Space-Time Tradeoffs

Prepared by

Dr. Rashmi S

Space-for-time tradeoffs

Varieties of space-for-time algorithms:

- *Input Enhancement* — preprocess the input (or its part) to store some info to be used later in solving the problem
  - counting sorts
  - string searching algorithms
- *Prestructuring* — preprocess the input to make accessing its elements easier
  - hashing
  - indexing schemes (e.g., B-trees)
- *Dynamic Programming*

Sorting by Counting

- Idea is to count, for each element of a list to be sorted, 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:
- e.g., if the count is 10 for some element, it should be in the 11th position (with index 10, if we start counting with 0) in the sorted array.
- Thus, we will be able to sort the list by simply copying its elements to their appropriate positions in a new, sorted list.
- This algorithm is called **comparison counting sort**

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

7.1 Example of sorting by comparison counting.

Comparison Counting

```
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

- If element values are integers between some lower bound *l and upper bound u, we can compute the frequency* of each of those values and store them in array *F[0..u − l]*.
- *Then the first F[0]* positions in the sorted list must be filled with *l, the next F[1] positions with l + 1,* and so on.
- All this can be done, of course, only if we can overwrite the given elements.

Time Efficiency

Basic operation is the comparison statement

A[i]<A[j]

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

Distribution Counting

- Consider a situation of sorting a list of items with some other information associated with their keys so that we cannot overwrite the list's elements.
- Then we can copy elements into a new array $S[0..n-1]$ to hold the sorted list as follows.
- The elements of A whose values are equal to the lowest possible value l are copied into the first $F[0]$ elements of S,

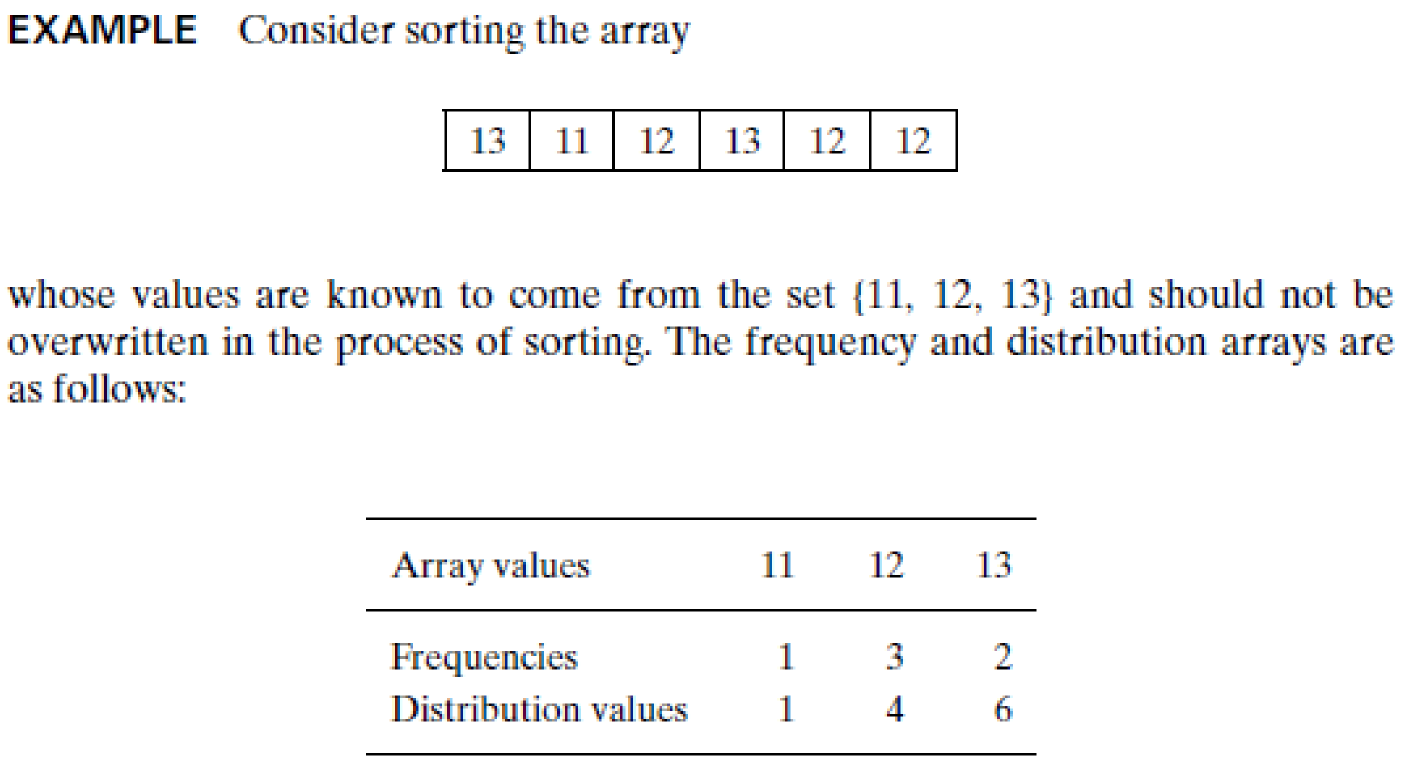
  i.e., positions 0 through $F[0]-1$;

  the elements of value $l+1$ are copied to positions from

  $F[0]$ to $(F[0]+F[1])-1$;

  and so on.
- The method is known as **distribution counting**

Distribution Counting



EXAMPLE Consider sorting the array

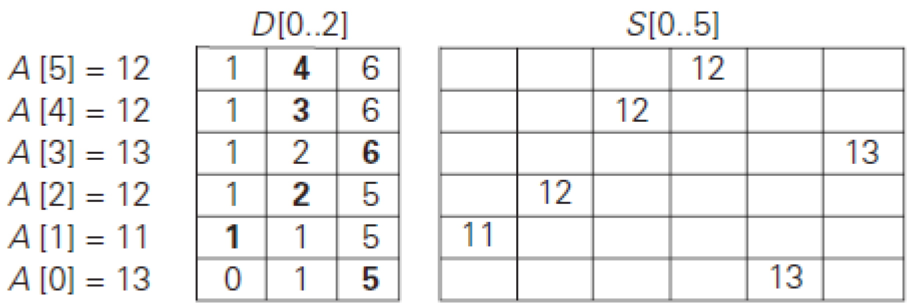
13	11	12	13	12	12

whose values are known to come from the set {11, 12, 13} and should not be overwritten in the process of sorting. The frequency and distribution arrays are as follows:

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

Note that the distribution 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

Distribution Counting



	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	

FIGURE 7.2 Example of sorting by distribution counting. The distribution values being decremented are shown in bold.

Distribution Counting

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

Time Efficiency

- Efficiency: $\Theta(n)$
  - Best so far but only for specific types of input

Strengths and Weaknesses of Space-Time Tradeoffs

- ✓ Strengths:
  - Suited to amortised situations
  - Particularly effective in accelerating access to data
- Weaknesses:
  - Can be space expensive and this might have empirical effects on speed

Space and Time Tradeoffs-
Input Enhancement in String Matching-
Boyer Moore Algorithm

Prepared by
Dr. Rashmi S

Boyer-Moore algorithm

Based on the same two ideas:

- comparing pattern characters to text from right to left
- precomputing shift sizes in two tables
  - *bad-symbol table* indicates how much to shift based on text's character causing a mismatch
  - *good-suffix table* indicates how much to shift based on matched part (suffix) of the pattern (taking advantage of the periodic structure of the pattern)

Boyer-Moore algorithm

- If the rightmost character of the pattern doesn’t match, BM algorithm acts as Horspool’s

- If the rightmost character of the pattern does match, BM compares preceding characters right to left until either all pattern’s characters match or a mismatch on text’s character c is encountered after k > o matches

Bad symbol shift in Boyer-Moore algorithm

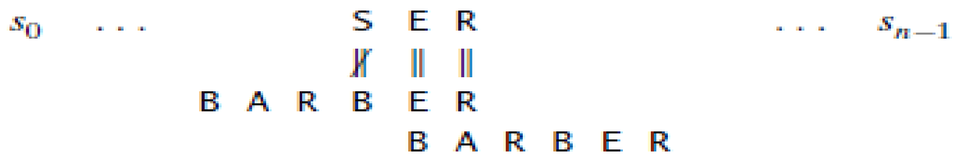
$$\begin{array}{cccccccccc}
s_0 & \dots & & c & s_{i-k+1} & \dots & s_i & \dots & s_{n-1} & \text{text} \\
 & & & \not\parallel & \parallel & & \parallel & & & \\
 & p_0 & \dots & p_{m-k-1} & p_{m-k} & \dots & p_{m-1} & & & \text{pattern}
\end{array}$$

- first one is guided by the text's character *c that caused* a mismatch in the pattern. It is called the ***bad symbol shift***
- Size of this shift = *t1(c) – k*
- *where t1(c) is the entry in the precomputed table used by Horspool's* algorithm
- *k is the number of matched characters*

$$\begin{array}{cccccccccc}
s_0 & \dots & & c & s_{i-k+1} & \dots & s_i & \dots & s_{n-1} & \text{text} \\
 & & & \not\parallel & \parallel & & \parallel & & & \\
 & p_0 & \dots & p_{m-k-1} & p_{m-k} & \dots & p_{m-1} & & & \text{pattern} \\
 & & & & p_0 & \dots & & p_{m-1} & &
\end{array}$$

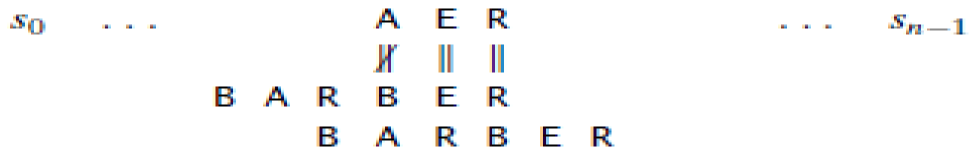
Examples

1.



shift the pattern by *t1(S) – 2 = 6 – 2 = 4 positions*

2.



shift the pattern by *t1(A) – 2 = 4 – 2 = 2 positions*

Bad symbol shift in Boyer-Moore algorithm

- If *$t_1(c) - k \le 0$, we do not want to shift the pattern by 0 or a negative* number of positions
- Bad-symbol shift *'d_1' is computed by the Boyer-Moore* algorithm either as *$t_1(c) - k$ if this quantity is positive and as 1 if it is negative* or zero.
- This can be expressed by the following compact formula:

$$\boldsymbol{d_1 = \max\{t_1(c) - k, 1\}}$$

Good-Suffix Shift in Boyer-Moore algorithm

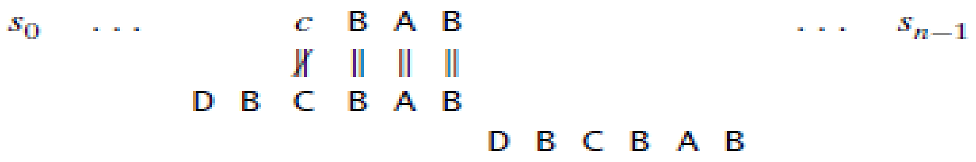
- Good-suffix shift ‘d2’ is applied after 0 < k < m last characters were matched
- We refer to the ending portion of the pattern as its suffix of size *k and denote it suff (k)*

- When there is another occurrence of *suff (k) not* preceded by the same character as in its rightmost occurrence
- Shift the pattern by the distance *d2 between such a second rightmost occurrence and its* rightmost occurrence

k	**pattern**	d_2
1	$\text{ABC}\overline{\text{B}}\text{A}\underline{\text{B}}$	2
2	$\overline{\text{AB}}\text{CB}\underline{\text{AB}}$	4

Good-Suffix Shift in Boyer-Moore algorithm

- If there is no other occurrence of *suff (k) not preceded by* the same character as its rightmost occurrence



- For the pattern DBCBAB and *k = 3*, shift the pattern by its entire length of 6 characters
- Good-suffix table of the Boyer-Moore algorithm—for the pattern ABCBAB

k	**pattern**	d_2
1	$\text{ABC}\overline{\text{B}}\text{A}\underline{\text{B}}$	2
2	$\overline{\text{AB}}\text{CB}\underline{\text{AB}}$	4
3	$\overline{\text{AB}}\text{C}\underline{\text{BAB}}$	4
4	$\overline{\text{AB}}\underline{\text{CBAB}}$	4
5	$\overline{\text{A}}\overline{\underline{\text{B}}}\underline{\text{CBAB}}$	4

Example of string matching with the Boyer-Moore algorithm.

$$d = \begin{cases} d_1 & \text{if } k = 0, \\ \max\{d_1, d_2\} & \text{if } k > 0, \end{cases}$$

where $d_1 = \max\{t_1(c) - k, 1\}$.

```
B  E  S  S  _  K  N  E  W  _  A  B  O  U  T  _  B  A  O  B  A  B  S
B  A  O  B  A  B
$d_1 = t_1(K) - 0 = 6$
               B  A  O  B  A  B
               $d_1 = t_1(\_) - 2 = 4$
                              B  A  O  B  A  B
               $d_2 = 5$
                              $d_1 = t_1(\_) - 1 = 5$
               $d = \max\{4, 5\} = 5$
                              $d_2 = 2$
                              $d = \max\{5, 2\} = 5$
                                             B  A  O  B  A  B
```

Boyer-Moore Algorithm

- Step 1 : Fill in the bad-symbol shift table
- Step 2 : Fill in the good-suffix shift table
- Step 3 : Align the pattern against the beginning of the text
- Step 4: Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and the text until either all *m character pairs are matched* (then stop) or a mismatching pair is encountered after *k* ≥ *0 character* pairs are matched successfully. In the latter case, retrieve the entry *t1(c) from the c's column of the bad-symbol table where c is the text's* mismatched character. If *k* > *0, also retrieve the corresponding d2* entry from the good-suffix table. Shift the pattern to the right by the number of positions computed by the formula

$$d = \begin{cases} d_1 \text{ if } k = 0, \\ \max\{d_1, d_2\} \text{ if } k > 0, \end{cases} \qquad \text{where } d_1 = \max\{t_1(c) - k, 1\}$$

Time Complexity

- When searching for the first occurrence of the pattern, the worst-case efficiency of the Boyer-Moore algorithm is known to be linear
- Though this algorithm runs very fast, especially on large alphabets, many people prefer its simplified versions, Horspool’s algorithm