Sorting Algorithms

Introduction to non-recursive sorting algorithms

Cours Premier Cycle (IBIS), INSA de Rouen

Habib Abdulrab

Contents

- (I) Basic Definitions: relation, total relation, order relation, strict order relation, alphabetic relation…
- (II) What is Sorting? Why Sorting?
- (III) Notion of Algorithmic Complexity, How to evaluate the Algorithmic Complexity?
- (IV) Some Sorting algorithms, and analysis of their complexity:
  - Sorting by Selection.
  - Sorting by Exchanging.
  - Sorting by Insertion.

(I) Basic Definitions (1)

- What is a *relation* R on a set E?
  - R is a subset of $E \times E$
  - Ex: E = {1, 2, 3}, R = {(1, 1), (1, 3), (2, 1)}
  - If (x , y) belongs to R, we say: x is *in relation with* y. It is denoted by: x R y.
- What is a *total* relation?
  - For each x , y, $x \neq y$, we have either x R y, or y R x.

Basic Definitions (2)

- What is a strict *order relation* R on a set E?
  - R is anti-reflexive: each element x of E verifies: (x , x) $\notin$ R. i.e. x *has no relation with* x.
  - R is transitive: if x R y and y R z then x R z.
  - Examples:
    - The relation < on the set of natural numbers: N.
    - The strict inclusion relation $\subset$ on $\wp$(E): the set of all the subsets of E.
    - Remarks: < is total, and $\subset$ is not total.

Basic Definitions (3)

- What is an *order relation* R on a set E?
  - R is *reflexive*: each element x of E verifies: x R x
  - R is *anti-symmetric*: if x R y and y R x then x = y.
  - R is *transitive*: if x R y and y R z then x R z.
  - Examples:
    - The relation ≤ on the set of natural numbers: N,
    - Alphabetic (Lexicogaphic) order on words (see next slight for a precise definition)

Alphabetic order R on words

- Recall first:
  - An alphabet A is a set of *letters*. Ex: A = {a, b}.
  - We denote by A* is the set of *words* on A. The *concatenation* of words is denoted by ., and the *empty* word is denoted by: ε .
  - Ex: A* = {ε, a, b, aa, ab, ba, bb, aaa, aab, …}
  - abb . baa = abbbaa
  - A word u is a *prefix* of v if v = u . w, with w is a word.

How to define the Alphabetic order R on words?

- We need first to introduce a strict order < on A.
  - Ex a < b.
- R is defined as follows:
  - If u is a prefix of v then u R v
  - If u = w x w' and v = w y w'', with w, w', w'' are words and x, y are letters such that x < y, then u R v.
- Exercise: verify that R is a total order relation.

(II) What is Sorting?

- Problem of Sorting:

  Input: a *sequence* S: $s_1, \ldots, s_n$ *totally ordered* by a total order R; $n \geq 0$.

  Output: a *permutation* S': $s_{1'}, \ldots, s_{n'}$ of S such that $s_{1'} R s_{2'}, s_{2'} R s_{3'}, \ldots, s_{n-1'} R s_{n'}$.

Why Sorting?

- Imagine how hard it would be to use a dictionary if its words were not listed in alphabetic order!
- What is the difference between searching an item in a non sorted and sorted big sequence of items: (ex: 10^6 items) ?
  - Linear searching of an item in a non sorted sequence
  - Dichotomist searching of an item in a sorted list

(III) Notion of Algorithmic *Complexity* (1)

- How to evaluate the performance of an algorithm?
- Different algorithms have different execution cost in *time* (number of operations performed by the algorithm), and in *space (size of the necessary data structures for its execution)*. This is called the *complexity* in time and in space of the algorithm.

Notion of Algorithmic *Complexity* (2)

- The algorithmic complexity allows the qualification of the performance of the algorithm, and its comparison with others.
- It is fundamental to study the algorithmic complexity. This allows to determine if an algorithm a is better than b, if it is *optimal,* if it is not to be used…

How to evaluate the Algorithmic *Complexity?*

- We choose all the fundamental operations x performed by the algorithm: comparisons, exchanging, multiplications…
- According to the size of the input n, we compute the number of the executions of the operations x as a function of n: O(1), O(n), $O(n^2)$, $O(n^k)$, O(Log n), O(n Log n), $O(K^n)$…

Notion of Algorithmic *Complexity* (3)

- There are different types of complexities: in the worst case Max(n), in average (the most important) Average(n), in the best case.
- Algorithms with complexities like O(1), O(n), O(Log n), O(n Log n) are usually used.
- Algorithms with complexities like $O(n^k)$ depend on the size of k, and are usually for problems of small size.
- Algorithms with complexities like $O(K^n)$ are usually rejected.

Examples of the complexity of some Sorting Algorithms

- Comparison between the complexity of Fast Sorting (QuickSort, HeapSort), and Naive Sorting (Studied in this course)

Algorithm for sorting N items	Complexity in the worst case	Complexity in average
Naive algorithms: by exchanging, selection, insertion…	N^2	N^2
QuickSort	N^2	N Log N
HeapSort	N Log N	N Log N

Examples of the complexity of some Sorting Algorithms (continued)

- See a graphical simulation of different sorting algorithms complexities, in:

  http://java.sun.com/docs/books/tutorial/essential/threads/

Different types of Sorting Algorithms

- Recursive and non recursive sorting
  - Informal Examples
- Sorting based on *exchanging* of elements of the sequence S
  - How to do the exchange procedure?
- Sorting without Exchanging the elements of the sequence S
  - When to use these algorithms?

(VI) Algorithms: preliminary remarks

- In all the following algorithms the input S should be seen as the *abstract date type: list*. It can be implemented by an array or differently.
- The result S', in all these algorithms, is computed as a permutation of S without creating any new data structures.
- We will suppose, for the computation of the algorithmic complexity, that S is implemented by an array, which allows the direct access to the k^{th} item. That is why we use the notation S[k], which can be replaced by the abstract operation: **$i^{ème}$(S, k)**.

Sorting by Selection (1)

- Input: a sequence S: $s_1, \ldots, s_n$; $n \geq 0$.
- Output: a sorted sequence S': $s_1', \ldots, s_n'$

- Idea of the algorithm SortingBySelection(S)
  - Select the minimum $m = s_i$ of S, and consider $T = S \setminus \{s_i\}$
  - Put m at the beginning of S', followed by SortingBySelection(T).

Sorting by Selection (2)

- Idea of the Iterative algorithm:
- A loop, for i = 1 to n-1 in which we *select* the i^{th} minimum of S, and we put it at its final place S[i].
  - At each step i, we put in s_i the minimum of all the items from i to n, as follows:

```
J ← i        /* the current minimum */
For k ← i+1 to n do
    IF S[K] < S[J] THEN J ← K. /* The current minimum becomes S[k] */
S[j] ↔ S[i];
```

Sorting by Selection (3)

- Input: a sequence S: $s_1, \ldots, s_n$; $n \geq 0$.
- Output: a sorted sequence S': $s_1', \ldots, s_n'$

- SortingBySelection(S)

```
For i ← 1 to n-1 do
    j ← i
    For k = i+1 to n do
            IF S[K] < S[J] THEN J ← K.
    S[j] ↔ S[i]
```

Sorting by Selection (4)

- The Complexity here does not depend on S => Max (S) = Average (S)
- Number of Comparisons:
  - (n-1) + (n-2) + …. + 1 = n(n-1)/2 = $O(n^2)$.
- Number of Exchanging:
  - (n-1) = O(n).
- The Complexity in time of TriBySelection = $O(n^2)$.
- The Complexity in space of TriBySelection = O(1).

Sorting by Selection (5)

- Exercises:
  - Run un example.
  - Find some Improvements.
  - Implementation in C.

Sorting by Exchanging (*Le tri à bulles*)

- Input: a sequence S: s_1, … , s_n; n ≥ 0.
- Output: a sorted sequence S': s_1', … , s_n'

- The Idea of SortingByExchanging(S)
  - Select the minimum m of S by visiting all the items from the end to the beginning, and by exchanging each two consecutive items which are not ordered.

Sorting by Exchanging (2)

- Idea of the Iterative algorithm:
- A loop, for i = 1 to n-1 in which we put the i^{th} minimum at S[i].
  - At the step i, we put in S[i] the minimum of all the items from i to n, by exchanging each two consecutive items which are not ordered, as follows:

For k = n downto i+1 do

IF S[K] < S[K-1] THEN S[K] ↔ S[K-1]

Sorting by Exchanging (3)

- Input: a sequence S: $s_1, \ldots, s_n$; $n \geq 0$.
- Output: Sorted sequence S': $s_1', \ldots, s_n'$

- SortingByExchanging(S)

```
For I =1 to n-1 do
    For k = n downto i+1 do
        IF S[K] < S[K-1] THEN S[K] ↔ S[K-1]
```

Sorting by Exchanging (4)

- Number of Comparisons (in average and in the worst case):
  - $(n-1) + (n-2) + \ldots + 1 = n(n-1)/2 = O(n^2)$.
- Number of Exchanging (in the worst case):
  - $(n-1) + (n-2) + \ldots + 1 = n(n-1)/2 = O(n^2)$.
- Number of Exchanging (in average): = $O(n^2)$. (Here the calculus is more complicated)
- The Complexity in time of TriByExchanging = $O(n^2)$.
- The Complexity in space of TriByExchanging = $O(1)$.

Sorting by Exchanging (5)

- Exercises:
  - Run un example.
  - Find some Improvements.
  - Implementation in C.

Sorting by Insertion

- Input: a sequence S: s_1, … , s_n. n ≥ 0.
- Output: a sorted sequence S': s_1', … , s_n'

- The Idea of SortingByInsertion(S)
  - Suppose that the first i-1 first items are already sorted
  - Insert s_i at its place among them.

Sorting by Insertion (2)

- Idea of the Iterative algorithm
  - A loop, for i = 2 to n, in which S[i] = x is the item to insert.
  - To insert x among the sorted items S[1],…,S[i-1], we transfer S[k] to S[k+1], while S[k] > x, for K = i-1, i-2, …
  - To stop this loop, we need to put in S[0] a special item smaller than all the items of S.
  - When S[k] ≤ x, we insert x at S[k].

Sorting by Insertion (3)

- Input: a sequence S: $s_1, \ldots, s_n$.
- Output: Sorted sequence S': $s_1', \ldots, s_n$

- SortingByInsertion(S) /* $s_0, s_1, \ldots, s_n$. */

```
For i=2 to n do
    k ← i-1, x ← S[i],
    while S[k] > x do
                S[k+1] ← S[k], k ← k-1,
    S[k+1] ← x
```

Sorting by Insertion (4)

- Number of Comparisons (in the worst case):
  - $(n-1) + (n-2) + \ldots + 1 = n(n-1)/2 = O(n^2)$.
- Number of Comparisons (in average): = $O(n^2)$. (Here the calculus is more complicated)
- Number of Transfers at each step (in average and in the worst case): Number of Comparisons + 1. Thus, the number of Transfers = $O(n^2)$.
- The Complexity in time of TriByInsertion = $O(n^2)$.
- The Complexity in space of TriByInsertion = $O(1)$.

Sorting by Insertion (5)

- Exercises:
  - Run un example.
  - Find some Improvements.
  - Implementation in C.

References

1) Structures de Données et Algorithmes, *A. Aho, J. Hopcroft, J. Ullman*. InterEditions.

2) Types de Données et Algorithmes, *C. Froidevaux, M-C. Gaudel, M. Soria*. Editions McGRAW-Hill.