Tópicos Avançados em Algoritmos

Hamilton José Brumatto

Bacharelado em Ciências da Computação - UESC

11 de fevereiro de 2019

Iterators e Algoritmos

Iterators

- É o ponto principal de importância no desenho do STL:
  - Permite que os algoritmos sejam construídos sem que exista uma preocupação sobre como os dados são armazenados nos containers.
  - Permite construir containers sem se preocupar em todo um conjunto de algoritmos para manipular os dados.
- Iterators são objetos estilo ponteiros e podem ser categorizados como:
  - *input*, *output*, *forward*, *bidirectional* e *random access*.
- Existe o conceito de intervalo para iterators: `[first,last)`, ou seja, `first` aponta para o primeiro elemento e `last` aponta para a primeira posição após o último elemento.

Operações básicas sobre Iterators

- Iterators são ditos válidos se a partir do `first`, através de `operator++` atinge-se todos os endereços da sequência, até que `last`, que não representa um endereço da sequência, apenas o fim do intervalo.
- Um caso importante é o intervalo vazio: `first == last`. Um intervalo vazio é válido, mas não inclui nenhum iterator.
- Todos algoritmos STL consideram que todos iterators estão em intervalos válidos.

Input Iterators

- São iterators que possuem as seguintes características:
  - Implementa `operator==` e `operator!=`, ou seja, permite a comparação "`first != last`"
  - Implementa `operator++` (pré-fixo e pós-fixo): permite a operação "`++first`", onde incrementa o iterator e retorna o novo.
  - Implementa o operador de indireção: "`*first`" retornando o valor para o qual o iterator aponta.
  - Oferece as operações acima em tempo constante.
- Todas estas características são atendidas pelos ponteiros tradicionais. Com a observação que o input iterator não precisa ter a capacidade de alterar o conteúdo, só ler.
- Quase todos os containers que utilizamos oferece este tipo de iterator.
- Um tipo especial de iterator `istream` também oferece estas características em stream de entrada de dados.

Input Iterators: exemplo de uso - find

```
template <class InputIterator, class T>
  InputIterator find (InputIterator first, InputIterator last, const T& val);
```

- A implementação:

Algoritmo genérico find

```
template <class InputIter, class T>
InputIter find (InputIter first, InputIter last, const T& val) {
  while (first!=last) {
    if (*first==val) return first;
      ++first;
  }
  return last;
}
```

Input Iterators: exemplo de uso - find

```
#include <iostream>
#include <algorithm>
#include <list>

using namespace std;
int main() {
 int a[10] = {12, 3, 25, 7, 11, 213, 7, 123, 20, -31};
 // Encontra a primeira ocorrência do elemento 7:
 int *ptr = find(a, a+10, 7);
 cout << ''Achou:  ''  << *ptr;
 cout << '' o próximo é:''  << *(++ptr) << endl;
 // Cria uma lista com os inteiros:
 list<int> lst(a,a+10);
 list<int>::iterator i = find(lst.begin(), lst.end(), 7);
 cout << ''Achou:  ''  << *i;
 cout << '' o próximo é:''  << *(++i) << endl;
 return 0;
}
```

Output Iterators

- Possui quase todas as qualidades do Input Iterator.
- A grande diferença é que os Input Iterators oferece a capacidade de ler os conteúdos, e não de escrever.
- Nos Output Iterators é necessário a capacidade de escrever conteúdos e não de ler.
- Estas características são importantes quando associamos Input Iterators com Input Stream (Stream de leitura de um console, por exemplo), e Output Iterators com Output Stream (Stream de escrita em um console).

Output Iterators: exemplode uso - copy

```
template <class InputIterator, class OutputIterator>
 OutputIterator copy (InputIterator first, InputIterator last,
                      OutputIterator result);
```

- A implementação:

Algoritmo genérico copy

```
template <class InputIterator, class OutputIterator>
OutputIterator find (InputIterator first, InputIterator last,
                     OutputIterator result) {
 while (first!=last) {
  *result = *first;
  ++result; ++first;
 }
 return result;
}
```

Output Iterators: exemplo de uso - copy

```
#include <iostream>
#include <algorithm>
#include <list>
using namespace std;
void prt(int i) { cout << i << ' '; }
int main() {
 int a[10] = {12, 3, 25, 7, 11, 213, 7, 123, 20, -31};
 // Copia a em b:
 copy(a, a+10, b);
 for_each(b,b+10,prt)
 cout << endl;
 // Cria uma lista com os inteiros:
 list<int> lst(a,a+10);
 // Copia na ordem inversa:
 copy(lst.rbegin(),lst.rend(),b;
 for_each(b,b+10,prt)
 cout << endl;
 return 0;
}
```

Forward Iterators

- Forward Iterators são iterators que agrega as qualidades de ambos Input e Output Iterators:
  - *leitura* e *escrita* de conteúdo e `operator!=`, `operator==` e `operator++`
  - Os Forward Iterators também conseguem salvar a posição em que estão para continuar, posteriormente a partir da posição salva.
  - Da mesma forma que os anteriores, as operações são em tempo constante.
- Exemplo de uso - `replace`

```
template <class ForwardIterator, class T>
  void replace (ForwardIterator first, ForwardIterator last,
                const T& old_value, const T& new_value);
```

Forward Iterators: Exemplo de uso - replace

- Implementação do algoritmo:

Algoritmo genérico replace

```
template <class ForwardIterator, class T>
void replace (ForwardIterator first, ForwardIterator last,
              const T& old_value, const T& new_value) {
 while ((first!=last)) {
  if (*first==old_value) *first=new_value
  ++first;
 }
}
```

Forward Iterators: exemplo de uso - replace

```
#include <iostream>
#include <algorithm>
using namespace std;

void prt(int i) { cout << i << ' '; }

int main() {
 int a[10] = {12, 3, 25, 7, 11, 213, 7, 123, 20, -31};

 replace(a, a+10, 7, 17);
 for_each(a,a+10,prt)
 cout << endl;
 return 0;
}
```

Bidirectional Iterators

- Todas as operações do `Forward Operator` mais,
- A operação `operator--` que permite reverter a operação na sequência.
- Ambas versões de prefixo e posfixo do operador são necessárias.
- Todas operações devem ser executadas em tempo constante
- Exemplo de uso - `reverse`

```
template <class BidirectionalIterator>
 void reverse (BidirectionalIterator first, BidirectionalIterator last);
```

Bidirectional Iterators: Exemplo de uso - `reverse`

- Implementação do algoritmo:

Algoritmo genérico `reverse`

```
template <class BidirectionalIterator>
void reverse (BidirectionalIterator first,
              BidirectionalIterator last) {
 while ((first!=last)&&(first!=--last)) {
  std::iter_swap (first,last);
  ++first;
 }
}
```

Bidirectional Iterators: exemplo de uso - reverse

```
#include <iostream>
#include <algorithm>
using namespace std;

void prt(int i) { cout << i << ' '; }

int main() {
 int a[10] = {12, 3, 25, 7, 11, 213, 7, 123, 20, -31};

 reverse(a, a+10);
 for_each(a,a+10,prt)
 cout << endl;
 return 0;
}
```

Random Access Iterators

- Reúne todas as especificações de operações em tempo constante:
  - Lê e caminha em uma direção (Input Iterator). Escreve e caminha em uma direção (Output Iterator).
  - Lê e Escreve e caminha em uma direção, com a possibilidade de salvar a posição e continuar a partir de posições salvas (Forward Iterator), ou em ambas direções (Bidirectional Iterator)
- Alguns algoritmos precisam que seja possível o acesso aleatório de uma posição em tempo constante, que é atendido por Random Access Iterators, com as operações em tempo constante:
  - Saltos em intervalos: `operator+`, `operator+=`, `operator-`, `operator-=`
  - Comparações de posições (na ordem da sequência): `operator<`, `operator<=`, `operator>`, `operator>=`

Random Access Iterators: Exemplo de uso - sort

```
padrão (1)    template <class RandomAccessIterator>
                void sort(RandomAccessIterator first, RandomAccessIterator last);
person. (2)   template <class RandomAccessIterator, class Compare>
                void sort(RandomAccessIterator first, RandomAccessIterator last,
                          Compare comp);
```

- `first`, `last`:
  - São `RandomAccess Iterators` para a posição inicial e final da sequência a ser ordenada. O intervalo é definido por `[first,last)`. O tipo que o ponteiro aponta deve ter a operação `swap` bem definida.
- `comp` (função ou objeto função):
  - Uma função binária que aceita dois elementos do intervalo de argumentos e retorna`bool`. O valor retornado indica se o elemento passado na primeira posição é considerado a ir antes do segundo na ordem desejada (estritamente fraca). A função não pode modificar os argumentos.

RandomAccess Iterators: exemplo de uso - sort

```
#include <iostream>
#include <algorithm>
using namespace std;

void prt(int i) { cout << i << ' '; }

int main() {
 int a[10] = {12, 3, 25, 7, 11, 213, 7, 123, 20, -31};

 sort(a, a+10);
 for_each(a,a+10,prt)
 cout << endl;
 return 0;
}
```

Hierarquia

- Entendendo a hierarquia dos Iterators fica simples a manipulação dos elementos da STL.
  - Forward Iterators são também Input e Output Iterators.
  - Bidirectional Iterators são também Forward e consequentemente Input e Output Iterators.
  - Random Access Iterators são também Bidirectional, logo, Forward e Input e Output iterators.
- Isto implica que:
  - Algoritmos que requer somente input ou output iterators pode ser usado com forward, bidirectional e random access iterators.
  - Algoritmos que requer forward iterators pode ser usado com bidirectional e random access iterators.
  - Algoritmos que requer bidirectional iterators pode ser usado com random access iterators

Hierarquia

- As categorias de iterators faz parte da especificação de containers e algoritmos
  - `lists` provê bidirectional iterators, e o algoritmo `find` requer forward iterators, logo podemos usar `find` com `list`.
  - O algoritmo `sort` requer random access iterators, logo não podemos usar `sort` com `list`
  - `deque` provê random access iterator, logo podemos usar `sort` com `deque`.
- A construção dos algoritmos e classes do STL são projetados para encorajar combinações que são eficientes, e desencorajar combinações que não são eficientes.
  - O algoritmo `binary_search` requer forward iterator, mas na descrição há a indicação de que se utilizar random access iterators obterá uma eficiência $O(\log n)$ enquanto que com forward é apenas $O(n)$.

Input e Output - Stream Iterators

- Os stream iterators, input e output são úteis para apontar dados em streams.
- Como exemplo de streams, temos os consoles de entrada e saída
- Estas classes são importantes quando os streams de entrada e saída não são os óbvios, por exemplo, para comunicação em rede.
- Também é possível utilizar vetores (ou strings) como buffer de streams.
- As classes `istream_iterator` e `ostream_iterator` criam objetos iterators, input para a primeira e output para a segunda, a partir de input streams (como o `cin`) e output streams (como o `cout`).

Input Streams

```
template <class T, class charT=char, class traits=char_traits<charT>,
          class Distance=ptrdiff_t>
 class istream_iterator
```

- Os construtores desta classe criam objetos do tipo input iterators
- Ao indicar um objeto do tipo `basic_istream` para o construtor, o iterator irá retirar valores deste objeto.
- Um construtor sem objeto indica um iterator de intervalo vazio.
- Podemos construir como iterator: `istream_iterator<int>(cin)`

Input Streams - Exemplo

- Vamos utilizar dois algoritmos: `copy`, que já apresentamos neste assunto. Copia os valores de um Input Iterator para um Output Iterator
- `back_inserter`

```
template <class Container>
 back_insert_iterator<Container> back_inserter (Container& x)
```

- Sendo que o `back_insert_iterator` é um Output Iterator especial que permite que algoritmos que substituem elementos (como o `copy`) passem a inserir elementos no final do container.
  - Para tanto é necessário que o container tenha especificado o método `push_back`

Input Streams - Exemplo

```
#include <iostream>
#include <iterator>
#include <list>
#include <algorithm>
using namespace std;

void prt(int i)  cout << i << ' ';

int main() {
 list<int> lst;

 copy(istream_iterator<int>(cin), istream_iterator<int>(),
      back_inserter(lst));
 for_each(lst.begin(), lst.end(), prt);
 cout << endl;
 return 0;
}
```

Input Streams - Exemplo (criando objetos)

```
#include <iostream>
#include <iterator>
#include <list>
#include <algorithm>
using namespace std;

void prt(int i)  cout << i << ' ';

int main() {
 list<int> lst;
 istream_iterator<int> in(cin);
 istream_iterator<int> fim;

 copy(in, fim, back_inserter(lst));
 for_each(lst.begin(), lst.end(), prt);
 cout << endl;
 return 0;
}
```

Iterator mutáveis

- Os iterators podem ser mutável ou constante.
- Os iterators mutáveis permitem que valores sejam atribuídos pela indireção apontada por *, os constantes não
- Alguns containers oferecem as duas versões: `vector`

Atributos	Definição
iterator	RandomAccessIterator para `tipo_do_valor`
const_iterator	RandomAccessIterator para `const tipo_do_valor`
reverse_iterator	`reverse_iterator<iterator>`
const_reverse_iterator	`reverse_iterator<const_iterator>`

- Podemos fazer as operações:

```
vector<T> v;
...
vector::iterator b = v.begin();
vector::const_iterator cb = v.cbegin();
```

Iterator mutáveis

- Muitas vezes a conversão de mutável para constante pode ser automática. O inverso não acontece:

```
const vector<int> v(100,0);
...
vector::iterator b = v.begin(); // Linha de erro
```

- A linha acima contém erro, pois como `v` é declarado constante, o iterator gerado por `begin()` será automaticamente convertido para constante, esta conversão é possível.
- Ao assumir o `b` do tipo mutável, a atribuição gera erro (não dá para converter constante para mutável.

Categorias de Iterators

Container	Iterator	Categoria
`T a[n]`	`T*`	random access mutável
`T a[n]`	`const T*`	random access constante
`vector<T>`	`vector<T>:iterator`	random access mutável
`vector<T>`	`vector<T>:const_iterator`	random access constante
`deque<T>`	`deque<T>:iterator`	random access mutável
`deque<T>`	`deque<T>:const_iterator`	random access constante
`list<T>`	`list<T>:iterator`	bidirecional mutável
`list<T>`	`list<T>:const_iterator`	bidirecional constante
`set<T>`	`set<T>:iterator`	bidirecional constante
`set<T>`	`set<T>:const_iterator`	bidirecional constante
`multiset<T>`	`multiset<T>:iterator`	bidirecional constante
`multiset<T>`	`multiset<T>:const_iterator`	bidirecional constante
`map<key,T>`	`map<Key,T>:iterator`	bidirecional mutável
`map<key,T>`	`map<Key,T>:const_iterator`	bidirecional constante
`multimap<Key,T>`	`multimap<Key,T>:iterator`	bidirecional mutável
`multimap<Key,T>`	`multimap<Key,T>:const_iterator`	bidirecional constante

Categorias de Algoritmos

- Como vimos os algoritmos da biblioteca STL são genéricos e se aplicam a um conjunto diverso de containers
- Eles podem ser classificados em 4 grandes categorias:
  - Algoritmos não mutáveis em sequências: opera nos containers sem modificar o seu conteúdo.
  - Algoritmos mutáveis em sequências: Opera nos containers, podendo modificar os seus conteúdos.
  - Algoritmos relacionado a ordenação: Realiza operações de ordenação, ou necessita containers ordenados.
  - Algoritmos numéricos generalizados: Algoritmos para operações numéricas.

Modificar a sequência ou uma cópia

- Muitos algoritmos realizam a ação sobre a própria sequência, como o `replace`
  - `replace` busca um valor na sequência e substitui por outro.
- O algoritmo `copy` gera uma cópia em outra sequência.
- É possível oferecer uma versão *copy* de alguns algoritmos, assim ao invés de modificar a própria sequência, irá gerar uma sequência nova.
- `replace_copy` faz exatamente isto cria nova sequência, substituindo um valor por outro.
- Nem sempre é possível, por exemplo `sort` não tem a versão `sort_copy`, pois o tempo de execução de sort é maior que copy. Pode-se fazer o copy, depois o sort que não perde eficiência.

Algoritmos com parâmetros do tipo `função`

- Alguns algoritmos necessitam uma função como parâmetro. Uma opção eficiente à função é um objeto função.
- Geralmente a função é um *predicado*, uma função que retorna um valor booleano.
- Os predicados geralmente são usados em ordenação para a comparação de valores.
- Duas funções especiais já são criadas: `template <T> less<T>` e `template <T> greater<T>`.
  - Para a primeira a classe T deve definir `operator<` e na segunda `operator>`.
  - Alguns algoritmos oferecem mais de uma versão, já colocando como padrão a função `less<T>`

Buscando um valor

- Busca elementos na sequência, de forma linear.
  - `find`: busca a primeira ocorrência um valor no intervalo.
  - `find_if`: busca um elemento no intervalo cujo valor satisfaz um predicado.
  - `find_if_not`: aplica a condição negativa do predicado no valor.
  - `find_end`: busca a última ocorrência de um valor no intervalo.
  - `find_first_of`: busca a primeira ocorrência de um valor pertencente a outro intervalo no primeiro intervalo.
  - `adjacent_find`: Encontra a primeira ocorrência de 2 elementos de mesmo valor adjacentes.
  - `search`: Busca se uma subsequência faz parte da sequência.
  - `search_n`: Busca se existem `n` elementos em uma sequência que satisfazem um valor ou um predicado.

Count

- Verifica o número de ocorrências na sequência:
  - `count`: Conta quantas vezes um valor aparece em uma sequência.
  - `count_if`: Conta quantos valores na sequência satisfazem um predicado.
  - `all_of`: Verifica se todos os elementos satizfazem uma condição.
  - `any_of`: Verifica se existe algum elemento que satisfaça uma condição.
  - `none_of`: Verifica se nenhum elemento satisfaz uma condição.
  - `for_each`: Aplica uma função a cada um dos elementos da sequência.

Comparações

- Compara as sequências para verifica que contém os mesmos elementos
  - `equal`: Compara duas sequências verificando se todos os elementos correspondem (são iguais ou satisfazem um predicado)
  - `is_permutation`: Verifica se os elementos da sequência 2 estão todos na sequência 1.
  - `mismatch`: Retorna a primeira posição onde duas sequências não se correspondem.

Exemplo: problema 1129 URI - Leitura Ótica

O professor João decidiu aplicar somente provas de múltipla escolha, para facilitar a correção. Em cada prova, cada questão terá cinco alternativas (A, B, C, D e E), e o professor vai distribuir uma folha de resposta para cada aluno. Ao final da prova, as folhas de resposta serão escaneadas e processadas digitalmente para se obter a nota de cada aluno. Inicialmente, ele pediu ajuda a um sobrinho, que sabe programar muito bem, para escrever um programa para extrair as alternativas marcadas pelos alunos nas folhas de resposta. O sobrinho escreveu uma boa parte do software, mas não pode terminá-lo, pois precisava treinar para a Maratona de Programação. Durante o processamento, a prova é escaneada usando tons de cinza entre 0 (preto total) e 255 (branco total).

Exemplo: problema 1129 URI - Leitura Ótica

Após detectar os cinco retângulos correspondentes a cada uma das alternativas, ele calcula a média dos tons de cinza de cada pixel, retornando um valor inteiro correspondente àquela alternativa. Se o quadrado foi preenchido corretamente o valor da média é zero (preto total). Se o quadrado foi deixado em branco o valor da média é 255 (branco total). Assim, idealmente, se os valores de cada quadrado de uma questão são (255, 0, 255, 255, 255), sabemos que o aluno marcou a alternativa B para essa questão. No entanto, como as folhas são processadas individualmente, o valor médio de nível de cinza para o quadrado totalmente preenchido não é necessariamente 0 (pode ser maior); da mesma forma, o valor para o quadrado não preenchido não é necessariamente 255 (pode ser menor).

Exemplo: problema 1129 URI - Leitura Ótica

O prof. João determinou que os quadrados seriam divididos em duas classes: aqueles com média menor ou igual a 127 serão considerados pretos e aqueles com média maior a 127 serão considerados brancos.
Obviamente, nem todas as questões das folhas de resposta são marcadas de maneira correta. Pode acontecer de um aluno se enganar e marcar mais de uma alternativa na mesma questão, ou não marcar nenhuma alternativa. Nesses casos, a resposta deve ser desconsiderada.
O professor João necessita agora de um voluntário para escrever um programa que, dados os valores dos cinco retângulos correspondentes às alternativas de uma questão determine qual a alternativa corretamente marcada, ou se a resposta à questão deve ser desconsiderada.

Entrada:

A entrada contém vários casos de teste. A primeira linha de um caso de teste contém um número inteiro **N** indicando o número de questões da folha de respostas ($1 \leqslant \mathbf{N} \leqslant 255$). Cada uma das **N** linhas seguintes descreve a resposta a uma questão e contém cinco números inteiros **A**, **B**, **C**, **D** e **E**, indicando os valores de nível de cinza médio para cada uma das alternativas da resposta ($0 \leqslant$ **A**, **B**, **C**, **D** e **E** $\leqslant 255$). O ultimo caso de teste é seguido por uma linha que contém apenas um número zero.

Saída:

Para cada caso de teste da entrada seu programa deve imprimir **N** linhas, cada linha correspondendo a uma questão. Se a resposta à questão foi corretamente preenchida na folha de resposta, a linha deve conter a alternativa marcada ('A', 'B', 'C', 'D'ou 'E'). Caso contrário, a linha deve conter o caractere '*'(asterisco).

Problema URI 1129 - Solução

```
#include <iostream> // cin, cout
#include <algorithm>// find_if
using namespace std;
class black_t{public:  bool operator() (int x){return x<=127;}}black;
int main() {
 char resp[5]={'A','B','C','D','E'};
 int n, *b, *f, a[5];
 cin >> n;
 while(n) {
  while(n--) {
   for(int i=0; i<5; i++) cin >> a[i];
   b = find_if(a,a+5,black);
   if(b!=(a+5)) {
    f = find_if(b+1,a+5,black);
    if(f==(a+5)) cout << *(resp+(b-a)) << endl;
    else cout << ''*'' << endl;
   } else cout << ''*'' << endl;
  }
  cin >> n;
 }
 return 0;
}
```

Cópia

- Realiza a cópia de elementos de um arranjo em outro:
  - `copy`: copia o intervalo de elementos.
  - `copy_n`: copia `n` elementos.
  - `copy_if`: copia elementos que satisfaz a condição, ou predicado.
  - `copy_bakward`: copia os elementos na ordem inversa.
  - `move`: copia para o novo arranjo deixando o arranjo anterior em um estado não específico, mas válido.
  - `move_bakward`: move em ordem inversa.
  - `swap`: troca os arranjos (mesmo se forem de tamanhos distintos)
  - `swap_ranges`: troca elementos de intervalos específicos.
  - `transform`: copia para um novo arranjo aplicando um algoritmo a cada item.

Alteração

- Algoritmos que alteram o próprio arranjo. Alguns possuem uma versão cópia que gera um novo arranjo alterado.
  - `replace (replace_copy)`: substitui um valor do arranjo por um novo valor.
  - `replace_if (replace_copy_if`: subistitui valores que satisfazem um predicado.
  - `fill`: preenche o arranjo com um valor.
  - `fill_n`: preenche `n` elementos com um valor.
  - `generate`: preenche o arranjo a partir do resultado de uma função.
  - `generate_n`: preenche `n` elementos do arranjo usando uma função.
  - `remove (remove_copy)`: remove um valor do arranjo.
  - `remove_if (remove_copy_if`: remove valores que satisfazem um predicado.
  - `unique (unique_copy)`: remove valores duplicados.
  - `reverse (reverse_copy)`: substitui o arranjo por sua versão reversa.
  - `rotate (rotate_copy)`: rotaciona os elementos a partir de um novo primeiro elemento.
  - `random_shuffle`: rearranja os valores no intervalo de forma aleatória.
  - `shuffle`: rearranja os valores no intervalo de forma aleatória usando um gerador aleatório uniforme (definido em <random>).

Partições

- Partições são definidas quando a primeira parte do subarranjo satisfaz um predicado e a segunda parte não.
  - `is_partitioned`: verifica se o arranjo está particionado segundo algum predicado.
  - `partition (partition_copy)`: rearranja a partição a partir de um predicado (partição não estável), retorna um iterator para o primeiro elemento da segunda partição.
  - `stable_partition`: rearranja a partição a partir de um predicado (partição estável, a ordem dos elementos de uma partição é mantida).
  - `partition_point`: retorna o ponto que começa a segunda partição.

Ordenações

- Realiza ordenações sobre um arranjo, ou verifica.
  - `sort`: ordena os elementos em um arranjo.
  - `stable_sort`: ordena os elementos em um arranjo preservando a ordem de elementos equivalentes.
  - `partial_sort (partial_sort_copy)`: reordena os elementos, mas somente são apresentados ordenados os primeiros elementos (de todo o conjunto). O resto do conjunto segue a sequência original que estavam na sequência.
  - `is_sorted`: verifica se o arranjo está ordenado.
  - `is_sorted_until`: Encontra o primeiro elemento fora da ordenação.
  - `nth_element`: Troca os elementos para que o elemento da posição `n` seja o correto naquela posição em uma ordenação.

Busca Binária

- Opera sobre um arranjo previamente ordenado (ou parcialmente ordenado).
  - `lower_bound`: encontra o primeiro elemento que não satisfaz a condição de `menor`. O elemento pode ser igual ao valor de referência.
  - `upper_bound`: encontra o primeiro elemento que é maior que o valor de referência.
  - `equal_range`: obtém um subarranjo de elementos idênticos.
  - `binary_search`: verifica se um determinado elemento existe em uma sequência ordenada.

Intercalação

- Operação de intercalação entre arranjos ordenados
  - `merge`: realiza a intercalação de dois arranjos em um terceiro.
  - `inplace_merge`: realiza a intercação de dois intervalos ordenados em um único ordenado.
  - `includes`: verifica se um arranjo ordenado contém outro arranjo ordenado.
  - `set_union`: realiza a união de dois arranjos ordenados (elementos repetidos em ambos arranjos aparecem uma vez no final.
  - `set_intersection`: realiza a interseção de dois arranjos ordenados, indicando somente elementos que aparecem em ambos arranjos.
  - `set_difference`: apresenta os elementos que estão no primeiro conjunto mas não no segundo.
  - `set_symmetric_difference`: apresenta os elementos que estão presentes em um dos arranjos mas não no outro (união - interseção).

Heap

- Heap é uma representação de uma árvore binária cheia (ou parcialmente cheia) na forma de vetor. O heap pode ser ordenado de forma que nenhum descendente seja maior que um ascendente (Heap Máximo), ou o inverso (Heap Mínimo).
  - `push_heap`: insere elemento em um heap ordenado.
  - `pop_heap`: remove um elemento de um heap ordenado.
  - `make_heap`: constrói um heap ordenado a partir de um arranjo.
  - `sort_heap`: reconstrói um arranjo ordenado a partir de um heap.

Min/Max

- Estas funções não necessariamente trabalham em arranjo, ou arranjo ordenado. Mas representam alguma ordem.
  - `min`: retorna o menor de dois valores.
  - `max`: retorna o maior de dois valores.
  - `minmax`: retorna um `pair`, a partir de dois valores, sendo o primeiro elemento o menor e o segundo o maior.
  - `min_element`: retorna o menor elemento de um intervalo.
  - `max_element`: retorna o maior elemento em um intervalo.
  - `minmax_element`: retorna um `pair` com o menor e maior elementos de um intervalo

Outros

- Outras funções de `<algorithm>`
  - `lexicographical_compare`: comparação lexicográfica para palavras e ordenação alfabética. Os arranjos são `strings`
  - `next_permutation`: transforma o arranjo em uma permutação lexicográfica maior.
  - `prev_permutation`: transforma o arranjo em uma permutação lexicográfica menor.

Algoritmos numéricos genéricos: <numeric>

- `accumulate`: soma os valores em um intervalo, usando uma função de acumulação.
- `adjacent_difference`: obtém a diferença de elementos adjacentes, usando uma função de diferença
- `inner_product`: obtém o produto interno de dois arranjos, usando uma função de produto e uma função de acumulação. O produto interno é obtido somando-se o produto de pares correspondentes de ambos arranjos.
- `partial_sum`: obtém a soma partial de um intervalo. Um elemento é a soma dos anteriores. Utiliza uma função de soma.
- `iota`: obtém um arranjo de valores sucessivos a partir de um valor inicial. Os valores são obtidos por `++val`.

Exemplo: Problema 1104 do URI - Troca de Cartas

Alice e Beatriz colecionam cartas de Pokémon. As cartas são produzidas para um jogo que reproduz a batalha introduzida em um dos mais bem sucedidos jogos de videogame da história, mas Alice e Beatriz são muito pequenas para jogar, e estão interessadas apenas nas cartas propriamente ditas. Para facilitar, vamos considerar que cada carta possui um identificador único, que é um número inteiro. Cada uma das duas meninas possui um conjunto de cartas e, como a maioria das garotas de sua idade, gostam de trocar entre si as cartas que têm. Elas obviamente não têm interesse emtrocar cartas idênticas, que ambas possuem, e não querem receber cartas repetidas na troca.Além disso, as cartas serão trocadas em uma única operação de troca: Alice dá para Beatriz um sub-conjunto com N cartas distintas e recebe de volta um outro sub-conjunto com N cartas distintas.

Exemplo: Problema 1104 do URI - Troca de Cartas

As meninas querem saber qual é o número máximo de cartas que podem ser trocadas. Por exemplo, se Alice tem o conjunto de cartas 1, 1, 2, 3, 5, 7, 8, 8, 9, 15 e Beatriz o conjunto 2, 2, 2, 3, 4, 6, 10, 11, 11, elas podem trocar entre si no máximo quatro cartas. Escreva um programa que, dados os conjuntos de cartas que Alice e Beatriz possuem, determine o número máximo de cartas que podem ser trocadas.

Entrada:

A entrada contém vários casos de teste. A primeira linha de um caso de teste contém dois números inteiros A e B, separados por um espaço em branco, indicando respectivamente o número de cartas que Alice e Beatriz possuem ($1 \leqslant A \leqslant 10^4$ e $1 \leqslant B \leqslant 10^4$). A segunda linha contém A números inteiros X_i, separados entre si por um espaço em branco, cada número indicando uma carta do conjunto de Alice ($1 \leqslant X_i \leqslant 10^5$). A terceira linha contém B números inteiros Y_i, separados entre si por um espaço em branco, cada número indicando uma carta do conjunto de Beatriz ($1 \leqslant Y_i \leqslant 10^5$). As cartas de Alice e Beatriz são apresentadas em ordem não decrescente.

O final da entrada é indicado por uma linha que contém apenas dois zeros, separados por um espaço em branco.

Saída:

Para cada caso de teste da entrada seu programa deve imprimir uma única linha, contendo um numero inteiro, indicando o número máximo de cartas que Alice e Beatriz podem trocar entre si.

Solução:

```
#include <iostream> // cin, cout
#include <vector> // vector
#include <algorithm> // unique, set_difference
using namespace std;
int main() {
 int a, b, i, j, int va[10000], vb[10000];
 vector<int> da(10000), db(10000);
 vector<int>::iterator d1, d2;
 cin >> a >> b;
 while(a) {
  for(i=0; i<a; i++) cin >> va[i];
  for(j=0; j<b; j++) cin >> vb[j];
  a = unique(va,va+a) - va;
  b = unique(vb,vb+b) - vb;
  d1 = set_difference(va,va+a,vb,vb+b,da.begin());
  d2 = set_difference(vb,vb+b,va,va+a,db.begin());
  cout << min(d1-da.begin(),d2-db.begin()) << endl;
  cin >> a >> b;
 }
 return 0;
}
```

Exemplo: Problema 1196 do URI: WERTYU

Um erro comum de digitação é colocar as mãos no teclado uma posição à direita da correta posição. Desta forma, "Q"é digitado como "W"e "J"é digitado como "K"e assim por diante. Você deve decodificar a mensagem desta maneira.

Entrada:

A entrada consiste em várias linhas de texto. Cada linha pode conter dígitos, espaços e letras maiúsculas. (exceto Q, A, Z), ou pontuação, exceto crase (‘) conforme mostrado acima. Teclas rotuladas como palavras [Tab, BackSp, Control, etc.] não são representados na entrada. Você deverá repassar cada letra ou símbolo de pontuação pelo símbolo imediatamente à esquerda. Os espaços de entrada simplesmente deverão ser ecoados (impressos) na saída.

Saída:

Para cada linha de entrada, imprima uma linha de saída correspondente com a mensagem decodificada.

Solução:

```
#include <iostream> // cin, cout
#include <string> // string
#include <vector> // vector
using namespace std;
template<class RandomAccessIterator, class T>
 void setVec(RandomAccessIterator first, RandomAccessIterator last,
             vector<T>& v) {
  RandomAccessIterator it=first+1;
  while(it!=last) {
   v.at((int)(*it))=*(first); ++it; ++first;
  }
  return;
 }
template<class InputIterator, class OutputIterator, class T>
 OutputIterator chgStr(InputIterator first, InputIterator last,
                       OutputIterator result, vector<T> v) {
  while(first!=last) {
   *result = v.at((int)(*first)); ++result; ++first;
  }
  return result;
 }
```

Solução:

```
int main() {
 vector<char> letras(127,0);
 string entrada;
 string linha1=''`1234567890-=='';
 string linha2=''QWERTYUIOP[]\'';
 string linha3=''ASDFGHJKL;''';
 string linha4=''ZXCVBNM,./'';
 string linha5='' '';
 setVec(linha1.begin(),linha1.end(),letras);
 setVec(linha2.begin(),linha2.end(),letras);
 setVec(linha3.begin(),linha3.end(),letras);
 setVec(linha4.begin(),linha4.end(),letras);
 setVec(linha5.begin(),linha5.end(),letras);

 while(getline(cin,entrada)) {
  string saida(entrada.size(),' ');
  chgStr(entrada.begin(),entrada.end(),saida.begin(),letras);
  cout << saida << endl;
 }
 return 0;
}
```