# CS 137 Week 7

## Big-Oh Notation, Linear Searching and Basic Sorting Algorithms

October 30th, 2017

# Big-Oh Notation

- Up to this point, we've been writing code without any consideration for optimization.
- There are two major ways we try to optimize code - one is for memory storage and the one we will focus on is time complexity.
- One major issue is quantifying measuring time complexity. For example, using how fast it runs is very machine dependent and can depend on everything from processor to type of operating system.
- To level the playing field, we use Big-Oh Notation.

# Definition

## Big-Oh Notation

Let $f(x)$ be a function from the real numbers to the real numbers. Define $O(f(x))$ to be the set of function $g(x)$ such that there exists a real number $C > 0$ and a real $X$ such that $|g(x)| \leq C|f(x)|$ for all $x \geq X$.

or more symbolically

## Big-Oh Notation

$$g(x) \in O(f(x))$$

$$\Leftrightarrow$$

$$\exists C \in \mathbb{R}_{>0}\ \exists X \in \mathbb{R}\ \forall x \in \mathbb{R}\ (x \geq X \Rightarrow |g(x)| \leq C|f(x)|)$$

# Examples

## Big-Oh Notation

$$g(x) \in O(f(x))$$

$$\Leftrightarrow$$

$$\exists C \in \mathbb{R}_{>0}\ \exists X \in \mathbb{R}\ \forall x \in \mathbb{R}\ (x \geq X \Rightarrow |g(x)| \leq C|f(x)|)$$

For example, $3x^2 + 2 \in O(x^2)$ since for all $x \geq 1$, we have

$$|3x^2 + 2| = 3x^2 + 2 \leq 5x^2 = 5|x^2|.$$

Note in the definitions, $X = 1$ and $C = 5$.

# Another Example

## Big-Oh Notation

$$g(x) \in O(f(x))$$

$$\Leftrightarrow$$

$$\exists C \in \mathbb{R}_{>0}\ \exists X \in \mathbb{R}\ \forall x \in \mathbb{R}\ (x \geq X \Rightarrow |g(x)| \leq C|f(x)|)$$

As another example $6\sin(x) \in O(1)$ since for all $x \geq 0$, we have that

$$|6\sin(x)| \leq 6|1|.$$

# Some Notes

- In computer science, most of our $f$ and $g$ functions take positive integers to positive integers so the absolute values are normally unnecessary for us. However this concept is used in other areas of mathematics (most notably Analytic Number Theory) so I'm presenting the general definition.
- Usually instead of $g(x) \in O(f(x))$, we write $g(x) = O(f(x))$ because we're sloppy.
- Note that we only care about behaviour as $x$ tends to infinity. Indeed we could consider $x$ approaching some fixed $a$ which is often done in analysis (calculus).
- In fact, in compute science, our functions are almost always $f : \mathbb{N} \to \mathbb{R}$ or even more likely, $f : \mathbb{N} \to \mathbb{N}$

# A Useful Theorem

To help with classifying functions in terms of Big-Oh notation, the following is useful.

## Limit Implication

For positive real valued functions $f$ and $g$, if $\lim_{x\to\infty} \frac{f(x)}{g(x)} < \infty$, then $f(x) = O(g(x))$.

Often times, this with L'Hôpital's rule can be used to help determine which functions grow faster very quickly. The converse of this statement is almost true (if we change the limit to the lim sup) but I will save this for another course.

# More examples

## Big-Oh Notation

$$g(x) \in O(f(x))$$
$$\Leftrightarrow$$
$$\exists C \in \mathbb{R}_{>0}\ \exists X \in \mathbb{R}\ \forall x \in \mathbb{R}\ (x \geq X \Rightarrow |g(x)| \leq C|f(x)|)$$

**Claim:** For any polynomial $g(x) = \sum_{i=0}^{n} a_i x^i$ then $g(x) \in O(x^n)$.

**Proof:** For all $x \geq 1$, we have

$$\begin{aligned} |a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0| &\leq |a_n|x^n + \ldots + |a_1|x + |a_0| \\ &\leq |a_n|x^n + \ldots + |a_1|x^n + |a_0|x^n \\ &\leq (|a_n| + \ldots + |a_1| + |a_0|)x^n \end{aligned}$$

and the bracket above is a constant with respect to $x$ and we are done.

# Another Example

**Claim:** $\log_a x \in O(\log_b x)$ for $a, b > 0$.

**Proof:** For all $x \geq 1$, we have

$$|\log_a x| = \log_a x = \frac{\log_b x}{\log_b a} = \frac{1}{\log_b a} |\log_b x|$$

and the bracket above is a constant with respect to $x$ and we are done.

# Another Example

**Claim:** $\log_a x \in O(\log_b x)$ for $a, b > 0$.

**Proof:** For all $x \geq 1$, we have

$$|\log_a x| = \log_a x = \frac{\log_b x}{\log_b a} = \frac{1}{\log_b a} |\log_b x|$$

and the bracket above is a constant with respect to $x$ and we are done.

**Summary:** We can ignore the base of the logarithm; it is rarely used in Big-Oh Notation.

# Some results

In what follows (for real valued functions), let $g_0(x) \in O(f_0(x))$ and $g_1(x) \in O(f_1(x))$. Then...

1. $g_0(x) + g_1(x) \in O(|f_0(x)| + |f_1(x)|)$
2. $g_0(x)g_1(x) \in O(f_0(x) \cdot f_1(x))$
3. $O(|f_0(x)| + |f_1(x)|) = O(\max\{|f_0(x)|, |f_1(x)|\})$

Note that the last bullet is actually a set equality! I'll prove 1 and 3 on the next slides.

Also note that if $f_0$ and $f_1$ are positive functions then bullets 1 and 3 can have their absolute values removed.

# Proofs

Prove that $g_0(x) + g_1(x) \in O(|f_0(x)| + |f_1(x)|)$.

**Proof:** Given $g_0(x) \in O(f_0(x))$ and $g_1(x) \in O(f_1(x))$, we know that there exists constants $C_0, C_1, X_0, X_1$ such that

$$|g_0(x)| \leq C_0|f_0(x)| \ \forall x > X_0 \qquad \text{and} \qquad |g_1(x)| \leq C_1|f_1(x)| \ \forall x > X_1$$

Now, let $X_2 = \max\{X_0, X_1\}$ and $C_2 = 2\max\{C_0, C_1\}$ to see that by the triangle inequality, for all $x > X_2$,

$$\begin{aligned}
|g_0(x) + g_1(x)| &\leq |g_0(x)| + |g_1(x)| \\
&\leq C_0|f_0(x)| + C_1|f_1(x)| \\
&\leq \max\{C_0, C_1\}(|f_0(x)| + |f_1(x)|) \\
&\quad + \max\{C_0, C_1\}(|f_0(x)| + |f_1(x)|) \\
&\leq 2\max\{C_0, C_1\}(|f_0(x)| + |f_1(x)|) \\
&\leq C_2(|f_0(x)| + |f_1(x)|)
\end{aligned}$$

and so $g_0(x) + g_1(x) \in O(|f_0(x)| + |f_1(x)|)$

# Proofs

Prove that $O(|f_0(x)| + |f_1(x)|) = O(\max\{|f_0(x)|, |f_1(x)|\})$.

**Proof:** Notice that

$$\begin{aligned}
\max\{|f_0(x)|, |f_1(x)|\} &\leq |f_0(x)| + |f_1(x)| \\
&\leq \max\{|f_0(x)|, |f_1(x)|\} + \max\{|f_0(x)|, |f_1(x)|\} \\
&\leq 2\max\{|f_0(x)|, |f_1(x)|\}
\end{aligned}$$

# More Properties

**Transitivity**

If $h(x) = O(g(x))$ and $g(x) = O(f(x))$ then $h(x) = O(f(x))$.
(where $f, g$ and $h$ are real valued functions).

# Even More Notation

## Inequalities

We write $f \ll g$ if and only if $f(x) = O(g(x))$

## Growth Rates of Functions

The following is true for $\epsilon$ and $c$ fixed positive real constants

$$1 \ll \log n \ll n^{\epsilon} \ll c^{n} \ll n! \ll n^{n}$$

In order left to right: constants, logarithms, polynomial (if $\epsilon$ is an integer), exponential, factorials.

# Final Note

- You should be aware that there are many other notations here for runtime.
- Big-Oh notation $O(f(x))$ is an upper bound notation ($\leq$)
- Little-Oh notation $o(f(x))$ is a weak upper bound notation ($<$)
- Big-Omega notation $\Omega(f(x))$ is a lower bound notation ($\geq$)
- Little-Omega notation $\omega(f(x))$ is a weak lower bound notation ($>$)
- Big-Theta (or just Theta) notation $\Theta(f(x))$ is an exact bound notation ($=$)

Ideally, we want the Big-Oh notation to be as tight as possible (so really we want to use $\Theta$ notation but it involves far too large of a detour). In our class when you are asked for the runtime or anything related to Big-Oh notation, make it the best possible bound.

# Algorithms

- Now that the abstraction is over, we would like to apply the notation to see how algorithms compare to one another.
- Our first algorithm we will look at is a linear searching algorithm (see next page).
- We will see that the runtime will usually depend on the type of case.
- Typically with Big-Oh notation (especially if unspecified) we will likely want to analyze the runtime of the worst case [sometimes the 'average/typical case' as well].

# Linear Search

```
#include <stdio.h>
/*Post: Returns index of value in a if found,
   -1 otherwise */
int lin_search(int a[], int n, int value) {
   for (int i = 0; i < n; i++) {
      if (a[i] == value) return i;
   }
   return -1; // value not found
}
int main(void) {
   int a[5] = {19,4,2,3,6};
   int len = sizeof(a)/sizeof(a[0])
   int val = lin_search(a,len,4);
   printf("%d\n",val);
   return 0;
}
```

# Analyzing the Code

There are three major lines of code

1. The for loop
2. The if statement
3. The return statement(s)

Typically, we assume that basic operations like arithmetic operations, return statements, relational comparisons are run in constant time (though strictly speaking this is false). Since we also only care about the analysis when `n` is large, we will think of `n` as being a large variable (in this case, it is the length of the array).

# Best Case analysis

- In the best case, the code will find `value` right away in `a[0]`.
- In this case, the for loop runs the initialization of `i`, then it checks the condition (it passes) then it checks the if condition which involves an array memory retrieval to get to `a[0]` and compares it to `value` and then since we have a match it executes the `return` statement.
- All of the aforementioned steps are basic and so we in this case get a constant time runtime, or in Big-Oh notation, this algorithm in the best case runs in $O(1)$ notation.

# Worst Case Analysis

- In the worst case, the element `value` is not in the array.
- In this case, we go through all the steps we did in the previous slide once for each element in the array since we never find a match and then add one at the end for the `return` statement.
- Thus, we will execute

$$1 + (1 + 1 + 1 + 1)n + 1 = O(n)$$

total number of steps.

# Average/Typical Case analysis

- It's a bit hard to state the average case here since depending on `a`, it might not contain the element `value`.
- We will consider the average case in this setting to mean that the value we are looking for is located in each possible location with even probability.
- In the formula on the previous page, we will actually take a sum from `i=0` to `i=n-1` of each of the possible places the value can occur.
- This gives (an extra one for the return statement triggered when we find the element in the $i$th spot)

$$\frac{1}{n}\left(1+\sum_{i=0}^{n-1}(1+1+1+1+1)(i+1)\right) = \frac{1}{n}+\frac{4}{n}\frac{n(n+1)}{2} = O(n)$$

# Post Mortem

- The previous example really shows why 'average/typical' case is tough to consider and deal with.
- Dealing with the worst case is a far safer and more universal metric to deal with and is often sufficient for our understanding.

# Handling Big-Oh Runtime Questions

Below is just a heuristic for how I try to approach runtime problems. They can be quite tricky if you're not careful.

1. Step one to analyzing the code is getting a 'holistic' view of what the code does.
2. Step two is to understand if you want best, worst or average case runtime.
3. Step three is to try to figure out how many times each line of code will run in each case.
4. Step four is to do manipulations based on theorems you know to reach a simplified answer.

# Sorting

- In the next several lectures, we're going to use these runtime skills to both create and analyze a collection of sorting algorithms.
- We will discuss Selection, Insertion, Merge and Quick Sorting algorithms.
- General paradigm: `void sort(int a[], int n)` takes an unsorted array to a sorted one
- For example, If `a = {19,4,10,16,3}`, then `void sort(a,5)` mutates `a` to `3,4,10,16,19`.

# Selection Sort

The Idea

1. Find the smallest element
2. Swap with the first element
3. Repeat with the rest of the array

# Example

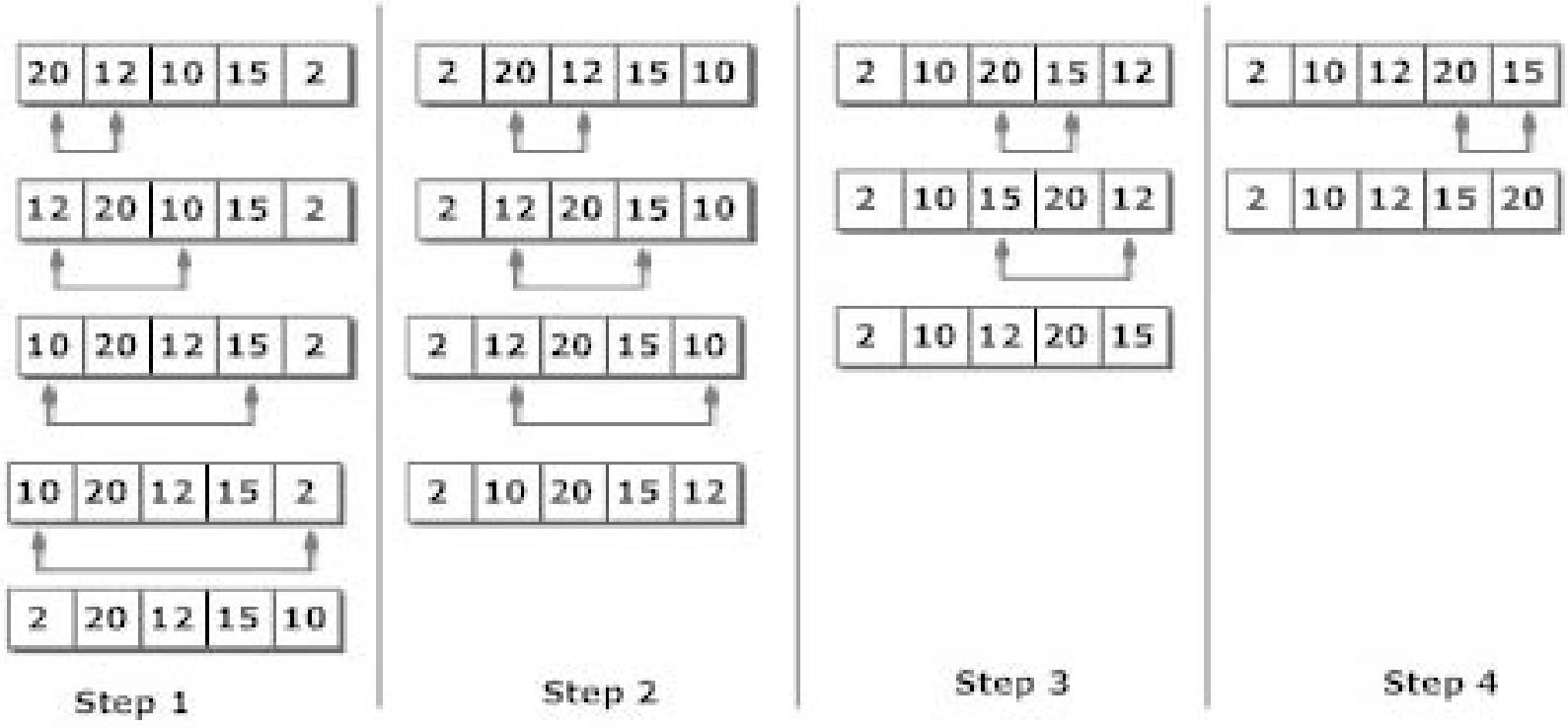



Figure: Selection Sort

https://www.programiz.com/dsa/selection-sort

# Algorithm

```
#include <stdio.h>
void selection_sort (int a[], int n) {
  for (int i = 0; i < n-1; i++) {
    //Find min position
    int min = i;
    for (int j = i+1; j<n; j++)
      if (a[j] < a[min]) min = j;
    //Swap
    int temp = a[min];
    a[min] = a[i];
    a[i] = temp;
  }
}
```

# Testing

```
int main (void){
   int a[] = {20,12,10,15,2};
   int n = sizeof(a)/sizeof(a[0]);
   selection_sort(a,n);
   for (int i = 0; i < n-1; i++) {
        printf("%d, ", a[i]);
   }
   printf("%d\n", a[n-1]);
   return 0;
}
```

# Time Complexity of Selection Sort

- Note that all the operations in the code are $O(1)$ operations so all we care about is how many times they are run.
- The outer loop with `i` will execute once for each `i` from 0 to $n-2$.
- The inner loop will run once for each time from `j` ranging from $i+1$ to $n-1$.
- Thus, letting $C$ be the number of constant time operations, the runtime of the above is

$$\begin{aligned}\sum_{i=0}^{n-2}\sum_{j=i+1}^{n} C &= C\sum_{i=0}^{n-2}(n-i)\\ &= Cn^2 - C\sum_{i=0}^{n-2} i\\ &= Cn^2 - C\frac{n(n-1)}{2}\\ &= O(n^2)\end{aligned}$$

# Important Sums

- $\sum_{i=1}^{n} i = O(n^2)$
- $\sum_{i=1}^{n} i^2 = O(n^3)$
- $\sum_{i=1}^{n} i^k = O(n^{k+1})$ for any $k \in \mathbb{N}$.
- $\sum_{i=1}^{n} n = O(n^2)$

# First Improvement

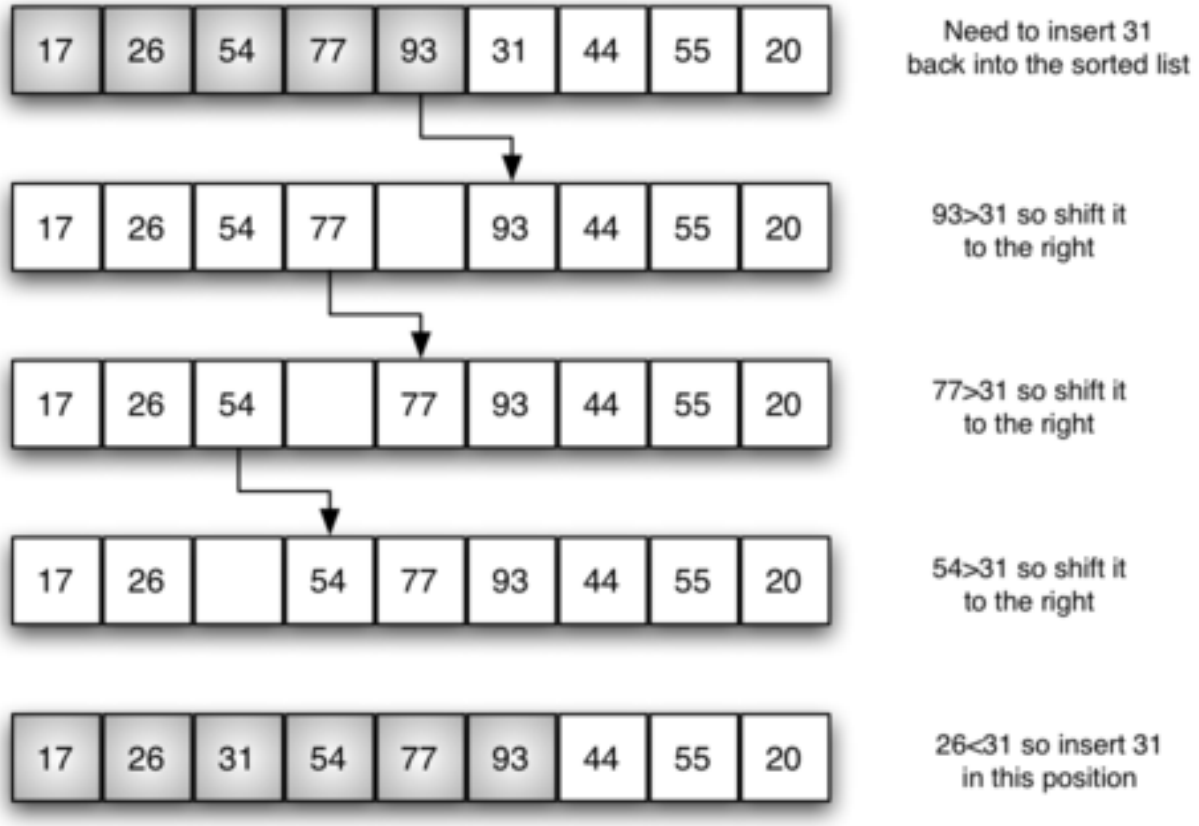
- Can we do better than the previous algorithm?
- Notice one significant flaw is that even when the array is sorted, we need to go through every single element twice when really one check that the array is sorted should only take $O(n)$ time.
- Can we improve the algorithm in the best case so that when the array is sorted or almost sorted, the algorithm doesn't take to much time?

# Insertion Sort

The Idea: For each element $x$...

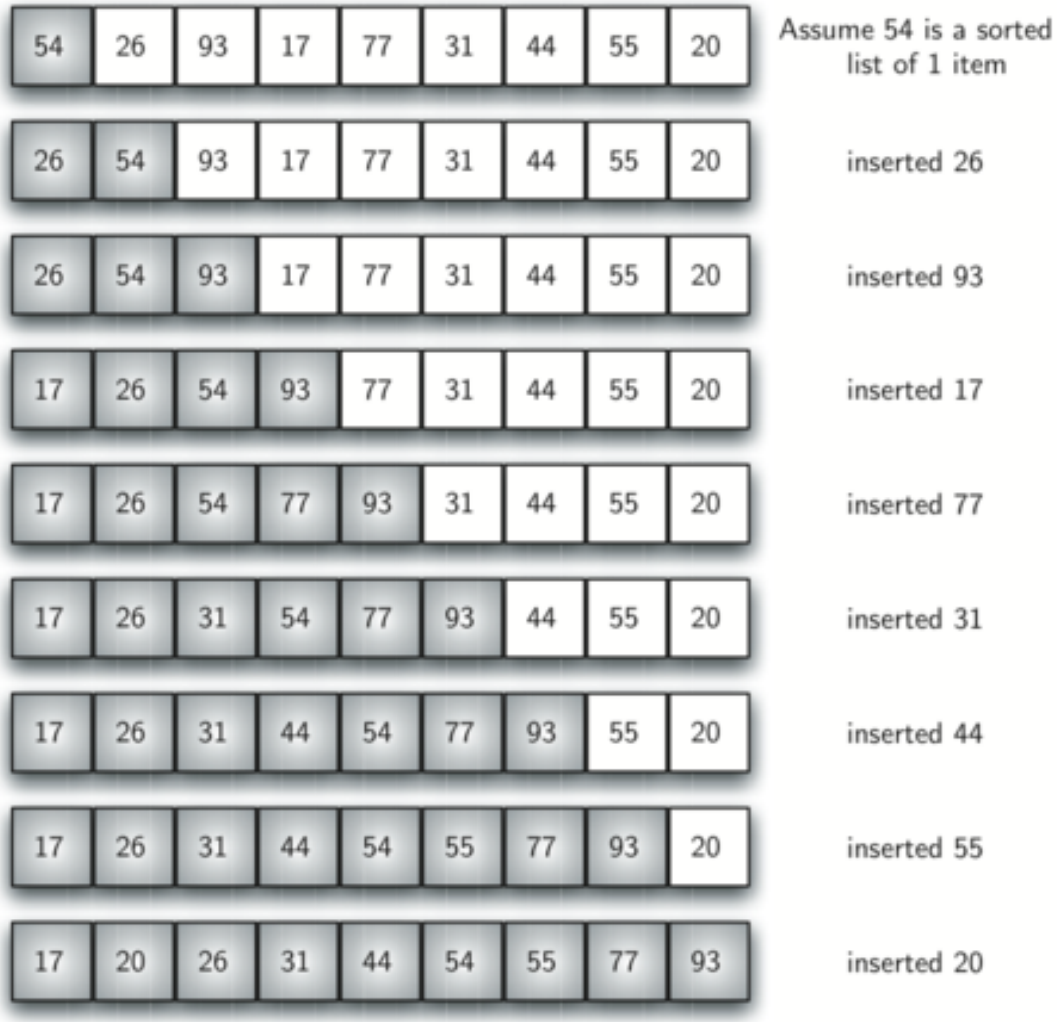
1. Find where $x$ should go
2. Shift elements $> x$
3. Insert $x$
4. Repeat

# Example Single Pass




http://interactivepython.org/courselib/static/pythonds/_images/insertionpass.png

# Example Full




http://interactivepython.org/courselib/static/pythonds/_images/insertionsort.png

# Insertion Sort

```
#include <stdio.h>
void insertion_sort (int *a, int n) {
  int i, j, x;
  for (i = 1; i < n; i++) {
    x = a[i];
    for (j = i; j > 0 && x < a[j - 1]; j--) {
      a[j] = a[j - 1];
    }
    a[j] = x;
  }
}
```

# Testing

```
int main(void){
  int a[] = {-10,2,14,-7,11,38};
  int n = sizeof(a)/sizeof(a[0]);
  insertion_sort(a,n);
  for (int i = 0; i < n-1; i++) {
    printf("%d, ", a[i]);
  }
  printf("%d\n", a[n-1]);
  return 0;
}
```

# Runtime of Insertion Sort

- Worst Case - All of the operations in this code execute in $O(1)$ time so the only question that remains is how many times they run. The outer look will execute up to $O(n)$ time. The inner loop depends on `i`. In fact, for a **reverse sorted array**, the inner loops runs `i` times for each `i` form 1 to $n-1$. That is

$$\sum_{i=1}^{n-1} i = 1 + 2 + \ldots + n - 1 = \frac{n(n-1)}{2} = O(n^2).$$

- Best Case - If the array is sorted, then the inner loop never executes. So we reduce the runtime to $O(n)$.
- 'Average' Case - the inner loop will run about $i/2$ times. This gives

$$\sum_{i=1}^{n-1} \frac{i}{2} = \frac{1}{2} \cdot \frac{n(n-1)}{2} = O(n^2).$$

# Summary of Insertion vs Selection Sort

- Insertion sort works well when the array is almost sorted.
- Selection sort in terms of runtime is dominated by insertion sort
- Both however in the worst case are $O(n^2)$.
- The final question is "Can we do better than $O(n^2)$ in the worst case?"
- The answer is a resounding "Yes!" and we'll see this next lecture.