ALGORITHMS AND COMPLEXITY THEORY

Chapter 3: Sorting Algorithms

Jules-R Tapamo

COMPUTER SCIENCE- DURBAN - FEBRUARY 2010

Contents

1 Introduction

More often programming problems include sorting procedures. We will in this part of the course, study sorting algorithms from the simplest to the more sophisticated ones.

2 Selection Sort

2.1 Principles

We want to sort n elements store in an array. Simple sorting algorithms are those which start by looking within the array, the smallest element, and then swap it with the one in the first position, then find the element with the second smallest value and swap it with the element in the second position in the array, and then continue until the $(n-1)^{th}$ element is processed. We will then obtain a sorted array. This method is called *selection sort*.

EXAMPLE 2.1.1

W e want to sort the following array in ascending order:

L	I	C	E	N	C	E

We will have the following sequence of execution

L	I	C	E	N	C	E	gives	**C**	I	L	E	N	C	E
C	I	L	E	N	C	E	gives	**C**	**C**	L	E	N	I	E
C	C	L	E	N	I	E	gives	**C**	**C**	**E**	L	N	I	E
C	C	E	L	N	I	E	gives	**C**	**C**	**E**	**E**	N	I	L
C	C	E	E	N	I	L	gives	**C**	**C**	**E**	**E**	**I**	N	L
C	C	E	E	I	N	L	gives	**C**	**C**	**E**	**E**	**I**	**L**	**N**

In this version of selection sort algorithm, to search the smallest element of the array to be sorted , we will compare elements between them and will only record the index of the smallest element, and when this element is found, swap it using its index. The pseudo-code of the algorithm is :

```
SelectionSort(int[] t, int n)
begin
int i,j,min,q
for i = 1 to n-1 do
    min = i
    for j = i+1 TO n  do
        if  t[j] < t[min] then
            min = j
        end if
    end for
    q = t[min]
    t[min]= t[i]
    t[i] = q
end for
```

The maximum running time of this algorithm is $O(n^2)$. Indeed for a given *i n -i* comparisons(c) are done, at least four assignments (a) and at most five are done, and one addition(o) is done. In total we have

$$(n-1)+(n-2)+(n-3)+...+1=\frac{n(n-1)}{2} \text{ comparisons,}$$

at least $4(n-1)$ assignments, at most $5(n-1)$ additions, thus the maximum running time is T_{max}(n) =$\frac{n(n-1)}{2}$ c+5(n-1)a+no = $\frac{n^2}{2}$c+$\frac{(-c+10a+2o)n}{2}$ $-5a=\Theta(n^2)$

3 Bubble Sort

3.1 Principles

For the bubble sort we think recursively. Given an array of n elements, place the biggest element at the end. And then apply recursively the algorithm to the $n-1$ first elements, if $n>1$. The greatest element is remounted at the end of the array as follows : assuming that the element with index k is the greatest of the k first elements of the array, the property to hold for k+1 we have :

- if the $(k+1)^{th}$ element is the greatest then there is nothing to do.
- else swap k^{th} and $(k+1)^{th}$ elements.

EXAMPLE 3.1.1

W e want to sort the list

C	K	G	L	B

step	array	operation	Resulting array
1rst	**C** K G L B	No change	C K G L B
	C **K** G L B	K > G , swap K and G	C G K L B
	C G **K** L B	No change	C G K L B
	C G K **L** B	L > B , swap L and B	C G K B L
2nd	**C** G K B L	No change	C G K B L
	C **G** K B L	No Change	C G K B L
	C G **K** B L	K > B , swap K and B	C G B K L
3rd	**C** G B K L	No change	C G B K L
	C **G** B K L	G > B , swap G and B	C B G K L
4th	**C** B G K L	C > B , swap C and B	B C G K L
	B C G K L	end	

The name bubble sort is given to this strategy of sort because if we visualize the process of sorting , the greatest element remounts towards the end of the array like air bubbles that remount at the water surface.

3.2 Algorithm

The pseudo-code of the algorithm can be specify as follows:

```
BUBBLESORT(int [] t, int n)
   int i,j,p
   for i = n downto 2 do
      for j = 1 to i-1 do
         if t[j] > t[j + 1] then
            p= t[j]
            t[j]= t[j+1]
            t[j+1]= p
         end if
      end for
   end for
```

By applying the same method as in the selection sort it is easy to prove that the complexity of bubble sort is $O(n^2)$

Bubble sort is an illustration of the mathematical property that says: " All permutation can be written as a product of of transpositions of two consecutive elements".

4 Insertion Sort

Insertion sort is the natural method used when we want to sort the elements of a list as we are entered them. If the k first element are already sorted, We will then need to place the $(k+1)^{th}$ element, We will then have to :

1. Determine its place amongst the $(k+1)$ elements
2. Do the necessary shifting for its placement;

4.1 Sequential insertion Sort

Here we will assume that we have an array of integers. Those elements are read and inserted one after another in the array of the preceding elements already sorted.

4.1.1 Algorithm

We can easily derive the following pseudo-code

```
SEQINSERTIONSORT(int [] t ,int n)
int i,j,q
for i = 2 to n do
   j = i
   q = t[i]
   while (( j>1) and (t[j-1] > q) do
      t[j] = t[j-1]
      j := j - 1
   end while
   t[j] = q
end for
```

Using the same method as in the selection sort it is shown that the complexity of sequential selection sort is algorithm is $\Theta(n^2)$.

EXAMPLE 4.1.1

W e want to sort the following list:

C	K	G	A	B

step	array	Operation	resulting array
1rst	C K G A B	Initialization, No Change	C K G A B
	C K G A B		
2nd	C **K** G A B	Save the value(K) at the 2^{nd} position	
		Compare K to C, No change	C K G A B
	C K G A B		
3rd	C K **G** A B	Save the value(G) at the 3^{rd} position	
		Compare G to K, Move K to the right	C K K A B
	C K K A B	compare G to C, No change	C K K A B
	C K K A B	insert G at the second position	C G K A B
	C G K A B		
4th	C G K **A** B	Save the value(A) at the 4^{th} position	
		compare A to K, move K to the right	C G K K B
	C G K K B	compare A to G, move G to the right	C G G K B
	C G G K B	compare A to C, move C to the right	C C G K B
	C C G K B	insert A at the first position	A C G K B
	A C G K B		
5th	A C G K **B**	Save the value(A) at the 5^{th} position	
		compare B and K, move K to the right	A C G K K
	A C G K K	compare B to G, move G to the right	A C G G K
	A C G G K	compare B to C, move C to the right	A C C G K
	A C C G K	compare B to A, No change	A C C G K
	A C C G K	insert B at the second position	A B C G K
	A B C G K		

4.2 binary insertion sort

To speed up the search of the position where to insert the element in the sorted part of the array, binary search will be used.

4.2.1 principles

The algorithm consist of : given an array t , if the k first elements are already sorted, use the binary method to search the place of the $(k+1)^{th}$ element .

4.2.2 Algorithm

word represents the element to be placed
istart and *iend* delimit the zone of the array within which *word* will be placed. The following algorithm will search where to insert *word.*

```
int BinarySearh (int [] t, int word ,int istart ,int iend)
int j,k,position
position = istart; j = iend
while  j > position do
   k = (position+j)/ 2
   if word ≤ t[k] then
      j = k
   else
      position = k + 1
   end if
end while
if  (word > t[position]) then
   position = position + 1
end if
return (position)
```

Binary sequential sort can then be performed using the following pseudo-code:

```
BinarySort (int [] t, int n)
int i,j,word,pos
for  i = 2 to n do
   word = t[i]
   pos = BinarySearch(t,word,1,i-1)
   for j=i downto pos+ 1 do
      t[j] = t[j-1]
   end for
   t[pos]=word
end for
```

4.2.3 Complexity

In the binary search, intervals are split until the place to insert word is found. Assuming that we will do m iterations to find the position of word, we then have:
$2^{m-1} \leq n-1 < 2^m$ then m = $\log_2$(n-1) if n is the number of elements in the array.
The execution time of the binary sort algorithm is then:

T(n) = $\sum_{i=2}^{n}(A + Blog_2(i-1) + C + \alpha(i))$

where $\alpha(i)$ is the time used to place a new element in the array. There are three possible cases:

1. *Worst Case*
   $\alpha(i) = (i-1)(e+a) + a$ where a is the cost of an arithmetic operation and e the cost of an assignment. Then

   T(n) = $\sum_{i=2}^{n}(A_1 i + Blog_2(i-1) + C_1)$

   but $log_2 i < i$ then $T(n) \leq \sum_{i=2}^{n}(A_1 i + B(i-1) + C_1) = \sum_{i=2}^{n}(A_2 i + B_2)$

   T(n) $\leq A_2(n-1) - B_2 + A_2\frac{n(n+1)}{2}$
   d'o T(n) = $O(n^2)$.

2. *Best case*
   $\alpha(i)$ = 0
   T(n) = $\sum_{i=2}^{n}(A + Blog_2(i-1)) = \sum_{i=1}^{n-1}(A + Blog_2(i))$
   T(n)=$A(n-1) + B\sum_{i=1}^{n} log_2(i) = A(n-1) + Blog_2(\prod_{i=1}^{n-1} i)$
   but n! = $\sqrt{2\pi n}(\frac{e}{n}^n[1 + O(\frac{1}{n})]$
   then $T(n) = O(n \log n)$.

3. *Average Case*
   $T(n) = \sum_{i=2}^{n}(A + Blog_2(i-1) + C + \alpha(i))$
   the average of $\alpha(i)$ is computed as follows:
   We have :
   0, 1, 2 , 3, . . . i-1 displacements and by summing we have $\frac{i(i-1)}{2}$, as we have i type of displacements (from 0 to i-1) the average is $\alpha_m(i) = \frac{i(i-1)}{2i} = \frac{i-1}{2}$
   T(n) = $\sum_{i=2}^{n}(A + Blog_2(i-1) + C + D\frac{i-1}{2})$
   as $log_2 n < n$ alors
   T(n) = $O(n^2)$

4.3 Comments on Insertion sort

Insertion sort is good middle-of-the-road choice for sorting lists of few thousand items or less. The insertion sort is over twice as fast as the bubble sort and almost 40% faster than the selection sort.

EXAMPLE 4.3.2

W e want to use binary insertion sort to sort the following array

C	K	G	A	B

(1) A single element C is already sorted

(2) We will then try to insert K in the array which consists of a single element C, as K is greater than C , we will then have the array C K as result, this will be done after one comparison , to know if we will insert K before or after C.

(3) We want then to insert G in the array C K ; instead of trying to find the place where to insert from C, we will

- split the array into two parts C and K ,
- compare G to C , as G is greater to C , G can only be inserted in the second part of the list, which consists of a single element K which is greater than G ,
- then G will be inserted between C and K, which results in the array C G K.

(4) We then have to insert A in the array C G K:

- split C G K into two parts C G and K,
- compare A to the last element (G) of C G, as G is greater than A, we will then recursively , using the same method, search for the position where to insert A in C G. split the list into two parts C and G , compare A to C , as A is smaller than C , A will be inserted in the first part of the array, which consists of a single element C (and C is greater than A) , means A will be inserted at the beginning of the array C G, which gives to the array A C G.
- We will then have at this step the list A C G K

(5) We then have to insert B in the array A C G K:

- split A C G K into two parts A C and G K,
- compare B to the last element (C) of A C, as C is greater than B, we will then recursively , search for the position where to insert B in A C. split the array into two parts A and C , compare B to A, as B is greater than A , B will be inserted between A and C, which gives the array A B C.
- We will then have as result the array A B C G K

5 Shell Sort

5.1 Principles

We want to sort an array into increasing order. The idea is to reorganize it in such a way that we obtain a sub-array sorted when selecting h^{th} elements. Such an array is said to be *h-ordered* and is formed by sub-arrays sorted independently. Elements will be inserted in the sub-arrays containing the elements *i-h, i-2h, i-3h*, etc. where h is constant and positive. We then obtain an array in which the 'h' disjoint series

of elements distant of h are sorted separately: it is not sufficient to sort separately the h series to obtain a sorted list: we have to recommence for $h' < h$. The shell sort algorithm performs the sorting by series of distance h_i , then h_{i-1},..., then h_1, where the h_i,...,h_1 form a decreasing sequence with h_1 = 1. For huge array the tests results show that it is preferable to choose the sequence h_i such that h_{i+1} = $3h_i$ + 1, it means it contains the incremental order: ..., 364, 121, 40,13,4,1 ; Then the algorithm will start from h_{m-2}, where m is the smallest integer such that $h_m \geq n$.
The pseudo-code of the algorithm is the following:

```
ShellSort (int [] t, int n)
int i,j,h,v
h=1
while (h < n/9) do
   h = 3*h +1
end while
while (h>0) do
   i=h+1
   while (i<n) do
      v =t[i];j = i
      while (j>h) and  t[j-h] >v do
         t[j] =t[j-h]; j = j-h
      end while
      t[j] = v; i =i+1
   end while
   h=h/3
end while
```

It is shown that in the Shell sort the number of comparisons cannot exceed
$N^{\frac{3}{2}}$ with the sequence:. . .,364,121,40,13,4,1.

5.2 Comments on Shellsort

Efficient for medium-size lists, The Shellsort algorithm is by far the fastest of the n^2 class sorting algorithm. It is more than 5 times faster than bubble sort and little over twice as fast as sequential insertion sort.

6 TreeSort

6.1 Principles

Given an array t , the tree sorting is performed as follows :

1. Build a binary tree such that the label of each node is an element of the array t, and for each node ,n, of this tree , labels of all nodes of the left subtree are less than the label of n , and labels of the nodes of the right subtree are greater or equal to the label of n.

2. Traverse the tree constructed at 1 in the order Left-root-right(symmetric or infix) and record the label of the node at each traversal.

EXAMPLE 6.1.3

W e want to sort the following list : -4 2 3 -1 3 1 2

We will represent a binary tree by

Binary tree is $\begin{cases} \emptyset(\text{ empty set }) & \text{if the tree is empty} \\ (R, LT, RT) & \text{otherwise (where } R \text{ is the root} \\ & \text{and LT (left subtree)} \\ & \text{and RT (light tree) are also binary trees.} \end{cases}$)

step 1: Building of the tree

value	tree
-4	(-4,∅,∅)
2	(-4,∅,(2,∅,∅))
3	(-4,∅,(2,∅,(3,∅,∅)))
-1	(-4,∅,(2,(-1,∅,∅),(3,∅,∅)))
3	(-4,∅,(2,(-1,∅,∅),(3,∅,(3,∅,∅))))
1	(-4,∅,(2,(-1,∅,(1,∅,∅)),(3,∅,(3,∅,∅))))
2	(-4,∅,(2,(-1,∅,(1,∅,∅)),(3,∅,(2,∅,∅),(3,∅,∅))))

step 2:

Symmetric tree traversal of the tree built at step1 gives the following result :

step	list
1	-4
2	-4 -1
3	-4 -1 1
4	-4 -1 1 2
5	-4 -1 1 2 2
6	-4 -1 1 2 2 3
7	-4 -1 1 2 2 3 3

6.2 Pseudo-code of Tree sort

The pseudo-code of tree sort is specify as follows:
Consider that the binary has the following type

```
struct bintree
  {
   bintree lt
   int     label
   bintree rt
  }
```

We want to sort the array t which contains n elements , The pseudo-code of the tree sorting is:

```
binsort(int [] t, int n)
  1: bintree bint
  2: bint = binTreeConst(t,n)
  3: Infix(bint)
```

The procedure binTreeConst constructing the binary is specified as follows :

```
bintree binTreeConst(int [] t, int n)
int i
bintree bint
insert(bint,T[1])
for i = 2 to n do
    insert(bint,T[i])
end for
return bint
```

The procedure insert will insert the elements as they are read from the array t. If the element to be inserted is *word* and t represents the binary tree where to insert, the insertion is specified as follows:

```
Insert(bintree t, int word)
if t = nil then
    t= construct(word,nil,nil)
else
    if word < t.label then
        Insert(t.lt,word)
    else
        Insert(t.rt,word)
    end if
end if
```

construct(r,t1,t2) constructs a binary tree with root labelled r and with t1 as a left subtree and t2 as a right subtree.

The pseudo-code of symmetric (infix) traversal is:

```
Infix(bintree t)
if t ≠ nil then
    Infix(t.lt)
    display(T.label)
    Infix(t.rt)
end if
```

Exercise: Analyze the temporal complexity of tree sort.

7 Quicksort

Quicksort is popular for three reasons

- It is easy to implement
- It is general purpose sort
- Most of the time it requires least resources.

We will first define a naive version of quicksort.

7.1 Naive Algorithm

Quicksort algorithm is based on the divide-and-conquer method. It first partition the array to sort into two parts and then sort independently those parts, The naive version can be specified as follows:

```
NaiveQuickSort(int [] t, int l, int r)
if (r>l) then
    i=PARTITION(l,r)
    NaiveQuickSort(t,l,i-1)
    NaiveQuickSort(t,i+1,r)
end if
```

Here l and r represent lower bound and upper bound of the sorting interval.

7.2 Quicksort analysis

The core part of the method is the procedure of partitioning which organizes the array according to the following conditions:

1. There exists an index i such that the element t[i] is at its final position ;
2. All the elements t[l],...,t[i-1] are less or equal to t[i];
3. All the elements t[i+1],...,t[r] are greater or equal to t[i].

The following simple and general strategy is used: choose the first element arbitrary t[r] as element ”pivot” to place in its final position . After that, traverse the array from the left , until an element greater than t[r] is found and traverse the array from the right until an element less than t[r] is found . The two elements which have stopped the traversal are not in their right place in the partitioning and are then swop. Continue the process until the t[r] is at the position such that all the element preceding it are less or equal and all those following it are greater or equal.
The pseudo-code of quicksort can be specify as follows:

```
QuickSort(int [] t, int l, int r)
int i,j,u,v
boolean utile
begin
if (r>l) then
    v=t[r];i= l-1;j=r
    utile = true
    while utile do
        i=i+1
        while (i <= r) and (t[i] < v) do
            i=i+1
        end while
        j=j-1
        while (j >= l) and (t[j] > v) do
            j=j-1
        end while
        if i>=j then
            utile=false
        else
            u = t[i]
            t[i]=t[j]
            t[j]=u
        end if
    end while
    u=t[i];t[i]=t[r];t[r]=u
    QuickSort(t,l,i-1) ;
    QuickSort(t,i+1,r)
end if
```

7.3 Quicksort complexity

7.3.1 Average Case Analysis of QuickSort

Assume that all the keys are distinct and that all permutations are equally probable. The recurrence relation verify by the number of comparisons C_n is the following:
$C_n = n + 1 + \frac{1}{n}\sum_{k=1}^{n}(C_{k-1} + C_{n-k})\ n \geq 2$
where $C_1 = C_0 = 0$
we have n + 1 comparisons of pivot with each of other elements, and each element k has probability$\frac{1}{n}$ to be chosen as a pivot, and then we will have two sub-arrays with sizes $k-1$ and $n-k$ respectively, We will then have have arrays of sizes $k-1$ and $n-k$ to be sorted.
We have $C_1 + C_2 + ... + C_{n-1} = C_{n-1} + C_{n-2} + ... + C_1$ thus $C_n = n + 1 + \frac{2}{n}\sum_{k=1}^{n} C_{k-1}$
we have following equality $nC_n - (n-1)C_{n-1} = n(n+1) - (n-1)n + 2C_{n-1}$
which gives $nC_n = (n+1)C_{n-1} + 2n$ thus
$\frac{C_n}{n+1} = \frac{C_{n-1}}{n} + \frac{1}{n+1} = \frac{C_{n-2}}{n-1} + \frac{2}{n} + \frac{2}{n+1} = ... = \frac{C_2}{3} + \sum_{k=3}^{n}\frac{2}{k+1}$
then $\frac{C_n}{n+1} = 2\sum_{k=1}^{n}\frac{1}{k} = \int_1^n \frac{1}{x}dx = 2Ln(n)$
$Cn = 2(n+1)Ln(n)$. We can then conclude that $C_n = O(nLn(n))$ and the running time of the Quicksort in the average case is an $O(nLn(n))$

7.3.2 Worst Case Analysis of Quicksort

The partition process working on an array t with k elements performs $k+1$ comparisons.
If $t[r]$ is the largest element, we will then have two array to sort in the further step, one empty (elements greater than $t[r]$) and another one with $k-1$ elements.
Means if at each time we have to perform the partition, $t[r]$ is largest element the total number of comparisons is:

$$C_n = \sum_{k=1}^{n}(k+1) = \frac{(n+1)(n+2)}{2} = \Theta(n^2)$$

EXAMPLE 7.3.4

U sing QuickSort to sort the list 6 3 1 2 4 5 we have the following steps:

1. At this stage r = 6 , l = 1 means we
   have $r > l$ we will then have
   v = t[r] = 5(the pivot), i = l-1 = 0, j = r = 6.
   utile = true.
   increment i
   from the current value of i(=1) (incrementing at each step) find the i such that $t[i] >= v$ or $i > r$, it will be i = 1
   decrement j
   from the current value of j(=5) (decrementing at each step) find the j such that $t[j] <= v$ or $j < l$, it will be j = 5
   because $i < j$, swap the contents of t[i] and t[j]
   and we have as result, the list 4 3 1 2 6 5
   utile still true
   increment i
   from the current value of i(=2) (incrementing at each step) find the i such that $t[i] >= v$ or $i > r$, it will be i = 5
   decrement j
   from the current value of j(=4) (decrementing at each step) find the j such that $t[j] <= v$ or $j < l$, it will be j = 4
   because $i > j$, utile will become false and the contents of t[i] and t[r] will be swop. The result is 4 3 1 2 **5** 6 . 5 is then at its final position.
   We then have to sort in place 4 3 1 2 and 6

2. the pivot here is 2, i = 1 (t[i]= 4) will stop the traversal left-right and j = 3(t[j]= 1) will stop the traversal (right-left) , then swap the contents of t[1] and t[3], the result is the list 1 3 4 2, i = 2 will stop the traversal (left-right) and j =1 will stop the traversal (right-left). Because $i > j$, utile will become false and the contents of t[2] and t[4] will be swop. The result is 1 **2** 4 3 . 2 is then at its final position. We then have to sort at this step 1 and 4 3

   2.1 1 is in final position because it is reduce to a single element , with consequence that $l = r$.

   2.2 for the list 4 3 the pivot is 3 , i = 3 and j = 3 will stop the traversals. because $i = j$ utile will become false and the contents of t[3] and t[4] will be swop The result is **3** 4. 3 is then at its final position. 4 is in its final position because it is reduce to a single element , with consequence that $l = r$.

   the result of this step is the sub-list : 1 2 3 4

3. 6 is in final position because it is reduce to a single element , with consequence that $l = r$.

4. **result**: the sorted list is 1 2 3 4 5 6

8 MergeSort

8.1 Analysis

Given an array t with size n. We want to sort t in increasing order. MergeSort is a divide-and-conquer approach which consists of :

- division the array into two parts as equal as possible, t1 and t2
- sort t1 and t2 recursively
- and merge them

The base case of the the recursion is the size of the sub array to be sorted, if it is less than a chosen threshold , then use the insertion sort to preform the sort.

8.2 algorithm

Let t be the array to be sorted and n its size, the algorithm is specify as follows:

```
MergeSort(int []t, int n)
int [] U
int [] V
int i,j,l
begin
if n ≤ threshold then
   InsertionSort(t,n)
else
   j = (1+n)/ 2; i = n - j
   for l = 1 to j do
      U[l] = t[l]
   end for
   for l = j +1 to n do
      V[l-j] = t[l]
   end for
   MergeSort(u,j)
   MergeSort(v,i)
   Merge(u,v,t,j,i)
end if
```

InsertionSort is insertion sort algorithm, Merge(u,v,t,i,j) merges two arrays U and V with sizes j and i respectively in the array t , *threshold* is size , of t, below which MergeSort is useless.

Complexity of the algorithm
- The splitting of array consumes a linear time $O(n)$
- The merger consumes a linear time $O(n)$
If $t(n)$ is the execution time of MergeSort(t,n) we will then have:
$t(n) = 2t([\frac{n}{2}]) + O(n)$
We can then conclude using master theorem (see chapter on Design Methods) one that $t(n) = O(n \log n)$.
Mergesort is a fast algorithm and need more memory to be performed.

EXAMPLE 8.2.1

U sing MergeSort to sort the list 6 3 1 2 4 5 we have the following steps:

1. *Split the list* : we have two sub-lists 6 3 1 and 2 4 5
2. Apply MergeSort to 6 3 1
   - 2.1 Split the list : we have the sub-lists 6 and 3 1
   - 2.2 Apply MergeSort to 6 which is a single element , then it is sorted
   - 2.3 Apply MergeSort to 3 1, we can just swap the two element to obtain the sorted pair 1 3
   - 2.4 Merge 6 and 1 3 , we obtain 1 3 6
3. Apply Merge sort to 2 4 5
   - 2.1 Split the list : we have the sub-lists 2 and 4 5
   - 2.2 Apply MergeSort to 2 which is a single element , then it is sorted
   - 2.3 Apply MergeSort to 4 5, the pair is sorted .
   - 2.4 Merge 2 and 4 5 , we obtain 2 4 5
4. Merge 1 3 6 and 2 4 5 and the result is 1 2 3 4 5 6

9 Radix Sort

Radix sort belongs to the class of bucket sort algorithms. We will first study the bucket sort algorithms.

9.1 Bucket Sort

Assuming that elements to be sorted consist of d digits, represented in radix k. If the radix is 2 then each number will be represented in binary. d has to be a constant. d and k are parameters to the algorithm and will affect the running time.
A bucket sort runs according to the following principle:

1. Distribute the elements over a number of buckets.The number of buckets is equal to the radix k. Elements are distributed by examining a particular field from each element.
   The running time of the distribution is $\Theta(n)$.
2. Sort the elements in each bucket. If there are n_i elements in the bucket i, any sorting algorithm can be used to sort the n_i elements. If QuickSort is used the number of comparisons is $S(n_i) = O(n_i \log n_i)$.

   Over all the k buckets we will have $\sum_{i=1}^{k} S(n_i)$ comparisons.
3. Combine the k buckets . This combination is an $O(n)$.

We Assume that the elements are uniformly distributed in the bucket, means there are $\frac{n}{k}$ elements in each bucket. The running time will be computed as follows:

- Sorting in bucket consumes $T_2(n) = \displaystyle\sum_{i=1}^{k} a\frac{n}{k}\log(\frac{n}{k}) = k(a\frac{n}{k}\log\frac{n}{k})an\log\frac{n}{k} = O(n\log\frac{n}{k})$
- The three steps of bucket sort will consume $T(n) = O(n) + O(n\log\frac{n}{k}) + O(n) = O(n\log\frac{n}{k})$. Considering the d digits, we then have the overall running time being $O(dn\log\frac{n}{k}) = O(n\log\frac{n}{k})$ as

d is constant.
If $k = O(n)$, for instance $k = \frac{n}{10}$, the running is $O(n \log 10) = O(n)$.

9.2 Radix sort

The principle of radix sort is as follows:

1. Allocate k buckets; Set p = rightmost digit
2. Distribute the elements from array into the buckets considering the p digit. In other words allocate to i bucket all the elements for which p digit is i.
3. Combine the elements from the buckets into array, with the elements in the bucket i preceding the elements in the buckets $i + 1$, for $k = 1, 2, \ldots, k - 1$
4. p = p-1; if $p \geq 1$, goto step 2

Note that the relative order of two elements placed in the same bucket is not changed.

EXAMPLE 9.2.1

U se radix sort to sort the following decimal numbers: 170, 045, 075, 090, 002, 024, 802, 065 .
The radix here is $k = 10$ and the number of digits is $d = 3$
The steps of the radix sort are

1. Sorting by least significant digit gives:
$\underbrace{17\mathbf{0}, 09\mathbf{0}}_{bucket1} \underbrace{00\mathbf{2}, 80\mathbf{2}}_{bucket2} \underbrace{02\mathbf{4}}_{bucket3} \underbrace{04\mathbf{5}, 07\mathbf{5}, 06\mathbf{5}}_{bucket4}$
2. Sorting by 10s places gives:
$\underbrace{0\mathbf{0}2, 8\mathbf{0}2}_{bucket1} \underbrace{0\mathbf{2}4}_{bucket2} \underbrace{0\mathbf{4}5}_{bucket4} \underbrace{0\mathbf{6}7}_{bucket5} \underbrace{1\mathbf{7}0, 0\mathbf{7}5}_{bucket6} \underbrace{0\mathbf{9}0}_{bucket7}$
3. Sorting by most significant digit gives:
$\underbrace{\mathbf{0}02, \mathbf{0}24, \mathbf{0}45, \mathbf{0}65, \mathbf{0}75, \mathbf{0}90}_{bucket1} \underbrace{\mathbf{1}70}_{bucket2} \underbrace{\mathbf{8}02}_{bucket3}$

Radix Sort algorithm is then:

RadixSort(digits []t, int n, int d)
for i = 1 **to** d **do**
  Use a stable sort to sort the Array A on digit i
end for

The running time of Radix sort is evaluated as follows:
For each digit, the algorithm consumes time in $\Theta(n + r)$ where $1 \leq r \leq k - 1$
For d digits, it take $\Theta(d(n + r))$ times
Since $r = O(n)$ because $k = O(n)$, and d is constant, $\Theta(d(n + r)) = \Theta(n)$.
Radix sort is not an in-pace algorithm. It is difficult to write a general purpose version of radix sort.

10 External Sort

We have previously studied the problem of sorting assuming that the elements to be sorted were all in the main memory. But in practical those cases are rare.

In general data are placed in secondary memories (Discs, magnetic tapes, ,etc.). As input/output time is in general greater than time spent to access the main memory, the execution time of external sort depends mostly on input/output operations. The access mode (sequential, random, etc.) is crucial in the design of external sorting.
The general schema of external sorting is the following:

- The construction of sorted sub-arrays.
- the dynamic allocation of sorted sub-arrays to different locations of the secondary memory,
- the fusion of sub-arrays.

The efficiency of an algorithm to solve these problems relies ont its ability to perform as less as possible input/output operations. It is also important to avoid the situation where all the sorted sub-arrays are on a single location when the sorting has not ended yet.

Bibliography

1. Cormen T. H., Leiserson C.E. and Rivest R. L., *Introduction to Algorithms*, McGraw-Hill, 1990.
2. Sedgewick R., *Algorithms in C++*,Part 1-4, Addison-Wesley, 2002.
3. Cormen T. H., Leiserson C.E. and Rivest R. L., *Introduction to Design and Analysis*, Addison Wesley, 2000.
4. Brassard G. and Bratley P., . *Fundamentals of Algorithmics*, Prentice-Hall, 1996.