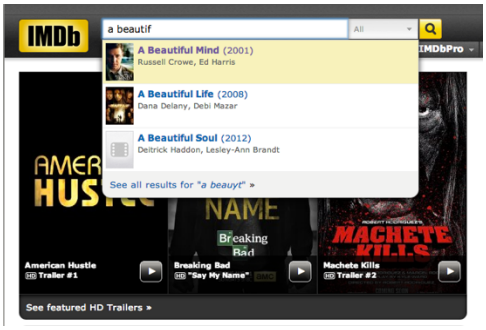
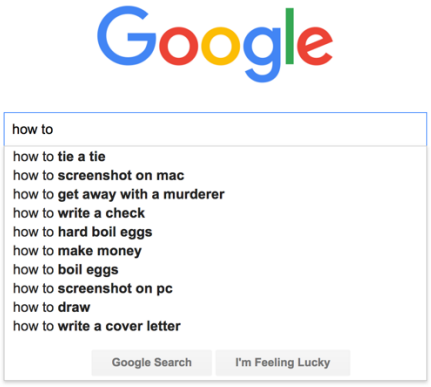
Algoritmos e Estruturas de Dados - ISCTE, 2021/2002 – Projeto A

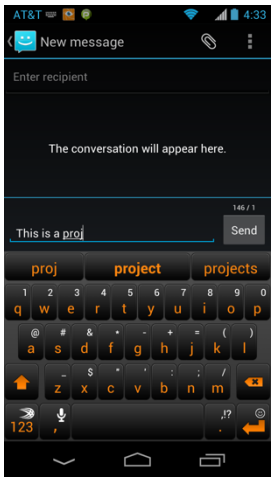
Autocomplete[1]

Escreva um programa que implementa preenchimento automático para um determinado conjunto de n termos, onde um termo é uma cadeia de carateres (*string*) de consulta (*query*) e um peso não negativo associado. Ou seja, dado um prefixo, encontre todas as consultas que começam com o prefixo fornecido, em ordem decrescente de peso.

O preenchimento automático é comum em aplicações atuais. À medida que o utilizador digita, o programa prevê a *query* completa (normalmente palavra ou frase) que o utilizador pretende digitar. O preenchimento automático é mais eficaz quando há um número limitado de *queries* prováveis. Por exemplo, o IMDb usa este mecanismo para exibir os nomes dos filmes conforme o utilizador os digita; os motores de pesquisa usam-no para apresentar sugestões à medida que o utilizador insere termos de pesquisa na web; os telemóveis utilizam-no para acelerar a entrada de texto.







Nos exemplos, a aplicação prevê a probabilidade de o utilizador estar a escrever uma dada *query* e apresenta a lista das principais *queries* correspondentes, por ordem decrescente de peso. Os pesos são determinados por dados históricos, como receita de bilheteira de filmes, frequências de *queries* de outros utilizadores do Google ou o histórico do utilizador do telemóvel. Para este trabalho, estarão definidos o conjunto de todas as consultas possíveis e pesos associados (e essas consultas e pesos não serão alterados).

O desempenho da funcionalidade do preenchimento automático é fundamental em muitos sistemas. Se considerar, por exemplo, um mecanismo de pesquisa que executa preenchimento automático numa *farm* de servidores e de acordo com estudos realizados, a aplicação tem apenas cerca de 50ms para produzir uma lista de sugestões para ser útil ao utilizador. Além disso, esse cálculo tem de ser efetuado a cada entrada de uma tecla na barra de pesquisa.

[1] https://www.cs.princeton.edu/courses/archive/fall20/cos226/assignments/autocomplete/specification.php

Neste trabalho, pretende-se que implemente o preenchimento automático através (1) da ordenação dos termos por *query*; (2) da procura binária para encontrar todas as *queries* que começam com um determinado prefixo; e (3) ordenação dos termos correspondentes por peso.

Parte 1: termo de preenchimento automático. Escreva um tipo de dados imutável `Term.java` que represente um termo de preenchimento automático — uma *string* de consulta e um peso inteiro associado.

Deve implementar a seguinte API, que suporta a comparação de termos por três ordens diferentes: (a) ordem **lexicográfica** por string de consulta (a ordem natural); (b) em ordem **decrescente** por peso (uma ordem alternativa); e (c) ordem **lexicográfica parcial,** por *string* de consulta, mas usando apenas os primeiros r caracteres (uma família de ordenações alternativas). A última ordem pode parecer estranha, mas será usada na Parte 3 para encontrar as *strings* de consulta que começam por um determinado prefixo (de comprimento r).

```
public class Term implements Comparable<Term> {

    // Initializes a term with the given query string and weight.
    public Term(String query, long weight)

    // Compares the two terms in descending order by weight.
    public static Comparator<Term> byReverseWeightOrder()

    // Compares the two terms in lexicographic order,
    // but using only the first r characters of each query.
    public static Comparator<Term> byPrefixOrder(int r)

    // Compares the two terms in lexicographic order by query.
    public int compareTo(Term that)

    // Returns a string representation of this term in the following format:
    // the weight, followed by a tab, followed by the query.
    public String toString()

    // unit testing (required)
    public static void main(String[] args)

}
```

Condições fronteira. Lance um `IllegalArgumentException` no construtor se a consulta for nula ou o peso for negativo. Lance um `IllegalArgumentException` em byPrefixOrder() se r for negativo.

Testes unitários. O método `main()` deve chamar cada construtor e método público diretamente e verificar se eles funcionam conforme prescrito (p.e. imprimindo resultados em *stdout*).

Requisitos de desempenho. As funções de comparação de *strings* devem levar um tempo proporcional ao número de caracteres necessários para resolver a comparação

Parte 2: pesquisa binária. Ao realizar a pesquisa binária num vetor ordenado que contém mais de uma chave igual à chave de pesquisa, o cliente pode querer saber o índice da *primeira* ou da *última* chave. Assim, implemente a seguinte API:

```
public class BinarySearchDeluxe {

    // Returns the index of the first key in the sorted array a[]
    // that is equal to the search key, or -1 if no such key.
    public static <Key> int firstIndexOf(Key[] a, Key key, Comparator<Key>
comparator)

    // Returns the index of the last key in the sorted array a[]
    // that is equal to the search key, or -1 if no such key.
    public static <Key> int lastIndexOf(Key[] a, Key key, Comparator<Key>
comparator)

    // unit testing (required)
    public static void main(String[] args)
}
```

Condições fronteira. Lance um `IllegalArgumentException` se qualquer argumento para `firstIndexOf()` ou `lastIndexOf()` for nulo.

Pré-condições. Pressuponha que o vetor argumento está ordenado (em relação ao comparador fornecido).

Testes unitários. O método main() deve chamar cada método público diretamente e verificar se funcionam como prescrito (por exemplo, imprimindo os resultados em *stdout*).

Requisitos de desempenho. Os métodos `firstIndexOf()` e `lastIndexOf()` devem executar no máximo em 1+⌈log2n⌉ comparações no pior caso (n é o comprimento do vetor). Uma comparação é uma chamada para comparator.compare().

Parte 3: preenchimento automático. Nesta parte, deve ser implementado um tipo de dados que fornece funcionalidade de preenchimento automático para um determinado conjunto de strings e pesos, usando `Term` e `BinarySearchDeluxe`. Para isso, ordene os *termos* por ordem lexicográfica; use a pesquisa binária para encontrar todas as strings de consulta que começam com um determinado prefixo; e ordene os termos correspondentes em ordem decrescente por peso. Construa o programa com um tipo de dados imutável Autocomplete com a seguinte API:

```
public class Autocomplete {
    // Initializes the data structure from the given array of terms.
    public Autocomplete(Term[] terms)

    // Returns all terms that start with the given prefix, in descending
order of weight.
    public Term[] allMatches(String prefix)

    // Returns the number of terms that start with the given prefix.
    public int numberOfMatches(String prefix)

    // unit testing (required)
    public static void main(String[] args)
}
```

Condições fronteira. Lance `IllegalArgumentException` no construtor se o vetor argumento for `null` ou qualquer entrada for `null`. Lance `IllegalArgumentException` em `allMatches()` e `numberOfMatches()` se seu argumento for nulo.

Testes unitários. O método `main()` chama cada construtor e método público diretamente e verifica se funcionam conforme prescrito (por exemplo, imprimindo resultados em *stdout*).

Requisitos de desempenho. A implementação deve satisfazer cada um dos seguintes requisitos de desempenho de pior caso: (i) o construtor deve fazer O (nlogn) comparações onde n é o número de termos; (ii) o método `allMatches()` deve fazer O (mlogm+logn) comparações, onde m é o número de termos correspondentes; (iii) O método `numberOfMatches()` deve fazer O(logn) comparações. Uma comparação é uma chamada para qualquer um dos métodos `compare()` ou `compareTo()` definidos em `Term`.

Formato de entrada. Existem vários ficheiros de entrada de amostra para teste. Cada ficheiro contém um inteiro **n** seguido por **n** pares de strings de consulta e pesos não negativos. Há um par por linha, com o peso e a corda separados por uma tabulação. Um peso pode ser qualquer número inteiro entre 0 e 263 − 1. Uma string de consulta pode ser qualquer sequência de caracteres Unicode, incluindo espaços (mas não mudanças de linha - *newline*). O ficheiro wiktionary.txt contém as 10.000 palavras mais comuns no Projeto Gutenberg, com pesos proporcionais às suas frequências. O ficheiro cities.txt contém 93.827 cidades, com pesos iguais às suas populações.

In this context, a *compare* is one call to any of the `compare()` or `compareTo()` methods defined in `Term`.

```
~/Desktop/autocomplete> more
wiktionary.txt
10000
   5627187200  the
   3395006400  of
   2994418400  and
   2595609600  to
   1742063600  in
   1176479700  i
   1107331800  that
   1007824500  was
    879975500  his
                ...
       392323  calves
```

```
~/Desktop/autocomplete> more cities.txt
93827
      14608512 Shanghai, China
      13076300 Buenos Aires, Argentina
      12691836 Mumbai, India
      12294193 Mexico City, Distrito
Federal, Mexico
      11624219 Karachi, Pakistan
      11174257 İstanbul, Turkey
      10927986 Delhi, India
      10444527 Manila, Philippines
      10381222 Moscow, Russia
                ...
             2 Al Khāniq, Yemen
```

Exemplo Em baixo, um cliente exemplo que recebe o nome de um ficheiro de entrada e um inteiro k como argumentos de linha de comando, lê os dados do ficheiro e em seguida lê repetidamente as *queries* de preenchimento automático do *stdin* e imprime os k principais termos correspondentes em ordem decrescente de peso.

```
public static void main(String[] args) {

    // read in the terms from a file
    String filename = args[0];
    In in = new In(filename);
    int n = in.readInt();
    Term[] terms = new Term[n];
    for (int i = 0; i < n; i++) {
        long weight = in.readLong();           // read the next weight
        in.readChar();                         // scan past the tab
        String query = in.readLine();          // read the next query
        terms[i] = new Term(query, weight);    // construct the term
    }

    // read in queries from standard input and print the top k matching terms
    int k = Integer.parseInt(args[1]);
    Autocomplete autocomplete = new Autocomplete(terms);
    while (StdIn.hasNextLine()) {
        String prefix = StdIn.readLine();
        Term[] results = autocomplete.allMatches(prefix);
        StdOut.printf("%d matches\n", autocomplete.numberOfMatches(prefix));
        for (int i = 0; i < Math.min(k, results.length); i++)
            StdOut.println(results[i]);
    }
}
```

Execuções ilustrativas

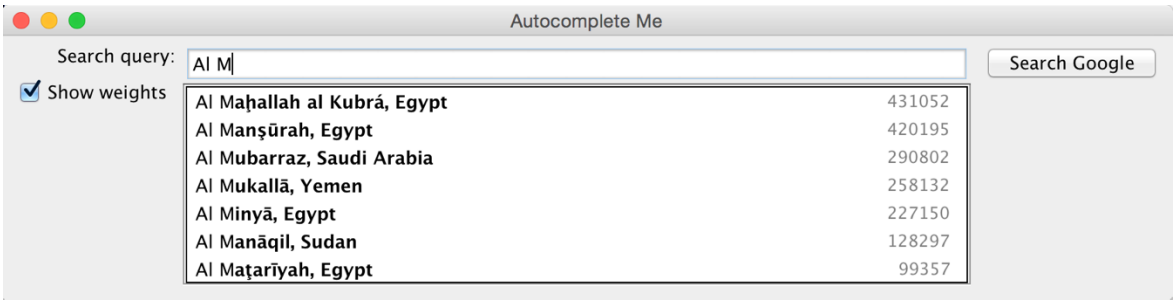
```
~/Desktop/autocomplete> java-algs4 Autocomplete wiktionary.txt 5

auto
2 matches
        619695 automobile
        424997 automatic
comp
52 matches
      13315900 company
       7803980 complete
       6038490 companion
       5205030 completely
       4481770 comply
the
38 matches
    5627187200 the
     334039800 they
     282026500 their
     250991700 them
     196120000 there
```

```
~/Desktop/autocomplete> java-algs4 Autocomplete cities.txt 7
M
7211 matches
    12691836  Mumbai, India
    12294193  Mexico City, Distrito Federal, Mexico
    10444527  Manila, Philippines
    10381222  Moscow, Russia
     3730206  Melbourne, Victoria, Australia
     3268513  Montréal, Quebec, Canada
     3255944  Madrid, Spain
Al M
39 matches
      431052  Al Maḩallah al Kubrá, Egypt
      420195  Al Manşūrah, Egypt
      290802  Al Mubarraz, Saudi Arabia
      258132  Al Mukallā, Yemen
      227150  Al Minyā, Egypt
      128297  Al Manāqil, Sudan
       99357  Al Maţarīyah, Egypt
```

GUI interativa (opcional). Descarregue e compile AutocompleteGUI.java.. O programa recebe o nome de um arquivo e um inteiro k como argumentos de linha de comando e fornece uma GUI para o utilizador inserir consultas. Apresenta os k principais termos correspondentes em tempo real. Quando o utilizador seleciona um termo, a GUI abre os resultados de uma pesquisa do Google com esse termo.

```
~/Desktop/autocomplete> java-algs4 AutocompleteGUI cities.txt 7
```



Entrega. Envie `Autocomplete.java`, `BinarySearchDeluxe.java` e `Term.java`. Não chame funções de biblioteca além daquelas em `java.lang`, `java.util` e `algs4.jar`. Por fim, envie o ficheiro readme.txt com as questões respondidas.

This assignment was developed by Matthew Drabick and Kevin Wayne. Versão original
https://www.cs.princeton.edu/courses/archive/fall20/cos226/assignments/autocomplete/specification.php