Algorithm Analysis (IV)

Dr. Antonio L. Bajuelos
FIU School of Computing & Information Sciences

COP-3530 - Data Structures

Module #1: Algorithm Analysis (part IV)

Outline:

- **Logarithms in the running time.**
- **Examples:**
  - **Binary Search.**
  - **Greatest Common Divisor (GCD).**
- **Summary of Algorithm Analysis.**

2

Logarithms in the Running Time

- The "**logarithm**" – some on most confusing aspect of analysis of algorithms.
- Logarithm General Rule:

> **An algorithm is O(log*N*) if it takes constant time, O(1), to cut the problem size by a fraction (which is usually ½)**

- Only special kinds of problems can be **O(log*N*)**.
- If the input is a list of *N* numbers then an algorithm must take (*N*) merely to read the input in. Thus, when we talk about *O*(log*N*) algorithms we usually presume that the input is pre-read.

3

O(log*N*) Algorithm: Binary Search

- The **Classic Search Problem:**
  - Given an integer X and integers A_0, A_1, . . . , A_{N-1}, **which are pre-sorted** and already in memory, find the index i such that $A_i = X$, or return $i = -1$ if X is not in the input.
- **Obvious solution:**
  - O(*N*) algorithm - Scan the list from left to right and find *i*.
- This algorithm **not take advantage of the fact that the list is pre-sorted**.

4

O(log*N*) Algorithm: Binary Search

- **The Classic Search Problem:**
  - Given an integer X and integers $A_0, A_1, \ldots, A_{N-1}$, **which are presorted** and already in memory, find i such that $A_i = X$, or return $i = -1$ if X is not in the input.
- **Better strategy (Binary Search):**
  - Compare X with middle item A[mid],
    - Go to **left half** **if** X **< A[mid]**
    - Go to **right half** **if** X **> A[mid]**
    - Repeat

5

O(log *N*) Algorithm: Binary Search

X = i

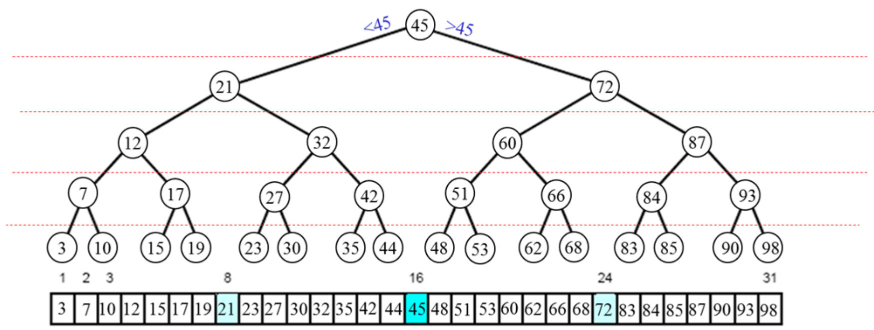
a	b	c	d	e	f	g	h	i	j

a	b	c	d	e	f	g	h	i	j

a	b	c	d	e	f	g	h	i	j

6

O(logN) Algorithm: Binary Search



O(logN) Algorithm: Binary Search

```
/**
 * Performs the standard binary search.
 * @return index where item is found, or -1 if not found.
 */
public static <AnyType extends Comparable<? super AnyType>>
int binarySearch( AnyType [ ] a, AnyType x )
{
    int low = 0, high = a.length - 1;
        while( low <= high )
            {
                int mid = ( low + high ) / 2;
                if( a[ mid ].compareTo( x ) < 0 )         // x > a[mid]
                    low = mid + 1;
                else if( a[ mid ].compareTo( x ) > 0 )  // x < a[mid]
                    high = mid - 1;
                else
                    return mid; // Found
            }
    return NOT_FOUND; // NOT_FOUND is defined as -1
}
```

- **Note:** The code reflects Java’s convention that arrays begin with index 0.

Algorithm Analysis of Binary Search

- Worst case?
  - When <u>*X* is not found</u>.
- How many iterations are executed before low > high?
  - After first iteration: $N/2$ items remaining
  - After 2nd iteration: $(N/2)/2 = N/4$ remaining
  - After k-th iteration?
    - $N/2^K$ remaining
  - Worst case?
    - Last iteration occurs when $N/2^K \geq 1$ and $N/2^{K+1} < 1$ item remaining
    - $2^k \leq N$ and $2^{k+1} > N$ [take log of both sides]
  - Number of iterations is $k \leq \log N$ and $k > \log N - 1$
  - **Binary Search is O(log*N*)**

9

Binary Search – Some Notes

- **Binary search - our first data structure implementation.**
- In Binary Search the "contains" operation is ***O*(log*N*)** time in **worst-case**.
- Binary search is ***O*(1)** in the best-case (item is in the middle).
- The find-in-sorted-array problem is **Ω(log*N*)** in worst-case (no algorithm can do better).
- All other operations (in particular insert-in-sorted-array) require ***O*(*N*)** time.
- The input dates would then need to be sorted once, but afterward accesses would be fast.

10

Euclid's Algorithm for GCD

- **Definition:** The greatest common divisor (**gcd**) of two integers is the largest integer that divides both. Examples: gcd(50,15) = 5.
- The problem:
  - ***Find the greatest common divisor of two positive integers, M and N.***
- A good solution: **The Euclid's Algorithm***
- The Euclid's Algorithm is based on the following two observations:
  - gcd(M,N) = gcd(N, M mod N), if $N \neq 0$
  - gcd(N,0) = N

*The original version of the Euclid's Algorithm appears as the solution to the Proposition VII.2 in the *Elements*:

- *Given two numbers not prime to one another, to find their greatest common measure.*

Euclid's Algorithm for GCD

- **Properties:**
  - gcd(M,N) = gcd(N, M mod N), if $N \neq 0$
  - gcd(N,0) = N

Example:

gcd(3084,1424) = ?

3084 = 1424*2 + 236	gcd(3084,1424)= gcd(1424,236)
1424 = 236*6 + 8	gcd(1424,236) = gcd(236,8)
236 = 8*29 + 4	gcd(236,8) = gcd(8,4)
8 = 4*2 + 0	gcd(8,4) = gcd(4,0)
	then **gcd(3084,1424) = 4**

Formal description of the Euclid's Algorithm

- **Input:** Two positive integers, M and N.
- **Output:** The **gcd** of M and N.
- **Internal computation:**
  - If $M < N$, exchange M and N.
  - Divide M by N and get the remainder, r.

    If $r = 0$, report N as the **gcd** of M and N.
  - Replace M by N and replace N by r. Return to the previous step.

Java version of the Euclid's Algorithm

```
public static long gcd( long m, long n )
{
    while( n != 0 )
    {
        long rem = m % n;
        m = n;
        n = rem;
    }
    return m;
}
```

- **Analysis:**
  - The running time of the algorithm depends on determining how long the sequence of remainders is.
  - It is easy to prove (homework!) that after two iterations of the loop while, the remainder is at most half of its original value.
  - Then the number of iterations is at most $2\log N = O(\log N)$ and establish the running time.
  - **Euclid's Algorithm is O(log*N*)**
- **Interesting fact:**
  - The constant can be improved to approximately 1.44logN, in the worst case (the case when M and N are consecutive Fibonacci numbers).

Why Recursion?

- If recursion is less efficient (in some cases), why use it?
  - It leads to elegant solutions and the code can be clearer and simpler.
  - Some problems with ADTs require recursion. Examples:
    - Tree traversals
    - Graph traversals
    - Search problems
  - In some cases, an algorithm with a recursive solution has a lesser computational complexity. Example: Insertion Sort vs. Merge Sort.

15

Summary of the Algorithm Analysis

- We can analyze the problem or the algorithm - usually **algorithm**
- We can consider Time or Space - usually time (**running time**)
- We can analyze the Best, Worst, or Average-case - usually **worst-case**.
- We can analyze the Upper, Lower, or Tight-bound - usually **upper or tight-bound.**
- Asymptotic complexity (**Big-Oh**) focuses on <u>behavior for large *N*</u> and is independent of the computer, programming language, coding, etc.

16

Summary of the Algorithm Analysis

- Simple programs usually have simple analyses, but this is not always the case. Example: sorting algorithm Shellsort (Chapter 7).
- An interesting kind of analysis is the lower-bound analysis. We will see an example of this in Chapter 7, where it is proved that any algorithm that sorts by using only comparisons requires $\Omega(N \log N)$ comparisons.
- Some of the algorithms described in this topic have real-life applications. The *gcd* algorithm is used in cryptography.
- Interesting algorithm is the algorithm for efficient exponentiation. (see Chapter 2, pp 47-49).

17

18