UNIT 3: Space and Time Tradeoffs

Sorting by counting

Space - Time (time-memory) Tradeoff

- case where an algorithm trades increased space(data storage) usage with decreased time (computation/response time) or visa-versa.

- **In 1980 Martin Hellman first proposed using a time–memory tradeoff for cryptanalysis** (study of analyzing information systems in order to study the hidden aspects of the systems)

Space - Time Trade-off : Idea

Input enhancement

preprocess the problem's input, in whole or in part, and store the additional information obtained to accelerate solving the problem afterward.

Pre-structuring

some processing is done before a problem in question is actually solved but, unlike the input-enhancement variety, it deals with access structuring. (example: hashing)

Types of tradeoff

- Lookup tables vs. recalculation
- Compressed vs. uncompressed data
- Re-rendering vs. stored images
- Smaller code vs. loop unrolling

Algorithms that also make use of space–time tradeoffs:
(some examples)

- Baby-step giant-step algorithm for calculating discrete logarithms
- Rainbow tables in cryptography
- The meet-in-the-middle attack uses a space–time tradeoff to find the cryptographic key
- Dynamic programming

Space–time tradeoffs: NOTE

- Space and time need not compete with each other in all design situations

- Algorithm may be designed with space efficient data-structure that minimizes both the running time and the space.

  Example: Adj matrix vs. Adj list to store graph input

Sorting by counting

- Input enhancement technique

Sorting by counting (comparison counting sort): Idea

- For each element of a list to be sorted, count the total number of elements smaller than this element and record the results in a table.
- These numbers will indicate the positions of the elements in the sorted list

Example:

if the count is 10 for some element, it should be in the 11th position (starting index = 0) in the sorted array.

Sort : 62, 31, 84, 96, 19, 47

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

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 in nondecreasing order

for $i \leftarrow 0$ **to** $n-1$ **do**

  $Count[i] \leftarrow 0$

for $i \leftarrow 0$ **to** $n-2$ **do**

  for $j \leftarrow i+1$ **to** $n-1$ **do**

    if $A[i] < A[j]$

      $Count[j] \leftarrow Count[j]+1$

    else

      $Count[i] \leftarrow Count[i]+1$

for $i \leftarrow 0$ **to** $n-1$ **do**

  $S[Count[i]] \leftarrow A[i]$

return S

Comparison counting sorting algorithm analysis

1. input's size: **n –** number of elements to be sorted.
2. basic operation: **comparison**

$$\boxed{A[i] < A[j]}$$

3. No worst, average, and best cases.
4. Let C(n) = number of times the basic operation is executed.

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

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

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

$$= \frac{n(n-1)}{2} \in \boxed{\Theta(n^2)}.$$

NOTE:

Algorithm in addition uses a linear amount of extra space, it can hardly be recommended for practical use.

Let’s check our understanding

- Is it possible to exchange numeric values of two variables, say, u and v, without using any extra storage?

- Will the comparison counting algorithm work correctly for arrays with equal values?

Note:

Comparison counting algorithm fails in the following scenarios:

- Elements to be sorted belong to a known small set of values (Example: 1, 2, 1, 1, 2, 2, 1, 2, 1, 1, 2, 2)
  - ✓ **Solution: Frequency counting**

- List cannot be overwritten with sorted elements

Distribution counting

- Input enhancement technique

Distribution counting: Idea

- compute the frequency of each element
- copy elements into a new array S[0 .. n - 1] to hold the sorted list

Note:

Distribution/frequency values indicate the proper positions for the last occurrences of their elements in the final sorted array. If we index array positions from 0 to n-1, the distribution values must be reduced by 1 to get corresponding element positions.

Input array to sort : 13, 11, 12, 13, 12, 12

Array values	11	12	13
Frequencies	1	3	2
Distribution values	1	4	6

It is more convenient to process the input array right to left

	D[0..2]			S[0..5]					
$A[5] = 12$	1	**4**	6				12		
$A[4] = 12$	1	**3**	6			12			
$A[3] = 13$	1	2	**6**						13
$A[2] = 12$	1	**2**	5		12				
$A[1] = 11$	**1**	1	5	11					
$A[0] = 13$	0	1	**5**					13	

ALGORITHM *DistributionCounting*$(A[0..n-1], l, u)$

```
//Sorts an array of integers from a limited range by distribution counting
//Input: An array A[0..n − 1] of integers between l and u (l ≤ u)
//Output: Array S[0..n − 1] of A's elements sorted in nondecreasing order
for j ← 0 to u − l do
        D[j] ← 0                               //initialize frequencies
for i ← 0 to n − 1 do
        D[A[i] − l] ← D[A[i] − l] + 1 //compute frequencies
for j ← 1 to u − l do
        D[j] ← D[j − 1] + D[j]        //reuse for distribution
for i ← n − 1 downto 0 do
    j ← A[i] − l
    S[D[j] − 1] ← A[i]
    D[j] ← D[j] − 1
return S
```

Distribution counting sorting algorithm analysis

- it makes just two consecutive passes through its input array → Linear algorithm
- better time-efficiency class than that of the most efficient sorting algorithms

However, this efficiency is obtained by

- **exploiting the specific nature of input lists**
- **in addition to trading space for time.**

Let's check our understanding

- Assuming that the set of possible list values is {a, b, c, d}, sort the following list in alphabetical order by the distribution counting algorithm:

  b, c, d, c, b, a, a, b.

- Is the distribution counting algorithm stable?