Space and Time Tradeoffs

Space-for-time tradeoffs

- Input enhancement:
  - Preprocess the problems input in whole or in part
  - Store the additional information to accelerate solving the problem
  - Counting methods for sorting
  - String matching

- Pre-structuring:
  - Extra space to facilitate faster and more flexible access to data
  - Some processing is done before a problem is solved
  - Deals with access structuring
  - Hashing
  - Indexing with B-trees

Space-for-time tradeoffs

- Time and space do not need to compete with each other
- Align them to minimize both running time and the space consumed
- An efficient presented structure, in turn, leads to a faster algorithm
- Consider data compression

Sorting by Counting

- Comparison-Counting sort
- For each element, count the number of smaller elements
- Record the result in a table
- The count indicates the position of element when sorted
- Copy each element to its proper position in a new array

Sorting by Counting

- Comparison-Counting sort

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 ← 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
```

Sorting by Counting

- Comparison-Counting sort

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

$$C(n) \in \theta\left(n^2\right)$$

- Same number of comparisons as selection sort
- Minimum number of key move possible

Sorting by Counting

- Comparison-Counting sort

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

Sorting by Counting

- Comparison-Counting sort

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

Sorting by Counting

- Distribution-Counting sort

- The value set is known in a sorted list
- Values can not be overwritten during sorting

- Find the frequency of each unique value
- Find the distribution of values
  - For first value → its frequency
  - For the rest → distribution of previous value + its own frequency
- For each value starting from right of array
- Place the value in position that its distribution suggests
- Decrease distribution value by one

Sorting by Counting

- Distribution-Counting sort

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 $(l \le u)$
//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
```

Sorting by Counting

- Distribution-Counting sort

13	11	12	13	12	12

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

	$D[0..2]$			$S[0..5]$					
$A[5] = 12$	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	

Sorting by Counting

- Distribution-Counting sort

- $C(n) \in \theta(n)$
- Two successive passes through the values

- Better time-efficiency than most sorting algorithms
  - Merge sort
  - Quick sort
  - Heap sort

- Efficiency is obtained by:
  - Knowing the input values
  - Trading space for time

String Matching

- Find occurrence of pattern of m characters
- in a longer text of n characters

- Brute-force
- Match corresponding pairs of characters in pattern and text
- Left to right
- If mismatch, then shift pattern on character to right
- Test the next trial
- $C_{worst}(n) \in O(nm)$
- $C_{average}(n) \in O(n+m)$

String Matching

- Find occurrence of pattern of m characters
- in a longer text of n characters

- Input enhancement
- Preprocess the pattern to get some information about it
- Store this info in a table
- Use this info during an actual search
  - Harpool's
  - Boye-Moore

String Matching

- Harpool's Algorithm
- Align patter with text starting from left
- Compare patter with text starting from right
- If all are equal, stop. Solution found!
- If mismatch, shift to right as efficient and as big as possible
- Consider last character of text *c* aligned with pattern

$$s_0 \quad \ldots \qquad\qquad\qquad\qquad c \quad \ldots \quad s_{n-1}$$

B A R B E R

String Matching

1. There is no c in pattern→ shift the pattern by its length

s_0 . . . S . . . s_{n-1}

$\nparallel$

B A R B E R

B A R B E R

2. Only c in pattern is the last character → shift the pattern by its length

s_0 . . . M E R . . . s_{n-1}

$\nparallel$ $\parallel$ $\parallel$

L E A D E R

L E A D E R

3. c occurs in last and middle → shift to align rightmost occurrence

s_0 . . . A R . . . s_{n-1}

$\nparallel$ $\parallel$

R E O R D E R

R E O R D E R

4. c occurs in middle but not last → shift to align rightmost occurrence

s_0 . . . B . . . s_{n-1}

$\nparallel$

B A R B E R

B A R B E R

String Matching

- Harpool's Algorithm
- Precompute shift sizes for all possible characters
- Store in a table
- Indicate shift size by:

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases}$$

- $C_{worst}(n) \in O(nm)$
- $C_{average}(n) \in O(n)$

String Matching

- Harpool's Algorithm
- Find BARBER in the given text

character c	A	B	C	D	E	F	. . .	R	. . .	Z	_
shift $t(c)$											

J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P

String Matching

- Harpool's Algorithm
- Find BARBER in the given text

character c	A	B	C	D	E	F	...	R	...	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R           B A R B E R
        B A R B E R       B A R B E R
          B A R B E R           B A R B E R
```