# Introduction Analysis of Algorithm

Lecture Notes

09-11-2017

Dr. Eva Iwer

# History

- In the ninth century, in a town Khiva(later called Khowarizm) in Uzbekistan, lived an Arab mathematician called Abu Ja'farMohamed Ben Musa, nicknamed after his town: **Al-Khowarizmi**
- He wrote a book on arithmetic and algebra, which made the Arabic numerals and the decimal system known in Europe.
- The word "Algorism" was derived from his nickname, before it was later on transformed to "Algorithm"

# What is an Algorithm?

- An algorithm is a set of instructions used to solve a problem.
- The Oxford Dictionary defines on an algorithm as a "process or rules for calculation"
- Concept: An algorithm is a sequence of unambiguous instructions for solving a problem, i.e. for obtaining a required output for any legitimate input in a finite amount of time

# U2 Problem

- "U2" has a concert that starts in 17 minutes and they must all cross a bridge to get there. All four men begin on the same side of the bridge. You must help them across to the other side. It is night. There is one flashlight. A maximum of two people can cross at one time. Any party who crosses, either 1 or 2 people, must have the flashlight with them. The flashlight must be walked back and forth, it cannot be thrown etc.
- Each band member walks at a different speed. A pair must walk together at the rate of the slower man's pace:
- Bono: 1 minute to cross Edge: 2 minutes to cross Adam: 5 minutes to cross Larry: 10 minutes to cross
- For example: if Bono and Larry walk across first, 10 minutes have elapsed when they get to the other side of the bridge. If Larry then returns with the flashlight, a total of 20 minutes have passed and you have failed the mission.
- Note: There is no trick behind this. It is the simple movement of resources in the appropriate order. There are two known answers to this problem. This is based on a question Microsoft gives to all prospective employees.
- Note: Microsoft expects you to answer this question in under 5 minutes! There are no tricks to this like meeting halfway or anything. Good luck.

# What is algorithm analysis?

- Running time of the algorithm?
- Needed memory?
- Size of generated code?
- Generates correct results?
- Is it easy to read, modify and debug?
- How does it deal with unexpected input?

- Making a choice when implementing an algorithm to solve a problem
  - Make it easy to understand, code and debug?
  - Make efficient use of computer resources?
- What does the running time of algorithms depend on?

# Running time of algorithms depends on:

- Input of the algorithm
- Quality of the code generated by the compiler
- Nature and speed of the instructions on the machine where the algorithm is executed
- The **time complexity** of the algorithm underlying the program

# Running time depends on input and size of input

- Running time should be defined as a function of the input
- Example: Running time of a sorting algorithm is relative to the number of items being sorted
- More precisely, the exact running time on the particular input (Not just input size)

# An algorithm works correctly

- If on every instance of the problem it claims to solve it.
- If you find one instance of the problem which it is unable to find a correct answer, the algorithm is incorrect.
- Important to define its domain of definition
- Computer have limit of the size of instances – Algorithm not!

# Analysis of Algorithm

- How good is an algorithm?
  - Correctness
  - Time efficiency
  - Space efficiency
- Does there exist a better algorithm?
  - Lower bounds
  - Optimality

# Time efficiency

- Is analyzed by determining the number of reputations of the basic operation as a function of input size
- Basic Operation: The operation that contributes most toward the running time of the algorithm
- $T(n) \approx C_{op}C(n)$
- n = input size
- T(n) = running time
- $C_{op}$ = Execution time for basic operation
- C(n) = Number of times basic operation is executed

# QUESTION

- How much longer will the algorithm run if we double its size?
  - n
  - $n^2$
  - $log_2(n)$
  - $\sqrt{n}$
- How much longer will the algorithm run if we square its size?
  - $log_2(n)$

# Best, average and worst case analysis

- Running time depends on input order
- Searching in sorted list vs unsorted list
- Item found in beginning, middle or end of the list?
- Item doesn't exist?
- These cases represent the best, average and worst cases
- $c_{worst}(n) = max$
- $c_{best}(n) = min$
- $c_{avg}(n) = average$

# Example

- Search a key k in a n-size array
- Input: key k and A[0, …., n-1]
- Output: Index of first element of A that matches k or -1
- What is the best, the worst and the average case?

# Worst Case Analysis

- Analyzing worst case scenarios is important:
  - Guarantees that the algorithm will never perform worse than this case
    - Crucial for applications such as GAMES
  - Represents an upper bound on the performance
  - All other cases must have same to better performances
- Many algorithms have the same performance in the best case
  - Linear and binary search perform 1 comparison in the best case
  - Analyzing the best case is not reliable
- Many algorithms perform their worst cases most of the times
  - When searching for an item in a list, the item is mostly not found

# Average case is difficult to determine

- Depends on input meaning
- Unlike the best and worst cases, the average case is a range of values (Not precise values)
- •Assumptions are adopted when computing the average case, therefore it's not accurate

# Formal algorithm performance notation

- Algorithm's efficiency is relative to its worse case running time
- "Growth rate" is how fast performance decreases as input size increases
- "Growth rate" is important because it's machine independent
- Why do we care about large data?
- Usually, wrong results are generated when input size is small
- Running time increases as input size get larger

# Asymptotic order of growth

- A way of comparing, that ignored constant factors and small inputs
- The asymptotic behavior of a function f(n) refers to the growth of f(n) as n gets large
- IGNORE SMALL VALUES
  - We are interested in estimating how slow the program will be on large inputs
- The slower the asymptotic rate, the better the algorithm
- THINK BIG

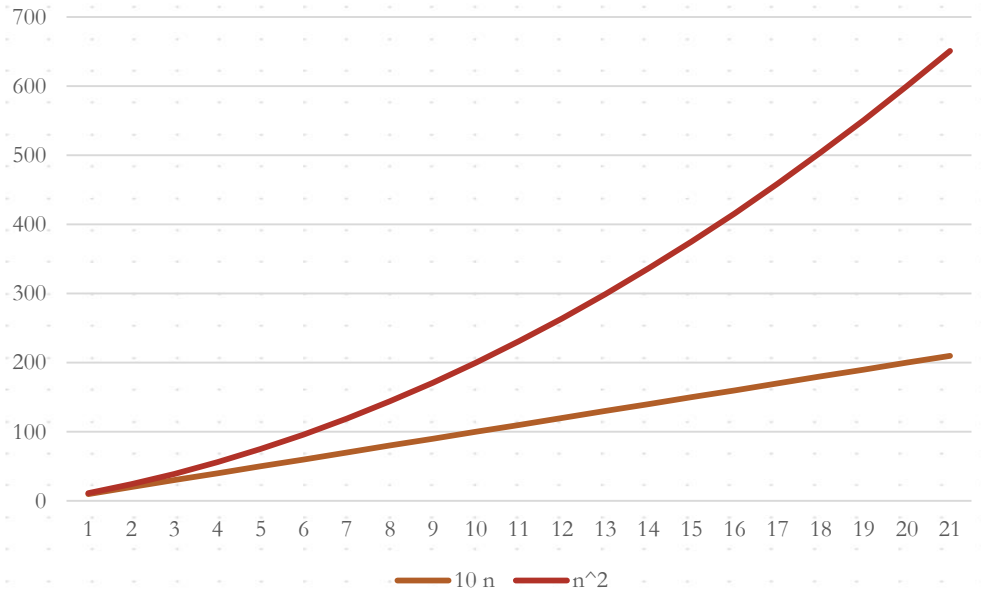
# Asymptotic Notations

- $O - notation$(Big O)
- Ω-notation (Big Omega)
- Θ-notation (Big Theta)

# O - notation

- A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n, i.e. if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \leq c * g(n) \; for \; all \; n \geq n_0$$



# Big O

- The O –Notation is for the upper bound
  - For a given function g(n), we denote by O (g(n)) the set of functions:

$$O(g(n)) = \{f(n): \exists\, c\ > 0, n_0 > 0 / 0 \leq f(n) \leq cg(n) \forall\, n\ \geq n_0\}$$

- The asymptotic O-notation represents the upper bound of a function within a constant factor
  - Only the higher order is considered
  - Lower orders are insignificant when n is large
  - The leading term's coefficient is ignored
  - Addition is performed by taking the maximum
    - $O(T_1) + O(T_2) = O(T_1+T_2) = \text{Max}(O(T_1), O(T_2))$
  - Multiplication is not changed, but written in a more compact way
    - $O(n) \times O(n) = O(n^2)$

# Examples

- $10n \in O(n^2)$
- $\sqrt{n} \in O(n)$
- $2n + 5 \in O(n)$
- $log_a n \in O(log_b n)$
- $n^2 \in O(n\ )$
- $\ln(n) \in O(\sqrt{n})$

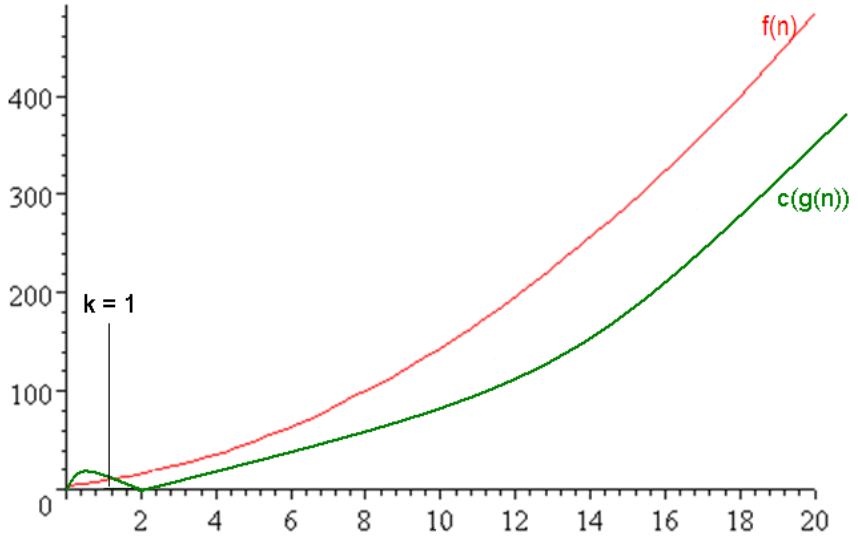
# Big O

- f(n) is O(n) means that the growth rate of f(n) is linear at worse
- f(n) is $O(n^2)$ means that the growth rate of f(n) is qudratic at worse
- Does this mean we should always pick a linear function over a quadratic one?
  - No!
  - The linear function might have a large overhead
  - $n^2$ milliseconds VS n decades
- Growth rate determines how a function is affected as the size of the input is increased
  - Does not determine the running time!

# Ω-notation

- A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below by some constant multiple of $g(n)$ for all large n, i.e. if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \geq c * g(n) for\ all\ n \geq n_0$$



# Ω-notation

- The Ω is for the lower bound
- For a given function g(n), we denote by Ω (g(n)) the set of functions:

$$\Omega(g(n)) = \{f(n): \exists\, c > 0, n_0 > 0 / 0 \leq cg(n) \leq f(n) \forall\, n \geq n_0\}$$

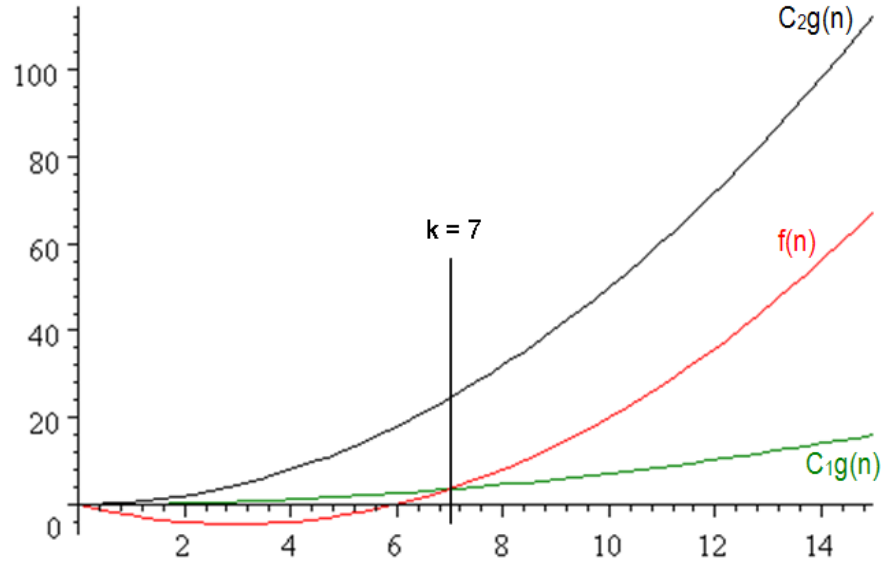
# Examples

- $\frac{n(n+1)}{2} \in \Omega(n)$

# Θ - notation

- A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded above and below by some constant multiple of $g(n)$ for all large n, i.e. if there exist some positive constants $c_1$ $and$ $c_2$ and some nonnegative integer $n_0$ such that

$$c_1\, g(n) \leq t(n) \leq c_2\, g(n) for\ all\ n \geq n_0$$



# Θ - notation

- The Θ is for the upper and lower bound
- For a given function g(n), we denote by Θ (g(n)) the set of functions:

$$\Theta(g(n)) = \{f(n): \exists\, c_1 > 0, c_2 > 0, n_0 > 0\ /$$

$$0 \leq c_1 \mathrm{g(n)} \leq f(n) \leq c_2 \mathrm{g(n)}\ \forall\, n \geq n$$

- f(n) = Θ(g(n)) means f(n) is within a constant multiple of g(n)

# Examples

- $\frac{n(n+1)}{2} \in \Theta(n^2)$

# Analysis of iterative algorithms

```
// Example for Analysis of iterative Algorithms
for (i = 0; i<n; i++)
{    for (j = 0; j < n; j++)
          F();
}
for (i = 0; i<n; i++)
{
     F();
     F();
     F();
}
F();
F();
F();
```

Assuming the cost of each "F()" function call is 3 units, and ignoring the cost of the "++" and "<" operations in the for loops, what is the running time of this algorithm?

# Analysis of iterative algorithms

- The running time of the previous algorithm is $T(n) = 3n^2 + 9n + 9$
- What's O(T(n))?
  - Lower orders are insignificant when n is large
  - The leading term's coefficient is ignored
  - O(T(n)) = O(3n2+ 9n + 9) = O(3n2) = O(n2)

# Running time of an algorithm

- An algorithm is made from a series of instructions within statements.
  - Cost of the instructions is measured in cycles
- The algorithm total cost is determined by adding all the statements costs together
  - Cost of statements within loops should be multiplied with the number of loops.

# Analysis of iterative algorithms

- Case n = 10
  - Running time for $3n^2$: $\frac{3(10)^2}{3(10)^2+9\,(10)+9} = 75,\ \%$
  - Running time for $9n$: $\frac{9(10)}{3(10)^2+9\,(10)+9} = 23\%$
  - Running time for $9$: : $\frac{9}{3(10)^2+9\,(10)+9} = 2\%$
- Case n = 100
  - Running time for $3n^2$: 97 %
  - Running time for $9n$: 3 %
  - Running time for $9$: 0.03 %
- Case n = 1000
  - Running time for $3n^2$ : 99,7 %
  - Running time for $9n$: 0.01 %
  - Running time for $9$: 0.0001%

# Classification of Algorithms

- Most algorithms have running times proportional to one of the following functions:
  - 1
  - $log_2$N
  - N
  - N$log_2$N
  - $N^2$
  - $N^3$
  - $2^N$
  - N!

# Classification of Algorithms

- 1
  - Running time is constant
  - Most instructions are executed once or at most few times
- $log_2$ N
  - Running time is logarithmic
  - Gets slightly slower when the size 'n' of the items grows
  - Occurs when large problems are solved by being transformed into smaller ones
    - Searching a sorted list using the binary search algorithm
    - Number if elements is divided by 2 after each iteration

# Classification of Algorithms

- N
  - Running time is linear
  - When N doubles, the running time doubles
  - Example: Searching in an unsorted list
- $Nlog_2N$
  - Occurs in algorithms that solve a problem by breaking it up into smaller sub-problems, solving them independently and then combining the solutions.
  - When N doubles, the running time is more than double
  - Example: Merge sort and quick sort
  - Input is processed by breaking the input data into smaller sets

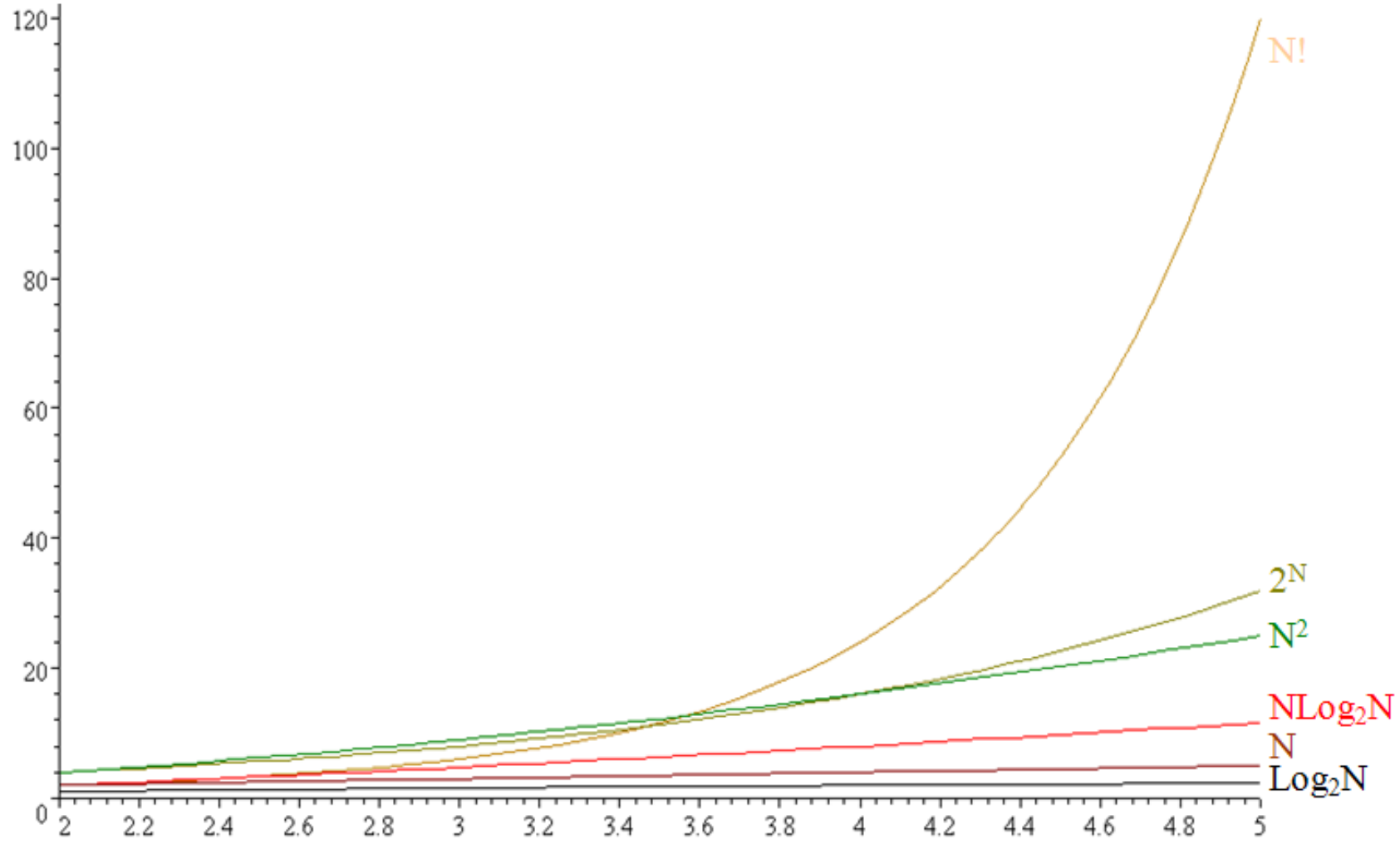
# Classification of Algorithm

- $N^2$
  - Running time is quadratic
  - Practical for relatively small problems
  - Occurs when data is processed in a double nested loop
  - Example: Exchange sort, bubble sort and insertion sort
  - N = 100: Running time is 10,000
- $N^3$
  - Running time is cubic
  - Practical of relatively small problems
  - Occurs when data is processed in a triple nested loop
  - Practical for small problems
  - N = 100: Running time is 1,000,000

# Classification of Algorithms

- $2^n$
  - Running time is exponential
  - Occurs in brute force algorithms, where all possible subsets of a set of data have to be generated
  - N = 100: Running time is 1267650600228229401496703205376
- N!
  - Running time is factorial
  - Occurs when the algorithm needs to generate all possible subsets of a set of data
  - N = 100: Running time is 9.33262154439441526816992388562667 e+157

# Classification of Algorithms



# Correctness

- Mathematical Induction
- Proof of Correctness

# Loop Invariant

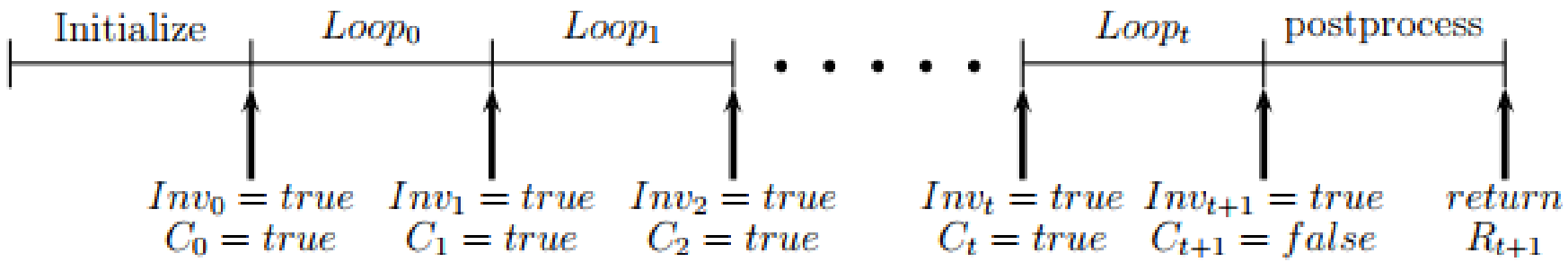
- A loop invariant is a property which is related to the variables in a loop, and is true at the beginning of each iteration.

# Proving Correctness

- Proving Correctness of an Iterative Algorithms:
- 1. State the loop invariant.
- 2. Prove that invariant holds for any number of iterations using mathematical induction:
  - (a) Prove that the loop invariant's base case holds – use values with index 0, that means use initialization values.
  - (b) Prove that if invariant holds after k iterations it will hold after k + 1.
- 3. Prove that the loop terminates.
- 4. Prove the correctness of the return value.

Here is a diagram of the algorithm execution:

- **Inv** is the invariant
- **C** is the loop condition
- enumerate iteration starting index 0 – in which case index $i$ means "the value of the local variable (or invariant, or loop conditional) after $i$ iterations of the loop".
- then during the $i$ iteration (marked as $Loop_i$ on the diagram) the indices of the local variables change from $i$ to $i+1$.
- We only look at values "in between" iterations – think during the while-loop condition checks and during the return statement. Those position are marked with tick on the diagram.
- the index $t$ is the index of the last iteration, during which indices of the local variables change from $t$ to $t+1$. Loop condition fails during the next check ($C_{t+1}$ is false) and we go to code after the loop. Remember – all variable have indices $t+1$.



# Prove that ALG1(A, B) returns AB

```
ALG1(A, B) // A,B are natural numbers
{
    S = 0
    I = 0
    while (I < B)
    {
        S = S + A
        I = I + 1
    }
    return S
}
```

# Prove the fast exponentiation function F E(A, M) returns $A^M$

```
FE(A,M)
{
    B = A;
    E = M;
    R = 1;
    while(E > 0)
    {
        if(E is odd)
        {
            R = R * B;
            E = E - 1;
        }
        else
        {
            B = B * B;
            E = E / 2;
        }
    }
    return R;
}
```