

Data Structures and Algorithms

Week 2

1. Complexity of algorithms
2. Correctness of algorithms
3. Asymptotic analysis
4. Special case analysis

Analysis of Algorithms

- Efficiency:
 - Running time
 - Space used
- Efficiency is defined as a function of the input size:
 - Number of data elements (numbers, points).
 - The number of bits of an input number.

DSA09

M. Böhlen

2

The RAM model

- It is important to choose the level of detail.
- The RAM model:
 - Instructions (each taking constant time) – we usually choose one type of instruction as a **characteristic** operation that is counted:
 - Arithmetic (add, subtract, multiply, etc.)
 - Data movement (assign)
 - Control flow (branch, subroutine call, return)
 - Comparison
 - Data types – integers, characters, and floats

Analysis of Insertion Sort/1

- Time to compute the **running time** as a function of the **input size** (exact analysis).

<code>for j := 2 to n do</code>	<code>cost</code>	<code>times</code>
<code> key := A[j]</code>	<code>c1</code>	<code>n</code>
<code> // Insert A[j] into A[1..j-1]</code>	<code>c2</code>	<code>n-1</code>
<code> i := j-1</code>	<code>0</code>	<code>n-1</code>
<code> while i>0 and A[i]>key do</code>	<code>c3</code>	<code>n-1</code>
<code>A[i+1] := A[i]</code>	<code>c4</code>	$\sum_{j=2}^n t_j$
<code>i--</code>	<code>c5</code>	$\sum_{j=2}^n (t_j-1)$
<code>A[i+1] := key</code>	<code>c6</code>	$\sum_{j=2}^n (t_j-1)$
	<code>c7</code>	<code>n-1</code>

DSA09

M. Böhlen

3

DSA09

M. Böhlen

4

Analysis of Insertion Sort/2

- The running time of an algorithm is the sum of the running times of each statement.
- A statement with cost c that is executed n times contributes $c*n$ to the running time.
- The total running time $T(n)$ of insertion sort is
 - $T(n) = c1*n + c2(n-1) + c3(n-1) + c4\sum_{j=2}^n t_j$
 $c5\sum_{j=2}^n (t_j-1) + c6\sum_{j=2}^n (t_j-1) + c7(n-1)$

Analysis of Insertion Sort/3

- Often the performance depends on the details of the input (not only length n).
- This is modeled by t_j .
- In the case of insertion sort the time t_j depends on the original sorting of the input array.

Performance Analysis

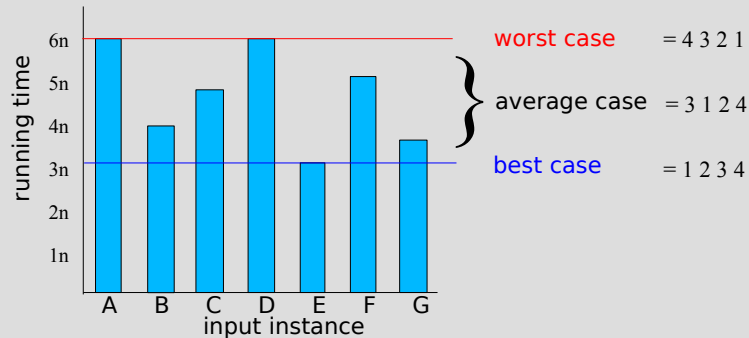
- Often it is sufficient to count the number of iterations of the core (innermost) part.
 - No distinction between comparisons, assignments, etc (that means roughly the same cost for all of them).
 - Gives precise enough results.
- In some cases the cost of selected operations dominates all other costs.
 - Disk I/O versus RAM operations.
 - Database systems.

Best/Worst/Average Case/1

- Analyzing insertion sort's $\sum_{j=2}^n (t_j-1)$
 - **Best case:** elements already sorted, $t_j=1$, innermost loop is zero, total running time is *linear* (time = $an+b$).
 - **Worst case:** elements sorted in inverse order, $t_j=j$, total running time is *quadratic* (time = an^2+bn+c).
 - **Average case:** $t_j=j/2$, total running time is *quadratic* (time = an^2+bn+c).

Best/Worst/Average Case/2

- For a specific size of input size n , investigate running times for different input instances:



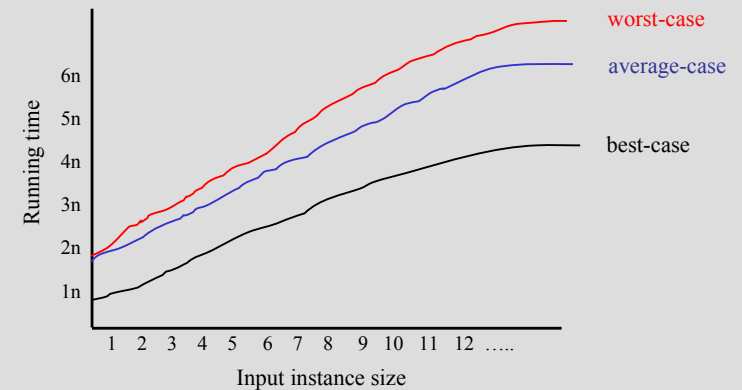
DSA09

M. Böhlen

9

Best/Worst/Average Case/3

- For inputs of all sizes:



DSA09

M. Böhlen

10

Best/Worst/Average Case/4

- Worst case** is usually used:
 - It is an upper-bound.
 - In certain application domains (e.g., air traffic control, surgery) knowing the **worst-case** time complexity is of crucial importance.
 - For some algorithms **worst case** occurs fairly often.
 - The **average case** is often as bad as the **worst case**.
 - Finding the **average case** can be very difficult.

DSA09

M. Böhlen

11

Analysis of Linear Search

```

INPUT: A[1..n] – a sorted array of integers,
        q – an integer.
OUTPUT: j s.t. A[j]=q. NIL if  $\forall j(1 \leq j \leq n): A[j] \neq q$ 

j := 1
while j <= n and A[j] != q do j++
if j <= n then return j
else return NIL
    
```

- Worst case running time: n
- Average case running time: $n/2$
- Best case running time: 0

DSA09

M. Böhlen

12

Binary Search

411

- Idea: Left and right bound. Elements to the right of r are bigger than search element, ...
- In each step half the range of the search space.

```
INPUT: A[1..n] – sorted (increasing) array of integers, q – integer.
OUTPUT: an index j such that A[j] = q. NIL, if  $\forall j (1 \leq j \leq n): A[j] \neq q$ 

l := 1; r := n
do
  m :=  $\lfloor (l+r)/2 \rfloor$ 
  if A[m] = q then return m
  else if A[m] > q then r := m-1
  else l := m+1
while l <= r
return NIL
```

DSA09

M. Böhlen

13

Analysis of Binary Search

- How many times is the loop executed?
 - With each execution the difference between l and r is cut in half.
 - Initially the difference is n .
 - The loop stops when the difference becomes 0 (less than 1).
 - How many times do you have to cut n in half to get 0?
 - $\log n$ – better than the brute-force approach of linear search (n).

DSA09

M. Böhlen

14

Linear vs Binary Search

8

- Costs of linear search: n
- Costs of binary search: $\log(n)$
- Should we care?
- Phone book with n entries:
 - $n = 200'000$, $\log n = \log 200'000 = 18$
 - $n = 2M$, $\log 2M = 21$
 - $n = 20M$, $\log 20M = 24$

DSA09

M. Böhlen

15

Correctness of Algorithms

- An algorithm is *correct* if for any legal input it terminates and produces the desired output.
- Automatic proof of correctness is not possible.
- There are practical techniques and rigorous formalisms that help to reason about the correctness of (parts of) algorithms.

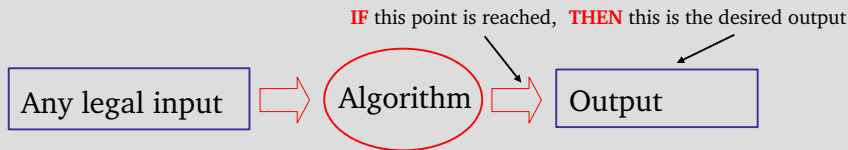
DSA09

M. Böhlen

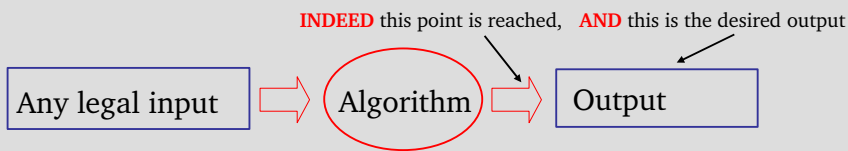
16

Partial and Total Correctness

Partial correctness



Total correctness



Assertions

- To prove partial correctness we associate a number of **assertions** (statements about the state of the execution) with specific checkpoints in the algorithm.
 - E.g., $A[1], \dots, A[j]$ form an increasing sequence
- **Preconditions** – assertions that must be valid *before* the execution of an algorithm or a subroutine (**INPUT**).
- **Postconditions** – assertions that must be valid *after* the execution of an algorithm or a subroutine (**OUTPUT**).

Pre/post-conditions

- Example:
 - Write a pseudocode algorithm to find the two smallest numbers in a sequence of numbers (given as an array).
- **INPUT:** an array of integers $A[1..n]$, $n > 0$
- **OUTPUT:** $(m1, m2)$ such that either
 - $m1 \in A < m2 \in A$ and for each $i \in [1..n]$: $m1 \leq A[i]$ and, if $A[i] \neq m1$, then $m2 \leq A[i]$, or
 - $m2 = m1 = A[1]$ if $\forall j, i \in [1..n]: A[i] = A[j]$.

Loop Invariants

- **Invariants:** assertions that are valid any time they are reached (many times during the execution of an algorithm, e.g., in loops)
- We must show three things about loop invariants:
 - **Initialization:** it is true prior to the first iteration.
 - **Maintenance:** if it is true before an iteration, then it is true after the iteration.
 - **Termination:** when a loop terminates the invariant gives a useful property to show the correctness of the algorithm

Example: Binary Search/1

- We want to show that q is not in A if NIL is returned.
- Invariant:**
 $\forall i \in [1..l-1]: A[i] < q$ (Ia)
 $\forall i \in [r+1..n]: A[i] > q$ (Ib)
- Initialization:** $l = 1, r = n$
the invariant holds because there are no elements to the left of l or to the right of r .
- $l=1$ yields $\forall j, i \in [1..0]: A[i] < q$
this holds because $[1..0]$ is empty
- $r=n$ yields $\forall j, i \in [n+1..n]: A[i] > q$
this holds because $[n+1..n]$ is empty

```

l := 1; r := n;
do
  m := [(l+r)/2]
  if A[m]=q then return m
  else if A[m]>q then r := m-1
  else l := m+1
while l <= r
return NIL
    
```

DSA09

M. Böhlen

21

Example: Binary Search/2

- Invariant:**
 $\forall i \in [1..l-1]: A[i] < q$ (Ia)
 $\forall i \in [r+1..n]: A[i] > q$ (Ib)
- Maintenance:** $l, r, m = \lfloor (l+r)/2 \rfloor$
- $A[m] \neq q$ & $A[m] > q, r = m-1$, A sorted implies
 $\forall k \in [r+1..n]: A[k] > q$ (Ib)
- $A[m] \neq q$ & $A[m] < q, l = m+1$, A sorted implies
 $\forall k \in [1..l-1]: A[k] < q$ (Ia)

```

l := 1; r := n;
do
  m := [(l+r)/2]
  if A[m]=q then return m
  else if A[m]>q then r := m-1
  else l := m+1
while l <= r
return NIL
    
```

DSA09

M. Böhlen

22

Example: Binary Search/3

116

- Invariant:**
 $\forall i \in [1..l-1]: A[i] < q$ (Ia)
 $\forall i \in [r+1..n]: A[i] > q$ (Ib)
- Termination:** $l, r, l \leq r$
- Two cases:
 - $l := m+1$ we get $\lfloor (l+r)/2 \rfloor + 1 > l$
 - $r := m-1$ we get $\lfloor (l+r)/2 \rfloor - 1 < r$
- The range gets smaller during each iteration and the loop will terminate when $l \leq r$ no longer holds.

```

l := 1; r := n;
do
  m := [(l+r)/2]
  if A[m]=q then return m
  else if A[m]>q then r := m-1
  else l := m+1
while l <= r
return NIL
    
```

DSA09

M. Böhlen

23

Example: Insertion Sort/1

- Invariant:**
- outside while loop
 - $A[1..j-1]$ is sorted
 - $A[1..j-1] \in A^{\text{orig}}$
- inside while loop:
 - $A[1..i], \text{key}, A[i+1..j]$
 - $A[1..i]$ is sorted
 - $A[i+1..j]$ is sorted
 - $A[k] > \text{key}, i+1 \leq k \leq j$

```

for j := 2 to n do
  key := A[j]
  i := j-1
  while i>0 and A[i]>key do
    A[i+1] := A[i]
    i--
  A[i+1] := key
    
```

DSA09

M. Böhlen

24

Example: Insertion Sort/2

- outside while loop
 - $A[1..j-1]$ is sorted
 - $A[1..j-1] \in A^{\text{orig}}$
- inside while loop:
 - $A[1..i]$, key, $A[i+1..j]$
 - $A[1..i]$ is sorted
 - $A[i+1..j]$ is sorted
 - $A[k] > \text{key}$, $i+1 \leq k \leq j$
- **Initialization:**
 - $j=2$: the invariant holds, $A[1..1]$ is trivially sorted.
 - $i=j-1$: $A[1..j-1]$, key, $A[j..j]$ where $\text{key}=A[j]$
 $A[1..j-1]$ is sorted (invariant of outer loop)
 $A[j..j]$ is sorted

```
for j := 2 to n do
  key := A[j]
  i := j-1
  while i>0 and A[i]>key do
    A[i+1] := A[i]
    i--
  A[i+1] := key
```

DSA09

M. Böhlen

25

Example: Insertion Sort/3

- outside while loop
 - $A[1..j-1]$ is sorted
 - $A[1..j-1] \in A^{\text{orig}}$
- inside while loop:
 - $A[1..i]$, key, $A[i+1..j]$
 - $A[1..i]$ is sorted
 - $A[i+1..j]$ is sorted
 - $A[k] > \text{key}$, $i+1 \leq k \leq j$

```
for j := 2 to n do
  key := A[j]
  i := j-1
  while i>0 and A[i]>key do
    A[i+1] := A[i]
    i--
  A[i+1] := key
```

Maintenance:

- $A[1..j-1]$ sorted + insert $A[j]$ \Rightarrow $A[1..j]$ sorted
- $A[1..i]$, key, $A[i+1..j]$ satisfies conditions because of condition $A[i] > \text{key}$ and $A[1..j-1]$ being sorted

DSA09

M. Böhlen

26

Example: Insertion Sort/4

- outside while loop
 - $A[1..j-1]$ is sorted
 - $A[1..j-1] \in A^{\text{orig}}$
- inside while loop:
 - $A[1..i]$, key, $A[i+1..j]$
 - $A[1..i]$ is sorted
 - $A[i+1..j]$ is sorted
 - $A[k] > \text{key}$, $i+1 \leq k \leq j$

```
for j := 2 to n do
  key := A[j]
  i := j-1
  while i>0 and A[i]>key do
    A[i+1] := A[i]
    i--
  A[i+1] := key
```

Termination:

- main loop, $j=n+1$: $A[1..n]$ sorted.
- $A[i] \leq \text{key}$: $(A[1..i], \text{key}, A[i+1..j])$ is sorted
- $i=0$: $(\text{key}, A[1..j])$ is sorted.

DSA09

M. Böhlen

27

Asymptotic Analysis

- Goal: to simplify the analysis of the running time by getting rid of details, which are affected by specific implementation and hardware
 - “rounding” of numbers: $1,000,001 \approx 1,000,000$
 - “rounding” of functions: $3n^2 \approx n^2$
- Capturing the essence: how the running time of an algorithm increases with the size of the input *in the limit*.
 - Asymptotically more efficient algorithms are best for all but small inputs

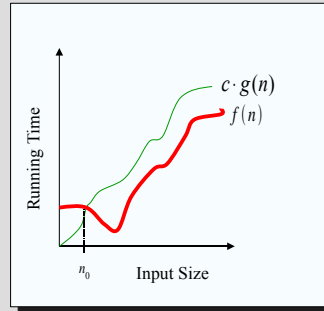
DSA09

M. Böhlen

28

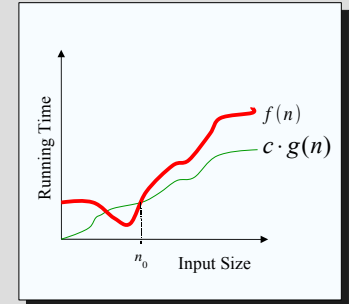
Asymptotic Notation/1

- The “big-Oh” O -Notation
 - asymptotic upper bound
 - $f(n) = O(g(n))$ iff there exists constants $c > 0$ and $n_0 > 0$, s.t. $f(n) \leq c g(n)$ for $n \geq n_0$
 - $f(n)$ and $g(n)$ are functions over non-negative integers
- Used for *worst-case* analysis



Asymptotic Notation/2

- The “big-Omega” Ω -Notation
 - asymptotic lower bound
 - $f(n) = \Omega(g(n))$ iff there exists constants $c > 0$ and $n_0 > 0$, s.t. $c g(n) \leq f(n)$ for $n \geq n_0$
- Used to describe *best-case* running times or lower bounds of algorithmic problems.
 - E.g., lower-bound of searching in an unsorted array is $\Omega(n)$.

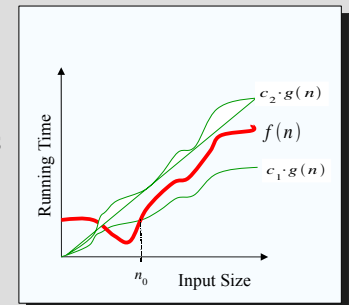


Asymptotic Notation/3

- Simple Rule: Drop lower order terms and constant factors.
 - $50 n \log n$ is $O(n \log n)$
 - $7n - 3$ is $O(n)$
 - $8n^2 \log n + 5n^2 + n$ is $O(n^2 \log n)$
- Note: Although $(50 n \log n)$ is $O(n^5)$, it is expected that an approximation is of the smallest possible order.

Asymptotic Notation/4

- The “big-Theta” Θ -Notation
 - asymptotically tight bound
 - $f(n) = \Theta(g(n))$ if there exists constants $c_1 > 0$, $c_2 > 0$, and $n_0 > 0$, s.t. for $n \geq n_0$ $c_1 g(n) \leq f(n) \leq c_2 g(n)$
- $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- $O(f(n))$ is often abused instead of $\Theta(f(n))$



Asymptotic Notation/5

- Two more asymptotic notations
 - "Little-Oh" notation $f(n) = o(g(n))$
non-tight analogue of Big-Oh
 - For every $c > 0$, there exists $n_0 > 0$, s.t.
 $f(n) < c g(n)$ for $n \geq n_0$
 - If $f(n) = o(g(n))$, it is said that $g(n)$ dominates $f(n)$.
 - "Little-omega" notation $f(n) = \omega(g(n))$
non-tight analogue of Big-Omega

Asymptotic Notation/6

- Analogy with real numbers
 - $f(n) = O(g(n)) \cong f \leq g$
 - $f(n) = \Omega(g(n)) \cong f \geq g$
 - $f(n) = \Theta(g(n)) \cong f = g$
 - $f(n) = o(g(n)) \cong f < g$
 - $f(n) = \omega(g(n)) \cong f > g$
- Abuse of notation: $f(n) = O(g(n))$
actually means $f(n) \in O(g(n))$

Comparison of Running Times

- Determining the maximal problem size.

Running Time $T(n)$ in μs	1 second	1 minute	1 hour
$400n$	2500	150'000	9'000'000
$20n \log n$	4096	166'666	7'826'087
$2n^2$	707	5477	42'426
n^4	31	88	244
2^n	19	25	31

Special Case Analysis

- How do we proceed if we have to check the correctness of a program?
- Consider extreme cases and make sure our solution works in all cases.
- The problem is then reduced to identifying special cases.
- This is related to INPUT and OUTPUT specifications.

Special Cases

- empty data structure (array, file, list, ...)
- single element data structure
- completely filled data structure
- entering a function
- termination of a function
- zero, empty string
- negative number
- border of domain
- start of loop
- end of loop
- first iteration of loop

Sortedness/1

- The following algorithm checks whether an array is sorted.

```
INPUT: A[1..n] – an array integers.  
OUTPUT: TRUE if A is sorted; FALSE otherwise  
for i := 1 to n  
  if A[i] ≥ A[i+1] then return FALSE  
return TRUE
```

- Analyze the algorithm by considering special cases.

Sortedness/2

```
INPUT: A[1..n] – an array integers.  
OUTPUT: TRUE if A is sorted; FALSE otherwise  
for i := 1 to n-1  
  if A[i] > A[i+1] then return FALSE  
return TRUE
```

- Start of loop, $i=1 \rightarrow$ OK
- End of loop, $i=n-1 \rightarrow$ OK
- First iteration, from $i=1$ to $i=2 \rightarrow$ OK
- $A=[1,1,1] \rightarrow$ OK
- Empty data structure, $n=0 \rightarrow ?$ (for loop)
- $A=[-1,0,1,-3] \rightarrow$ OK
- $A=[3,2,1] \rightarrow ?$ (ascending, descending)

Binary Search, Variant 1

- Analyze the following algorithm by considering special cases.

```
l := 1; r := n  
do  
  m :=  $\lfloor (l+r)/2 \rfloor$   
  if A[m] = q then return m  
  else if A[m] > q then r := m-1  
  else l := m+1  
while l < r  
return NIL
```

Binary Search, Variant2

- Analyze the following algorithm by considering special cases.

```
l := 1; r := n
while l < r do {
  m := ⌊(l+r)/2⌋
  if A[m] ≤ q
    then l := m+1 else r := m
}
if A[l-1] = q
  then return q else return NIL
```

DSA09

M. Böhlen

41

Binary Search, Variant3

- Analyze the following algorithm by considering special cases.

```
l := 1; r := n
while l ≤ r do
  m := ⌊(l+r)/2⌋
  if A[m] ≤ q
    then l := m+1 else r := m
if A[l-1] = q
  then return q else return NIL
```

DSA09

M. Böhlen

42

A Quick Math Refresher

- Arithmetic progression

$$\sum_{i=0}^n i = 1+2+3+\dots+n = \frac{n(1+n)}{2}$$

- Geometric progression

– given an integer n_0 and a real number $0 < a \neq 1$

$$\sum_{i=0}^n a^i = 1+a+a^2+\dots+a^n = \frac{1-a^{n+1}}{1-a}$$

– geometric progressions exhibit exponential growth

DSA09

M. Böhlen

43

Miscellaneous/1

- Manipulating summations:

$$- \sum_j c a_j = c \sum_j a_j$$

$$- \sum_j (a_j + b_j) = \sum_j a_j + \sum_j b_j$$

DSA09

M. Böhlen

44

Miscellaneous/2

- Manipulating logarithms and powers:
 - $a \log b = \log b / \log a$
 - $\log a^b = b \log a$
 - $\log(ab) = \log a + \log b$
 - $a^{mn} = (a^m)^n$
 - $a^m a^n = a^{m+n}$
 - $a^{a \log(b)} = b$

DSA09

M. Böhlen

45

Summations

- The running time of insertion sort is determined by a nested loop.

```
for j := 2 to n
  key := A[j]
  i := j-1
  while i>0 and A[i]>key
    A[i+1] := A[i]
    i := i-1
  A[i+1] := key
```

- Nested loops correspond to summations: $\sum_{j=2}^n (j-1)$

DSA09

M. Böhlen

46

Proof by Induction/1

- We want to show that property P is true for all integers $n \geq n_0$.
- **Basis:** prove that P is true for n_0 .
- **Inductive step:** prove that if P is true for all k such that $n_0 \leq k \leq n-1$ then P is also true for n .
- Example $S(n) = \sum_{i=0}^n i = \frac{n(n+1)}{2}$ for $n \geq 1$
- Basis $S(1) = \sum_{i=0}^1 i = \frac{1(1+1)}{2}$

DSA09

M. Böhlen

47

Proof by Induction/2

- Inductive Step

$$S(k) = \sum_{i=0}^k i = \frac{k(k+1)}{2} \text{ for } 1 \leq k \leq n-1$$
$$S(n) = \sum_{i=0}^n i = \sum_{i=0}^{n-1} i + n = S(n-1) + n =$$
$$= (n-1) \frac{(n-1+1)}{2} + n = \frac{(n^2 - n + 2n)}{2} =$$
$$= \frac{n(n+1)}{2}$$

DSA09

M. Böhlen

48

Summary

- Algorithmic complexity
- Correctness of algorithms
 - Pre/Post conditions
 - Invariants
- Asymptotic analysis
 - Big O notation
 - Growth of functions and asymptotic notation
- Special case analysis

Next Week

- Divide-and-conquer
- Merge sort
- Writing recurrences to analyze the running time of recursive algorithms.