Riportare soluzione probl. a una solur. di un problema più piccolo di dim.

Recursion

1

Ci deve essere un punto talmente piccolo da avere solur. banale

The Recursion Pattern

- Recursion: when a method calls itself
- Classic example – the factorial function:
  n! = 1· 2· 3· ··· · (n-1)· n
- Recursive definition: $f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & \text{else} \end{cases}$

```
public static int factorial(int n) throws IllegalArgumentException {
  if (n < 0)
    throw new IllegalArgumentException( );   // argument must be nonnegative
  else if (n == 0)
    return 1;                                // base case
  else
    return n * factorial(n−1);               // recursive case
}
```

2

Valutare complessità significa valutare al crescere di n quante volte devo attivare una procedura, che costa

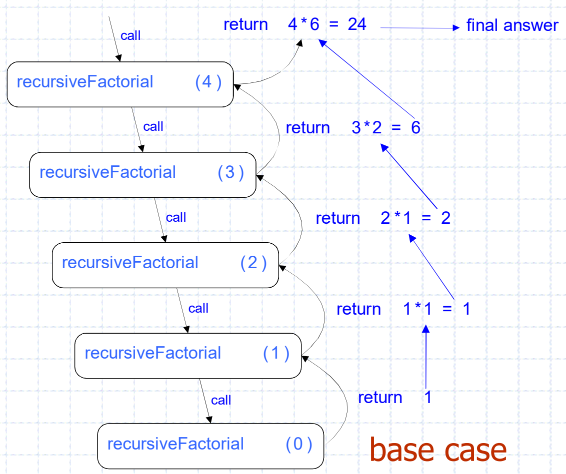
Content of a Recursive Method

- Base case(s)
  - Values of the input variables for which we perform no recursive calls are called base cases (there should be at least one base case).
  - Every possible chain of recursive calls must eventually reach a base case.
- Recursive calls
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

3

Tutti segnati in stack.

Recursion Example



4

Binary Search

Search for an integer in an ordered list

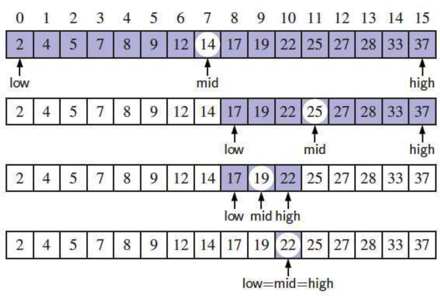
```
/**
 * Returns true if the target value is found in the indicated portion of the data array.
 * This search only considers the array portion from data[low] to data[high] inclusive.
 */
public static boolean binarySearch(int[ ] data, int target, int low, int high) {
  if (low > high)
    return false;                                        // interval empty; no match
  else {
    int mid = (low + high) / 2;
    if (target == data[mid])
      return true;                                       // found a match
    else if (target < data[mid])
      return binarySearch(data, target, low, mid − 1);   // recur left of the middle
    else
      return binarySearch(data, target, mid + 1, high);  // recur right of the middle
  }
}
```

base cases

Visualizing Binary Search

- We consider three cases:
  - If the target equals data[mid], then we have found the target.
  - If target < data[mid], then we recur on the first half of the sequence.
  - If target > data[mid], then we recur on the second half of the sequence.

We are looking for number 23



Analyzing Binary Search

- Runs in O(log n) time.
  - The remaining portion of the list is of size high – low + 1
  - After one comparison, this becomes one of the following:

$\sim n$

$$(\text{mid}-1)-\text{low}+1=\left\lfloor\frac{\text{low}+\text{high}}{2}\right\rfloor-\text{low}\le\frac{\text{high}-\text{low}+1}{2}$$

$$\text{high}-(\text{mid}+1)+1=\text{high}-\left\lfloor\frac{\text{low}+\text{high}}{2}\right\rfloor\le\frac{\text{high}-\text{low}+1}{2}.$$

  - Thus, each recursive call divides the search region in half; hence, there can be at most log n levels

7

Design of recursive algorithms

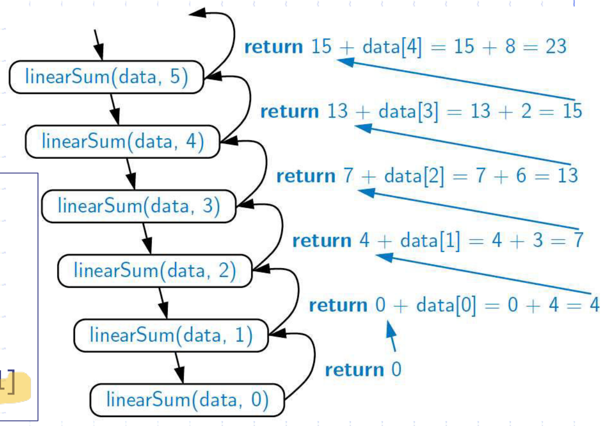
Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.

- Linear recursion: recurs at most once
  - Performs a single recursive call. This step may have a test that decides which of several possible recursive calls to make, but it should ultimately make just one of these calls
- Binary or Double recursion: each call performs at most other two calls
- Multiple recursion: each call performs more than two calls

8

Example of Linear Recursion

linearSum(data, 5)
data = [4, 3, 6, 2, 8]




```
Algorithm linearSum(A, n):

if n = 0 then
  return 0
else
  return
linearSum(A, n - 1) + A[n - 1]
```

Somma degli ultimi n-1 elementi + ultimo elemento

9

Defining Arguments for Recursion

- In creating recursive methods, it is important to define the methods in ways that facilitate recursion.
- This sometimes requires we define additional parameters that are passed to the method.
- For example, we defined the array reversal method as reverseArray(A, i, j), not reverseArray(A)

```
/** Reverses the contents of subarray data[low] through data[high] inclusive. */
public static void reverseArray(int[ ] data, int low, int high) {
  if (low < high) {                          // if at least two elements in subarray
    int temp = data[low];                    // swap data[low] and data[high]
    data[low] = data[high];
    data[high] = temp;
    reverseArray(data, low + 1, high − 1);   // recur on the rest
  }
}
```

10

ultima operaz: chiamata ricorsiva

Tail Recursion

- Tail recursion occurs when a linearly recursive method makes its recursive call as its last step (i.e. the array reversal).
- Such methods can be easily converted to non-recursive methods (which saves on some resources).
- Example:

```
Algorithm IterativeReverseArray(A, i, j ):
    Input: An array A and nonnegative integer indices i and j
    Output: The reversal of the elements in A starting at
  index i and ending at j
    while i <  j do
      Swap A[i ] and A[ j ]
      i  = i + 1
      j  = j - 1
    return
```


Tail Recursion (iterative version of binary Search)

```
public static boolean binarySearchIterative(int[] data, int target) {

  int low = 0;
  int high = data.length - 1;
  while (low <= high) {
    int mid = (low + high) / 2;
    if (target == data[mid])          // found a match
      return true;
    else if (target < data[mid])
      high = mid - 1;                 // only consider values left of mid
    else
      low = mid + 1;                  // only consider values right of
  mid
  }
  return false;                       // loop ended without success
}
```

Computing Powers

- The power function, $p(x,n)=x^n$, can be defined recursively:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \text{else} \end{cases}$$

- This leads to an power function that runs in O(n) time (we make n recursive calls)

Recursive Squaring

- A more efficient linearly recursive algorithm by using repeated squaring:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x,(n-1)/2)^2 & \text{if } n > 0 \text{ is odd} \\ p(x,n/2)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

- For example,

$2^4 = 2^{(4/2)2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16$

$2^5 = 2^{1+(4/2)2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32$

$2^6 = 2^{(6/2)2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64$

$2^7 = 2^{1+(6/2)2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128$

Recursive Algorithm

Algorithm Power(x, n):
Input: A number x and integer n = 0
Output: The value x^n
if n = 0 **then**
return 1
if n is odd **then**
y = Power(x, (n - 1)/ 2)
return x · y ·y
else
y = Power(x, n/ 2)
return y · y

Power(2, 6): return y*y=64
Power(2, 3): return 2*y*y=8
Power(2, 1): return 2*y*y=2
Power(2, 0): return 1

15

Analysis

Algorithm Power(x, n):
Input: A number x and integer n = 0
Output: The value x^n
if n = 0 **then**
return 1
if n is odd **then**
y = Power(x, (n - 1)/ 2)
return x · y · y
else
y = Power(x, n/ 2)
return y · y

Each time we make a recursive call we halve the value of n; hence, we make log n recursive calls. That is, this method runs in O(log n) time.

It is important that we use a variable twice here rather than calling the method twice.

16

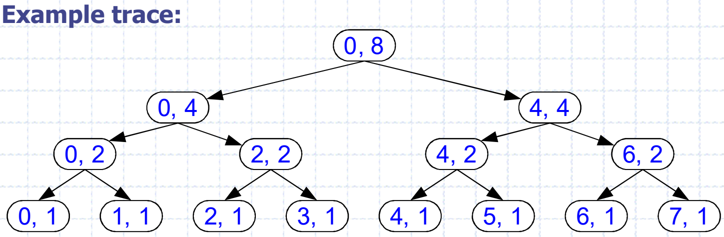
Ricorsione doppia

A Binary Recursive Method

- Problem: add all the numbers in an integer array A:

```
Algorithm BinarySum(A, i, n):
  Input: An array A and integers i and n
  Output: The sum of the n integers in A starting at index i
 if n = 1 then
 return A[i]
 return BinarySum(A, i, n/ 2) + BinarySum(A, i + n/ 2, n/ 2)
```

- **Example trace:**



Anche qui O(n): le chiamate sono le stesse.

Computing Fibonacci Numbers

- Fibonacci numbers are defined recursively:

$F_0 = 0$
$F_1 = 1$
$F_i = F_{i-1} + F_{i-2}$ for $i > 1$.

- Recursive algorithm (first attempt):

Algorithm BinaryFib(k):
  Input: Nonnegative integer k
  Output: The kth Fibonacci number F_k
if k <= 1 **then**
  return k
else
  return BinaryFib(k – 1) + BinaryFib(k – 2)

$f(K) = n_K > 2n_{K-2} > 2 \cdot 2n_{K-4} > 2 \cdot 2 \cdot 2n_{K-6}$

Mi fermo se $K - 2s = 0$

2 lo devo moltiplicare $s = \frac{K}{2}$

$f(K) > 2^{\frac{K}{2}}$

Analysis

- Let n_k be the number of recursive calls by BinaryFib(k)
  - $n_0 = 1$
  - $n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
  - $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
  - $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67$
- Note that n_k is at least doubles n_{k-2}
- That is, $n_k > 2^{k/2}$. It is exponential!

$n_7 > n_6$, almeno il doppio

19

A Better Fibonacci Algorithm

- Use linear recursion instead

```
Algorithm LinearFibonacci(k):
  Input: A nonnegative integer k
  Output: Pair of Fibonacci numbers (F_k , F_k-1)
 if k = 1 then
    return (k, 0)
 else
    (i, j) = LinearFibonacci(k – 1)
    return (i +j, i) *
```

- LinearFibonacci makes k–1 recursive calls

20

* Calcolo gli ultimi 2 numeri.