GNG 1106
Fundamentals of Engineering Computation
Computing Algorithms

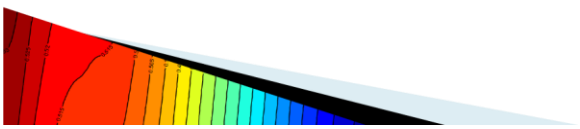
Gilbert Arbez

Fall 2016

Simple Algorithms for Sorting and Searching

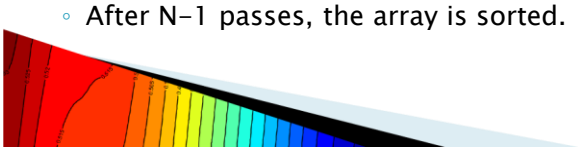
- A fundamental task in computing is sorting an array of data. Numerical data can be sorted in increasing or decreasing order while characters and character strings can be sorted in alphabetical order.
- Another fundamental task in computing is searching an array of data. Searching basically consists of determining if a key is present in an array, and if so returning its position. A key can be a number, a character or a string of characters. The search algorithm to be used depends on whether the data to be searched is sorted into a specific order.
- Many algorithms exist for sorting and searching arrays. We shall study a few simple ones.

Topic 1: Sorting Algorithms



Replacement Sort

- The replacement sort is a simple sorting algorithm used to organize a one dimensional array into increasing or decreasing numerical order.
- Iteration 1 of the algorithm design for sorting an array into increasing numerical order is described below.
  - During the first pass through the array, the largest value in the array is found and swapped with the element in the last position.
  - During the next pass through the array, the largest element in the top sub-array consisting of all elements except the last one, is found and swapped with the element in the before last position.
  - During the next pass through the array, the largest element in the top sub-array consisting of all elements except the last two, is found and swapped with the element in the third-last position.
  - ⋮
  - After N-1 passes, the array is sorted.



Function Design – Replacement Sort

`void sort_repl(int arr[], int num)`

Parameters

`num` - number of elements in the array

`arr` - reference to an array of integers to sort.

Logic/Algorithm

Use `pass` to count number of passes and initialise with 1

Loop while `pass` (incrementing `pass`) is less than `num` (number of elements in the array):

Use `largestPos` to store position of largest value and initialise to 0 (i.e. initially assume that value at index 0 has largest value).

Use `ix` to index into array start at index 1.

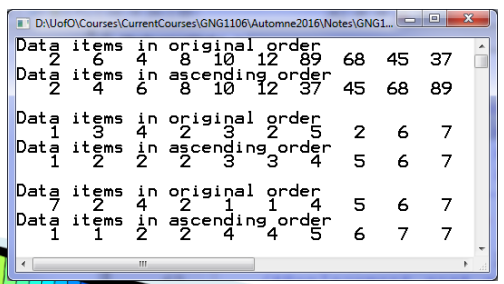
Loop while `ix` (incrementing `ix`) less then or equal to `num-pass` (i.e. scan array until last element of sub-array):

If `arr[ix]` is larger than `arr[largestPos]`, assign `ix` to `largestPos`

Exchange values of `arr[num-pass]` and `arr[largestPos]`

5

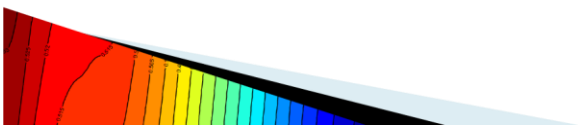
- Only `num-1` passes through the array are required.
  - Note that the function is generalized by using the parameter `num`
- During a pass:
  - The position of the largest number in the sub-array under consideration is found and the value is swapped with the number in the last position of the sub-array.
- It is clear that after the first pass, the largest value is in the last position.
- After a pass the number of elements in the sub-array under consideration is reduced by one (`num-pass`).
- The replacement sort algorithm is simple and reasonably efficient.
- See CodeBlocks Project ReplacementSort



6

Bubble Sort

- The bubble sort is a simple sorting algorithm used to organize a one dimensional array in increasing or decreasing numerical order. This algorithm is called "bubble sort" because the small values rise to the top of the array like bubbles in a champagne glass while the larger values sink to the bottom of the array, as the algorithm executes.
- Iteration 1 of algorithm design for the bubble sort is described below.
  - During a pass through the array, successive pairs of elements are compared.
  - If a pair of elements is in increasing order then the elements are not swapped.
  - If a pair of elements is in decreasing order then their position in the array is swapped.
  - Many passes must be performed through the array:
    - after the first pass the largest element in the array will be in the last position;
    - after the second pass, the next largest element in the array will be in the before last position;
    - ⋮
    - after the N-1th pass, the N-1th largest element is in the second position and the smallest element is in the first position.



Function Design – Bubble Sort

`void sort_bubble(int num, int arr[])`

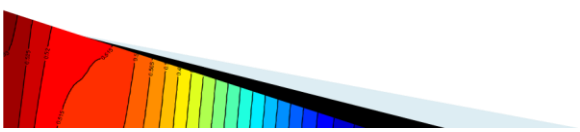
Parameters

`num` - number of elements in the array

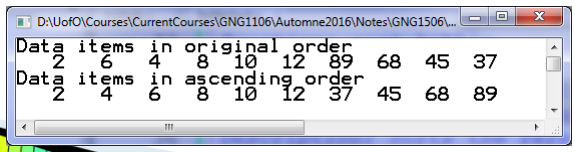
`arr` - reference to an array of integers to sort.

Logic/Algorithm

Use `pass` to count number of passes and initialise with 1
Loop while `pass` (incrementing `pass`) is less than `num` (number of elements in the array):
    Use `ix` to index into array start at index 1.
    Loop while `ix` (incrementing `ix`) less then or equal to `num-2` (i.e. scan array until second last element of array):
        If `arr[ix]` greater than `arr[ix+1]`, exchange the values of `arr[ix]` and `arr[ix+1]`



- See CodeBlocks project BubbleSort
- Only `num-1` passes through the array are required.
- During a pass:
  - `a[0]` is compared to `a[1]` and are swapped if required,
  - `a[1]` is compared to `a[2]` and are swapped if required,
  - `a[2]` is compared to `a[3]` and are swapped if required,
  - ⋮
  - `a[8]` is compared to `a[9]` and are swapped if required.
  - It is clear that after the first pass, the largest value 89, is in the last position: `a[9]`.
- The bubble sort algorithm is simple but inefficient. The algorithm requires about $(num)^2$ comparisons in order to sort an array of `num` elements.
  - Is the bubble sort faster than the replacement sort?
  - Can the bubble sort be made as efficient as the replacement sort?



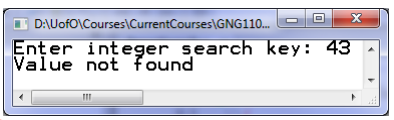
9

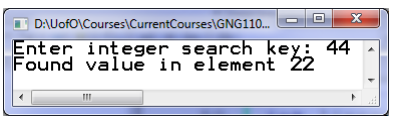
Topic 2: Searching Algorithms

10

Linear Search

- Algorithm design (iteration 1): The linear search technique consists of comparing sequentially each element of a one dimensional array with a key and returning the position of the key in the array as soon as it is found.
- On average, (`num`)/2 comparisons are required until the key is found in an array of `num` elements.
- If the array is sorted then the binary search technique is preferable since it will find the key much more rapidly.
- See CodeBlocks project LinearSearch





11

Function Design – Linear Search

```
int linearSearch(int key, int num, int arr[])
```

Parameters

`key` - the value to search in the array
`num` - the number of elements in the array
`arr` - reference to an array of integer values

Note how an indeterminate loop is used to make the algorithm efficient.

Return Value

Position (index) of key in array or -1 if key not found.

Logic/Algorithm

Use `pos` to store position of key and initialise to -1 (assume the value of `key` does not exist in the array.

Use `ix` to index into array and initialize to 0.

Loop while `ix` (incrementing `ix`) less then or equal to `num-1` (i.e. scan array until last element of array) and `pos` is equal to -1 (key not found):

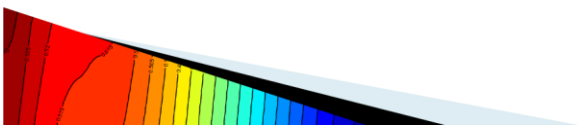
If `arr[ix]` equal to `key`, assign `ix` to `pos`.

Return the value of `pos`.

12

Binary Search

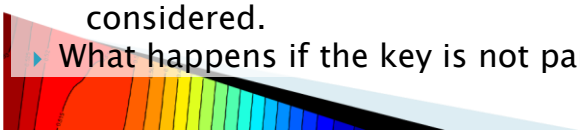
- If a one dimensional array of numbers is sorted then the binary search technique should be used instead of the linear search technique since the latter it is much faster.
- In the binary search technique, we use the fact that the array is sorted in increasing numerical order; i.e.: **`array[0]`** contains the smallest value and **`array[num-1]`** contains the largest value.



13

Binary Search – Algorithm

- The key is compared with the element in the middle of the array.
- If the key is less than the middle element, then we search the bottom half of the array.
- If the key is greater than the middle element, then we search the top half.
- Now we compare the key to the middle element of the top or bottom half of the array and divide accordingly.
- We continue comparing with the middle element and splitting the elements until we find the key.
- After each comparison one of two things has happened:
  - ➔we have found the key, or
  - ➔we have eliminated half of the elements being considered.
- What happens if the key is not part of the table?



14

Function Design – binary search

`int binarySearch(int key, int num, int arr[])`

Parameters:

- `key` - the value to search in the array
- `num` - the number of elements in the array
- `arr` - reference to an array of integer values

Return Value:

Position (index) of key in array or -1 if key not found.

Logic/algorithm

Use `pos` to store position of key and initialise to -1 (assume the value of `key` does not exist in the array.

Initialise `low` to 0 and `high` to num-1 (start with the whole array)

Repeat while `low` is less than or equal to `high` (the subarray still contains values) and `pos` equals -1 (key not found):

Assign `(high-low)/2` to `middle` (to see if `key` is found in middle)

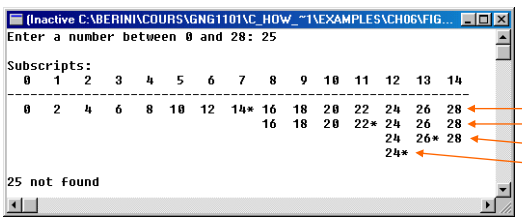
If `arr[middle]` equal `key`, assign `middle` to `pos`

Otherwise if `key` less than `arr[middle]`, assign `center-1` to `high` (drop the upper half of the subarray)

Otherwise assign `center+1` to `low` (drop the lower half of the subarray)

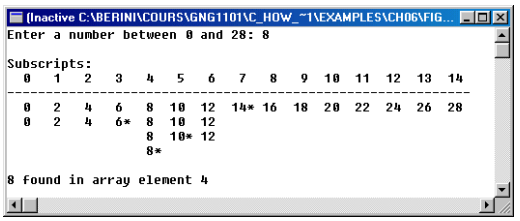
Return the value of `pos`

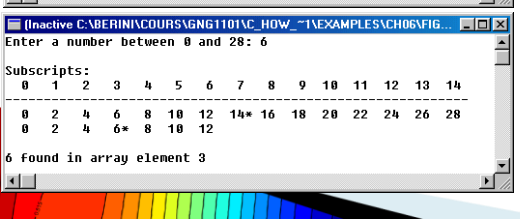
- Example executions of CodeBlocks project BinarySearch where the array under consideration is listed.



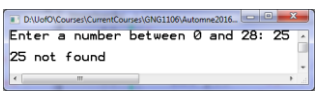
BinarySearch is called with `low=0` and `high=14`

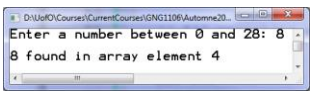
```
middle=7,  low=8,  high=14
middle=11, low=12, high=14
middle=13, low=12, high=12
middle=12, low=13, high=12
return -1
```

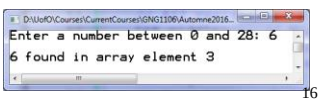




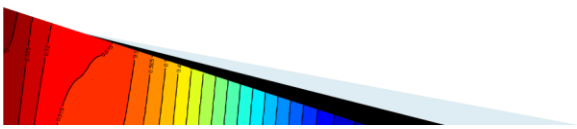
You can trace through the execution for the remaining example runs. Use the debugger to follow the changes in the variables.



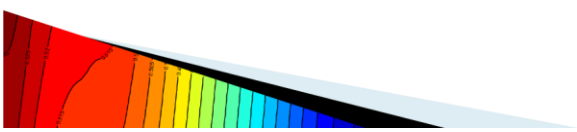




- The binary search technique is very efficient since at each iteration, we eliminate half of the elements in the sub-array that is being considered.
  - An array of 2^{20} elements requires a maximum of 20 comparisons.
  - An array of 2^{30} elements requires a maximum of 30 comparisons.
    - 2^{30} elements is about 10^9 elements; we would require on average 500 million comparisons using the linear search technique in order to find a key.

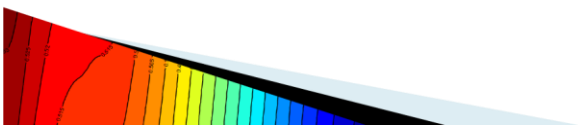


Topic 3: Recursion



Recursive Functions

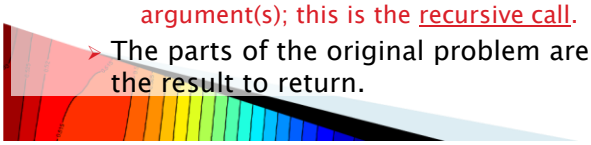
- Thus far, we have studied and created programs that have a hierarchical structure.
  - Instructions and functions are executed in a sequential manner from top to bottom.
- To solve certain types of problems, it is practical to use a function that can call itself; such functions are termed recursive functions.
  - A recursive function can call itself directly or indirectly via another function.



19

Recursive Functions: Operation

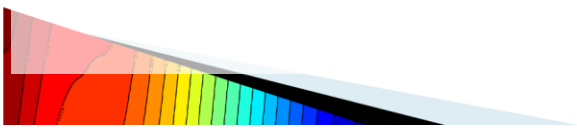
- In general, a recursive function works as follows:
  - The first call originates from a calling function (e.g. `main()`), and arguments are passed to the function.
  - A recursive function can solve directly only the simplest case for any given problem.
    - If the simplest case is identified from the value of the arguments, then the function returns the result for this case.
  - If the simplest case is not requested, then the function breaks down the problem into two parts:
    - A part that it can solve immediately but only with the results from the other part and,
    - A part that cannot be solved immediately but that is a simplified version of the original problem and tends towards the simplest case.
      - The function then calls itself with the simpler case(s) sent in as argument(s); this is the recursive call.
    - The parts of the original problem are usually combined to give the result to return.



20

Recursive Functions: Operation (continued)

- All recursive calls are executed while the original and previous calls are suspended.
  - The recursion will end only if the arguments in the recursive calls converge to the simplest case (often called the base case).
  - When the simplest case is attained, the series of recursive function calls are completed and return right up to the original function call which provides the desired result to the calling function (e.g. `main()`).



Recursive Function: The Factorial

- The factorial of a positive integer *n* is denoted *n!* and is defined as:

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$$

  with: 1! = 1 and 0! = 1

- The factorial can also be defined mathematically as a recursion:

$$n! = n \cdot (n-1)!$$

- Since the definition of the factorial is recursive, we may program a recursive function to compute the factorial of an integer.
  - The simplest case (base case) is when the argument is equal to 1 or 0:
    1! = 1 or 0! = 1
  - The recursive call should contain two parts if the argument is greater than 1:

Part that is solved immediately using results from the simpler case.

$n! = n \times (n-1)!$

Recursive call with the argument that tends towards the simplest case.

Design

`int factorial(int integer)`

Parameters

integer– integer value

Return Value

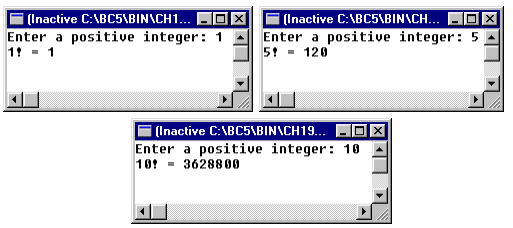
integer!

Logic/algorithm

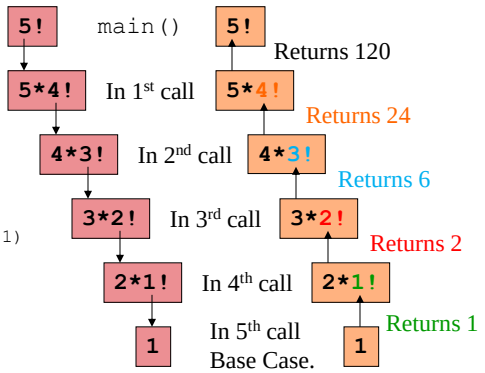
Uses recursion:

- Base case: `integer` ≤ 1
  - The factorial is 1: `fact` = 1
- Other cases
  - Finds the factorial of `integer` – 1 and multiplies by `integer`, i.e.
  - `fact` = `integer` X `factorial(integer - 1)`
- Returns the value of `fact`

See the CodeBlocks project RecursiveFactorial



Progression of recursive calls for the case 5!



Recursive Function: The Fibonacci Series

- The Fibonacci series is:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21…
  - Any number (except for the first 2) in the Fibonacci series is obtained by adding the previous two numbers.
- Mathematically, the expression for a number in the Fibonacci series is defined recursively:
  - Fibonacci(0) = 0
  - Fibonacci(1) = 1
  - Fibonacci(n) = Fibonacci(n−1) + Fibonacci(n−2)
  - E.g.:
    - The 0th Fibonacci number is 0
    - The 1st Fibonacci number is 1
    - The 2nd Fibonacci number is 1
    - The 3rd Fibonacci number is 2

`int fibonacci(int nth)`

Parameters

nth - integer giving the position in the series of the desired Fibonacci number

Return Value

The nth Fibonacci number.

Logic/Algorithm

Uses recursion:

- Base case: `nth` ≤ 1
  - The Fibonacci number is nth:

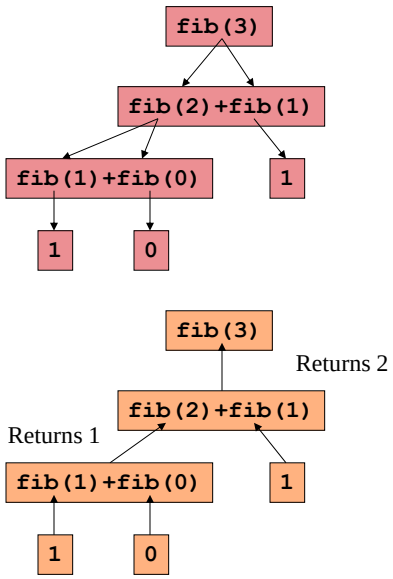
```
fibNumber = nth
```

- Other cases
  - The Fibonacci number is the sum of the two previous Fibonacci numbers, i.e.

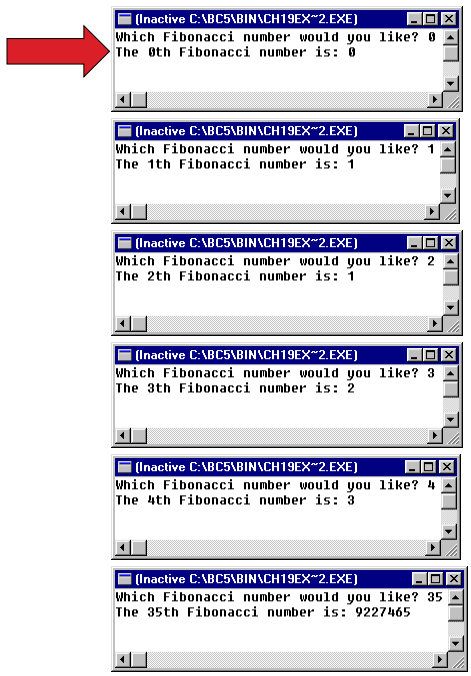
```
fibNumber = fibonacci(nth) +
            fibonacci(nth))
```

- Returns the value of `fibNumber`

Progression of recursive calls for the case `fibonacci(3)`.

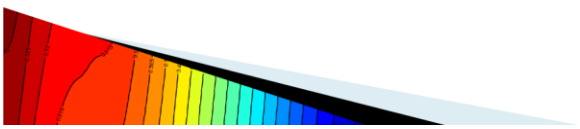


Output generated by the program found in the CodeBlocks project RecursiveFibonacci.

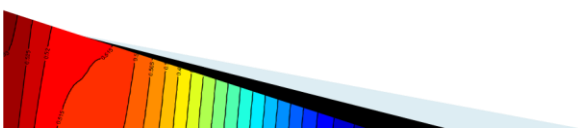


Recursion or Iteration?

- Recursion and iteration have many similarities.
  - Both can be used to repeat a prescribed number of commands.
    - Repetition by using a looping structure.
    - Repetition via recursive function calls.
  - Both have a test condition that determines when to end repetition.
    - Loops have a condition based on the final value of a counter.
    - Recursive functions have the "simplest" or "base" case.
  - Both converge toward their stopping conditions during execution.
    - Counters converge toward their final value during looping.
    - Arguments of a recursive function are simplified toward the base case.
  - Both can execute infinitely.
    - Loops may have an incorrect termination expression.
    - Recursive functions may have an improperly specified base case.
- Any problem that can be resolved recursively can also be resolved iteratively.



- Recursion has a major disadvantage.
  - Recursion invokes many times the function call mechanism, which is expensive in execution time and in memory usage.
- A recursive approach however is preferred when the nature of a problem's solution is recursive and an equivalent iterative approach would be difficult and cumbersome to implement compared to the recursive solution… assuming the performance penalty is unimportant.
- Careful with the recursive approach for computing a Fibonacci number.
  - Each call to the recursive function `fibonacci` can result in two other recursive calls to `fibonacci`.
  - The number of calls required to compute the n^{th} Fibonacci number is of the order of 2^n, which can rapidly become a very large number.
    - Eg: The 10^{th} Fibonacci number requires about $2^{10} \cong 10^3$ calls to `fibonacci`.
      - The 20^{th} Fibonacci number requires about $2^{20} \cong 10^6$ calls to `fibonacci`.
      - The 35^{th} Fibonacci number requires about $2^{35} \cong 35 \times 10^9$ calls to `fibonacci`.
  - Even though a recursive function to compute a Fibonacci number is an elegant solution, an iterative solution is preferable since a single loop of order n is required to compute the n^{th} Fibonacci number.



Next Module