PES UNIVERSITY

Text Book: Introduction to the Design and Analysis of Algorithms
Author: Anany Levitin 2 nd Edition

Unit-4

1. Space and Time Trade-Offs

Consider, as an example, the problem of computing values of a function at many points in its domain. If it is time that is at a premium, we can precompute the function's values and store them in a table. This is exactly what human computers had to do before the advent of electronic computers, in the process burdening libraries with thick volumes of mathematical tables. Though such tables have lost much of their appeal with the widespread use of electronic computers, the underlying idea has proven to be quite useful in the development of several important algorithms for other problems. In somewhat more general terms, the idea is to preprocess the problem's input, in whole or in part, and store the additional information obtained to accelerate solving the problem afterward. This approach is called input ***enhancement***.

The other type of technique that exploits space-for-time trade-offs simply uses extra space to facilitate faster and/or more flexible access to the data and this approach is called ***prestructuring.*** There is one more algorithm design technique related to the space-for-time trade-off idea: ***dynamic programming***. This strategy is based on recording solutions to overlapping subproblems of a given problem in a table from which a solution to the problem in question is then obtained.

I. **Input Enhancement**
   - Preprocess the problem's input, in whole or in part, and store the additional information obtained to accelerate solving the problem afterward.
   - Eg: Counting methods of sorting, Horspool's algorithm, Boyer-Moore's algorithm.

Sorting by Counting

1. **Comparison Counting Sorting**
   - For each element of the list, count the total number of elements smaller than this element.
   - These numbers will indicate the positions of the elements in the sorted list.
2. **Distribution Counting Sorting**
   - Suppose the elements of the list to be sorted belong to a finite set (aka domain).
   - Count the frequency of each element of the set in the list to be sorted.

- Scan the set in order of sorting and print each element of the set according to its frequency, which will be the required sorted list.

1 Comparison Counting Sorting

- For each element of the list, count the total number of elements smaller than the element.
- These numbers indicates the positions (0-based) of the elements in the sorted list.

Eg: **62 31 84 96 19 47**

lesser elements: **3 1 4 5 0 2**

ALGORITHM *ComparisonCountingSort*$(A[0..n-1])$

```
//Sorts an array by comparison counting
//Input: An array A[0..n − 1] of orderable elements
//Output: Array S[0..n − 1] of A's elements sorted
for i ← 0 to n − 1 do Count[i] ← 0
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] < A[j]
            Count[j] ← Count[j] + 1
        else Count[i] ← Count[i] + 1
for i ← 0 to n − 1 do S[Count[i]] ← A[i]
return S
```

Example of sorting by comparison counting

Array $A[0..5]$		62	31	84	96	19	47
Initially	$Count[]$	0	0	0	0	0	0
After pass $i = 0$	$Count[]$	3	0	1	1	0	0
After pass $i = 1$	$Count[]$		1	2	2	0	1
After pass $i = 2$	$Count[]$			4	3	0	1
After pass $i = 3$	$Count[]$				5	0	1
After pass $i = 4$	$Count[]$					0	2
Final state	$Count[]$	3	1	4	5	0	2
Array $S[0..5]$		19	31	47	62	84	96

The time efficiency of this algorithm:

It should be quadratic because the algorithm considers all the different pairs of an n-element array. More formally, the number of times its basic operation, the comparison $A[i]<A[j]$, is executed is equal to the sum we have encountered several times already:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) = \frac{n(n-1)}{2}$$

Thus, the algorithm makes the same number of key comparisons as selection sort and in addition uses a linear amount of extra space. On the positive side, the algorithm makes the minimum number of key moves possible, placing each of them directly in their final position in a sorted array.