Introduction to Computer Science
Shimon Schocken
IDC Herzliya

Lecture 8-3

Algorithms

Abu Abdullah
Muhammad ibn Musa
al-Khwarizmi

Outline

Introduction

- Computational problems
- Algorithms

Search algorithms

- Sequential search
- Binary search
- Comparison

Running-time analysis

- Performance monitoring
- Order of ...

Sort algorithms

- Selection sort
- Insertion sort
- Merging
- Merge sort

Typical run-time functions

Proof techniques

- Induction
- Contradiction

Square root by binary search

- Algorithm
- Correctness proof

GCD algorithm

- Algorithm
- Correctness proof

Binary search

- Correctness proof

Proof by Induction

A predicate P is stated.

To prove by induction that P is true for every natural number n, we do as follows:

- Base step: We prove that P is true for 0 (or for 1)
- Inductive hypothesis: We assume that P is true for k
- Induction step: We prove that if P is true for k, it follows that P is true for k+1.

Example: prove that $1 + 2 + 3 + \ldots + n = \frac{1}{2} n (n + 1)$

Strong induction:

- Base case: Prove that P is true for 0 (or 1)
- Inductive hypothesis: Assume that P(i) is true for all numbers 0 (or 1) <= i <= k
- Inductive step: Given the inductive hypothesis, prove that P(k+1) is true.

Proof by contradiction

A predicate P is stated.

To prove by contradiction that P is true, we do as follows:

- Base assumption: Assume that P is false
- Proof: Start with the base assumption and show that some known property/fact is false
- Conclude: That since the only thing that could be false in the proof is the base assumption, the base assumption must be false (meaning that P is true).

Example: prove that there is an infinite number of primes.

The proof is based on the fact that every number is either a prime or a product of primes.

Base assumption: the assertion is false: there is a largest prime p_k.

Let $p_1, p_2, \dots, p_k$ be all the primes and consider the following number:

$N = p_1 \times p_2 \times \dots \times p_k + 1$

N is larger than p_k, so N is not prime. So, N must be a product of some of the primes $p_1, p_2, \dots, p_k$. But, none of these primes divides N, so N is not a product of any of the primes.

We've reached a contradiction, leading to the conclusion that the assertion must be true.

Proof why induction works (by contradiction)

<u>Theorem:</u> If we prove by induction that P is true, then P must be true for all numbers.

<u>Proof</u> (by contradiction):

Suppose we proved by induction that P is true for all numbers 1 .. n.

Suppose now that P is actually false for some numbers. Therefore, there exists a smallest $k \leq n$ for which P(k) is false.

In the induction's base case, we showed that P(1) is correct. Therefore it must be that $k > 1$.

Since k is the smallest value for which P(k) is false, it must be that P(k-1) is true.

But, in the induction step, we showed that if P(k-1) is true, it must be that P(k) is also true.

Contradiction: P(k) cannot be false for any $1 \leq k \leq n$

Therefore the theorem is correct and the proof by induction method works.

$$\forall \text{ predicates } P,\ (P(0) \wedge \forall k[P(k) \Rightarrow P(k+1)]) \Rightarrow \forall n P(n)$$

Outline

Introduction

- Computational problems
- Algorithms

Search algorithms

- Sequential search
- Binary search
- Comparison

Running-time analysis

- Performance monitoring
- Order of ...

Sort algorithms

- Selection sort
- Insertion sort
- Merging
- Merge sort

Typical run-time functions

Proof techniques

- Induction
- Contradiction

Square root by binary search

- Algorithm
- Correctness proof

GCD algorithm

- Algorithm
- Correctness proof

Binary search

- Correctness proof

Square root by binary search

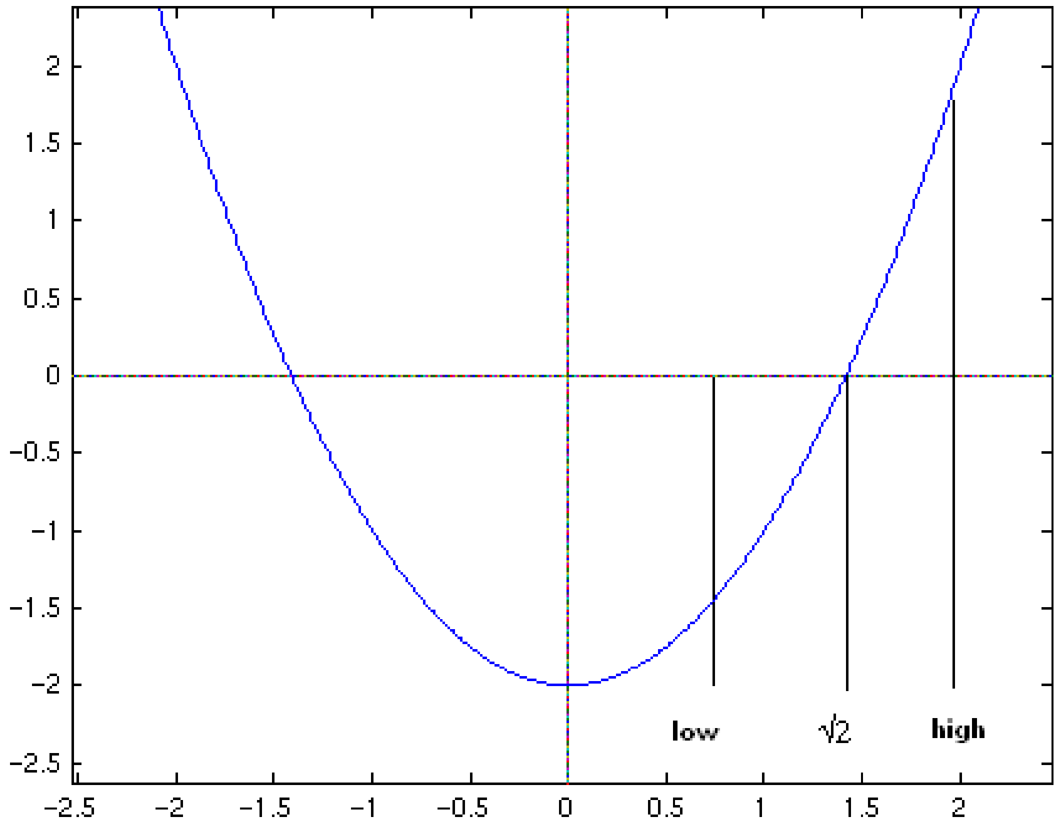
Input: a positive real number *x*, and a precision requirement ε

Output: a real number *r* such that $|r-\sqrt{x}| \leq \varepsilon$

```
// Computes sqrt(x) with an epsilon precision
sqrt(x, epsilon):
  low = 0
  high = x
  while (high - low > epsilon)
    mid = (high + low) / 2
    if (mid * mid > x)
       high = mid
    else
       low = mid
  return low
```

To find $\sqrt{2}$,

we solve $f(x) = x^2 - 2 = 0$



<u>Mean Value Theorem</u>:

if f(low) < 0 and f(high) > 0
then there is x, low<x<high
with f(x)=0.

Sample run

```
sqrt(x, epsilon) {
  low = 0
  high = x
  while (high - low > epsilon) {
    mid = (high + low) / 2
    if (mid * mid > x)
       high = mid
    else
       low = mid
  }
  return low
}
```

__General observation:__

Binary search can be used to approximate the value of any function f(x) as long as f is continuous and monotonous and you know how to compute its inverse.

Sample run: Computes sqrt(2) with precision 0.05

	mid	mid*mid	low	high
After 0 rounds	--	--	0	2
After 1 round	1	1	1	2
After 2 rounds	1.5	2.25	1	1.5
After 3 rounds	1.25	1.56..	1.25	1.5
After 4 rounds	1.37..	1.89..	1.37..	1.5
After 5 rounds	1.43..	2.06..	1.37..	1.43..
After 6 rounds	1.40..	1.97..	1.40..	1.43..

Output: 1.40..

Algorithm correctness

Loop invariant lemma:

At each step of the algorithm low $\leq \sqrt{x} \leq$ high.

Proof (by induction on the iteration number):

Base case: in iteration 0 we have

low = 0 $\leq \sqrt{x} \leq$ high = x

Induction step: in iterations > 0:

If mid > $\sqrt{x}$ the code sets high = mid
and thus high > $\sqrt{x}$

If mid $\leq \sqrt{x}$ the code sets low = mid
and thus low $\leq \sqrt{x}$

Theorem: When the algorithm terminates it returns a value r that satisfies $|r - \sqrt{x}| \leq \varepsilon$.

Proof: The algorithm terminates when high - low $\leq \varepsilon$, and returns low.

At this point, by the lemma:

low $\leq \sqrt{x} \leq$ high $\leq$ low + ε.

Thus low $\leq \sqrt{x} \leq$ low + ε

Thus $|\text{low} - \sqrt{x}| \leq \varepsilon$.

```
sqrt(x, epsilon) {
  low = 0
  high = x
  while (high - low > epsilon) {
    mid = (high + low) / 2
    if (mid * mid > x)
       high = mid
    else
       low = mid
  }
  return low
}
```

Open questions:

- Does the algorithm always terminate?
- How Fast?

Running-time

In each iteration, the value of (*high-low*) decreases by a factor of 2.

At the beginning, (*high-low*) = x; at the end, (*high-low*) goes below ε

How many times can you divide x by 2 before it goes below ε?

Answer: $\log_2(x / \varepsilon) = \log_2 x + \log_2 \varepsilon^{-1}$

Thus the run-time is order of $\log_2 x + \log_2 \varepsilon^{-1}$

```
sqrt(x, epsilon) {
  low = 0;
  high = x;
  while (high-low > epsilon) {
    mid = (high+low)/2;
    if (mid*mid > x)
       high = mid;
    else
       low = mid;
  }
  return low;
}
```

Outline

Introduction

- Computational problems
- Algorithms

Search algorithms

- Sequential search
- Binary search
- Comparison

Running-time analysis

- Performance monitoring
- Order of ...

Sort algorithms

- Selection sort
- Insertion sort
- Merging
- Merge sort

Typical run-time functions

Proof techniques

- Induction
- Contradiction

Square root by binary search

- Algorithm
- Correctness proof

GCD algorithm

- Algorithm
- Correctness proof

Binary search

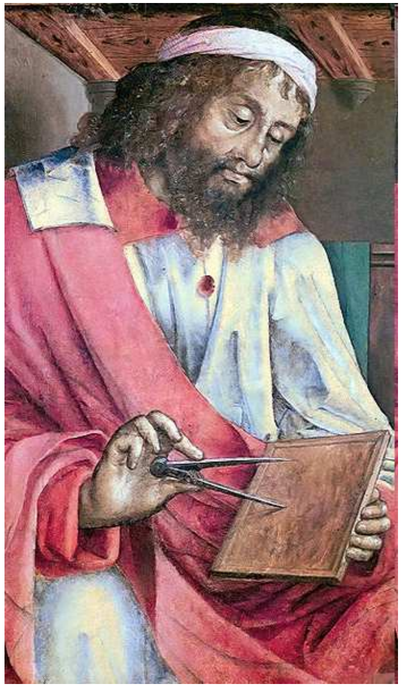
- Correctness proof

Greatest Common Divisor (GCD)

- Published by Euclid 2,200 years ago
- Definition: The GCD of two natural numbers x, y is the largest integer j that divides both numbers (without remainder).
- Notation: we say that j is the largest number such that j|x, and j|y.
- The GCD Problem: Input: Two natural numbers x, y ; Output: GCD(x,y)

Euclid's GCD Algorithm

```
gcd(x,y) {
  while (y != 0) {
    rem = x % y
    x = y
    y = rem
  }
  return x
}
```



Euclid
(born 300 BC)

Sample run of Euclid's algorithm

Euclid's GCD Algorithm

```
gcd(x,y) {
  while (y != 0) {
    rem = x % y
    x = y
    y = rem
  }
  return x
}
```

Example: GCD(72,120)

	rem	x	y
After 0 rounds	--	72	120
After 1 rounds	72	120	72
After 2 rounds	48	72	48
After 3 rounds	24	48	24
After 4 rounds	0	24	0

Output: 24

Observations:

- 24 is not only the GCD of 72 and 120, it is also the GCD of x and y in every iteration
- Y becomes smaller in every iteration.

Correctness of Euclid's algorithm

Theorem:

When Euclid's GCD(x,y) algorithm terminates, it returns the GCD of x and y

Notation: Let g = GCD(x,y) for the original values of x and y

Loop Invariant Lemma:

For all steps $k \geq 0$, GCD(x,y) = g for the current values of x and y.

(proof in next slide).

Euclid's GCD Algorithm

```
gcd(x,y) {
  while (y != 0) {
    rem = x % y
    x = y
    y = rem
  }
  return x
}
```

Proof of the theorem:

The method returns x when y=0.

By the loop invariant lemma, at this point GCD(x,y) = g.

But GCD(x,0) = x for every x (since x|0 and x|x).

Thus g = x, which is the value returned by the method.

Still Missing: The algorithm always terminates.

Correctness of Euclid's algorithm (proof of the loop invariant lemma)

Support Lemma: For all integers x, y : $GCD(x,y) = GCD(x\%y , y)$

Proof: Let $x = ay + b$, where $y > b \geq 0$. Thus $x\%y = b$.

(1) If $g|x$, and $g|y$, we also have $g|(x-ay)$, i.e. $g|b$.
Thus $GCD(b,y) \geq g = GCD(x,y)$.

(2) Let $g' = GCD(b,y)$, then $g'|(x-ay)$ and $g'|y$, so we also have $g'|x$.
Thus $GCD(x,y) \geq g' = GCD(b,y)$.

(3) It follows that $GCD(x,y) \geq GCD(b,y) \geq GCD(x,y)$.

Therefore $GCD(x,y) = GCD(b,y) = GCD(x\%y , y)$

Loop Invariant Lemma:

For all steps $k \geq 0$, $GCD(x,y) = g$ for the current values of x and y.

Proof: By induction on k.

Base step: For $k = 0$, x and y are the original values so clearly $GCD(x,y) = g$.

Induction step:

- Let x, y denote the values after k steps. We assume that $GCD(x,y) = g$.
- Let x', y' denote the values after k+1 steps.

  We need to show that $GCD(x',y') = GCD(x,y)$.

  According to the code: $x' = y$ and $y' = x \% y$.
  Thus the proof follows from the support lemma.

Outline

Introduction

- Computational problems
- Algorithms

Search algorithms

- Sequential search
- Binary search
- Comparison

Running-time analysis

- Performance monitoring
- Order of ...

Sort algorithms

- Selection sort
- Insertion sort
- Merging
- Merge sort

Typical run-time functions

Proof techniques

- Induction
- Contradiction

Square root by binary seacrh

- Algorithm
- Correctness proof

GCD algorithm

- Algorithm
- Correctness proof

Binary search

- Correctness proof

Proof by induction that the binary search algorithm finds the correct value

<u>Theorem:</u>

if a value exists in a sorted array, the binary search algorithm will find it.

<u>Proof:</u> by induction on k = the array's length

Base step: if k = 0 then low = 0 and high = 0 – 1 = -1. Therefore low > high and the algorithm will report failure correctly.

Inductive hypothesis: Assume that we can correctly find the value in sorted arrays of size $0 \le i \le k-1$. We will prove that we can also find the value correctly in sorted arrays of size k.

Inductive step: According to the algorithm, we look at A[$\frac{1}{2}$ k]. There are three cases:

1. If A[$\frac{1}{2}$ k] = searched value, then the algorithm found it.

2. If A[$\frac{1}{2}$ k] > searched value, then since the array is sorted, the searched value must exist somewhere in the range A[0 .. $\frac{1}{2}$ k]. The length of this sorted array is less than k. Therefore, according to the inductive hypothesis, the algorithm will find it.

3. If A[$\frac{1}{2}$ k] < searched value, then since the array is sorted, the searched value must exist somewhere in the range A[($\frac{1}{2}$ k)+1 .. n]. The same argument follows.

```
// Find x in a sorted array
// by binary search

low = 0
high = N-1;
while (low <= high) {
  med = (low + high) / 2
  if (x = A[med])
    return med
  if (x < A[med])
    high = med - 1
  else
    low = med + 1
}
return -1
```