Part One Foundations

Will now begin to focus on the design and implementation of algorithms. Here is a description of all the chapters that will be encountered.

- Chapter 1: Overview of algorithms and their place in the world
- Chapter 2: Introduction to Algorithms and Examples
- Chapter 3: Notation Definition
- Chapter 4: Divide and Conquer
- Chapter 5: Probabilistic Analysis

Chapter 1: Algorithms

`Algorithm`: set of instructions tailored to solve a problem

- receive **input** and give **output**
- example: take numbers in random order to sort them from least to greatest
- algorithm can only be correct if for each possible instance of a problem, algorithm outputs the correct solution to problem
- incorrect algorithms can be useful if their error rate is low enough

`Instance of Problem`: consists of input that satisfies constraints imposed in problem statement needed to compute solution to problem

Examples of Problems to be solved

- sorting human genomes
- traveling salesman problem
- cryptography and online secret keeping, secure transactions'
- resource allocation

`Data Structure`: method of storing data in a computer in a way the meets constraints

`Parallelism`: splitting work amongst more than one processor cores and threads to speed up computation times

Exercises: Algorithms

1. Example of application the requirements computing convex hull

- sorting the mail
- finding the best way to package an item for shipping

2. measures of efficiency

- amounts of memory required, space. Reliability and error rate. Simplicity and ability for humans to understand it

3. describe data structure

- dictionaries: it has insane look up times however if there are too many collisions then it could begin to slow down with the greater amount of times store inside

4. compare contrast the shortest path and traveling salesman

- they are similar since both are trying to find the best 'path'

5. what sort of problem has only one solution

- decrypting encrypted files
- Finding the shortest path across all points in graph

Algorithms as a Technology

Time and space are two things that are limited. Time that a computer takes to compute a thing and amount of memory and hard drive space in which to do it in

`Efficiency`: time and space required to compute solution compared against other algorithms

Algorithms and other Technologies

- total system performance can heavily depend on algorithms used in the running of the system
- knowledge and understanding of algorithms is the thing that separates software engineers from script kiddies

Exercises: Algorithms as a Technology

1. Example of application that requires algorithm as fundamental of function

- computer graphics and ability to display same image regardless of hardware, display, CPU architecture and graphics card. Layers upon layers of abstraction

2. at which point is insertion sort better than merge sort

- finds at which value n are both equations equal.
- 8n^2 = 64nlogn
- n = 8logn
- n / log(n) = 8

3. answer is simply 1

- same method as above

Problems

Comparison of Running Times

Time Complexity	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\log n$	0	1.77	3.55	4.94	6.43	7.51	9.51
$\sqrt{n}$	1	7.7	60	294	1637	5669	56693
n	1	60	3600	86400	2678400	32140800	3214080000
$n \log n$	1	60	3600	86400	2.68e7	3.21e7	3.21e9
n^2	0	107	12803	426515	1.72e7	2.41e8	3.1e10
n^3	1	3600	1.3e6	7.46e9	7.2e12	1e15	1e19
2^n	1	21600	4.7e10	6.4e14	1.9e19	3.3e22	3.3e28
$n!$	2	??	?? ??	??	??	??	??

Chapter 2: Getting Started

This chapter is about the framework that will be pursued further down the road in the pursuit of getting the student familiar with inner workings of the way the book is structured.

`Pseudo Code`: method of describing a set of instructions or algorithms that is implementation agnostic. Can be interpreted and implemented in a variety of different manners. Algorithm described in the clearest manner

Insertion Sort

- Solution to sorting Problem is insertion sort in which given a set of numbers we try to order them from smallest to greatest
- To prove that algorithm is correct, we must show that

`Loop invariant`: section of the array that are already sorted.

- `Initialization`: certain properties of array are true before algorithm starts
- `Maintenace`: if certain things are true before iteration of loop, will remain true for next iteration
- `Termination`: certain properties arise from evaluation of loop

Things that are proven with insertion algorithm

- At start sub array A[0..j] where j = 1, is an array with a single element, as such it is trivially sorted
- at each iteration, the right most element being sorted is swapped with the previous element, if it does not reach the correct position it is swap again
- when j reaches the value of length of A, the algorithm stops and by the array should be sorted.

Rust Implementation

```rust
pub fn insertion_sort<T: PartialOrd + Clone>(arr: &mut [T]){
    let len = arr.len();
    for  j in 2..len{
        let key = arr[j].clone();
        let mut i = j - 1;
        while i > 0 && arr[i] > key{
            arr.swap(i+2, i);
            i = i - 1;
        }
        arr[i+1] = key;
    }
}
```

Exercises

```
def  linear_search(seq, key):
    for  i in range(0, len(seq)):
        if seq[i] == key:
            return i
    return None
```

2 Define N as natural number Set N as number if of bits defining INT data type Define set A as to be size N+1, contains N sized integers Define set B as to be size N+1, contains N sized integers Define set C as to be size N+1, contains N sized integers

for index i in range 0 to N: value of C[i] = A[i] + B[i]

```
let N: usize;
//N = 10; //can be be any number
// N defines the size of N
let a: Vec<intN> = Vec::with_capacity(n+1);
let b: Vec<intN> = Vec::with_capacity(n+1);
let mut c: Vec<intN> = Vec::with_capacity(n+1);
for i in 0..n{
    c[i] = a[i] + b[i];
}
```

Analyzing algorithms

- To analyze an algorithm means to be able to predict the runtime, memory and communication bandwidth costs of running said algorithm in accordance with the cost associated with data of size N
- `Random-Access Machine`: implementation of technology in which instructions can only run sequentially.
- Certain instructions or procedures can be completed in a single instruction however most require many others.
  - For example sorting can only be done in multiple instructions, not just one
  - add, subtract, multiply, divide, remainder, floor, ceiling are examples of constant time instructions
  - shifting bits to left is equivalent to multiplying by 2^k where k is number of position of bits being shifted, it is most complicated than this however this simplification is what would be worked with
- RAM model analysis is considered useful in predicting actual performance

Analysis of Insertion Sort

- Time it takes for insertion sort to finish depends on the size of array, and the already sorted level of array.

- Running time of algorithm is the sum of running times of all steps, procedures and operations with relation to size n.
-

`Input Size`: depends on problem being solved. Can mean size of set/array, the actual number being used in computation. Can mean total number of bits. Number of nodes and vertices in graph.

`Running Time`: number of primitive steps/operations being executed. Certain operations take constant time, meaning they perform the same last c number of time regardless of N size of input

Example With Insertion Sort

- the Best case: the array is already sorted, and the algorithm simply verifies this fact. Linear time
- the best case: array is sorted in incorrect order. Every single item has to be checked with every other item: Quadratic time complexity

Worst-case and Average Case analysis

- Book focuses only on finding the worst possible case.
  - Gives upper bound on running time of algorithm, guarantees that algorithm will not run any longer
  - in real life, worst case performance occurs when a search is performed and data is not found
  - average case is often as bad as the worst case
- `Averange Case`: when we consider all possible inputs to be equally likely

Order of Growth

`Order of Growth`: Rate of Growth, leading term of polynomial thing. The Highest order term. Nothing else matters. We can have to estimate the true time complexity then we have everything else. If worst case of algorithm A is better than the worst case for B, then it is considered that A is better than B

Exercises

1 n^3

2 Problem is sorting an array of numbers from smallest to largest.

Pseudo Code

- Define Procedure FindSmallestIndexInArray
  - input Array A, sequence of arbitrary numbers
  - let I be natural number := 0
  - for each index and value in Array:
    - if value is less than A[I]

            · I := index
        – return I
- Define Procedure: RecursiveSelectionSort
    – input: array A
    – if A has one element
        * return A
    – let I := FindSmallestIndexInArray(A)
    – swap values for A[I], A[0]
    – let A_1 be array of one element A[0]
    – let A_2 be array excluding first element of A, A[1:]
    – let B := RecursiveSelectionSort(A_2)
    – return union of A_1 and B

Procedure FindSmallestIndexInArray has time complexity of O(N)

Procedure RecursiveSelectionSort worst time complexity as O(N^2). However best case would be O(N), because it will keep on calling itself recursively until it reaches a array size of 1

- FindSmallestIndexInArray is called N times for array size N

```
def find_smallest(A:list):
    smallest_idx = 0
    for (i,x) in enumerate(A):
      if x < A[smallest_idx]:
        smallest_idx = i
    return smallest_idx

def selection_sort(A):
    if len(A) == 1:
        return A
    small_idx = find_smallest(A)
    A[0], A[small_idx] = A[small_idx], A[0]
    result = [A[0]]
    result.extend(selection_sort(A[1:]))
    return result

import random
l = [random.randint(0,100) for i in range(100)]
print(l)
print(selection_sort(l))
```

3 O(0.5N)

4 By analysing the runtime of the algorithm with a variaty of different test datas and benchmark and try to give a better case than worst case.

Designing Algorithms

- there are multiple school of tought when it comes to designning and implementing algorithms. One is `incremental` in which a problem is solved a little bit at a time
  - in insertion sort array A[1..j-1] is sorted, we then take element A[j] into the proper location. After which array A[1..j] becomes sorted
- another method is `divide and conquer` in which a bigger problem in split into smaller problems to be solved individually

Divide and Conquer

`Recursion`: procedure that calls itself.some algorithms are recursive in nature

- **Divide** problem into smaller sub problems
- **Conquer** either solve a sufficiently small problem or further split into smaller problems
- **Combine** solutions of smaller problems into large problem
- merge sort is an example of this approach

```
Define Procedure MergeSort
Input Array A
if A has one element, return A. it is already sorted
split A into A_1 and A_2
B_1 = MergeSort(A_1)
B_2 = MergeSort(A_2)
Merge B_1 and B_2
```

To merge two sorted arrays

- Start with two stacks that are in sorted order
- peek both stacks and compare
- push into output array
- repeat until both stack are empty

```
def merge_sort(A):
    if len(A) > 1:

        mid = len(A)//2
        L = A[:mid]
        R = A[mid:]
        merge_sort(L)
        merge_sort(R)
        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                A[k] = L[i]
                i += 1
            else:
```

```
            A[k] = R[j]
            j += 1
        k += 1
    while i < len(L):
        A[k] = L[i]
        i += 1
        k += 1

    while j < len(R):
        A[k] = R[j]
        j += 1
        k += 1
```

- Initialization
  - take in array, and compute the appropiate indexes of the things that are to be worked with.
- Maintenace
  - as each sub array is merged, one array has number of elements lesst than or equal to the other sub-array
  - the difference between the number of elements in each array never exceeds one
- Termination
  - runs until it reaches the end of the array and or if it reaches the sentinal value

Analyzing divide-and-conquer Algorithms

- when algorthms calls themselves they are described as `recurrance equation`.
- if a problem is small enough, generally this allows it to be solved in virtually contant time. However with the size of the problem the time complexity grows a lot faster.

Merge Sort Analysis

- each time the algorithm calls itself it divides the time complexity by a half. However this is done twice each time. So the time complexity if merge sort is O(n), linear complexity
- at a certain point the time complexity of merge sort becomes bettwer than insertion sort.
- time complexity is nlogn

Exercises illustration of merge sort

rewrite to not use sentinal values

use induction in order to time complexity if nlogn

```
def binary_search
```

- Chapter 3 # Chapter 3:

Chapter 4:

Chapter 5: