Algorithms

X. Zhang
Fordham Univ.

Real World applications of algorithms

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

Example of algorithms

- Algorithms for set operations
  - Union: take two sets A and B as input, and generate $A \cup B$ as output
  - Intersection, Difference, Cartesian product,
- Data structures and algorithms that operate on them
  - Data structure: set, list, tree, graph are widely used in computer system for storing information
  - Algorithms for these data structure are critical for most computer system
    - merge two sets,
    - sort a list,
    - search in a tree,
    - finding shortest path in a graph,
  - a CS course is devoted to data structure

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 sequence of well-defined instructions for completing a task with a finite amount of effort in a finite amount of time
- a sequence of instructions that can be mechanically performed in order to solve a problem

Key aspects of an algorithm

- An algorithm must be precise
  - clear and detailed enough for someone (or something) to execute it
  - One way to ensure:
    - use actual computer code, which is guaranteed to be unambiguous
    - pseudocode is often used, readable by humans
  - We will use English-like pseudocode
    - With some special notations…

Key aspects of an algorithm

- An algorithm operates on input and generates output
  - E.g., The “looking up a name in phonebook” algorithm has two inputs: the phone book and the name to look up; generates one output: the phone number
  - E.g., Input to FindMax algorithm: a list of numbers; output is the maximum value in the list
- An algorithm completes in a finite number of steps
  - This is a non-trivial requirement since certain methods may sometimes run forever!

Algorithms and Computers

- Algorithms have been used for thousands of years and have been executed by humans (possibly with pencil and paper)
  - Algorithm for performing long division
  - Algorithm for conversion between different base numeral systems
- Work on algorithms exploded with development of 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
  - Search for better and more efficient algorithms continues
- Interestingly enough, some problems have been shown to have no algorithmic solution

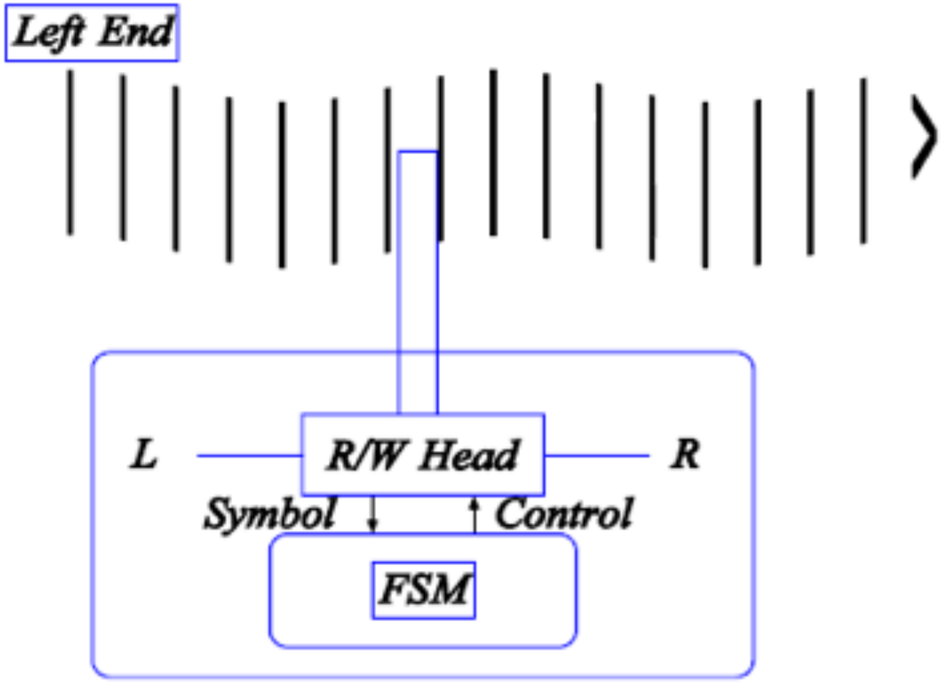
Halting Problem

- Halting Problem: given a description of a computer program and input to the program, decide whether the program finishes running or continues to run forever.
- Alan Turing proved in 1936: a general algorithm to solve the halting problem for *all* possible program-input pairs cannot exist.
  - a mathematical definition of a computer and program, what became known as a Turing machine;
  - the halting problem is *undecidable* over Turing machines
- Turing (a novel about computation) by Christos H. Papadimitriou, CS Professor at UC Berkeley

Uncomputable problem

- Alan Turing (1912-1954)
  - English mathematician, logician, cryptanalyst, and computer scientist
  - *Turing, A. M., "On Computable Numbers, with an Application to the Entscheidungsproblem", Proceedings of the London Mathematical Society, Series 2, 42:230-265 and 43:544-546, 1937.*

Turing Machine



Although simple, one can simulate a general computer using a TM

Universal Turing Machine

- A Turing machine that is able to simulate any other Turing machine is called a universal Turing machine
  - Read the description of the TM to be simulated from the tape …
- This is similar to a general-purpose computer
  - CPU reads the program (Word, Internet Explorer, PowerPoint, MediaPlayer …) from the disk, and carries out the instructions specified in the program line by line …

Stored-program computer

- **Also called von Neumann architecture**
  - named after mathematician and early computer scientist John von Neumann (12/28/1903 – 2/8/1957), “the last of the great mathematicians”, “
  - Central processing unit (CPU): capable of performing arithmetic operations, read & write memory, branch operations, …
  - Memory: stores both instructions and data
- Such architecture makes computer a general purpose machine => one can write diff programs to make computer do diff tasks

Now to more easy topics

Searching and Sorting Algorithms

- Two of the most studied classes of algorithms in CS:
  - 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
  - E.g. phone book
- Searching and sorting algorithms as introduction to the topic of algorithms

Searching Algorithms

- Problem: determine if an element x is in a list L
- We will look at two simple searching algorithms
  - Linear search
  - Binary search
- List: elements stored in a list in a sequential way
  - There is a first element, second element, ..
  - To make life easier: we use L[i] or L_i to refer to the i-th element in list L, we refer i as the index of element L_i
  - $L = (l_1, l_2, .., l_n)$
  - Elements are not necessarily ordered

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-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 “ FOUND"
```

- Note:
  - Repeat: means do step 2 for i=1, i=2, i=3,…i=n
  - We indent line 2 to show that it’s part of the loop/iteration
  - Return: means exits the algorithm and returns the output

Efficiency of Linear Search Algorithm

- If x appears once in L, on average how many comparisons (line 2) would the algorithm to make on average?
  - On average n/2 comparisons
- If x does not appear in L, how many comparisons would the algorithm make?
  - n comparisons
- Would such an algorithm be useful for finding someone in a large (unsorted) phone book?
  - No, it would require scanning through entire phone book!
  - Need a better way!

Binary Search Algorithm Overview

- Binary search algorithm assumes that L is sorted
  - Ascending order or descending order
- This algorithm need not examine each element, it maintains a “window" in which element x may reside
  - window is defined by indices min and max which specify the leftmost and rightmost boundaries in L
  - In the beginning, x can be anyway in L, i.e., min=1, max=n
  - At each iteration of the algorithm, the window is cut in half
    - Remember number guessing game ?
    - I am thinking about the number between 1 and 100, you guess it by asking question such as “Is the number larger than 30”?

Binary Search Algorithm

- Binary Search Algorithm assuming L has been sorted in ascending order

```
1 set min to 1 and set max to  n
2 Repeat until min > max
3       Set midpoint to (min + max)/2
4       Compare x to L[midpoint], three possible results:
          (a) if x = L[midpoint] then return “FOUND"
          (b) if x > L[midpoint] then set min to (midpoint + 1)
          (c) if x < L[midpoint] then set max to (midpoint -1)
5 return  “FOUND"
```

- Note: the repeat loop spans lines 2-4.
- Can you modify the algorithm to work for L sorted in descending order?

Binary Search Example

- Use binary search to find element “4" in sorted list (1 3 4 5 6 7 8 9). List values of min, max and midpoint after each iteration of step 4. How many values are compared to “4"?
  1 Min = 1 and max = 8 and midpoint = 1/2 (1 + 8) = 4 (round down). Since L[4] = 5 and since 4 < 5 we execute step 4c and max = midpoint - 1 = 3.
  2 Now min = 1, max = 3 and midpoint = 1/2 (1 + 3) = 2. Since L[2] = 3 and 4 > 3, we execute step 4b and min = midpoint + 1 = 3.
  3 Now min = 3, max = 3 and midpoint = 1/2 (3 + 3) = 3. Since L[3] = 4 and 4 = 4, we execute step 4a and return “FOUND.“
- we check three values: 3, 4, and 5.
- Since we cut the window in half each iteration, it will shrink very quickly (about $\log_2 n$ comparisons).

Analysis of Algorithms

- An algorithm is a set of instructions that solves a problem for all possible input instances
- There may be many algorithms solving one problem and all of these are not equally good
  - 12 sorting algorithms described in Wikipedia
- One criteria for evaluating an algorithm is efficiency
  - Of course, correctness is first consideration
- Analysis of Algorithm: determining the efficiency of an algorithm

What's in an algorithm?

- Consider this problem: find the largest number in a list of numbers, given by L, i.e., (L_1, L_2, …, L_n)

- How would you solve the problem?

- How to specify your solution?

Algorithm analysis

How to evaluate algorithms?

- When solving tasks, what are we most concerned with?
  - Most of us are pretty concerned with time, and time is actually the main concern in evaluating the efficiency of algorithms
  - Space: maximum amount of memory the algorithm requires at any time
  - There is a trade-off between time and space efficiency
- We will focus on time, although for some problems, space can actually be the main concern.

How to measure time efficiency?

- 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, running time is usually impacted by the specific input, so how do we handle that?

Run Time Complexity

- Standard solution: number of operations performed by the algorithm w.r.t. the size of the input
  - Size of the input: the length of the list to be sorted/ searched, the number of nodes/edges in the graph, …
- Inputs of same size sometimes results in different numbers of operations
  - E.g., linear search, 1 v.s. n
  - focus on worst-case performance, i.e., assume hardest input possible (most unlucky case)
  - E.g., worst case input for linear search is when item to be searched is not in the list or last element in the list

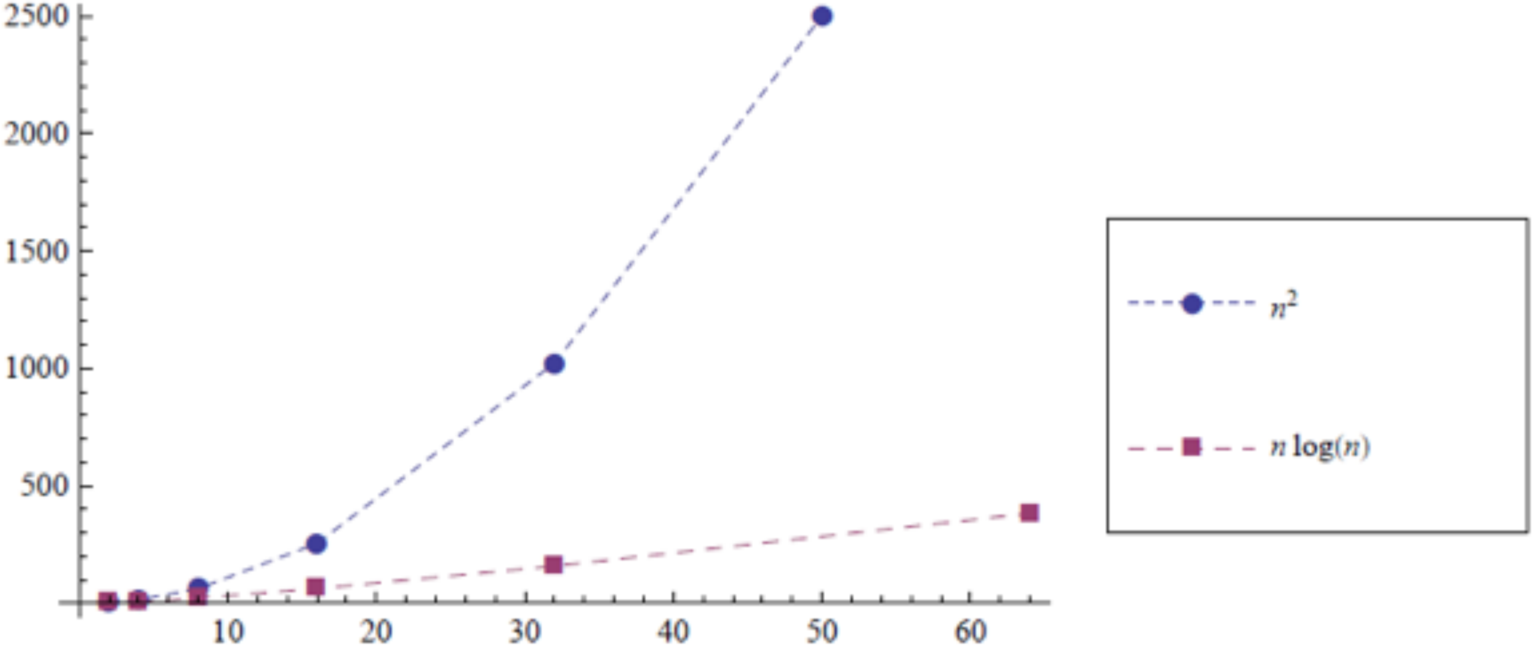
Running time of BubbleSort and MergeSort

- One way to find out number of operations:
  - implement the algorithm as a computer program (which also record # of operations)
  - run program on inputs of various length
  - record # of operations performed and find out worst-case, average-case, …
- E.g.: bubblesortOps(n) and mergesortOps(n) represent avg # of operations performed to sort list with n elements

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

Run Time Complexity

- From the data, we can determine closed formulas for bubblesortOps(n) and mergesortOps(n)
  - bubblesortOps(n) = n^2
  - mergesortOps(n) = n log2 n



Analysis of Linear Search Algorithm

- Linear Search Algorithm

```
repeat as i varies from 1 to n
    if L[i ] = x then return "FOUND" and stop
return "FOUND"
```

- How many comparison operations does it perform?
- The algorithm checks at most n elements against x,
  - worst-case: requires n comparisions.
  - This occurs 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 in 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

- 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 log2n comparisons
- If n is not a power of 2, you will need to round up the number of comparisons
  - i.e., it requires $\lceil \log_2 n \rceil$ comparisons
  - Thus if there are 3 elements it may require 3 comparisons

Linear Search vs Binary Search

- linear search: requires n comparisons worst case
- binary search: requires log2n comparisions worst case
- Which one is faster? Is the difference significant?
  - binary search algorithm is much faster, in that it requires many fewer comparisons
  - If a list has 1 million elements,
    - linear search requires 1,000,000 comparisons
    - binary search requires only about 20 comparisons!
- But binary search requires list to be sorted first
  - sorting requires nlog2n operations, 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.

Sorting algorithm

Sorting Algorithms

- Sorting algorithms are one of the most heavily studied topics in Computer Science
  - Sorting is critical to improve searching efficiency
- There are many well known sorting algorithms in Computer Science, we focus on two:
  - BubbleSort: a very simple but inefficient sorting algorithm
  - MergeSort: a slightly more complex but efficient sorting algorithm

BubbleSort Algorithm Overview

- BubbleSort: repeatedly scan the list, in each iteration “bubbles" largest element in unsorted part of the list to the end, e.g., for list 9 2 8 4 1 3
  - After 1 iteration, largest element in last position, 2 8 4 1 3 9
  - After 2 iterations, largest element in last position and second largest element in second to last position, 2 4 1 3 8 9
  - 3rd: 2 1 3 4 8 9
  - 4th: 1 2 3 4 8 9
  - 5-th iteration: 1 2 3 4 8 9 (done!)
- requires n-1 iterations
  - at (n-1)-th iteration, only one item left, must already be in proper position (i.e., the smallest must be in the leftmost position)

BubbleSort Algorithm

- Input: n-element list L = $(l_1, l_2, .., l_n)$
- Bublesort 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 $l_{j+1}$
```

- i controls which part of list is checked each iteration. (Only unsorted part is checked.)
  - In 1st iteration, check everything, $l_1, l_2, \ldots l_{n-1}$
  - In 2nd iteration, check everything except last element, $l_1, l_2, \ldots, l_{n-2}$
  - …
- Inner loop (2-3): bubble up largest element in unsorted part of list

BubbleSort Example

- Use BubbleSort to sort list of number (9 2 8 4 1 3) into increasing order.
- 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
  - it divides the problem into smaller problems
  - solves the smaller problems
  - then combines solutions to smaller problems, to find solution to original problem
- Much more efficient than bubblesort algorithm
- Key: combining two sorted lists into a sorted list is very easy
  - How would you combine (1 4 7 8) and (2 5 9 10 11)?
  - place your fingers at the start of each list, copy over the smaller element, then advance that one finger.
  - Above description is not mechanical enough … What if no where to advance the finger? When to stop?

MergeSort Algorithm

- MergeSort Algorithm
- function mergesort(L)

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

  - Note: l_1 = mergesort(left half of L) means:
  - set the result of mergesort (left half of L) to list l_1
  - We have intuitively solved merge(l_1,l_2) in last slide, can you write out its algorithm?

Description of MergeSort

- MergeSort is a recursive function
  - That means it calls itself
- If 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
- Mergesort completely sort left half of the list before it starts sorting the right half

Example of MergeSort

- Trace mergesort with input (9 2 8 4 1 3)

Analysis of BubbleSort

- Analyzing 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 BubbleSort

- 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?

$$(n-1)+(n-2)+\ldots+3+2+1$$

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

Analysis of BubbleSort

- 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
- 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; : : : ; n is 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 =

Analysis of BubbleSort

- So BubbleSort requires 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 MergeSort

- Analysis of mergesort
- number of comparisons grows proportional to n log2n
  - n log2n grows much more slowly than n^2
  - so do not use bubblesort unless for a very short list