Alberto Pettorossi

Computer Science Department
University of Roma Tor Vergata

Theory of Computation

Part IV

Second Edition

1. Complexity Analysis of Sorting and Searching
2. Recursion and Backtracking
3. Computational Complexity and NP-complete Problems
4. Multiplication of Integers, Polynomials, and Matrices. Fast Fourier Transform.

ARACNE

00173 Roma, via R. Garofalo, 133 A–B
tel. (39) 06 72672233 telefax (39) 06 72672222

www.aracne–editrice.it
e–mail: info@aracne–editrice.it

ISBN: 88–7999–114–0

I edizione: dicembre 1994
II edizione: ottobre 2002

PREFACE

These lecture notes introduce the reader to some basic notions of Analysis of Algorithms and Complexity Theory. They are based on the references listed at the end of each chapter. We also derived some material from the notes of a course given by L. Valiant at Edinburgh, Scotland, in 1979.

In the first chapter we look at sorting and searching algorithms and we study their complexity. The second chapter is devoted to the backtracking technique for exploring search spaces of various kinds. In the third chapter we present some preliminary notions of computational complexity, including NP and NP-complete problems. Finally, in the last chapter we consider the digital Fourier Transform and the problem of fast integer, polynomial, and matrix multiplication.

I would like to thank my students, my colleagues at the Computer Science Department of the University of Roma Tor Vergata, and in particular, Mauro Gaspari and Maurizio Proietti at IASI-CNR, Roma.

Alberto Pettorossi
October 2002.

CHAPTER 1.

COMPLEXITY ANALYSIS OF SORTING AND SEARCHING

1. BASIC DEFINITIONS OF COMPLEXITY MEASURES
2. SOME SORTING ALGORITHMS
3. BINARY SEARCH
4. LINEAR SELECTION
5. MINIMAL SPANNING TREE

1. COMPLEXITY ANALYSIS OF SORTING AND SEARCHING.

CONTENTS

1. BASIC DEFINITIONS ON COMPLEXITY MEASURES.

Let us assume that we have a notion of 'size of the input' to an algorithm and a notion of 'step of execution' of an algorithm.

DEFINITION 1. The time complexity of an algorithm A on an input x is the number of steps taken during the execution of A for the input x. The *worst case time complexity* of an algorithm is the function $\lambda m.T(m)$ iff for all m, T(m) is the least number such that for all inputs of size m the algorithm stops after at most T(m) steps.

By abuse of language, the function $\lambda m.T(m)$ will also be denoted by T(m).

Given an algorithm A and an input x, there is no guarantee that one can determine the time complexity of the algorithm A for the input x simply because the halting problem is undecidable. Often, however, one can perform the time complexity analysis because it is assumed that the algorithm under consideration, does terminate.

DEFINITION 2. The *expected time complexity* of an algorithm A for the input distribution D (i.e., input x occurs with probability $D_m(x)$ where m is the size of x) is $\lambda m.T(m)$ iff for all m

$$T(m) = \Sigma_{\text{all inputs x of size m}} D_m(x) \times (\text{time complexity of the algorithm A on the input x}).$$

D is an infinite number of distribuitions $D_1, D_2, \ldots, D_m, \ldots$, one for each size of the input.

If each D_i (for $1 \leq i$) is *constant* for all inputs of size i, that is, all inputs of size i have the same probability, then the expected time complexity is also called the *average time complexity*.

Often we do not exhibit the explicit form of the function T(m). Instead, we only say that T(m) is in O(V(m)), or $\Omega(V(m))$, or $\Theta(V(m))$ for some function V(m) [Pettorossi 91, Chapter 7].

Let us also introduce the following definitions which will be useful below. They refer to

the notions of 'problem' and 'algorithm for solving a problem' which we may leave informal for our purposes here (see also [Pettorossi 94,Chapter 4]).

DEFINITION 3. The *worst case time complexity* of a problem is the worst case time complexity of the algorithm for it that, among all algorithms which solve the problem, has minimal worst case time complexity (w.r.t. a given notion of minimality).

DEFINITION 4. The *expected time complexity* of a problem is the expected time complexity of the algorithm for it that, among all algorithms which solve the problem, has minimal expected time complexity (w.r.t. a given notion of minimality).

In order to design optimal algorithms, one looks for *upper bounds* and *lower bounds* of problems.

To show that U(n) is an upper bound on the (worst case or expected) time-complexity of a problem, it suffices to present an algorithm that solves the problem and it has (worst case or expected) time-complexity $f(n) = O(U(n))$.

To show that L(n) is a lower bound on the (worst case or expected) time-complexity of a problem, one need to prove that every algorithm which solves the problem has (worst case or expected) time-complexity $f(n) = \Omega(L(n))$.

Instead of time complexity, one may also consider space complexity, and thus, one may introduce definitions similar to the above ones w.r.t. space complexity, instead of time complexity. For other introductory notions see [Pettorossi 91, Chapter 7].

2. SOME SORTING ALGORITHMS.

2.1 SELECTION SORT.

---

procedure SelectSort(n elements)
  if n=1 **then** return **else**
  begin find the maximum of n elements; SelectSort(remaining n–1 elements) **end**

Worst case complexity analysis (counting the number of binary comparisons):

$SS(n) = n-1 + SS(n-1)$

$SS(1) = 0$.

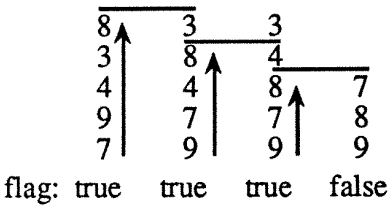
$SS(n)$ is $O(n^2)$ time complexity.

An iterative variant of the Selection Sort is: BUBBLE SORT. Here is a Pascal program for it, where the 'bubble' is the minimum element (not the maximum).

```
program bubble_sort;
const N=14;   var k,j,t : integer;   var flag : boolean;   var A : array [1..N] of integer;

begin   (* --- this is a simple way of producing an array to be sorted --- *)
for j:=1 to N do A[j]:=2*j;   writeln('Given array:'); for j:=1 to N do write(A[j]:3); writeln;

k:=N;
flag:=true; (* --- flag=true : at least one more pass through the array is needed --- *)
```

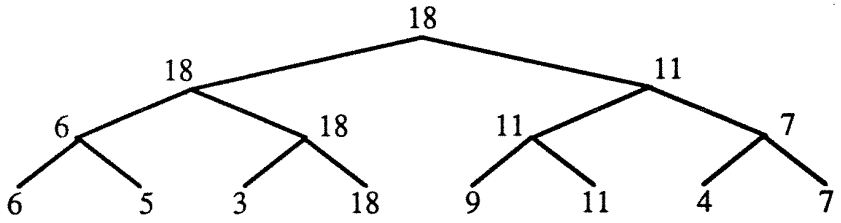
```
while flag do begin k:=k-1; flag:=false;
            for j:= 1 to k do if A[j] < A[j+1] then
                              begin t:=A[j]; A[j]:=A[j+1]; A[j+1]:=t; flag:=true
                              (* --- the bubble goes towards j=N --- *)
                              end
            end;
writeln('Sorted array:');   for j:=1 to N do write(A[j]:3);
end.

(* Given array:
      2 4 6 8 10 12 14 16 18 20 22 24 26 28
   Sorted array:
     28 26 24 22 20 18 16 14 12 10 8 6 4 2  *)
```

The complexity of Bubble Sort is $\Theta(n^2)$. Here is an example of the behaviour of Bubble sort on the numbers: 7, 9, 4, 3, 8.

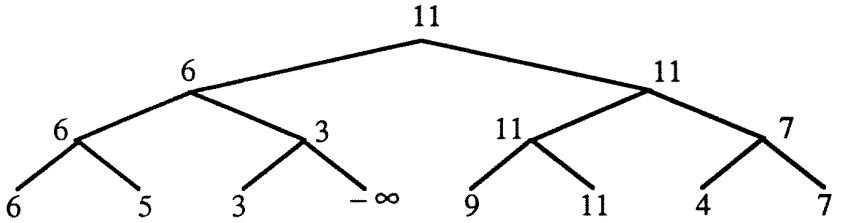


2.2 TREE SELECTION.

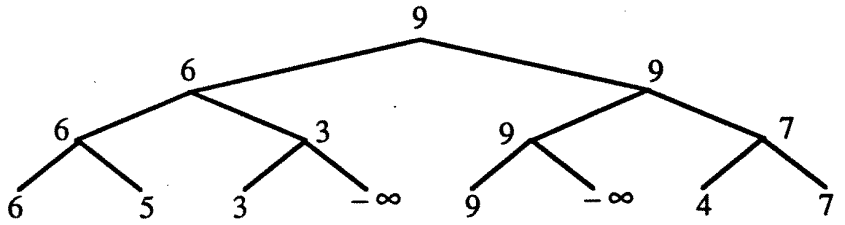
It is an improved version of the Selection Sort. At each selection the information gained in the previous selections is used.
Initialization: construct a tree as balanced as possible (that is, the root-to-leaf paths are all of length k or k+1, for some k): e.g.



At the i-th step (i = 1, 2, ..., n–1) replace the largest element found in the previous step by $-\infty$ and repair the tree. For instance:
1st step:



2nd step:



etc.

Worst case complexity analysis (counting the number of binary comparisons).
Suppose $n = 2^r$. Then the depth of the tree is $r = \log_2 n$. At each repair we perform no more that $\log_2 n$ comparisons. Initially we perform n–1 binary comparisons.

The total complexity is: $(n-1)\log_2 n + n - 1$.

If n is not an exact power of 2 then we pad with extra $-\infty$'s. The resulting complexity will be: $(n-1)\lceil \log_2 n \rceil + n - 1$.

An improved version of the Tree Selection is the Heap Sort (see pages added below).

2.3 BINARY INSERTION SORT.

As a subroutine we need to insert an element into an ordered list of n elements.

---

procedure Insert (into an ordered list of size 2^r-1);
begin compare with the middle element;
Insert (into the appropriate half of size $2^{r-1}-1$) **end.**

Worst case complexity analysis (counting the number of binary comparisons):

$IS(2^r-1) = 1 + IS(2^{r-1}-1)$

$IS(1) = 1.$

Thus, $IS(2^r-1) = r$.

If we have to insert an element into a list of size n, for any n, we can pad the list with $-\infty$'s so that it becomes of length 2^r-1, where $r = \lceil \log_2 (n+1) \rceil$.
Thus, $IS(n) = \lceil \log_2 (n+1) \rceil$ for any n.

BIS(n) = **if** n=1 **then** return **else**
begin BIS(n-1);
insert the last n-th element into the sorted list of the first n-1 elements
end.

Worst case complexity analysis (counting the number of binary comparisons) [Pettorossi 91, page 197]:

$B(n) = B(n-1) + \lceil \log_2 n \rceil$

$B(1)=0.$

Thus, $B(n) = \Sigma_{k=1,\ldots,n} \lceil \log_2 k \rceil$. We have: $B(n) < n\lceil \log_2 n \rceil$.
The exact value (as it can be verified by induction) is:

$$B(n) = n\lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1.$$

If $n = 2^r$ then $B(n) = n \log_2 n - n + 1$.

2.4 MERGE SORT.

We need a subroutine which takes as input two sorted lists: $a_1,\ldots, a_n$ and $b_1,\ldots, b_m$ and merges them together into a sorted list: $c_1,\ldots, c_{n+m}$.

---

procedure merge(a,b,c);
i := j := k := 1;
loop: **if** $a_i > b_j$ **then begin** $c_k := a_i$; i := i+1; k := k+1 **end**
else begin $c_k := b_j$; j := j+1; k := k+1 **end**
if i≤n and j≤m **then** goto loop **else** 'complete' list c.

Worst case complexity analysis for merge (counting the number of binary comparisons): m+n–1.

---

procedure merge_sort(list of n elements);
begin if n = 1 **then** return **else**
begin mergesort(1st half of list); mergesort(2nd half of list); merge two sorted halves
end
end.

Worst case complexity analysis (counting the number of binary comparisons).

Assume that n is a power of 2.

$$MS(n) = 2\,MS(n/2) + n - 1$$
$$MS(1) = 0.$$

Thus, $MS(n) = n \log_2 n - n + 1$.

If n is not a power of 2 we have:

$$MS(n) = MS(\lfloor n/2 \rfloor) + MS(\lceil n/2 \rceil) + n - 1$$
$$MS(1) = 0,$$

where $\lfloor m \rfloor$ is the largest integer ≤ m and $\lceil m \rceil$ is the smallest integer ≥ m. Thus,

$$MS(n) = n\lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1$$ (as for the Binary Insertion Sort).

Below is an implementation of Merge Sort in Pascal [Pettorossi 93]. The procedure merge works as follows.

Given the sorted portion of the array X (between the indices ax and bx inclusive) and the sorted portion of the array Y (between the indices ay and by inclusive), merge constructs the sorted portion of the array Z (between the indices az and bz inclusive) by taking elements from the given portions of X and Y such that the elements of those portions which are not taken, are all greater than the ones which are taken [Pettorossi 91, page 190].

```
type integerarray = array [1..d] of integer;

procedure merge(ax,bx,ay,by,az,bz : integer;  X,Y : integerarray;  var Z : integerarray);
var kx, ky, kz : integer; b : boolean;
begin kx:=ax; ky:=ay; kz:=az;
      while kz<=bz do
      begin  if ky>by  then b:=true else
             if kx>bx  then b:=false else b:= X[kx] <= Y[ky];
             if b then begin Z[kz]:=X[kx]; kx:=kx+1; kz:=kz+1 end
                  else begin Z[kz]:=Y[ky]; ky:=ky+1; kz:=kz+1 end
      end
end

procedure merge_sort(az,bz : integer; var Z : integerarray);
var m1,m2 : integer;  var X,Y : integerarray;
begin if (bz–az)>0 then
      begin m1 := (bz–az+1) div 2;   m2 := (bz–az+1)–m1;
      for i := 1 to m1 do X[i] := Z[az–1+i];  for i := 1 to m2 do Y[i] := Z[az–1+m1+i];
      mergesort(1,m1,X);   mergesort(1,m2,Y);   merge(1,m1,1,m2,az,bz,X,Y,Z)
      end
end.
```

2.5 QUICKSORT. (see pages added below)

2.6 HEAPSORT. (see pages added below)

2.7 EXPECTED TIME FOR SORTING.

In [Pettorossi 91, page 199] we have seen that the best algorithm for sorting n distinct elements, based on binary comparisons, has worst case complexity (counting the number of binary comparisons)

$S(n) \geq \lceil \log_2 (n!) \rceil = \lceil n \log_2 n - n \log_2 e + (\log_2 n)/2 + O(1) \rceil > n \log_2 n - 1.443\ n.$

We have obtained this result by viewing any sorting algorithm as a tree of binary comparisons, also called *binary decision tree.* A binary decision tree is a tree with nodes with 2 sons (called internal nodes) and nodes with 0 sons (called leaves). The internal nodes are labeled by binary comparisons, and the leaves are labeled by total orders.

To find the solution out of the n! possible ones for any input of size n, the best algorithm, that is, the best binary decision tree, must make for *some* input of size n, a sequence of binary decisions of length at least $\lceil \log_2 (n!) \rceil$. (See also Fig. 1, for inputs of size 3.)

This is due to the fact that any binary decision tree T for n elements (and thus, also the best one) has at least as many leaves as the n! permutations of the elements, and it has a path of length not smaller than $\lceil \log_2 (n!) \rceil$.

Thus, S(n) is, by definition, the worst case complexity (counting the number of binary comparisons) of the sorting problem.

We also have an upper bound for S(n). Indeed, we have that:

$S(n) \leq n\lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1,$

which is the worst case complexity of Binary Insertion Sort.

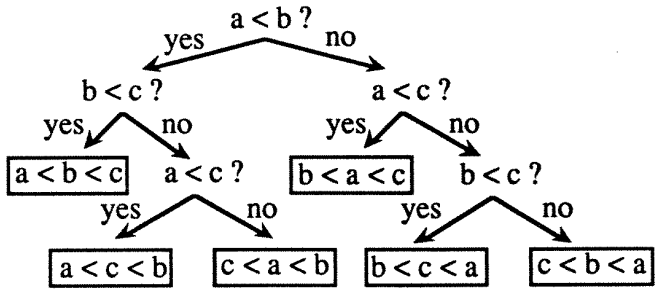


Figure 1. A decision tree for sorting the 3 distinct elements: a, b , and c.

Other sorting algorithms exist (for instance, Ford-Johnson Algorithm) which allow us to derive more accurate upper bounds for S(n), but we will not present them here.

We will now show that, as the worst case complexity S(n), also the average complexity (counting the number of binary comparisons) $S_A(n)$ of the best sorting algorithm for inputs of size n, is not smaller than $n \log_2 n$.

Also for this complexity analysis we will consider that a sorting algorithm is the same as a binary decision tree. When referring to a decision tree tout court, we will assume that it is binary.

We will also assume that the n elements to be sorted are all distinct. In case they are not, the reader may refer to [Knuth 75].

Note that for the worst case complexity for sorting 3 elements is $S(3) = \lceil \log_2 (3!) \rceil = 3$, while the average complexity of the algorithm of Fig. 1 is $(2+3+3+2+3+3)/6 = 8/3 < 3$ (consider the length of all root-to-leaf paths, and their number).

Can we expect that the average complexity $S_A(n)$ of the sorting algorithm which is the best w.r.t. the average complexity, is asymptotically better than the worst case complexity S(n) of the sorting algorithm which is the best w.r.t. the worst case complexity?

The answer is 'no', because, as we now show, the average complexity (counting the number of binary comparisons) $S_A(n)$ of the sorting algorithm which is the best w.r.t. the average complexity, for inputs of size n, is not smaller than $n \log_2 n$, which is a function of the same order of $S(n) = \lceil n \log_2 n \rceil$.

Let $L(T_m)$ be the sum of the lengths of all root-to-leaf paths in a decision tree T_m with m leaves (with $m \geq 1$), that is, $L(T_m) = \Sigma_{p \in \text{root-to-leaf paths}}$ length(p).

$L(T_m)$ is computed by assuming that if two root-to leaf paths p and q have a common portion from the root to a given node, then the length of that portion should be considered twice, once for the length of p and once for the length of q.

Let MinL(m) be the smallest value of $L(T_m)$ for all decision trees T_m's with m leaves.

THEOREM 1. $MinL(m) \geq m \log_2 m$, for all $m \geq 1$.
PROOF. By complete induction on m. For m=1 it is obvious.
Assume that $MinL(m) \geq m \log_2 m$, for all $m<k$. We need to prove that $MinL(k) \geq k \log_2 k$.

Any decision tree T_k with k leaves, is made out of a first subtree T_i with i leaves and a second subtree T_{k-i} with k–i leaves for $0<i<k$. Thus, for all shapes of T_k, we have:

$$L(T_k) = i + L(T_i) + (k-i) + L(T_{k-i}) = k + L(T_i) + L(T_{k-i}).$$

Thus, $MinL(k) = MIN_{0<i<k} [k + L(i) + L(k-i)] = k + MIN_{0<i<k} [L(i) + L(k-i)]$. By inductive hypothesis, we have:

$$MinL(k) \geq k + MIN_{0<i<k} [i \log_2 i + (k-i) \log_2 (k-i)].$$

The derivative of $[i \log_2 i + (k-i) \log_2 (k-i)]$ w.r.t. i is 0 for i=k/2. This derivative is negative for i<k/2 and it is positive for i>k/2.
Thus, we get the value of $MIN_{0<i<k} [i \log_2 i + (k-i) \log_2 (k-i)]$ for i=k/2. It is: $k \log_2 (k/2)$.
Thus,

$$MinL(k) \geq k + k \log_2 (k/2) = k \log_2 k.$$ ■

Let the *expected depth* of a tree T_m with m leaves be:

$$D_E(T_m) = \Sigma_{p \in \text{root-to-leaf paths}} \text{probability}(p) \times \text{length}(p).$$

If the probability of each path is the constant value r then

$$D_E(T_m) = r \times \Sigma_{p \in \text{root-to-leaf paths}} \text{length}(p).$$

THEOREM 2. If for all $n \geq 1$ each of the n! permutations of a given sequence of n elements to be sorted has probability 1/(n!), then the minimum value of $D_E(T)$ over all binary decision trees T for sorting n elements, is not smaller than $\log_2 (n!)$.
PROOF. Any decision tree T (or sorting algorithm) for sorting n elements has at least n! leaves. It may have more than n! leaves, but their labels are all taken from the set of the n! possible total orders of the n elements.

Since the decision tree is modeling a deterministic algorithm, for each set of root-to-leaf paths in T with identical total order as leaf label, there is only one path with probability 1/(n!), while all other paths in the set have probability 0.

The presence of root-to-leaf paths with probability 0 in a binary decision tree T cannot determine a decrease of $D_E(T)$.

Indeed, given any decision tree T for sorting n elements, if T has a subtree with all its leaves with probability 0, we can transform T into a new decision tree, say newT, as follows:

> "replace any subtree N which has a son-subtree Z with every leaf with probability 0 (or Z is a leaf with probability 0) by the other son-subtree of N" (see Fig. 2).

Obviously, after performing the transformation depicted in Fig. 2, we get the tree newT which still sorts n elements, simply because there is no input for T such that the arc leading from N to Z is taken (otherwise at least one leaf on or below Z has probability different from 0). In the derived tree newT the sum of the lengths of the paths is decreased w.r.t. T, and therefore, $D_E(newT) \leq D_E(T)$.

Thus, when looking for the minimum value of $D_E(T)$ over all binary decision trees T for sorting n elements, we may restrict ourselves to consider the minimum over the binary

decision trees with exactly n! leaves, each leaf being labeled by one of the n! distinct total orders, and having probability 1/(n!).

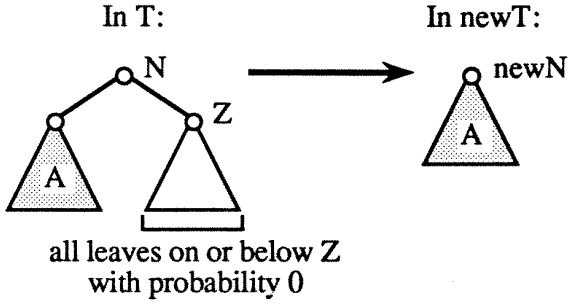


Figure 2. Getting from T a new decision tree newT with possibly smaller D_E.

Thus, we have:

$D_E(T_{n!}) = (1/(n!)) \times \Sigma_{\ p \in \text{ root-to-leaf paths}}\ \text{length}(p)$,

where the number of root-to-leaf paths is n!.

By Theorem 1, for every tree $T_{n!}$ with n! leaves we have:

$D_E(T_{n!}) \geq (1/(n!)) \times \text{MinL}(n!) = (1/(n!)) \times (n!)\ (\log_2 (n!)) = \log_2 (n!)$.

Recall that, by the Stirling formula, $\log_2 (n!) \approx n\ (\log_2 n) - 1.443\ n + O(\log_2 n)$. ■

Another way of stating the result of Theorem 2 is to say that the best sorting algorithm which sorts a sequence of n elements, has expected time complexity (when counting the number of binary comparisons) of at least $\log_2 (n!)$, when all n! permutations of the n elements are equally likely.

In other words, we get, as desired, that the average complexity (counting the number of binary comparisons) $S_A(n)$ of the sorting algorithm which is the best w.r.t. the average complexity, for inputs of size n, is not smaller than $\log_2 (n!)$.

2.8 INFORMATION THEORETIC LOWER BOUNDS.

The result which tells us that the worst case complexity (counting the number of binary comparisons) S(n) of the best sorting algorithm for inputs of size n, is at least $\log_2 (n!)$, can be generalized to every class of algorithms which can be viewed as binary decision trees.

Thus, we have that the worst case complexity (counting the number of the binary comparisons) of the best binary decision tree for inputs of size n, is at least the logarithm (in base 2) of the number of different possible outputs, out of which the binary decision tree determines the desired one.

Lower bounds which are obtained in this way, that is, by counting the number of binary comparisons needed for determining the required answer, are called 'information theoretic lower bounds'.

Example. Inserting a new element into a list of size n: since the new element can go into n+1 places, the worst case complexity (counting the number of binary comparisons) is at least $\log_2 (n+1)$. This proves that the Binary Insertion Sort is asymptotically optimal. ■

Example. Finding the maximum of n elements: since there are n possible outcomes the

information theoretic lower bound (counting the number of binary comparisons) is $\log_2 n$. This is, however, a very weak lower bound, in the sense that we know that the lower bound is n–1 (because all elements should be compared). Thus, the information theoretic lower bound may be a lower bound which is not tight. ■

Exercise. Find the worst case complexity of the Binary Insertion Sort and the Merge Sort when n=8 (8×3 – 8 + 1 = 17).
What is the information theoretic lower bound of sorting 8 elements? ($\lceil \log_2 (8!) \rceil = 16$). ■

2.9 EXTERNAL SORTING.

(see pages added below)

2.10 RADIX SORTING.

If the elements to be sorted are integers or strings over a finite alphabet, we can make use of the internal properties of the elements to be sorted. It this case sorting can be performed in less than $n \log_2 n$ time [Aho-Hopcroft-Ullman 75, page 77].

Let us suppose, in fact, that we have to sort a sequence of n (not necessarily distinct) elements, taken from the set {0, ..., m–1}. We can sort it as follows.

Consider m queues, each corresponding to a unique i, for $0 \le i \le m-1$. Initially they are all empty. Scan the input sequence and place every element in the queue corresponding to its value. This takes O(n) steps. At the end we concatenate the queues according to the corresponding i. This takes O(m) steps. If m is O(n) then sorting can be done in linear time. (This particular algorithm is called BUCKET SORT.)

3. BINARY SEARCH.

Searching for an element in a sorted array from the index a to the index b, by dividing into two halves.

```
function BinarySearch(k:key; a,b:integer): integer;
var m : integer;
begin if a>b then writeln('inverted limits') else
      if a=b then if k=A[a] then BinarySearch := a else writeln('key not found')
      else  begin m:=(a+b) div 2;
            if k<=A[m] then BinarySearch := BinarySearch(k,a,m)
            else BinarySearch := BinarySearch(k,m+1,b)
            end
end.
```

Worst case complexity analysis (counting the number of binary comparisons).
Assume that n is a power of 2.

$BS(n) \le BS(n/2) + 3$
$BS(1) = 3.$

Thus, $BS(n) = O(\log_2 n)$ [Pettorossi 91, page 193].

4. LINEAR SELECTION.

(see pages added below)

5. MINIMAL SPANNING TREE.

Given an undirected weighted graph we look for one of its minimal spanning trees. This problem, also called 'shortest spanning tree', has been considered in [Pettorossi 91, Chapter 4] and we have already seen there how to solve it by using Kruskal's algorithm (whose time complexity is $O(n^2 \log_2 n)$). This algorithm is *greedy*, in the sense that it finds the 'global optimum' by choosing at each step the 'local optimum'.

There is also a greedy algorithm by E. W. Dijkstra (whose time complexity is $O(n^2)$) which we now present. (See the following pages.)

REFERENCES

[Aho-Hopcroft-Ullman 75] Aho, A. V., Hopcroft, J. E., and Ullman, J. D.: "The Design and Analysis of Computer Algorithms" Addison-Wesley, 1975.

[Hopcroft-Ullman 79] Hopcroft, J. E., and Ullman, J. D.: "Introduction to Automata Theory, Languages, and Computation" Addison-Wesley, 1979.

[Knuth 75] Knuth, D. E.: "The Art of Computer Programming. Sorting and Searching" Vol. 3, Addison-Wesley, 1975.

[Pettorossi 91] Pettorossi, A.: "Quaderni di Informatica. Parte I", Unitor, Roma, 1991.

[Pettorossi 94] Pettorossi, A.: "Theory of Computation. Part III", Aracne, Roma, 1994.

[Pettorossi 93] Pettorossi, A.: "Learning Pascal through Exercises", Aracne, Roma, 1993.