# CISC 1400
# Discrete Structures
## Chapter 8
## Algorithms

Gary M. Weiss

Fordham University Department of Computer and Information Sciences


Summer, 2019

# What is an algorithm?

- There are many ways to define an algorithm

# What is an algorithm?

- There are many ways to define an algorithm
  - An algorithm is a step-by-step procedure for carrying out a task or solving a problem

# What is an algorithm?

- There are many ways to define an algorithm
  - An algorithm is a step-by-step procedure for carrying out a task or solving a problem
  - an unambiguous computational procedure that takes some input and generates some output

# What is an algorithm?

- There are many ways to define an algorithm
  - An algorithm is a step-by-step procedure for carrying out a task or solving a problem
  - an unambiguous computational procedure that takes some input and generates some output
  - a set of well-defined instructions for completing a task with a finite amount of effort in a finite amount of time

# What is an algorithm?

- There are many ways to define an algorithm
  - An algorithm is a step-by-step procedure for carrying out a task or solving a problem
  - an unambiguous computational procedure that takes some input and generates some output
  - a set of well-defined instructions for completing a task with a finite amount of effort in a finite amount of time
  - a set of instructions that can be mechanically performed in order to solve a problem

# Key aspects of an algorithm

- An algorithm must be precise

# Key aspects of an algorithm

- An algorithm must be precise
  - The description of an algorithm must be clear and detailed enough so that someone (or something) can execute it

# Key aspects of an algorithm

- An algorithm must be precise
  - The description of an algorithm must be clear and detailed enough so that someone (or something) can execute it
  - One way to ensure this is to describe it using actual computer code, which is guaranteed to be unambiguous

# Key aspects of an algorithm

- An algorithm must be precise
  - The description of an algorithm must be clear and detailed enough so that someone (or something) can execute it
  - One way to ensure this is to describe it using actual computer code, which is guaranteed to be unambiguous
  - This is hard to read so pseudocode is often used instead, which is designed to be readable by humans

# Key aspects of an algorithm

- An algorithm must be precise
  - The description of an algorithm must be clear and detailed enough so that someone (or something) can execute it
  - One way to ensure this is to describe it using actual computer code, which is guaranteed to be unambiguous
  - This is hard to read so pseudocode is often used instead, which is designed to be readable by humans
  - Since we assume no programming background, we will use English but will try hard to be clear and precise

# Key aspects of an algorithm

- An algorithm must be precise
  - The description of an algorithm must be clear and detailed enough so that someone (or something) can execute it
  - One way to ensure this is to describe it using actual computer code, which is guaranteed to be unambiguous
  - This is hard to read so pseudocode is often used instead, which is designed to be readable by humans
  - Since we assume no programming background, we will use English but will try hard to be clear and precise
- An algorithm operates on input and generates output

# Key aspects of an algorithm

- An algorithm must be precise
    - The description of an algorithm must be clear and detailed enough so that someone (or something) can execute it
    - One way to ensure this is to describe it using actual computer code, which is guaranteed to be unambiguous
    - This is hard to read so pseudocode is often used instead, which is designed to be readable by humans
    - Since we assume no programming background, we will use English but will try hard to be clear and precise
- An algorithm operates on input and generates output
- An algorithm completes in a finite number of steps
    - This is a non-trivial requirement since certain methods may sometimes run forever!

# Algorithms (as per Donald Knuth)

An algorithm should have the following characteristics:

# Algorithms (as per Donald Knuth)

An algorithm should have the following characteristics:

1. **Finiteness:** It should terminate after a finite number of steps.

# Algorithms (as per Donald Knuth)

An algorithm should have the following characteristics:

1. **Finiteness:** It should terminate after a finite number of steps.
2. **Definiteness:** Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.

# Algorithms (as per Donald Knuth)

An algorithm should have the following characteristics:

1. **Finiteness:** It should terminate after a finite number of steps.
2. **Definiteness:** Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.
3. **Input:** An algorithm finitely many inputs, i.e., quantities given to it initially before the algorithm begins. (This includes the case of no input whatsoever.)

# Algorithms (as per Donald Knuth)

An algorithm should have the following characteristics:

1. **Finiteness:** It should terminate after a finite number of steps.
2. **Definiteness:** Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.
3. **Input:** An algorithm finitely many inputs, i.e., quantities given to it initially before the algorithm begins. (This includes the case of no input whatsoever.)
4. **Output:** An algorithm must produce finitely many (but at least one!) outputs, i.e., quantities having a specified relation to the inputs.

# Algorithms (as per Donald Knuth)

An algorithm should have the following characteristics:

1. **Finiteness:** It should terminate after a finite number of steps.
2. **Definiteness:** Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.
3. **Input:** An algorithm finitely many inputs, i.e., quantities given to it initially before the algorithm begins. (This includes the case of no input whatsoever.)
4. **Output:** An algorithm must produce finitely many (but at least one!) outputs, i.e., quantities having a specified relation to the inputs.
5. **Effectiveness:** All the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a person using pencil and paper.

# Algorithms (as per Donald Knuth)

An algorithm should have the following characteristics:

1. **Finiteness:** It should terminate after a finite number of steps.
2. **Definiteness:** Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.
3. **Input:** An algorithm finitely many inputs, i.e., quantities given to it initially before the algorithm begins. (This includes the case of no input whatsoever.)
4. **Output:** An algorithm must produce finitely many (but at least one!) outputs, i.e., quantities having a specified relation to the inputs.
5. **Effectiveness:** All the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a person using pencil and paper.

# Algorithms (as per Donald Knuth)

An algorithm should have the following characteristics:

1. **Finiteness:** It should terminate after a finite number of steps.
2. **Definiteness:** Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.
3. **Input:** An algorithm finitely many inputs, i.e., quantities given to it initially before the algorithm begins. (This includes the case of no input whatsoever.)
4. **Output:** An algorithm must produce finitely many (but at least one!) outputs, i.e., quantities having a specified relation to the inputs.
5. **Effectiveness:** All the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a person using pencil and paper.

Reference: Donald E. Knuth, *The Art of Computer*

# Applications of algorithms

- Algorithms can implement many of the operations we study in this book, such as set membership and union

# Applications of algorithms

- Algorithms can implement many of the operations we study in this book, such as set membership and union
  - Data structures and the algorithms that operate on them are so important to CS that most CS majors are required to take a course on data structures

# Applications of algorithms

- Algorithms can implement many of the operations we study in this book, such as set membership and union
    - Data structures and the algorithms that operate on them are so important to CS that most CS majors are required to take a course on data structures
        - Sets and Sequences are examples of data structures

# Applications of algorithms

- Algorithms can implement many of the operations we study in this book, such as set membership and union
  - Data structures and the algorithms that operate on them are so important to CS that most CS majors are required to take a course on data structures
    - Sets and Sequences are examples of data structures
    - membership is a set operation implemented using an algorithm

# Applications of algorithms

- Algorithms can implement many of the operations we study in this book, such as set membership and union
  - Data structures and the algorithms that operate on them are so important to CS that most CS majors are required to take a course on data structures
    - Sets and Sequences are examples of data structures
    - membership is a set operation implemented using an algorithm
    - union and intersection are also set operations implemented using an algorithm

# Applications of algorithms

- Algorithms can implement many of the operations we study in this book, such as set membership and union
  - Data structures and the algorithms that operate on them are so important to CS that most CS majors are required to take a course on data structures
    - Sets and Sequences are examples of data structures
    - membership is a set operation implemented using an algorithm
    - union and intersection are also set operations implemented using an algorithm
    - How might you implement these operations?

# Applications of algorithms

- Algorithms can implement many of the operations we study in this book, such as set membership and union
  - Data structures and the algorithms that operate on them are so important to CS that most CS majors are required to take a course on data structures
    - Sets and Sequences are examples of data structures
    - membership is a set operation implemented using an algorithm
    - union and intersection are also set operations implemented using an algorithm
    - How might you implement these operations?
  - Without such structures and without efficient algorithms for operating on them, you could never play a video game

# Applications of algorithms

- Algorithms can implement many of the operations we study in this book, such as set membership and union
  - Data structures and the algorithms that operate on them are so important to CS that most CS majors are required to take a course on data structures
    - Sets and Sequences are examples of data structures
    - membership is a set operation implemented using an algorithm
    - union and intersection are also set operations implemented using an algorithm
    - How might you implement these operations?
  - Without such structures and without efficient algorithms for operating on them, you could never play a video game
- Algorithms can also used to implement mathematical processes/entities. Most mathematical functions are implemented using computer algorithms

# Real world applications of algorithms

Algorithms are also used to solve specific, complex, real world problems:

# Real world applications of algorithms

Algorithms are also used to solve specific, complex, real world problems:

- Google's success is largely due to its PageRank algorithm, which determines the "importance" of every web page

# Real world applications of algorithms

Algorithms are also used to solve specific, complex, real world problems:

- Google's success is largely due to its PageRank algorithm, which determines the "importance" of every web page
- Prim's algorithm can be used by a cable company to determine how to connect all of the homes in a town using the least amount of cable

# Real world applications of algorithms

Algorithms are also used to solve specific, complex, real world problems:

- Google's success is largely due to its PageRank algorithm, which determines the "importance" of every web page
- Prim's algorithm can be used by a cable company to determine how to connect all of the homes in a town using the least amount of cable
- Dijkstra's algorithm can be used to find the shortest route between a city and all other cities

# Real world applications of algorithms

Algorithms are also used to solve specific, complex, real world problems:

- Google's success is largely due to its PageRank algorithm, which determines the "importance" of every web page
- Prim's algorithm can be used by a cable company to determine how to connect all of the homes in a town using the least amount of cable
- Dijkstra's algorithm can be used to find the shortest route between a city and all other cities
- The RSA encryption algorithm makes e-commerce possible by allowing for secure transactions over the Web

# Algorithms and computers

- Algorithms have been used for thousands of years and have been executed by humans (possibly with pencil and paper)

# Algorithms and computers

- Algorithms have been used for thousands of years and have been executed by humans (possibly with pencil and paper)
    - We all know the algorithm for performing long division

# Algorithms and computers

- Algorithms have been used for thousands of years and have been executed by humans (possibly with pencil and paper)
  - We all know the algorithm for performing long division
- Work on algorithms exploded with the development of fast digital computers and are a cornerstone of Computer Science

# Algorithms and computers

- Algorithms have been used for thousands of years and have been executed by humans (possibly with pencil and paper)
  - We all know the algorithm for performing long division
- Work on algorithms exploded with the development of fast digital computers and are a cornerstone of Computer Science
  - Many algorithms are only feasible when implemented on computers

# Algorithms and computers

- Algorithms have been used for thousands of years and have been executed by humans (possibly with pencil and paper)
  - We all know the algorithm for performing long division
- Work on algorithms exploded with the development of fast digital computers and are a cornerstone of Computer Science
  - Many algorithms are only feasible when implemented on computers
- But even with today's fast computers, some problems still cannot be solved using existing algorithms
  - The search for better and more efficient algorithms continues

# Algorithms and computers

- Algorithms have been used for thousands of years and have been executed by humans (possibly with pencil and paper)
    - We all know the algorithm for performing long division
- Work on algorithms exploded with the development of fast digital computers and are a cornerstone of Computer Science
    - Many algorithms are only feasible when implemented on computers
- But even with today's fast computers, some problems still cannot be solved using existing algorithms
    - The search for better and more efficient algorithms continues
- Interestingly enough, some problems have been shown to have no algorithmic solution (e.g., the "halting problem")

# Searching and sorting algorithms

- Two of the most studied classes of algorithms in CS are searching and sorting algorithms
  - Search algorithms are important because quickly locating information is central to many tasks
  - Sorting algorithms are important because information can be located much more quickly if it is first sorted
- Searching and sorting algorithms are often used to introduce the topic of algorithms and we follow this convention

# Search algorithms

- Problem: determine if an element $x$ is in a list $L$
- We will look at two simple search algorithms
  - Linear search
  - Binary search
- The elements in $L$ have some ordering, so that there is a first element, second element, etc.
- These algorithms can easily be applied to sets since we do not exploit this ordering (i.e., we do not assume the elements are sorted).

# Linear search algorithm

The algorithm below will search for an element $x$ in List $L$ and will return "FOUND" if $x$ is in the list and "NOT FOUND" otherwise. $L$ has $n$ items and $L[i]$ refers to the $i^{\text{th}}$ element in $L$.

**Linear Search Algorithm**

1. **repeat** as $i$ varies from 1 to $n$
2.     if $L[i] = x$ then return "FOUND" and stop
3. return "NOT FOUND"

Note: The repeat loop spans lines 1 and 2.

# Efficiency of linear search algorithm

- If $x$ appears once in $L$, on average how many comparisons (line 2) would you expect the algorithm to make ...

# Efficiency of linear search algorithm

- If $x$ appears once in $L$, on average how many comparisons (line 2) would you expect the algorithm to make ...
  - in the worst case?

# Efficiency of linear search algorithm

- If $x$ appears once in $L$, on average how many comparisons (line 2) would you expect the algorithm to make ...
  - in the worst case? $n$ comparisons
  - on average?

# Efficiency of linear search algorithm

- If $x$ appears once in $L$, on average how many comparisons (line 2) would you expect the algorithm to make ...
  - in the worst case? $n$ comparisons
  - on average? $n/2$ comparisons
  - in the best case?

# Efficiency of linear search algorithm

- If $x$ appears once in $L$, on average how many comparisons (line 2) would you expect the algorithm to make ...
  - in the worst case? $n$ comparisons
  - on average? $n/2$ comparisons
  - in the best case? 1 comparison

# Efficiency of linear search algorithm

- If $x$ appears once in $L$, on average how many comparisons (line 2) would you expect the algorithm to make ...
  - in the worst case? $n$ comparisons
  - on average? $n/2$ comparisons
  - in the best case? 1 comparison
- If x does *not* appear in L, how many comparisons would you expect the algorithm to make?

# Efficiency of linear search algorithm

- If $x$ appears once in $L$, on average how many comparisons (line 2) would you expect the algorithm to make ...
  - in the worst case? $n$ comparisons
  - on average? $n/2$ comparisons
  - in the best case? 1 comparison
- If x does *not* appear in L, how many comparisons would you expect the algorithm to make?
  - $n$ comparisons

# Efficiency of linear search algorithm

- If $x$ appears once in $L$, on average how many comparisons (line 2) would you expect the algorithm to make ...
  - in the worst case?$n$ comparisons
  - on average?$n/2$ comparisons
  - in the best case? 1 comparison
- If x does *not* appear in L, how many comparisons would you expect the algorithm to make?
  - $n$ comparisons
- Would such an algorithm be useful for finding someone in a large (unsorted) phone book?

# Efficiency of linear search algorithm

- If $x$ appears once in $L$, on average how many comparisons (line 2) would you expect the algorithm to make ...
  - in the worst case? $n$ comparisons
  - on average? $n/2$ comparisons
  - in the best case? 1 comparison
- If x does *not* appear in L, how many comparisons would you expect the algorithm to make?
  - $n$ comparisons
- Would such an algorithm be useful for finding someone in a large (unsorted) phone book?
  - No, it would require scanning through the entire phone book (phone books are sorted for a reason)!

# Efficiency of linear search algorithm

- If $x$ appears once in $L$, on average how many comparisons (line 2) would you expect the algorithm to make ...
  - in the worst case?$n$ comparisons
  - on average?$n/2$ comparisons
  - in the best case? 1 comparison
- If x does *not* appear in L, how many comparisons would you expect the algorithm to make?
  - $n$ comparisons
- Would such an algorithm be useful for finding someone in a large (unsorted) phone book?
  - No, it would require scanning through the entire phone book (phone books are sorted for a reason)!
  - What if we had to check 1,000 people to see if they are in the phone book?

# Efficiency of linear search algorithm

- If $x$ appears once in $L$, on average how many comparisons (line 2) would you expect the algorithm to make ...
  - in the worst case? $n$ comparisons
  - on average? $n/2$ comparisons
  - in the best case? 1 comparison
- If x does *not* appear in L, how many comparisons would you expect the algorithm to make?
  - $n$ comparisons
- Would such an algorithm be useful for finding someone in a large (unsorted) phone book?
  - No, it would require scanning through the entire phone book (phone books are sorted for a reason)!
  - What if we had to check 1,000 people to see if they are in the phone book?
    - Then it would be even worse!

# Binary search algorithm overview

- The binary search algorithm assumes that $L$ is sorted
- This algorithm need not need explicitly examine each element
- At any given time it maintains a "window" in which element $x$ may reside
  - The window is defined by the indices *min* and *max* which specify the leftmost and rightmost boundaries in $L$
- At each iteration of the algorithm the window is cut in half

**Binary Search Algorithm**

1. Initialize $min \leftarrow 1$ and $max \leftarrow n$
2. **Repeat** until $min > max$
3. $\quad midpoint = \frac{1}{2}(min + max)$
4. $\quad$ compare $x$ to $L[midpoint]$
   a. $\quad$ if $x = L[midpoint]$ then return "FOUND"
   b. $\quad$ if $x > L[midpoint]$ then $min \leftarrow midpoint + 1$
   c. $\quad$ if $x < L[midpoint]$ then $max \leftarrow midpoint - 1$
5. return "NOT FOUND"

Note: the repeat loop spans lines 2–4.

# Binary search example

Use binary search to find the element "40" in the sorted list (10 30 40 50 60 70 80 90). List the values of *min*, *max* and *midpoint* after each iteration of step 4. How many values are compared to "40"?

# Binary search example

Use binary search to find the element "40" in the sorted list (10 30 40 50 60 70 80 90). List the values of *min*, *max* and *midpoint* after each iteration of step 4. How many values are compared to "40"?

1. $Min = 1$ and $max = 8$ and $midpoint = \frac{1}{2}(1+8) = 4$ (round down). Since $L[4] = 50$ and since $40 < 50$ we execute step 4c and $max = midpoint - 1 = 3$.

# Binary search example

Use binary search to find the element "40" in the sorted list (10 30 40 50 60 70 80 90). List the values of *min*, *max* and *midpoint* after each iteration of step 4. How many values are compared to "40"?

1. $Min = 1$ and $max = 8$ and $midpoint = \frac{1}{2}(1+8) = 4$ (round down). Since $L[4] = 50$ and since $40 < 50$ we execute step 4c and $max = midpoint - 1 = 3$.
2. Now $min = 1$, $max = 3$ and $midpoint = \frac{1}{2}(1+3) = 2$. Since $L[2] = 30$ and $40 > 30$, we execute step 4b and $min = midpoint + 1 = 3$.

# Binary search example

Use binary search to find the element "40" in the sorted list (10 30 40 50 60 70 80 90). List the values of *min*, *max* and *midpoint* after each iteration of step 4. How many values are compared to "40"?

1. $Min = 1$ and $max = 8$ and $midpoint = \frac{1}{2}(1+8) = 4$ (round down). Since $L[4] = 50$ and since $40 < 50$ we execute step 4c and $max = midpoint - 1 = 3$.
2. Now $min = 1$, $max = 3$ and $midpoint = \frac{1}{2}(1+3) = 2$. Since $L[2] = 30$ and $40 > 30$, we execute step 4b and $min = midpoint + 1 = 3$.
3. Now $min = 3$, $max = 3$ and $midpoint = \frac{1}{2}(3+3) = 3$. Since $L[3] = 40$ and $40 = 40$, we execute step 4a and return "FOUND."

During execution of the algorithm we check three values: 3, 4, and 5. Since we cut the list in half each iteration, it will shrink very quickly (the search will require about $\log_2 n$ comparisons).

# Comparison of linear and binary search

- In the worst case, linear search will need to go through the entire list of $n$ elements

# Comparison of linear and binary search

- In the worst case, linear search will need to go through the entire list of $n$ elements
- In the worst case, binary search will need to go through about $\log_2 n$ elements

# Comparison of linear and binary search

- In the worst case, linear search will need to go through the entire list of $n$ elements
- In the worst case, binary search will need to go through about $\log_2 n$ elements
- Binary search is much more efficient
  - If $n = 1$K we have $1,024$ vs. 10 comparisons
  - If $n = 1$M we have ˜$1,000,000$ vs. 20 comparisons
  - If $n = 1$G we have ˜$1,000,000,000$ vs 30 comparisons

# Comparison of linear and binary search

- In the worst case, linear search will need to go through the entire list of $n$ elements
- In the worst case, binary search will need to go through about $\log_2 n$ elements
- Binary search is much more efficient
  - If $n = 1K$ we have $1,024$ vs. $10$ comparisons
  - If $n = 1M$ we have ~$1,000,000$ vs. $20$ comparisons
  - If $n = 1G$ we have ~$1,000,000,000$ vs $30$ comparisons
- The drawback is that binary search requires sorting, and this requires a decent amount of work
  - But sorting only has to be done once and this will be worthwhile if we need to search the list many times

# Sorting algorithms

- Sorting algorithms are one of the most heavily studied topics in Computer Science
- Sorting is critical if information is to be found efficiently (as we saw binary search exploits the fact that a list is sorted)
- There are many well known sorting algorithms in Computer Science
- We will study 2 sorting algorithms
  - BubbleSort: a very simple but inefficient sorting algorithm
  - MergeSort: a slightly more complex but efficient sorting algorithm

# BubbleSort algorithm overview

- BubbleSort works by repeatedly scanning the list and in each iteration "bubbles" the largest element in the unsorted part of the list to the end
  - After 1 iteration largest element in last position
  - After 2 iterations largest element in last position and second largest element in second to last position
  - ...
- requires $n - 1$ iterations since at last iteration the only item left must already be in proper position (i.e., the smallest must be in the leftmost position)

# BubbleSort algorithm

BubbleSort will sort the $n$-element list $L = (l_1, l_2, \ldots, l_n)$

**BubbleSort Algorithm**

1. **Repeat** as $i$ varies from $n$ down to 2
2.     **Repeat** as $j$ varies from 1 to $i - 1$
3.         If $l_j > l_{j+1}$ swap $l_j$ with $l_{j+1}$

- The outer loop controls how much of the list is checked each iteration. Only the unsorted part is checked. In the first iteration we check everything.
- The inner loop allows us to bubble up the largest element in the unsorted part of the list

# BubbleSort example

Use BubbleSort to sort the list of number (9 2 8 4 1 3) into increasing order. Note that corresponds to Example 8.3 in the text.

Try it and compare your solution to the solution in the text.

- How many comparisons did you do each iteration?
- Can you find a pattern?
- This will be useful later when we analyze the performance of the algorithm.

# MergeSort algorithm overview

- MergeSort is a *divide-and-conquer* algorithm
  - this means it divides the sorting problem into smaller problems
  - solves the smaller problems
  - then combines the solutions to the smaller problems to solve the original problem
- this deceptively simple algorithm is nonetheless much more efficient than the bubblesort algorithm
- It exploits the fact that combining two sorted lists is very easy
  - How would you sort (1 4 7 8) and (2 5 9)?

# MergeSort algorithm overview

- MergeSort is a *divide-and-conquer* algorithm
  - this means it divides the sorting problem into smaller problems
  - solves the smaller problems
  - then combines the solutions to the smaller problems to solve the original problem
- this deceptively simple algorithm is nonetheless much more efficient than the bubblesort algorithm
- It exploits the fact that combining two sorted lists is very easy
  - How would you sort (1 4 7 8) and (2 5 9)?
  - You would place your finger at the start of each list, copy over the smaller element under each finger, then advance that one finger.

**MergeSort Algorithm**

**function** mergeSort($L$)

1. if $L$ has one element then return($L$); otherwise continue
2. $l_1 \leftarrow$ mergeSort(left half of $L$)
3. $l_2 \leftarrow$ mergeSort(right half of $L$)
4. $L \leftarrow$ merge($l_1, l_2$)
5. return($L$)

# Description of MergeSort algorithm

- mergeSort is a recursive function
  - That means it calls itself
- If the input list contains one element it is trivially sorted so mergeSort is done
- Otherwise mergeSort calls itself on the left and right half of the list and then merges the two lists
- Each of these two calls to itself may lead to additional calls to itself
- Note that mergeSort will completely sort the left side of the original list before it actually starts sorting the right side

# Example of MergeSort algorithm

How would mergesort sort the list (9 2 8 4 1 3) into increasing order?

To help show what is going on, the sorted lists that are about to be merged are shown in bold.

$$\boxed{9\ 2\ 8\ 4\ 1\ 3} \rightarrow \boxed{9\ 2\ 8}\ \boxed{4\ 1\ 3} \rightarrow \boxed{9\ 2}\ \boxed{8}\ \boxed{4\ 1\ 3}$$

$$\boxed{\mathbf{9}}\ \boxed{\mathbf{2}}\ \boxed{8}\ \boxed{4\ 1\ 3} \rightarrow \boxed{\mathbf{2\ 9}}\ \boxed{\mathbf{8}}\ \boxed{4\ 1\ 3} \rightarrow \boxed{2\ 8\ 9}\ \boxed{4\ 1\ 3}$$

$$\boxed{2\ 8\ 9}\ \boxed{4\ 1\ 3} \rightarrow \boxed{2\ 8\ 9}\ \boxed{4\ 1}\ \boxed{3} \rightarrow \boxed{2\ 8\ 9}\ \boxed{\mathbf{4}}\ \boxed{\mathbf{1}}\ \boxed{3}$$

$$\boxed{2\ 8\ 9}\ \boxed{\mathbf{1\ 4}}\ \boxed{\mathbf{3}} \rightarrow \boxed{\mathbf{2\ 8\ 9}}\ \boxed{\mathbf{1\ 3\ 4}} \rightarrow \boxed{1\ 2\ 3\ 4\ 8\ 9}$$

# Analysis of algorithms

- We are done introducing our algorithms. Now we will analyze them.

# Analysis of algorithms

- We are done introducing our algorithms. Now we will analyze them.
- An algorithm is a set of instructions that solves a problem for all input instances

# Analysis of algorithms

- We are done introducing our algorithms. Now we will analyze them.
- An algorithm is a set of instructions that solves a problem for all input instances
- But there may be many algorithms that can solve a problem and all of these are not equally good

# Analysis of algorithms

- We are done introducing our algorithms. Now we will analyze them.
- An algorithm is a set of instructions that solves a problem for all input instances
- But there may be many algorithms that can solve a problem and all of these are not equally good
- One criteria for evaluating an algorithm is *efficiency*

# Analysis of algorithms

- We are done introducing our algorithms. Now we will analyze them.
- An algorithm is a set of instructions that solves a problem for all input instances
- But there may be many algorithms that can solve a problem and all of these are not equally good
- One criteria for evaluating an algorithm is *efficiency*
- The task of determining the efficiency of an algorithm is referred to as the *analysis of algorithms*

# Analysis of algorithms

- We are done introducing our algorithms. Now we will analyze them.
- An algorithm is a set of instructions that solves a problem for all input instances
- But there may be many algorithms that can solve a problem and all of these are not equally good
- One criteria for evaluating an algorithm is *efficiency*
- The task of determining the efficiency of an algorithm is referred to as the *analysis of algorithms*
- Here we will learn to analyze only simple algorithms
  - There are entire courses on analysis of algorithms

# How do we measure efficiency?

- When solving tasks, what are we most concerned with?

# How do we measure efficiency?

- When solving tasks, what are we most concerned with?
  - Hopefully most of us are concerned with correctness. But to be considered an algorithm the procedure must be correct (although a designer needs to make sure of this).

# How do we measure efficiency?

- When solving tasks, what are we most concerned with?
  - Hopefully most of us are concerned with correctness. But to be considered an algorithm the procedure must be correct (although a designer needs to make sure of this).
  - Most of us are pretty concerned with *time* and time is actually the main concern in evaluating the efficiency of algorithms

# How do we measure efficiency?

- When solving tasks, what are we most concerned with?
  - Hopefully most of us are concerned with correctness. But to be considered an algorithm the procedure must be correct (although a designer needs to make sure of this).
  - Most of us are pretty concerned with *time* and time is actually the main concern in evaluating the efficiency of algorithms
  - Space is also a concern, which, for algorithms, means what is the maximum amount of memory the algorithm will require at any one time

# How do we measure efficiency?

- When solving tasks, what are we most concerned with?
  - Hopefully most of us are concerned with correctness. But to be considered an algorithm the procedure must be correct (although a designer needs to make sure of this).
  - Most of us are pretty concerned with *time* and time is actually the main concern in evaluating the efficiency of algorithms
  - Space is also a concern, which, for algorithms, means what is the maximum amount of memory the algorithm will require at any one time
  - We will focus on time, although for some problems, space can actually be the main concern.

# How do we measure the time of an algorithm?

- We could run the algorithm on a computer and measure the time it takes to complete
  - But what computer do we run it on? Different computers have different speeds.
  - We could pick one benchmark computer, but it would not stick around forever
  - Worse yet, the time taken by the algorithm is usually impacted by the specific input, so how do we handle that?

# The run-time complexity of an algorithm

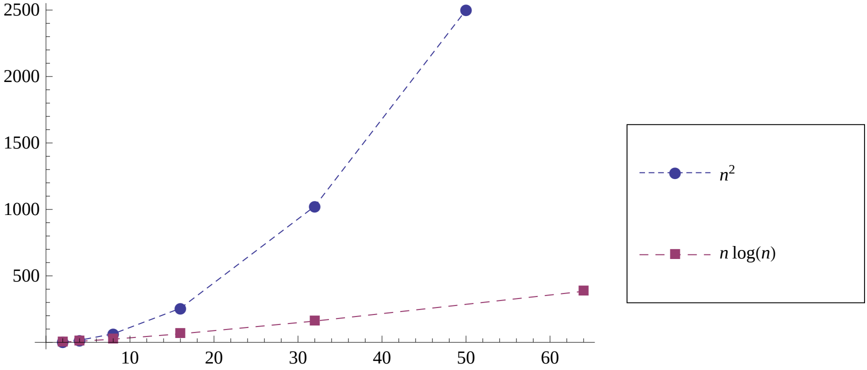
- The standard solution is to focus on the *run-time complexity* of an algorithm
- We determine how the number of operations involved in the algorithms grows relative to the length of the input
- Since inputs of the same length may still take different numbers of operations, we usually focus on the worst-case performance
  - We assume that the input is the hardest input possible

# The running time of BubbleSort and MergeSort

- We can implement BubbleSort and MergeSort as a computer program
  - Then we can run them on various length lists and record the number of operations performed
  - Let bubblesortOps($n$) and mergesortOps($n$) represent the number of operations performed when the list has n elements
- The results might look like those below

| $n$ | 2 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|
| bubblesortOps($n$) | 4 | 16 | 64 | 256 | 1024 | 4096 |
| mergesortOps($n$) | 2 | 8 | 24 | 64 | 160 | 384 |

- We can plot the data from the previous table to get a better visual picture of the growth rate for these functions



# Run-time complexity of BubbleSort and MergeSort

- Using the data in the Table, can we determine closed formulas for bubblesortOps and mergesortOps?

# Run-time complexity of BubbleSort and MergeSort

- Using the data in the Table, can we determine closed formulas for bubblesortOps and mergesortOps?
  - We can see that $\text{bubblesortOps}(n) = n^2$

# Run-time complexity of BubbleSort and MergeSort

- Using the data in the Table, can we determine closed formulas for bubblesortOps and mergesortOps?
  - We can see that $\text{bubblesortOps}(n) = n^2$
  - This is not easy to see, but $\text{mergesortOps}(n) = n \log_2 n$

# Run-time complexity of BubbleSort and MergeSort

- Using the data in the Table, can we determine closed formulas for bubblesortOps and mergesortOps?
  - We can see that $\text{bubblesortOps}(n) = n^2$
  - This is not easy to see, but $\text{mergesortOps}(n) = n \log_2 n$
- Normally one does not determine the run time complexity this way, but rather by analyzing the algorithm.
- We will show how to do this for some simple algorithms

**Linear Search Algorithm**

1. **repeat** as $i$ varies from 1 to $n$
2. if $L[i] = x$ then return "FOUND" and stop
3. return "NOT FOUND"

- How many operations will the linear search algorithm require?

# Analysis of linear search algorithm

**Linear Search Algorithm**

1. **repeat** as $i$ varies from 1 to $n$
2.     if $L[i] = x$ then return "FOUND" and stop
3. return "NOT FOUND"

- How many operations will the linear search algorithm require?
- As stated earlier, since the algorithm checks at most $n$ elements against $x$, the worst-case complexity requires $n$ comparisons.

# Analysis of linear search algorithm

**Linear Search Algorithm**

1. **repeat** as $i$ varies from 1 to $n$
2.     if $L[i] = x$ then return "FOUND" and stop
3. return "NOT FOUND"

- How many operations will the linear search algorithm require?
- As stated earlier, since the algorithm checks at most $n$ elements against $x$, the worst-case complexity requires $n$ comparisons.
  - Note that this performance occurs only when $x$ is not in the list or is the last element in the list.

**Linear Search Algorithm**

1. **repeat** as $i$ varies from 1 to $n$
2. if $L[i] = x$ then return "FOUND" and stop
3. return "NOT FOUND"

- How many operations will the linear search algorithm require?
- As stated earlier, since the algorithm checks at most $n$ elements against $x$, the worst-case complexity requires $n$ comparisons.
  - Note that this performance occurs only when $x$ is not in the list or is the last element in the list.
- What is the best-case complexity of the algorithm?

# Analysis of linear search algorithm

**Linear Search Algorithm**

1. **repeat** as $i$ varies from 1 to $n$
2. if $L[i] = x$ then return "FOUND" and stop
3. return "NOT FOUND"

- How many operations will the linear search algorithm require?
- As stated earlier, since the algorithm checks at most $n$ elements against $x$, the worst-case complexity requires $n$ comparisons.
  - Note that this performance occurs only when $x$ is not in the list or is the last element in the list.
- What is the best-case complexity of the algorithm?
  - 1, which occurs when $x$ is the first item on the list

# Average case complexity

- If you know that the element $x$ to be matched is on the list, what is the average-case complexity of the algorithm?

# Average case complexity

- If you know that the element $x$ to be matched is on the list, what is the average-case complexity of the algorithm?
  - The average case complexity of the algorithm should be $n/2$, since on average you should have to search half of the list

# Average case complexity

- If you know that the element $x$ to be matched is on the list, what is the average-case complexity of the algorithm?
    - The average case complexity of the algorithm should be $n/2$, since on average you should have to search half of the list
- At least for introductory courses on algorithms, the worst-case complexity is what is reported, since it is generally much easier to compute than the average case complexity.

# Analysis of binary search algorithm

- The binary search algorithm, which assumes a sorted list, repeatedly cuts the list to be searched in half
  - If there is 1 element, it will require 1 comparison
  - If there are 2 elements, it may require 2 comparisons
  - If there are 4 elements, it may require 3 comparisons
  - If there are 8 elements, it may require 4 comparisons

# Analysis of binary search algorithm

- The binary search algorithm, which assumes a sorted list, repeatedly cuts the list to be searched in half
  - If there is 1 element, it will require 1 comparison
  - If there are 2 elements, it may require 2 comparisons
  - If there are 4 elements, it may require 3 comparisons
  - If there are 8 elements, it may require 4 comparisons
  - In general, if there are $n$ elements, how many comparisons will be required?

# Analysis of binary search algorithm

- The binary search algorithm, which assumes a sorted list, repeatedly cuts the list to be searched in half
  - If there is 1 element, it will require 1 comparison
  - If there are 2 elements, it may require 2 comparisons
  - If there are 4 elements, it may require 3 comparisons
  - If there are 8 elements, it may require 4 comparisons
  - In general, if there are *n* elements, how many comparisons will be required?
    - It will require $log_2 n$ comparisons

# Analysis of binary search algorithm

- The binary search algorithm, which assumes a sorted list, repeatedly cuts the list to be searched in half
  - If there is 1 element, it will require 1 comparison
  - If there are 2 elements, it may require 2 comparisons
  - If there are 4 elements, it may require 3 comparisons
  - If there are 8 elements, it may require 4 comparisons
  - In general, if there are $n$ elements, how many comparisons will be required?
    - It will require $log_2 n$ comparisons
- If $n$ is not a power of 2, you will need to round up the number of comparisons
  - Thus if there are 3 elements it may require 3 comparisons

# Comparison of linear and binary search algorithms

- The linear search algorithm requires $n$ comparisons worst case
- The binary search algorithm requires $log_2 n$ comparisons worst case
- Which one is faster? Is the difference significant?

# Comparison of linear and binary search algorithms

- The linear search algorithm requires $n$ comparisons worst case
- The binary search algorithm requires $log_2 n$ comparisons worst case
- Which one is faster? Is the difference significant?
- The binary search algorithm is much faster, in that it requires many fewer comparisons
  - If a list has 1 million elements then linear search requires 1,000,000 comparisons while binary search requires only about 20 comparisons!

# Comparison of linear and binary search algorithms

- The linear search algorithm requires $n$ comparisons worst case
- The binary search algorithm requires $log_2 n$ comparisons worst case
- Which one is faster? Is the difference significant?
- The binary search algorithm is much faster, in that it requires many fewer comparisons
  - If a list has 1 million elements then linear search requires 1,000,000 comparisons while binary search requires only about 20 comparisons!
- But the binary search algorithm requires that the list is sorted, whereas linear search does not.

# Comparison of linear and binary search algorithms

- The linear search algorithm requires $n$ comparisons worst case
- The binary search algorithm requires $log_2 n$ comparisons worst case
- Which one is faster? Is the difference significant?
- The binary search algorithm is much faster, in that it requires many fewer comparisons
  - If a list has 1 million elements then linear search requires 1,000,000 comparisons while binary search requires only about 20 comparisons!
- But the binary search algorithm requires that the list is sorted, whereas linear search does not.
- Since sorting requires $n\, log_2 n$ operations, which is more than $n$ operations, it only makes sense to sort and then use binary search if many searches will be made

# Comparison of linear and binary search algorithms

- The linear search algorithm requires $n$ comparisons worst case
- The binary search algorithm requires $log_2 n$ comparisons worst case
- Which one is faster? Is the difference significant?
- The binary search algorithm is much faster, in that it requires many fewer comparisons
  - If a list has 1 million elements then linear search requires 1,000,000 comparisons while binary search requires only about 20 comparisons!
- But the binary search algorithm requires that the list is sorted, whereas linear search does not.
- Since sorting requires $n \, log_2 n$ operations, which is more than $n$ operations, it only makes sense to sort and then use binary search if many searches will be made
  - This is the case with dictionaries, phone books, etc.

# Comparison of linear and binary search algorithms

- The linear search algorithm requires $n$ comparisons worst case
- The binary search algorithm requires $log_2 n$ comparisons worst case
- Which one is faster? Is the difference significant?
- The binary search algorithm is much faster, in that it requires many fewer comparisons
    - If a list has 1 million elements then linear search requires 1,000,000 comparisons while binary search requires only about 20 comparisons!
- But the binary search algorithm requires that the list is sorted, whereas linear search does not.
- Since sorting requires $n\, log_2 n$ operations, which is more than $n$ operations, it only makes sense to sort and then use binary search if many searches will be made
    - This is the case with dictionaries, phone books, etc.
    - This is not the case with airline reservation systems!

# Analysis of the BubbleSort algorithm

- Analyzing the BubbleSort algorithm means determining the number of comparisons required to sort a list
- Recall that BubbleSort works by repeatedly bubbling up the largest element in the unsorted part of the list
- We can determine the number of comparisons by carefully analyzing the BubbleSort example we worked through earlier, when we sorted (9 2 8 4 1 3)
    - But we need to generalize from this example, so our analysis holds for all examples

# Analysis of the BubbleSort algorithm (cont'd)

- If we apply BubbleSort to (9 2 8 4 1 3) how many comparisons do we do each iteration?

# Analysis of the BubbleSort algorithm (cont'd)

- If we apply BubbleSort to (9 2 8 4 1 3) how many comparisons do we do each iteration?
  - On iteration 1 we do 5 comparisons (6 unsorted numbers)

# Analysis of the BubbleSort algorithm (cont'd)

- If we apply BubbleSort to (9 2 8 4 1 3) how many comparisons do we do each iteration?
  - On iteration 1 we do 5 comparisons (6 unsorted numbers)
  - On iteration 2 we do 4 comparisons (5 unsorted numbers)

# Analysis of the BubbleSort algorithm (cont'd)

- If we apply BubbleSort to (9 2 8 4 1 3) how many comparisons do we do each iteration?
  - On iteration 1 we do 5 comparisons (6 unsorted numbers)
  - On iteration 2 we do 4 comparisons (5 unsorted numbers)
  - On iteration 3 we do 3 comparisons (4 unsorted numbers)

# Analysis of the BubbleSort algorithm (cont'd)

- If we apply BubbleSort to (9 2 8 4 1 3) how many comparisons do we do each iteration?
  - On iteration 1 we do 5 comparisons (6 unsorted numbers)
  - On iteration 2 we do 4 comparisons (5 unsorted numbers)
  - On iteration 3 we do 3 comparisons (4 unsorted numbers)
  - On iteration 4 we do 2 comparisons (3 unsorted numbers)

# Analysis of the BubbleSort algorithm (cont'd)

- If we apply BubbleSort to (9 2 8 4 1 3) how many comparisons do we do each iteration?
  - On iteration 1 we do 5 comparisons (6 unsorted numbers)
  - On iteration 2 we do 4 comparisons (5 unsorted numbers)
  - On iteration 3 we do 3 comparisons (4 unsorted numbers)
  - On iteration 4 we do 2 comparisons (3 unsorted numbers)
  - On iteration 5 we do 1 comparison (2 unsorted numbers)

# Analysis of the BubbleSort algorithm (cont'd)

- If we apply BubbleSort to (9 2 8 4 1 3) how many comparisons do we do each iteration?
  - On iteration 1 we do 5 comparisons (6 unsorted numbers)
  - On iteration 2 we do 4 comparisons (5 unsorted numbers)
  - On iteration 3 we do 3 comparisons (4 unsorted numbers)
  - On iteration 4 we do 2 comparisons (3 unsorted numbers)
  - On iteration 5 we do 1 comparison (2 unsorted numbers)
  - On iteration 6 we do 0 comparisons (1 unsorted number)

# Analysis of the BubbleSort algorithm (cont'd)

- If we apply BubbleSort to (9 2 8 4 1 3) how many comparisons do we do each iteration?
  - On iteration 1 we do 5 comparisons (6 unsorted numbers)
  - On iteration 2 we do 4 comparisons (5 unsorted numbers)
  - On iteration 3 we do 3 comparisons (4 unsorted numbers)
  - On iteration 4 we do 2 comparisons (3 unsorted numbers)
  - On iteration 5 we do 1 comparison (2 unsorted numbers)
  - On iteration 6 we do 0 comparisons (1 unsorted number)
- So how many total comparisons for a list with 6 items?

# Analysis of the BubbleSort algorithm (cont'd)

- If we apply BubbleSort to (9 2 8 4 1 3) how many comparisons do we do each iteration?
  - On iteration 1 we do 5 comparisons (6 unsorted numbers)
  - On iteration 2 we do 4 comparisons (5 unsorted numbers)
  - On iteration 3 we do 3 comparisons (4 unsorted numbers)
  - On iteration 4 we do 2 comparisons (3 unsorted numbers)
  - On iteration 5 we do 1 comparison (2 unsorted numbers)
  - On iteration 6 we do 0 comparisons (1 unsorted number)
- So how many total comparisons for a list with 6 items?
  - Number of comparisons = $5 + 4 + 3 + 2 + 1 = 15$

# Analysis of the BubbleSort algorithm (cont'd)

- If we apply BubbleSort to (9 2 8 4 1 3) how many comparisons do we do each iteration?
  - On iteration 1 we do 5 comparisons (6 unsorted numbers)
  - On iteration 2 we do 4 comparisons (5 unsorted numbers)
  - On iteration 3 we do 3 comparisons (4 unsorted numbers)
  - On iteration 4 we do 2 comparisons (3 unsorted numbers)
  - On iteration 5 we do 1 comparison (2 unsorted numbers)
  - On iteration 6 we do 0 comparisons (1 unsorted number)
- So how many total comparisons for a list with 6 items?
  - Number of comparisons = $5 + 4 + 3 + 2 + 1 = 15$
- So how many comparisons for a list with $n$ items?

# Analysis of the BubbleSort algorithm (cont'd)

- If we apply BubbleSort to (9 2 8 4 1 3) how many comparisons do we do each iteration?
  - On iteration 1 we do 5 comparisons (6 unsorted numbers)
  - On iteration 2 we do 4 comparisons (5 unsorted numbers)
  - On iteration 3 we do 3 comparisons (4 unsorted numbers)
  - On iteration 4 we do 2 comparisons (3 unsorted numbers)
  - On iteration 5 we do 1 comparison (2 unsorted numbers)
  - On iteration 6 we do 0 comparisons (1 unsorted number)
- So how many total comparisons for a list with 6 items?
  - Number of comparisons = $5 + 4 + 3 + 2 + 1 = 15$
- So how many comparisons for a list with $n$ items?
  - $(n-1) + (n-2) + \cdots + 2 + 1$, or

$$\sum_{i=1}^{n-1} i$$

# Analysis of the BubbleSort algorithm (cont'd)

- We want to know how the number of operations grows with $n$

# Analysis of the BubbleSort algorithm (cont'd)

- We want to know how the number of operations grows with $n$
- This is not obvious with the summation so we need to replace it with a closed formula

# Analysis of the BubbleSort algorithm (cont'd)

- We want to know how the number of operations grows with $n$
- This is not obvious with the summation so we need to replace it with a closed formula
  - We can do this since it is known that

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

# Analysis of the BubbleSort algorithm (cont'd)

- We want to know how the number of operations grows with $n$
- This is not obvious with the summation so we need to replace it with a closed formula
  - We can do this since it is known that

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

  - This was proven in the section on induction but is also based on the sum of $n$ values equaling $n$ times the average value

# Analysis of the BubbleSort algorithm (cont'd)

- We want to know how the number of operations grows with $n$
- This is not obvious with the summation so we need to replace it with a closed formula
  - We can do this since it is known that

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

  - This was proven in the section on induction but is also based on the sum of $n$ values equaling $n$ times the average value
    - The average value of $1, 2, \ldots, n$ is $\frac{1}{2}(n+1)$

# Analysis of the BubbleSort algorithm (cont'd)

- We want to know how the number of operations grows with $n$
- This is not obvious with the summation so we need to replace it with a closed formula
  - We can do this since it is known that

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

  - This was proven in the section on induction but is also based on the sum of $n$ values equaling $n$ times the average value
    - The average value of $1, 2, \ldots, n$ is $\frac{1}{2}(n+1)$
  - In this case, we are summing up to $n-1$ and not $n$, so substituting $n-1$ for $n$ we get:

# Analysis of the BubbleSort algorithm (cont'd)

- We want to know how the number of operations grows with $n$
- This is not obvious with the summation so we need to replace it with a closed formula
  - We can do this since it is known that

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

  - This was proven in the section on induction but is also based on the sum of $n$ values equaling $n$ times the average value
    - The average value of $1, 2, \ldots, n$ is $\frac{1}{2}(n+1)$
  - In this case, we are summing up to $n-1$ and not $n$, so substituting $n-1$ for $n$ we get:
    - Number BubbleSort comparisons $= \frac{1}{2}(n-1)n = \frac{1}{2}(n^2 - n)$

# Analysis of the BubbleSort algorithm (cont'd)

- So BubbleSort requires $\frac{1}{2}(n^2 - n)$ comparisons

# Analysis of the BubbleSort algorithm (cont'd)

- So BubbleSort requires $\frac{1}{2}(n^2 - n)$ comparisons
- Computer scientists usually focus on the highest order term, so we say that the number of comparisons in bubblesort grows as $n^2$ or as the square of the length of the list

# Analysis of the BubbleSort algorithm (cont'd)

- So BubbleSort requires $\frac{1}{2}(n^2 - n)$ comparisons
- Computer scientists usually focus on the highest order term, so we say that the number of comparisons in bubblesort grows as $n^2$ or as the square of the length of the list
- BubbleSort can have problems if the list is very long

# Analysis of the MergeSort algorithm

- Let $a_n$ be the worst-case number of comparisons used by MergeSort algorithm when sorting a set of size $n$.

# Analysis of the MergeSort algorithm

- Let $a_n$ be the worst-case number of comparisons used by MergeSort algorithm when sorting a set of size $n$.
- MergeSort consists of the following steps when sorting an $n$-element set:

# Analysis of the MergeSort algorithm

- Let $a_n$ be the worst-case number of comparisons used by MergeSort algorithm when sorting a set of size $n$.
- MergeSort consists of the following steps when sorting an $n$-element set:
  - Sort left half, at cost $a_{n/2}$

# Analysis of the MergeSort algorithm

- Let $a_n$ be the worst-case number of comparisons used by MergeSort algorithm when sorting a set of size $n$.
- MergeSort consists of the following steps when sorting an $n$-element set:
  - Sort left half, at cost $a_{n/2}$
  - Sort right half, at cost $a_{n/2}$

# Analysis of the MergeSort algorithm

- Let $a_n$ be the worst-case number of comparisons used by MergeSort algorithm when sorting a set of size $n$.
- MergeSort consists of the following steps when sorting an $n$-element set:
  - Sort left half, at cost $a_{n/2}$
  - Sort right half, at cost $a_{n/2}$
  - Merge the two halves, at cost at most $n-1$.

# Analysis of the MergeSort algorithm

- Let $a_n$ be the worst-case number of comparisons used by MergeSort algorithm when sorting a set of size $n$.
- MergeSort consists of the following steps when sorting an $n$-element set:
  - Sort left half, at cost $a_{n/2}$
  - Sort right half, at cost $a_{n/2}$
  - Merge the two halves, at cost at most $n-1$.
- So

$$a_n = \begin{cases} 2a_{n/2} + n - 1 & \text{if } n \text{ is even and } n \geq 2 \\ 0 & \text{if } n = 1 \end{cases}$$

- By induction, can prove

$$a_n = n \log_2 n - n + 1 \qquad \text{if } n \text{ is a power of 2.}$$

  (See next slide.)
- As we saw earlier, $n\,log_2 n$ grows much more slowly than $n^2$, so no one would ever use bubblesort unless the lengths of the lists are guaranteed to be small

# Analysis of the MergeSort algorithm (cont'd)

**Theorem**

*Let*

$$a_n = \begin{cases} 2a_{n/2} + n - 1 & \text{if } n \text{ is even and } n \geq 2 \\ 0 & \text{if } n = 1 \end{cases}$$

*Then*

$$a_n = n \log_2 n - n + 1 \qquad \text{if } n \text{ is a power of 2.}$$

# Analysis of the MergeSort algorithm (cont'd)

**Theorem**

*Let*

$$a_n = \begin{cases} 2a_{n/2} + n - 1 & \text{if } n \text{ is even and } n \geq 2 \\ 0 & \text{if } n = 1 \end{cases}$$

*Then*

$$a_n = n \log_2 n - n + 1 \qquad \text{if } n \text{ is a power of 2.}$$

**Proof.**

Let $n = 2^k$ and let $b_k = a_{2^k} = a_n$. We then have

$$b_k = \begin{cases} 2b_{k-1} + 2^k - 1 & \text{if } k \geq 2, \\ 0 & \text{if } k = 0 \end{cases}$$

and we want to show that

$$b_k = 2^k \cdot k - 2^k + 1 \qquad \text{for } k \in \mathbb{N}$$

**Proof (cont'd).**

We're trying to show that if

$$b_k = \begin{cases} 2b_{k-1} + 2^k - 1 & \text{if } k \geq 1, \\ 0 & \text{if } k = 0 \end{cases}$$

then

$$b_k = 2^k \cdot k - 2^k + 1 \qquad \text{for } k \in \mathbb{N}$$

# Analysis of MergeSort (cont'd)

**Proof (cont'd).**

We're trying to show that if

$$b_k = \begin{cases} 2b_{k-1} + 2^k - 1 & \text{if } k \geq 1, \\ 0 & \text{if } k = 0 \end{cases}$$

then

$$b_k = 2^k \cdot k - 2^k + 1 \qquad \text{for } k \in \mathbb{N}$$

Use induction on $k$.

# Analysis of MergeSort (cont'd)

**Proof (cont'd).**

We're trying to show that if

$$b_k = \begin{cases} 2b_{k-1} + 2^k - 1 & \text{if } k \geq 1, \\ 0 & \text{if } k = 0 \end{cases}$$

then

$$b_k = 2^k \cdot k - 2^k + 1 \qquad \text{for } k \in \mathbb{N}$$

Use induction on $k$.

**Base case:** Let $k = 0$. Then

$$b_0 = 0$$
$$2^0 \cdot 0 - 2^0 + 1 = 0 - 1 + 1 = 0$$

and so the formula is okay when $k = 0$.

**Proof (cont'd).**

**Induction step:** Let $m \in \mathbb{Z}^+$ and suppose that our formula is true when $k = m - 1$. Then

$$b_{m-1} = 2^{m-1} \cdot (m-1) - 2^{m-1} + 1$$

We then have

$$\begin{aligned} b_m &= 2b_{m-1} + 2^m - 1 \qquad \text{by recurrence relation} \\ &= 2\left(2^{m-1} \cdot (m-1) - 2^{m-1} + 1\right) + 2^m - 1 \\ &= 2^m \cdot m - 2^m + 1 = b_m, \end{aligned}$$

as required. □

# Big-O notation

- Sometimes the exact formula for the operation count is too unwieldy:
  - The number of operations may depend on extraneous factors (programmer skill, compiler quality, system load, etc.).
  - It may be too difficult to obtain this formula.
  - The formula may be too complicated, thereby obscuring what's really going on.
- Real question: How fast does number of operations increase with input size $n$, as $n$ gets large?
- We seek an *asymptotic upper bound*, given by $O$-notation ("Big-$O$ notation").

# Big-O notation (cont'd)

Main ideas is to get an upper bound on operation count, as follows:

- Hide low-order terms. So $n^2 + n = O(n^2)$.
- Hide multiplicative constants. So $3n^2 = O(n^2)$.

# Big-O notation (cont'd)

Main ideas is to get an upper bound on operation count, as follows:

- Hide low-order terms. So $n^2 + n = O(n^2)$.
- Hide multiplicative constants. So $3n^2 = O(n^2)$.

Some examples:

- $3n^2 + 2n = O(n^2)$.
- $2n^4 + n =$

# Big-O notation (cont'd)

Main ideas is to get an upper bound on operation count, as follows:

- Hide low-order terms. So $n^2 + n = O(n^2)$.
- Hide multiplicative constants. So $3n^2 = O(n^2)$.

Some examples:

- $3n^2 + 2n = O(n^2)$.
- $2n^4 + n = O(n^4)$.
- $2^n + n^{42} =$

# Big-O notation (cont'd)

Main ideas is to get an upper bound on operation count, as follows:

- Hide low-order terms. So $n^2 + n = O(n^2)$.
- Hide multiplicative constants. So $3n^2 = O(n^2)$.

Some examples:

- $3n^2 + 2n = O(n^2)$.
- $2n^4 + n = O(n^4)$.
- $2^n + n^{42} = O(2^n)$.

# Big-O notation (cont'd)

Main ideas is to get an upper bound on operation count, as follows:

- Hide low-order terms. So $n^2 + n = O(n^2)$.
- Hide multiplicative constants. So $3n^2 = O(n^2)$.

Some examples:

- $3n^2 + 2n = O(n^2)$.
- $2n^4 + n = O(n^4)$.
- $2^n + n^{42} = O(2^n)$.

But also note that $2n^4 + n = O(n^{10})$; the $O$-notation only provides an upper bound.

# Big-O notation (cont'd)

- It's reasonable to drop lower-order terms.

# Big-O notation (cont'd)

- It's reasonable to drop lower-order terms.
- Why drop multiplicative constants?
  - We often care more about rate at which function grows than the exact amount of time algorithm takes for input of given size.
  - Example: Let $f_1(n) = 1000n^2$ and $f_2(n) = n^3$. Then $f_1(n) = O(n^2)$ and $f_2 = O(n^3)$. So $f_1$ is asymptotically better than $f_2$, even though $f_2(n) \le f_1(n)$ when $n \le 1000$.

- It's reasonable to drop lower-order terms.
- Why drop multiplicative constants?
  - We often care more about rate at which function grows than the exact amount of time algorithm takes for input of given size.
  - Example: Let $f_1(n) = 1000n^2$ and $f_2(n) = n^3$. Then $f_1(n) = O(n^2)$ and $f_2 = O(n^3)$. So $f_1$ is asymptotically better than $f_2$, even though $f_2(n) \le f_1(n)$ when $n \le 1000$.
- Asymptotic complexity of some algorithms we've considered:

| Algorithm | Time Complexity |
|---|---|
| linear search | $O(n)$ |
| binary search | $O(\log_2 n)$ |
| insertion sort | $O(n^2)$ |
| bubblesort | $O(n^2)$ |
| mergesort | $O(n \log_2 n)$ |

# Big-O notation (cont'd)

**Definition**

Let $f, g\ \mathbb{N} \to \mathbb{R}^{\geq 0}$. We say that $f(n) = O\big(g(n)\big)$ (as $n \to \infty$) iff there exist constants $c \geq 0$ and $n_0 \in \mathbb{N}$ such that

$$0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\,.$$

**Definition**

Let $f, g\ \mathbb{N} \to \mathbb{R}^{\geq 0}$. We say that $f(n) = O\big(g(n)\big)$ (as $n \to \infty$) iff there exist constants $c \geq 0$ and $n_0 \in \mathbb{N}$ such that

$$0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 .$$

**Example**

Let $f(n) = 2n^4 + 7n^3 + 100n^2 \log_2 n$. Prove that $f(n) = O(n^4)$.

**Solution**

*Note that for $n \geq 1$, we have*

$$0 \leq f(n) \leq 2n^4 + 7n^4 + 100n^4 = 114n^4.$$

*So the $O$-condition holds with $n_0 = 1$ and $c = 114$.*