CPE 231 Algorithms

Space and Time Trade-offs

Dr. Taweechai Nuntawisuttiwong

Contents

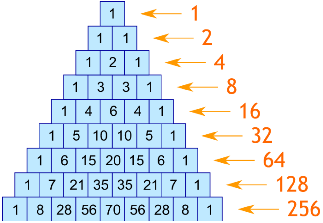
Spac and Time Trade-offs

Space and Time Trade-Offs

Johann Wolfgang von Goethe

Things which matter most must never be at the mercy of things which matter less.

- Space and time trade-offs are crucial considerations in algorithm design, impacting both theoreticians and practitioners.
- **Example**: Computing values of a function at many points. Precomputing these values and storing them in tables saves time at the cost of additional space.



Key Techniques

- **Input Enhancement**: Preprocessing input to speed up problem-solving.
- **Prestructuring**: Using extra space for faster data access.
- **Dynamic Programming**: Storing solutions to overlapping subproblems.

Input Enhancement Technique

Definition

Preprocessing or preconditioning the input to store additional information, which accelerates solving the problem later.

Examples of Algorithms Using Input Enhancement:

- Counting methods for sorting.
- Boyer-Moore algorithm for string matching.
- Horspool's simplified string matching algorithm.

Prestructuring Technique

Definition

Using extra space to facilitate faster or more flexible access to data.

Note: Structuring data before solving the problem for quicker access.

Examples of Algorithms Using Prestructuring:

- **Hashing** for efficient data retrieval.
- **Indexing with B-trees** for managing large sets of data.

Dynamic Programming and Time-Space Optimization

- Recording solutions to overlapping subproblems in a table (dynamic programming).
- **Example**: Optimizing both time and space, such as graph traversal using adjacency lists over matrices in sparse graphs.

Sorting by Counting

Sorting by Counting

- An approach to sorting that relies on counting occurrences and positions of elements rather than comparing them.
- It is especially efficient for lists of integers with a limited range of values.

Input-Enhancement Technique:

This technique preprocesses the input data by counting occurrences or comparisons, then uses this information to speed up the sorting process.

Comparison-Counting Sort Algorithm

- For each element in the array, count how many elements are smaller than it.
- This count determines the position of the element in the final sorted array.

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
```

Comparison-Counting Sort Algorithm

Example: Given the array {62, 31, 84, 96, 19, 47}

Array A[0..5]		62	31	84	96	19	47
Initially	Count[ ]						
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 A[0..5]							

Comparison-Counting Sort Algorithm

Time Efficiency:

The basic operation:

The number of basic operation:

- If the elements to be sorted are drawn from a small set of possible values, counting can be used to optimize the sorting process.

Distribution Counting

- A more general counting-based approach where the exact positions of elements in the sorted array are determined using their frequencies.

  1. **Frequency Calculation**: First, count how many times each element appears in the array.
  2. **Distribution Array**: Calculate the cumulative sum of frequencies, which tells us where the elements should be placed in the sorted array.

Distribution Counting Sort Algorithm

1. Initialize frequency array D.
2. Compute the frequency of each element.
3. Accumulate frequencies to create the distribution array.
4. Use the distribution array to place elements into their correct sorted positions.

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

Distribution Counting Sort Algorithm

Example: Given the array {13, 11, 12, 13, 12, 12}

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

Advantages of Sorting by Counting

- **Time Efficiency**: Sorting by counting can achieve linear time complexity $O(n)$ when the range of values is limited.
- **No Comparisons Needed**: It does not rely on element comparisons, making it ideal for specific scenarios (e.g., sorting integer keys).
- **Direct Placement**: Each element is placed directly in its final position, reducing the number of key moves.

Input Enhancement in String Matching

String Matching

- String matching involves finding a pattern (P) of length m within a larger text (T) of length n.
- **Basic approach**: Brute-force matching compares characters from left to right and shifts the pattern by one position after a mismatch.
  - Worst-case time complexity: $O(n \cdot m)$ On average, brute-force string matching has time complexity $O(n+m)$.

Input Enhancement Technique for String Matching

- Preprocess the pattern to extract useful information that accelerates the string matching process.
- The **Boyer-Moore algorithm** compares the pattern characters right-to-left during each trial.
- We will explore a simplified version: **Horspool's Algorithm**, which is easier to implement.

Horspool's Algorithm

- Compare pattern characters with text from right to left.
- Shift the pattern to the right based on mismatches to skip unnecessary comparisons.
- The size of the shift is determined by the character aligned against the last character of the pattern.
- Horspool's algorithm determines the shift size using a **shift table**.

$$
\begin{array}{cccccccccc}
s_0 & \ldots & & & & & c & \ldots & s_{n-1} \\
 & & \text{B} & \text{A} & \text{R} & \text{B} & \text{E} \; \text{R} & &
\end{array}
$$

Horspool's Algorithm – Handling Mismatches

1. Character not in the pattern

$$\begin{array}{ccccccccccccccc}
s_0 & \dots & & & & & \mathtt{S} & & & & & & \dots & s_{n-1} \\
 & & & & & & \not\parallel & & & & & & & \\
 & & \mathtt{B} & \mathtt{A} & \mathtt{R} & \mathtt{B} & \mathtt{E} \; \mathtt{R} & & & & & & & \\
 & & & & & & & \mathtt{B} & \mathtt{A} & \mathtt{R} & \mathtt{B} & \mathtt{E} & \mathtt{R} &
\end{array}$$

2. Character in the pattern but not at the last position

$$\begin{array}{cccccccccc}
s_0 & \dots & & & & & & \mathtt{B} & \dots & s_{n-1} \\
 & & & & & & & \not\parallel & & \\
 & & \mathtt{B} & \mathtt{A} & \mathtt{R} & \mathtt{B} & \mathtt{E} & \mathtt{R} & & \\
 & & & & \mathtt{B} & \mathtt{A} & \mathtt{R} & \mathtt{B} \; \mathtt{E} \; \mathtt{R} & &
\end{array}$$

3. Character at the last position but not found elsewhere

$$\begin{array}{cccccccccc}
s_0 & \dots & & & & \mathtt{M} & \mathtt{E} & \mathtt{R} & \dots & s_{n-1} \\
 & & & & & \not\parallel & \parallel & \parallel & & \\
 & & \mathtt{L} & \mathtt{E} & \mathtt{A} & \mathtt{D} & \mathtt{E} & \mathtt{R} & & \\
 & & & & & & & & \mathtt{L}\;\mathtt{E}\;\mathtt{A}\;\mathtt{D}\;\mathtt{E}\;\mathtt{R} &
\end{array}$$

4. Character at the last position and found elsewhere

$$\begin{array}{ccccccccccc}
s_0 & \dots & & & & & & \mathtt{A} & \mathtt{R} & \dots & s_{n-1} \\
 & & & & & & & \not\parallel & \parallel & & \\
 & & \mathtt{R} & \mathtt{E} & \mathtt{O} & \mathtt{R} & \mathtt{D} & \mathtt{E} & \mathtt{R} & & \\
 & & & & & \mathtt{R} & \mathtt{E} & \mathtt{O} & \mathtt{R} & \mathtt{D}\;\mathtt{E}\;\mathtt{R} &
\end{array}$$

The Shift Table

- Initialize all table entries to the pattern length m.
- For each character in the pattern (except the last one), compute the distance to the last character, $t(c)$, and update the table.

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 $i \leftarrow 0$ **to** $size-1$ **do** $Table[i] \leftarrow m$
for $j \leftarrow 0$ **to** $m-2$ **do** $Table[P[j]] \leftarrow m-1-j$
return *Table*

Horspool's Algorithm – Pseudocode

1. Precompute the shift table based on the pattern.
2. Align the pattern with the start of the text.
3. Compare characters from right to left:
   - If a mismatch occurs, shift the pattern based on the shift table.
   - If all characters match, return the index of the match.
4. Repeat until the pattern moves beyond the text or a match is found.

```
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

Example: Searching for the pattern BARBER in a text that comprises English letters and spaces (denoted by underscores).

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

Boyer-Moore Algorithm

- Boyer-Moore uses both **bad-symbol** and **good-suffix** rules to determine shifts.
- **Bad-Symbol Rule**: Shifts the pattern based on the character in the text that caused a mismatch.
- **Good-Suffix Rule**: Shifts the pattern based on a successful match of a suffix within the pattern itself.
- The Boyer-Moore algorithm can often shift the pattern by larger amounts compared to Horspool's algorithm.

Bad-Symbol Rule

1. Bad Symbol Not in the Pattern:
   - Shift the pattern by $t_1(S) - 2 = 6 - 2 = 4$

$$
\begin{array}{cccccccccccccc}
s_0 & \dots & & & & \mathrm{S} & \mathrm{E} & \mathrm{R} & & & & & \dots & s_{n-1} \\
 & & & & & \nparallel & \parallel & \parallel & & & & & & \\
 & & \mathrm{B} & \mathrm{A} & \mathrm{R} & \mathrm{B} & \mathrm{E} & \mathrm{R} & & & & & & \\
 & & & & & & \mathrm{B} & \mathrm{A} & \mathrm{R} & \mathrm{B} & \mathrm{E} & \mathrm{R} & &
\end{array}
$$

2. Bad Symbol in the Pattern:
   - Shift the pattern by $t_1(A) - 2 = 4 - 2 = 2$

$$
\begin{array}{cccccccccccc}
s_0 & \dots & & & & \mathrm{A} & \mathrm{E} & \mathrm{R} & & & \dots & s_{n-1} \\
 & & & & & \nparallel & \parallel & \parallel & & & & \\
 & & \mathrm{B} & \mathrm{A} & \mathrm{R} & \mathrm{B} & \mathrm{E} & \mathrm{R} & & & & \\
 & & & & \mathrm{B} & \mathrm{A} & \mathrm{R} & \mathrm{B} & \mathrm{E} & \mathrm{R} & &
\end{array}
$$

Bad-Symbol Rule

The shift size for the bad-symbol rule is calculated as

$$d1 = \max(t1(c) - k, 1)$$

where

- $t1(c)$ is the precomputed shift for the bad symbol.
- k is the number of matched characters before the mismatch occurred.
- $\max(t1(c) - k, 1)$ ensures that the shift is at least one position to avoid overlapping comparisons.

Good-Suffix Rule

- During preprocessing, the Good-Suffix Table is constructed by analyzing the suffixes of the pattern and identifying:
  - Occurrences of the suffix within the pattern.
  - The largest prefix of the pattern that matches a suffix.
- For each suffix, the table indicates how far the pattern can be shifted when that suffix is matched in the text.

Example:

k	pattern	d_2
1	ABCBAB	
2	ABCBAB	
3	ABCBAB	
4	ABCBAB	
5	ABCBAB	

Combining Bad-Symbol and Good-Suffix Shifts

- When both the Bad-Symbol Rule and the Good-Suffix Rule apply, the algorithm shifts the pattern by the larger of the two shift values.
- The shift size is computed as:

$$d = \max(d1, d2)$$

Note: By using both rules, the Boyer-Moore algorithm can maximize the shift size, minimizing the number of comparisons and achieving optimal performance.

Boyer-Moore Algorithm – Example

Example: Searching for the pattern BAOBAB in a text.

The bad-symbol table:

c	A	B	C	D	...	O	...	Z	_
$t(c)$									

The good-suffix table:

k	pattern	d_2
1	BAOBAB	
2	BAOBAB	
3	BAOBAB	
4	BAOBAB	
5	BAOBAB	

Boyer-Moore Algorithm – Example

Example: Searching for the pattern BAOBAB in a text.

B E S S _ K N E W _ A B O U T _ B A O B A B S

Efficiency Comparison

Horspool's Algorithm:

- Simpler to implement.
- Performs well for random texts, $O(n)$.
- Worst-case time complexity: $O(nm)$.

Boyer-Moore Algorithm:

- More complex, but can achieve faster shifts.
- Worst-case time complexity: $O(n)$.

Hashing

Hashing

- Hashing is a technique used to implement **dictionaries** efficiently, allowing for operations like searching, insertion, and deletion.
- A dictionary consists of a set of elements (e.g., student records, citizen records) where each element has a **key** used for identification.
- Map keys to a **hash table** using a **hash function**.
- **Hash Table**: A one-dimensional array of size m, where each element is stored at an index determined by the hash function.

Hash Function

A hash funciton needs to satisfy somewhat conflict rquirements:

- A hash table's size should not be excessively large compared to the number of keys.
- A hash function needs to distribute keys among the cells of the hash table as evenly as possible.
- A hash function has to be easy to compute.

Example:

K is and integer:

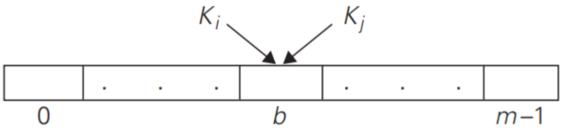
$$h(K) = K \mod m$$

K is a character string $c_0 c_1 \ldots c_{s-1}$ and $ord(K)$ is position in the alphabet:

$$h(K) = \left(\sum_{i=0}^{s-1} ord(c_i) \right) \mod m$$

Hash Collisions

- Occurs when two or more keys are assigned the same hash value, i.e., they map to the same location in the hash table.
- **Open Hashing** (Separate Chaining):
  - Uses linked lists to store multiple keys hashed to the same table index.
- **Closed Hashing** (Open Addressing):
  - Stores all keys directly in the hash table, probing to find the next available spot when a collision occurs.



Open Hashing (Separate Chaining)

- Each cell in the hash table points to a linked list that contains all the keys hashed to that index.
- If a collision occurs, the key is simply added to the linked list at that table index.

Example: Given size of hash table is m:

$h(A) = 1 \mod 13 = 1$
$h(FOOL) = (6 = 15 + 15 + 12) \mod 13 = 9$
$h(ARE) = (1 + 18 + 5) \mod 13 = 11$
$h(SOON) = (19 + 15 + 15 + 14) \mod 13 = 11$

Open Hashing (Separate Chaining)

Example:a hash table construction with separate chaining.

keys	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
hash addresses	1	9	6	10	7	11	11	12

0	1	2	3	4	5	6	7	8	9	10	11	12

Efficiency of Open Hashing

- **Load factor** $\alpha = n/m$: Ratio of the number of keys n to the size of the table m.
- Average number of chained links:
  - Successful searches: $S \approx 1 + \frac{\alpha}{2}$.
  - Unsuccessful searches: $U = \alpha$.
- Performance depends on the length of the linked lists.
- $O(1)$ in the average case if the number of keys n is about equal to the hash table's size m.

Closed Hashing (Open Addressing)

- All keys are stored within the hash table itself, without linked lists.
- **Linear Probing**: When a collision occurs, the algorithm checks the next available slot in the table, continuing until an empty cell is found.
- The table is treated as circular, meaning if the end is reached, probing wraps around to the beginning.

Closed Hashing (Open Addressing)

Example: A hash table construction with linear probing.

keys	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
hash addresses	1	9	6	10	7	11	11	12

0	1	2	3	4	5	6	7	8	9	10	11	12
	A											
	A								FOOL			
	A					AND			FOOL			
	A					AND			FOOL	HIS		
	A					AND	MONEY		FOOL	HIS		
	A					AND	MONEY		FOOL	HIS	ARE	
	A					AND	MONEY		FOOL	HIS	ARE	SOON
PARTED	A					AND	MONEY		FOOL	HIS	ARE	SOON

Closed Hashing (Open Addressing)

Example: Search the word “LIT” and “KID”.

Efficiency of Closed Hashing

- Average number of access the hash table:
  - Successful searches: $S \approx \dfrac{1}{2}\left(1 + \dfrac{1}{1-\alpha}\right)$.
  - Unsuccessful searches: $U \approx \dfrac{1}{2}\left(1 + \dfrac{1}{(1-\alpha)^2}\right)$.

α	$\frac{1}{2}(1 + \frac{1}{1-\alpha})$	$\frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$
50%	1.5	2.5
75%	2.5	8.5
90%	5.5	50.5

Problems with Closed Hashing

- **Clustering**: Long sequences of filled cells can slow down searching, insertion, and deletion.
- **Lazy Deletion**: A key is marked as deleted rather than removed to avoid breaking the probing sequence.