Lecture 1: Asymptotics, Recurrences, Elementary Sorting

Instructor: Saravanan Thirumuruganathan

Outline

1. Introduction to Asymptotic Analysis
    - Rate of growth of functions
    - Comparing and bounding functions: O, Θ, Ω
    - Specifying running time through recurrences
    - Solving recurrences
2. Elementary Sorting Algorithms
    - Bubble, Insertion and Selection sort
    - Stability of sorting algorithms

In-Class Quizzes

- **URL:** `http://m.socrative.com/`
- **Room Name: 4f2bb99e**

Analyzing Algorithms

Time Complexity:

- Quantifies amount of **time** an algorithm needs to complete as a *function* of input size

Space Complexity:

- Quantifies amount of **space** an algorithm needs to complete as a *function* of input size

Function: Input size Vs $\{$Time, Space$\}$

Analyzing Algorithms

Best Case Complexity:

- of an algorithm is the **function** that determines the **minimum** number of steps taken on any problem instance of size n

Worst Case Complexity:

- ... **maximum** ...

Average Case Complexity:

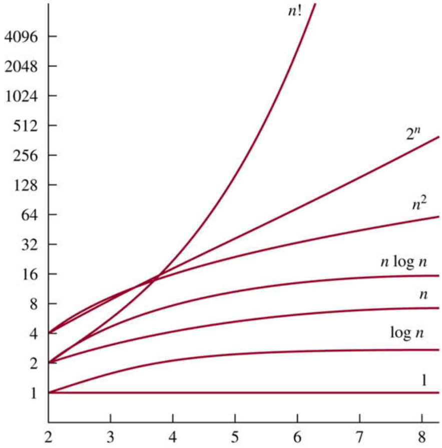
- ... **average** ...

Function: Input size Vs {Time, Space}

Growth Function $T(n)$

- Input is positive integers $n = 1, 2, 3, \ldots$
- Asymptotically positive (returns positive numbers for large n)
- How does $T(n)$ grow when n grows?
- n is size of input
- $T(n)$ is the amount of time it takes for an algorithm to solve some problem

Rate of Growth of Functions



Quiz!

Question:

- You have a machine that can do million operations per second.
- Your algorithm requires n^2 steps
- Suppose size of input is 1 million
- How long does the algorithm takes for this input?

Quiz!

Answer:

- Algorithm will take $(1M)^2$ operations
- Machine can do $1M$ operations per second
- Running time $= \frac{(1M)^2}{1M} = 1M$ seconds
- $1M$ seconds $= \frac{1M}{60*60*24} =$ Approximately 12 days

Why does it matter?

Running time of different algorithms for various input sizes

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long

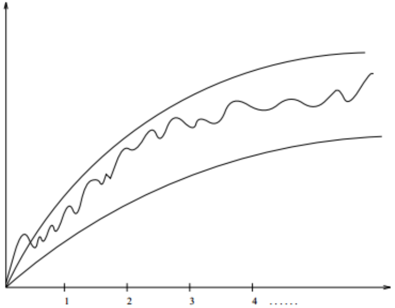
[1] Table 2.1 from K&T Algorithm Design. Very long means it takes more than 10^{25} years.

Why does it matter?

- The "Big Data" era
- Can Google/Facebook/... use it?

Functions in Real World

This is how functions look in the real world![2]



[2] Skiena Lecture notes

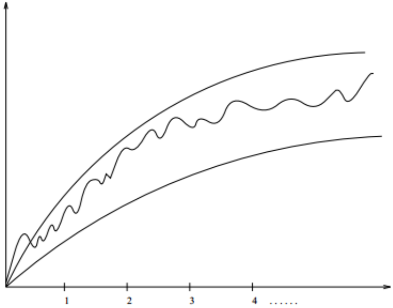
Solution: Analyze Asymptotic Behavior

- Analyze the asymptotic behavior of algorithms
- What happens to $f(n)$ when $n \to \infty$?

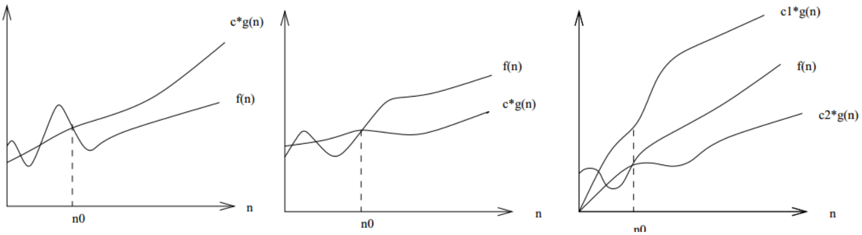
	$T(n) = 1000n$	$T(n) = n^2$
$n = 10$	$10K$	100
$n = 100$	$100K$	$10K$
$n = 1000$	$1M$	$1M$
$n = 10K$	$10M$	$100M$
$n = 100K$	$100M$	$10B$

Solution: Bound the functions

- Identify known functions (such as $n, n^2, n^3, 2^n, \ldots$) that can "bound" $T(n)$
- How to bound? - asymptotic **upper, lower** and **tight** bounds
- Find a function $f(n)$ such that $T(n)$ is **proportional** to $f(n)$
- Why proportional (as against **equal**)?
- Ignore aspects such as programming language, programmer capability, compiler optimization, machine specification etc



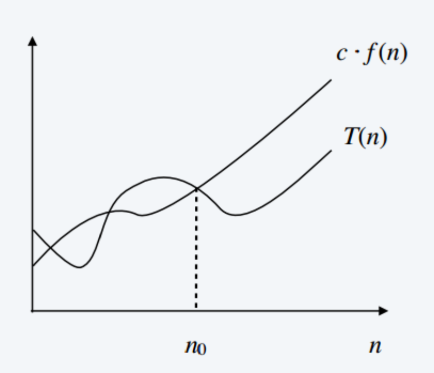
O, Ω, Θ[3]



[3] CLRS book

Big-O Notation

Upper bounds: $T(n)$ is $O(f(n))$ if there exists **constants** $c > 0$ and $n_0 \geq 0$ such that $T(n) \leq c \cdot f(n)$ for all $n \geq n_0$



[4]

[4] From K&T: Algorithm Design

Big-O Notation

Upper bounds: $T(n)$ is $O(f(n))$ if there exists **constants** $c > 0$ and $n_0 \geq 0$ such that $T(n) \leq c \cdot f(n)$ for all $n \geq n_0$

Example: $T(n) = 32n^2 + 17n + 1$. Is $T(n)$ in $O(n^2)$?

- Yes! Use $c = 50$, $n_0 = 1$
- Simple Proof:

$$\begin{aligned} T(n) &\leq 32n^2 + 17n + 1 \\ &\leq 32n^2 + 17n^2 + 1n^2 \\ &\leq 50n^2 \\ &\leq cn^2 \\ c = 50 &\text{ and } n_0 = 1 \end{aligned}$$

Note: Not necessary to find the smallest c or n_0

Quiz!

Example: $T(n) = 32n^2 - 17n + 1$. Is $T(n)$ in $O(n^2)$?

Big-O Notation

Example: $T(n) = 32n^2 - 17n + 1$. Is $T(n)$ in $O(n^2)$?

- Yes! Use $c = 50$, $n_0 = 1$
- Simple Proof:

$$\begin{aligned} T(n) &\leq 32n^2 - 17n + 1 \\ &\leq 32n^2 + 17n + 1 \\ &\leq 32n^2 + 17n^2 + 1n^2 \\ &\leq 50n^2 \\ &\leq cn^2 \end{aligned}$$

$c = 50$ and $n_0 = 1$

Quiz!

Example: $T(n) = 32n^2 - 17n + 1$. Is $T(n)$ in $O(n^3)$?

Big-O Notation

Example: $T(n) = 32n^2 - 17n + 1$. Is $T(n)$ in $O(n^3)$?

- Yes! Use $c = 50$, $n_0 = 1$
- Simple Proof:

$$\begin{aligned}
T(n) &\leq 32n^2 - 17n + 1 \\
&\leq 32n^2 + 17n + 1 \\
&\leq 32n^2 + 17n^2 + 1n^2 \\
&\leq 50n^2 \\
&\leq 50n^3 \\
&\leq cn^3
\end{aligned}$$

$$c = 50 \text{ and } n_0 = 1$$

Quiz!

Example: $T(n) = 32n^2 + 17n + 1$. Is $T(n)$ in $O(n)$?

Big-O Notation

Example: $T(n) = 32n^2 + 17n + 1$. Is $T(n)$ in $O(n)$?

- No!
- Proof by contradiction

$$\begin{aligned} 32n^2 + 17n + 1 &\leq c \cdot n \\ 32n + 17 + \frac{1}{n} &\leq c \\ 32n &\leq c \text{ (ignore constants for now)} \\ n &\leq c \text{ (ignore constants for now)} \end{aligned}$$

This inequality does not hold for $n = c + 1$!

Set Theoretic Perspective

- $O(f(n))$ is set of all functions $T(n)$ where there exist positive constants c, n_0 such that $0 \leq T(n) \leq c \cdot f(n)$ for all $n \geq n_0$
- Example: $O(n^2) = \{\ n^2, \ldots, 32n^2 + 17n + 1,\ 32n^2 - 17n + 1, \ldots, n, 2n, \ldots\}$
- Notation: $T(n) = O(f(n))$ or $T(n) \in O(f(n))$

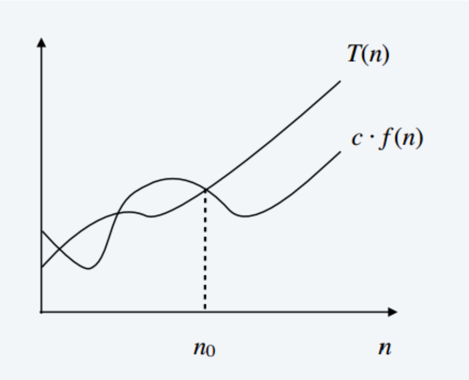
Limit based Perspective

- $T(n)$ is $O(f(n))$ if $\limsup_{n\to\infty} \frac{T(n)}{f(n)} < \infty$
- Example: $32n^2 + 17n + 1$ is $O(n^2)$

$$\begin{aligned}
\limsup_{n\to\infty} \frac{T(n)}{f(n)} &= \frac{32n^2 + 17n + 1}{n^2} \\
&= 32 + \frac{17}{n} + \frac{1}{n^2} \\
&= 32 < \infty
\end{aligned}$$

Big-Omega Notation

Lower bounds: $T(n)$ is $\Omega(f(n))$ if there exists **constants** $c > 0$ and $n_0 \geq 0$ such that $T(n) \geq c \cdot f(n)$ for all $n \geq n_0$



[5]

[5] From K&T: Algorithm Design

Big-Omega Notation

Lower bounds: $T(n)$ is $\Omega(f(n))$ if there exists **constants** $c > 0$ and $n_0 \geq 0$ such that $T(n) \geq c \cdot f(n)$ for all $n \geq n_0$

Example: $T(n) = 32n^2 + 17n + 1$. Is $T(n)$ in $\Omega(n^2)$?

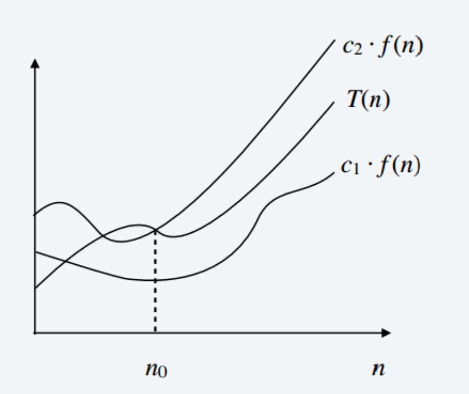
- Yes! Use $c = 32$, $n_0 = 1$
- Simple Proof:

$$\begin{aligned} T(n) &\geq 32n^2 + 17n + 1 \\ &\geq 32n^2 \\ &\geq cn^2 \end{aligned}$$

$$c = 32 \text{ and } n_0 = 1$$

Big-Theta Notation

Tight bounds: $T(n)$ is $\Theta(f(n))$ if there exists **constants** $c_1 > 0, c_2 > 0$ and $n_0 \geq 0$ such that $c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$ for all $n \geq n_0$



[6]

[6] From K&T: Algorithm Design

Big-Theta Notation

Tight bounds: $T(n)$ is $\Theta(f(n))$ if there exists **constants** $c_1 > 0, c_2 > 0$ and $n_0 \geq 0$ such that $c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$ for all $n \geq n_0$

Example: $T(n) = 32n^2 + 17n + 1$. Is $T(n)$ in $\Omega(n^2)$?

- Yes! Use $c_1 = 32$, $c_2 = 50$ and $n_0 = 1$
- Combine proofs from before

Theorem: For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Limit based Definitions

Let $\limsup_{n\to\infty} \frac{T(n)}{f(n)} = c$

- If $c < \infty$ then $T(n)$ is $O(f(n))$ (typically c is zero)
- If $c > 0$ then $T(n)$ is $\Theta(f(n))$ (also $O(f(n))$ and $\Omega(f(n))$)
- If $c = \infty$ then $T(n)$ is $\Omega(f(n))$

Some Big-O tips

- Big-O is one of the most useful things you will learn in this class!
- Big-O ignores *constant* factors through c
  - Algorithm implemented in Python might need a larger c than one implemented in C++
- Big-O ignores small inputs through n_0
  - Simply set a large value of n_0
- Suppose $T(n)$ is $O(f(n))$. Typically, $T(n)$ is messy while $f(n)$ is simple
  - $T(n) = 32n^2 + 17n + 1$, $f(n) = n^2$
- Big-O hides constant factors. Some times using an algorithm with worser Big-O might still be a good idea (e.g. sorting, finding medians)

Survey of Running Times

Complexity	Name	Example
$O(1)$	Constant time	Function that returns a constant (say 42)
$O(\log n)$	Logarithmic	Binary Search
$O(n)$	Linear	Finding Max of an array
$O(n \log n)$	Linearithmic	Sorting (for e.g. Mergesort)
$O(n^2)$	Quadratic	Selection sort
$O(n^3)$	Cubic	Floyd-Warshall
$O(n^k)$	Polynomial	Subset-sum with k elements
$O(2^n)$	Exponential	Subset-sum with no cardinality constraints

Dominance Rankings[7]

- $n! \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \log n \gg 1$
- Exponential algorithms are useless even at $n >= 50$
- Quadratic algorithms at around $n \geq 1M$
- $O(n \log n)$ at around $n \geq 1B$

[7] Skiena lecture notes

Closer Look at $T(n)$

- So far we assumed someone gave us $T(n)$
- What is n? (Program Analysis)
- How do we get $T(n)$? (Recurrences)

Program Analysis

```
for i=1 to n
{
    constant time
    operations
}
```

```
for i=1 to n
{
    for j=1 to n
    {
        constant time
        operations
    }
}
```

Recurrences

- Typically programs are lot more complex than that
- Recurrences occur in recursion and divide and conquer paradigms
- Specify running time as a function of n and running time over inputs of smaller sizes
- Examples:
    - fibonacci(n) = fibonacci($n-1$) + fibonacci($n-2$)
    - $T(n) = 2T(\frac{n}{2}) + n$
    - $T(n) = T(n-1) + n$

Solving Recurrences

- Unrolling
- Guess and prove by induction (aka Substitution)
- Recursion tree
- Master method

Unrolling

Let $T(n) = T(n-1) + n$. Base case: $T(1) = 1$

$$
\begin{aligned}
T(n) &= T(n-1) + n \\
&= n + T(n-1) \\
&= n + n - 1 + T(n-2) \\
&= n + n - 1 + n - 2 + T(n-3) \\
&= n + n - 1 + n - 2 + n - 3 + \ldots + 4 + 3 + 2 + 1 \\
&= \frac{n(n+1)}{2} \\
&= 0.5n^2 + 0.5n \\
&= O(n^2)
\end{aligned}
$$

Quiz!

Solve $T(n) = 2T(n-1)$. Base case: $T(1) = 1$

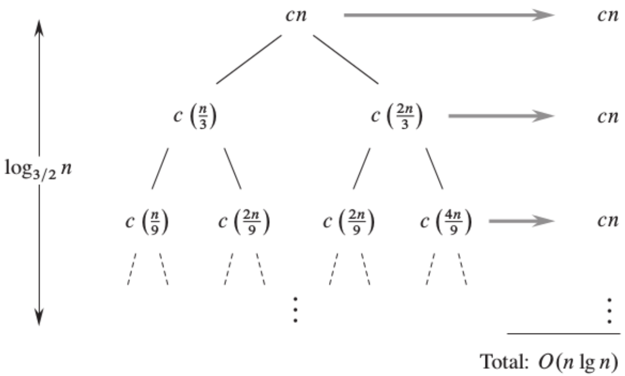
Quiz!

Solve $T(n) = 2T(n-1)$. Base case: $T(1) = 1$

$$\begin{aligned}
T(n) &= 2T(n-1) \\
&= 2(2T(n-2)) \\
&= 2(2(2T(n-3))) \\
&= 2^3 T(n-3) \\
&= 2^i \ldots 2T(n-i) \\
&= 2^n \\
&= O(2^n)
\end{aligned}$$

Recursion Tree

$$T(n) = T(\tfrac{2n}{3}) + T(\tfrac{n}{3}) + cn$$



[8]

[8] From CLRS

Logarithms[9]

- Logarithm is an *inverse* exponential function
- $b^x = n$ implies $x = \log_b n$
- If $b = 2$, logarithms reflect how many times we can double something until we get n or halve something till we get 1
- Example: $2^4 = 16$, $\log_2 16 = \lg 16 = 4$
- Example: You need $\lg 256 = 8$ bits to represent $[0, 255]$
- Identities:
  - $\log_b(xy) = \log_b(x) + \log_b(y)$
  - $\log_b a = \frac{\log_c a}{\log_c b}$
  - $\log_b b = 1$ and $\log_b 1 = 0$

[9] Skiena Lectures

Master Method

- A "black box" method to solve recurrences that occur from Divide and Conquer algorithms
- Divide, Conquer and Combine steps
- $T(n) = aT(\frac{n}{b}) + f(n)$
- Assumes that all sub-problems are of **equal** size
- a - number of sub-problems (equivalently, #recursive calls)
- b - the rate at which the problem shrinks
- Note: a and b must be constants (it cannot be for e.g. $\sqrt{n}$)
- $f(n)$ - complexity of the combine step

Master Method

Master Theorem: Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants (for e.g. they cannot be $\sqrt{n}$). Let $T(n)$ be defined on the non-negative integers by recurrence as

$$T(n) = aT(\frac{n}{b}) + n^d$$

Then $T(n)$ has the following asymptotic bounds:

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{log_b a}) & \text{if } a > b^d \end{cases}$$

Master Method - Examples [10]

$T(n) = 2T(\frac{n}{2}) + n$

- $a = 2$, $b = 2$ and $d = 1$ (as $n = n^1$)
- $b^d = 2^1 = 2 = a$
- By case 1, $T(n) = O(n^d \log n) = O(n \log n)$

[10] From Tim Roughgarden's notes

Master Method - Examples [11]

$T(n) = 2T(\frac{n}{2}) + n^2$

- $a = 2$, $b = 2$ and $d = 2$
- $b^d = 2^2 = 4 > a$
- By case 2, $T(n) = O(n^d) = O(n^2)$

[11] From Tim Roughgarden's notes

Master Method - Examples [12]

$T(n) = 4T(\frac{n}{2}) + n$

- $a = 4$, $b = 2$ and $d = 1$ (as $n = n^1$)
- $b^d = 2^1 = 2 < a$
- By case 3, $T(n) = O(n^{\log_b a}) = O(n^{\log_2 4}) = O(n^2)$

[12] From Tim Roughgarden's notes

Quiz! [13]

Solve $T(n) = 8T(\frac{n}{2}) + 1000n^2$. Let's solve it step by step!

[13] From Wikipedia

Quiz![14]

Solve $T(n) = 8T(\frac{n}{2}) + 1000n^2$

- $a = 8,\ b = 2$ and $d = 2$
- $b^d = 2^2 < a$
- Falls in Case 3 of Master Theorem
- $O(n^{\log_b a}) = O(n^{\log_2 8}) = O(n^3)$

[14] From Wikipedia

Sorting Problem

- **Input:** A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
- **Output:** A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$
- **Example:** $\langle 4, 2, 1, 3, 5 \rangle$ to $\langle 1, 2, 3, 4, 5 \rangle$
- Assume distinct values (doesn't affect correctness or analysis)

Applications of Sorting[15]

1. Direct applications
   - Sorting a list of names in dictionary
   - Sorting search results based on Google's ranking algorithm
2. Problems made simpler after sorting
   - Finding median, frequency distribution
   - Finding duplicates
   - Binary search
   - Closest pair of points
3. Non-obvious applications
   - Data compression (e.g. Huffman encoding)
   - Computer Graphics (e.g Convex hulls)
   - Many many more !

[15] From Slides of Kevin Wayne

Sorting Algorithms

- Comparison based sorting
  - Time complexity measured in terms of comparisons
  - Elementary algorithms: Bubble, Selection and Insertion sort
  - Mergesort and Quicksort
- Non-comparison based sorting
  - Bucket, Counting and Radix sort

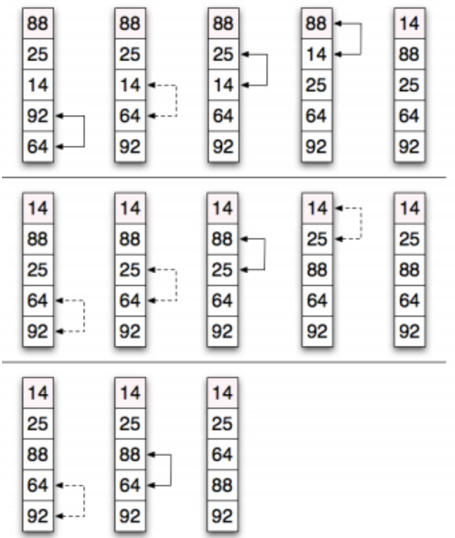
Facets of Sorting Algorithms

- Best, Average and Worst case time complexity
- Worst case space complexity
- **In-Place:** Transforms the input data structure with only constant additional space
- **Stability:** The relative order of items with same key values. E.g. $\langle 100_1, 400, 100_2, 200 \rangle \rightarrow \langle 100_1, 100_2, 200, 400 \rangle$
- **Adaptive:** Can it leverage the "sortedness" of input?

Bubble Sort

- **Basic Idea:**
  - Compare adjacent elements
  - Swap them if they are in wrong order
  - Repeat till entire array is sorted

Bubble Sort[16]



[16] http://www.cs.miami.edu/~ogihara/csc220/slides/Ch08.pdf

Bubble Sort

Pseudocode:

```
BubbleSort(A):
    for i = 1 to A.length - 1
        for j = A.length downto i+1
            if A[j] < A[j-1]
                swap(A[j-1], A[j])
```

Loop Invariant: First $i - 1$ elements are in sorted order

Better Implementation: Count swaps within an iteration. Terminate if no swaps.

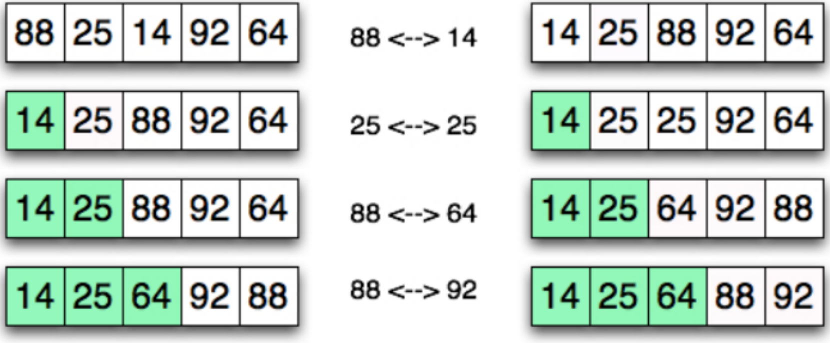
Bubble Sort Properties:

- Best case time complexity: $O(n)$
- Worst case time complexity: $O(n^2)$
- Adaptive: Yes
- In-Place: Yes
- Stability: Yes

Selection Sort

- **Basic Idea:**
  - Find smallest element and exchange it with A[1]
  - Find second smallest element and exchange it with A[2]
  - Repeat process till entire array is sorted

Selection Sort[17]



[17] http://www.cs.miami.edu/~ogihara/csc220/slides/Ch08.pdf

Selection Sort

Pseudocode:

```
SelectionSort(A):
    for i = 1 to A.length
        k = i
        for j = i+1 to A.length
            if A[j] < A[k]
                k = j
        swap(A[i], A[k])
```

Loop Invariant: First $i - 1$ elements are in sorted order

Quiz!

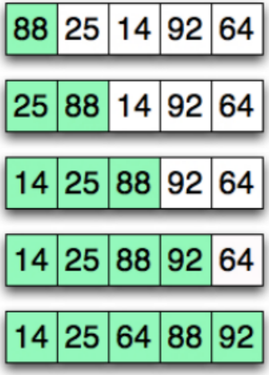
Selection Sort Properties:

- Best case time complexity: $O(n^2)$
- Worst case time complexity: $O(n^2)$
- Adaptive: No
- In-Place: Yes
- Stability: No (but can be made to one with some effort)

Insertion Sort

- Best of the elementary sorting algorithms
- **Basic Idea:**
  - Start with an empty sorted array
  - Pick an element and insert into the right place
  - Repeat till all elements are handled

Insertion Sort[18]



[18] http://www.cs.miami.edu/~ogihara/csc220/slides/Ch08.pdf

Pseudocode:

```
InsertionSort(A):
    for i = 2 to A.length
        key = A[i]
        j = i - 1
        while j > 0 and A[j] > key
            A[j+1] = A[j]
            j = j - 1
        A[j+1] = key
```

Loop Invariant: At start of i-th iteration, subarray A[1..i-1] consists of elements originally in A[1..i-1] but in sorted order

Quiz!

Insertion Sort Properties:

- Best case time complexity: $O(n)$
- Worst case time complexity: $O(n^2)$
- Adaptive: Yes
- In-Place: Yes
- Stability: Yes

Quiz!

Suppose you have an array with identical elements. Which sort would take the least time?

Quiz!

Suppose you have an array with identical elements. Which sort would take the least time?

- Insertion sort
- Bubble sort

Quiz!

Suppose you have an array that is k-sorted - i.e. each element is at most k away from its target position. For example, $\langle 1, 3, 0, 2 \rangle$ is a $2-$sorted array.

Quiz!

Suppose you have an array that is k-sorted - i.e. each element is at most k away from its target position. For example, $\langle 1, 3, 0, 2 \rangle$ is a $2-$sorted array.

- Insertion sort

Summary

Major Topics:

- Asymptotics, O, Ω, Θ, Recurrences, Master method
- Sorting, facets of Sorting, elementary Sorting algorithms