Universidad Carlos III de Madrid
www.uc3m.es

Lesson 7

Algorithms with arrays

Programming

Grade in Computer Engineering

Outline

1. **Search**
   - **Linear**
   - **Binary**
2. **Sort**
   - **Bubble**
   - **Selection**
   - **Insertion**

1. **Search**
   - **Linear**
   - **Binary**
2. Sort
   - Bubble
   - Selection
   - Insertion

Search algorithms aim at finding a value in a collection (usually, the first occurrence)

- Input: Array of values *list*, value to find *e*
- Output: Position of *e* in *a*; -1 if not found

Find a value in an array of integers

```
public static int find(int [] list, int e)
   list = {5, 6, 3, 1, 8, 9, 0, 2, 4, 1, 7}
   e = 1
```

Output:

```
3
```

Looks for the value *e* sequentially in *list*:

```
location = -1;
i = 0;
found = false;

while ( (!found) && (i < list.length))

    if (list[i] == e)
        location = i;
        found = true;

    i++;

return location;
```

At most, *list.length* tests are needed

Without further assumptions on *list*, it is the most efficient (non-parallel) search algorithm

```java
/** Linear search
 * @param list Sequence of elements
 * @param e Element to find
 * @return Position of e in list; -1 if not found */

public static int linearSearch(int [] list, int e) {
    int location = -1;
    boolean found = false;

    for(int i=0; i<list.length && !found; i++)
        if(list[i] == e) {
            location = i;
            found = true;
        }

    return location;
}
```

search.basic.Algorithms

If *List* is ordered, we can narrow the search to one half of the array:

```
location = -1;
left   = 0;
right  = list.length - 1;
middle = list.length / 2;
found  = false;

while ( (left <= right) and (!found) )

    if ( list[middle] == e )
        found = true;
        location = middle;

    else if (e < list[middle] )
        right = middle – 1;

    else
        left = middle + 1;

    middle = (left+right) / 2;

return location;
```

At most, *Log(list.length)* comparison are needed

```java
/** Binary search
 * @param list Ordered sequence of elements
 * @param e Element to find
 * @return Position of e in list; -1 if not found */

public static int binarySearch(int [] list, int e) {
    int location = -1;
    int left   = 0;
    int right  = list.length - 1;
    int middle = list.length / 2;
    boolean found = false;

    while(left <= right && !found) {
        if(e == list[middle]) {
            found = true;
            location = middle;
        } else if(e < list[middle]) {
            right = middle-1;
        } else {
            left  = middle+1;
        }
        middle = (right+left) / 2;
    }

    return location;
}
```

search.basic.Algorithms

1. Search
   - Linear
   - Binary
2. **Sort**
   - **Bubble**
   - **Selection**
   - **Insertion**

Sort algorithms aim at rearranging the values of a collection to position them in order (usually, in increasing order)

Input: Array of values *list*

Output: Array of values *list** ordered

Sort array

```
public static void sort(int [] list)
   list = {5, 6, 3, 1, 8, 9, 0, 2, 4, 1, 7}
```

Output:

```
   list = {0, 1, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Direct sorting algorithms

Most popular

- Direct Swapping (***Bubble*** *sort*)
- Direct Insertion (***Insertion*** *sort*)
- Direct Selection (***Selection*** *sort*)

Features

- Simple algorithms
- Not very efficient: complexity is $\mathcal{O}(n^2)$
- Can be used with small arrays

Advanced sorting algorithms

Most popular

- Shell
- Quicksort
- Heapsort

Features

- Sophisticated algorithms
- Efficient: complexity is $\mathcal{O}(n * \log n)$
- Are used with large arrays

Just for fun! Bogosort: Random reordering of the array

H. Gruber, M. Holzer and O. Ruepp: *Sorting the Slow Way: An Analysis of Perversely Awful Randomized Sorting Algorithms*, 4th International Conference on Fun with Algorithms, Castiglioncello, Italy, 2007, Lecture Notes in Computer Science 4475, pp. 183-197.

Idea:

Compare an element *list[i]* with the adjacent value *list[i+1]*

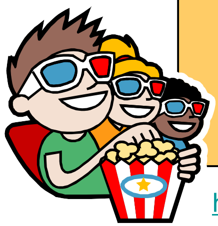
If *list[i]* > *list[i+1]*, the values are swapped

Repeat the procedure for the complete array while swaps are performed

```
do {
    swapped = false;

    for (i=0; i <= list.length-2; i++)
        if (list[i] > list[i+1])
            swap(list[i], list[i+1]);
            swapped = true;

} while (swapped);
```



http://www.youtube.com/watch?v=UnK5ueUgc88

2. Sorting algorithms

Bubble sort implementation

```
/** Bubble sort
 * @param list Array to sort */

public static void bubbleSort(int [] list) {
    boolean swapped;

    do {
        swapped = false;

        for(int i=0; i <= list.length-2; i++) {
            if(list[i] > list[i+1]) {
                int temp = list[i];
                list[i] = list[i+1];
                list[i+1] = temp;

                swapped = true;
            }
        }

    } while(swapped);
}
```

sorting.basic.Algorithms

Worst case: The array is reversed $\mathcal{O}(n^2)$

The outer while is executed *n* times, since swapping is always performed

Best case: Array is ordered $\mathcal{O}(n)$

swapped is not changed from false to true

Average: $\mathcal{O}(n^2)$

Swapping

Comparison

```
/** Bubble sort

public static void bubbleSort(int [] list) {
    boolean swapped;

    do {
        swapped = false;

        for(int i=0; i <= list.length-2; i++) {
            if(list[i] > list[i+1]) {
                int temp = list[i];
                list[i] = list[i+1];
                list[i+1] = temp;

                swapped = true;
            }
        }

    } while(swapped);
}
```

Idea:

For each value of the list (at position *i*),

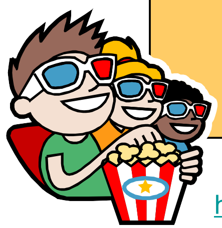
- Finds the smallest value (at position *minPos*) of the elements *i+1,…, list.length-1*

- If *list[i]* > *list[minPos]*, the values are swapped

```
for (i=0; i <= list.length-2; i++)
    minPos = i;
    for (int j=i+1; j < list.length; j++)

        if (list[j] < list[minPos])
            minPos = j;

    swap(list[i], list[minPos])
```



http://www.youtube.com/watch?v=TW3_7cD9L1A

2. Sorting algorithms

Selection sort implementation

```java
/** Selection sort ...

public static void selectionSort(int [] list) {

    for(int i=0; i <= list.length-2; i++) {
        int minPos = i;

        for(int j=i+1; j < list.length; j++)
            if(list[j] < list[minPos])
                minPos = j;

        int temp = list[i];
        list[i] = list[minPos];
        list[minPos] = temp;
    }
}
```

sorting.basic.Algorithms

- The number of comparison operations does not depend on the initial order of the values. It will be equal to the number of evaluations of the condition of the *if* $\mathcal{O}(n^2)$
- The number of swap-related operations depends on the initial order of the values

Swapping

Comparison

```
/** Selection sort ...

public static void selectionSort(int [] list) {

    for(int i=0; i <= list.length-2; i++) {
        int minPos = i;

        for(int j=i+1; j < list.length; j++)
            if(list[j] < list[minPos])
                minPos = j;

        int temp = list[i];
        list[i] = list[minPos];
        list[minPos] = temp;
    }
}
```

Idea:

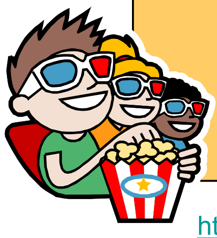
Assumes that the elements `0, …, i-1` of the list are ordered

Finds the position *k* in `0, …, i-1` where the element at position *i* should be placed

(Simultaneously) Shift to the right the values at `k, …, i-1` and inserts `list[i]` at position *k*

```
for (i=1; i < list.length; i++)
    e = list[i];
    j = i-1;
    while( (j >= 0) && (list[j] > e) )
        list[j+1] = list[j];
        j = j-1;

    list[j+1] = e;
```



http://www.youtube.com/watch?v=Fr0SmtN0IJM

Insertion sort implementation

```
/** Insertion sort ...

public static void insertionSort(int [] list) {

    for(int i=1; i < list.length; i++) {
        int e = list[i];
        int j = i-1;

        while(j>=0 && list[j] > e) {
            list[j+1] = list[j];
            j--;
        }

        list[j+1] = e;
    }
}
```

sorting.basic.Algorithms

Worst case: The array is reversed **$\mathcal{O}(n^2)$**

The inner while is executed until *j* < 0 (max. number of iterations)

Best case: Array is ordered **$\mathcal{O}(n)$**

The inner while is never executed

Average: **$\mathcal{O}(n^2)$**

Swapping

Comparison

```
/** Insertion sort

public static void insertionSort(int [] list) {

    for(int i=1; i < list.length; i++) {
        int e = list[i];
        int j = i-1;

        while(j>=0 && list[j] > e) {
            list[j+1] = list[j];
            j--;
        }

        list[j+1] = e;
    }
}
```

Algorithms can be compared according to the number of comparisons performed in the best case, worst case, and average case

Being *n* the length of the array:

Algorithm	Best ≈	Worst ≈	Average ≈
Bubble	n	n^2	n^2
Selection	n^2	n^2	n^2
Insertion	n	n^2	n^2
Quicksort	$n \cdot \log(n)$	n^2	$n \cdot \log(n)$

Bubble sort is the simplest, but also has a the higher worst-case execution time. Nevertheless, it behaves quite well with ordered arrays

Selection sort is easy to implement and more efficient that *Bubblesort*, but it behaves very bad even if the array is ordered (it cannot be known if the array is already sort at any iteration)

Insertion sort is simple to implement and behaves quite well for almost ordered arrays. It is also more efficient in practice

Develop a program to test the execution time of the three basic sorting methods for different array sizes = {1000, 2000, …, 20000}

The program must run 5 times each algorithm for an array size with different initial values.

The program must generate three text files (*bubble.txt*, *selection.txt*, *insertion.txt*) with this structure:

<array size> <average> <best time> <worst time>
<array size> <average> <best time> <worst time>
…

Represent the results (array size vs. average time) in a table and graphically (use Microsoft Excel).

bubble.txt

```
1000 4833840.0 3403156 8024134
2000 1.4096526E7 13973424 14377736
3000 3.1497941E7 31175959 32105338
4000 5.5688851E7 54822051 56
5000 8.6352455E7 85646500 86
6000 1.24527935E8 123430771
7000 1.69686803E8 168417844
8000 2.28952953E8 224974962
9000 2.83831813E8 281557013
10000 3.48586231E8 343487511
11000 4.23960098E8 421367031
12000 4.98431085E8 496283885
13000 5.84650563E8 579794918
14000 6.77815099E8 675493952
15000 7.77365111E8 773596854
16000 8.84868174E8 882779445
17000 1.00763699E9 100141714
18000 1.158422369E9 11433812
19000 1.294025106E9 12850064
20000 1.455827793E9 14340907
```

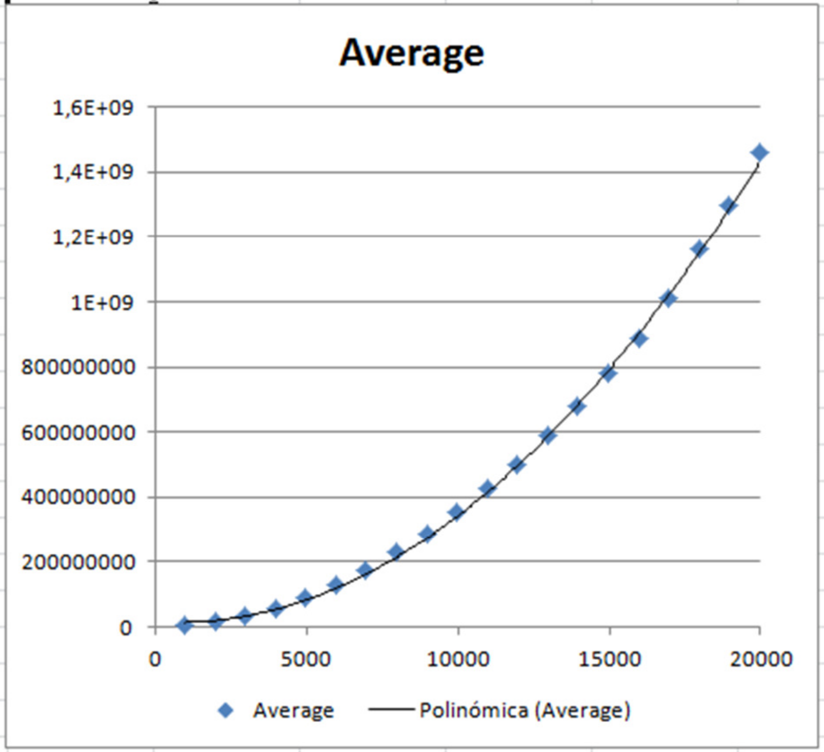
insertion.txt

```
1000 7630089.0 442934 8024134
2000 2.0468606E7 1737022 14377736
3000 4.5609526E7 3918934 32105338
4000 8.0775307E7 7016045 568
5000 1.25229588E8 10558045 8
6000 1.81017196E8 15933869 1
7000 2.47538263E8 21345869 1
8000 3.29694486E8 27784048 2
9000 4.12642319E8 34824538 2
10000 5.04918739E8 43222672
11000 6.15219224E8 51432095
12000 7.222861E8 62455075 50
13000 8.4770264E8 72818054 5
14000 9.83445688E8 85170811
15000 1.126760943E9 95896057
16000 1.283408785E9 11162703
17000 1.456268087E9 12470090
18000 1.67652615E9 141928374
19000 1.868753374E9 15816388
20000 2.082522708E9 17257242
```

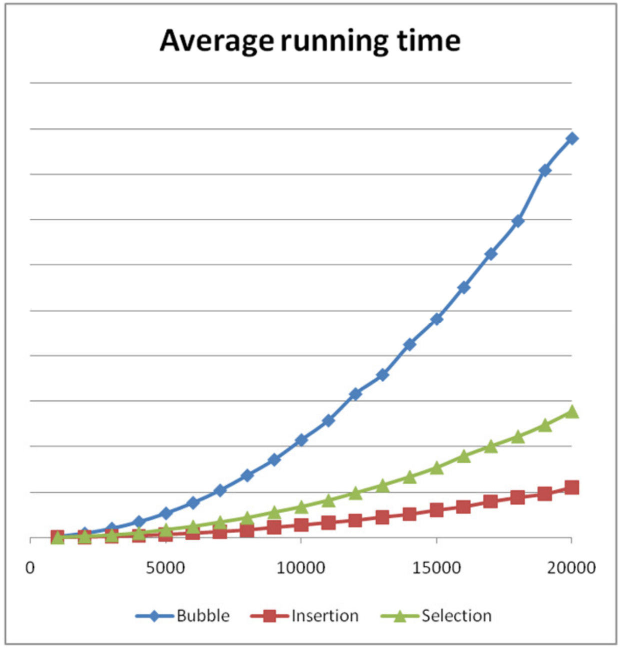
selection.txt

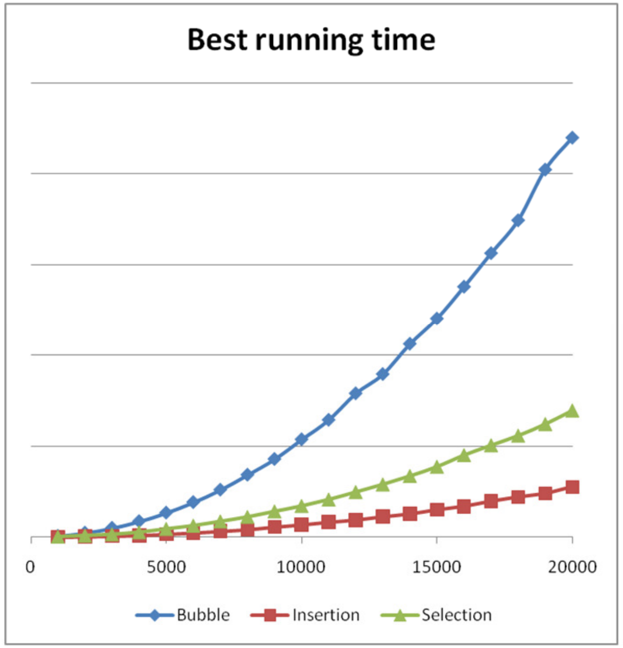
```
1000 6591493.0 1134222 8024134
2000 1.8635566E7 4493867 14377736
3000 4.163877E7 10073557 32105338
4000 7.3630293E7 17875246 56861696
5000 1.14445285E8 27892093 86805655
6000 1.64928936E8 40197917 125393660
7000 2.25588811E8 55526540 170736644
8000 3.0151258E8 71769387 238769942
9000 3.77440163E8 91183657 287583059
10000 4.6111341E8 112099303 355747379
11000 5.63095155E8 136419084 430040899
12000 6.59540332E8 160644998 501597131
13000 7.74420045E8 189278735 588853542
14000 8.9756198E8 219361050 680047464
15000 1.030052255E9 251590566 781549610
16000 1.171092416E9 285293592 889809180
17000 1.330757093E9 322367508 1022621241
18000 1.532992852E9 366285869 1199534019
19000 1.706986963E9 408624141 1310799766
20000 1.908151566E9 449594990 1473585120
```

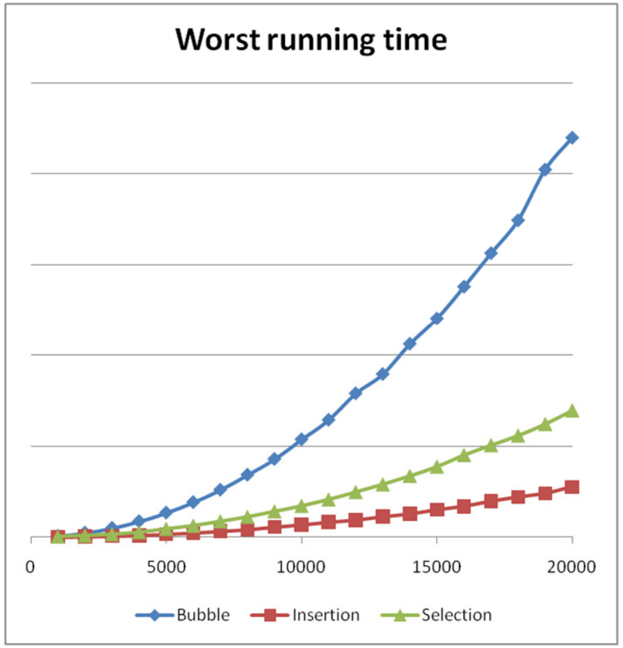
Size	Average	Best	Worst
1000	4833840	3403156	8024134
2000	1,41E+07	13973424	14377736
3000	3,15E+07	31175959	32105338
4000	5,57E+07	54822051	56861696
5000	8,64E+07	85646500	86805655
6000	1,25E+08	123430771	125393660
7000	1,70E+08	168417844	170736644
8000	2,29E+08	224974962	238769942
9000	2,84E+08	281557013	287583059
10000	3,49E+08	343487511	355747379
11000	4,24E+08	421367031	430040899
12000	4,98E+08	496283885	501597131
13000	5,85E+08	579794918	588853542
14000	6,78E+08	675493952	680047464
15000	7,77E+08	773596854	781549610
16000	8,85E+08	882779445	889809180
17000	1,01E+09	1001417149	1022621241
18000	1,16E+09	1143381212	1199534019
19000	1,29E+09	1285006475	1310799766
20000	1,46E+09	1434090715	1473585120



Bubble | Insertion | Selection | All







Results for 5 executions with random values in [0, 10)

1. **Search**
   - **Linear**
   - **Binary**
2. **Sort**
   - **Bubble**
   - **Selection**
   - **Insertion**

Search

Linear search

Binary search

Use? Binary search if values are sorted; otherwise, linear search

Sort

Bubble sort

Selection sort

Insertion sort

Use? None of them, go for *Quicksort*

Recommended lectures

H. M. Deitel, P. J. Deitel. *Java: How to Program. Prentice Hall, 2011 (9th Edition)*, **Chapter 19** [link]

Programming – Grado en Ingeniería Informática	
Authors	
Of this version: Juan Gómez Romero *Based on the work by:* Ángel García Olaya Manuel Pereira González Silvia de Castro García Gustavo Fernández-Baillo Cañas	Universidad Carlos III de Madrid