Recitation 1

Online: Xinyi Zhao xz2833@nyu.edu

GCASL 475: Yifan Jin yj2063@nyu.edu

New York University

Basic Algorithms (CSCI-UA.0310-005)

Introduction

- Review last week’s assignment
  - Mainly focus on theoretical solutions and proof
  - Provide pseudocode instead of real codes
  - Encourage sharing opinions
- Bonus problems: 2 interview type problems (Leetcode)
- Q & A

This week: Go over the required background needed for the course

- Induction
- Recursions
- Stability
- Proof of correctness of algorithms using loop invariant

Induction

Mathematical induction is a mathematical proof technique.

1. (Normal) Induction
2. Strong Induction

(Normal) Induction

1. Base case(s): check that the statement holds for the base case(s) and how to identify them

2. Induction step:

   2.1: induction assumption: assume that the statement holds for k

   2.2: induction conclusion: show that the statement holds for k+1

(Normal) Induction

Exercise 1: Prove $1+2+\ldots+n = n(n+1)/2$

(Normal) Induction

Exercise 1:   Prove $1+2+\ldots+n = n(n+1)/2$

Base Case: When $n = 1$, $1 = 1*2/2 = 1$, proved

(Normal) Induction

Exercise 1: Prove 1+2+...+n = n(n+1)/2

Base Case: When n = 1, 1 = 1*2/2 = 1, proved

Induction Step:

Assume it holds for k, 1+2+..+k = k(k+1)/2.

Then for k+1, 1+2+..+k+(k+1) = k(k+1)/2 + (k+1) = (k+1)(k+2)/2

hence it holds for k+1

Hence proved

(Normal) Induction

Exercise 2: Prove 1 + 2 + 2^2 + … + 2^n = 2^(n+1) - 1

Base Case: ?

Induction Step: ?

Strong Induction

1. Base case(s): check that the statement holds for the base case(s) and how to identify them

2. Induction step:

   2.1: induction assumption: assume that the statement holds for 1,2,...,k

   2.2: induction conclusion: show that the statement holds for k+1

Strong Induction

Exercise 1: Prove every positive integer bigger than one (n > 1) can be factored into some prime factor(s)

Base Case:

n = 2, 2 is a prime, then it got a prime factor (itself), proved

Induction Step:

induction assumption: assume for all integers m, with 1<m<=k, m has at least one prime factor

induction conclusion: for k+1, if k+1 is a prime, there is nothing to prove;

if k+1 is not a prime, by definition, there exist integers a and b, such that ab = k+1, with 1<a<k+1, 1<b<k+1. By the induction assumption, a has a prime factor p, then p is a prime factor of k+1.

Hence proved.

Strong Induction

Exercise 2: Prove 12|(n^4-n^2)

Strong Induction

Exercise 2: Prove $12|(n^4-n^2)$

Suppose we still use the normal induction.

Base Case: $12|(1^4 - 1^2) = 12|(1- 1) = 0$, proved

Induction Step:

Assume $12|(k^4 - k^2)$,

Then $(k+1)^4 - (k+1)^2 = (k^4 - k^2) + \underline{4k^3 + 6k^2 + 2k}$

Strong Induction

Exercise 2: Prove 12|(n^4-n^2)

Base Case:

Prove it holds for k=1..6 (Skip the computation)

Strong Induction

Exercise 2: Prove $12|(n^4-n^2)$

Base Case:

Prove it holds for k=1..6 (Skip the computation)

Induction Step:

Assume it holds for 1..k. And we define M = k - 5, then k+1 = (M+6)

Obviously it also holds for M

Then $(M+6)^4 - (M+6)^2 = (M^4 - M^2) + 12(2M^3 + 15M^2 + 71M + 105)$

Hence proved.

Comparison

Normal Induction:

In order to prove it holds for k+1, you just need to guarantee that it holds k.

Strong Induction:

In order to prove it holds for k+1, you need to guarantee that it holds for all integers less than k+1, since you will utilize the proof for some integers according to the problem requirement.

Recursions

Defined by two properties:

- A simple *base case* (or cases) — a terminating condition that does not use recursion to produce an answer (usually quite straightforward)
- A *recursive step* — a set of rules that reduces all successive cases toward the base case.

Recursions

The Fibonacci sequence is a classic example of recursion:

- Base cases:
  - $\mathrm{Fib}(0) = 0$
  - $\mathrm{Fib}(1) = 1$
- Recursion formula:
  - For all integers $n > 1$, $\mathrm{Fib}(n) = \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2)$.

Recursions

Now assume we have such recursions:

T(1)=1

T(n)=2*T(n/2)+n

For simplicity, let’s assume n = 2^k where k is an non-negative integer.

How to solve this recursion?

Recursions

Guess and prove

Guess: T(n) = 2*T(n/2)+n

= 2*(2*T(n/4) + n/2) + n = 4*T(n/4) +2n

= 4*(2*T(n/8) + n/4) +2n = 8*T(n/8) + 3n

=...= n*T(n/n) + kn (where n = 2^k)

Since T(1) = 1 and k = log_2(n),

= n*log_2(n) + n

Recursions

Guess and prove

Prove: $T(n) = n*\log_2(n) + n$

Which induction should we choose?

Recursions

Guess and prove

Prove: T(n) = n*log_2(n) + n

Which induction should we choose?

- Strong induction

Recursions

Guess and prove

Prove: T(n) = n*log_2(n) + n

Base Case: When n=1, T(1) = 1*log_2(1) + 1 = 1, proved.

Recursions

Guess and prove

Prove: T(n) = n*log_2(n) + n

Base Case: When n=1, T(1) = 1*log_2(1) + 1 = 1, proved.

Induction Step:

induction assumption: assume that the statement holds for n=1,2,...,k-1

induction conclusion: show that the statement holds for n=k (you only need to use the assumption for n=k/2), T(k) = 2*T(k/2)+k = 2*(k/2 * log_2(k/2) + k/2) +k = k*log_2(k) + k

Hence proved.

Stability

Definition of stability of a sorting algorithm:

A sorting algorithm is stable if objects with equal keys appear in the same order in the sorted output as in the unsorted input.

For example consider a list of Student Objects, with two fields, Student.name and Student.age. Now given a list of Student objects, a stable sort with respect to age would order the list in the increasing order (say) of the student ages but if two students have the same age, they will appear in the same order as in the unsorted list.

Stability

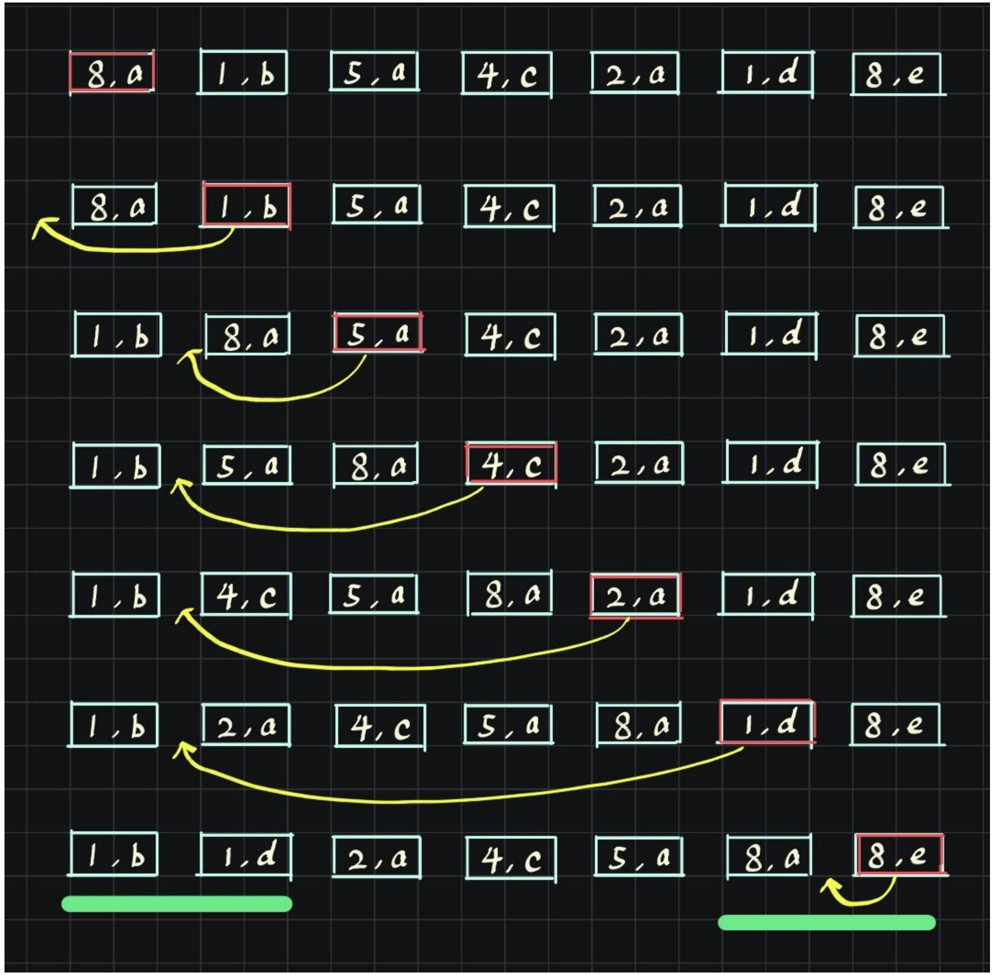
Exercise: why insertion sort is stable?

Input: a list of [age, name] - [8,a] [1,b] [5,a] [4,c] [2,a] [1,d] [8,e]

TODO: Use insertion sort to sort them with respect to age in the increasing order

Stability

Exercise: why insertion sort is stable?



Proof of correctness of algorithms using Loop Invariants

We use loop invariants to help us understand why an algorithm is correct.

Three steps of the proof:

- Initialization: It is true prior to the first iteration of the loop.
- Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.
- Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Proof of correctness of algorithms using Loop Invariants

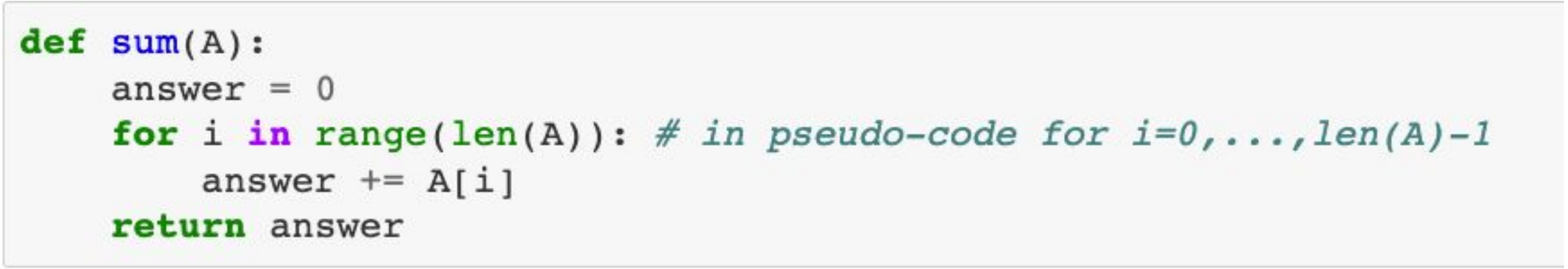
Similarity to induction: In induction, to prove that a property holds, you prove a base case and an inductive step.

Here, Initialization is like the base case, and Maintenance is like the inductive step.

Difference from the usual use of induction: In induction the inductive step is used infinitely; here, we stop the “induction” when the loop terminates.

Proof of correctness of algorithms using Loop Invariants

Example: Sum of a list of numbers - A[1], A[2], …, A[n]



```
def sum(A):
    answer = 0
    for i in range(len(A)): # in pseudo-code for i=0,...,len(A)-1
        answer += A[i]
    return answer
```

Proof of correctness of algorithms using Loop Invariants

Loop Invariant: At the start of iteration i of the loop, the variable *answer* should contain the sum of the numbers from the subarray A[1:i-1].

Proof of correctness of algorithms using Loop Invariants

Loop Invariant: At the start of iteration i of the loop, the variable *answer* should contain the sum of the numbers from the subarray A[1:i-1].

Initialization: At the start of the first loop the loop invariant states: 'At the start of the first iteration of the loop, the variable *answer* should contain the sum of the numbers from an empty array. The sum of the numbers in an empty array is 0, and this is what *answer* has been set to.

Proof of correctness of algorithms using Loop Invariants

Loop Invariant: At the start of iteration i of the loop, the variable *answer* should contain the sum of the numbers from the subarray A[1:i-1].

Initialization: At the start of the first loop the loop invariant states: 'At the start of the first iteration of the loop, the variable *answer* should contain the sum of the numbers from an empty array. The sum of the numbers in an empty array is 0, and this is what *answer* has been set to.

Maintenance: Assume that the loop invariant holds at the start of iteration i. Then it must be that *answer* contains the sum of numbers in subarray A[1:i-1]. In the body of the loop we add A[i] to *answer*. Thus at the start of iteration i+1, *answer* will contain the sum of numbers in A[1:i], which is what we needed to prove.

Proof of correctness of algorithms using Loop Invariants

Loop Invariant: At the start of iteration i of the loop, the variable *answer* should contain the sum of the numbers from the subarray A[1:i-1].

Initialization: At the start of the first loop the loop invariant states: 'At the start of the first iteration of the loop, the variable *answer* should contain the sum of the numbers from an empty array. The sum of the numbers in an empty array is 0, and this is what *answer* has been set to.

Maintenance: Assume that the loop invariant holds at the start of iteration i. Then it must be that *answer* contains the sum of numbers in subarray A[1:i-1]. In the body of the loop we add A[i] to *answer*. Thus at the start of iteration i+1, *answer* will contain the sum of numbers in A[1:i], which is what we needed to prove.

Termination: When the for-loop terminates i=n. Now the loop invariant gives: The variable *answer* contains the sum of all numbers in subarray A[1:n]. This is exactly the value that the algorithm should output, and which it then outputs. Therefore the algorithm is correct.

Q & A

Thank you