Day 17

SPACE AND TIME TRADE-OFFS

SORTING BY COUNTING

STRING MATCHING: HORSPOOL

Space-for-Time Tradeoffs

Input enhancement: preprocess the input and store it and use later.

Example:

Log Table

cuemath
THE MATH EXPERT

											Mean Difference								
	0	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	7	8	9
10	0000	0043	0086	0128	0170	0212	0253	0294	0334	0374	4	8	12	17	21	25	29	33	37
11	0414	0453	0492	0531	0569	0607	0645	0682	0719	0755	4	8	11	15	19	23	26	30	34
12	0792	0828	0864	0899	0934	0969	1004	1038	1072	1106	3	7	10	14	17	21	24	28	31
13	1139	1173	1206	1239	1271	1303	1335	1367	1399	1430	3	6	10	13	16	19	23	26	29
14	1461	1492	1523	1553	1584	1614	1644	1673	1703	1732	3	6	9	12	15	18	21	24	27
15	1761	1790	1818	1847	1875	1903	1931	1959	1987	2014	3	6	8	11	14	17	20	22	25
16	2041	2068	2095	2122	2148	2175	2201	2227	2253	2279	3	5	8	11	13	16	18	21	24
17	2304	2330	2355	2380	2405	2430	2455	2480	2504	2529	2	5	7	10	12	15	17	20	22
18	2553	2577	2601	2625	2648	2672	2695	2718	2742	2765	2	5	7	9	12	14	16	19	21
19	2788	2810	2833	2856	2878	2900	2923	2945	2967	2989	2	4	7	9	11	13	16	18	20
20	3010	3032	3054	3075	3096	3118	3139	3160	3181	3201	2	4	6	8	11	13	15	17	19
21	3222	3243	3263	3284	3304	3324	3345	3365	3385	3404	2	4	6	8	10	12	14	16	18
22	3424	3444	3464	3483	3502	3522	3541	3560	3579	3598	2	4	6	8	10	12	14	15	17
23	3617	3636	3655	3674	3692	3711	3729	3747	3766	3784	2	4	6	7	9	11	13	15	17
24	3802	3820	3838	3856	3874	3892	3909	3927	3945	3962	2	4	5	7	9	11	12	14	16
25	3979	3997	4014	4031	4048	4065	4082	4099	4116	4133	2	3	5	7	9	10	12	14	15
26	4150	4166	4183	4200	4216	4232	4249	4265	4281	4298	2	3	5	7	8	10	11	13	15
27	4314	4330	4346	4362	4378	4393	4409	4425	4440	4456	2	3	5	6	8	9	11	13	14
28	4472	4487	4502	4518	4533	4548	4564	4579	4594	4609	2	3	5	6	8	9	11	12	14
29	4624	4639	4654	4669	4683	4698	4713	4728	4742	4757	1	3	4	6	7	9	10	12	19
30	4771	4786	4800	4814	4829	4843	4857	4871	4886	4900	1	3	4	6	7	9	10	11	13

Space-for-Time Tradeoffs

Input enhancement use in CS:

- counting sorts
- string searching algorithms

Prestructuring: preprocess the input to make **accessing** its elements easier

- hashing
- indexing schemes (e.g., B-trees)

Dynamic programming: next chapter

Sorting by Counting

For each x = A[i], count how many other elements in A are less than x

This indicates the index of x in a sorted array.

Sort list by copying elements into a new list.

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

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 ← 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: Example

Advantages: only *n* key moves (optimal!)

Drawbacks:

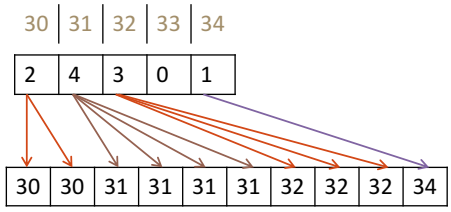
- Quadratic time efficiency to construct Count[] (worksheet)
- Requires linear amount of extra space, for Count[]

Sorting by Distribution Counting

Suppose the values come from a small interval *l*..*u* of integers, say: [30..34]

A = | 32 | 30 | 34 | 31 | 32 | 32 | 31 | 30 | 31 | 31 |

Count of each element in array A, to determine the distribution of the values:



Then construct sorted array.

Input Enhancement in String Matching

Problem: Find a *pattern* (a String of size *m*) in some *text* (a larger String of size *n*).

- Worst case O(nm)
- Average case O(n+m)

Optimizations:

- Preprocess the pattern to get some information about it
- Store this information in a table
- Then use this information during an actual search for the pattern in the text

Horspool's algorithm

It is a simplified version of the Boyer-Moore algorithm (discussed next time)

It preprocesses pattern to generate a shift table that determines how much to shift the pattern when a mismatch occurs

Always makes a shift based on the text's character *c* aligned with the last character in the pattern according to the shift table's entry for *c*

Horspool's algorithm

Consider searching for the following pattern:

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

B A R B E R

When comparing, we begin with the last character of the pattern, moving to the left.

If all the pattern's characters match, then a matching substring has been found.

If mismatch, then move the pattern to the right.

Read text L→R, match pattern R→L.

Horspool's algorithm

When moving to the right, aim for largest possible shift.

Look at character *c*:

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

B A R B E R

We will consider four cases.

Case 1: No *c*'s in the pattern. Suppose *c* is the letter S:

$s_0 \quad \ldots \quad$ S $\quad \ldots \quad s_{n-1}$

B A R B E R

B A R B E R

Shift pattern by its entire length.

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

B A R B E R

Horspool's algorithm

Case 2: There is an occurrence of *c* in the pattern, but it is not the last one.

Suppose *c* is the letter B:

$s_0 \quad \ldots \quad$ B $\quad \ldots \quad s_{n-1}$

B A R B E R

B A R B E R

Shift the pattern so that the right-most occurrence of *c* in the pattern aligns with *c* in the text.

Horspool's algorithm

Case 3: *c* happens to be the last character in the pattern but there are no *c*'s among its other m-1 characters.

Suppose *c* is the letter R:

Shift pattern by its entire length.

Horspool's algorithm

Case 4: *c* happens to be the last character in the pattern and there are other *c*'s among its other *m*-1 characters.

Suppose *c* is the letter R:

Shift the pattern so that the right-most occurrence of *c* among the first *m*-1 characters should be aligned with *c* in the text.

Shift-tables

Input enhancement: Precompute shift sizes and store them in a table

Table is indexed by all possible characters that can be encountered in a text.

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases} \quad \textbf{(7.1)}$$

For example, for the pattern BARBER, the table's entries for E, B, R and A which will be 1, 2, 3, and 4, respectively.

All other entries will be 6

Constructing Shift-tables

ALGORITHM *ShiftTable*$(P[0..m-1])$

```
//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern P[0..m − 1] and an alphabet of possible characters
//Output: Table[0..size − 1] indexed by the alphabet's characters and
//          filled with shift sizes computed by formula (7.1)
for i ← 0 to size − 1 do Table[i] ← m
for j ← 0 to m − 2 do Table[P[j]] ← m − 1 − j
return Table
```

For example, for the pattern BARBER, the table's entries for E, B, R and A which will be 1, 2, 3, and 4, respectively.

Horspool's Algorithm

ALGORITHM *HorspoolMatching*$(P[0..m-1], T[0..n-1])$

```
//Implements Horspool's algorithm for string matching
//Input: Pattern P[0..m − 1] and text T[0..n − 1]
//Output: The index of the left end of the first matching substring
//         or −1 if there are no matches
ShiftTable(P[0..m − 1])        //generate Table of shifts
i ← m − 1                      //position of the pattern's right end
while i ≤ n − 1 do
    k ← 0                      //number of matched characters
    while k ≤ m − 1 and P[m − 1 − k] = T[i − k] do
        k ← k + 1
    if k = m
        return i − m + 1
    else i ← i + Table[T[i]]
return −1
```

Horspool's Algorithm: Example

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

The actual search in a particular text proceeds as follows:

```
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
```

Runtime?