Introduction to Computer Science
Shimon Schocken
IDC Herzliya

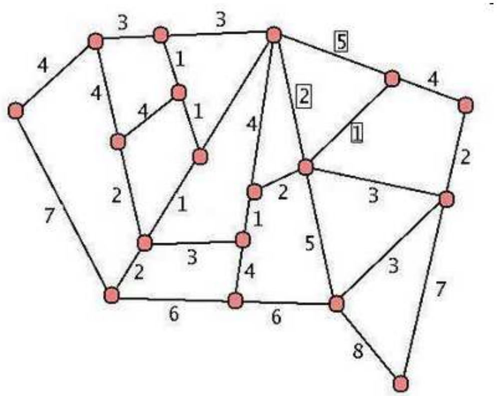
Lecture 8-1

Algorithms

Computational problems

<u>A computational problem</u> 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.



Outline

Introduction

- Computational problems
- Algorithms

Search algorithms

- Motivation
- Sequential search
- Binary search
- Comparison

Running-time analysis

- Performance monitoring
- Big O analysis

Algorithms

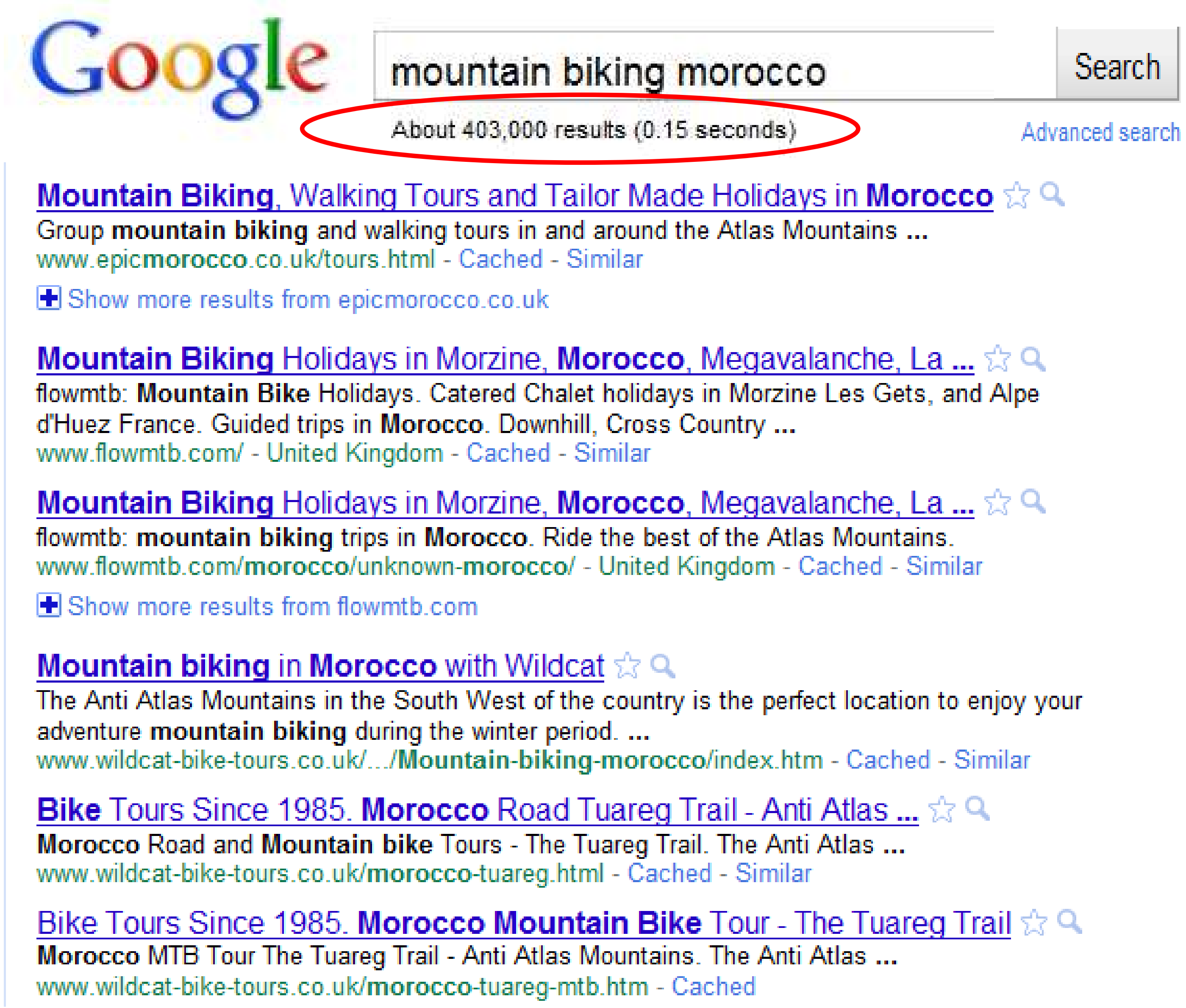
Algorithm: A specification how to solve a computational problem.

- We wish algorithms to be:
  - Correct: produce the correct output for each possible input
  - Efficient: use as little resources as possible (time, space)
- There are usually many different algorithms for each computational problem
- The algorithm's description must be such that one can write a program from it.

Example:

```
// Testing whether a number x is prime:
for j = 2 .. x-1
  if j|N
    return "x is composite"
return "x is prime"
```

Example: Search engines



Search engine -- behind the scene

The search engine (SE) index:

- A list of words; each word is associated with a list of URL's that mention it
- The lists are maintained by hard-working robots

Typical search scenario:

- User enters a keyword
- The SE searches the index
- The SE returns a list of URLs that mention this word; the list is sorted by PageRank

The search engine must be

- Reliable
- Efficient

<u>Opening the black box:</u>

- Searching algorithms
- Sorting algorithms.

keyword	URLs
mohican	11, 4, 5
more	2, 11
morgan	13, 100, 1 ,7
mormon	4, 83
morning	12, 4
morocco	1, 7, 4, 5
mortal	17
mortgage	81, 9
mountain	10, 3, 5, 4
nader	9
nalini	5, 11, 12, 95
name	17, 2, 8
namibia	5, 17
nancy	3, 51, 7, 9, 1
never	19
nike	55, 21
ninex	17, 3, 308
nitro	91, 7

Sequential search

Input: a value x and a list of N values

Output: if x is found, its location; else -1

Strategy: march through the list

What is the running-time of sequential search?

- On which input?
- We normally carry out worst-case analysis
- Worst-case running time is N steps.

keyword	URLs
mohican	11, 4, 5
more	2, 11
morgan	13, 100, 1 ,7
mormon	4, 83
morning	12, 4
morocco	1, 7, 4, 5
mortal	17
mortgage	81, 9
mountain	10, 3, 5, 4
nader	9
nalini	5, 11, 12, 95
name	17, 2, 8
namibia	5, 17
nancy	3, 51, 7, 9, 1
never	19
nike	55, 21
ninex	17, 3, 308
nitro	91, 7

Binary search

Input: a value *x* and a *sorted list* of N values

Output: if x is found, its location; else -1

Strategy: Divide and conquer

What is the running time of binary search?

- It's the number of times you can divide n by 2
- Worst-case running time is $\log_2 N$ steps.

keyword	URLs
mohican	11, 4, 5
more	2, 11
morgan	13, 100, 1 ,7
mormon	4, 83
morning	12, 4
morocco	1, 7, 4, 5
mortal	17
mortgage	81, 9
mountain	10, 3, 5, 4
nader	9
nalini	5, 11, 12, 95
name	17, 2, 8
namibia	5, 17
nancy	3, 51, 7, 9, 1
never	19
nike	55, 21
ninex	17, 3, 308
nitro	91, 7

Sequential search revisited

	0	1	1	3	4	5	6	7	8	9
Data:	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
```

<u>If the array is of size N, how many steps will it take to find an item?</u>

- In the best case? 1
- In the worst case? N
- On average? (*) $(1 + 2 + 3 + \ldots + N) / N = \frac{1}{2}(N + 1)$

(Asuming a uniform distribution)

Binary search revisited

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

```
// Find the index of x in a sorted array

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

Binary search: 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
```

Divide and Conquer!

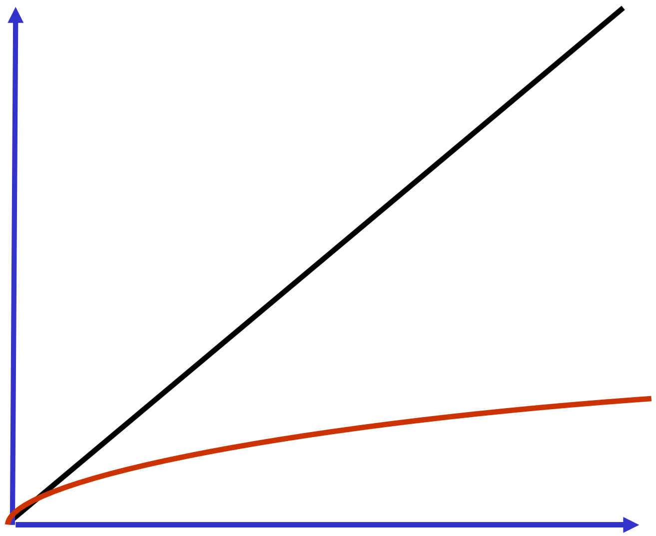
How many iterations in this loop?

- 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? $\log_2 N$

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

Why logarithmic running time is sweet

Input size: N	Run-time: Seq. N	Binary $\log_2 N$
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 is $\log_2 N$ 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
- When the size of the Internet doubles, each search requires one more step.

Not bad!

Outline

Introduction

- Computational problems
- Algorithms

Search algorithms

- Motivation
- Sequential search
- Binary search
- Comparison

Running-time analysis

- Performance monitoring
- Big O analysis

Empirical testing of Java's performance on 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 = 1000000000;
    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);
  }
}

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

	Timed operation	Run-time (in ms) on Shimon's old PC
Loop overhead:	....	1,047
Double division:	`d=1.0/d;`	16,140
Method call:	`obj.m();`	2,406
Object creation:	`obj = new SimpleObject();`	10,937

These performance figures vary greatly on different machines

Thus, although empirical testing is useful, it is quite useless from a theoretical point of view.

Counting program operations

- We can count the number of operations that the algorithm performs:
  - Arithmetic: `(low + high)/2`
  - Comparison: `if (x = a[med]) ...`
  - Assignment: `low = med + 1`
  - Branching: `while ( low <= high )`
  - Etc.

```
while (low <= high)
    med = (low + high) / 2
    if (x = a[med])
        return med
    if (x < a[med])
        high = med - 1
    else
        low = med + 1
```

- But these operations ...
  - Are not atomic
  - Are not low-level
  - Don't run in the same time
  - Run in different times on different hardware / software platforms.

- Thus counting operations is also quite useless from a theoretical point of view.

Running time analysis

The actual running time of a any given algorithm depends upon:

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

Let's make all these factors irrelevant

Formal run-time analysis:

Neutralize all the platform-specific details; Focus instead on one thing only:

Running-time of the algorithm as a function of the input size: t(N).

Running time analysis

We seek a function t(N) which will be invariant over hardware and software.

Example: print a multiplication table of size N by N

```
print "enter the table's size:"
read N
for i = 0 .. N-1
    for j = 0 .. N-1
        print i * j;
    println
```

- Let's assume that each operation takes 1 time unit
- Set up (print/read): 2 time units
- Inner loop: N * 1 + 1 time units
- Outer loop: N iterations
- Total run time: 2 + N * (N + 1)
- $t(N) = N^2 + N + 2$

<u>Running time analysis:</u>

- The running time t(N) is often a polynomial function in N, the input's size
- Instead of looking at t(N), we ignore all the constants and all the terms except for the highest degree of N
- Example: if $t(N) = N^2 + N + 2$ we say that the running time is "order of N^2"
- Indeed, for realistically large N's, the high order term dominates the running-time
- <u>Running time analysis</u>: a nice example of how to focus on the big picture.