Chapter 1: Analysis of Algorithms

1.1 General Overview: Problems, Algorithms and Complexity

- A computation problem
  - A problem may be from any application areas, OS, database, graphic, etc
  - It must be well defined, i.e. include desired input/output
- Example: Sorting Problem
  Input: A sequence of n numbers $a_1, a_2, ..., a_n$
  Output: A permutation $(b_1, b_2, ..., b_n)$ such that $b_1 \leq b_2 \leq ... \leq b_n$

- Major goal: To find the most efficient (best running time) algorithm for solving a problem.

- In general, running time is expressed in terms of a single variable, the size of the problem instance.

  Example:

  For most scheduling problems, the number of tasks, the number of processors and the total processing time.

  For most of the graph problem, the number of vertices, the number of edges and the total weight on vertices/edges etc.

- Time complexity of algorithms:
  - polynomial time algorithm ("efficient algorithm")
  - exponential time algorithm ("inefficient algorithm")

  Example: $n^5$ and $2^n$, if $n = 50$, 5 mins and 35 years
We study two major groups of problems:

- Many problems can be solved in polynomial time, i.e. $O(n \log n)$ or $O(n^2)$
  Example: sorting and searching algorithms

- Some other problems such as traveling salesperson problem, the best algorithms (up to now) are exponential time algorithms.

  Traveling Salesperson problem: Given a directed graph $G$ where each edge has a positive weight. Find a cycle which includes all vertices in $G$ such that the total weight of the cycle is minimum.

  Of course, we still would like to find more efficient algorithms to solve problems in this group. So that, we can solve the problem by using computer programs.

Some problems that are not included in our study:

- Some problems are unsolvable (undecidable) by using computer algorithms
  Example: Given an arbitrary program and an arbitrary input to that program, we cannot decide whether or not the program will eventually halt.

- Some problems require exponential number of solutions.
  Example: Print all subsets of $n$ numbers.

1.2 Designing & Analyzing Algorithms

Designing Algorithm

- Use data structures & some common techniques

- Basic Data Structures:
  arrays/record/linked lists structures; stacks/queues;
  trees - binary tree, heaps, etc
  graphs - DFS, BFS, spanning trees
- Some common approaches/techniques:
  - Recursive algorithms & Divide and conquer algorithms
  - Dynamic programming algorithms
  - Greedy algorithms
  - Backtracking algorithms
  - Branch and bound algorithms
  - Brute force algorithms
  - Randomized algorithms

- For divide-and-conquer approach
  - divide - break the problem into a number of subproblems
  - conquer - solve subproblems recursively. If subproblem sizes small enough, then, just solve subproblems in a straightforward manner.
  - combine: combine solutions to the subproblems into the solution for the original problem

Example: Merge Sort – to sort an array of n elements

- divide array into two subarrays each with n/2 elements
- unless a subarray is sufficiently small; otherwise, recursively sort elements in subarray (by repeating “divide” step)
- combine sorted subarrays by merging them into a single sorted array.

- Algorithm:

```plaintext
Merge(A,u,m,v) {
    merge two sorted lists A[u, u+1, ..., m] & A[m+1, m+2, ..., v]
}

Mergesort(A,p,r) {
    If (p=r) return;
    q=(p+r)/2;
    Mergesort(A,p,q);
    Mergesort(A,q+1,r);
    Merge(A,p,q,r);
}
```
• Analyzing Algorithm

• Time complexity; Analysis of correctness; Space complexity

• Running Time of an algorithm

  ▪ number of primitive operations or steps executed
  ▪ assume machine independent, constant time per step
  ▪ depends on input size (e.g. number of input elements in sorting algorithm)
  ▪ sometime, also depends on input itself, e.g. if input list is in sorted order, insertion sort needs minimum number of steps
  ▪ define a running time function for any input instance, e.g. \( T(n) = c_1 n + (c_2 + c_3)[(n)(n+1)/2] + c_4 \)
  ▪ kinds of analysis :
    ▪ worst case : \( T(n) = \text{max time on any input of size } n. \)
      most popular analysis case
    ▪ average case : \( T(n) = \text{average time over all inputs of size } n \)
      sometime it is used; very difficult to analyze
    ▪ best case : never use

  ▪ Problem : above running time analysis is too complex.
  Solution : use asymptotic analysis

  ignore constants; drop low-order terms; interest to look at the growth rate as \( n \) getting bigger; use big O notation (upper bound)

  \( T(n) = c_1 n + (c_2 + c_3)[(n)(n+1)/2] + c_4 \) is \( O(n^2) \)
• Running time analysis of Mergesort

For the algorithm contains a recursive call to itself, use recurrence equation

\[ T(n) = \begin{cases} 
O(1) & \text{if } n=1 \\
2T(n/2)+O(n) & \text{if } n \geq 1 
\end{cases} \]

\[ T(n) = 2T(n/2) + O(n) \]
\[ = 2[2T(n/4) + O(n/2)] + O(n) \]
\[ = 4T(n/4) + 2O(n) \]
\[ = 4[2T(n/8) + O(n/4)] + 2O(n) \]
\[ = 8T(n/8) + 3O(n) \]
\[ \ldots \text{ in general } T(x) = \text{constant for small number } x \]
\[ = nT(1) + \log n O(n) \]
\[ = O(n \log n) \]
• Correctness of an algorithm
  - Provide a proof to show that the algorithm does what it is supposed to do.
  - For a divide and conquer algorithm, usually, use proof by induction.
  - Example: Correctness of Mergesort

    It is obvious that the algorithm works correctly when there is only 1 element.

    Assume that the algorithm works for any number of elements \( k < n \).

    For \( n \) element, the array is divided into 2 subarrays, each with \( (n/2) \) elements.

    By the induction hypothesis, each subarray is sorted correctly.

    It is easy to see that merge() function merges two sorted subarray correctly.

• Space complexity of an algorithm

  • Amount of memory a program needs to run to completion
  
  • Storage include: code, variables space, stack space etc

  • In general, only look at additional storage that is needed during program execution

  • For Mergesort,

    Input storage is \( O(n) \) to hold array of \( n \) elements
    Additional \( O(\log n) \) storage to hold stack variables for recursive calls
    Need additional \( O(n) \) storage in merge() to hold sorted data
You should review the following topics (1.3-1.4, Chap. 2 and Chap. 3):

1.3 Asymptotic Notation

1.4 Recurrences

Chapter 2: Sorting Algorithms: Quick Review

2.1 Popular sorting algorithms

- Insertion sort; Selection sort; Bubble sort
  
  Average case, worst case: $\Theta(n^2)$

- Merge sort; Heap sort
  
  Average case, worst case: $\Theta(n \lg n)$

- Quick sort
  
  Use divide and conquer strategy

  ```
  Quicksort(A,p,r) {
    if(r > p) {
      q=partition(A,p,r)
      Quicksort(A,p,q-1)
      Quicksort(A,q+1,r)
    }
  }
  ```

  Main step: Partition(A,p,r) // $\Theta(n)$

  Where $A[q]$ is in place.

  Worst case $T(n) = T(n-1) + T(0) + \Theta(n) = \Theta(n^2)$
- Average case: $\Theta(n \ lg \ n)$

- Can we improve Quicksort() algorithm such that the worst case running time is $\Theta(n \ lg \ n)$?

  Yes, use $O(n)$ liner time order statistics algorithm (a.k.a. median-of-medians algorithm) to find a good partition element, i.e. the median, before calling Partition(). The Partition() splits n elements to two $n/2$ elements. The running time is $T(n) = 2T(n/2) + \Theta(n)$. Note: This is not a practical method.

2.2 Linear time sorting algorithms

- So far, we have covered several comparison sorting algorithms, i.e. using comparisons of elements to obtain the sorted order.

- The worst case performance: $O(n \ lg \ n)$ or $O(n^2)$

- Can we do better? What is the lower bound for comparison sorting methods? i.e. Given the best comparison sorting method, what is the worst case running time?

- Theorem: the lower bound of all comparison sorting algorithms is $O(n \ lg \ n)$

- Heapsort, mergesort and quicksort are asymptotically optimal comparison sorts.

- So, if there any linear time non-comparison sorting method? Yes.
• Counting Sort

• Assume the numbers to be sorted are integers in the range of 1 to k

CountingSort (A,B,k)
Input : array A with n elements, where each A[i] is an element of \{1,2,...,k\}
Output : sorted list in array B

/* need additional storage : array C[1..k] */
step 1 : set C[i] = 0 for i = 1 to k \(O(k)\)
step 2 : for j = 1 to n /* count # elements equal to i using array C */ \(O(n)\)
    C[A[j]] = C[A[j]]+1
step 3 : for j = 2 to k /* count # of elements <= j */ \(O(k)\)
    C[j] = C[j]+ C[j-1]
step 4 : for j = n downto 1 /* compute sorted list in array B */ \(O(n)\)
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]]-1

Example :

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[i]</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

\[k = 6\]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

index of C

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>B[i]</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

• O(n+k) if n > k, then O(n) . Note : Counting sort is a stable sorting alg.

• If k is very large k >> n then counting sort is not useful... use Radix sort
• Radix sort

• Let assume each number is at most d digits, each digit is 0 1 2 ...9
• May assume 0s are packed at the beginning of numbers < d digits
• For i = d to 1

\[
\text{use a counting sort to sort array on } i^{th} \text{ digit}
\]
/* if it is not a stable algorithm then we may get wrong output */

• Example :

| 329 | 720 | 720 | 329 |
| 457 | 355 | 329 | 355 |
| 657 | 3th | 436 | 2nd | 436 | 1st | 436 |
| 839 | => | 457 | => | 839 | => | 457 |
| 436 | 657 | 355 | 657 |
| 720 | 329 | 457 | 720 |
| 355 | 839 | 657 | 839 |

• Running time : each pass O(n+k) /* k is small now */

\[
\text{total : O(dn + dk)}
\]
when d is a constant and k is small, we have O(n)

• Is Radix sort better than Counting sort? may be….

• for example : k = n² then running time for counting sort = O(k) = O(n²)

• for radix sort, numbers can be stored using 2 lg n bits per number (binary numbers in computer)

\[
\text{i.e. [ lg n bits | lg n bits]}
\]
using 2 pass, each pass needs k=n bins (for lg n bits)
each pass is O(n), total = O(2n) = O(n)

• Suggestion : use Radix sort when n >= 2000
Chapter 3 Common Data Structures

3.1 Priority Queue

<table>
<thead>
<tr>
<th></th>
<th>Insertion</th>
<th>Deletion (largest key)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted array based</td>
<td>O(1)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Unsorted pointer based</td>
<td>O(1)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Sorted array based</td>
<td>O(N)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Sorted pointer based</td>
<td>O(N)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Binary Search Tree (balanced)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>Maxheap</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>

3.2 Tables

<table>
<thead>
<tr>
<th></th>
<th>Insertion</th>
<th>Deletion</th>
<th>Retrieval</th>
<th>Traversal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted array based</td>
<td>O(1)</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Unsorted pointer based</td>
<td>O(1)</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Sorted array based</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(log n)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Sorted pointer based</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Binary Search Tree (balanced)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

3.3 Hash Tables

- Supports the following dictionary operations
  - Insert(x,T) : insert new record x into T
  - Delete(x,T) : delete record x from T
  - Search(x,T) : retrieve record x from T
- Assume each key is equally likely to be hashed into any slot
  Let n = total number of keys and m = number of slots in the table
  \[
  \text{load factor } \alpha = \frac{n}{m}; \quad \text{i.e. how full the table is.}
  \]
  - Separate chaining :
    \[
    1 + \alpha / 2 \quad \text{for a successful search (at least one comparison)}
    \]
    \[
    \alpha \quad \text{for an unsuccessful search}
    \]
3.4 Binary Search Tree

- Search tree operations, mostly $O(\log n)$: search, min, max, predecessor, successor, insert and delete

- Main property of a binary search tree:

  Let $x$ be a node; $y$ be any node in the left subtree of $x$; $z$ be any node in the right sub-tree of $x$, then $\text{key}[x] \geq \text{key}[y]$ and $\text{key}[x] \leq \text{key}[z]$

- Example:

![Binary Search Tree Diagram]

- Operations

  - to visit all nodes $O(n)$ - inorderWalk: 234578; preorderWalk: 532478; postorderWalk: 243875
  - search: $O(h)$
  - minimum (left most leaf): $O(h)$
  - maximum (right most leaf): $O(h)$
  - insertion (start from root, go down to right location): $O(h)$
  - sorting (insert n nodes; inorderWalk): $O(n \cdot h)$
  - deletion (need to consider few cases): $O(h)$
• Problem: most operations are $O(h)$

Worst case: $h = n \Rightarrow$ not better than linked list

Solution: try to maintain smallest height, i.e. height = $O(\log n)$

Method:
• restructure tree if necessary to maintain $O(\log n)$ height
• only need to worry about delete() and insert()
• balanced tree: red-black trees & AVL trees (binary tree); B-trees (may have >2 children)
• for query: min, max, successor, search etc; running time: $O(\log n)$

3.5 Introduction to AVL Trees

○ The subtrees of each node differ by at most 1 in their height.
○ Maintain height of $O(\log n) \Rightarrow O(\log n)$ in searching
○ Deletion and insertion take $O(\log n)$
○ Each node maintain a balance factor (= height of right subtree - height of left subtree). Example:
o Each node in AVL tree has a balance factor of –1, 0 or 1

o Only insert() and delete() operations change the AVL tree structures

o Insert() : Node is inserted into AVL Trees in the same manner as an ordinary binary search tree, i.e. nodes are always inserted as leaf nodes

After insertion, travels back (modify search strategy for this) along the path it took to find the point of insertion, and checks the balance at each node on the path.

If a node is found to be unbalanced (that is, it has a balance factor of either -2 or +2), then a rotation is performed to maintain it as AVL tree.

Note : Can be showed that only one (single or double ) rotation is needed.

o delete() : Node is deleted from AVL Trees in the same manner as an ordinary binary search tree

After deletion, travels back along the path it took to find the point of deletion, and checks the balance at each node on the path.

Note : Can be showed that multiple rotations may be needed.
There are four types of rotations:

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>Type of rotations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>-1</td>
<td>-</td>
<td>Single right rotation</td>
</tr>
<tr>
<td>-2</td>
<td>+1</td>
<td>-</td>
<td>Double (left, then right) rotations</td>
</tr>
<tr>
<td>+2</td>
<td>-</td>
<td>+1</td>
<td>Single left rotation</td>
</tr>
<tr>
<td>+2</td>
<td>-</td>
<td>-1</td>
<td>Double (right, then left) rotations</td>
</tr>
</tbody>
</table>

![Diagram showing tree rotations](image)
Example: Insert 55
We will need a double rotation to adjust the above tree.

1\textsuperscript{st} rotation: left rotation at 30 (i.e. A=30, B=50)
2\textsuperscript{nd} rotation : right rotation at 80 (i.e. B=80, A=50)

- There are many websites with tutorial on AVL tree implementation.
3.6 Graphs

- $G = (V, E)$, where $V$ is a set of vertices and $E$ is a set of edges
- $|E| = O(|V|^2)$, commonly, use $E$ and $V$ instead of $|V|$ and $|E|
- Running time: use sizes of $V$ and $E$
- Graph searching/traversals
  - Goal: to explore vertices and edges of directed or undirected graphs

**Breadth-First Search** // visit all vertices adjacent to a vertex before going forward

```java
bfs(in v:vertex)
{
    q.createQueue()
    mark v as visited   // may use an array Visited[]
    q.enqueue(v)       // add v into q

    while(!q.isEmpty())
    {
        d.dequeue(w)   // delete 1st element w from Queue
        for (each unvisited vertex u adjacent to w)
        {
            mark u as visited
            q.enqueue(u)
        }
    }
}
```

Running time: at most $V$ loops and visit all edges $\Rightarrow O(V+E)$
Note: this is linear time for graph since the size of adjacency list is $O(V+E)$
Example:

Assume vertices in adjacency list are stores in alphabetical order.

The order of visiting is:
a, b, c, d, e, f, g, h (do not reach i and j)

Assume directed graph (edges point downward direction).
The order if visiting is:
a, b, c, d, e, f, g, h (do not reach i and j)

**Depth-First Search** // visit all vertices that are reachable from v

```java
dfs(in v:vertex)
{
    // Traverses a graph beginning at vertex v using dfs
    mark v as visited // may use an array visited[] to indicate this
    for (each unvisited vertex u adjacent to v)
        dfs(u)
}
```

Assume vertices in adjacency list are stores in alphabetical order.

The order of visiting is:
a, b, d, h, e, f, c, g (do not reach i and j)

Assume directed graph (edges point downward direction).
The order if visiting is:
a, b, d, h, e, c, f, g (do not reach i and j)

- Running time: O(n+e) if we are using adjacency list, O(n²) if we are using adjacency matrix
- Note: dfs() algorithm can be used (with minor modification) to determine many problems such as “whether a graph has a cycle” and “connected components of a graph”
3.7 Application of Graphs

(i) Connected Components

- Is an undirected graph connected?

Just use DFS(v) or BFS(v) starts with any node v. At the end of DFS() or BFS(), check if there are any unvisited nodes. If yes $\rightarrow$ “not connected”; otherwise, all nodes are visited $\rightarrow$ “connected”

- Get all connected components of an undirected graph.

Assume n nodes are labeled 0,1,2,…, n-1

for (j=0; j<n, j++)
    if ( node j is not visited) DFS(j);

Example :

Note: Each time DFS() is called, it visits all nodes in the same connected component. The “for loop” checks to make sure all nodes are visited. The total time is O(n+e) for adjacency list.
(ii) Cycle detection

- Modify dfs() to detect if there is a cycle in a given graph G=(V,E)

- For undirected graphs

  Idea: A cycle in undirected graph has 3 or more edges. If a vertex v is trying to visit a visited vertex u and DFS predecessor of v != u, then “find a cycle”.

  dfs(in v:vertex)
  {
      visited[v] = True // mark v as visited
      for (each vertex u adjacent to v)
          if (visited[u] == true) {
              if (dfs_predecessor[v] != u) then “cycle”
          } else {
              dfs_predecessor[u]=v
              dfs(u)
          }
  }

- For directed graphs

  Idea of the algorithm: At any time during dfs(v), all the nodes for which inProgress is true, form a path. Thus, if the next node u has inProgress true, there already must have been a path from v to v, and therefore a cycle is formed.

  for each v in V {
      visited[v] = false;
      inProgress[v] = false; // add this to track progress …
  }

  // Do the search
  for each v in V
      if (visited[v] == 0) {
          if (cycle_dfs(v)) done! // cycle msg is printed in cycle_dfs()
      }
bool cycle_dfs(Vertex v) {
    visited[v] = true;
    inProgress[v] = true;

    for (each vertex u adjacent to v)
        if (inProgress[u])
            print "detect a cycle!", return true
        else if (visited[u] == false)
            if (cycle_dfs(u))
                return true
    inProgress[v] = false;
    return false;
}

// may consider to merge inProgress[] & visited[] to

Example 1 (no cycle should be declared):

```
C → B
  ↓   
  A → E
```

Example 2 (cycle A, C, B, A should be declared):

```
C → B
  ↓   
  A → E
```
3.8 : String Matching

Reference :

- *Introduction to Algorithms by Cormen*, Leiserson, Rivest and Stein
- *Algorithms*, By Johnsonbaugh and Schefer

3.6.1 Simple string matching algorithm

Classical Problem Statement : Given a pattern \( p \) and a text \( t \). We want to find an occurrence of \( p \) in \( t \).

Terminology :

Size of \( p = |p| = m \) and size of \( t = |t| = n \), assume \( m \leq n \)

Let \( p[i...j] \) = substring of \( p \) from position \( i \) to position \( j \).  
\( p = p[0...m-1] \).

Let \( t[i...j] \) = substring of \( t \) from position \( i \) to position \( j \).  
\( t = t[0...n-1] \).

Simple Algorithm :

Input : \( p \) and \( t \)  
Output : smallest index \( i \) such that \( t[i...i+m-1] = p \), or \(-1\) if no such index exists.

\[
i=0 \quad \text{// current starting position of a substring in } t
\]
\[
\text{while (i+m<=n) } \{
\quad j=0 \quad \text{// current position of chars } t[i+j] \& p[j]
\quad \text{while (t[i+j] == p[j]) } \{
\quad \quad j++
\quad \quad \text{if (j >= m) return i } \quad \text{// found a match}
\quad \}
\quad i++
\}
\]
return \(-1\);
Example: Let $p = 001$ and $t = 01001$

\[
\begin{array}{ccc}
    i=0 & j=0 & j=1 \\
    p & 001 & 001 \\
    t & 01001 & 01001 \\
\end{array}
\]

------------------------------------------------------------------------

\[
\begin{array}{ccc}
    i=1 & j=0 \\
    p & 001 \\
    t & 01001 \\
\end{array}
\]

------------------------------------------------------------------------

\[
\begin{array}{cccc}
    i=2 & j=0 & j=1 & j=2 \\
    p & 001 & 001 & 001 \\
    t & 01001 & 01001 & 01001 \\
\end{array}
\]

------------------------------------------------------------------------

\[
\begin{array}{cccc}
    i=3 & j=0 & j=1 & j=2 \\
    p & 001 & 001 & 001 \\
    t & 01001 & 01001 & 01001 \\
\end{array}
\]

Return 3

\[
\begin{array}{cccc}
    i=3 & j=0 & j=1 & j=2 \\
    p & 001 & 001 & 001 \\
    t & 01001 & 01001 & 01001 \\
\end{array}
\]

Running Time: $O((n-m+1)*m) = O(m*n)$
3.6.2 The Knuth-Morris-Pratt algorithm

- Goal: To improve the previous algorithm to $O(n+m)$

- examples:

(a) $p = \text{knot}$
   $t = \text{knowledge}$

What should we do when ‘t’ does not match ‘w’?
“kno” contains only one ‘k’
we can shift pattern $p$ by $|\text{kno}| = 3$ positions to continue at “wledge”

(b) $p = \text{papapar}$
   $t = \text{papapapapapapap...}$

What should we do when ‘r’ does not match ‘p’?
“pappa” contains ‘pa’ at the end which match ‘pa’ at the front
we can shift pattern $p$ by 3 positions and continue at “pappappap...”

(c) $p = \text{papapax}$
   $t = \text{papapax...}$

Notes: $p[0...4] = t[0...4]$; $st[4] = 2$; $p[2...4] = t[2...4] = p[0...2]$

- Need to compute a shift_table[$k$] for $p$, $k = -1, 0, 1, 2, ..., m-1$
  // note: shift_table[m-1] is not required

  shift_table[$k$] = $s$ means:
  - $p[0...k] = t[i...i+k]$ and $p[k+1] \neq t[i+k+1]$
  - shift $p$ forward by $s$ chars (minimum shift)
  - $p[0...k-s] = t[i+s...i+k]$ // OR $p[0...k-s] = p[s...k]$
  - continue to match $p[k-s+1]$ with $t[i+k+1]$ etc.
To build shift table:

- minimum shift is one position
- shift_table[-1] = 1 // does not match p[0]
- shift_table[0] = 1 // match p[0] but does not match p[1]

For each k > 0:

- Find the smallest s > 0 such that p[0…k-s] = p[s…k], shift_table[k] = s
- if there is no s such that p[0…k-s] = p[s…k], then shift_table[k] = k+1


\[P[0] = p[6] \quad // s = 6\]
\[p[0…2] = p[4…6] \quad // s = 4\]
\[p[0…4] = p[2…6] \quad // s = 2\]

Smallest s is 2, so shift_table[6] = 2


There is no s such that p[0…6-s] = p[s…6]
so shift_table[6] = 7

- Example: shift_table for “pappar”

<table>
<thead>
<tr>
<th>k</th>
<th>Shift_table[k]</th>
<th>p</th>
<th>a</th>
<th>p</th>
<th>p</th>
<th>a</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

28
• Knuth-Morris-Pratt algorithm:

Input: p and t
Output: smallest index i such that t[i...i+m-1] = p, or -1 if no such index exists.

Compute shift_table[]; // will cover this later O(m)

i=0    // current starting position of a substring in t
j=0    // matching position, i.e. t[i+j] with p[j]
while (i+m<=n) {
    while (t[i+j] == p[j]) {
        j++
        if (j >= m) return i    // find a match
    }
    i = i + shift_table[j-1]  // update i and j so that :
j = max(j - shift[j-1], 0)  // t[i...i+j-1] = p[0...j-1]
}
return -1;

• Example: Let p = pappar and t = pappappapparr

i = 0    j = 0...5    \rightarrow    i = 3    j = 2

\underline{pappar}
\underline{pappappapparr}

i = 3    j = 2...5    \rightarrow    i = 6    j = 2

\underline{pappar}
\underline{pappappapparr}

i = 6    j = 2...6    \rightarrow    return i = 6

\underline{pappar}
\underline{pappappapparr}
• **Running Time : O(m+n)**

Compute shift_table[] is O(m) // will look at this later

Total steps : i+j (non-decreasing) values + i (increasing) values

i is at most n steps, i+j >= i

→ at most n steps

→ O(n)

• **Algorithm to compute shift_table :**

Note : shift_table[k] is the smallest s > 0 such that p[0...k-s] = p[s...k]

Input : p
Output : shift_table[]

Recall :

- minimum shift is one position
- shift_table[-1] = 1  // does not match p[0]
- shift_table[0] = 1  // match p[0] but does not match p[1]

For each k > 0 :

- Find the smallest s > 0 such that p[0...k-s] = p[s...k], shift_table[k] = s
- if there is no s such that p[0...k-s] = p[s...k], then shift_table[k] = k+1
shift_table[0] = shift_table[-1] = 1

i=1; j=0 ;
while (i+j< m) {
    if (p[i+j] == p[j]) {
        shift_table[i+j] = i        // p[i...i+j] = p[0...j]
        j++
    } else {
        if (j == 0) shift_table[i] = i+1    // p[i] != p[0]
        // p[i...i+j-1] = p[0...j-1] & p[i+j] != p[j]
        // if shift_table[j-1] = s, update i and j
        // ⇒ p[i+s...i+j-1] = p[0...j-1-s]

        i = i + shift_table[j-1]
        j = max(j – shift[j-1], 0)        // & update j position
    }
}

- Correctness of KMP shift table computation (Optional)

We need to show that the following statements are always true before and after the while() loop (i.e. a loop invariant) (⇒ the shift table is computed correctly) :

(a) st[k] are correct for all -1 <= k < i+j
(b) p[0...j-1] = p[i...i+j-1] and    // ⇒ st[i] = ... = st[i+j-1] <= i
(c) p[0...j-1+(i-e)] != p[e...i+j-1] for 0 < e < i    // ⇒ st[i] = ... = st[i+j-1] >= i

0   j-1   j-1+(i-e)   e   i   i+j-1

| X | Y | ... | Z | X | ... |

i.e. XY != ZX for any |Y| = |Z| > 0 ⇒ cannot match before i
Case I: At the beginning i=1, j =0, before entering the loop

- st[k] is correct for st[-1] and st[0]  \( \rightarrow \) (a) is OK
- p[0…-1] and p[1..0]  \( \rightarrow \) both empty strings  \( \rightarrow \) (b) is OK
- OK, no e in between 0 < e < 1  \( \rightarrow \) (c) is OK

Case II: if (p[i+j] == p[j]) { st[i+j] = i ; j = j+1}

// show (a),(b) and (c) are true after this case

Let j’ = j+1 and i’ = i

- p[0…j-1] = p[i…i+j-1]  \( \rightarrow \) p[0…j’-1] = p[i…i’+j’-1]  \( \rightarrow \) (b) is OK

- XY != ZX  \( \rightarrow \) XYu != ZXv where u, v are any two chars
  \( \rightarrow \) p[0…j’-1+(i’-e)] != p[e… i’+j’-1] for 0 < e < i’
  \( \rightarrow \) (c) is OK

- from (b) st[i+j] <= i  and from (c) st[i+j] >= i
  \( \rightarrow \) st[i+j] = i  \( \rightarrow \) (a) is OK

Case III: if (p[i+j] != p[j]) {
  if (j==0) st[i] = i+1
  i = i + st[j-1] and j = max(j-st[j-1],0)
}

// show (a),(b) and (c) are true after this case

- (**) are correct, since st[k] is defined for all k < i+j
  \( \rightarrow \) update i and j with st[j-1] i.e. min shift to continue matching
  \( \rightarrow \) (b) and (c) are OK

- (*) j != 0  \( \rightarrow \) (a) is OK, since we do not compute st[i+j]
- (*) j == 0  \( \rightarrow \) p[i] != p[0]  \( \rightarrow \) st[i] != i
  from (c) \( \rightarrow \) st[i] >= i
  \( \rightarrow \) st[i] = i+1  \( \rightarrow \) (a) is OK
• Running Time: \(O(m)\)

You may use the same argument from last page to get \(O(m)\)

• Example: Let \(p = pappar\)

\[
\text{shift_table[-1]} = \text{shift_table[0]} = 1
\]

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
p[] & p & a & p & a & r
\end{array}
\]

\[
\begin{array}{lll}
i = 1 & j = 0 & p[1] \neq p[0] \rightarrow \text{shift_table}[1] = 2 \\
i = 2 & j = 0 & p[2] = p[0] \rightarrow \text{shift_table}[2] = 2 \\
i = 3 & j = 0 & p[3] = p[0] \rightarrow \text{shift_table}[3] = 3 \\
i = 5 & j = 0 & p[5] \neq p[0] \rightarrow \text{shift_table}[5] = 6
\end{array}
\]

• Example: Let \(p = papapaxy\)

\[
\text{shift_table[-1]} = \text{shift_table[0]} = 1
\]

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
p[] & p & a & p & a & p & a & x & y
\end{array}
\]

\[
\begin{array}{lll}
i = 1 & j = 0 & p[1] \neq p[0] \rightarrow \text{shift_table}[1] = 2 \\
i = 2 & j = 0 & p[2] = p[0] \rightarrow \text{shift_table}[2] = 2 \\
i = 6 & j = 0 & p[6] \neq p[0] \rightarrow \text{shift_table}[6] = 7 \\
i = 7 & j = 0 & p[7] \neq p[0] \rightarrow \text{shift_table}[7] = 8
\end{array}
\]
### 3.6.3 The Rabin-Karp algorithm

- **Worst Case Running Time**: $O(m \times n)$

- **Average (or expected) running time**: $O(m+n)$

- Assume $p$ and $t$ are strings of $\{0,1\}$ only

- For text string $t$, there are $(n-m+1)$ sub strings of size $m$
  i.e. $t[0...m-1], t[1...m], t[2...m+1], ..., t[n-m...n-1]$

- Define a hash function $h()$ to compute a fingerprint table for $t$
  
  $h()$ takes any input string of size $m$ and returns a value in $\{0,1, ..., q-1\}$
  i.e. $h(s) \rightarrow \{0,1,...q-1\}$ and $|s| = m$

  Compute fingerprint[i] i.e. $h(t[i...i+m-1])$ for $i = [0,n-m]$ // $O(n)$

- Hash function should distribute $(n-m+1)$ strings evenly into $q$ numbers
  
  $\Rightarrow$ There are $(n-m+1)/q$ strings in each number $[0...q-1]$
  $\Rightarrow$ only $(n-m+1)/q$ string with same value as $h(p)$

  $\Rightarrow$ Compare $(n-m+1)/q$ strings with $p$, total time is $((n-m+1) \times m)/q$
  $\Rightarrow$ If $q > m$, $((n-m+1) \times m)/q = O(n)$
Example : p = 010100, t = 0010110101001010011

\[ h(i) = h( t[i\ldots i+m-1] ) = \sum_{j=0}^{m-1} (t[i+j] \times 2^{m-1-j}) \mod q \quad \text{//} \, j=0 \ldots \, m-1 \]

for \(|p| = m = 6\) and \(q = 7\),

Fingerprint Table :

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Σ</td>
<td>11</td>
<td>22</td>
<td>45</td>
<td>26</td>
<td>53</td>
<td>42</td>
<td>20</td>
<td>41</td>
<td>18</td>
<td>37</td>
<td>10</td>
<td>20</td>
<td>41</td>
<td>19</td>
</tr>
<tr>
<td>h(i)</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

- \(h(p) = 6 \rightarrow \text{only need to check strings with fingerprint value = 6}\)

- We can compute the table in \(O(n)\) using the following formula

\[ h(i+1) = ( t[i+m]+2 \times (h(i) \times 2^{m-1} \times t[i]) ) \mod q \]

Example : \(h(2) = 3\);

\[
\begin{align*}
h(3) &= 0 + 2 \times (3 - 2^5 \times 1) \mod 7 = -58 \mod 7 = 5 \\
h(4) &= 1 + 2 \times (5 - 2^5 \times 0) \mod 7 = (1+10) \mod 7 = 4
\end{align*}
\]

Note : For a negative number \(-x \rightarrow -x \mod q = y - x\)

Where \(y\) is the smallest integer with \(y > x > 0\) and \(y\) is multiple of \(q\)