ISEP
Algorithmics and Advanced programming
February 8th 2017

Tutorial course 1 : Algorithmic Complexity

Instructions : Prepare a report including the source code and the results in a pdf file and upload it to moodle into the folder "Report Tutorial course"

1 Objective

— Understand the different types of algorithmic complexity.
— Compare the running time of different algorithms.
— Get familiar with sorting algorithms and their complexity.

2 The input data

The running time depends mainly on the size of the input data to be processed.

In this tutorial course, we will use the following utility class to generate random data.

```
1  import java.util.Random;
2
3  public class RandomData {
4          public static int[] generate1d(int nbVals, int min, int max){
5                  int[] res= new int[nbVals];
6                  Random generator = new Random();
7                  for(int i=0; i != nbVals; ++i){
8                          res[i]= (int)((generator.nextLong()% ((long)max-min))+min);
9                  }
10                 return res;
11         }
12 }
```

The function `generate1d()` returns an array of `nbVals` numbers ranging from `min` to `max`. The numbers are generated randomly.

3 Minimim of an array

Consider the problem of finding the minimum element of an array of numerical values. Design an algorithm to solve this problem. How long does it take to find the minimum of such a sequence ? What is the complexity of your algorithm.

Use the function `generate1d()` to generate arrays of different sizes ranging from 0 to 500 by 10.

Run the algorithm you designed for the generated arrays and calculate the running time. Make a plot of the running time versus the number of elements. Comment on the results. Do the experimental results agree with the theorethical complexity of your algorithm ?

4 Sorting algorithms

In this section, you will be ask to use different sorting algorithms to sort random arrays of different sizes in ascending order.
The basic operations of sorting algorithms are *comparison* and *exchange*. In order to *exchange* values in a (fixed-size) array, one has to swap elements. This can be achieved by the following code :

```java
public static void swap(int[] data, int i, int j){
int tmp= data[i];
data[i]= data[j];
data[j]= tmp;
}
```

What is the algorithmic complexity of this function ?

4.1 *Selection sort*

Selection sort is a sorting algorithm based on the fact that when an array is sorted (in ascending order), given an element i, this is the minimum of the elements of indices greater or equal to the idex of i. The selection sort works as follows :

1. Look through the entire array for the smallest element, once find it, swap it (the smallest element) with the first element of the array.
2. Next, look for the smallest element in the remaining array (an array of (n-1) elements) and swap it with the second element.
3. Next, look for the smallest element in the remaining array (the array without first and second elements) and swap it with the third element, and so on.

Running times are compounded (multiplied) in nested loops, as we can see in the following implementation selection sort functions.

```java
public static int minimumIndex(int[] data, int begin, int end){
    int res= begin;
                for(int i=begin+1; i != end; ++i){
                    if(data[i] < data[res]){
                                res= i;
                        }
                }
                return res;
        }

public static void sort(int[] data){
    if(data.length < 2){return;}
    for(int i=0; i != data.length-1; ++i){
        swap(data, i, minimumIndex(data, i, data.length));
    }
}
```

What is the between the running time and the number of elements to sort ? What is the complexity of the *Selection Sort* ?

```
public class SelectionSort {
  <<sort-main>>
  <<swap>>
  <<selection-sort-functions>>
 }
```

Run the *Selection Sort* algorithm for arrays of different sizes ranging from 0 to 500 by 10 (use `generate1d()` function). Make a plot of the running time versus the number of elements. Comment on the results. Do the experimental results agree with the theorethical complexity of this algorithm ?

4.2 Bubble Sort

Another way to approach sorting is to consider that in a sorted array, any two adjacent elements are in relative order. Bubble sort takes (all) adjacent pairs of elements and orders them. This process is repeated until all adjacent pairs are ordered. Try to understand the relationship between the running time and the number on elements to sort with bubble sort, by reading the following implementation functions.

```
public static void sort(int[] data){
    if(data.length < 2){return;}
    boolean hadToSwap= false;
    do{
        hadToSwap=false;
        for(int i= 0; i != data.length-1; ++i){
            if(data[i] > data[i+1]){
                swap(data, i, i+1);
                hadToSwap= true;
} }
    }while(hadToSwap);
}
```

```
public class BubbleSort {
  <<sort-main>>
  <<swap>>
  <<bubble-sort-functions>>
  }
```

Run the *Bubble Sort* algorithm for arrays of different sizes ranging from 0 to 500 by 10 (use `generate1d()` function). Make a plot of the running time versus the number of elements. Comment on the results. Do the experimental results agree with the theorethical complexity of this algorithm ?

4.3 Merge Sort

Merge sort is based on the *Divide and conquer* paradigm. *Merge sort* involves the following steps :

1. Recursively divide the array into two (or more) subarrays.
2. Sorte ach subarray(Conquer)
3. Mergethemintoone(inasmartway !)

Given the following implementation to merge two sorted consecutive intervals *[begin, middle[* and *[middle, end[*, can you tell the relationship between the intervals sizes and the running time ?

```
public static void mergeSorted(int data[], int begin, int middle, int end){
        int[] tmp= new int[middle-begin];
        System.arraycopy(data, begin, tmp, 0, tmp.length);
        int i=0, j=middle, dest=begin;
        while((i< tmp.length) && (j<end)){
                data[dest++]= (tmp[i] < data[j]) ? tmp[i++] : data[j++] ;
        }
  while(i < tmp.length){
      data[dest++]= tmp[i++];
        }
}
```

What is the algorithmic complexity of this algorithm ? How much memory does it use ?

Given the following implementation of a merge sort, can you tell the relationship between the running time and the number of elements to sort ?

```
public class MergeSort {
  <<sort-main>>
  <<merge-sorted>>
  public static void sort(int[] data){
        sort(data, 0, data.length);
  }
        public static void sort(int[] data, int begin, int end){
                if((end-begin) < 2){return;}
                int middle= (end+begin)/2;
                sort(data, begin, middle);
                sort(data, middle, end);
                mergeSorted(data, begin, middle, end);
        }
}
```

What is the relationship between the running time of the *merge sort* algorithm and the number of elements ? What is its complexity ? Run the *Merge Sort* algorithm for arrays of different sizes ranging from 0 to 500 by 10 (use `generate1d()` function). Make a plot of the running time versus the number of elements. Comment on the results.

4.4 Quick Sort

Another way to "divide and conquer" for a sorting algorithm is to pre-process the data before dividing so as to make the merging trivial. *Quicksort* does so by partitioning the data according to a pivot element. This enables in-place sorting without any external memory.

To be sure to reduce the intervals when doing the recursive calls after the partition around the pivot, we make sure that the pivot is processed and we will recurse excluding this value. To enable that, we have to move the pivot value at the beginning of the second interval (so just after the first one). We will then be able to skip this value in the recursive call (cf. line 11).

```
public static int partition(int[] data, int begin, int end, int pivotIdx){
      swap(data, pivotIdx, --end);
      pivotIdx= end;
      int pivot= data[pivotIdx];
          //invariant is that everything before begin is known to be < pivot
          // and everything after end is known to be >= pivot
        while(begin != end){
            if(data[begin] >= pivot){
                swap(data, begin, --end);
        }else{
                ++begin;
                }
        }
    swap(data, pivotIdx, begin);
    return begin;
  }
```

```
public static void sort(int[] data){
        sort(data, 0, data.length);
}

public static void sort(int[] data, int begin, int end){
        if((end-begin) < 2){ return; }
        int m= partition(data, begin, end, (end+begin)/2);
         sort(data, begin, m);
        sort(data, m+1, end); // +1 for convergence }
```

What is the relationship between the running time of the *quick sort* algorithm and the number of elements ? What is its complexity ? Run the *quick sort* algorithm for arrays of different sizes ranging from 0 to 500 by 10 (use `generate1d()` function). Make a plot of the running time versus the number of elements. Comment on the results.

4.5 Comparison of Sorting algorithms

Make a unique plot of running time versus the number of elements for the 4 sorting algorithms : *bubble sort*, *selection sort*, *merge sort* and *quick sort*. Comment on your graphic.

5 Binary search

Consider the problem of searching a value in a sortes array. An efficient algorithm to solve this problem is called *binary search*. The principle is to compare the searching value with the median. Whether the median is lower or greater than the searched value, this one comparison allows to discard half (respectively the upper or lower half) of the array, by transitivity of the ordering relationship. The following algorithm is an implementation of *binary search :*

```
1  public class Dichotomy {
2
3  public static void main(String[] args){
4    int nbVals= Integer.parseInt(args[0]);
5    int[] data= new int[nbVals];
6    for(int i=0; i != data.length; ++i){
7      data[i]= 2*i;
8    }
9    System.out.println(lowerBound(data,  Integer.parseInt(args[1])));
10 }
11
12 public static int indexOfOrdered(int[] data, int v){
13   int res=lowerBound(data, v);
14   if((res==data.length) || (data[res] != v)){
15         res= -1;
16   }
17 return res;
18 }
19 // index of first element >= v
20  public static int lowerBound(int[] data, int v){
21          return lowerBound(data, v, 0, data.length);
22  }
23
24 public static int lowerBound(int[] data, int v, int begin, int end){
25   if(begin == end){ return begin;}
26   int m= (begin + end)/2;
27   return data[m] < v ? lowerBound(data, v, m+1, end) : lowerBound(data, v, begin, m);
28 }
29
30 public static int lowerBoundTCO(int[] data, int v){
31   int begin=0, end=data.length;
32   while(begin != end){
33     int m= (begin + end)/2;
34     if(data[m] < v){
35      begin= m+1;
36     }
37     else{
38          end= m;
39         }
40   }
41   return begin;
42  }
43 }
```

What is the algorithmic complexity ? What is the memory complexity ? Make a plot running time versus number of elements for arrays of different sizes.