

Lectures 8.1 - 8.2

Algorithms

Computational problems

A computational problem describes an input-output relationship. Examples:

- Prime number problem:
 - Input: an integer number
 - Output: 1 if the number is prime, 0 otherwise
- Sorting problem:
 - Input: A list of numbers
 - Output: Same list, sorted
- File compression problem:
 - Input: A file
 - Output: A compressed file
- Image indexing problem:
 - Input: A digital image
 - Output: An English description of the picture
- Travelling salesman problem (TSP):
 - Input: A list of cities and distances among them
 - Output: The minimal distance route that visits every city exactly once.
- Etc.

Algorithms

Algorithm: A specification of how to solve a computational problem

- An algorithm must be specific and well-defined, so that one can write a program from it without any doubt or misunderstanding
- An algorithm must work for all possible inputs of the problem
- We wish algorithms to be:
 - Correct: produce the correct output for each input
 - Efficient: run as quickly as possible, using as little memory as possible
- There are usually many different algorithms for each computational problem
- We will sometimes use pseudo code to describe algorithms.

Example:

```
// Testing whether input N is prime:
for j = 2 .. N-1
  if j | N
    output "N is composite" and stop
output "N is prime"
```

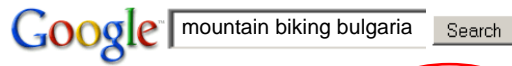
Our pseudo code rules

- A liberal combination of "commands" and "English"
- Code blocks are implied by the indentation; {} are sometimes used
- The objective: Self-explanatory algorithm specification.

Outline

- Introduction
 - Computational problems
 - Algorithms
- ➔ ■ Search algorithms
 - Motivation
 - Sequential search
 - Binary search
 - Comparison
- Sorting algorithms
 - Selection sort
 - Insertion sort
- Running-time analysis
 - Definition / big O notation
 - Examples
- Induction proof technique
 - Definition
 - Example
- More algorithms
 - Euclid's GCD algorithm
 - Square root by binary search
 - Newton-Raphson method
 - MergeSort
- More fun to go

Motivation: Search engine in action



Results 1 - 10 of about 920,000 for [mountain biking bulgaria](#). (0.06 seconds)

[Mountain biking in Bulgaria - travel and tourism](#)

Bulgarian Travel Guide www. **P e a k V i e w . b g** - Tourist Clubs **Mountain Biking ...**
Interests: **mountain bike**, computers, design, skiing, caves. ...
www.peakview.bg/biking/main.htm - 9k - [Cached](#) - [Similar pages](#) - [Note this](#)

[Biking and bike tours in Bulgaria](#)

Biking and bike tours in Bulgaria - mountain biking, ... Please, make your inquiries for individual and group **biking in Bulgaria** here, or Call: + 359 885 ...
www.motoroads.com/tours_bulgaria_biking.html - 56k - [Cached](#) - [Similar pages](#) - [Note this](#)

[Mountain biking in Vitosha near Sofia, Bulgaria](#)

Mountain cycling in Bulgaria, Vitosha. Rent a **bike** in Sofia!
www.sofiahotels.net/sofia/city_info/mountain_biking_en.html - 20k - [Cached](#) - [Similar pages](#) - [Note this](#)

[Mountainbiking in Bulgaria - Rodopi, Pirin and Rila Mountains ...](#)

Biking - 35km asphalt/dirt road downhill Overnight in **mountain** chalet - doubles. ... stopping to visit Rila Monastery, **Bulgaria's** finest cultural site. ...
www.infohub.com/TRAVEL/SIT/sit_pages/12130.html - 19k - [Cached](#) - [Similar pages](#) - [Note this](#)

[DudeGirl.com - Dude Girl Trips - Bulgaria - Rila Mountains](#)

The Rough Guide to **Bulgaria**. Learn about our 2004 **mountain biking** trip in the Rila Mountains! Cozy outdoor dining areas like this one are the perfect ...
www.dudegirl.com/dg_trips_rila.htm - 24k - [Cached](#) - [Similar pages](#) - [Note this](#)

Search engine -- behind the scene

- The Google index:
 - A list of words; each word has a list of URL's that mention it
 - Indexing robots work hard to maintain this index continuously
- Typical search scenario:
 - User enters some keywords
 - Google *searches* the index
 - Google returns a list of URLs that mention this word; the list is sorted by PageRank
- The search engine must be
 - Reliable
 - Efficient
- Opening the black box:
 - Searching algorithms
 - Sorting algorithms.

keyword	URLs
alpine	11,4,5
biking	2,11
cnn	13,100,1,7
bulgaria	4,83
dance	12,41
england	1,7,4,5
jaguar	17
jordan	81,9
linguist	10,3,5,4
mountain	9
oslo	5,11,12,95
premium	17,2,8
robert	5,17
russia	3,5,7,9,41
skeleton	19
truth	21,55

Sequential search in practice



- Input: a word x and a list of N words
- Output: if x is found, its location; else -1
- Strategy: march through the list

Running-time analysis

- We always think about the worst-case
- Worst-case time for searching a given item among n items: n steps
- Program set up time: a cycles;
Each algorithmic step takes c cycles;
Total worst-case running-time: $a + cN$ cycles
- Since N dominates the picture, we say that the running time is $\sim N$.

keyword	URLs
alpine	11,4,5
biking	2,11
cnn	13,100,1,7
bulgaria	4,83
dance	12,41
england	1,7,4,5
jaguar	17
jordan	81,9
linguist	10,3,5,4
mountain	9
oslo	5,11,12,95
premium	17,2,8
robert	5,17
russia	3,5,7,9,41
skeleton	19
truth	21,55

Binary search in practice



Algorithm:

- Input: a word x and a sorted list of N words
- Output: if x is found, its location; else -1
- Strategy (informal): Divide and conquer

Running-time analysis:

- How many times can we divide N by 2?
- Run-time: $\sim \log_2 N$

keyword	URLs
alpine	11,4,5
biking	2,11
cnn	13,100,1,7
bulgaria	4,83
dance	12,41
england	1,7,4,5
jaguar	17
jordan	81,9
linguist	10,3,5,4
mountain	9
oslo	5,11,12,95
premium	17,2,8
robert	5,17
russia	3,5,7,9,41
skeleton	19
truth	21,55

Outline

- Introduction
 - Computational problems
 - Algorithms
- Search algorithms
 - Motivation
 - ➔ Sequential search
 - Binary search
 - Comparison
- Sorting algorithms
 - Selection sort
 - Insertion sort
- Running-time analysis
 - Definition / big O notation
 - Examples
- Induction proof technique
 - Definition
 - Example
- More algorithms
 - Euclid's GCD algorithm
 - Square root by binary search
 - Newton-Raphson method
 - MergeSort
- More fun to go

Sequential search

Array A:

0	1	1	3	4	5	6	7	8	9
20	7	51	97	2	9	72	83	91	15

```
// Find the index of x in an array
for i = 0 .. N-1
  if A[i] = x
    return i;
return -1;
```

If the array is of size N , how many steps will it take to find an item

- In the best case?
- In the worst case?
- On average?

Efficiency

	0	1	1	3	4	5	6	7	8	9
Array A:	20	7	51	97	2	9	72	83	91	15

```
// Find the index of x in an array
for i = 0 .. N-1
  if a[i] = x
    return i;
return -1;
```

Suppose that the input size is N values

- If x is the j-th value in the list, the search will terminate after j steps
- Average-case run-time:
on average, the number of steps = $(1 + 2 + 3 + \dots + N) / N = \frac{1}{2}(N + 1)$
- Worst-case run-time: the number of steps will be N

Binary search

	0	1	1	3	4	5	6	7	8	9
Sorted array (A):	2	7	9	15	20	51	72	83	91	97

```
// 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
```

Sample run (x = 72):

Iteration	low	high	med	Test
0	0	9	4	72 > 20
1	5	9	7	72 < 83
2	5	6	5	72 > 51
3	6	6	6	72 = 72

Observation:

Looking for *any value* in this array will always take at most 4 steps.

Efficiency

```

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
    
```

Divide
and
Conquer!

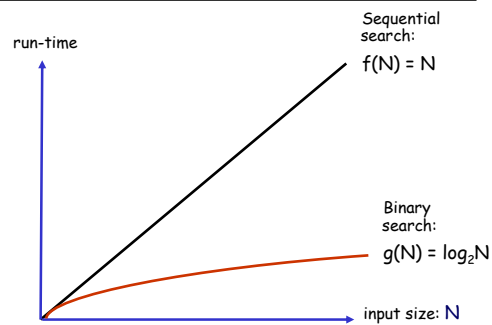
How many iterations in this loop?

- In each iteration we halve the value of (*high - low*)
- At the beginning, (*high - low*) = $N - 0 = N$
- How many times can you halve N before it becomes less than 1?
Answer: $\log_2 N$

Thus, the number of steps to find any value is $\log_2 N$

Running-time comparison

Input size: n	Seq. Binary	
	Run-time: n	$\log_2 n$
8	8	3
16	16	4
32	32	5
64	64	6
100	100	7
1,000	1,000	10
1,000,000	1,000,000	20
1,000,000,000	1,000,000,000	30



- Why $\log_2 N$ is so attractive?
- Because $\log_2(2N) = \log_2 N + 1$
- A search engine has to search 1 billion records; it takes 30 steps;
Sometimes soon it will have to search 2 billion records; this will take 31 steps
- Each time the size of the Internet doubles, there is one more step to do
- Not bad ...

Outline

- Introduction
 - Computational problems
 - Algorithms
- Search algorithms
 - Motivation
 - Sequential search
 - Binary search
 - Comparison
- ➔ ■ Sorting algorithms
 - Selection sort
 - Insertion sort
- Running-time analysis
 - Definition / big O notation
 - Examples
- Induction proof technique
 - Definition
 - Example
- More algorithms
 - Euclid's GCD algorithm
 - Square root by binary search
 - Newton-Raphson method
 - MergeSort
- More fun to go

Selection sort

```
for j = 0 .. N-1
  min = j
  for k = j+1 .. N
    if (A[k] < A[min])
      min = k
  // Switch
  if (min != j)
    temp = A[min]
    A[min] = A[j]
    A[j] = temp
```

Unsorted:	13	10	7	3	15	1	4	11
Step 0:	1	10	7	3	15	13	4	11
Step 1:	1	3	7	10	15	13	4	11
Step 2:	1	3	4	10	15	13	7	11

Etc.

Efficiency

```
for j = 0 .. N-1
  min = j
  for k = j+1 .. N
    if (A[k] < A[min])
      min = k
  // Switch
  if (min != j)
    temp = A[min]
    A[min] = A[j]
    A[j] = temp
```

Let N be the size of the array we have to sort

In step 0, we scan N-1 numbers

In step 1, we scan N-2 numbers

In step 2, we scan N-3 numbers

Etc.

Thus, the number of steps is $(N-1) + (N-2) + \dots + 1 = \frac{1}{2} N(N-1) = \frac{1}{2} N^2 - \frac{1}{2} N$
= order of magnitude of N^2

How good is N^2 ?

If you have to sort 100,000 numbers, it will take 10,000,000,000 steps.

Insertion sort

Unsorted:	10	2	14	5	4	18	3	7
Step 0:	2	10	14	5	4	18	3	7
Step 1:	2	10	14	5	4	18	3	7
Step 2:	2	5	10	14	4	18	3	7
Step 3:	2	4	5	10	14	18	3	7

Etc.

Basic idea

- In step j, entries $A[0] \dots A[j-1]$ are sorted.
We move item $A[j]$ to the left until it reaches its correct location
- This requires only pair-wise switches
- The correct location of $A[j]$ within $A[0 \dots j-1]$ can be found using either sequential search or binary search.

Efficiency

```
// Sorts an array A of length N using insertion sort
for k = 1 .. N-1 {
  x = A[k]
  j = k - 1
  // insert x to its right place
  while (A[j] > x and j >= 0) {
    A[j+1] = A[j]
    j--
  }
  A[j+1] = x
}
```

Efficiency

How many switches we have to do in the worst case?

$$1 + 2 + 3 + \dots + N = \frac{1}{2} N(N+1) = \\ = \text{order of } N^2$$

Outline

- Introduction
 - Computational problems
 - Algorithms
- Search algorithms
 - Motivation
 - Sequential search
 - Binary search
 - Comparison
- Sorting algorithms
 - Selection sort
 - Insertion sort
- ➡ ■ Running-time analysis
 - Definition / big O notation
 - Examples
- Induction proof technique
 - Definition
 - Example
- More algorithms
 - Euclid's GCD algorithm
 - Square root by binary search
 - Newton-Raphson method
 - MergeSort
- More fun to go

Running time analysis

- When we analyze the running time of a given algorithm, we have to count all the basic operations that the algorithm performs:
 - Arithmetic: `(low + high)/2`
 - Comparison: `if (x > 0) ...`
 - Assignment: `temp = x`
 - Branching: `while (true) { ... }`
 - Etc,
- But ...
 - These are not machine-level operations
 - Not all operations take the same amount of time
 - Different operations take different times on different hardware and software platforms.

Example: testing the Java performance of your computer

```
import java.util.*;

public class PerformanceEvaluation {
    public static void main(String[] args) {
        int i = 0;
        double d = 1.618;
        SimpleObject obj;
        final int numIterations = 100000000;
        long startTime = System.currentTimeMillis();

        for (i=0 ; i<numIterations ; i++){
            // Put here the operation you wish to time
            // d = 1.0 / d;
            // obj.m();
            // obj = new SimpleObject();
        }

        long duration = System.currentTimeMillis() - startTime;
        System.out.println("Duration in ms: " + duration);
    }
}

class SimpleObject {
    private int x = 0;
    public void m() { x++; }
}
```

	Body of the loop	Run-time (in ms) on Shimon's home PC
Loop overhead:	--	1,047
Double division:	<code>d=1.0/d;</code>	16,140
Method call:	<code>obj.m();</code>	2,406
Object creation:	<code>o = new SimpleObject();</code>	10,937

Running time analysis

The running time of a any given program depends upon:

- The algorithm
- The input
- The language used for the implementation
- The compiler used
- The OS
- The computer hardware
- Other programs running on the computer
- And more.

Formal run-time analysis:

We wish to neutralize all these platform-specific details, focusing instead on the **running-time of the algorithm as a function of one thing only: the input size.**

Running time analysis

- We seek a function `runningTime(N)` which will be invariant over hardware, languages or compilers.

Example:

```
// Create a multiplication table of size N
for j = 0 .. N-1
  for k = 0 .. N-1
    print j * k;
  print // move to next line
}
```

- Ignoring all constant factors, the running-time of this algorithm is about N^2
- Big O notation: We say that the running-time is $O(N^2)$
- Importantly, we don't analyze the running-time of the code; Rather, we analyze the running time of the algorithm.

Running time analysis: Asymptotic and focused on the worst-case

When asked to analyze the performance of some algorithm, we do an *asymptotic worst case* analysis of its running time

Asymptotic:

- Formal, exact, depends only on the algorithm
- Focuses on large input sizes

Worst Case:

- Our analysis must hold for *all* inputs
- Thus we consider the worst-case input

Big O notation: Given input size N , the running time will be expressed as $O(\text{some nicely stated function of } N)$

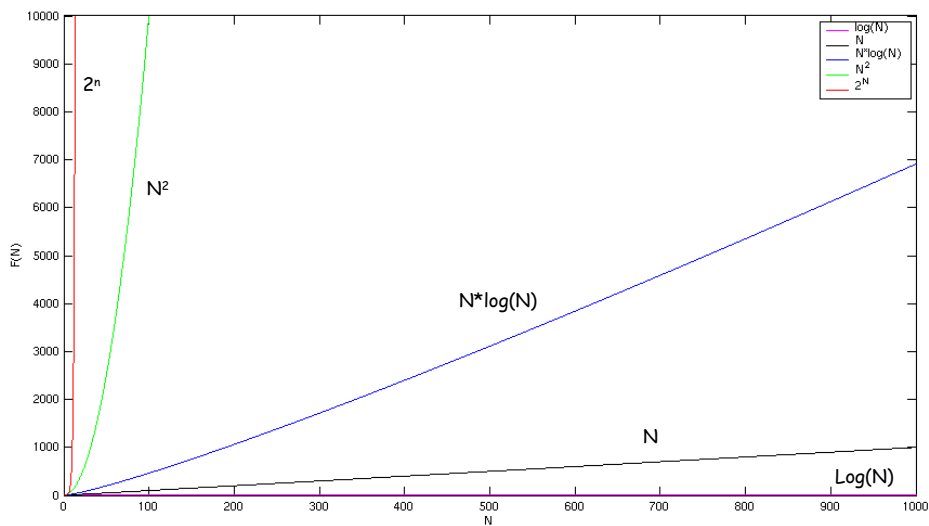
Examples:

- Sequential search: $O(N)$
- Binary search: $O(\log_2 N)$
- Selection sort: $O(N^2)$
- Etc.

A more formal definition of running-time:

- When we say that the running-time of an algorithm is $O(f(n))$, we mean that there exists some constant C such that the actual running-time $\leq C f(n)$
- Thus, $f(n)$ provides an upper bound on the running-time, up to a constant.

Functions that typically come up in run-time analysis



Counting primes (Version 1)

```
// Counts how many primes exist up to N
numPrimes = 0
for i = 2 .. N {
  isPrime = true;
  for j = 2 to i
    if j|i
      isPrime = false;
  if isPrime
    numPrimes++;
}
```

Formal analysis:

- Number of iterations is $1 + 2 + 3 + \dots + N = \frac{1}{2}(N^2 + N)$
- Running-time is $O(N^2)$

Empirical analysis: (Using the program from next slide)

Input size (N)	10K	20K	30K	40K
Run-time (T)	0.78s	3s	6.8s	12s
N_k / N_0	1	2	3	4
T_k / T_0	1	4	9	16

A quadratic growth function

Can we do better ?

Counting primes (Version 1, Java implementation, with time stamps)

```
public class CountPrimes {
  final static int n = 10000;
  public static void main (String args[]) {
    int numPrimes = 0;
    boolean isPrime;
    long startTime = System.currentTimeMillis();
    int numPrimes = 0;
    for (int i = 2; i < N; i++) {
      isPrime = true;
      for (int j = 2; j < i; j++)
        if (i % j == 0)
          isPrime = false;
      if (isPrime)
        numPrimes++;
    }
    System.out.println("There are " + numPrimes + " primes below " + n);
    long endTime = System.currentTimeMillis();
    System.out.println("Run-time = " + (endTime - startTime));
  }
}
```

Counting primes (Version 2)

```
// Counts how many primes exist up to N
numPrimes = 0
for i = 2 .. N {
  isPrime = true;
  limit = sqrt(i) + 1;
  for j = 2 to limit
    if j | i
      isPrime = false;
  if isPrime
    numPrimes++;
}
```

Formal analysis:

- Number of iterations is $\sqrt{2} + \sqrt{3} + \sqrt{4} + \dots + \sqrt{N} < N\sqrt{N}$
- We see that the run-time is at most $O(N\sqrt{N})$
- We can try to get a tighter bound.

Empirical analysis:

Input size (N)	10K	20K	30K	40K
Run-time (ms) with sqrt	16	47	63	94
Run-time (ms) without sqrt	780	3000	6800	12000

Significant improvement compared to the quadratic running-time.

Outline

- Introduction
 - Computational problems
 - Algorithms
- Search algorithms
 - Motivation
 - Sequential search
 - Binary search
 - Comparison
- Sorting algorithms
 - Selection sort
 - Insertion sort
- Running-time analysis
 - Definition / big O notation
 - Examples
- Induction proof technique
 - Definition
 - Example
- More algorithms
 - Euclid's GCD algorithm
 - Square root by binary search
 - Newton-Raphson method
 - MergeSort
- More fun to go

Induction: \forall predicates P , $(P(0) \wedge \forall k[P(k) \Rightarrow P(k+1)]) \Rightarrow \forall n P(n)$

Induction proofs in math

A predicate P is stated, typically on the natural numbers

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 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 it is also true for $k+1$

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

Induction proofs in algorithm analysis

To prove by induction that a given algorithm is correct, we ...

- Prove that at step 0 (or 1) the algorithm produces the correct output
- Prove that if the output is correct at step k , it follows that the output is also correct at step $k+1$.

Why induction works

Theorem: Someone tells you that a predicate P is true for all numbers $1 \dots n$.
If you prove that P is true by induction, then P must be true for all numbers $1 \dots n$.

Proof (by contradiction):

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

Suppose now that P is actually false for some numbers.

Therefore, there exists a smallest $k \leq n$ for which $P(k)$ is false.

Either $k = 1$, or $k > 1$.

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 induction works.

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

Theorem:

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

Proof: 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 an array of size $0 \leq k < n$

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 and thus less than n , so 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 length of this sorted array is less than k and thus less than n , so according to the inductive hypothesis the algorithm will find it.

```
// 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
```

Outline

- Introduction
 - Computational problems
 - Algorithms
 - Search algorithms
 - Motivation
 - Sequential search
 - Binary search
 - Comparison
 - Sorting algorithms
 - Selection sort
 - Insertion sort
 - Running-time analysis
 - Definition / big O notation
 - Examples
 - Induction proof technique
 - Definition
 - Example
 - More algorithms (*)
 - Euclid's GCD algorithm
 - Square root by binary search
 - Newton-Raphson method
 - MergeSort
 - More fun to go
- (*) This discussion is based on Noam Nisan, to whom I am grateful.

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)

	<u>rem</u>	<u>x</u>	<u>y</u>
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).

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.

Euclid's GCD Algorithm:

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

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
 - Motivation
 - Sequential search
 - Binary search
 - Comparison
- Sorting algorithms
 - Selection sort
 - Insertion sort
- Running-time analysis
 - Definition / big O notation
 - Examples
- Induction proof technique
 - Definition
 - Example
- More algorithms
 - Euclid's GCD algorithm
 - • Square root by binary search
 - Newton-Raphson method
 - MergeSort
- More fun to go

Square root by binary search

The square root problem:

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

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

```
// 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
}
```

Sample run of the binary search

```

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
}

```

Sample run: Computes sqrt(2) with precision 0.05

	<u>mid</u>	<u>mid*mid</u>	<u>low</u>	<u>high</u>
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 $mid \leq \sqrt{x}$ the code sets $low = mid$
and thus $low \leq \sqrt{x}$

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

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

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

At this point, by the lemma:
 $low \leq \sqrt{x} \leq high \leq low + \epsilon$.

Thus $r \leq \sqrt{x} \leq r + \epsilon$

Thus $|r - \sqrt{x}| \leq \epsilon$.

```

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 of finding square root by binary search

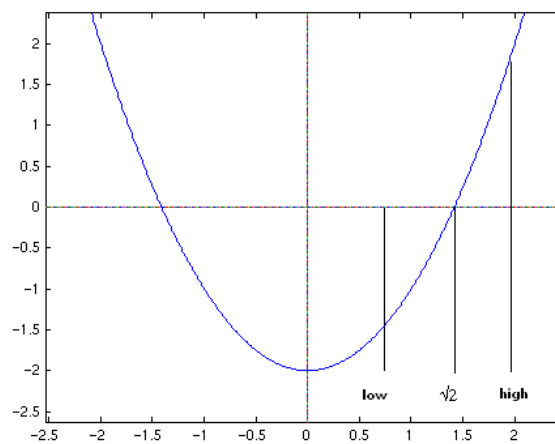
- In each iteration, the value of $(high-low)$ decreases by a factor of 2.
- At the beginning, $(high-low) = x$; at the end, $(high-low)$ goes below ϵ .
- How many times do you have to divide x by 2 before it goes below ϵ ?
- Answer: $\log_2(x/\epsilon)$
- The running-time is $O(\log_2 x + \log_2 \epsilon^{-1})$

```
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;  
}
```

Binary search can be used to find the roots of any *continuous* function f

Mean Value Theorem: if $f(low) < 0$ and $f(high) > 0$
then there is $low < x < high$ with $f(x) = 0$.

In our case, to find $\sqrt{2}$, we solved $f(x) = x^2 - 2 = 0$



Outline

- Introduction
 - Computational problems
 - Algorithms
- Search algorithms
 - Motivation
 - Sequential search
 - Binary search
 - Comparison
- Sorting algorithms
 - Selection sort
 - Insertion sort
- Running-time analysis
 - Definition / big O notation
 - Examples
- Induction proof technique
 - Definition
 - Example
- More algorithms
 - Euclid's GCD algorithm
 - Square root by binary search
 - Newton-Raphson method
 - MergeSort
- More fun to go

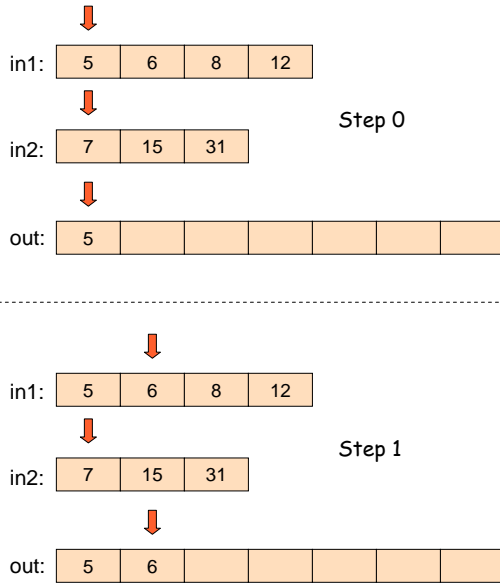


Looking back at some sorting algorithms

How long does it take to sort N elements?

- Selection sort: $O(N^2)$
- Insertion sort: $O(N^2)$
- It turns out that:
 - The running-time of all simple sorting algorithms is $O(N^2)$
 - The running-time of the best sorting algorithms is $O(N \log_2 N)$
- We will now see one such algorithm: MergeSort
 - Merging two sorted arrays: (5, 17, 19) and (3, 7, 52) become (3, 5, 7, 17, 19, 52)
 - In-place merging: (5, 17, 19, 3, 7, 52) becomes (3, 5, 7, 17, 19, 52)
 - MergeSort: (19, 52, 5, 7, 3, 17) becomes (3, 5, 7, 17, 19, 52)

Merging two sorted arrays



Input: sorted arrays a and b
Output: merged and sorted array c

Algorithm (informal):

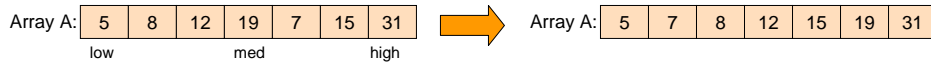
```
int[] out = new int[in1.length +
                  in2.length]
```

```
int i1 = i2 = i = 0
```

In each step:

```
if (in1[i1] < in2[i2])
    out[i++] = in1[i1++]
else
    out[i++] = in2[i2++]
```

In-place merging

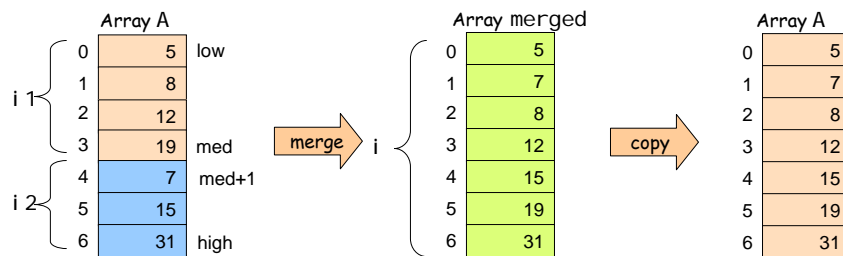


Merging two arrays (previous slide):

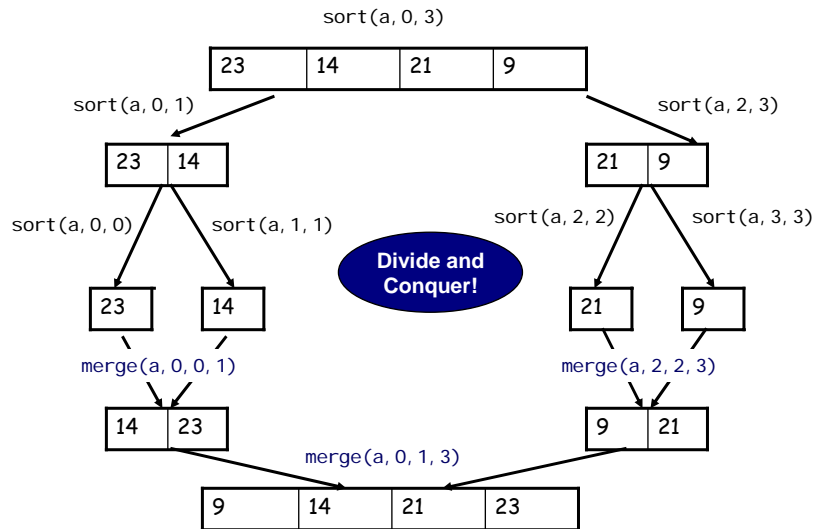
```
if (in1[i1] < in2[i2])
    out[i++] = in1[i1++]
else
    out[i++] = in2[i2++]
```

In-place merging:

```
i = low..high, i1 = low..med, i2 = (med+1)..high
if (A[i1] < A[i2])
    merged[i++] = A[i1++]
else
    merged[i++] = A[i2++]
// Copy merged onto A
```



Sample run



Outline

- Introduction
 - Computational problems
 - Algorithms
- Search algorithms
 - Motivation
 - Sequential search
 - Binary search
 - Comparison
- Sorting algorithms
 - Selection sort
 - Insertion sort
- Running-time analysis
 - Definition / big O notation
 - Examples
- Induction proof technique
 - Definition
 - Example
- More algorithms
 - Euclid's GCD algorithm
 - Square root by binary search
 - Newton-Raphson method
 - MergeSort
- More fun to go



Abu Abdullah
Muhammad ibn Musa
al-Khwarizmi