Program Structure and Algorithms

Sid Nath

Lecture 1

Agenda

- Welcome!!!
- Introduction
- Administrative
- Lecture

About Me

- Adjunct Faculty at NEU!
- Ph.D. in CSE from UC San Diego
- 15+ years of industry experience
- Combinatorial optimization, algorithms and Machine Learning
  - https://www.linkedin.com/in/siddhartha-nath-6454b01/
  - Google Scholar: https://tinyurl.com/yc87zw7p

Key Announcements

- Textbook is “Intro to Algo” 4th edition
- All electronic submissions MUST be typed (LaTeX/Word)
  - Learn LaTeX using Overleaf
- Python IDE is your choice (recommend VSCode)

Course Overview

- Goal: Learn principles/techniques in algorithm design & analysis.
  - design: recursion, divide and conquer, greedy method, dynamic programming
  - data structures: arrays, linked lists, stacks, queues, ….
  - analysis: correctness, complexity analyses
- Foundations
  - Data structures
  - growth of functions
  - recurrences
- Greedy algorithms
- Dynamic programming
- Prerequisites: basic math, Python (preferred)

Evaluations

- Homeworks (4): 15%
- Quiz (~7): 20%
- Midterm exam: 20%
- Programming Assignments (2): 15%
- Final exam: 25%
- Attendance: 5% (Qwickly, absent after 20min)

Grading

- >94% A
- 89%-94% A-
- 83%-89% B+
- 76%-83% B
- 69%-76% B-
- <69% C

- Absolute vs. relative decided based on overall class performance
- Canvas shows incorrect % with extra credits! So, plz calculate yourself or ask me.

Expectations

- Focus on learning technique of algorithm design and analysis
  - Develop your own algorithms, show correctness and analyze efficiency
  - If you can grasp the material well, you do not need to worry about grades
- Foundation for pretty much any career path you take
  - Competitive advantage
  - High growth
- Questions via email or Camino Discussions

How to be Successful in INFO 6205

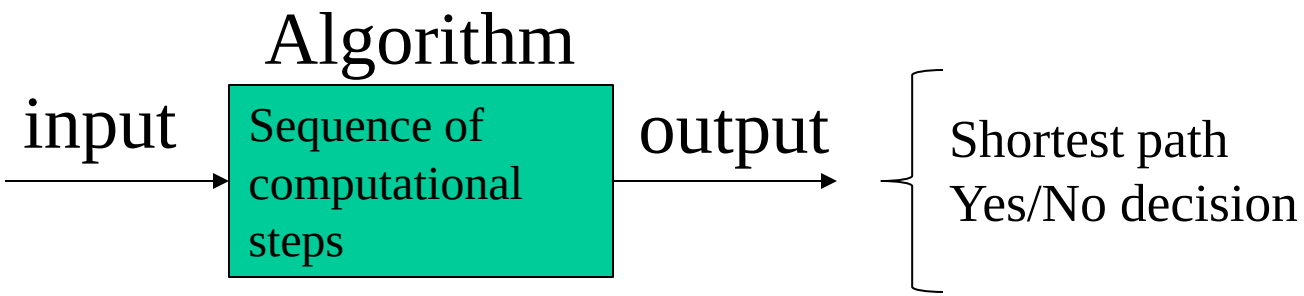
- Think and solve problems by yourself first before consulting colleagues, looking up online, ChatGPT, etc.
- Given a problem, develop an algorithm (!= code)
- Run your initial algorithm on the smallest possible (tiny) testcase (or, example) first
  - Think of corner cases, adversarial examples
  - If your algorithm works across both tiny and adversarial examples, proceed
  - Always articulate your algorithm in English and assess correctness

Two kinds of problems

- Non optimization problem vs Optimization problem with an objective
- Non optimization problem
    - Sorting
    - Matrix multiplication
    - Interested in designing a fastest algorithm to find a solution
- Optimization problem with an objective
    - UPS delivery: minimizing the delivery time
    - Build a house with fixed budget: maximizing area inside
    - Interested in
        - Finding a solution with the optimal objective value
        - Fastest algorithm
- Problem size: number of inputs

Role of Algorithm in Computing

- Definition: a finite set of precise instructions for performing a computation or for solving a problem.
- Application:
  - Optimal binary search tree



Complexity of an algorithm = f(input size)
e.g., Running time, space

Example

Describe an algorithm to find the MAX VALUE in the list of integers $a_1, a_2, \ldots, a_n$

Sample Solution:

- Set the temporary max equal to a_1
- Iterate i over 2:n, and compare each a_i with temp max; if larger replace temp max with a_i

```
Procedure max(a1, a2, a3, …, an: integers)
    max = a1
    for i=2 to n
        if max < ai then max = ai
    output max
```

- Number of steps: 1 + (n-1) + (n-1) + 1 = 2n

Applications

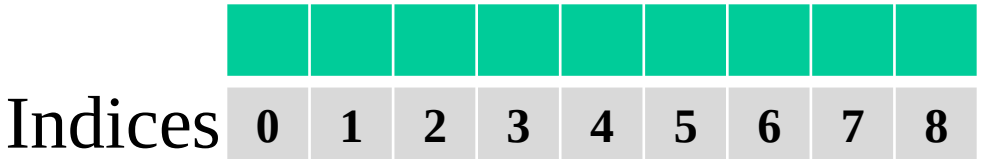
- Search engine
- Greedy – shortest path, assignment
- Dynamic programming – optimization of buffers on a net, similarity in DNA sequences in Human Genome Project
- Linear programming – optimization

Data Structures

- A fundamental concept to help design and code algorithms / solve problems efficiently
- Data structure types
  - Linear
    - Arrays, linked lists, stacks, queues, matrices
  - Non-linear
    - Trees, graphs, heaps, hash tables
  - Python-specific
    - Lists, tuples, dictionaries
- Operations: CRUD
  - Create, Read, Update, Delete

Arrays

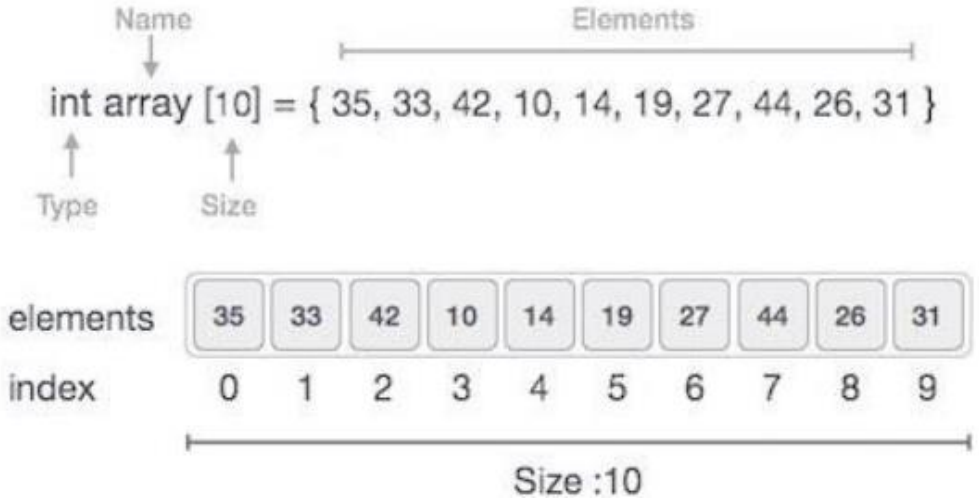
- Fundamental container/collection to hold a fix number of items
- Each item is often referred to as an “element”
- Each location of an element has a numerical index
  - We use the index to refer to / identify the element



- Index starts with 0 and ends with (length of array – 1)

Arrays

- Each item MUST be of the same type
  - When t is type, t[] is the type of an array with elements of type t; thus must be declared
  - E.g., t can int, char
  - int[] A = alloc_array(int, 10); // A is an array of integers with 10 elements



Array Use Cases

- Student grades, max or min temp per day in a month
- Graphics: store RGB values
- Keyboard buffer: keystrokes are temporarily stored in an array called the keyboard buffer
- Implement other data structures such as stacks, queues
- Complex math operations such as matrix operations
- Game board representation such as in chess, tic-tac-toe
- …..

Arrays

- A[i], A[i+1] addresses differ by the size of the type in programming language
- Access elements as A[i]; $0 \leq i < n$
- Basic operations
  - Iterate / traverse
    - for (int i = 0; i < 10; i++) { A[i] }
  - Insert at an index
  - Delete at an index
  - Search for an element
  - Update at an index
    - A[i] = e; // e is a literal / variable of type of A

Arrays

- Python allows negative index (A[-1] is the last element)

```
from array import *

arrayName = array(typecode, [Initializers])
```

Typecode	Value
b	Represents signed integer of size 1 byte
B	Represents unsigned integer of size 1 byte
c	Represents character of size 1 byte
i	Represents signed integer of size 2 bytes
I	Represents unsigned integer of size 2 bytes
f	Represents floating point of size 4 bytes
d	Represents floating point of size 8 bytes

Arrays

```
from array import *

array1 = array('i', [10,20,30,40,50])

for x in array1:
   print(x)
```

```
10
20
30
40
50
```

Create

```
print (array1[0])

print (array1[2])
```

```
10
30
```

Read

```
array1.insert(1,60)

for x in array1:
   print(x)
```

```
10
60
20
30
40
50
```

Insert

```
array1.remove(40)

for x in array1:
   print(x)
```

```
10
60
20
30
50
```

Delete

Arrays

```
array1 = array('i', [10,20,30,40,50])

print (array1.index(40))
```

```
3
```

Search

```
array1[2] = 80

for x in array1:
   print(x)
```

```
10
20
80
40
50
```

Update

Numpy Arrays

```
import numpy as np

arr1D = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr1D)

arr2D = np.array([[1, 2, 3], [4, 5, 6]])

print(arr2D)

print("Dimension of arr2D: {}".format(arr2D.ndim))

## Indexing
print('Last element from 2nd dim: ', arr2D[1, -1])

print(arr1D[4:])

print(arr1D[:4])

print(arr1D[-3:-1]) ##Slice from index 3 from the end to index 1 from
the end
```

Numpy Arrays

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

## Slice [start:end]; default start=0, default end= length of arr in
that dim
## Slice + step [start:end:step]; default step=1

print(arr[1:5:2]) ## Return every other element from index 1 to 5

print(arr[::2]) ## Return every other element of arr
```

Numpy Arrays Reshaping

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr2D = arr.reshape(4, 3) ## 1D to 2D

print(newarr2D)

newarr3D = arr.reshape(2, 3, 2)

print(newarr3D)
```

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

newarr = arr.reshape(-1)

print(newarr)
```

Lists (Python)

- Versatile container for comma-separated items
- Also, referred to as “dynamic arrays”
- Key advantage over arrays is that they don’t need to be declared
- Items can be of different types

```
list1 = ['physics', 'chemistry', 1997, 2000]
list2 = [1, 2, 3, 4, 5 ]
list3 = ["a", "b", "c", "d"]
```

List Operations

```
list1 = ['physics', 'chemistry', 1997, 2000]
list2 = [1, 2, 3, 4, 5, 6, 7 ]
print ("list1[0]: ", list1[0])
print ("list2[1:5]: ", list2[1:5])
```

list1[0]: physics
list2[1:5]: [2, 3, 4, 5]

Access

```
list[2] = 2001
print ("New value available at index 2 : ")
print (list[2])
```

Value available at index 2 :
2001

Update

```
del list1[2]
print ("After deleting value at index 2 : ")
print (list1)
```

After deleting value at index 2 :
[‘physics’, ‘chemistry’, 2000]

Delete

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

Array Advantages

- Constant time access
  - Direct indexing
- Efficient use of memory
  - Contiguous memory locations, cache locality
  - Predictable memory footprint due to size fixed at declaration
- Simplicity and ease of use
  - Simplest data structures, all programming languages support arrays
  - All elements are of the same type – uniformity
- Dimensional versatility
  - Extend to 2D, 3D, etc.
- Data integrity
  - Indices are fixed

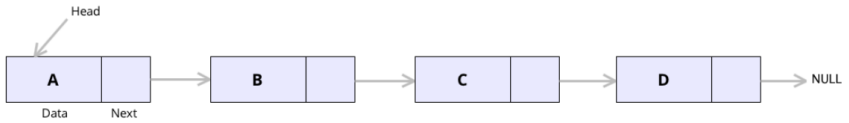
Array Disadvantages

- Fixed/static sizing
  - Memory limit is fixed at declaration – overallocation wastes memory
  - Resizing can be complex, deep copy of elements
- Insertion/deletion overhead
  - Shifting elements is a costly operation
- Sequential access
  - Finding an element without knowing its index
- Data homogeneity
  - All data types are the same can be limiting in many applications

LinkedLists

- Alternative to arrays, dynamic data structure
- Each item in a linked list contains a data element of some type and a pointer to the next item in the list

```
class Node:
        def __init__(self, data):
                self.data = data
                self.next = None
```



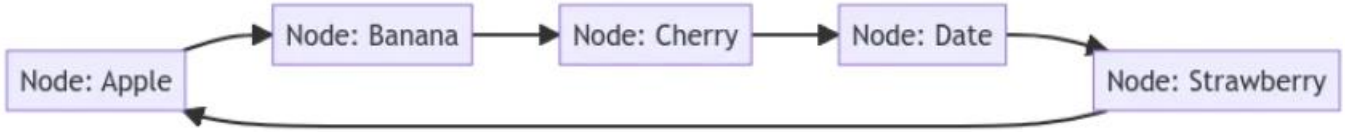
- Operations
  - Insert(index) O(1) for first/last, O(n) for arbitrary
  - Delete(index) O(1) for first/last, O(n) for arbitrary
  - Update(index) O(1) for first/last, O(n) for arbitrary
  - Size() O(1)

LinkedLists Types

- Singly linked lists – each node contains data and a link to the next node (unidirectional)

- Doubly linked lists – each node contains data, a link to the next node and a link to the previous node (bidirectional)

- Circular linked list – the last node in the list points back to the first node



LinkedList Use Cases

- Music playlist – users can add/remove songs
- Browser history (doubly linked list)
- Undo functionality in many software apps (IDE, Word, pptx, etc.)
- Memory management in the OS
- Task scheduling (e.g., round-robin schedulers use a circular linked list)
- Implement other complex data structures – stacks, queues, trees, …

LinkedList Advantages

- Dynamic sizing
- Efficient memory utilization
- Ease of insertion/deletion, no need to shift data even when a node is inserted in the middle of the linkedlist
- On-the-fly memory allocation of a new node

LinkedList Disadvantages

- Additional memory to store links to next/previous nodes
- Sequential traversal, complexity in implementations
- Can be inefficient w.r.t. caching
  - No locality guarantees as memory may not be contiguous
- Manual memory management when nodes are deleted
  - Free up allocated memory in many programming languages

Lecture 1 summary

- What is an algorithm?
- Applications of algorithms
- Data structure types
- Review of arrays and linkedlists using Python