CPSC 131
Data Structures Concepts

Dr. Anand Panangadan
apanangadan@fullerton.edu

Goals for today

- Experimental analysis
- Asymptotic analysis

Key terms

- Key terms
  - Experimental analysis
  - Asymptotic analysis
  - Worst-case analysis
  - Big-Oh notation
  - Constant time operations O(1)
  - O(n) operations

Comparing data structures

- Is one program/function/data structure *better* than another?
- Two approaches to answering this question:
  1. Experimental analysis
  2. Asymptotic analysis

Experimental Analysis

- Implement the two programs
- Implement a main function that loads both with same data
- Call the same data structure operations on both
  - Insert elements
  - Delete elements …
- Measure time spent by each data structure

Experimental Analysis

- Problems
  - Can do experiments only a limited data set
  - Other factors impact running time:
    - What other programs are running on the computer
  - Does the running time depend on the computer itself?
    - Do the results hold on a different computer?
  - Does the running time depend on the implementation?
    - Do the results hold on a different programming language?

Asymptotic Analysis

- Analysis without actually running any code
- **Asymptotic:** approaching a value closely
- Key idea:
  - We are interested in running time for large data sets
  - How *fast* will running time increase as we increase data size?
    - Rate of increase is more important than the actual time

Asymptotic Analysis

- Represent by *n* the most important factor
  - For data structures, *n* is the number of elements in the data structure
- Analysis
  - How does running time increase in terms of *n*?
  - Don’t care about *constant factors*
    - Focus on the big picture
      - Not details like initialization
  - Only consider the worst-case

Example: printing within a loop

- Consider the function:

```
void print(int n) {
   for (int i=0; i < n; ++i) {
       cout << i << endl;
   }
}
```

- How many steps did we have to do?
  - n, 2n (there are two things being printed), 3n, 3n+2, …?
  - Too complex!
- Do the number of steps increase *in proportion to* n?
  - Yes!

Example: printing within a loop

- Number of operations increases in proportion to n
- "function print takes time on the order of n"
- Written as O(n)
- So also commonly spoken as "Oh of n"

Printing a linked list

- But what about the fact that we had to
  - initialize the loop
  - Printing required cout
  - We also printed endl
  - …
- Don't care about *constant factors*
  - Focus on the big picture
    - Not details like initialization

Classes of functions

Number of steps	Big-O class	
1 5 1000	O(1)	Constant time Does not depend on n The best possible
n 2n 1000n + 50	O(n)	Proportional to n
n^2 $5n^2$ $5n^2 + 10n + 50$	O(n^2)	Increases quadratically with n Much worse than O(n) Nested loops take this time

Comparing

- Arrays
- Singly linked lists
- Doubly linked lists

Comparing

- Operations
  - Create empty
  - Get front element
  - Add/ remove front element
  - Get back element
  - Add/ remove back element
  - Clear data structure
  - Get/add/remove i^{th} element

What about memory requirement?

- So far, we have been speaking of running time
- What about how much memory/space is occupied by a data structure?
- Can also use O(n) concept:
  - Does memory usage go up proportionately to number of elements?

CPSC 131
Data Structures

Dr. Shilpa Lakhanpal
shlakhanpal@fullerton.edu

Analyzing an algorithm

- Experimental analysis
- Asymptotic analysis

Key terms, when analyzing an algorithm

- Key terms
  - Experimental analysis
  - Asymptotic analysis
  - Worst-case analysis
  - Big-O notation

Comparing data structures

- Is one program/function/data structure *better* than another?
- Two approaches to answering this question:
  1. Experimental analysis
  2. Asymptotic analysis

Experimental Analysis

- Implement the two programs
- Implement a main function that loads both with same data
- Call the same data structure operations on both
  - Insert elements
  - Delete elements …
- Measure time spent by each data structure

Experimental Analysis

- Problems
  - Can do experiments only a limited data set
  - Other factors impact running time:
    - What other programs are running on the computer
  - Does the running time depend on the computer itself?
    - Are the results same on a different computer?
  - Does the running time depend on the implementation?
    - Are the results same for a different programming language?

Asymptotic Analysis

- Analysis without actually running any code
- **Asymptotic:** approaching a value closely
- Key idea:
  - We are interested in running time for large data sets
  - How *fast* will running time increase as we increase data size?
    - Rate of increase is more important than the actual time

Asymptotic Analysis

- Represent the most important factor by *n*
  - For data structures, *n* is the number of elements in the data structure
- Analysis
  - How does running time increase in terms of *n*?
  - Don’t care about *constant factors*
    - Focus on the big picture
      - Not details like initialization
  - Only consider the worst-case

Example: printing within a loop

- Consider the function:

```
void print(int n) {
   for (int i=0; i < n; ++i) {
       cout << i << endl;
   }
}
```

- How many steps did we have to do?
  - n, 2n (are there two things being printed ?), 3n, 3n+2, …?
  - Too complex!
- Do the number of steps increase *in proportion to* n?
  - Yes!

Example: printing within a loop

- Number of operations increases in proportion to n
- "function print takes time on the order of n"
- Written as O(n)
- So also commonly spoken as "Oh of n"

Example: printing within a loop

- But what about the fact that we had to
  - initialize the loop
  - Printing required cout
  - We also printed endl
  - …
- Don’t care about *constant factors*
  - Focus on the big picture
    - Not details like initialization

Classes of functions

Number of steps	Big-O class	
1 5 1000	O(1)	Constant time Does not depend on n The best possible
n 2n 1000n + 50	O(n)	Proportional to n
n^2 $5n^2$ $5n^2 + 10n + 50$	$O(n^2)$	Increases quadratically with n Much worse than O(n) Nested loops take this time

What about memory requirement?

- So far, we have been speaking of running time
- What about how much memory/space is occupied by a data structure?
- Can also use O(n) concept:
  - Does memory usage go up proportionately to number of elements?

What we will learn today

- Linear search
- Binary search
- Constant time operations
- Big O notation
- Asymptotic notation
- Algorithm analysis

References

- CSUF CPSC 131 Slides: Algorithm analysis, Dr. Anand Panangadan

CPSC 131
Data Structures

Dr. Shilpa Lakhanpal
shlakhanpal@fullerton.edu

Algorithm Analysis

An algorithm with runtime complexity T(N) has a lower bound and an upper bound.

Lower bound

A function f(N) that is ≤ the best case T(N), for all values of N ≥ 1.

Upper bound

A function f(N) that is ≥ the worst case T(N), for all values of N ≥ 1.

Asymptotic notation

Asymptotic notation is the classification of runtime complexity that uses functions that indicate only the growth rate of a bounding function.

3 Asymptotic notations:

O notation

O notation provides a growth rate for an algorithm's upper bound.

Ω notation

Ω notation provides a growth rate for an algorithm's lower bound.

Θ notation

Θ notation provides a growth rate that is both an upper and lower bound.

Table 4.4.1: Notations for algorithm complexity analysis.

Notation	General form	Meaning
O	$T(N) = O(f(N))$	A positive constant c exists such that, for all N ≥ 1, $T(N) \leq c * f(N)$.
Ω	$T(N) = \Omega(f(N))$	A positive constant c exists such that, for all N ≥ 1, $T(N) \geq c * f(N)$.
Θ	$T(N) = \Theta(f(N))$	$T(N) = O(f(N))$ and $T(N) = \Omega(f(N))$.

To analyze how runtime of an algorithm scales as the input size increases:

1) First determine how many operations the algorithm executes for a specific input size, N.
2) Then, the big-O notation for that function is determined.

Algorithm runtime analysis often focuses on the **worst-case runtime** complexity

The **worst-case runtime** of an algorithm is the runtime complexity for an input that results in the longest execution

Other runtime analyses include **best-case runtime** and **average-case runtime**.

Runtime analysis example:

Given an algorithm, Count the number of operations:

```
maxVal = numbers[0]
for (i = 0; i < N; ++i)
{
    if (numbers[i] > maxVal)
    {
        maxVal = numbers[i]
     }
}
```

Number of operations in worst case ?

Algorithm

An algorithm is a sequence of steps, including at least 1 terminating step, for solving a problem.

Recursive algorithm / function

- breaks the problem into smaller subproblems
- applies itself i.e. **calls itself** to solve the smaller subproblems

Base case A case where a recursive algorithm completes without applying itself to a smaller subproblem.

```
Factorial(N)
{
    if (N == 1)
        return 1
    else
        return N * Factorial(N - 1)
}
```

Data structure

A data structure is a way of organizing, storing, and performing operations on data.

Table 4.9.1: Basic data structures.

Data structure	Description
Record	A ***record*** is the data structure that stores subitems, with a name associated with each subitem.
Array	An ***array*** is a data structure that stores an ordered list of items, with each item is directly accessible by a positional index.
Linked list	A ***linked list*** is a data structure that stores an ordered list of items in nodes, where each node stores data and has a pointer to the next node.
Binary tree	A ***binary tree*** is a data structure in which each node stores data and has up to two children, known as a left child and a right child.
Hash table	A ***hash table*** is a data structure that stores unordered items by mapping (or hashing) each item to a location in an array.
Heap	A ***max-heap*** is a tree that maintains the simple property that a node's key is greater than or equal to the node's childrens' keys. A ***min-heap*** is a tree that maintains the simple property that a node's key is less than or equal to the node's childrens' keys.
Graph	A ***graph*** is a data structure for representing connections among items, and consists of vertices connected by edges. A ***vertex*** represents an item in a graph. An ***edge*** represents a connection between two vertices in a graph.

Abstract data type An abstract data type (ADT) is a data type described by predefined user operations, such as “remove data from front," without indicating how each operation is implemented.

4.10 Abstract data types

Table 4.10.1: Common ADTs.

Abstract data type	Description	Common underlying data structures
List	A ***list*** is an ADT for holding ordered data.	Array, linked list
Stack	A ***stack*** is an ADT in which items are only inserted on or removed from the top of a stack.	Linked list
Queue	A ***queue*** is an ADT in which items are inserted at the end of the queue and removed from the front of the queue.	Linked list
Deque	A ***deque*** (pronounced "deck" and short for double-ended queue) is an ADT in which items can be inserted and removed at both the front and back.	Linked list
Bag	A ***bag*** is an ADT for storing items in which the order does not matter and duplicate items are allowed.	Array, linked list
Set	A ***set*** is an ADT for a collection of distinct items.	Binary search tree, hash table
Priority queue	A ***priority queue*** is a queue where each item has a priority, and items with higher priority are closer to the front of the queue than items with lower priority.	Heap
Dictionary (Map)	A ***dictionary*** is an ADT that associates (or maps) keys with values.	Hash table, binary search tree

ADTs allow programmers to focus on choosing which ADTs best match a program's needs

Points to think about:

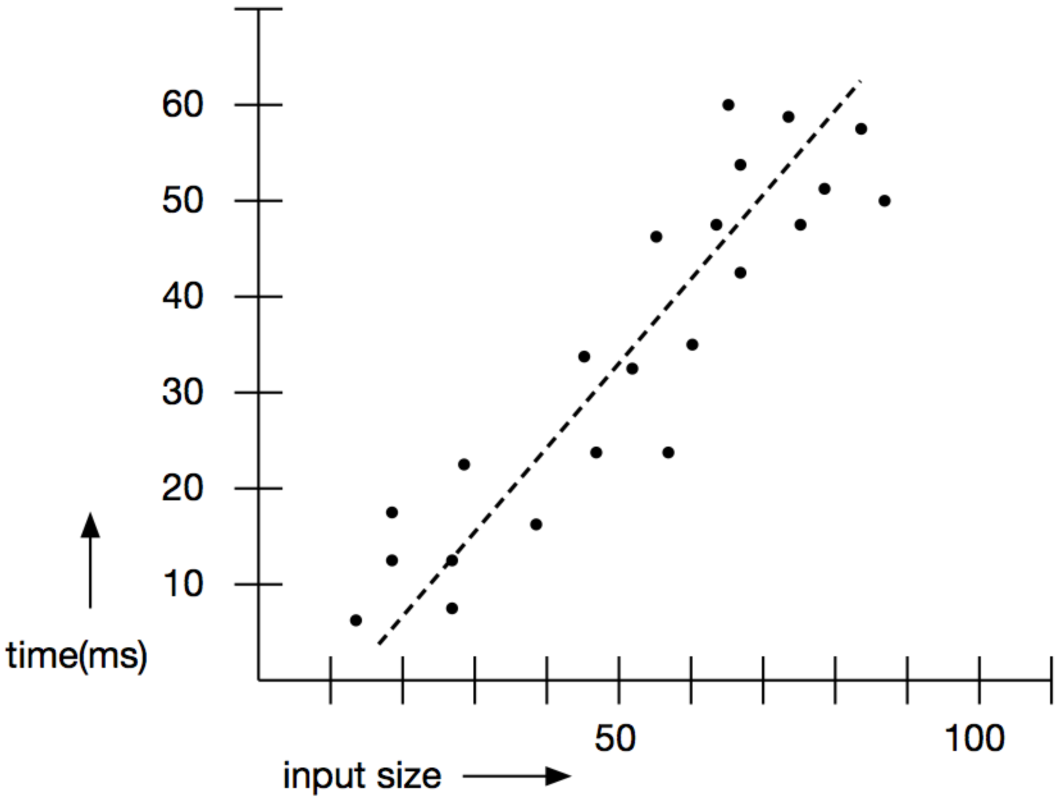
- What ADT to use to reversing a list of data elements or say, Display list of users in reverse chronological order

Analysis of Algorithms

- This course’s goal: the design of “good” data structures and algorithms
- Data structures: systematic way of organizing and accessing data
- Algorithms: Step-by-step procedure for performing a task in a finite time.
- What does “good” mean?
    - Running time—fast
    - Space usage—small
- Usually means a trade-off
    - Faster often requires more memory for extra pointers
    - Smaller often requires more complex algorithms
- Essential point: **Running time increases with input size**

Experimental Studies

- Run the algorithm with many different input sizes
- Measure the running time
- Plot the results

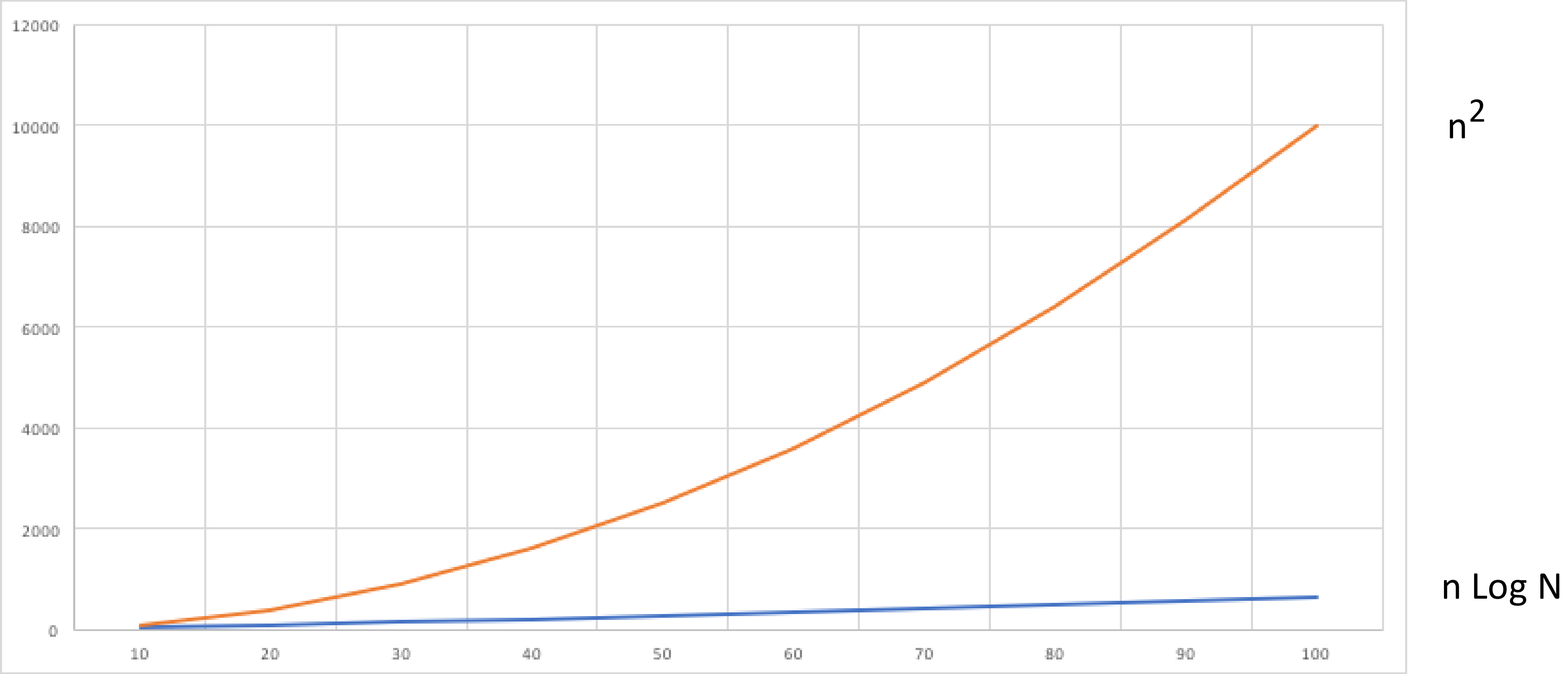


Experiment on running time of algorithm.

- Three major limitations:
  - Limited set of inputs; may omit important ones
  - Must always run in same hardware and software environment
  - Must actually implement the algorithm
- Need a general approach that:
  - Considers all possible inputs
  - Allows relative comparisons, independent of hardware and software.
  - Can be done through study, not implementation and lengthy experimentation.

Asymptotic Notation

- Define algorithm in pseudo-code—a series of primitive operations.
- Focus on growth rate as a function of input size, not actual run times.



- The big(O) notation:
    - “function f is the order of g(n)”
    - “function f is big-Oh of g(n)”
    - “function f is O(n)”
    - “function f is Order n”
- Allows us to characterize an algorithm’s run time in general terms as a function of the input size.
- Ignores constants and lower order terms
    - Constants aren’t affected by size
    - if there are n Log n and n^2 components, n^2 will dominate.
- Allows us to compare algorithms and choose the one with the slowest growth rate. “n Log n is better than n^2.”

Asymptotic Analysis

n	log n	n	n log n	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84E+19
128	7	128	896	16,384	2,097,152	3.40E+38
256	8	256	2,048	65,536	16,777,216	1.16E+77
512	9	512	4,608	262,144	134,217,728	1.34E+154

Growth rates of runtimes.

Running Time (us)	Maximum Problem Size (n)		
	1 second	1 minute	1 hour
400n	2,500	150,000	9,000,000
$2n^2$	707	5,477	42,426
2^n	19	25	31

Maximum problem sizes.

Running Time	New Maximum Problem Size
400n	256m
$2n^2$	16m
2^n	m + 8

New maximum problem sizes:
CPU 256 times faster

Big-O Examples

```
int LinearSearch(int numbers[], int numbersSize, int key)
{
    for (int i = 0; i < numbersSize; ++i)
    {
        if (numbers[i] == key)
        {
            return(i);
        }
    }
    return(-1); // not found
}
```

Binary Search

Search for 30

Round 1

0	1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80	90
lwr				mid				upr

Round 2

0	1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80	90
lwr	mid		upr					

Round 3

0	1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80	90
		lwr mid	upr					

Search for 75

Round 1

0	1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80	90
lwr				mid				upr

Round 2

0	1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80	90
					lwr	mid		upr

Round 3

0	1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80	90
							lwr mid	upr

Round 4

0	1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80	90
						upr	lwr	

range is empty

```
int BinarySearch(int numbers[], int numbersSize, int key)
{
    int    mid = 0;
    int    lwr = 0;
    int    upr = numbersSize - 1;
    while (upr >= lwr)
    {
        mid = (upr + lwr) / 2;
        if (numbers[mid] < key)
        {
            lwr = mid + 1;
        }
        else if (numbers[mid] > key)
        {
            upr = mid - 1;
        }
```

```
        else
        {
            return(mid);
    }   }
    return(-(mid + 1)); // mid indicates insertion point
}
```

Selection Sort

Round 1

20	10	40	50	30
i	s			

swap

10	20	40	50	30

Round 2

10	20	40	50	30
	i & s			

swap with self

10	20	40	50	30

Round 3

10	20	40	50	30
		i		s

swap

10	20	30	50	40

Round 4

10	20	30	50	40
			i	s

swap

10	20	30	40	50

```
void SelectionSort(int numbers[], int numbersSize)
{
    int         indexSmallest;
    int         temp;
    for (int i = 0; i < numbersSize; ++i)
    {
        indexSmallest = i;
        for (int j = i + 1; j < numbersSize; ++j)
        {
            if (numbers[j] < numbers[indexSmallest])
            {
                indexSmallest = j;
            }
        }
        temp = numbers[i];
        numbers[i] = numbers[indexSmallest];
        numbers[indexSmallest] = temp;
    }
}
```

Amortization

Financial: Amortization is paying off an amount owed over time by making planned, incremental payments of principal and interest.

Computer Science: To even out the costs of running an algorithm over many iterations, so that high-cost iterations are much less frequent than low-cost iterations, which lowers the average running time per iteration.

Example: Increase the Extension Size, Reduce the Number of Copy Operations

	Extend by 1			Double Each Extension		
Insert #	Size	Capacity	Copies	Size	Capacity	Copies
1	1	1	0	1	1	
2	2	2	1	2	2	1
3	3	3	2	3	4	2
4	4	4	3	4	4	
5	5	5	4	5	8	4
6	6	6	5	6	8	
7	7	7	6	7	8	
8	8	8	7	8	8	
9	9	9	8	9	16	8
10	10	10	9	10	16	
11	11	11	10	11	16	
12	12	12	11	12	16	
13	13	13	12	13	16	
14	14	14	13	14	16	
15	15	15	14	15	16	
16	16	16	15	16	16	
		Total Copies	**120**		Total Copies	**15**