Quick sort. Selection.
Lower bound of comparison sort.
Other sorting algorithms

Chapter 3

2016

Quick sort – the idea

- An important drawback of merge sort is that sorting is not “in place” because the merge operation is using an additional array - $\Theta(n)$ additional storage. Also the combine step takes $\Theta(n)$ time. On the contrary, insertion sort and selection sort are “in place” sorts.
- Quick sort is also based on divide and conquer. However it takes a different approach making the division difficult and the combine easy. Actually there is no combine step. Additionally, sorting is done in place and the algorithm is very fast in the average case.
- Quick sort for sorting $A[p..r]$ takes two steps:
  - *Divide* (also called *partitioning*): the sequence $A[p..r]$ is partitioned in two non-empty sub-sequences $A[p..q]$ and $A[q+1..r]$ such that $A[i] \leq A[j]$ for all $i \in p..q$ and for all $j \in q+1..r$. Note that q is an output of the partitioning stage.
  - *Conquer*: sub-sequences $A[p..q]$ and $A[q+1..r]$ are sorted recursively. Note that because of how partitioning was done, the combine step is not needed.

Quick sort – the basic algorithm

```
QUICK-SORT(A, p, r)
1. if p < r then
2.     q ← PARTITION(A, p, r)
3.     QUICK-SORT(A, p, q)
3.     QUICK-SORT(A, q + 1, r)
```

To sort an array *A* the QUICK-SORT algorithm should be called as follows: QUICK-SORT(*A*,1,length[*A*]).

Partitioning is very important to obtain an efficient algorithm:

- should be done in linear time
- should be done in place
- the value of *q* should be more or less the mid of the array *A* in order to get an $\Theta(n \lg n)$ algorithm in the average case
- to avoid infinite looping, the returned value *q* should always satisfy $q < r$ (if *q* is equal to *r* then there is no partitioning !)

Quick sort is sometimes called *partition sort*.

Partitioning

```
PARTITION(A, p, r)
1.  x ← A[p]
2.  i ← p − 1
3.  j ← r + 1
4.  while TRUE do
5.        repeat
6.              j ← j − 1
7.        until A[j] ≤ x
8.        repeat
9.              i ← i + 1
10.       until A[i] ≥ x
11.       if i < j then
12.         exchange A[i] ↔ A[j]
13.       else
14.             return j
```

- **Subscripts *i* and *j* are never out of the array *a* bounds**. After the first pass *j* cannot be less then *p* because *x* = *A*[*p*] and *i* will be *p*. If the loop is re-entered then at the beginning of the loop *i*<*j* and *A*[*p*..*i*] ≤ *x* ≤ *A*[*j*..*r*]. This means that after the execution of the first two repeat loops *j* cannot be lower than the old value of *i* and *i* cannot be greater than the old value of *j*.
- After the first pass *i*=*p*. If partitioning stops after this pass then *j*=*p* < *r*. If the second pass is executed then *j* will be decremented at least twice times. **This proves that the returned value *j* is strictly less than *r***.
- **At the end of the partitioning algorithm *A*[*p*..*j*] ≤ *x* ≤ *A*[*j*+1..*r*].** If the algorithm stops after a single pass then this is obvious because *j* = *p*. Otherwise at the beginning of each pas (starting with the second) we have *i*<*j* and *A*[*p*..*i*] ≤ *x* ≤ *A*[*j*..*r*]. Each pass increases *i* and decreases *j* until eventually *i* ≥ *j*.
- **Partitioning takes Θ(*r*-*p*) time.**

Partitioning example

- x = 6, p = 1, r = 9,
  i = 0, j = 10
- 6 2 4 8 9 1 5 10 3
- j = 9, i = 1
- 3 2 4 8 9 1 5 10 6
- j = 7, i = 4
- 3 2 4 5 9 1 8 10 6
- j = 6, i = 5
- 3 2 4 5 1 9 8 10 6
- j = 5, i = 6
- return q = 5, stop !

If the array is already sorted then the partitioning algorithm returns $j = p$. So the partition is unbalanced:

- $A[p..p]$ of size 1
- $A[p+1..r]$ of size $p-r$

This partition is called a *bad partition*.
If the sizes of $A[p..q]$ and $A[q+1..r]$ are approximately equal then the partition is called a *good partition*.

Intuitive analysis of quick sort

- If there is a good split, i.e. the array is split evenly, we get:

  $T(n) = 2 \times T(n/2) + \Theta(n)$, so $T(n) = \Theta(n \times \lg n)$ like in merge sort.
- If there is a bad split, i.e. the array is unevenly split, we get $T(n) = T(n\text{-}1) + \Theta(n)$. Using the iteration method we get:

  $T(n) = \Sigma^{n}_{k=1}\Theta(k) = \Theta(\Sigma^{n}_{k=1}k) = \Theta(n^2)$

  The worst case occurs when the input array is sorted or reverse sorted.
- If all the inputs are equally likely then the average case running time of quick sort is $\Theta(n \times \lg n)$. The intuition is that if the input is random then some of the partitions will be balanced and some not. On the average, the partitioning procedure will produce a mixture of good and bad partitions.
- To simplify, let us assume that the good and bad partitions are alternating. Consider the case when we have a first bad partitioning followed by a second good one and a case with a first good partitioning. It can be easily noticed that the two cases are quite similar in efficiency. So, intuitively the bad partition can be ignored and on the average we will obtain an execution time $\Theta(n \times \lg n)$ like in the best case.

Randomized versions of quick sort – the idea

- Quick sort is very efficient if we assume that all the permutations of the input are equally likely. We have seen that if the input is almost sorted we can reach the worst case. Also the worst case occurs when the partitioning procedure produces bad partitions. We can use a randomized version of quick sort to avoid these situations.
- *Randomized algorithm* = an algorithm with the behavior dependent on the values of a random number generator.
- *Random number generator* = a function RANDOM(a,b) that produces a random integer number r, $a \leq r \leq b$, when called for integers $a < b$.
- There are two approaches to randomize quick sort:
  - Generate a random permutation of the input sequence, before calling quick sort. So practically, there will be no bad input. Generating a random permutation of an input array $A[1..n]$ can be done in time $\Theta(n)$. How ? Hint: *Fisher–Yates shuffle* – see next slides.
  - Randomize the partitioning procedure to generate a random partition. The idea is to use a random pivot.

Quick sort with randomized partitions

RANDOMIZED-PARTITION(A, p, r)

1. $i \leftarrow$ RANDOM(p, r)
2. exchange $A[p] \leftrightarrow A[i]$
3. **return** PARTITION(A, p, r)

RANDOMIZED-QUICK-SORT(A, p, r)

1. **if** $p < r$ **then**
2. $\quad q \leftarrow$ RANDOMIZED-PARTITION(A, p, r)
3. $\quad$ RANDOMIZED-QUICK-SORT(A, p, q)
3. $\quad$ RANDOMIZED-QUICK-SORT$(A, q + 1, r)$

- Before starting the partitioning process we exchange $A[p]$ with a random element of the array $A[p..r]$. In this way we actually generate a random pivot.
- The detailed analysis of the randomized version of quick sort is given in section 8.4 of the textbook and it assumes some mathematical insight.

Random permutation using Fisher–Yates shuffle

RANDOM-PERMUTATION(A, n)

1. **for** $i = n, 2$ **do**
2. $\quad k \leftarrow$ RANDOM$(1, i)$
3. $\quad$ exchange $A[k] \leftrightarrow A[i]$

1. Initially *A* is the unit permutation, *A*[*i*] = *i* for all $1 \le i \le n$. Think of writing down numbers 1, 2, ..., *n*, initially unmarked.
2. Pick a random number *k* between 1 and the number of unmarked numbers (initially *n*).
3. Counting from the low end, mark out the k^{th} number not yet marked out, and write it down elsewhere.
4. Repeat from step 2 until all the numbers have been marked out.
5. The sequence of numbers written down in step 3 is a random permutation of the original permutation.

Example:

[1 2 3 4 5] *i*=5, *k*=3 ⇒ [1 2 5 4 3] *i*=4, *k*=1 ⇒ [4 2 5 1 3] *i*=3, *k*=2 ⇒ [4 5 2 1 3] *i*=2, *k*=2 ⇒ [4 5 2 1 3]

Selection – problem formulation

- The *i^{th} order statistic* of a set of n elements is its i^{th} smallest element or the element with *rank* i.
- The *minimum* of a set is the 1^{st} order statistic for i=1.
- The *maximum* of a set of n elements is the n^{th} order statistic for $i = n$.
- The *median*:
  - When n is odd, $n = 2k+1$, there is a single median at $i = (n+1)/2 = k+1$.
  - When n is even, $n = 2k$, there are two medians at $n/2 = k$ and $n/2+1 = k+1$.
  - Note that for either parity the median occurs at $\lfloor (n+1)/2 \rfloor$ and $\lceil (n+1)/2 \rceil$.
- The *selection problem*:
  - Input: a set A of n distinct numbers and a number i such that $1 \leq i \leq n$.
  - Output: an element $x \in A$ such that x is larger than exactly i-1 elements of A.
- Selection can be solved using sorting:
  - first sort A and than take the i^{th} element. However there are better algorithms for solving the selection problem in a straightforward way, without the need to sort.
- The *maximum and minimum problems*, which are special instances of the selection problem, can be solved in $\Theta(n)$ time.
- The maximum (minimum) problem require at least n-1 comparisons.

Selection – randomized solution

```
RANDOMIZED-SELECT(A, p, r, i)
1. if p = r then
2.     return A[p]
3. q ← RANDOMIZED-PARTITION(A, p, r)
4. k ← q − p + 1
5. if i ≤ k then
6.     return RANDOMIZED-SELECT(A, p, q, i)
7. else
8.         return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

• This algorithm follows the idea of randomized quick sort. However we have a single recursive call instead of two.

• Line 3 partitions $A[p..r]$ into 2 nonempty sub-arrays $A[p..q]$ and $A[q+1..r]$.

• Line 4 finds the number k of elements in $A[p..q]$.

• The algorithm then determines the location of the i^{th} element. If $i \leq k$ then it is located in the lower sub-array. If $i > k$ then it is located in the upper sub-array. If $i = k$ then we have found it.

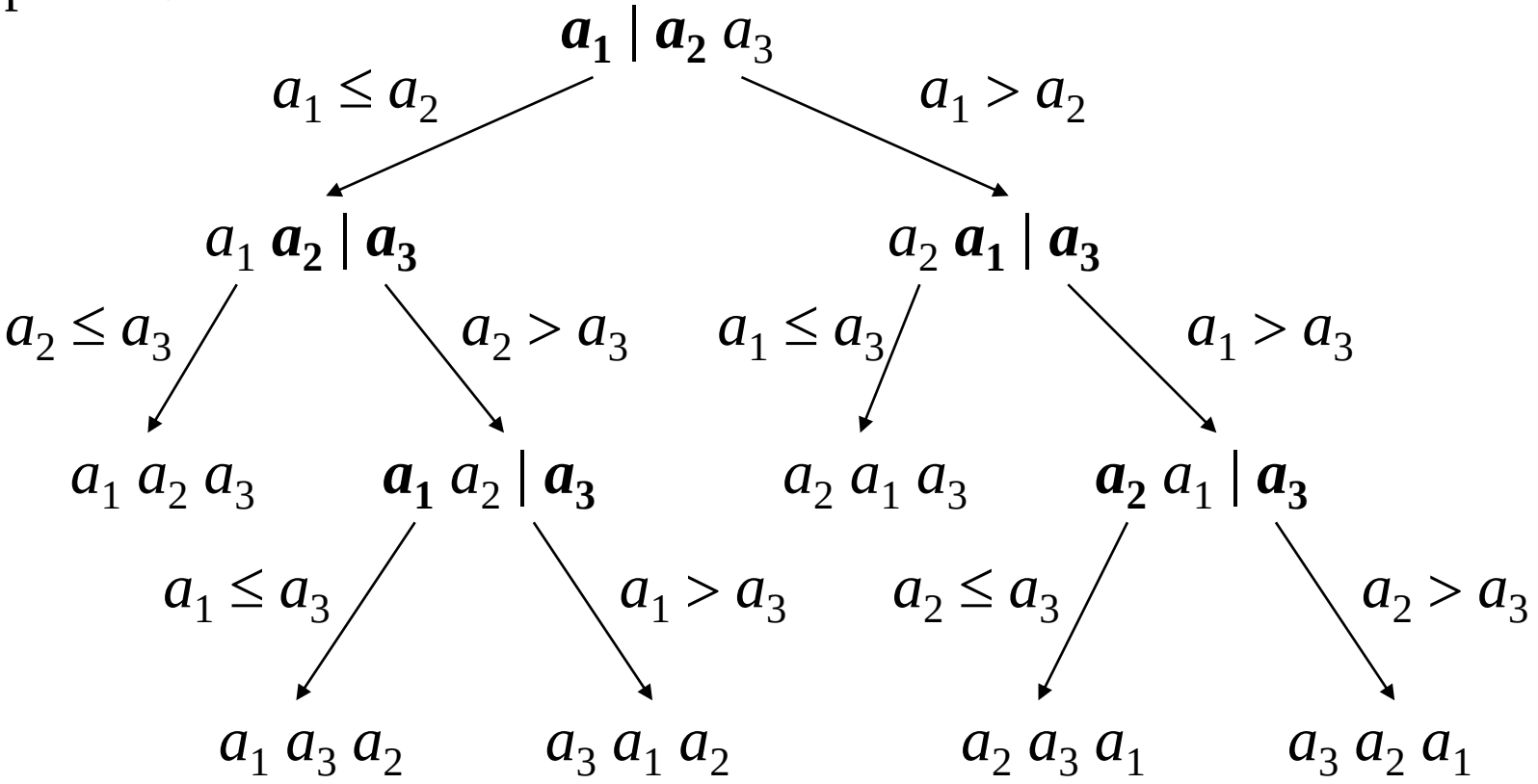
• Worst case time is $\Theta(n^2)$ and average case time in $\Theta(n)$.

Comparison sort

- Insertion sort, merge sort, selection sort and quick sort are all based on the operation of comparing two elements. Therefore, they are called *comparison sorts*.
- A comparison sort can be modeled using a *decision tree*.
  - There is one decision tree for a particular comparison sort algorithm and a given *n* (length of the array)
  - The tree shows where the algorithm splits based on a comparison between two elements of the array
  - The tree shows all the possible execution traces of the sorting algorithm
  - The longest path from the root to a leaf node (i.e. the tree height) gives the maximum number of comparisons of the algorithm
  - The tree has *n*! leaves, one leaf for each possible sort. A sort is a permutation of the array elements.

Decision tree for insertion sort

- In insertion sort the array is conceptually divided into a sorted array (leftmost) and the unsorted rest (rightmost) – separated with a vertical bar in the picture.
- The first element of the rightmost array in inserted into the leftmost array such that the resulted leftmost array stays sorted. The insertion is based on comparing successively the first element of the rightmost array with the elements of the leftmost array (starting with the last) – compared elements are shown in bold in the picture.



Lower bound for comparison sort

- *Lower bound theory* looks for finding a lower bound of the execution time for every algorithm aimed at solving a given problem. So a lower bound is problem specific, rather than algorithm specific.
- If you know a lower bound then, from the practical point of view, this means that you are definitely sure that you cannot do better than the lower bound on every algorithm for solving that problem.
- Consider the sorting problem using a comparison sort algorithm. Assume your array has n elements.
- You algorithm can be represented as a decision tree with $n!$ leaves. Assuming that this tree has height h, this means that your comparison algorithm will require in the worst case at least h comparisons.
- Also note that a decision tree is a binary tree, so $2^h \geq n!$.
- Using the Stirling asymptotic approximation of $n! \approx (n/e)^n$, we get $2^h \geq (n/e)^n$ which clearly shows that $h \geq n \lg n - n \lg e$. Therefore, the worst case execution time of every comparison sort algorithm has a lower bound of $n \lg n$, i.e. it is $\Omega(n \lg n)$.

Count sort - idea

- Let *a* be an array, let *b* be the array obtained after sorting *a* using a stable sorting algorithm (i.e. it preserves the relative order of equal elements in *a*). Then the position of an element $a[i] = x$ of *a* in *b* is equal to the number of elements of *a* that are strictly lower than *x* plus the number of elements in *a* that are equal to *x* and precede *x* in *a* plus 1.
- Example: let $a = [4, 5, 6, 3, 4, 8, 2,4]$. Let $a[5] = 4$ be an element of *a*. There are 2 elements lower than $a[5]$ in *a* ($a[4] = 3$ and $a[7] = 2$) and there is 1 element in *a* that precedes $a[5]$ and is equal to 4 ($a[1]$). So, a stable sorting algorithm will place $a[5]$ on position 2+1+1 = 4 in the resulted sorted array *b*.
- Counting sort computes the place of an element in the sorted array by counting the number of elements that are strictly lower than it and the number of elements that are equal to it and precede it.
- However, in the usual setting this approach would require $O(n^2)$ execution time. Homework: write the algorithm !
- However, we can do better, even than quick sort, but only by considering additional constraints on input *a* !

Distributions count sort – the idea

- Let us assume that the elements $a[i]$ are taken from the set of integers $\{1, 2, \ldots, k\text{-}1, k\}$. The idea of counting can be applied by thinking that the elements of a can be distributed in k arrays such that all the elements in array i are equal to i, for all $1 \leq i \leq k$. Actually, we only need to count how many elements are distributed to each array, and then apply the idea of sorting by counting.
- Counting distributions uses an array c of length k and can be done in $O(n)$ time using a single traversal of the original array a ($n = length[a]$). For every element $x = a[i]$ of a we only need to increment its number of occurrences $c[x]$ in a.
- Then, we count how many elements $c'[i]$ are in the array such that $c'[i] \leq i$. $c'[i] = \Sigma_{1 \leq j \leq i}\, c[j]$, for all $1 \leq i \leq k$. This can be done "in place" using a single traversal of array c in $O(k)$ time.
- Finally, the elements of a are transferred to their positions in the sorted array b using the information saved in c' (actually c because of "in place" computation). This can be done in time $O(n)$, using a single downwards traversal of a.

Distributions count sort – the algorithm

DISTRIBUTIONS-COUNT-SORT(A, B, k)

1. **for** $i \leftarrow 1,k$ **do**
2. $\quad C[i] \leftarrow 0$
3. $\triangleright$ Compute the number $C[i]$ of elements equal to i
4. **for** $j \leftarrow 1,length[A]$ **do**
5. $\quad C[A[j]] \leftarrow C[A[j]] + 1$
6. $\triangleright$ Compute the number $C[i]$ of elements less or equal than i
7. **for** $i \leftarrow 2,k$ **do**
8. $\quad C[i] \leftarrow C[i] + C[i-1]$
9. $\triangleright$ Transfer the elements to the sorted array
10. **for** $j \leftarrow length[A],1$ **do**
11. $\quad B[C[A[j]]] \leftarrow A[j]$
12. $\quad C[A[j]] \leftarrow C[A[j]] - 1$

- Initially: *A* = [4, 1, 4, 2, 4, 1, 5, 4].
- After steps 4 – 5: *C* = [2, 1, 0, 3, 1].
- After steps 7 – 8: C = [2, 3, 3, 6, 7].
- The execution time is O(*n*+*k*). When *k* is *O*(*n*) the execution time is O(*n*).

Brief review of pointers

- Pointer = a variable that contains the address of another variable.

```
int a;
int b[5];
/* p is a pointer to int. */
int *p;
p = &a;
/* q is a pointer to an array of int, */
int (*q)[5];
q = &b;
/* r is an array of pointers to int. */
int *r[5];
r[2] = &b[2];
/* t is a pointer to a pointer to int. */
int **t;
t = &p;
/* f is a function returning a pointer to char. */
char *f();
/* g is a pointer to a function returning a char. */
char (*g)();
```

ANSI C Functions for Sorting and Searching

- `qsort()` implements a quick-sort algorithm to sort "in place" an array of *num* elements, each of *width* bytes, starting at address *base*. *compare* is a pointer to a user-supplied routine that compares two array elements and returns a value specifying their relationship.
- **`void`** `qsort(`**`void`** `*`*`base`*`, size_t` *`num`*`, size_t` *`width`*`, `**`int`** `(*`*`compare`*`)(`**`const void`** `*`*`elem1`*`, `**`const void`** `*`*`elem2`*`));`
  - *base* = pointer to the start of target array
  - *num* = size of target array in number of elements
  - *width* = size of an element of the target array in number of bytes
  - *compare* = pointer to a function for comparing two array elements; if first argument is strictly less than the second argument returns –1 else if the first argument equals the second argument returns 0 else returns +1.
- `bsearch()` performs a binary search of a sorted array. *base*, *num*, *width*, *compare* have similar meanings with `qsort()`. *key* is the value being sought
- **`void`** `*bsearch(`**`const void`** `*`*`key`*`, `**`const void`** `*`*`base`*`, size_t` *`num`*`, size_t` *`width`*`, `**`int`** `(*`*`compare`*`)(`**`const void`** `*`*`elem1`*`, `**`const void`** `*`*`elem2`*`));`
  - *key* = pointer to the key to search for
  - *base*, *num*, *width*, *compare* = similar to `qsort()`
  - Return value: a pointer to an occurrence of *key* in the array pointed to by *base*. If *key* is not found, the function returns `NULL`.

Sorting integers and strings

- Sample input:

```
i
5
55
22
11
44
33
c
5
ee
bb
aa
dd
cc
x
```

- Sample output:

```
Array:
55 22 11 44 33
Sorted array:
11 22 33 44 55
Array:
ee
bb
aa
dd
cc
Sorted array:
aa
bb
cc
dd
ee
```

An Example (I)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Maximum number of array elements. */
#define NMAX 100

/* Maximum number of characters per string. */
#define LMAX 20

int compare_int(const void *pa,const void *pb);
int compare_str(const void *pa,const void *pb);
void read_int(int *pn,int x[]);
void read_str(int *pn,char *y[]);
void write_int(int n,int x[]);
void write_str(int n,char *y[]);

int x[NMAX+1];
char *y[NMAX+1];
int n;
char cmd[5];
```

An Example (II)

```
int compare_int(const void *pa,const void *pb) {
  int a = *((int*)pa);
  int b = *((int*)pb);

  if (a < b) {
    return -1;
  }
  else if (a == b) {
    return 0;
  }
  else {
    return 1;
  }
}

int compare_str(const void *pa,const void *pb) {
  char *a = *(char **)pa;
  char *b = *(char **)pb;

  return strcmp(a,b);
}
```

An Example (III)

```
void read_int(int *pn,int x[]) {
  int i;
  scanf("%d",pn);

  for (i=1;i<=*pn;i++) {
    scanf("%d",&x[i]);
  }
}

void read_str(int *pn,char *y[]) {
  char buf[LMAX+1];
  int i;
  scanf("%d",pn);

  for (i=1;i<=*pn;i++) {
    scanf("%s",buf);
    y[i] = (char *)calloc(strlen(buf)+1,sizeof(char));
    strcpy(y[i],buf);
  }
}
```

An Example (IV)

```
void write_int(int n,int x[]) {
  int i;
  for (i=1;i<=n;i++) {
    printf("%d ",x[i]);
  }
  printf("\n");
}

void write_str(int n,char *y[]) {
  int i;
  for (i=1;i<=n;i++) {
    printf("%s\n",y[i]);
  }
}
```

An Example (V)

```
int main() {
  char c;
  scanf("%s",cmd);
  c = tolower(cmd[0]);
  while (c != 'x') {
    if (c == 'i') {
      read_int(&n,x);
      printf("Array:\n");
      write_int(n,x);
      qsort((void *)(&x[1]),n,sizeof(int),&compare_int);
      printf("Sorted array:\n");
      write_int(n,x);
    }
    else if (c == 'c'){
      read_str(&n,y);
      printf("Array:\n");
      write_str(n,y);
      qsort((void *)(y+1),n,sizeof(char *),&compare_str);
      printf("Sorted array:\n");
      write_str(n,y);
    }
    else { printf("Unknown command !"); }
    scanf("%s",cmd);
    c = tolower(cmd[0]);
  }
  return 0;
}
```

Sorting a list in Python

- Python provides predefined functions for list sorting:
  - Function *sorted* that takes a list and produces a sorted list
  - Method *sort* of class *list* that sorts a list "*in place*"
- De exemplu:

```
>>> sorted([4,-1,20,3,5])
[-1, 3, 4, 5, 20]
>>> l = [4,-1,20,3,5]
>>> l.sort()
>>> l
[-1, 3, 4, 5, 20]
>>> l = [4,-1,20,3,5]
>>> l.sort(reverse=True)
>>> l
[20, 5, 4, 3, -1]
>>> from operator import itemgetter
>>> l = [('a',2),('b',-1),('c',3)]
>>> sorted(l,key=itemgetter(1),reverse=True)
[('c', 3), ('a', 2), ('b', -1)]
```

Import function *itemgetter* from module *operator*

Sort decreasingly according to the 2nd field. Fields are numbered increasingly starting with 0.

Reading and sorting a list in Python

- Let us consider a list of non-null natural numbers. The lists terminates with an integer number ≤ 0. We must read the list and print the sorted list.
- The list is input as an arbitrary sequence of numbers separated by spaces, possibly on different input lines. Let assume that the list contains at least one element.
- We need:
  - A function that reads an integer from the standard input
  - A main program (script) that:
    - Read numbers using a loop, until a number ≤ 0 is read
    - Sort the list and output the result
- Example of running the program:

```
C:\__users\Lucru\python>sortare.py
1 2 3 4
2 10 23 45 11

23
90

7 8 9
0
Number of elements: 14
Sorted list : [1, 2, 2, 3, 4, 7, 8, 9, 10, 11, 23, 23, 45, 90]
```

A script can be run directly from the command-line

Reading a list in Python

```
# Global variables used by getint() function
getint_i = 0
getint_n = 0
def getint():
    global getint_i,getint_n,getint_buf
    if getint_i < getint_n:
        getint_i += 1
        return getint_buf[getint_i]
    else:
        getint_buf = [int(s) for s in input().split()]
        while getint_buf == []:
            getint_buf = [int(s) for s in input().split()]
        getint_n = len(getint_buf)-1
        getint_i = 0;
        return getint_buf[getint_i]
```

One-line comment

Global variables

Read a line of numbers until the line is non-empty

- Input is read line by line.
- Each line is transformed into a list of numbers *getint_buf* with *getint_n*+1 elements with indices from 0 to *getint_n*.
- Variable *getint_i* represents the next element "unread" yet from *getint_buf*.

Python script for reading and sorting a list of numbers

```
my_list = []
n = 0

x = getint()
while x>0:
    n += 1
    my_list += [x]
    x = getint()
my_list.sort()
print('Number of elements:',n)
print('Sorted list:',my_list)
```

Appending a list to a given list

Problems

1. Let us consider a natural number $n \geq 2$. A generalized unit matrix is a $n \times n$ squared matrix with elements 0 and 1 such that on each row and each column there is a single 1. For example, A is a generalized unit matrix, while B is not. Design an algorithm to generate a random generalized unit matrix.

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

2. Professor Olay is a consultant for an oil company. The company must deploy a pipe system that crosses from east to west an oil field with n oil sinks. Each sink must be joined with a main pipe via a north-south junction pipe. The problem is to position the main pipe such that the total length of the junction pipes is minimized.

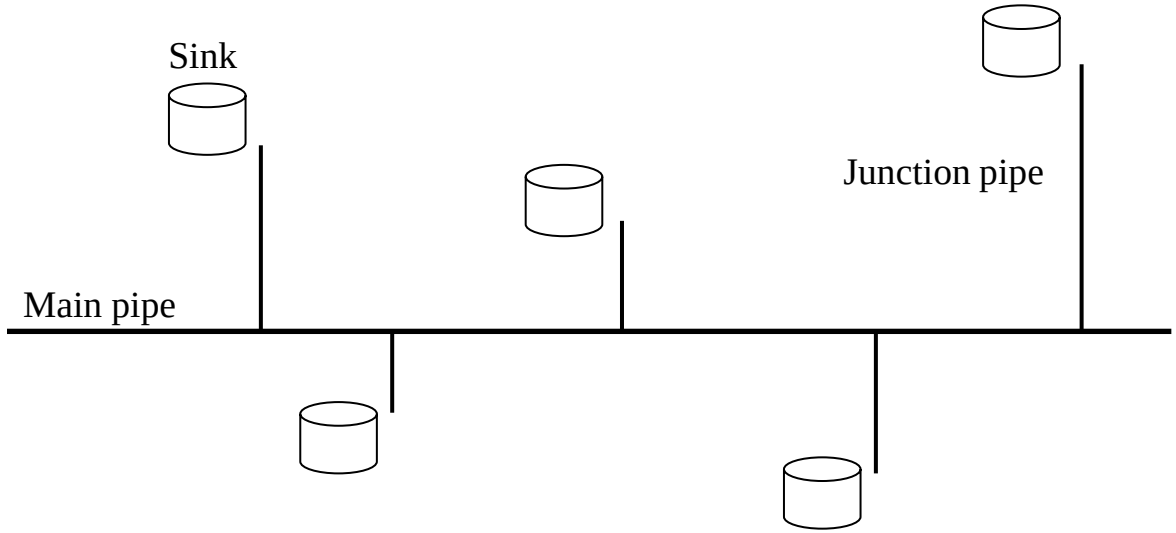
Problems (continuation)

a. Show how to position the main pipe for n=2 and n=3 sinks.

b. Show how to position the main pipe in the general case. Mathematically prove that your answer is correct.

c. Devise an efficient $O(n)$ algorithm for positioning the main pipe. Your algorithm takes the coordinates of the sinks and produces the coordinate of the main pipe.

d. Implement a C/Python program that computes the position of the main pipe for various input data sets that are read from standard input. A data set contains the value of n followed by the n values of the (vertical) coordinates of the sinks. The end of the input data sets is marked with a 0. For each input data set your program will compute and write to standard output the position of the main pipe.



Problems (continuation)

3. Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ in $O(1)$ time, where a and b are natural numbers between 0 and k. Your algorithm should use $\Theta(n + k)$ preprocessing time.
4. For n distinct elements $x_1, x_2, ..., x_n$ with positive weights $w_1, w_2, ..., w_n$ such that $\Sigma_{1 \le i \le n} w_i = 1$, the weighted (lower) median is the element x_k satisfying $\Sigma_{x_i < x_k} w_i < 1/2$ and $\Sigma_{x_i > x_k} w_i \ge 1/2$.
   a. Argue that the median of $x_1, x_2, ..., x_n$ is the weighted median of the x_i with $w_i = 1/n$ for all $1 \le i \le n$.
   b. Show how to compute the weighted median of n elements in $O(n \times \lg n)$ worst-case time using sorting
   c. Show how to compute the weighted median in Θ(n) worst-case time using a linear-time median algorithm

Problems (continuation)

5. The post-office location problem is defined as follows. We are given n points $p_1, p_2, ..., p_n$ with associated weights $w_1, w_2, ..., w_n$. We wish to find a point p (not necessarily one of the input points) that minimizes the sum $\Sigma_{1 \le i \le n} w_i \, d(p,p_i)$ where $d(a,b)$ is the distance between points a and b.

   a. Argue that the weighted median is a best solution for the 1-dimensional post-office location problem, in which points are simply real numbers and the distance between points a and b is $d(a,b) = |a - b|$.

   b. Find the best solution for the 2-dimensional post-office location problem, in which the points are (x, y) coordinate pairs and the distance between points $a = (x_1, y_1)$ and $b = (x_2, y_2)$ is the Manhattan distance given by $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$.