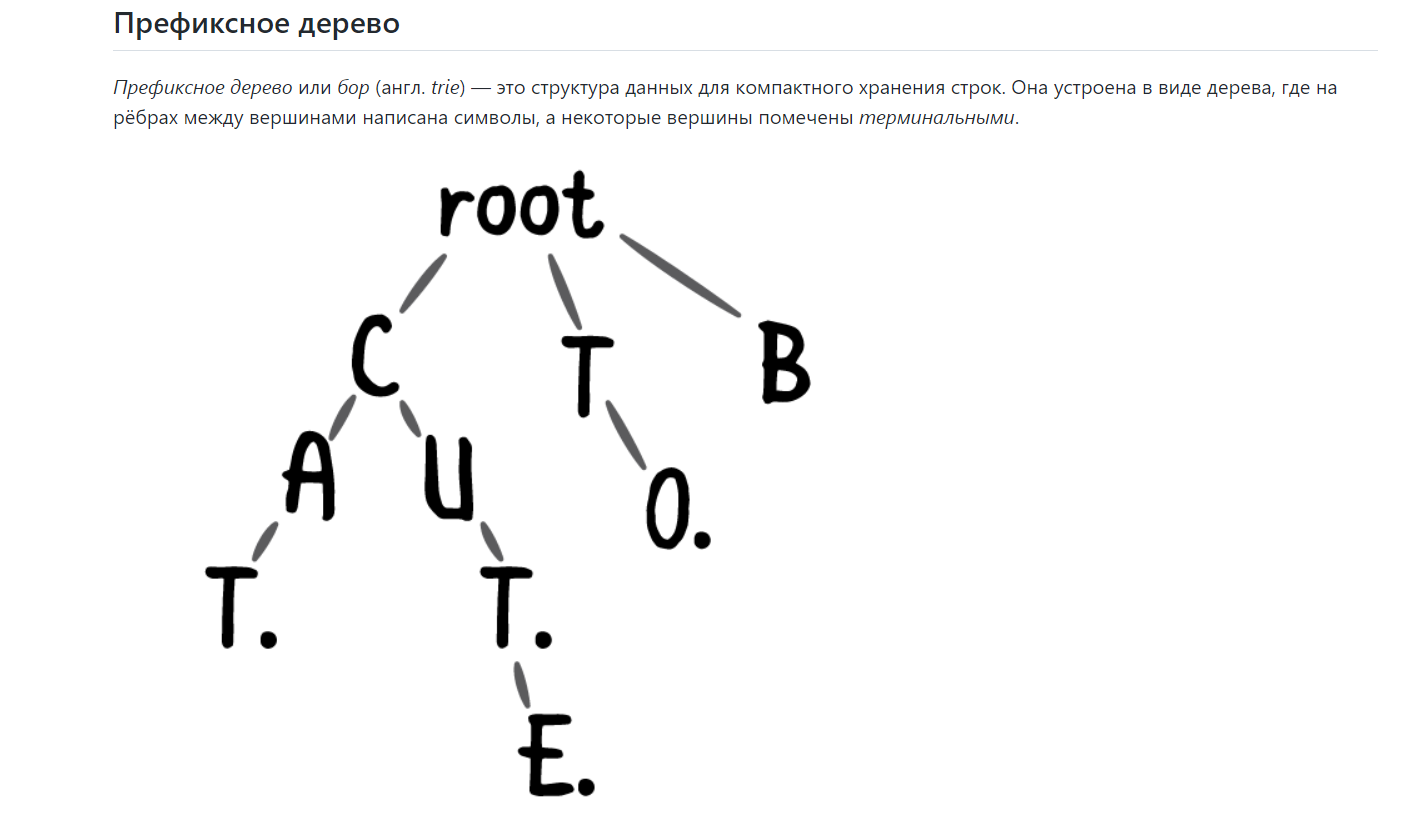
Алгоритм Ахо-Корасик

1. **Историческая справка**

Алгоритм Ахо — Корасик — алгоритм [поиска подстроки](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B8%D1%81%D0%BA_%D0%BF%D0%BE%D0%B4%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B8), разработанный [Альфредом Ахо](https://ru.wikipedia.org/wiki/%D0%90%D1%85%D0%BE,_%D0%90%D0%BB%D1%8C%D1%84%D1%80%D0%B5%D0%B4) и [Маргарет Корасик](https://ru.wikipedia.org/w/index.php?title=%D0%9A%D0%BE%D1%80%D0%B0%D1%81%D0%B8%D0%BA,_%D0%9C%D0%B0%D1%80%D0%B3%D0%B0%D1%80%D0%B5%D1%82&action=edit&redlink=1) в 1975 году[[1]](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%90%D1%85%D0%BE_%E2%80%94_%D0%9A%D0%BE%D1%80%D0%B0%D1%81%D0%B8%D0%BA#cite_note-1), реализует поиск множества подстрок из [словаря](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D1%81%D0%BE%D1%86%D0%B8%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2) в данной [строке](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D0%BE%D0%BA%D0%B0_(%D1%82%D0%B8%D0%BF_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)).

1. **Принцип устройства и Особенности.**

Пусть дан набор строк в алфавите размера k суммарной длины n. Алгоритм Ахо-Корасик строит префиксное дерево для этого набора строк, а затем по этому дереву строит автомат, который может использоваться в различных строковых задачах — например, для нахождения всех вхождений каждой строки из данного набора в произвольный текст за линейное время.  
  


Бор сам по себе можно использовать для разных задач:

* Хранение строк — бор может занимать гораздо меньше места, чем массив или set строк.
* Сортировка строк — можно вывести все строки в лексикографическом порядке.
* Просто множество строк — как мы увидим, в него легко добавлять и удалять слова, а также проверять, есть ли слово в боре.

Бор состоит из ссылающихся друг на друга вершин. В вершине обычно хранится следующая информация:

* терминальная ли вершина (конечная)
* ссылки на детей
* возможно, какая-нибудь дополнительная зависящая от задачи информация о слове. Например, количество таких слов, заканчивающихся в вершине — так можно реализовать мультисет.

Пример алгоритма

Изображение выглядит как текст, часы

Автоматически созданное описание белые вершины – терминальные

Фактически, вершины бора можно понимать как состояния **конечного детерминированного автомата**. Находясь в каком-либо состоянии, мы под воздействием какой-то входной буквы переходим в другое состояние — т.е. в другую позицию в наборе строк. Например, мы находимся на вершине “ca” и под воздействием буквы “a” мы перейдём в состояние “caa”.

Т.е. мы можем понимать рёбра бора как переходы в автомате по соответствующей букве. Однако одними только рёбрами бора нельзя ограничиваться. Если мы пытаемся выполнить переход по какой-либо букве, а соответствующего ребра в боре нет, то мы тем не менее должны перейти в какое-то состояние, соответствующее наидлиннейшему собственному суффиксу строки  (наидлиннейшему из имеющихся в боре), и попытаться выполнить переход по букве из него.

Например, пусть есть соответствующий бор и мы пытаемся понять, может ли нам встретиться такое сочетание букв, как “abc” и мы под воздействием буквы “c” перешли по суффиксной ссылки в состояние “abc”

**Суффиксная ссылка** для каждой вершины p — это вершина, в которой оканчивается наидлиннейший собственный суффикс строки, соответствующей вершине p. Единственный особый случай — корень бора; для удобства суффиксную ссылку из него проведём в себя же. Теперь мы можем переформулировать утверждение по поводу переходов в автомате так: пока из текущей вершины бора нет перехода по соответствующей букве (или пока мы не придём в корень бора), мы должны переходить по суффиксной ссылке. (имеется рекурсивная зависимость)

**Особенности:**

* функция перехода определена через суффиксную ссылку, а суффиксная ссылка — через функию переходов;
* для построения суффиксных ссылок небходимо знать информацию только выше по бору от текущей вершины до корня.

Это позволяет реализовать функции поиска переходов по символу и суффиксных ссылок ленивым образом при помощи взаимной рекурсии.

1. **Оценка временной сложности с обоснованием.**

**Алгоритм работает за линейное время от суммарной длины искомых строк.**

Доказательство. Для каждой из искомых строк x = a1 . . . an ∈ K, рассматривается вычисление значения π для всех префиксов этой строки, ui = a1 . . . ai .

Алгоритм 5 Алгоритм Ахо–Корасик: построение функции π

Дано: множество искомых строк K = {x1, . . . , xk}.

1: построить префиксное дерево для K   
2: for all a ∈ Σ do   
3: if в дереве есть вершина a then   
4: добавить её в очередь  
 5: определить π(a) = ε   
6: else   
7: добавить петлю по a в корне   
8: while в очереди есть вершины do   
9: извлечь вершину u из очереди   
10: for all a ∈ Σ do   
11: if в дереве есть вершина ua then   
12: пусть v = π(u)   
13: while в дереве нет вершины va do  
 14: v = π(v)   
15: определить π(ua) = va  
 16: пометить, что все строки x ∈ K, найденные в va, найдены и в ua   
17: добавить вершину ua в конец очереди

Утверждается, что суммарное число выполнений строчки 14, для всех i, ограничено сверху числом n. Доказательство по сути повторяет доказательство аналогичного свойства алгоритма Кнута–Морриса– Пратта: рассматривается последовательность |π(u1)|, |π(u2)|, . . . , |π(un)|, и утверждается, что значение |π(ui)| увеличивается не более n раз. Общее же число шагов вычисления ограничено количеством уменьшений этого значения. Пусть vi = π(ui) — вычисленное значение для каждого префикса. Вычисляя очередной префикс vi+1, алгоритм проходит последовательность строк π(ui), π 2 (ui), . . . , π `i (ui); каждое применение функции π уменьшает длину строки не менее чем на единицу, а продление π `i (ui) на один символ может увеличить длину на единицу. Поэтому число выполнений строчки 14 на этом шаге не превосходит |vi+1| − |vi | + 1. Сумма по всем i даёт верхнюю оценку n.

ОДНАКО!!!

Если таблицу переходов автомата хранить как красно-черное дерево – расход памяти снижается до О(n), однако вычислительная сложность поднимается до O((H +n)log l + k),

Где n = общая длина всех слов в словаре, l = размер алфавита, k = общая длин всех совпадений   
H = длина текста, в котором производится поиск

1. ***выводы. Плюсы и минусы алгоритма, его применимость***

### Алгоритм Ахо-Корасик:

### долго инициализируется

### потребляет большое количество оперативной памяти

### имеет большую скорость работы.  Широко применяется в системном программном обеспечении, например, используется в утилите поиска grep

### (grep — утилита командной строки, которая находит на вводе строки, отвечающие заданному регулярному выражению, и выводит их, если вывод не отменён специальным ключом) ПРИМЕНИМОСТЬ: Поиск всех строк из заданного набора в тексте

### Нахождение лексикографически наименьшей строки данной длины, не содержащей ни один из данных образцов

### Нахождение кратчайшей строки, содержащей вхождения одновременно всех образцов

### Нахождение лексикографически наименьшей строки длины L, содержащей данные образцы в сумме k раз

### 5. Список литературы :

### Математические основы алгоритмов (Александр Охотин, июнь 2021)

### Алгоритм Ахо-Корасик (статья Хабр)

### Алгоритм Ахо-Корасик (статья Итмо + видео на YouTube)

### Кормен. Алгоритмы - построение и анализ.

### 6. Код программы:

import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
  
*/\*\*  
 \* Реализация алгоритма Ахо-Корасика  
 \*/*public class AhoCorasick {  
  
 private List<Entry> entryList;  
 private int alphabetSize;  
 private List<Vertex> vertexList;  
 private int vertexCount;  
  
 public AhoCorasick(List<Entry> entryList, int alphabetSize) {  
 this.entryList = entryList;  
 this.alphabetSize = alphabetSize;  
 }  
  
 // Строка набора  
 public static class Entry {  
 private String value;  
  
 public String getValue() {  
 return value;  
 }  
  
 public void setValue(String value) {  
 this.value = value;  
 }  
  
 @Override  
 public String toString() {  
 return "Entry{" +  
 "value='" + value + '\'' +  
 '}';  
 }  
 }  
  
 // Вершина бора (состояние конечного детерминированного автомата)  
 public static class Vertex {  
 private int next[]; // массив сыновей  
 private boolean isLeaf; // терминальная вершина  
 private int parent; // родительская вершина  
 private int charIndexFromParent;// символ перехода от родителя  
 private int suffixLink; // суффиксная ссылка  
 private int go[]; // массив переходов, используемый для вычисления суффиксных ссылок  
  
 public Vertex(int alphabetSize) {  
 this.next = newIntArray(alphabetSize);  
 this.go = newIntArray(alphabetSize);  
 this.parent = -1;  
 this.suffixLink = -1;  
 }  
  
 private int[] newIntArray(int size) {  
 int[] ints = new int[size];  
 Arrays.*fill*(ints, -1);  
 return ints;  
 }  
  
 @Override  
 public String toString() {  
 return "Vertex{" +  
 "next=" + Arrays.*toString*(next) +  
 ", go=" + Arrays.*toString*(go) +  
 ", isLeaf=" + isLeaf +  
 '}';  
 }  
 }  
  
 public List<Entry> getEntryList() {  
 return entryList;  
 }  
  
 // добавить строку в набор  
 public boolean addEntry(Entry entry) {  
 return entryList.add(entry);  
 }  
  
 // поиск всех строк из заданного набора в тексте  
 public void solve(String text) {  
 vertexList = new ArrayList<>();  
 vertexList.add(new Vertex(alphabetSize));  
 vertexList.add(new Vertex(alphabetSize));  
 vertexCount = 1;  
  
 for (Entry entry : entryList) {  
 addToTrie(entry);  
 }  
  
 int v = 0;  
 for (char ch : text.toCharArray()) {  
 v = go(v, charToIndex(ch));  
 Vertex vertex = vertexList.get(v);  
 if (vertex.isLeaf) { // посещение терминальной вершине - это нахождение строки из заданного набора  
 break;  
 }  
 }  
  
 }  
  
 // добавить в бор  
 private void addToTrie(Entry entry) {  
 int v = 0;  
 for (char ch : entry.value.toCharArray()) {  
 int sym = charToIndex(ch);  
 if (vertexList.get(v).next[sym] == -1) {  
 Vertex vertex = vertexList.get(vertexCount);  
 vertex.suffixLink = -1;  
 vertex.parent = v;  
 vertex.charIndexFromParent = sym;  
  
 vertexList.add(new Vertex(alphabetSize));  
 vertexList.get(v).next[sym] = vertexCount++;  
 }  
 v = vertexList.get(v).next[sym];  
 }  
 vertexList.get(v).isLeaf = true;  
 }  
  
 private int charToIndex(char ch) {  
 return ch - 'a';  
 }  
  
 // обычный переход  
 private int go(int v, int sym) {  
 Vertex vertex = vertexList.get(v);  
 if (vertex.go[sym] == -1) {  
 if (vertex.next[sym] != -1) {  
 vertex.go[sym] = vertex.next[sym];  
 } else {  
 vertex.go[sym] = v == 0 ? 0 : go(goBySuffixLink(v), sym);  
 }  
 }  
 return vertex.go[sym];  
 }  
  
 // переход по суффиксной ссылке (к вершине, в которой оканчивается наидлиннейший собственный суффикс строки)  
 private int goBySuffixLink(int v) {  
 Vertex vertex = vertexList.get(v);  
 if (vertex.suffixLink == -1) {  
 if (v == 0 || vertex.parent == 0) {  
 vertex.suffixLink = 0;  
 } else {  
 vertex.suffixLink = go(goBySuffixLink(vertex.parent), vertex.charIndexFromParent);  
 }  
 }  
 return vertex.suffixLink;  
 }  
}

Пример задачи, использующая алгоритм Ахо-Корасика (в гитхабе условия и входные)  
  
import java.io.BufferedReader;  
import java.io.File;  
import java.io.InputStreamReader;  
import java.io.PrintStream;  
import java.util.Arrays;  
  
public class Solution {  
  
 private static String[] *genes*;  
 private static int[] *health*;  
  
 public static void main(String[] args) throws Exception {  
 PrintStream out = new PrintStream(System.*out*);  
  
 long startMillis = System.*currentTimeMillis*();  
  
 BufferedReader in = new BufferedReader(new InputStreamReader(System.*in*));  
 int n = Integer.*parseInt*(in.readLine());  
  
 *genes* = new String[n];  
 int longestKey = 0;  
  
 String[] genesItems = in.readLine().split(" ");  
  
 Trie trie = new Solution().new Trie();  
 for (int i = 0; i < n; i++) {  
 String genesItem = genesItems[i];  
 *genes*[i] = genesItem;  
  
 trie.insert(*genes*[i], i);  
  
 if (*genes*[i].length() > longestKey)  
 longestKey = *genes*[i].length();  
 }  
  
 *health* = new int[n];  
  
 String[] healthItems = in.readLine().split(" ");  
  
 for (int i = 0; i < n; i++) {  
 int healthItem = Integer.*parseInt*(healthItems[i]);  
 *health*[i] = healthItem;  
 }  
  
 startMillis = System.*currentTimeMillis*();  
  
 int s = Integer.*parseInt*(in.readLine());  
 long maxH = 0, minH = Long.*MAX\_VALUE*;  
  
 for (int sItr = 0; sItr < s; sItr++) {  
 String[] firstLastd = in.readLine().split(" ");  
  
 int first = Integer.*parseInt*(firstLastd[0]);  
 int last = Integer.*parseInt*(firstLastd[1]);  
  
 String d = firstLastd[2];  
 long total = 0;  
 int keyLen = d.length();  
  
 for (int i = 0; i < keyLen; i++) {  
 total += trie.find(d.substring(i, keyLen > longestKey && i + longestKey < keyLen? i + longestKey : keyLen), first, last);  
 }  
  
 if (total > maxH)  
 maxH = total;  
  
 if (total < minH)  
 minH = total;  
 }  
  
 in.close();  
 out.println(minH + " " + maxH);  
 out.close();  
 }  
  
 class Trie {  
 private final short SIZE = 26;  
 private TrieNode root;  
  
 public Trie () {  
 this.root = new TrieNode();  
 }  
  
 class TrieNode {  
 private TrieNode[] children = new TrieNode[SIZE];  
 private int index[];  
 private boolean isEnd;  
  
 public TrieNode(){  
 isEnd = false;  
 }  
  
 @Override  
 public String toString(){  
 return (index != null? Arrays.*toString*(index) + " " : "")  
 + Arrays.*toString*(Arrays.*stream*(children).filter(p -> p != null).toArray());  
 }  
 }  
  
 public void insert(String key, int index) {  
 int arrayPos;  
  
 TrieNode node = root;  
  
 for (int i = 0; i < key.length(); i++) {  
 arrayPos = key.charAt(i) - 'a';  
  
 if (node.children[arrayPos] == null) {  
 node.children[arrayPos] = new TrieNode();  
 }  
  
 node = node.children[arrayPos];  
 }  
  
 if (node.index == null) {  
 node.index = new int[1];  
 } else {  
 node.index = Arrays.*copyOf*(node.index, node.index.length + 1);  
 }  
  
 node.index[node.index.length - 1] = index;  
 node.isEnd = true;  
 }  
  
 public long find(String key, int start, int end) {  
 int arrayPos;  
  
 TrieNode node = root;  
 long result = 0;  
 int keyLen = key.length();  
  
 for (int i = 0; i < keyLen; i++) {  
 arrayPos = key.charAt(i) - 'a';  
  
 if (node.children[arrayPos] != null) {  
 node = node.children[arrayPos];  
  
 if (node.isEnd  
 && ( start <= node.index[node.index.length - 1]  
 || end >= node.index[0])) {  
  
 int sI = -1, eI = -1;  
 for (int j = 0, k = node.index.length - 1; j < node.index.length && k >= 0; j++, k--) {  
 if (sI == -1) {  
 if (node.index[j] >= start)  
 sI = j;  
 else if (node.index[k] < start && k + 1 < node.index.length && node.index[k + 1] >= start)  
 sI = k + 1;  
 }  
  
 if (eI == -1) {  
 if (node.index[k] <= end)  
 eI = k;  
 else if (node.index[j] > end && j - 1 >= 0 && node.index[j - 1] <= end)  
 eI = j - 1;  
 }  
  
 if (j == k || (sI != -1 && eI != -1))  
 break;  
 }  
  
 if (sI != -1 && eI != -1) {  
 for (int j = sI; j <= eI; j++) {  
 result += *health*[node.index[j]];  
 }  
 }  
 }  
 } else  
 break;  
 }  
  
 return result;  
 }  
  
 @Override  
 public String toString() {  
 return root.toString();  
 }  
 }  
}

import java.io.\*;  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
import java.util.Scanner;  
  
public class TimeTest {  
 public static void main(String[] args) throws IOException {  
  
 File file = new File("src/TimeAhoCorasick.txt");  
 Writer writer = new FileWriter("src/TimeAhoCorasick.txt");  
   
  
 int counter = 0;  
 double start = System.*nanoTime*();  
 for (int j = 1; j < 1000 + counter; j++) {  
 List<AhoCorasick.Entry> entryList = new ArrayList<>();  
 List<String> list = Arrays.*asList*("he", "she", "hers", "his");  
  
 for (String s : list) {  
 AhoCorasick.Entry entry = new AhoCorasick.Entry();  
 entry.setValue(s);  
 entryList.add(entry);  
 }  
  
 AhoCorasick ahoCorasick = new AhoCorasick(entryList, 26);  
 ahoCorasick.solve("ahishers");  
 double end = System.*nanoTime*();  
 double finish = end - start;  
 writer.write(String.*format*("%.2f", finish) + "\n");  
 counter += 1000;  
 }  
 writer.close();  
 }  
}