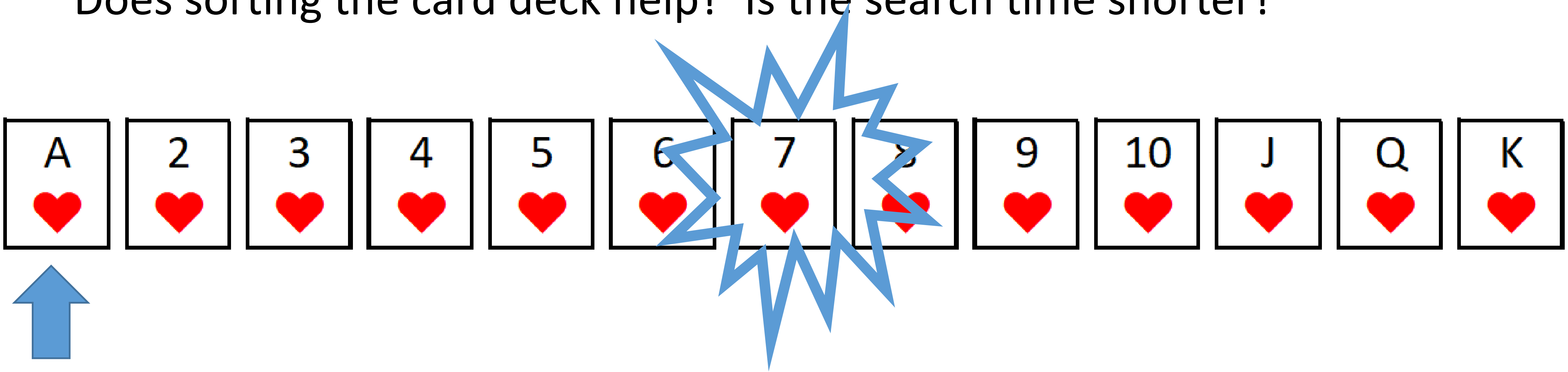


Linear Search Algorithm

The card you are looking for might be the first card or it might be the last card or it could be anywhere in between.

You would have to compare each card one by one.

Does sorting the card deck help? Is the search time shorter?



Linear Search Algorithm

Best case scenario – the card we are searching for is the first card.

Worst case scenario – the card we are searching for is the last card.

Sorting the inform .

What happens wh t?

When you look up irst word on the first page and keep checking eve

The **Oxford English Dictionary** has a new "**last word**" – "zyzzyva", which is a genus of tropical weevil native to South America typically found in palm trees.

Hope you are not looking for "zyzzya".

C:\Users\frenc\Desktop\VM\CSE3318\LinearSearch.c - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

LinearSearch.c

```c
/* C program for Linear Search */
#include <stdlib.h>
#include <stdio.h>

/* Function to print an array */
void printArray(int A[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n\n");
}

int main(int argc, char *argv[])
{
    int arr[] = {12, 11, 5, 13, 7, 6};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    int i = 0;
    int FoundIt = 0;
    int SearchValue = atoi(argv[1]);
```

C source file | length : 709 lines : 36 | Ln : 29 Col : 33 Pos : 564 | Windows (CR LF) | UTF-8 | INS

```c
/* C program for Linear Search */
#include <stdlib.h>
#include <stdio.h>

/* Function to print an array */
void printArray(int A[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n\n");
}

int main(int argc, char *argv[])
{
    int arr[] = {12, 11, 5, 13, 7, 6};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    int i = 0;
    int SearchValue = atoi(argv[1]);

    printArray(arr, arr_size);

    while (i < arr_size && arr[i] != SearchValue)
    {
        i++;
    }

    if (arr[i++] == SearchValue)
        printf("Found %d in array after %d checks\n", SearchValue, i);
    else
        printf("%d not found in array\n", SearchValue);

    return 0;
}
```

```c
/* C program for Linear Search */
#include <stdlib.h>
#include <stdio.h>

/* Function to print an array */
void printArray(int A[], int size)
{
   int i;
   for (i=0; i < size; i++)
      printf("%d ", A[i]);
   printf("\n\n");
}
```

```
int main(int argc, char *argv[])
{
    int arr[] = {12, 11, 5, 13, 7, 6};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    int i = 0;
    int SearchValue = atoi(argv[1]);

    printArray(arr, arr_size);
```

```
while (i < arr_size && arr[i] != SearchValue)
{
    i++;
}
```

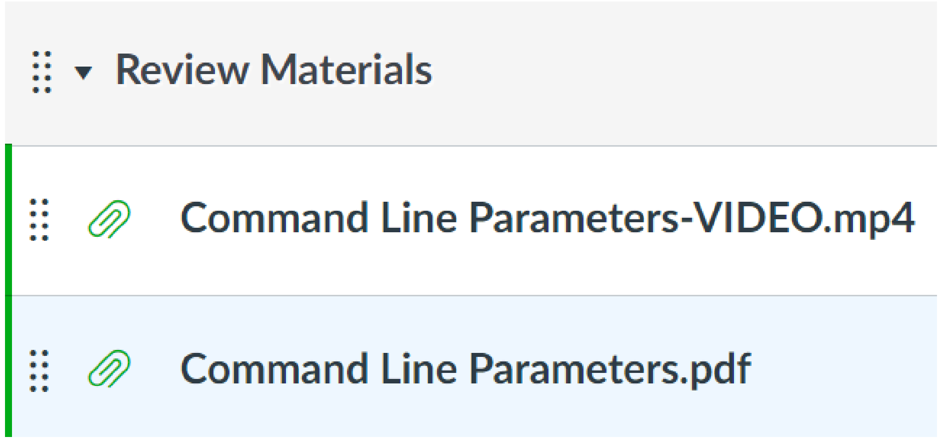
```
    if (arr[i] == SearchValue)
        printf("Found %d in array after %d checks\n",
                SearchValue, i+1);
    else
        printf("%d not found in array\n", SearchValue);

    return 0;
}
```

Notes

This code used several features that you need to be familiar with

1. Used `gcc` to compile from the Linux command line
2. Used command line parameters to give input to the program.



Linear Search Algorithm

Adding the logic to end the search when the value is found helps with the number of searches but only if the search item is found early in the search.

Linear Search is not very useful except for very small sets of data or when sorting would actually take more time.

For example, in class, after everyone turns in their written quiz, I have a stack of unsorted papers. You approach me after class and ask to see your paper to make sure you put on ID on it. To find your quiz, I have to perform a linear search.

It would not be beneficial to sort the stack alphabetically by last name and then find your paper.

Linear Search Algorithm

But, when I store the papers in my filing cabinet, I do alphabetize them first.

Why?

One reason is because it is easier for me to enter them into the gradebook when they are alphabetized because the gradebook is alphabetized.

Also, once the papers are sorted (alphabetized), I can perform a better, much more efficient search if someone asks to see their paper…

Binary Search Algorithm

Binary search is an efficient algorithm for finding an item from a sorted list of items.

It works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed down the possible locations to just one.

One of the most common ways to use binary search is to find an item in an array.

Binary Search

Search 23

0	1	2	3	4	5	6	7	8	9
2	5	8	12	16	23	38	56	72	91

*C:\Users\frenc\Desktop\VM\CSE3318\BinarySearchIterative.c - Notepad++

```c
// C program to implement iterative Binary Search
#include <stdio.h>
#include <stdlib.h>

/* Function to print an array or a portion of an array */
void printArray(int A[], int start, int end)
{
	int i;
	for (i=start; i <= end; i++)
	{
		if (i != end)
			printf("%d,", A[i]);
		else
			printf("%d}\n", A[i]);
	}
}


// Return index of SearchValue or -1 if SearchValue not found
int binarySearch(int SortedArray[], int left, int right, int SearchValue)
{
	int middle = (right + left) / 2;
	
	while (left <= right && SortedArray[middle] != SearchValue)
	{
		// If SearchValue greater, ignore left half
		if (SortedArray[middle] < SearchValue)
			left = middle + 1;
		else /* ignore right half */
			right = middle - 1;

		if (left <= right && SortedArray[middle] != SearchValue)  // print array
		{
			printf("Now searching this portion of the array -> {");
			printArray(SortedArray, left, right);
		}

		middle = (right + left) / 2;
	}
	

	if (SortedArray[middle] == SearchValue)  // Found it!
		return middle;
	else
		return -1; // Did not find it
}

int main(void)
{
```

C source file length : 1,814 lines : 72 Ln : 33 Col : 10 Pos : 844 Windows (CR LF) UTF-8 INS

Conditional Compile

```c
#include <stdio.h>
#include <string.h>

int main(void)
{
    char DayOfWeek[10];

    #ifdef Monday
        strcpy(DayOfWeek, "Monday");
    #elif Wednesday
        strcpy(DayOfWeek, "Wednesday");
    #else
        strcpy(DayOfWeek, "Friday");
    #endif

    printf("Today is %s\n\n", DayOfWeek);

    return 0;
}
```

3 different ways to compile this code

```
gcc CondComp.c

gcc CondComp.c -D Monday

gcc CondComp.c -D Wednesday
```

Conditional Compile

```c
#include <stdio.h>
#include <string.h>

int main(void)
{
    char DayOfWeek[10];

    #ifdef Monday
        strcpy(DayOfWeek, "Monday");
    #elif Wednesday
        strcpy(DayOfWeek, "Wednesday");
    #else
        strcpy(DayOfWeek, "Friday");
    #endif

    printf("Today is %s\n\n", DayOfWeek);

    return 0;
}
```

```
gcc CondComp.c -E

int main(void)
{
  char DayOfWeek[10];

  strcpy(DayOfWeek, "Friday");

  printf("Today is %s\n\n", DayOfWeek);

  return 0;
}
```

Conditional Compile

```c
#include <stdio.h>
#include <string.h>

int main(void)
{
    char DayOfWeek[10];

    #ifdef Monday
        strcpy(DayOfWeek, "Monday");
    #elif Wednesday
        strcpy(DayOfWeek, "Wednesday");
    #else
        strcpy(DayOfWeek, "Friday");
    #endif

    printf("Today is %s\n\n", DayOfWeek);

    return 0;
}
```

```
gcc CondComp.c -E -D Monday

int main(void)
{
  char DayOfWeek[10];

  strcpy(DayOfWeek, "Monday");

  printf("Today is %s\n\n", DayOfWeek);

  return 0;
}
```

Conditional Compile

```c
#include <stdio.h>
#include <string.h>

int main(void)
{
    char DayOfWeek[10];

    #ifdef Monday
        strcpy(DayOfWeek, "Monday");
    #elif Wednesday
        strcpy(DayOfWeek, "Wednesday");
    #else
        strcpy(DayOfWeek, "Friday");
    #endif

    printf("Today is %s\n\n", DayOfWeek);

    return 0;
}
```

```
gcc CondComp.c -E -D Wednesday

int main(void)
{
  char DayOfWeek[10];

  strcpy(DayOfWeek, "Wednesday");

  printf("Today is %s\n\n", DayOfWeek);

  return 0;
}
```

```c
int main(void)
{
    //conditional compile
    #ifdef ARRAYSIZE11
        int SortedArray[] = {2, 3, 4, 10, 11, 15, 40, 42, 47, 49, 50};
    #else
        int SortedArray[] = {2, 3, 4, 10, 11, 15, 40, 42, 47, 49};
    #endif

    int NumberOfElements = sizeof(SortedArray) / sizeof(SortedArray[0]);
    int SearchValue = 0;

    printf("Enter search value ");
    scanf("%d", &SearchValue);

    printf("\nSearch array -> {");
    printArray(SortedArray, 0, NumberOfElements-1);

    int result = binarySearch(SortedArray, 0, NumberOfElements - 1, SearchValue);

    (result == -1) ? printf("Element is not present in SortedArray\n")
                   : printf("Element is present at index %d\n", result);
    return 0;
}
```

```c
// Return index of SearchValue or -1 if SearchValue not found
int binarySearch(int SortedArray[], int left, int right, int SearchValue)
{
    int middle = (right + left) / 2;

    while (left <= right && SortedArray[middle] != SearchValue)
    {
        // If SearchValue greater, ignore left half
        if (SortedArray[middle] < SearchValue)
            left = middle + 1;
        else /* ignore right half */
            right = middle - 1;

        if (left <= right && SortedArray[middle] != SearchValue)  // print array
        {
            printf("Now searching this portion of the array -> {");
            printArray(SortedArray, left, right);
        }

        middle = (right + left) / 2;
    }


    if (SortedArray[middle] == SearchValue)  // Found it!
        return middle;
    else
        return -1; // Did not find it
}
```

Binary Search Algorithm

```
Enter search value 4

Search array -> {2,3,4,10,11,15,40,42,47,49}
Now searching this portion of the array -> {2,3,4,10}
Now searching this portion of the array -> {4,10}
Element is present at index 2

Enter search value 15

Search array -> {2,3,4,10,11,15,40,42,47,49}
Now searching this portion of the array -> {15,40,42,47,49}
Now searching this portion of the array -> {15,40}
Element is present at index 5

Enter search value 49

Search array -> {2,3,4,10,11,15,40,42,47,49}
Now searching this portion of the array -> {15,40,42,47,49}
Now searching this portion of the array -> {47,49}
Now searching this portion of the array -> {49}
Element is present at index 9
```

Binary Search Algorithm

```
Enter search value 5

Search array -> {2,3,4,10,11,15,40,42,47,49,50}
Now searching this portion of the array -> {2,3,4,10,11}
Now searching this portion of the array -> {10,11}
Element is not present in SortedArray

Enter search value 2

Search array -> {2,3,4,10,11,15,40,42,47,49,50}
Now searching this portion of the array -> {2,3,4,10,11}
Now searching this portion of the array -> {2,3}
Element is present at index 0

Enter search value 50

Search array -> {2,3,4,10,11,15,40,42,47,49,50}
Now searching this portion of the array -> {40,42,47,49,50}
Now searching this portion of the array -> {49,50}
Now searching this portion of the array -> {50}
Element is present at index 10
```

```c
// Return index of SearchValue or -1 if SearchValue not found
int binarySearch(int SortedArray[], int left, int right, int SearchValue)
{
    int middle = 0;

    if (left <= right)
    {
        middle = (right + left) / 2;

        // Check if SearchValue is present at mid
        if (SortedArray[middle] == SearchValue)
            return middle;

        // If SearchValue greater, ignore left half
        if (SortedArray[middle] < SearchValue)
            return binarySearch(SortedArray, middle + 1, right, SearchValue);

        // If SearchValue is smaller, ignore right half
        else
            return binarySearch(SortedArray, left, middle - 1, SearchValue);

        if (left <= right)  // print array
        {
            printf("Now searching this portion of the array -> {");
            printArray(SortedArray, left, right);
        }
    }

    // Did not find it
    return -1;
}
```

R
E
C
U
R
S
I
O
N

Binary Search Algorithm

Linear search on an array of *n* elements might have to make as many as *n* guesses.

What about a binary search?

If we have 4 elements in an array, the 1st guess, if incorrect, cuts the searchable array down to half of the original size which would leave 2 elements to search.

The 2nd guess, if incorrect, cuts the searchable array down to 1 element.

The next guess is either right or wrong and no more searching is required.

4 elements required, at most, 3 guesses and 2 "divide in half" actions.

Binary Search Algorithm

If we have 8 elements in an array, the 1st guess, if incorrect, cuts the searchable array down to half of the original size which would leave 4 elements to search.

The 2nd guess, if incorrect, cuts the searchable array down to 2 elements.

The 3rd guess, if incorrect, cuts the searchable array down to 1 element.

The next guess is either right or wrong and no more searching is required.

8 elements required, at most, 4 guesses and 3 "divide in half" actions.

Binary Search Algorithm

If we have 16 elements in an array, the 1st guess, if incorrect, cuts the searchable array down to half of the original size which would leave 8 elements to search.

The 2nd guess, if incorrect, cuts the searchable array down to 4 elements.

The 3rd guess, if incorrect, cuts the searchable array down to 2 elements.

The 4th guess, if incorrect, cuts the searchable array down to 1 element.

The next guess is either right or wrong and no more searching is required.

16 elements required, at most, 5 guesses and 4 "divide in half" actions.

Binary Search Algorithm

If we have 32 elements in an array, the 1st guess, if incorrect, cuts the searchable array down to half of the original size which would leave 16 elements to search.

The 2nd guess, if incorrect, cuts the searchable array down to 8 elements.

The 3rd guess, if incorrect, cuts the searchable array down to 4 elements.

The 4th guess, if incorrect, cuts the searchable array down to 2 elements.

The 5th guess, if incorrect, cuts the searchable array down to 1 element.

The next guess is either right or wrong and no more searching is required.

32 elements required, at most, 6 guesses and 5 "divide in half" actions

Binary Search Algorithm

4 elements -> 3 guesses -> and 2 "divide in half" actions

8 elements -> 4 guesses -> and 3 "divide in half" actions

16 elements -> 5 guesses -> and 4 "divide in half" actions

32 elements -> 6 guesses -> and 5 "divide in half" actions

See the pattern?

Every time the size of the array doubled, we added one more guess.

Binary Search Algorithm

Let *m* represent the number of times the array was halved and *m*+1 represent the number of guesses. Let *n* represent the number of elements in the array.

n elements	*m* halving actions	*m*+1 total guesses
4	2	3
8	3	4
16	4	5
32	5	6

We can describe the number of guesses, in the worst case, as "the number of times we can repeatedly halve, starting at *n*, until we get the value 1, plus one."

Binary Search Algorithm

"the number of times we can repeatedly halve, starting at *n*, until we get the value 1, plus one."

n **elements**	***m*** **halving actions**	***m*+1** **total guesses**
4	2	3
8	3	4
16	4	5
32	5	6

Does this sound familiar? Does this pattern look familiar?

Binary Search Algorithm

"the number of times we can repeatedly halve, starting at *n*, until we get the value 1, plus one."

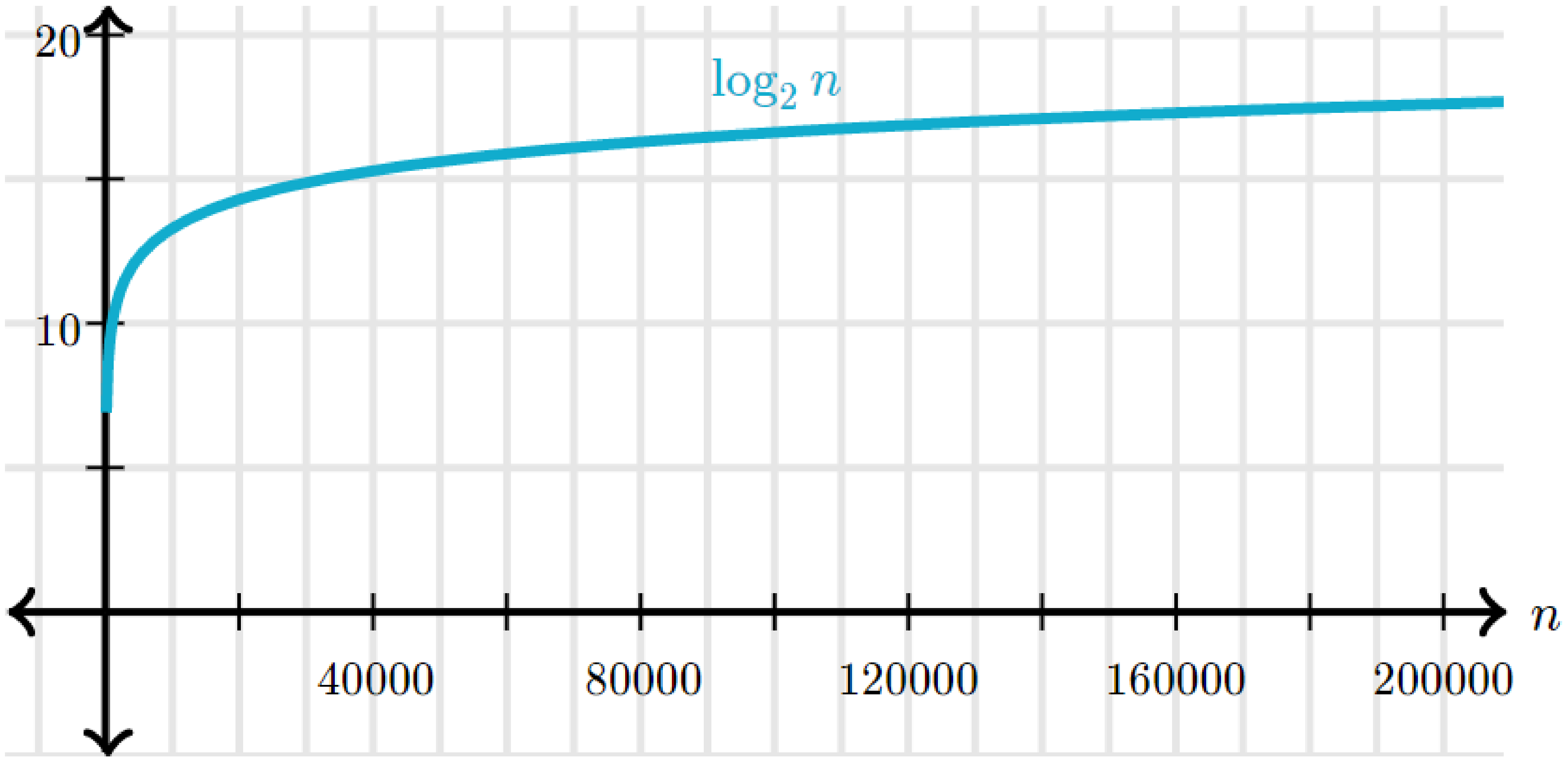
n elements	*m* halving actions	*m*+1 total guesses
4	2	3
8	3	4
16	4	5
32	5	6

n	$\log_2 n$
1	0
2	1
4	2
8	3
16	4
32	5
64	6

This is describing the mathematical function base 2 logarithm of *n*

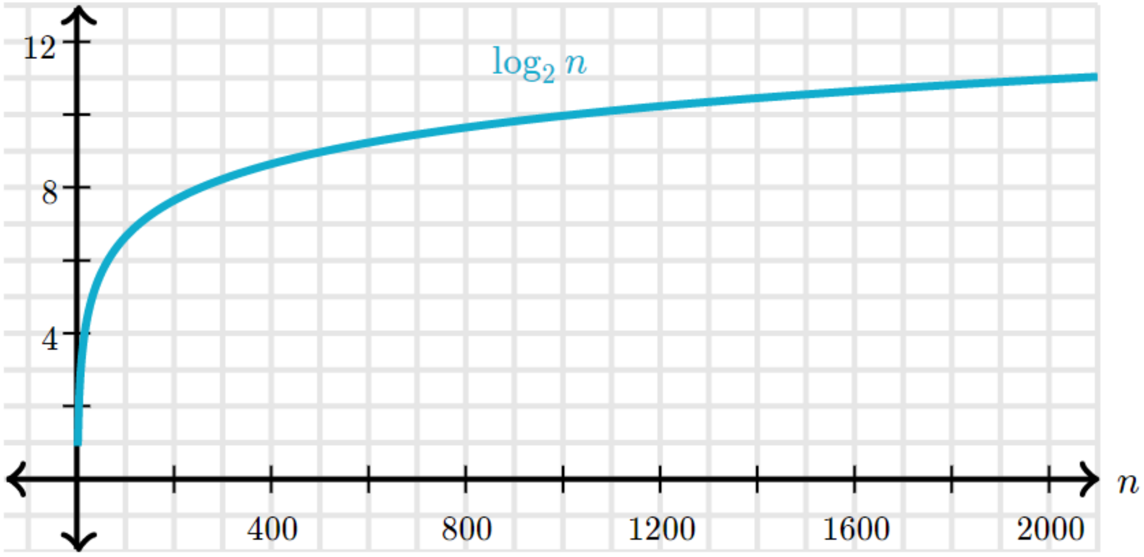
Binary Search Algorithm

n	$\log_2 n$
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024	10
1,048,576	20
2,097,152	21



Binary Search Algorithm

n	$\log_2 n$
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024	10
1,048,576	20
2,097,152	21

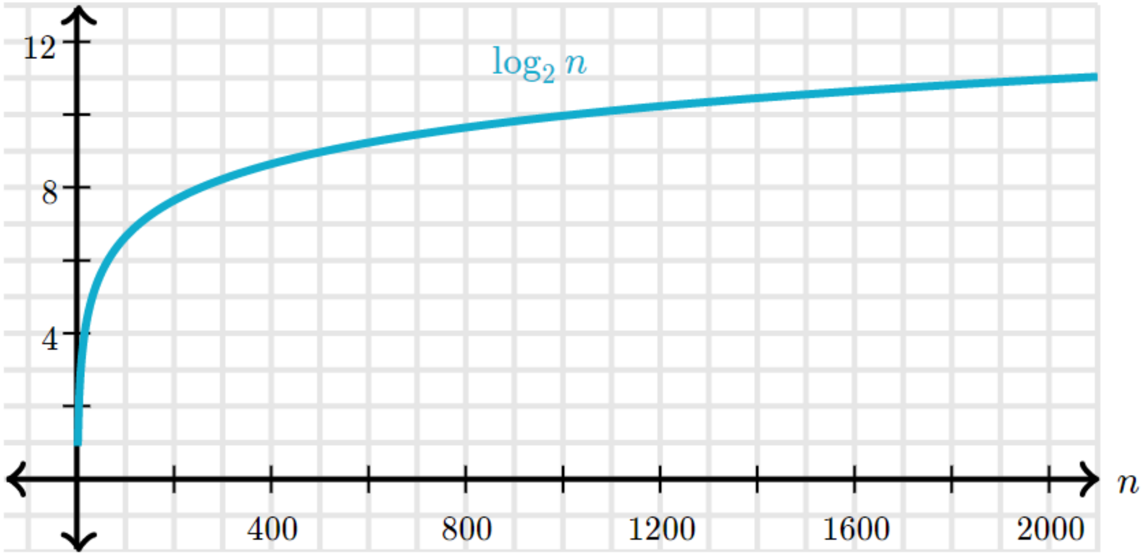


The logarithm function grows very slowly.

Logarithms are the inverse of exponentials, which grow very rapidly.

Binary Search Algorithm

n	$\log_2 n$
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024	10
1,048,576	20
2,097,152	21



Remember that $\log_2 n = x$ means $n = 2^x$

$\log_2 128 = 7$ because $2^7 = 128$

$\log_2 2097152 = 21$ because $2^{21} = 2{,}097{,}152$

Binary Search Algorithm

It is easy to calculate the runtime of a binary search algorithm where the number of elements being searched, *n*, is exactly a power of 2.

If *n* is 128, how many guesses will a binary search require (worst case)?

For a *n* of 128, a binary search will require at most

$\log_2 128$

A total of 8 guesses (7 + 1).

Binary Search Algorithm

What if *n* isn't a power of 2?

We would look at the closest lower power of 2.

For an *n* of 1000 (remember that *n* is the number of elements in the array), what is the closest **lower** power of 2?

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
1	2	4	8	16	32	64	128	256	512	1024

Binary Search Algorithm

For an *n* of 1000 (remember that *n* is the number of elements in the array), what is the closest **lower** power of 2?

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
1	2	4	8	16	32	64	128	256	512	1024

It might be tempting to go with 2^{10} since 1024 is SO much closer to 1000 than 512 (2^9).

Why is that not a good plan?

Binary Search Algorithm

2^9 (512) vs 2^{10} (1024) when n = 1000

Since $\log_2 512$ is 9 and $\log_2 1024$ is 10, we can estimate that $\log_2 1000$ is between 9 and 10.

The actual value of $\log_2 1000$ is approximately 9.97.

Since we don't have 0.97th of a guess, we take just the 9 and add that one final guess to get a total number of guesses of 10 (worst case).

Binary Search Algorithm

Did we need to get out the calculator to get the exact value of $\log_2 n$ in this case?

What if we had just used 1024 because it closest to 1000?

$\log_2 1024$ is 10. Adding that one final guess would make the total number of guesses 11.

$\log_2 512$ is 9. Adding that one final guess would make the total number of guesses 10.

Binary Search Algorithm

1000

1st guess -> 500
2nd guess -> 250
3rd guess -> 125
4th guess -> 62
5th guess -> 31

6th guess -> 15
7th guess -> 7
8th guess -> 3
9th guess -> 1
Final guess

TOTAL -> 10

Using 1024 for our estimate leads to a number of guesses of 11

Using 512 for our estimate leads to a number of guess of 10.

10 is correct.

This is why we would use the closest **lower** power of 2.

Binary Search Algorithm

The Tycho-2 Catalog is an astronomical catalog of more than 2.5 million of the brightest stars.

2,539,913 of the brightest stars in the Milky Way to be more precise.

Each star is numbered using its Guide Star region number (0001-9537) and a five-digit star number within each region, separated by a decimal point.

Binary Search Algorithm

Let's say that someone pays to have a star named for you. You are presented with a certificate like this.

How many guesses would it take (worst case) to find your star in the Tycho-2 Catalog which has 2,539,913 entries?

Binary Search Algorithm

Your certificate listed the Tycho-2 catalog number.

STAR ASTRONOMICAL COORDINATES:
Catalog Number: TYC 4149-1136-1
Constellation: Ursa Major

What is the worst case scenario for a linear search?

2,539,913

Binary Search Algorithm

What is the worst case scenario for a binary search?

n is 2,539,913 which is not a power of 2.

What is the closest **lower** power of 2?

2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
2048	4096	8192	16384	32768	65536	131072	262144	524288	1048576	2097152

Do we need to calculate 2^{22}?

Not really – we know that would go way beyond 2,539,913 and we want the closest **lower** power so 2^{21} is what we need.

Binary Search Algorithm

So 2^{21} is the closest lower power of 2.

$\log_2 2097152$ is 21

Adding the 1 final guess gives us

22 total guesses is the worst case scenario.

22 is just a little bit better than searching 2,539,913 entries for our star!!

Asymptotic Notation

Previously, we analyzed linear search and binary search by determining the maximum number of guesses we need to make.

The next thing we what to determine is how long these algorithms take to run.

The running times of linear search and binary search include the time needed to make and check guesses, but there's more steps to factor into how LONG the search will run.

Time matters - not just guesses.

Asymptotic Notation

The running time of an algorithm depends on how long it takes a computer to run the lines of code of the algorithm.

This running time is dependent on several factors including

- speed of the computer
- programming language
- compiler that translates the program from the programming language into code that runs directly on the computer

Asymptotic Notation

Two of the factors that determine the running time of an algorithm are

how long the algorithm takes, in terms of the size of its input

how fast a function grows with the input size

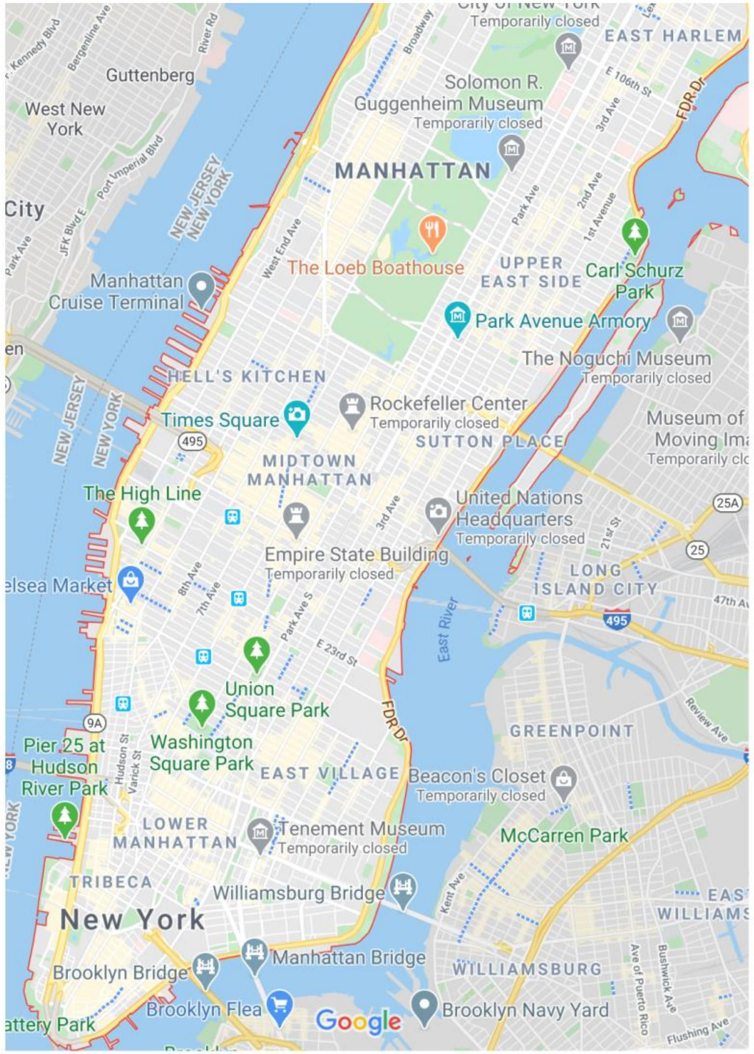
Asymptotic Notation

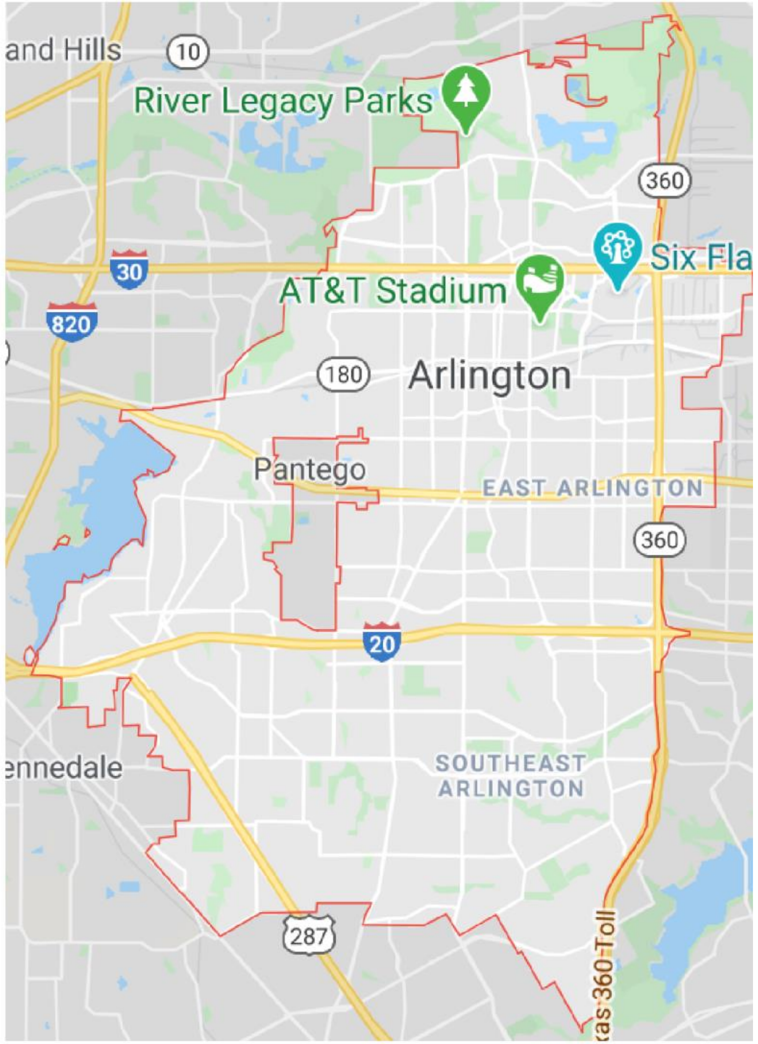
Let's take the first factor

how long the algorithm takes, in terms of the size of its input

We saw how the maximum number of guesses in linear search and binary search increased as the length of the array increased.

How about a GPS algorithm?





Manhattan, NY – 22.82 square miles – 12,000+ intersections

Arlington, TX – 99.69 square miles – not 12,000 intersections

Asymptotic Notation

The second factor that determines the running time of an algorithm is

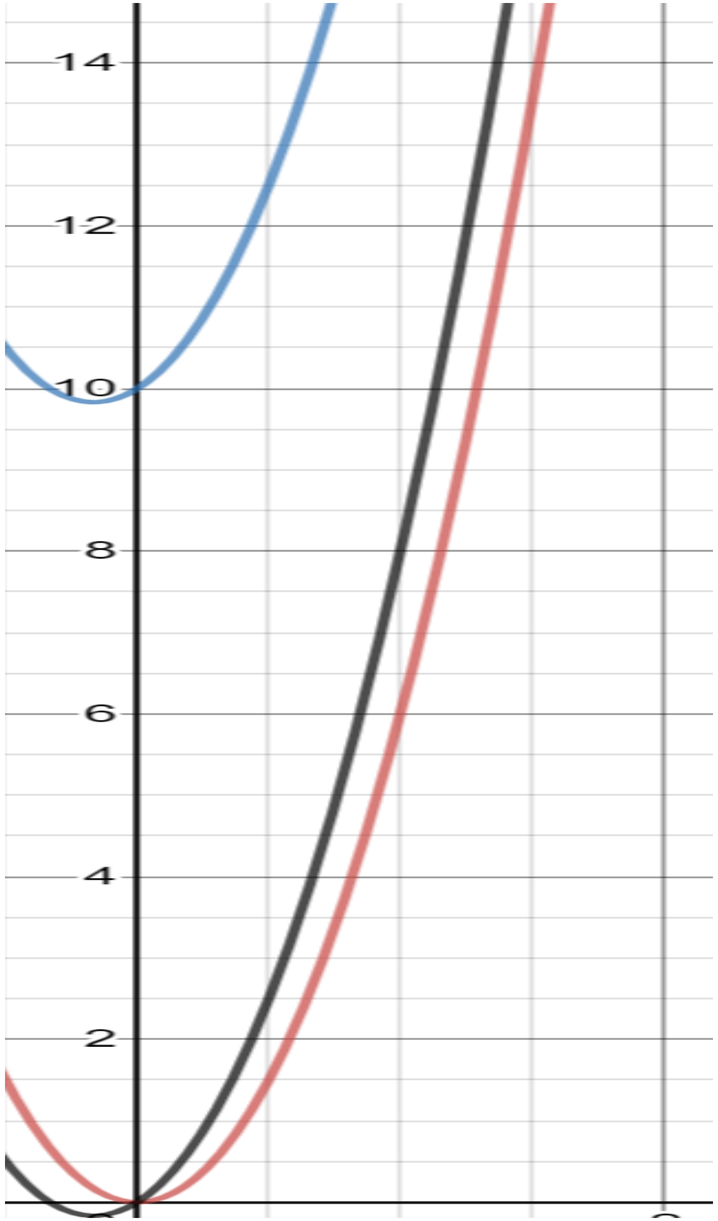
how fast a function grows with the input size

This is called rate of growth of the running time.

We will discover that simplifying our functions allows us to better compare different run times and has no impact on the rate of growth.

We will see how to make a function more manageable by discarding less important parts of a function.

Asymptotic Notation



Let's compare the following 3 equations

$6x^2$

$6x^2 + 2x$

$6x^2 + 2x + 10$

At smaller values, they are different..

But what happens as x grows?

Asymptotic Notation

But what happens as x grows?

$6x^2$

$6x^2 + 2x$

$6x^2 + 2x + 10$

x = 5

$6(5)^2$	=	150
$6(5)^2 + 2(5)$	=	160
$6(5)^2 + 2(5) + 10$	=	170

x = 500

$6(500)^2$	=	1,500,000
$6(500)^2 + 2(500)$	=	1,501,000
$6(500)^2 + 2(500) + 10$	=	1,501,010

x = 1,000,000

$6(1{,}000{,}000)^2$	=	6,000,000,000,000
$6(1{,}000{,}000)^2 + 2(1{,}000{,}000)$	=	6,000,002,000,000
$6(1{,}000{,}000)^2 + 2(1{,}000{,}000) + 10$	=	6,000,002,000,010

Asymptotic Notation

The larger x becomes, the less difference there is between the outcomes of the equations.

So if the running time of an algorithm is calculated as

$6n^2 + 2n + 100$

we can simplify the equation to just n^2 to describe the running time.

Dropping the coefficient 6 and the remaining terms ($2n + 100$) does not have a large enough impact on the overall running time as n approaches infinitely.

It doesn't really matter what coefficients we use - as long as the running time is $an^2 + bn + c$ for some numbers $a > 0$, b and c, there will always be a value of an^2 is greater than $bn + c$ and this difference increases as n increases.

Asymptotic Notation

By dropping the less significant terms and the constant coefficients, we can focus on the important part of an algorithm's running time

its rate of growth

without getting lost in details that could complicate our understanding and mislead us when comparing the running time of algorithms

When we drop the constant coefficients and the less significant terms, we use **asymptotic notation**.

We'll see three forms of it

- big-Θ notation (theta)
- big-O notation
- big-Ω notation (omega)

The definition of **asymptotic** is a line that approaches a curve but never touches.

A curve and a line that get closer but do not intersect are examples of a curve and a line that are **asymptotic** to each other.

Asymptotic Notation

Why is the coefficient usually not that important with algorithms?

Typically, we just want to compare the running times of algorithms that perform the same task.

> Algorithms tend to have different dominant terms (meaning they are in different complexity classes), which will allow us to easily identify which algorithms have faster running times for large values of *n*.

Calculating the coefficients for the running time of algorithms is often time consuming, error prone and inexact.

Identifying the most significant term for the running time is more straight forward.

Algorithms in the same complexity class that perform the same task typically have similar coefficients with some small differences that indicate improvements between algorithms in the same complexity class.

Asymptotic Notation

The words "typically" and "usually" and "similar" and "tend to" were used pretty heavily when describing why it's OK to drop coefficients and other terms.

We will soon see that the value of n plays a role is using these not-so-concrete descriptors.

We will study sorting algorithms (like insertion sort) that have a n^2 run time.

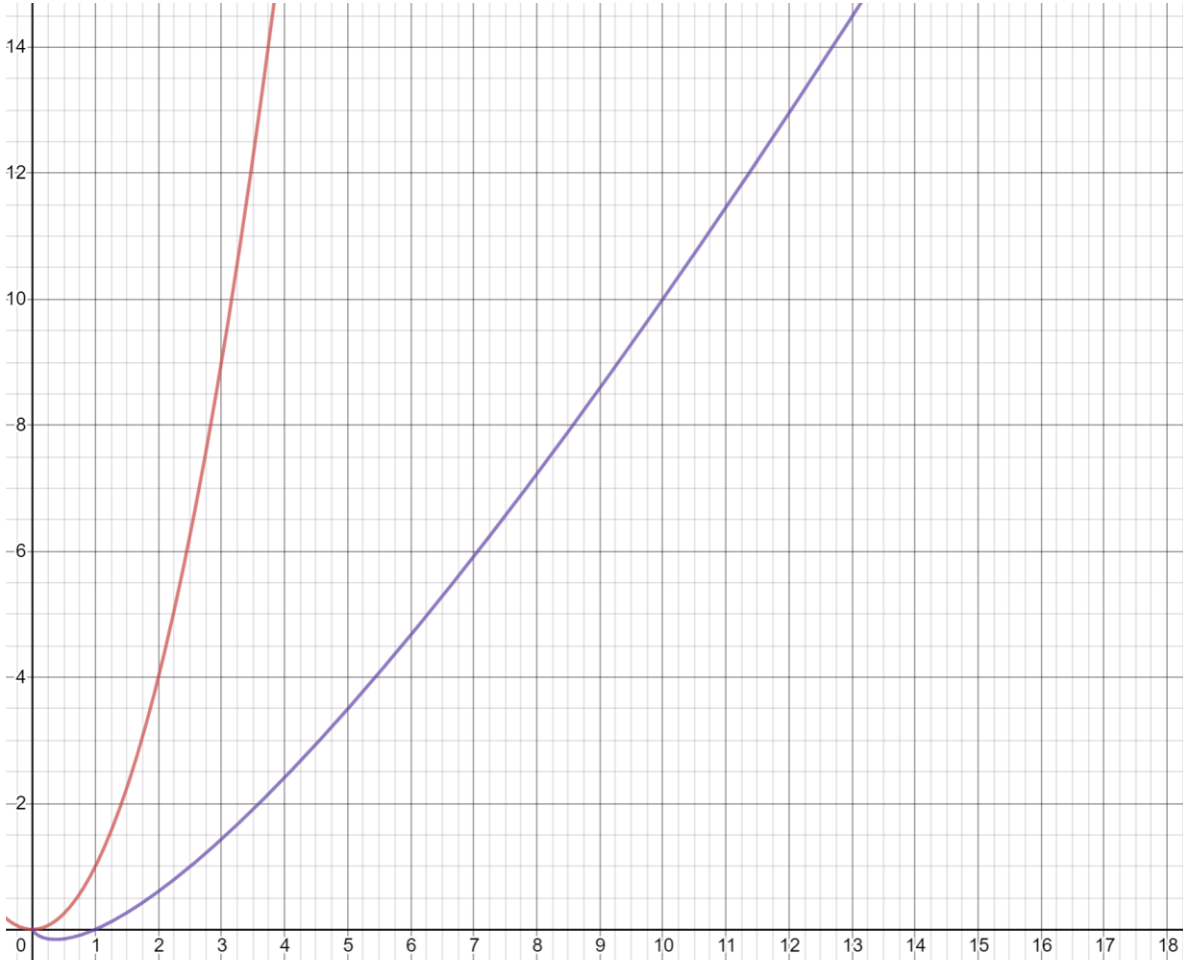
We will study sorting algorithms (like merge sort) have $n\log_2 n$ run times.

Asymptotic Notation

We have compared n^2 to $n\log_2 n$ run times.

We can see that n^2 rises much quicker than $n\log_2 n$, which, in general terms, means that $n\log_2 n$ is the better algorithm in terms of running time.

So why would we ever use an algorithm with an n^2 run time?



Asymptotic Notation

For smaller values of n, those coefficients can matter.

If the n^2 algorithms have small coefficients and the $n\log_2 n$ algorithms have large coefficients, then the coefficients will come into play when the value of n is smaller. Only with larger values of n do we see n^2 algorithms running slower than the $n\log_2 n$ algorithms.

Asymptotic notation can be a really useful to talk about and compare algorithms.

It is definitely not without its limitations.

Asymptotic Notation

Let's look at a linear search function that returns when it finds the value

```
for (i = 0; i < array_size; i++)
{
     if (array[i] == SearchValue)
     {
          return i;
     }
}
return -1;
```

Asymptotic Notation

For each loop, several things happen

- check that `i < array_size`
- compare `array[i]` with `SearchValue`
- possibly return `i`
- increment `i`

```
for (i = 0; i < array_size; i++)
{
   if (array[i] == SearchValue)
   {
      return i;
   }
}
return -1;
```

Asymptotic Notation

Each step takes a constant amount of time each time the loop executes

- check that `i < array_size`
- compare `array[i]` with `SearchValue`
- possibly return `i`
- increment `i`

Let's call the sum of all those times c_1.

So if the for loop iterates n times, then the time needed for all iterations can be expressed as $c_1 n$

Asymptotic Notation

So what is the value of the c_1 in this $c_1 n$ formula?

The answer is

it depends

What is the

- speed of the computer?
- the programming language used?
- the compiler or interpreter that translates the source program into runnable code?
- other factors?

Asymptotic Notation

Are there steps in addition to the for loop steps?

`i` is initialized to 0

-1 will be returned when `SearchValue` is not found in the array

Let's sum up this time and call it c_2

```
for (i = 0; i < array_size; i++)
{
   if (array[i] == SearchValue)
   {
      return i;
   }
}
return -1;
```

Asymptotic Notation

So, the total time for linear search in the worst case is

$c_1 n + c_2$

We know the bigger n gets, the less significant c_1 and c_2 become.

So much so that we can drop them and just say the worst case is

n

where n is the size of the array to be searched.

Asymptotic Notation

The average running time of linear search grows as the array grows.

The notation used to describe this behavior is

$\Theta(n)$

Big Theta of *n*