CSEN 703 - Analysis and Design of Algorithms

Lecture 1 - Introduction and Analyzing Algorithms

Dr. Nourhan Ehab
nourhan.ehab@guc.edu.eg

Department of Computer Science and Engineering
Faculty of Media Engineering and Technology

Outline

1. Administrivia
2. Course Overview
3. Analyzing Algorithms
4. Recap

Course Staff

GUC German University in Cairo

- Lecturer: Dr. Nourhan Ehab
  - Email: nourhan.ehab@guc.edu.eg
  - Office: C7.309
- TA:
  - Eng. Yasmeen Khaled
    - Email: yasmeen.khaled@guc.edu.eg
    - Office: C6.305 (inside)
  - Eng. Salma Kishk
    - Email: salma.osama-kishk@guc.edu.eg
    - Office: C3.205
  - Eng. Farida Helmy
    - Email: farida.el-dessouky@guc.edu.eg
    - Office: C3.205

Grading Scheme and Communication Channels

Quizzes (2/3)	20%
Programming Assignments (2/2)	15%
Midterm	25%
Final	40%

Grading Scheme and Communication Channels

Quizzes (2/3)	20%
Programming Assignments (2/2)	15%
Midterm	25%
Final	40%

- Course Material:
  On the CMS
- Piazza Course Page:
  https://piazza.com/guc.edu.eg/other/csen703

Quizzes Policy

- Short quizzes (30 minutes).
- Timings, Content, and Location will be announced one week ahead.

Programming Assignments Policy

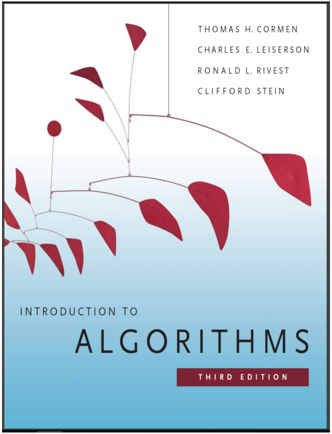
- Three programming assignments in Java.
- Assignments grades will be based primarily on public and private test cases.
- Public test cases will be posted before submission.

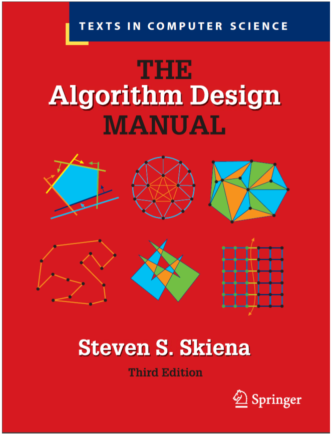
Some Ground Rules

GUC German University in Cairo

- Plagiarism is not tolerated. You are not allowed to copy from one another or from online sources. You must submit your own work.
- Cross attendance in tutorials is not allowed.
- You need to send us an email beforehand if you will come to office hours.

Textbooks





Lecture slides will include the relevant reading material.

Due Credits

The presented material is based on:

1. Previous editions of the course at the GUC due to Dr. Wael Aboulsaadat, Dr. Haythem Ismail, Dr. Amr Desouky, and Dr. Carmen Gervet.
2. Stony Brook University's Analysis of Algorithms Course.
3. MIT's Introduction to Algorithms Course.
4. Stanford's Design and Analysis of Algorithms Course.

Outline

1. Administrivia
2. Course Overview
3. Analyzing Algorithms
4. Recap

What Do You Expect We Will Do?

What Do You Expect We Will Do?

What is an algorithm?

An algorithm is a sequence of steps to transform an input into an output.

What Do You Expect We Will Do?

What is an algorithm?

An algorithm is a sequence of steps to transform an input into an output.

We seek to design algorithms that are correct, efficient, and easy to implement.

What Do You Expect We Will Do?

GUC German University in Cairo

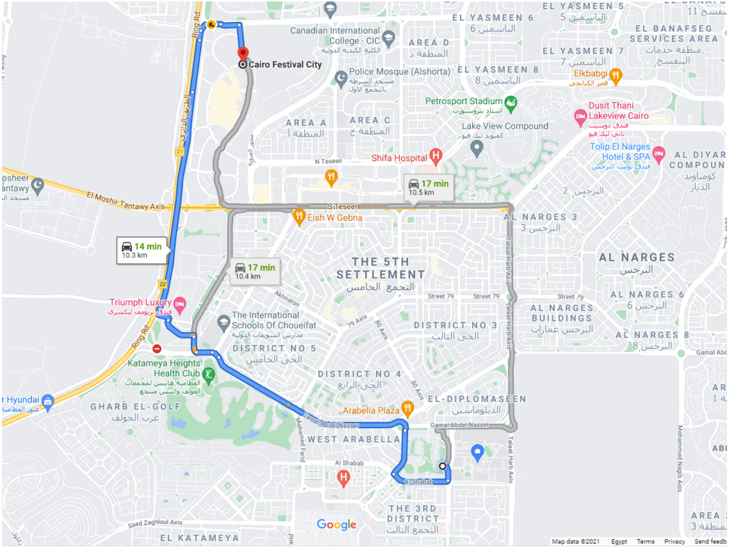
What is an algorithm?

An algorithm is a sequence of steps to transform an input into an output.

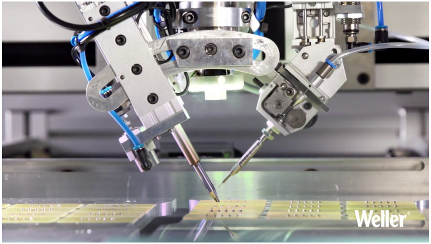
We seek to design algorithms that are correct, efficient, and easy to implement.

What do we mean by efficient?

Example: Google Maps



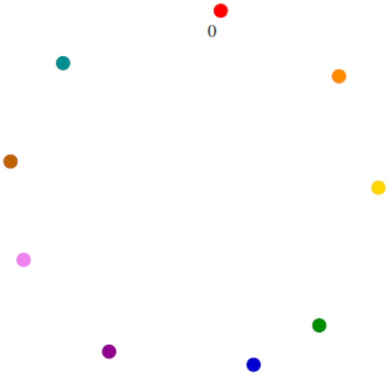
Another Example: Robot Tour Optimization



Example

Suppose you have a robot arm equipped with a tool, say a soldering iron. To enable the robot arm to do a soldering job, we must construct an ordering of the contact points, so the robot visits (and solders) the points in some order. We seek the order which minimizes the testing time (i.e. travel distance) it takes to assemble the circuit board.

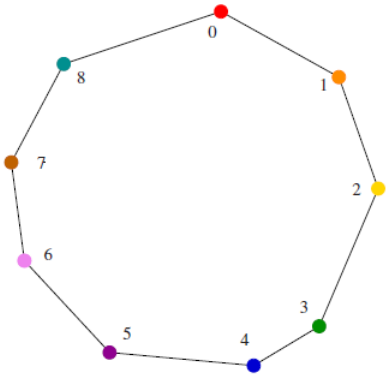
Robot Tour Optimization



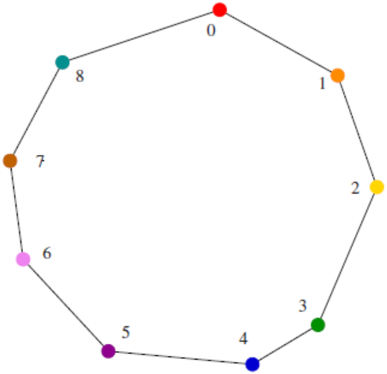
Proposed Solution 1: Nearest Neighbour

```
NearestNeighbor(P)
Pick and visit an initial point p_0 from P ;
p = p_0 ;
i = 0 ;
while there are still unvisited points do
    i = i + 1 ;
    Let p_i be the closest unvisited point in P to p_{i-1} ;
    Visit p_i ;
end
return distance to p_0 from p_{n-1};
```

Proposed Solution 1: Nearest Neighbour

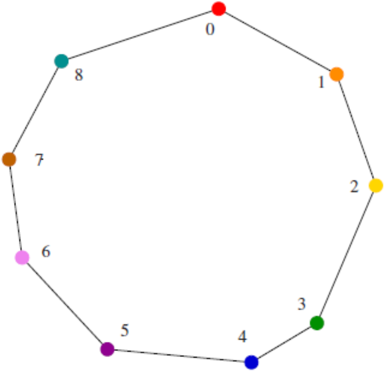


Proposed Solution 1: Nearest Neighbour



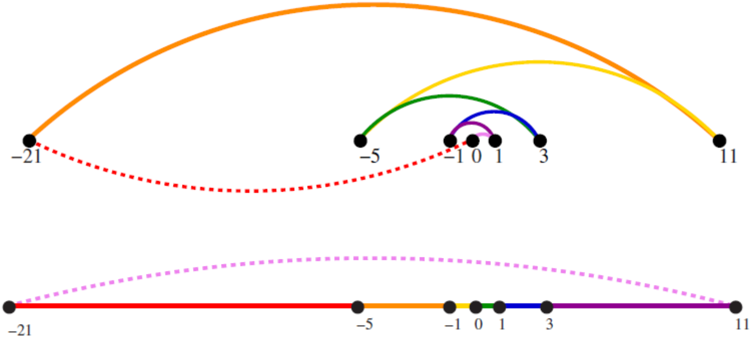
This is efficient and easy to implement.

Proposed Solution 1: Nearest Neighbour

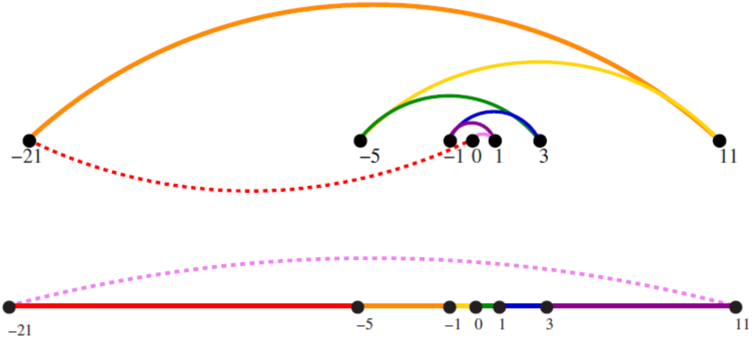


This is efficient and easy to implement. But wrong!!

Nearest Neighbour is Wrong!



Nearest Neighbour is Wrong!



Starting from the leftmost point will not fix the problem.

Proposed Solution 2: Brute Force

```
BruteForce(P)
d = ∞ ;
for each permutation P_i of the n! permutations of P do
    if cost(P_i) ≤ d then
        d = cost(P_i);
        P_min = P_i;
    end
end
return P_min;
```

Proposed Solution 2: Brute Force

BruteForce(P)

$d = \infty$;

for each *permutation* P_i *of the* $n!$ *permutations of* P **do**

  if $cost(P_i) \leq d$ **then**

    $d = cost(P_i)$;

    $P_{min} = P_i$;

  end

end

return P_{min};

This is correct.

Proposed Solution 2: Brute Force

BruteForce(P)
$d = \infty$;
for each *permutation* P_i *of the* $n!$ *permutations of* P **do**
    if $cost(P_i) \leq d$ **then**
        $d = cost(P_i)$;
        $P_{min} = P_i$;
    end
end
return P_{min};

This is correct. But not efficient!

Correctness and Efficiency

Key Takeaway

Correctness and efficiency can not always be simultaneously accomplished.

Correctness and Efficiency

Key Takeaway

Correctness and efficiency can not always be simultaneously accomplished. We need to have a way to systemically study algorithms.

Applications of Algorithms

- Search Engines.
- Streaming Services.
- Recommendation Systems.
- Cryptography.
- E-commerce and E-banking.
- Scheduling and Resource Allocation.
- Artificial Intelligence.
- Among many others.

Reasons Why You Should Study This Course!

GUC German University in Cairo

1. Algorithms are the bread-and-butter of computer science (and its applications).

Reasons Why You Should Study This Course!

GUC German University in Cairo

1. Algorithms are the bread-and-butter of computer science (and its applications).
2. It teaches you how to reason about algorithms' correctness and determine their efficiency.

Reasons Why You Should Study This Course!

GUC German University in Cairo

1. Algorithms are the bread-and-butter of computer science (and its applications).
2. It teaches you how to reason about algorithms' correctness and determine their efficiency.
3. It teaches you clever algorithm design techniques with fancy names like Divide and Conquer, Greedy algorithms, and Dynamic Programming.

Reasons Why You Should Study This Course!

GUC German University in Cairo

1. Algorithms are the bread-and-butter of computer science (and its applications).
2. It teaches you how to reason about algorithms' correctness and determine their efficiency.
3. It teaches you clever algorithm design techniques with fancy names like Divide and Conquer, Greedy algorithms, and Dynamic Programming.
4. It exposes you to algorithms used in diverse fields.

Reasons Why You Should Study This Course!

GUC German University in Cairo

1. Algorithms are the bread-and-butter of computer science (and its applications).
2. It teaches you how to reason about algorithms' correctness and determine their efficiency.
3. It teaches you clever algorithm design techniques with fancy names like Divide and Conquer, Greedy algorithms, and Dynamic Programming.
4. It exposes you to algorithms used in diverse fields.
5. The course covers a wide variety of common interview questions.

Reasons Why You Should Study This Course!

1. Algorithms are the bread-and-butter of computer science (and its applications).
2. It teaches you how to reason about algorithms' correctness and determine their efficiency.
3. It teaches you clever algorithm design techniques with fancy names like Divide and Conquer, Greedy algorithms, and Dynamic Programming.
4. It exposes you to algorithms used in diverse fields.
5. The course covers a wide variety of common interview questions.
6. It is fun :)

Course Plan

Week	Content
1	Introduction
2	Off
3	Asymptotic Analysis
4,5	Divide and Conquer Algorithms
6	Amortized Analysis
7	Greedy Algorithms
8,9	Dynamic Programming
10,11,12	Graph Algorithms

Outline

Reasoning about (In)Correctness

Reasoning about (In)Correctness

- To show that an algorithm is incorrect, we just need to find a counter example.

Reasoning about (In)Correctness

- To show that an algorithm is incorrect, we just need to find a counter example.
- Think small, exhaustively, and consider edge cases.

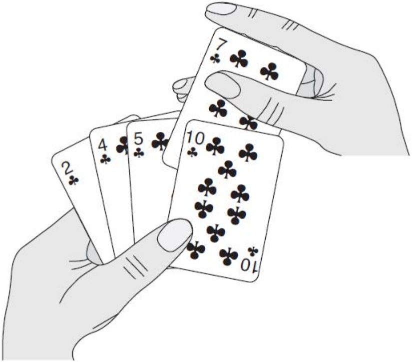
Reasoning about (In)Correctness

- To show that an algorithm is incorrect, we just need to find a counter example.
- Think small, exhaustively, and consider edge cases.
- Failing to find a counter example does not mean that an algorithm is correct!

Reasoning about (In)Correctness

- To show that an algorithm is incorrect, we just need to find a counter example.
- Think small, exhaustively, and consider edge cases.
- Failing to find a counter example does not mean that an algorithm is correct!
- Proving that an algorithm is correct requires a formal proof.

Insertion Sort



Given a sequence of cards in one hand, produce a sorted sequence of the same cards according to their value.

Insertion Sort - Intuition

The Insertion Sort Algorithm (https://visualgo.net/en/sorting):

- Input: Start with an empty left hand, cards facing down on the table.
- Incremental Steps:
  1. Remove one card at a time from the table.
  2. Insert it into the right position in the left hand by comparing it with each of the cards in the hand from right to left.
  3. At all time, the cards held in the left hand are sorted.
- Output: Cards sorted in the left hand.

Insertion Sort - Pseudo Code

```
Insertion Sort(A)
for j = 2 to n do
    key = A[j] ;
    i = j − 1 ;
    for i > 0 and A[i] > key do
        A[i + 1] = A[i] ;
        i = i − 1 ;
    end
    A[i + 1] = key ;
end
return A;
```

Trace this on $A = [5, 2, 4, 6, 1, 3]$.

Insertion Sort - Pseudo Code

GUC German University in Cairo

```
Insertion Sort(A)
for j = 2 to n do
    key = A[j] ;
    i = j − 1 ;
    for i > 0 and A[i] > key do
        A[i + 1] = A[i] ;
        i = i − 1 ;
    end
    A[i + 1] = key ;
end
return A;
```

Trace this on $A = [5, 2, 4, 6, 1, 3]$.

Is this algorithm correct?

Invariants and Algorithms Correctness

- Invariants are often the go-to way to prove the correctness of an algorithm.

Invariants and Algorithms Correctness

- Invariants are often the go-to way to prove the correctness of an algorithm.
- An invariant is a condition that does not change if the system is working correctly.

Invariants and Algorithms Correctness

- Invariants are often the go-to way to prove the correctness of an algorithm.
- An invariant is a condition that does not change if the system is working correctly.
- A loop invariant is a condition which holds true before a loop is executed and after each subsequent iteration.

Invariants and Algorithms Correctness

- Invariants are often the go-to way to prove the correctness of an algorithm.
- An invariant is a condition that does not change if the system is working correctly.
- A loop invariant is a condition which holds true before a loop is executed and after each subsequent iteration.
- We must show three things using a loop invariant.
  1. Initialization: It is true prior to the first iteration of the loop.
  2. Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.
  3. Termination: When the loop terminates, the invariant is still correct.

Insertion Sort - Correctness

Loop Invariant: $A[1, \ldots, j-1]$ is sorted.

Insertion Sort - Correctness

Loop Invariant: $A[1, \ldots, j-1]$ is sorted.

1. Initialization: $A[1]$ is trivially sorted.

Insertion Sort - Correctness

Loop Invariant: $A[1, \ldots, j-1]$ is sorted.

1. Initialization: $A[1]$ is trivially sorted.
2. Maintenance: Elements move one position to the right until key is inserted. Hence, $A[1, \ldots, j-1]$ is always sorted.

Insertion Sort - Correctness

Loop Invariant: $A[1, \ldots, j-1]$ is sorted.

1. Initialization: $A[1]$ is trivially sorted.
2. Maintenance: Elements move one position to the right until key is inserted. Hence, $A[1, \ldots, j-1]$ is always sorted.
3. Termination: Loop terminates for $j = n + 1$. It follows from maintenance that $A[1, \ldots, n]$ is sorted.

Analyzing Algorithms

- Analyzing an algorithm has come to mean predicting the resources that the algorithm requires.

Analyzing Algorithms

- Analyzing an algorithm has come to mean predicting the resources that the algorithm requires.
- Mostly, it is computational time that we want to measure.

Analyzing Algorithms

- Analyzing an algorithm has come to mean predicting the resources that the algorithm requires.
- Mostly, it is computational time that we want to measure.
- Computational time is more appropriately measured using the number of steps an algorithm performs with respect to the input size.

Analyzing Algorithms

- Analyzing an algorithm has come to mean predicting the resources that the algorithm requires.
- Mostly, it is computational time that we want to measure.
- Computational time is more appropriately measured using the number of steps an algorithm performs with respect to the input size.
- We shall assume a generic one processor, random-access machine (RAM) model of computation as our implementation technology.

Analyzing Algorithms

- Analyzing an algorithm has come to mean predicting the resources that the algorithm requires.
- Mostly, it is computational time that we want to measure.
- Computational time is more appropriately measured using the number of steps an algorithm performs with respect to the input size.
- We shall assume a generic one processor, random-access machine (RAM) model of computation as our implementation technology.
- In the RAM model, instructions are executed one after another, with no concurrent operations. Each instruction has a constant cost.

Runtime Analysis

- Best Case: The input structure/size that helps reduce the number of steps to a minimum.
- Worst Case: The input structure/size increases the number of steps to a maximum.
- Average Case: Based on average input. Often roughly as bad as the worst case.

Insertion Sort - Best Case Analysis

Best Case:

Insertion Sort - Best Case Analysis

GUC German University in Cairo

Best Case: The input is already sorted.

Insertion Sort - Best Case Analysis

Best Case: The input is already sorted.

```
1  Insertion Sort(A)
2  for j = 2 to n do
3  |   key = A[j] ;
4  |   i = j − 1 ;
5  |   for i > 0 and A[i] > key do
6  |   |   A[i + 1] = A[i] ;
7  |   |   i = i − 1 ;
8  |   end
9  |   A[i + 1] = key ;
10 end
11 return A;
```

Line	Cost	Times
2	c_1	
3	c_2	
4	c_3	
5	c_4	
6	c_5	
7	c_6	
9	c_7	
11	c_8	

Insertion Sort - Best Case Analysis

Best Case: The input is already sorted.

```
1  Insertion Sort(A)
2  for j = 2 to n do
3  |   key = A[j] ;
4  |   i = j − 1 ;
5  |   for i > 0 and A[i] > key do
6  |   |   A[i + 1] = A[i] ;
7  |   |   i = i − 1 ;
8  |   end
9  |   A[i + 1] = key ;
10 end
11 return A;
```

Line	Cost	Times
2	c_1	n
3	c_2	
4	c_3	
5	c_4	
6	c_5	
7	c_6	
9	c_7	
11	c_8	

Insertion Sort - Best Case Analysis

Best Case: The input is already sorted.

```
1  Insertion Sort(A)
2  for j = 2 to n do
3  |   key = A[j] ;
4  |   i = j − 1 ;
5  |   for i > 0 and A[i] > key do
6  |   |   A[i + 1] = A[i] ;
7  |   |   i = i − 1 ;
8  |   end
9  |   A[i + 1] = key ;
10 end
11 return A;
```

Line	Cost	Times
2	c_1	n
3	c_2	$n-1$
4	c_3	
5	c_4	
6	c_5	
7	c_6	
9	c_7	
11	c_8	

Insertion Sort - Best Case Analysis

Best Case: The input is already sorted.

```
1  Insertion Sort(A)
2  for j = 2 to n do
3      key = A[j] ;
4      i = j − 1 ;
5      for i > 0 and A[i] > key do
6          A[i + 1] = A[i] ;
7          i = i − 1 ;
8      end
9      A[i + 1] = key ;
10 end
11 return A;
```

Line	Cost	Times
2	c_1	n
3	c_2	$n-1$
4	c_3	$n-1$
5	c_4	
6	c_5	
7	c_6	
9	c_7	
11	c_8	

Insertion Sort - Best Case Analysis

Best Case: The input is already sorted.

```
1  Insertion Sort(A)
2  for j = 2 to n do
3      key = A[j] ;
4      i = j − 1 ;
5      for i > 0 and A[i] > key do
6          A[i + 1] = A[i] ;
7          i = i − 1 ;
8      end
9      A[i + 1] = key ;
10 end
11 return A;
```

Line	Cost	Times
2	c_1	n
3	c_2	$n-1$
4	c_3	$n-1$
5	c_4	$n-1$
6	c_5	
7	c_6	
9	c_7	
11	c_8	

Insertion Sort - Best Case Analysis

Best Case: The input is already sorted.

```
1  Insertion Sort(A)
2  for j = 2 to n do
3      key = A[j] ;
4      i = j − 1 ;
5      for i > 0 and A[i] > key do
6          A[i + 1] = A[i] ;
7          i = i − 1 ;
8      end
9      A[i + 1] = key ;
10 end
11 return A;
```

Line	Cost	Times
2	c_1	n
3	c_2	$n-1$
4	c_3	$n-1$
5	c_4	$n-1$
6	c_5	0
7	c_6	
9	c_7	
11	c_8	

Insertion Sort - Best Case Analysis

Best Case: The input is already sorted.

```
1  Insertion Sort(A)
2  for j = 2 to n do
3      key = A[j] ;
4      i = j − 1 ;
5      for i > 0 and A[i] > key do
6          A[i + 1] = A[i] ;
7          i = i − 1 ;
8      end
9      A[i + 1] = key ;
10 end
11 return A;
```

Line	Cost	Times
2	c_1	n
3	c_2	$n-1$
4	c_3	$n-1$
5	c_4	$n-1$
6	c_5	0
7	c_6	0
9	c_7	
11	c_8	

Insertion Sort - Best Case Analysis

Best Case: The input is already sorted.

```
1  Insertion Sort(A)
2  for j = 2 to n do
3  |   key = A[j] ;
4  |   i = j - 1 ;
5  |   for i > 0 and A[i] > key do
6  |   |   A[i + 1] = A[i] ;
7  |   |   i = i - 1 ;
8  |   end
9  |   A[i + 1] = key ;
10 end
11 return A;
```

Line	Cost	Times
2	c_1	n
3	c_2	$n-1$
4	c_3	$n-1$
5	c_4	$n-1$
6	c_5	0
7	c_6	0
9	c_7	$n-1$
11	c_8	

Insertion Sort - Best Case Analysis

Best Case: The input is already sorted.

```
1  Insertion Sort(A)
2  for j = 2 to n do
3      key = A[j] ;
4      i = j − 1 ;
5      for i > 0 and A[i] > key do
6          A[i + 1] = A[i] ;
7          i = i − 1 ;
8      end
9      A[i + 1] = key ;
10 end
11 return A;
```

Line	Cost	Times
2	c_1	n
3	c_2	$n-1$
4	c_3	$n-1$
5	c_4	$n-1$
6	c_5	0
7	c_6	0
9	c_7	$n-1$
11	c_8	1

Insertion Sort - Best Case Analysis

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5(0) \\
&\quad + c_6(0) + c_7(n-1) + c_8 \\
&= (c_1 + c_2 + c_3 + c_4 + c_7)n + (-c_2 - c_3 - c_4 - c_7 + c_8)(1)
\end{aligned}
$$

Insertion Sort - Best Case Analysis

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5(0) \\
&\quad + c_6(0) + c_7(n-1) + c_8 \\
&= (c_1 + c_2 + c_3 + c_4 + c_7)n + (-c_2 - c_3 - c_4 - c_7 + c_8)(1) \\
&= an + b
\end{aligned}
$$

Insertion Sort - Best Case Analysis

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5(0) \\
&\quad + c_6(0) + c_7(n-1) + c_8 \\
&= (c_1 + c_2 + c_3 + c_4 + c_7)n + (-c_2 - c_3 - c_4 - c_7 + c_8)(1) \\
&= an + b \Rightarrow \text{Linear function}
\end{aligned}
$$

Insertion Sort - Worst Case Analysis

Worst Case:

Insertion Sort - Worst Case Analysis

Worst Case: The input is reversely sorted.

Insertion Sort - Worst Case Analysis

Worst Case: The input is reversely sorted.

```
1  Insertion Sort(A)
2  for j = 2 to n do
3      key = A[j] ;
4      i = j − 1 ;
5      for i > 0 and A[i] > key
        do
6          A[i + 1] = A[i] ;
7          i = i − 1 ;
8      end
9      A[i + 1] = key ;
10 end
11 return A;
```

Line	Cost	Times
2	c_1	
3	c_2	
4	c_3	
5	c_4	
6	c_5	
7	c_6	
9	c_7	
11	c_8	

Insertion Sort - Worst Case Analysis

Worst Case: The input is reversely sorted.

```
1  Insertion Sort(A)
2  for j = 2 to n do
3      key = A[j] ;
4      i = j − 1 ;
5      for i > 0 and A[i] > key
        do
6          A[i + 1] = A[i] ;
7          i = i − 1 ;
8      end
9      A[i + 1] = key ;
10 end
11 return A;
```

Line	Cost	Times
2	c_1	n
3	c_2	
4	c_3	
5	c_4	
6	c_5	
7	c_6	
9	c_7	
11	c_8	

Insertion Sort - Worst Case Analysis

Worst Case: The input is reversely sorted.

```
1  Insertion Sort(A)
2  for j = 2 to n do
3      key = A[j] ;
4      i = j − 1 ;
5      for i > 0 and A[i] > key
        do
6          A[i + 1] = A[i] ;
7          i = i − 1 ;
8      end
9      A[i + 1] = key ;
10 end
11 return A;
```

Line	Cost	Times
2	c_1	n
3	c_2	$n-1$
4	c_3	
5	c_4	
6	c_5	
7	c_6	
9	c_7	
11	c_8	

Insertion Sort - Worst Case Analysis

Worst Case: The input is reversely sorted.

```
1  Insertion Sort(A)
2  for j = 2 to n do
3      key = A[j] ;
4      i = j − 1 ;
5      for i > 0 and A[i] > key
         do
6          A[i + 1] = A[i] ;
7          i = i − 1 ;
8      end
9      A[i + 1] = key ;
10 end
11 return A;
```

Line	Cost	Times
2	c_1	n
3	c_2	$n-1$
4	c_3	$n-1$
5	c_4	
6	c_5	
7	c_6	
9	c_7	
11	c_8	

Insertion Sort - Worst Case Analysis

Worst Case: The input is reversely sorted.

```
1  Insertion Sort(A)
2  for j = 2 to n do
3      key = A[j] ;
4      i = j - 1 ;
5      for i > 0 and A[i] > key
        do
6          A[i + 1] = A[i] ;
7          i = i - 1 ;
8      end
9      A[i + 1] = key ;
10 end
11 return A;
```

Line	Cost	Times
2	c_1	n
3	c_2	$n-1$
4	c_3	$n-1$
5	c_4	$\Sigma_{j=2}^{n} j$
6	c_5	
7	c_6	
9	c_7	
11	c_8	

Insertion Sort - Worst Case Analysis

Worst Case: The input is reversely sorted.

```
1  Insertion Sort(A)
2  for j = 2 to n do
3      key = A[j] ;
4      i = j − 1 ;
5      for i > 0 and A[i] > key
         do
6          A[i + 1] = A[i] ;
7          i = i − 1 ;
8      end
9      A[i + 1] = key ;
10 end
11 return A;
```

Line	Cost	Times
2	c_1	n
3	c_2	$n-1$
4	c_3	$n-1$
5	c_4	$\Sigma_{j=2}^{n} j$
6	c_5	$\Sigma_{j=2}^{n} j-1$
7	c_6	
9	c_7	
11	c_8	

Insertion Sort - Worst Case Analysis

Worst Case: The input is reversely sorted.

```
1  Insertion Sort(A)
2  for j = 2 to n do
3      key = A[j] ;
4      i = j − 1 ;
5      for i > 0 and A[i] > key
        do
6          A[i + 1] = A[i] ;
7          i = i − 1 ;
8      end
9      A[i + 1] = key ;
10 end
11 return A;
```

Line	Cost	Times
2	c_1	n
3	c_2	$n-1$
4	c_3	$n-1$
5	c_4	$\Sigma_{j=2}^{n} j$
6	c_5	$\Sigma_{j=2}^{n} j-1$
7	c_6	$\Sigma_{j=2}^{n} j-1$
9	c_7	
11	c_8	

Insertion Sort - Worst Case Analysis

Worst Case: The input is reversely sorted.

```
1  Insertion Sort(A)
2  for j = 2 to n do
3      key = A[j] ;
4      i = j − 1 ;
5      for i > 0 and A[i] > key
       do
6          A[i + 1] = A[i] ;
7          i = i − 1 ;
8      end
9      A[i + 1] = key ;
10 end
11 return A;
```

Line	Cost	Times
2	c_1	n
3	c_2	$n-1$
4	c_3	$n-1$
5	c_4	$\sum_{j=2}^{n} j$
6	c_5	$\sum_{j=2}^{n} j-1$
7	c_6	$\sum_{j=2}^{n} j-1$
9	c_7	$n-1$
11	c_8	

Insertion Sort - Worst Case Analysis

Worst Case: The input is reversely sorted.

```
1  Insertion Sort(A)
2  for j = 2 to n do
3  |   key = A[j] ;
4  |   i = j − 1 ;
5  |   for i > 0 and A[i] > key
   |    do
6  |   |   A[i + 1] = A[i] ;
7  |   |   i = i − 1 ;
8  |   end
9  |   A[i + 1] = key ;
10 end
11 return A;
```

Line	Cost	Times
2	c_1	n
3	c_2	$n-1$
4	c_3	$n-1$
5	c_4	$\Sigma_{j=2}^{n} j$
6	c_5	$\Sigma_{j=2}^{n} j-1$
7	c_6	$\Sigma_{j=2}^{n} j-1$
9	c_7	$n-1$
11	c_8	1

Insertion Sort - Worst Case Analysis

$$
\begin{aligned}
T(n) \quad &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(\tfrac{1}{2}n^2 + \tfrac{1}{2}n - 1) \\
&+ c_5(\tfrac{1}{2}n^2 - \tfrac{1}{2}n) + c_6(\tfrac{1}{2}n^2 - \tfrac{1}{2}n) + c_7(n-1) + c_8 \\
\\
&= (\tfrac{1}{2}c_4 + \tfrac{1}{2}c_5 + \tfrac{1}{2}c_6)n^2 \\
&+ (c_1 + c_2 + c_3 + \tfrac{1}{2}c_4 - \tfrac{1}{2}c_5 - \tfrac{1}{2}c_6 + c_7)n \\
&+ (-c_2 - c_3 - c_4 - c_7 + c_8)(1)
\end{aligned}
$$

Insertion Sort - Worst Case Analysis

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(\tfrac{1}{2}n^2 + \tfrac{1}{2}n - 1) \\
&\quad + c_5(\tfrac{1}{2}n^2 - \tfrac{1}{2}n) + c_6(\tfrac{1}{2}n^2 - \tfrac{1}{2}n) + c_7(n-1) + c_8 \\
&= (\tfrac{1}{2}c_4 + \tfrac{1}{2}c_5 + \tfrac{1}{2}c_6)n^2 \\
&\quad + (c_1 + c_2 + c_3 + \tfrac{1}{2}c_4 - \tfrac{1}{2}c_5 - \tfrac{1}{2}c_6 + c_7)n \\
&\quad + (-c_2 - c_3 - c_4 - c_7 + c_8)(1) \\
&= an^2 + bn + c
\end{aligned}
$$

Insertion Sort - Worst Case Analysis

$$
\begin{aligned}
T(n) \quad &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(\tfrac{1}{2}n^2 + \tfrac{1}{2}n - 1) \\
&+ c_5(\tfrac{1}{2}n^2 - \tfrac{1}{2}n) + c_6(\tfrac{1}{2}n^2 - \tfrac{1}{2}n) + c_7(n-1) + c_8 \\[1em]
&= (\tfrac{1}{2}c_4 + \tfrac{1}{2}c_5 + \tfrac{1}{2}c_6)n^2 \\
&+ (c_1 + c_2 + c_3 + \tfrac{1}{2}c_4 - \tfrac{1}{2}c_5 - \tfrac{1}{2}c_6 + c_7)n \\
&+ (-c_2 - c_3 - c_4 - c_7 + c_8)(1) \\[1em]
&= an^2 + bn + c \Rightarrow \text{Quadratic Function}
\end{aligned}
$$

Insertion Sort - Worst Case Analysis

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(\tfrac{1}{2}n^2 + \tfrac{1}{2}n - 1) \\
&+ c_5(\tfrac{1}{2}n^2 - \tfrac{1}{2}n) + c_6(\tfrac{1}{2}n^2 - \tfrac{1}{2}n) + c_7(n-1) + c_8 \\
&= (\tfrac{1}{2}c_4 + \tfrac{1}{2}c_5 + \tfrac{1}{2}c_6)n^2 \\
&+ (c_1 + c_2 + c_3 + \tfrac{1}{2}c_4 - \tfrac{1}{2}c_5 - \tfrac{1}{2}c_6 + c_7)n \\
&+ (-c_2 - c_3 - c_4 - c_7 + c_8)(1) \\
&= an^2 + bn + c \Rightarrow \text{Quadratic Function}
\end{aligned}
$$

What about the average case?

Outline

1. Administrivia
2. Course Overview
3. Analyzing Algorithms
4. Recap

Points to Take Home

1. Reasoning about Correctness/ Incorrectness of Algorithms.
2. Running Time Analysis for Iterative Algorithms.
3. Reading Material:
   - The Algorithm Design Manual, Chapter 1: Sections 1.1, 1.2, 1.3, and 1.4.
   - Introduction to Algorithms, Chapter 2: Sections 2.1 and 2.2.

Next Lecture: Asymptotic Analysis