Algorithms and Problem Solving

LECTURE 1

Outline

- Algorithm: The Role of Algorithms in Computing - What are algorithms
- Algorithms as technology,
- Evolution of Algorithms,
- Design of Algorithm,
- Need of Correctness of Algorithm,
- Confirming correctness of Algorithm – sample examples,
- Iterative algorithm design issues.
- Problem solving Principles: Classification of problem, problem solving strategies,
- classification of time complexities (linear, logarithmic etc.)

Algorithm: The Role of Algorithms in Computing - What are algorithms

- The term Algorithm was coined by the Persian mathematician al-Khowarizmi in the ninth century.
- The algorithm is a set of rules which are used to solve real life problems.
- The algorithm provides a loose form of a solution in a pseudo programming language.
- Given the algorithm, it is easy to program the solution.

What are algorithms

We can treat an algorithm as a set of finite instructions which solves a particular problem when applied it is applied to that problem with legal inputs.

Algorithm:-

- The Algorithm is set of rules defined in specific order to do certain computation and carry out some predefined task. It is a step procedure to solve the problem.
- f the algorithm is correct, then the program should produce correct output on valid input, otherwise, it should generate an appropriate error message.
- For example, to find the division A/B, correctly written program would return value of A/B for B >0, and it would show the error message like "Invalid divisor" for B = 0.

Algorithms

Properties/Characteristics of algorithms:

- **Input** from a specified set,
- **Output** from a specified set (solution),
- **Definiteness** of every step in the computation,
- **Correctness** of output for every possible input,
- **Finiteness** of the number of calculation steps,
- **Effectiveness** of each calculation step

Algorithms

- A tool for solving a well-specified computational problem

- Algorithms must be:
  - Correct: For each input produce an appropriate output
  - Efficient: run as quickly as possible, and use as little memory as possible – more about this later

Algorithms Cont.

- A well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

- Written in a pseudo code which can be implemented in the language of programmer's choice.

Correct and incorrect algorithms

- Algorithm is correct if, for every input instance, it ends with the correct output. We say that a correct algorithm solves the given computational problem.

- An incorrect algorithm might not end at all on some input instances, or it might end with an answer other than the desired one.

- We shall be concerned only with correct algorithms.

Evaluation of Algorithm

Evaluation of Algorithm

- Though some people credit Babylonians with the development of the first algorithms, it was the unknown Indian mathematicians who developed and used the concept of zero and decimal positional number system. This allowed the development of basic algorithms for arithmetic operations, square roots, cube roots, and the like.
- The famous sanskrit grammarian Panini gave data structures like Maheshwar Sutra, which greatly facilitated compact rules and algorithms for phonetics, phonology, morphology, and syntax of the sanskrit language. He gave a formal language theory, almost parallel to the modern theory and gave concepts which are parallel to modern mathematical functions.

Evaluation of Algorithm

- during 1940s and 50s, the emphasis was on building the hardware, developing programming systems so that the computers can be used in commercial, scientific, and engineering problems.
- Though Alan Turing had given the idea of effective procedure in 1936,
- Soon it was realized that systematic methods of coding are required. This led to concepts like structured programming.
- The next logical step was seeking the proof of correctness of an algorithm, as many applications were identified where proven correctness was essential.
- the efficiency of an algorithm became a major issue. This led to a search for more efficient algorithms and an in-depth study of hard algorithms.
- Prof. Donald Knuth coined the term algorithm analysis in his monumental series of books,

Evaluation of Algorithm

- Complexity theory classes was developed based on Tractable and Intractable problems
- Another fact which emerged was that randomness can be an aid in solving many
- difficult problems.
- A variety of algorithms came up which used randomness as a basic component, for example, genetic algorithms, simulated annealing, and statistically defined algorithms. This area is also of current interest.

Algorithm as a technology

Algorithm as a Technology

Parameter	Faster Computer A	Slower Computer B
Speed	Executes 10^10 Instructions per second	Executes 10^7 instructions per second
Faster/Slower	A is 1000 time faster than B	B is 1000 time slower than A
Sorting algorithms run	Insertion sort	Merge Sort
Time complexity of Algorithm	c1 * n^2	c2 * n * logn
Value of Constant in time complexity	c1 is 2	c2 is 50
To Sort 10 Million Number (10^7)	A takes 2. (10^7) ^ *2 instruction/ 10^10 instructions/second	B takes 50* (10^7) log 10^7 instruction / 10^7 instructions/second
Time	~ 20,000 second (More than 5.5 Hours	~ 1163 Seconds (Less than 20 Minutes)

Iterative Algorithm Design Issues

Iterative Algorithm Design Issues

- Characteristics of most of the common and real-life algorithms, that is, they have at least one iterative component or a loop.
- the idea is that a part of the algorithm statements will be executed repeatedly thousands or even millions of times.
- Even a small amount of excess time spent within the loop can lead to a major loss in the efficiency of the algorithm.
- The situation can even be worse if there are loops within loops or nested iterations.
- There are algorithms in which the depth of nesting itself is controlled by an outer loop!
- The iterative components or loops are thus a major contributor to (in)efficiencies of algorithms.

Problem Solving

What is Problem Solving?

- **Problem solving** is the application of ideas, skills, or information to achieve the solution to a problem or to reach a desired outcome. Let's talk about different types of problems and different types of solutions.

Types of Problems

- A **well-defined problem** is one that has a clear goal or solution, and problem solving strategies are easily developed. In contrast,
- **poorly-defined problem** is the opposite. It's one that is unclear, abstract, or confusing, and that does not have a clear problem solving strategy.
- **routine problem** is one that is typical and has a simple solution. In contrast,
- **non-routine problem** is more abstract or subjective and requires a strategy to solve.

Problem solving Strategies

- The first strategy you might try when solving a routine problem is called an **algorithm**. Algorithms are step-by-step strategies or processes for how to solve a problem or achieve a goal.
- Another solution that many people use to solve problems is called **heuristics**. Heuristics are general strategies used to make quick, short-cut solutions to problems that sometimes lead to solutions but sometimes lead to errors. Heuristics are based on past experiences.

Problem solving Strategies

1. **Brute force** is a straightforward approach to solve a problem based on the problem's statement and definitions of the concepts involved.
2. **Greedy Algorithms** The solution is constructed through a sequence of steps, each expanding a partially constructed solution obtained so far. At each step the choice must be locally optimal – this is the central point of this technique.
3. **Divide-and-Conquer** Given an instance of the problem to be solved, split this into several smaller sub-instances (of the same problem), independently solve each of the sub-instances and then combine the sub-instance solutions so as to yield a solution for the original instance.
4. **Dynamic Programming** is a Bottom-Up Technique in which the smallest sub-instances are *explicitly* solved first and the results of these used to construct solutions to progressively larger sub-instances.
5. **Backtracking and branch-and-bound: generate and test methods** The method is used for state-space search problems. The solving process solution is based on the construction of a **state-space tree**, whose nodes represent states, the root represents the initial state, and one or more leaves are goal states.

Problem Solving Principal
Steps to solve Problem

- Identify a problem
- Understand the problem
- Identify alternative ways to solve a problem
- Select beat way to solve a problem from the list of alternative solutions
- Evaluate the solution

Time Complexity of Algorithms

Algorithm Analysis

- Why should we analyze algorithms?
  - Predict the resources that the algorithm requires
    - Computational time (CPU consumption)
    - Memory space (RAM consumption)
    - Communication bandwidth consumption
  - The **running time** of an algorithm is:
    - The total number of operations executed
    - Also known as **algorithm complexity**

Time Complexity

- Time complexity of an algorithm signifies the total time required by the program to run to completion.
- Time complexity of an algorithm is measured by its rate of growth relative to the standard function.
- **Cases of time complexity are:**
- **Worst-case**
  - An upper bound on the running time for any input of given size
- **Average-case**
  - Assume all inputs of a given size are equally likely
- **Best-case**
  - The lower bound on the running time

Time Complexity – Example

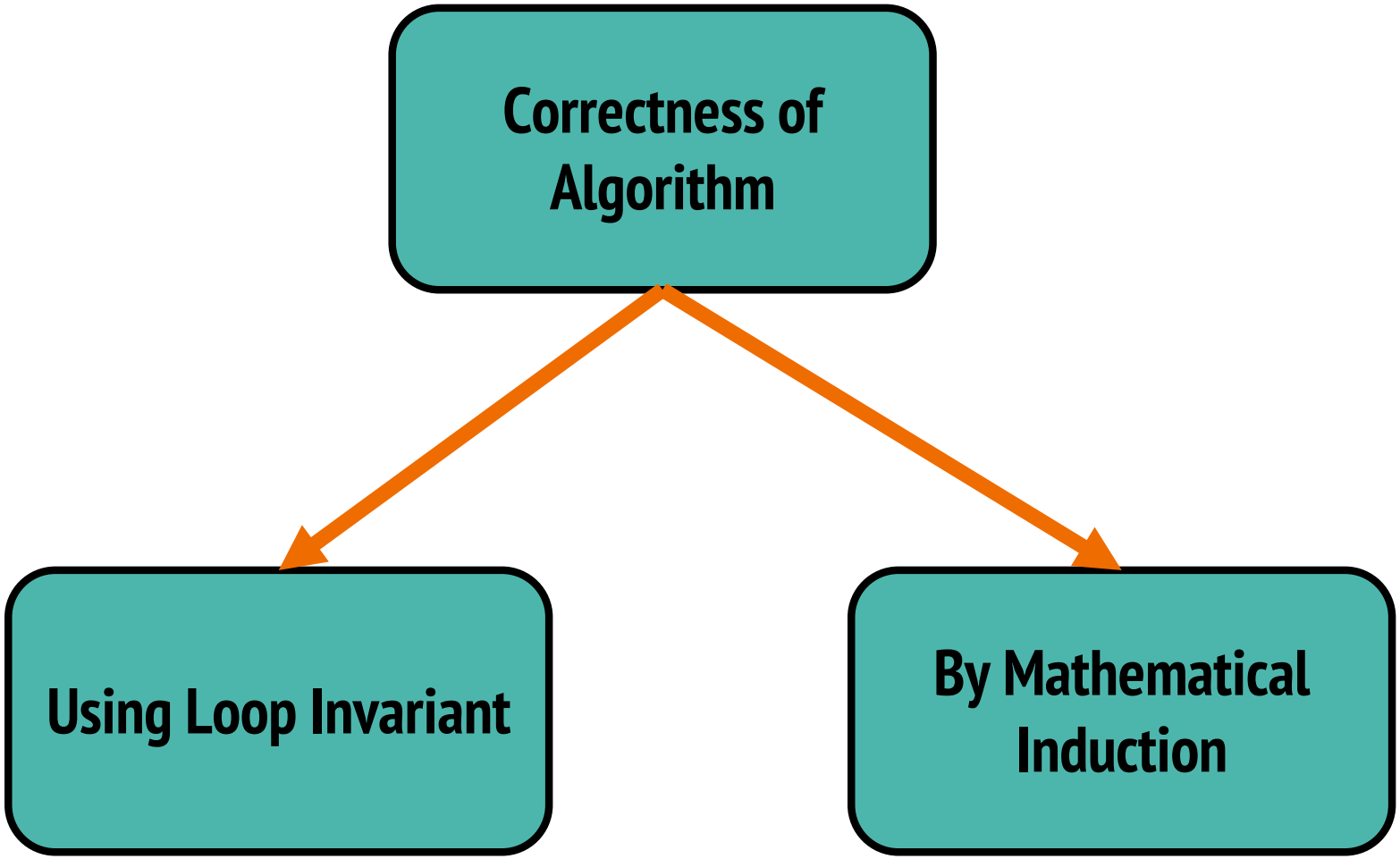
- Sequential search in a list of size n
  - Worst-case:
    - **n** comparisons
  - Best-case:
    - **1** comparison
  - Average-case:
    - **n/2** comparisons
- The algorithm runs in **linear time**
  - Linear number of operations

Classification of Time Complexity

Notation	Complexity	Description	Example
O(1)	Constant	Simple statement	**Addition**
O(log(n))	Logarithmic	Divide in half	**Binary search**
O(n)	Linear	loop	**Linear search**
O(n*log(n))	Linearithmic	Divide & Conquer	**Merge sort**
$O(n^2)$	Quadratic	Double loop	**Check all pairs**
$O(n^3)$	Cubic	Triple loop	**Check all triples**
$O(2^n)$	Exponential	Exhaustive search	**Check all subsets**
O(n!)	Factorial	Recursive function	**Factorial**

Correctness of Algorithm

Correctness of Algorithm



What does an algorithm ?

- An algorithm is described by:
  - Input data
  - Output data
  - ***Preconditions***: specifies restrictions on input data
  - ***Postconditions***: specifies what is the result
- Example: Binary Search
  - Input data: a:array of integer; x:integer;
  - Output data: found:boolean;
  - Precondition: a is sorted in ascending order
  - Postcondition: found is true if x is in a, and found is false otherwise

Correct algorithms

- **An algorithm is correct if:**
  - for **any correct input data**:
    - it **stops** and
    - it produces **correct output.**

  - Correct input data: satisfies precondition
  - Correct output data: satisfies postcondition

Proving correctness

- An algorithm = a list of actions
- **Proving that an algorithm is totally correct:**
  1. ***Proving that it will terminate***
  2. ***Proving that the list of actions applied to the precondition imply the postcondition***
  - This is easy to prove for simple sequential algorithms
  - This can be complicated to prove for repetitive algorithms (containing loops or recursively)
    - use techniques based on ***loop invariants*** and ***induction***

Example – a sequential algorithm

Swap1(x,y):

```
aux := x
x   :=  y
y   :=  aux
```

Precondition:

x = a and y = b

Postcondition:

x = b and y = a

Example – a repetitive algorithm

Algorithm Sum_of_N_numbers

Input: a, an array of N numbers Output: s, the sum of the N numbers in a

```
s:=0;
k:=0;
While (k<N) do
  k:=k+1;
  s:=s+a[k];
end
```

We use techniques based on loop invariants and induction

Loop invariants

- A loop invariant is a logical predicate such that: if it is satisfied before entering any single iteration of the loop then it is also satisfied after the iteration

Example: Loop invariant for Sum of n numbers

Algorithm Sum_of_N_numbers

Input: a, an array of N numbers

Output: s, the sum of the N numbers in a

```
s:=0;
k:=0;
While (k<N) do
    k:=k+1;
    s:=s+a[k];
    end
```

Loop invariant = induction hypothesis:
At step k, S holds the sum of the first k numbers

Using loop invariants in proofs

We must show the following 3 things about a loop invariant:

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

Example: Proving the correctness of theSum algorithm (1)

Induction hypothesis: S= sum of the first k numbers

1. Initialization: The hypothesis is true at the beginning of the loop:

   Before the first iteration: k=0, S=0. The first 0 numbers have sum zero (there are no numbers) => hypothesis true before entering the loop

Example: Proving the correctness of theSum algorithm (2)

Induction hypothesis: S= sum of the first k numbers

2. Maintenance: If hypothesis is true before step k, then it will be true before step k+1 (immediately after step k is finished)

We assume that it is true at beginning of step k: "S is the sum of the first k numbers"

We must prove that after executing step k, at the beginning of step k+1:

"S is the sum of the first k+1 numbers"

We calculate the value of S at the end of this step

K:=k+1, s:=s+a[k+1] => s is the sum of the first k+1 numbers

Example: Proving the correctness of theSum algorithm (3)

- Induction hypothesis: S= sum of the first k numbers

3. Termination: When the loop terminates,

the hypothesis implies the correctness of the algorithm

The loop terminates when k=n=> s= sum of first k=n numbers

=> postcondition of algorithm, DONE

Loop invariants and induction

- **Proving loop invariants is similar to mathematical induction**:
  - showing that the invariant holds before the first iteration corresponds to the base case, and showing that the invariant holds from iteration to iteration corresponds to the inductive step.

Mathematical induction - Review

- Let T be a theorem that we want to prove. T includes a natural parameter n.
- Proving that T holds for all natural values of n is done by proving following two conditions:
  1. T holds for n=1
  2. For every n>1 if T holds for n-1, then T holds for n

1= Base case
2= Inductive Step

Mathematical induction - Review

- **Strong Induction**: a variant of induction where the inductive step builds up on all the smaller values
- Proving that T holds for all natural values of n is done by proving following two conditions:
  1. T holds for n=1
  2. For every n>1 if T holds *for all k<= n-1*, then T holds for n

Mathematical induction review – Example1

- Theorem: The sum of the first n natural numbers is $n^*(n+1)/2$
- Proof: by induction on n
  1. Base case: If n=1, $s(1)=1=1^*(1+1)/2$
  2. Inductive step: We assume that $s(n)=n^*(n+1)/2$,

  and prove that this implies $s(n+1)=(n+1)^*(n+2)/2$, for all n>=1

  $s(n+1)=s(n)+(n+1)=n^*(n+1)/2+(n+1)=(n+1)^*(n+2)/2$

Mathematical induction review – Example2

- Theorem: Every amount of postage that is at least 12 cents can be made from 4-cent and 5-cent stamps.
- Proof: by induction on the amount of postage
- Postage (p) = m * 4 + n * 5
- **Base case:**
  - Postage(12) = 3 * 4 + 0 * 5
  - Postage(13) = 2 * 4 + 1 * 5
  - Postage(14) = 1 * 4 + 2 * 5
  - Postage(15) = 0 * 4 + 3 * 5

Mathematical induction review – Example2 Continue

- Inductive step: We assume that we can construct postage for every value from 12 up to k. We need to show how to construct k + 1 cents of postage. Since we have proved base cases up to 15 cents, we can assume that $k + 1 \geqslant 16$.
- Since $k+1 \geqslant 16$, $(k+1)-4 \geqslant 12$. So by the inductive hypothesis, we can construct postage for (k + 1) − 4 cents: $(k + 1) - 4 = m * 4 + n * 5$
- But then $k + 1 = (m + 1) * 4 + n * 5$. So we can construct k + 1 cents of postage using (m+1) 4-cent stamps and n 5-cent stamps

Correctness of algorithms

- Induction can be used for proving the correctness of repetitive algorithms:
  - Iterative algorithms:
    - Loop invariants
      - Induction hypothesis = loop invariant = relationships between the variables during loop execution
  - Recursive algorithms
    - Direct induction
      - Hypothesis = a recursive call itself ; often a case for applying *strong* induction

Example: Correctness proof for Decimal to Binary Conversion-

Algorithm Decimal_to_Binary

Input: n, a positive integer

Output: b, an array of bits, the bin repr. of n, starting with the least significant bits

```
t:=n;
k:=0;
While (t>0) do
    k:=k+1;
    b[k]:=t mod 2;
    t:=t div 2;
end
```

It is a repetitive (iterative) algorithm, thus we use loop invariants and proof by induction

Example: Correctness proof for Decimal to Binary Conversion-

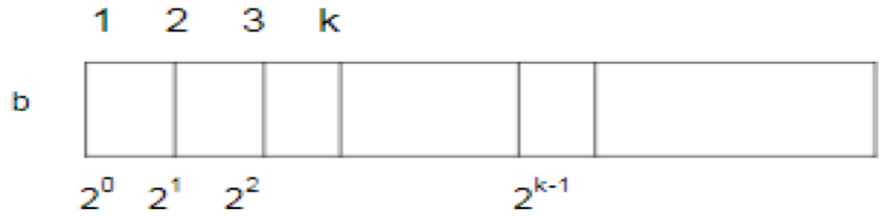
Algorithm Decimal_to_Binary

Input: n, a positive integer

Output: b, an array of bits, the bin repr. of n, starting with the least significant bits

```
t:=n;
k:=0;
While (t>0) do
    k:=k+1;
    b[k]:=t mod 2;
    t:=t div 2;
end
```

At step k, b holds the k least significant bits of n,and the value of t, when shifted by k, corresponds to the rest of the bits



Example: Correctness proof for Decimal to Binary Conversion-

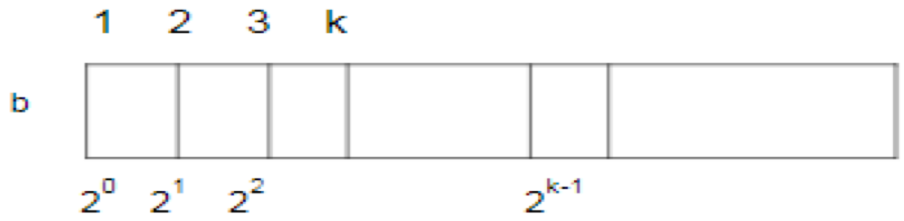
Algorithm Decimal_to_Binary

Input: n, a positive integer

Output: b, an array of bits, the bin repr. of n, starting with the least significant bits

```
t:=n;
k:=0;
While (t>0) do
    k:=k+1;
    b[k]:=t mod 2;
    t:=t div 2;
end
```

Loop invariant: If m is the integer represented by array b[1..k], then n=t*2k+m



Example: Proving the correctness of the conversion algorithm

- **Induction hypothesis=Loop Invariant**: If m is the integer represented by array b[1..k], then n=t*2^k+m
- **To prove the correctness of the algorithm, we must prove the 3 conditions:**
  1. ***Initialization: The hypothesis is true at the beginning of the loop***
  2. ***Maintenance: If hypothesis is true for step k, then it will be true for step k+1***
  3. ***Termination: When the loop terminates, the hypothesis implies the correctness of the algorithm***

Example: Proving the correctness of the conversion algorithm (1)

- Induction hypothesis: If m is the integer represented by array b[1..k], then n=t*2^k+m

1. The hypothesis is true at the beginning of the loop:

   k=0, t=n, m=0(array is empty)

   n=n*2^0+0

Example: Proving the correctness of the conversion algorithm (2)

- Induction hypothesis: If m is the integer represented by array b[1..k], then n=t*2^k+m

2. If hypothesis is true for step k, then it will be true for step k+1

At the start of step k: assume that n=t*2^k+m, calculate the values at the end of this step

If t=even then: t mod 2==0, m unchanged, t=t / 2, k=k+1=> (t / 2) * 2 ^ (k+1) + m = t*2^k+m=n

If t=odd then: t mod 2 ==1, b[k+1] is set to 1, m=m+2^k , t=(t- 1)/2, k=k+1 => (t-1)/2*2^(k+1)+m+2^k=t*2^k+m=n

Example: Proving the correctness of the conversion algorithm (3)

- Induction hypothesis: If m is the integer represented by array b[1..k], then n=t*2^k+m

3. When the loop terminates, the hypothesis implies the correctness of the algorithm

The loop terminates when t=0 =>

n=0*2^k+m=m

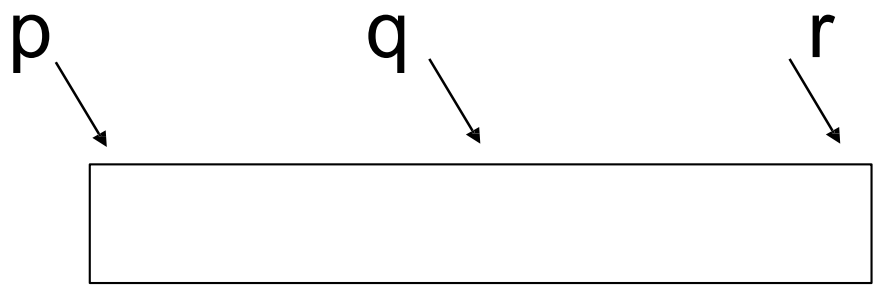
n==m, proved

Proof of Correctness for Recursive Algorithms

- To prove recursive algorithms, we have to:
  1. Prove the partial correctness (the fact that the program behaves correctly)
     - *we assume that all recursive calls with arguments that satisfy the preconditions behave as described by the specification, and use it to show that the algorithm behaves as specified*
  2. Prove that the program terminates
     - any chain of recursive calls eventually ends and all loops, if any, terminate after some finite number of iterations.

Example - Merge Sort

```
MERGE-SORT(A,p,r)
```




```
MERGE-SORT(A,p,q)
MERGE-SORT(A,q+1,r)
MERGE(A,p,q,r)

Precondition:

Postcondition:
```

Correctness proofs for Recursive Algorithm

n_1, n_2, … n_r are some values smaller than n but bigger than small_value

- **Base Case**: Prove that RECURSIVE works for n = small_value
- **Inductive Hypothesis:**
  - Assume that RECURSIVE works correctly for n=small_value, …, **k**
- **Inductive Step**:
  - Show that RECURSIVE works correctly for n = **k + 1**

Correctness proofs for Recursive Algorithm

- Proving that an algorithm is totally correct means:
  1. Proving that it will ***terminate***
  2. Proving the list of ***actions*** applied to the ***Precondition*** imply the ***postcondition***
- How to prove **repetitive algorithms:**
  - ***Iterative*** algorithms: use ***Loop invariants***, Induction
  - ***Recursive*** algorithms: use induction using as hypothesis the recursive call

That's all for this lecture!