Murdoch
UNIVERSITY

Topic 11

More On Algorithms

ICT167 Principles of
Computer Science

Murdoch UNIVERSITY

OBJECTIVES

- Be able to give a rough **estimate of the running time** (in basic steps) of simple algorithms
- Explain the concept of **recursion**
- Give **recursive algorithms** for simple problems
- **Trace** the operation of recursive calls
- Be able to **implement** simple recursive algorithms in Java

Murdoch UNIVERSITY

OBJECTIVES

- Be able to implement a **binary search** of a sorted array using recursion

Reading:

Savitch Chapter 11 plus extra material

Example Algorithm

- Consider the common problem of finding (searching) a target value in a *sorted* array and returning some index at which it appears (or an indication if it does not appear at all)
- The next slide provides pseudo-code for a straight-forward solution for the case with an array of integers
  - The algorithm will return the index at which the target value first appears or –1 if the target value does not appear in the array

Pseudo-code

```
Given an array a of integers and a target
integer value
let len = length(a)
i = 0
while ((i < len) and (a[i] < target))
   i = i + 1
endwhile
answer = -1
if (i < len) then
   if (a[i] == target) then
      answer = i
   endif
endif
```

Murdoch UNIVERSITY

Time Complexity of Algorithms

- When designing software and choosing between several ideas for algorithms it is often useful to get a rough idea of how long the algorithm will take to run
- Formal measures of this are called measures of time complexity of an algorithm
- For example with our search algorithm we can say:

Time Complexity of Algorithms

- To search in 1000 items it might take about 500 iterations of the loop on average, or at worst 1000 iterations
- If we knew that it took 1 second to search through 1000 items then we might guess it would take about 1000 seconds (about 17 minutes) to search through 1 million items
- The time taken is roughly proportional to the size of the array to search
- And we might be able to say that some other algorithm for doing the same job was significantly slower or quicker

Time Complexity of Algorithms

- Measuring time complexity, estimating it and inventing quick algorithms is a big area of computer science research
  - We look at time complexity again later in this topic
- Note that there are other reasons to choose between one algorithm and another in specific circumstances
  - For example, space complexity measures of how much memory an algorithm needs

Recursion

- One way of inventing quick algorithms for some problems is to use a **recursive** approach
- *"An object is recursive if it partially consists of or is defined in terms of itself."* - N. Wirth
- An **algorithm** is a step-by-step set of rules to solve a problem; it must eventually terminate with a solution

Recursion

- A **recursive algorithm** uses itself to solve one or more subcases
  - That is, in problem-solving using recursion, a solution is expressed in terms of itself
- Recursive methods implement recursive algorithms
- A recursive method is one whose definition includes a call to itself

Recursion as a Problem Solving Tool

- Solution to task T:
  - Solve task T1, which is identical in nature to task T, but smaller than T
- Example task:
  - Search a dictionary for a word

A Recursion Algorithm

```
If it is a one page dictionary then scan the
page for the word
else
   open dictionary near the middle
   determine which half contains the word
   if word is in first half then
      search 1st half of dictionary for word
   else
      search 2nd half of dictionary for word
   end if-else
end if-else
end algorithm
```

Murdoch UNIVERSITY

Recursive Definitions

- A recursive definition contains
  - A base part which contains the terminating condition to stop the recursion, and
  - A recursive part, where each successive call to itself must be a "smaller version of itself" so that a base case is eventually reached

Example 1

- Definition of an integer constant (eg: 571) (decimal notation) is:
  - Any decimal digit, or
  - Any decimal digit followed by an integer constant
- Base: Any decimal digit (0 through 9)
- Recursive: Any decimal digit followed by an integer constant
- Recursive part reduces to the base part with repeated applications

Murdoch UNIVERSITY

Example 2

- The Fibonacci numbers:

  1, 1, 2, 3, 5, 8, 13, 21, …
- The first number is 1
- The second number is 1
- Each of the other numbers is the sum of preceding two numbers

Recursive Definition: Example 2

- fib(1): 1 // base part
- fib(2): 1
- fib(n): fib(n - 1) + fib(n - 2)
  // recursive part for n > 2
- Eg:
- fib(3) = fib(2) + fib(1)

Recursive Methods

- Methods designed to solve problems by calling themselves
- Characteristics of a recursive solution:
  - Calls a method to solve a smaller problem of the same type
  - Size of problem diminishes in successive calls
  - A base case is solvable directly
  - That is, a recursive method **must** have a terminating condition – the recursive definition on the previous slide demonstrates this

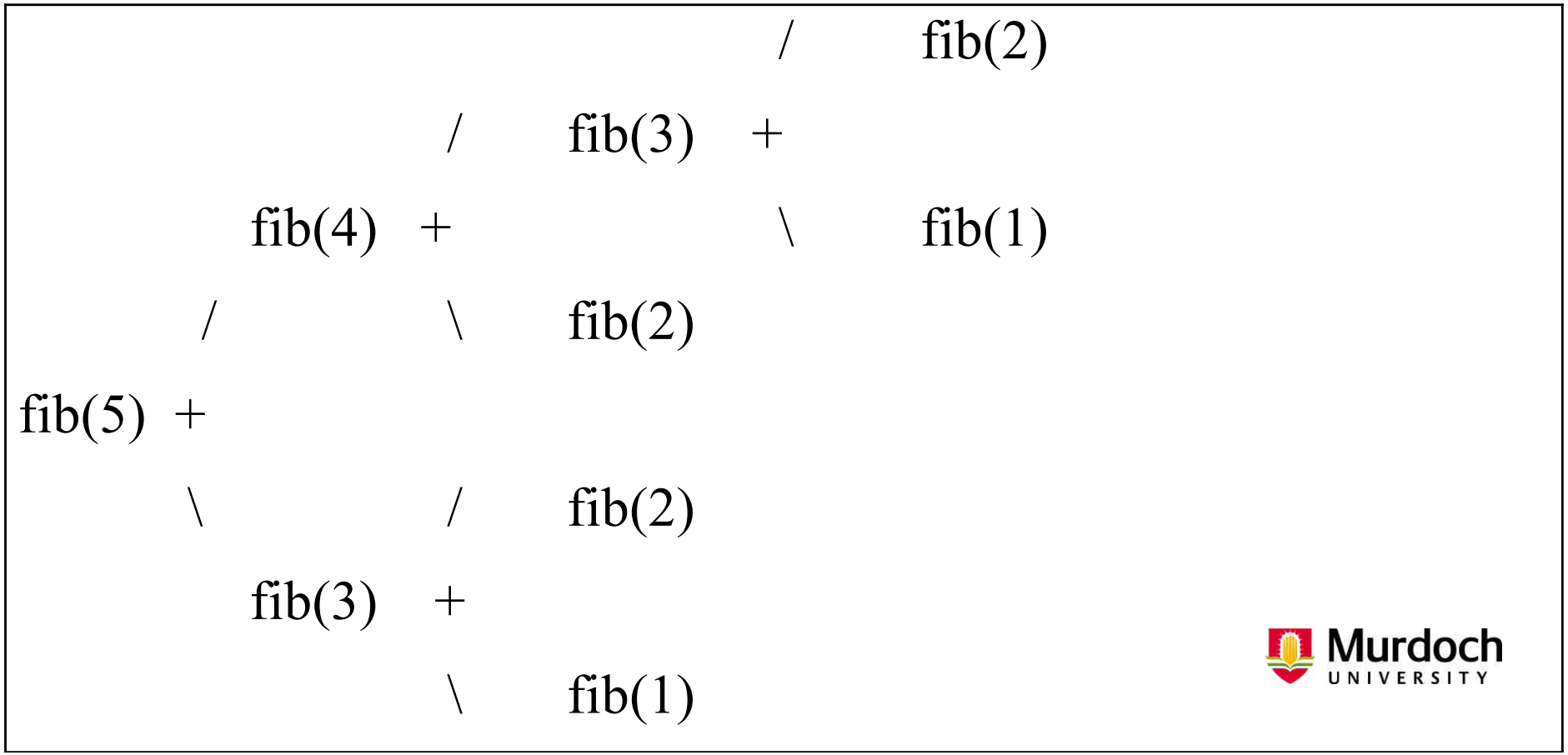
Recursive Fibonacci Method

```
static int fib(int n)
//pre-condition: n >= 0
{
   if (n <= 2)    // base case
      return 1;
   else           // recursive step
      return fib(n - 1) + fib(n - 2);
} // end fib
// a call to method fib
int x = fib(5);
// x will have the 5th Fibonacci number
```

Murdoch UNIVERSITY

Recursive Fibonacci Method

- Invocations of method fib during calculation of the 5th Fibonacci number



Murdoch UNIVERSITY

Another Example

- A recursive function for summing array elements
- Task: Sum the first **n** elements of array **A**
- sum (A, n) is:
- A[0], if n = 1 // base case
- A[n-1] + sum (A, n-1), if n > 1 // recursive step

Murdoch UNIVERSITY

Recursive Sum Method

```
static int sum(int[] A, int n)
//pre-condition: A.length >= n
{
   if (n == 1)
      return  A[0];    //base case
   return A[n-1] + sum(A, n-1);
}
```

Recursive Sum Method

Let A =

3	2	7	5	1	…
[0]	[1]	[2]	[3]	[4]	[5]

and n = 4

sum	3	2	7	5	1	…

= sum	3	2	7	+ 5

= sum	3	2	+ 7	+ 5

= sum	3	+ 2	+ 7	+ 5

= ((3 + 2) + 7) + 5

= (5 + 7) + 5

= 12 + 5

= 17

Murdoch UNIVERSITY

RecursiveSumArray.java

```
// RecursiveSumArray.java
// Sums the elements of an array recursively
// Written by P S Dhillon
public class RecursiveSumArray {
  public static void main( String[] args) {
    int[] anArray =
            {98,76,65,105,45,1,199,15,88,100};
  // determine sum of elements of the array
  int arraySum;
  arraySum = Sum(anArray, anArray.length);
```

Murdoch UNIVERSITY

RecursiveSumArray.java

```
System.out.println("The numbers are:");
for(int i = 0; i < anArray.length; i++)
   System.out.println( anArray[i]);
   System.out.println("The sum of array
                       values is: " + arraySum);
   System.out.println("End of program.");
} // end main
```

RecursiveSumArray.java

```
    static int  Sum(int[] A, int n)
    //pre-condition: A.length >= n
    {
       if (n == 1)
          return  A[0];      //base case
       return  A[n-1] + Sum(A, n-1);
    } // end Sum
}//end of class RecursiveSumArray
```

Murdoch UNIVERSITY

Designing A Recursive Solution

- A common strategy is:
- Given a problem of size **n**, split the problem into two sub-problems
  - A problem of size 1 which is directly solvable //the base case
  - A problem of size **n - 1** that involves recursion

Designing A Recursive Solution

- Example:
- A method to multiply two integer numbers **m** and **n**
  - Assume we know our addition table but not the multiplication table!

```
// m * n by repeated addition
Multiply(m, n):
m, if n = 1    // base case
               // recursive step
m + Multiply(m, n-1), if n > 1
```

Murdoch UNIVERSITY

Designing A Recursive Solution

```
//  Recursive multiply method
//  Performs multiplication using the + operator
static int Multiply(int m, int n)
// PRE: Assigned(m) && Assigned (n) && n > 0
// POST: returns  m * n
{
   if (n == 1)
      return  m;  // base case
   else           // recursive step
      return  m + Multiply (m, n - 1);
}
```

Murdoch UNIVERSITY

Designing A Recursive Solution

Example of a call to the previous method:

```
Scanner input = new Scanner(System.in);
System.out.print("Enter an integer: ");
int x = input.nextInt();
System.out.print("Another integer: ");
int y = input.nextInt();
System.out.println("\nThe product of " +
   x+" and "+y+" is: " + Multiply(x, y));
// Alternatively,
int result = Multiply(x,y);
```

Murdoch UNIVERSITY

Exercise for Topic 11

- Give a recursive Java method for writing out any given String in reverse order

Recursion: Pros and Cons

- A powerful problem solving tool - elegant and concise
- Not necessarily more efficient than non-recursive (looping = iterative) solution
- Recursive routines can be slower and require more memory space due to overheads associated with function calls
- Can be difficult to debug and may result in infinite recursion

Recursion: Pros and Cons

- Infinite recursion is worse than infinite loop
  - It makes the computer “hang up” by using up all available memory (*stack overflow*)
- Note that there are general techniques for getting rid of recursion from an algorithm and making an iterative version (but the idea might have been recursive originally and it might be easier to understand the recursive version)

To Recurse or Not To Recurse? That is the Question

- Choose recursion when
  - The problem is stated recursively and the recursive solution appears less complex
  - That is, when it makes the code easier to understand and when efficiency is not important
- Choose a non-recursive algorithm when
  - Both versions appear equally complex
- Methods re-written without recursion typically have loops, so they are called **iterative** methods

To Recurse or Not To Recurse? That is the Question

- Iterative methods generally run faster and use less memory space than recursive methods
- If the use of a table is an option
  - Use **table lookup** (see next slide)

Table Lookup

- Replaces a sequence of instructions with a simple array lookup
- Out-performs both recursive and iterative algorithms

```
public static int Tfib(int n)
//  PRE: (n >= 0) && (n < 8)
//  POST: value returned is nth Fibonacci number
{
   int[] fibTable = {1,1,2,3,5,8,13,21};
   return fibTable[n];
}
```

Murdoch UNIVERSITY

Binary Search

- Recall the common problem of finding a target value in a **sorted** array and returning some index at which it appears (or an indication if it does not appear at all)
- Here is another (recursive) idea for a solution:
  - Start in the middle and (if the target value is not there) search either the first half or the second half depending on where the target would be

Binary Search

- **Here is pseudocode:**
- *given array* **a** *of integers and target integer value*
- *output binsearch(a, 0, length(a)-1, target)*
-
- ***binsearch(int array a, int first, int last, int target)***
- *if (first>last) return –1*
- *mid= (first+last)/2 (integer division)*
- *if (a[mid]==target) return mid*
- *if (a[mid]>target)*
- * return binsearch(a, first, mid-1, target)*
- *else*
- * return binsearch(a, mid+1, last, target)*
-
- The idea of binsearch is to find an index in the range first to last inclusive such that the target value appears there in the array. Here is one possible Java implementation ...

Murdoch UNIVERSITY

Binary Search

- **Here is pseudocode:**

```
Given array a of integers and target value
output binsearch(a, 0, length(a)-1, target)
binsearch(int array a, int first, int last,
                                        int
   target)
   if (first>last) return -1
   mid = (first+last)/2   // integer division
   if (a[mid] == target) return mid
```

Murdoch UNIVERSITY

Binary Search

```
if (a[mid] > target)
   return binsearch(a,first,mid-
1,target)
else
   return
binsearch(a,mid+1,last,target)
```

Binary Search

- The idea of `binsearch` is to find an index in the range `first` to `last` inclusive such that the `target` value appears there in the array
- Here is one possible Java implementation ...

Binary Search Class

```
/**
Class for searching an already sorted array of
integers.
To search the sorted and completely filled array
b, use the following:
 ArraySearcher bSearcher = new ArraySearcher(b);
 int index = bSearcher.find(target);
where index will be given an index of where
target is located
otherwise index will be set to -1 if target is
not in the array
*/
```

Murdoch UNIVERSITY

Binary Search Class

```
public class ArraySearcher {
   private int[] a;
   // constructor
   public ArraySearcher(int[] theArray)
   /** Precondition: theArray is full and is sorted
                     from lowest to highest */
   {
      a = theArray;
      // a is now another name for theArray
   } // end constructor ArraySearch
```

Binary Search Class

```
/** If target is in the array, returns the index of
    an occurrence of target.
    Returns -1 if target is not in array*/
public int find(int target)
{
   int len = a.length - 1;
   return binarySearch(target,0,len);
}
```

Murdoch UNIVERSITY

Binary Search Class

```
/** Uses binary search to search for target in
    a[first] through a[last] inclusive
    Returns the index of target if target is found.
    Returns -1 if target is not found. */
private int binarySearch(int target, int
                                  first, int last)
{
   int result = -1;
   int mid;
   if (first > last)
      result = -1;
   else {
```

Murdoch UNIVERSITY

Binary Search Class

```
            mid = (first + last) / 2;
            if (target == a[mid])
               result = mid;
            else if (target < a[mid])
               result = binarySearch(target,first,
                                     mid-1);
            else   // (target > a[mid])
               result = binarySearch(target,mid+1,
                                     last);
         }
         return result;
      } // end binarySearch
} // end class ArraySearcher
```

Murdoch UNIVERSITY

Binary Search Demo

```
import java.util.*;
public class ArraySearcherDemo {
   public static void main(String[] args) {
      int [] a = new int[10];
      System.out.println("Enter 10 integers in
                          increasing order.");
      System.out.println("One per line.");
      Scanner keyboard=new Scanner(System.in);
      for (int i = 0; i < 10; i++)
         a[i] = keyboard.nextInt();
      System.out.println();
```

Murdoch UNIVERSITY

Binary Search Demo

```
System.out.print("a["+i+"]="+a[i]+" ");
System.out.println();
System.out.println();
ArraySearcher finder = new
                          ArraySearcher(a);
String ans;
do {
   System.out.println("Enter a value to
                              search for:");
   int target = keyboard.nextInt();
   int result = finder.find(target);
```

Binary Search Demo

```
            if (result < 0)
               System.out.println(target + " is
                                  not in the array.");
            else
               System.out.println(target + " is at
                                  index " + result);
            System.out.println("Again?(yes/no)");
            ans = keyboard.next();
         }while (ans.equalsIgnoreCase("yes");
         System.out.println("May you find what
                            you're searching for.\n");
      } // end main
   } // end class ArraySearcherDemo
```

Murdoch UNIVERSITY

How Long Does It Take?

- It is a bit harder to analyze the time complexity of binary search (than the simple iterative version given earlier in the topic)
- Eg: to search through 1000 items we (in a couple of operations) break the problem down into a search through 500 items, then 250 items, then 125 items, then 63, then 32, then 16, then 8, then 4, then 2, then we must have found our target (or returned –1)
  - There are about 10 such steps

How Long Does It Take?

- In general to search through N items, we take $\log_2(N)$
  - Recall 1000 is about 2 to the tenth
- To search though 1 million items only takes twice as long!!
- The individual steps may take a little longer (i.e. consist of several basic operations) but, for large N, this is outweighed

How Long Does It Take?

- Eg: made up figures ...

search times	simple linear	binary
1 item	.001 sec	.01 sec
10 items	.01 sec	.03 sec
1000 items	1 sec	0.1 sec
1 million items	17 minutes	0.2 sec

- So binary search is a much better searching algorithm

Murdoch UNIVERSITY

Algorithm Efficiency

- We have seen that choosing the right algorithm for the job can sometimes make enormous differences to the efficiency of programs
- Many important problems and possible algorithmic solutions have been studied for complexity and other efficiency issues
- This is a big area of computer science research. This is important for several different types of situations

Algorithm Efficiency

- Eg: getting a *really big job done faster*
  - Allocate school students to university places in less than one hour instead of several days, or
  - Timetabling, or
  - Many scientific and engineering applications, or
  - Internet searches, or
  - Searching and sorting in large databases, etc.

Murdoch UNIVERSITY

Algorithm Efficiency

- Eg: getting a *reasonably large job done very fast*
  - Graphics in virtual reality, or
  - Games, or
  - Finding words in files or emails on a PC, etc.

More On Efficiency

- Note that you will sometimes see the big-oh notation to express the order of magnitude measure on how long an algorithm takes to solve a problem
- Eg: saying that our simple linear search algorithm is O(N) means that its running time is proportional to N where N is the size of the data

More On Efficiency

- You will see $O(\log_2(N))$ for binary search and $O(N^2)$ for some sorting algorithms, etc.
- These give the implementer a rough idea of which algorithms are best for the problem
- You may also see reports that certain **problems** are $O(N)$ or $O(N^2)$ or $O(N^3)$ or $O(\log_2 N)$, etc.

More On Efficiency

- This means that it has been mathematically proved that this is the best time complexity possible for any algorithm to solve that problem
  - It is impossible to find a better algorithm
- Eg: to sort N items takes O(N log N) steps on average
- No algorithm (even one not yet invented) can do better than that on average

More On Efficiency

- Insertion sort takes $O(N^2)$ steps on average, quicksort takes O(N log N) steps on average. Quicksort is best possible (in a certain sense)...

sorting	insertion	quicksort
1 item	.001 sec	.010 sec
1000 items	17 minutes	100 sec
1 million items	32 years	2 days

Grouping Algorithms by Efficiency

- Most algorithms execute in **polynomial time**, expressed as $O(N^a)$, constant $a > 0$
- Eg: $O(N)$ is linear time
- $O(N^2)$ is quadratic time
- $O(N^3)$ is cubic time
- Algorithms whose running time is independent of problem size are known as **constant time** algorithms
- Big-O notation: $O(1)$

Murdoch UNIVERSITY

Grouping Algorithms by Efficiency

- Algorithms requiring time proportional to a^N (where *a* is a constant) are known as **exponential algorithms**
- Execution times for exponential algorithms increase extremely fast with problem size
- Exponential algorithms are not suitable for any values of *N* except very small

Murdoch UNIVERSITY

Growth Rates for Selected Algorithms

- Average running times of some searching and sorting algorithms

Algorithm	**Efficiency -average case**
Sequential search	$O(N)$
Binary search	$O(\log_2 N)$
Bubble sort	$O(N^2)$
Selection sort	$O(N^2)$
Quick sort	$O(N \log_2 N)$

Calculating Running Time in Big-O Notation

- An algorithm without loops or recursion requires O(1) time
- An algorithm with *N* iterations requires O(*N*) time
- Eg:

```
for i = 1 to N
   statements without any more looping
endfor
```

Murdoch UNIVERSITY

Calculating Running Time in Big-O Notation

- An algorithm with one loop nested inside another has quadratic efficiency O(N^2)
- Eg:

```
for i = 1 to N
   for j = 1 to N
      statements without more looping
   endfor
endfor
```

Murdoch UNIVERSITY

END OF TOPIC 11

Murdoch UNIVERSITY