**ASYMPTOTIC NOTATION**

Big ‘O’ Notation

* As size of inputs increase to infinity, how does the run time of the program grow

Linear time -> reading through a string.

* O(n)
  + Time required is proportional to number of characters
* Omega (1) -> at best
* O (n) -> at worst

Constant time -> reading the value of a variable

* O(1)
* Omega (1) -> best case
* O (1) -> worst case
* Theta (1) -> same worst and best case.

Exponential time

* O(n^2)

Logarithmic time

* O (log n)
  + Binary search
  + Cut array in half with each operation
  + Ex. Size 8 🡪 3 operations (Log s3 8)
* Omega (1) 🡪 best case
* O (log n) 🡪 worst case

**BINARY SEARCH**

Recursive function

* Find midpoint, if key is greater than midpoint, set min to midpoint +1 and leave max as max. 🡪 repeat

Binary search tree

* Left side of tree, less than or equal to nodes value
* Right side, greater than or equal to nodes value

**BUBBLE SORT**

* [ 3 2 9 6 5]
  + Check 3 and 2, 2 less than 3, move 3 to the right
    - [2 3 9 6 5]
  + Repeat, 3 and 9, no swap
  + Look at 9 and 6, out of order, swap.
  + After first complete path, [ 2 3 6 9 5]
    - Need to go through list again.
* After one iteration, the largest iteration will be in its right place.
* n^2 🡪 have to make n iterations through a list, checking all n elements through each pass through.

**COMMAND LINE ARGUMENTS**

int main (int argc, string argv[])

* argv[0] -> always file name
* can print the characters using a multi dimensional array
* arguments above argv[0] are command line arguments executed prior to compilation.

**INSERTION SORT**

Idea – compare unsorted elements (one by one) to sorted portion.

* Append that element to left or right of the sorted portion.

1. Run from unsorted left to right
2. Run in sorted from right to left until that element is greater than the element on the left.

* Worst case -> Sum i = 1, to n -1 = n (n – 1) / 2 = n^2 /2 – n / 2.
  + O (n^2)
* Best case -> Omega (n)

**LINEAR SEARCH**

* Going through each item takes same amount of time

**SELECTION SORT**

**Idea** – divide list into sorted and unsorted. Select and put unsorted into sorted.

* Find smallest unsorted element by looking through every unsorted element.
  + Move it into sorted portion of the list. Put it at beginning of the list.
  + Swap first and lowest.
  + We now know 4 is in final location. So can ignore it on next iteration.
* Find smallest unsorted element in n-1 remaining items.
  + Append the new lowest to sorted portion of the list. Swap what is sitting next to that first element with that new lowest item.

Worst Case – O (n^2)

Best Case - Omega (n^2)