



Maratona de Programação UNIFEI



Aula Extra | Standard Template Library (STL)

Introdução



Existe um bom livro?

Sim, o **guia de consulta rápida** de Joel Saade é uma mão na roda, especialmente quando apenas queremos nos lembrar sobre funções e algoritmos, e suas assinaturas. Guia de referência, mas sem muita profundidade teórica.

Introdução

A **STL** (***Standard Template Library***) é uma biblioteca de algoritmos e estruturas de dados genéricas, integrada à biblioteca padrão de C++ através de mecanismos de **templates** (ou **gabaritos**, em português). Criada por **Alexander Stepanov** e **Meng Lee** (Hewlett-Packard), adicionada ao C++ em 1994.

Mas o que são templates (gabaritos)?

São um recurso poderosíssimo da linguagem C++, que possibilita especificar, **com um único segmento de código**, uma gama inteira de classes relacionadas. Podemos, por exemplo, escrever um **único template** de classe para uma classe “pilha”, e então fazer com que o C++ gere várias classes templates separadas, tais como pilha de **int**, de **float**, de **double**, de **string** e assim por diante.

Exemplo

```
#include <stdio.h>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
    // Cria estrutura de dados e iterador (para percorre-la)
    vector<int> idades;
    vector<int>::iterator it;

    // insere dados
    idades.push_back(20);
    idades.push_back(17);
    idades.push_back(21);

    // imprime desordenado
    printf("Vetor Desordenado: ");
    for(it = idades.begin(); it != idades.end(); ++it)
    {
        printf("%d ", *it);
    }

    // ordena vetor
    sort(idades.begin(), idades.end());
    printf("\nVetor Ordenado: ");
    for(it = idades.begin(); it != idades.end(); ++it)
    {
        printf("%d ", *it);
    }

    return 0;
}
```

Repare que foi necessária a inclusão e duas bibliotecas: uma para utilização da classe **vector** e outra para a utilização do algoritmo de **ordenação**.

Introdução

A STL é uma **caixa de ferramentas**, que auxilia e traz soluções para muitos problemas de programação que envolvem estruturas de dados.

Estruturas de Dados

Uma estrutura de dados é uma forma de **armazenar e organizar dados**, provendo maneiras eficientes para a realização de inserções, consultas, buscas, atualizações e remoções.

Sabemos que as estruturas de dados em si **não representam a solução de um problema**. No entanto, conhecer as estruturas e utilizar a mais adequada em uma determinada situação **pode ser a diferença** entre acertar um problema e tomar uma penalização por limite de tempo excedido.

Introdução

Como o objetivo desta disciplina é a programação competitiva, não focaremos nossos esforços na fundamentação teórica das estruturas de dados ([ECO003/013](#)) e na análise dos algoritmos de ordenação e busca ([CCO005](#)).

Nós precisamos apenas conhecer o maior número possível delas e – o mais importante de tudo – **SABER UTILIZÁ-LAS!**

Precisamos conhecer:

- Pontos fortes e fracos
- Complexidade

Conhecer bem a STL irá nos **ajudar muito** nas competições. A utilização de suas estruturas e algoritmos prontos nos alcançará um alto nível de **produtividade e eficiência**.

Para começar, existem três conceitos básicos importantes que precisam ser conhecidos sobre a STL, para que possamos utilizá-la bem:

- Os **Containers** são as estruturas que armazenam valores de um tipo de dado (**int**, **float**, **string**, etc.) e encapsulam a estrutura de dados em si;
- Os **algoritmos** correspondem às ações a serem executadas sobre os containers (**ordenação**, **pesquisa**, etc.) e são utilizadas através de chamadas de funções;
- Os **iteradores** são componentes que percorrem os elementos de um container da mesma forma que um índice percorre os elementos de um *array* comum.

Containers

Containers são estruturas de dados implementadas na STL que armazenam valores (de tipos básicos ou criados).
Estão divididos em duas categorias:

- **Sequenciais:** `vector`, `deque` e `list`.
- **Adaptadores:** `stack`, `queue` e `priority_queue` (heap)
- **Associativos Classificados:** `set` e `map` (Balanced BST)

Dispõem de gerenciamento automático de memória, o que permite que o **tamanho do container varie dinamicamente**, aumentando ou diminuindo de acordo com a necessidade do programa.

Containers Sequenciais

- Nos **containers sequenciais**, os elementos estão em uma **ordem linear** na estrutura.
- Os tipos podem ser **básicos** (**int**, **float**, etc.) ou **criado** pelo programador (**structs**, **classes**)
- Os containers sequenciais são:
 - **vector**, **deque** e **list**.



Container: **vector**

Representam o mesmo tipo de estrutura de um array em C, e podem ser manipulados com a mesma eficiência, mas mudam seu tamanho dinamicamente e automaticamente.

- Neles, os elementos são **armazenados de forma contígua** e podem ser acessados aleatoriamente através do **operador []**.
- Sua utilização é importante especialmente quando **não conhecemos o tamanho do vetor** com antecedência.
- Suas principais operações são: **push_back()**, **at()**, **operador []**, **erase()**, **empty()**, **clear()** e **swap()**.
- Devemos utilizar um **iterador** para percorrê-lo.
- É necessário incluir o cabeçalho **<vector>**
- São bastante eficientes no acesso aos elementos e na inserção e remoção de elementos no seu fim (*push* e *pop_back*). Para operações de inserção e remoção de elementos em outros lugares, são piores que outras estruturas como **list** e **deque**.

Exemplo:

```

#include <vector>
#include <stdio.h>

using namespace std;

int main()
{
    vector<float> medidas;
    vector<float>::iterator it;

    // Imprimimos o tamanho do vetor (inicial)
    printf("Tamanho Inicial: %d\n", medidas.size());

    medidas.push_back(15.6);
    medidas.push_back(23.6);
    medidas.push_back(2.9);
    medidas.push_back(17.3);
    medidas.push_back(11.9);
    medidas.push_back(7.7);

    // Imprimimos o tamanho do vetor (final)
    printf("Tamanho Final: %d\n", medidas.size());

    // Podemos acessar itens individualmente
    printf("\nSegundo item: %.1f \n", medidas[1]);
    printf("Quinto item: %.1f \n", medidas[4]);
    printf("Terceiro item: %.1f \n", medidas.at(2));
    printf("Primeiro item: %.1f \n", medidas.at(0));

    // Imprime vetor com iterador
    printf("\nVetor Inicial: ");
    for(it = medidas.begin(); it < medidas.end(); ++it)
        printf("%.1f ", *it);
    printf("\n");

    // Podemos apagar itens (precisamos do iterador)
    medidas.erase(medidas.begin()); // apaga primeiro item
    medidas.erase(medidas.begin() + 2); // Apaga o terceiro item (primeiro já removido)

    // Imprime vetor com iterador
    printf("Vetor Resultante: ");
    for(it = medidas.begin(); it < medidas.end(); ++it)
        printf("%.1f ", *it);
    printf("\n");

    return 0;
}

```

Container: deque

Deque (acrônimo de “*double-ended queue*”) é, como o próprio nome nos indica, uma fila com duas extremidades. São um tipo de container de tamanho dinâmico que pode ser expandido ou diminuído nas duas extremidades (*front* e *back*).

- Também permitem o acesso direto a todos os seus elementos.
- Proveem funcionalidade similar aos vectors, mas com inserção e remoção eficiente de elementos também no começo da sequência, não somente no fim.
- Não é possível garantir que todos os seus elementos estarão alocados sequencialmente na memória. Acesso de elementos utilizando offset de um ponteiro leva a um comportamento inesperado.
- Suas principais operações são: **push_back()**, **push_front()**, **at()**, **erase()**, **empty()**, **clear()** e **swap()**, operador **[]**.
- É necessário incluir o cabeçalho **<deque>**
- Exemplo:

```

#include <stdio.h>
#include <deque>

using namespace std;

int main()
{
    deque<int> dados;

    // Podemos inserir elementos no inicio e fim da fila
    dados.push_front(1);
    dados.push_front(2);
    dados.push_front(3);
    dados.push_back(9);
    dados.push_back(10);

    printf("Tamanho da deque: %d.\n", dados.size());

    printf("Elementos: ");
    for(deque<int>::iterator it = dados.begin(); it < dados.end(); ++it)
        printf("%d ", *it);
    printf("\n");

    // também podemos remover elementos das duas extremidades
    dados.pop_back();
    dados.pop_front();

    // Acesso aleatório (inesperado)
    printf("Terceiro elemento (?): %d \n", dados[2]);

    printf("Elementos: ");
    for(deque<int>::iterator it = dados.begin(); it < dados.end(); ++it)
        printf("%d ", *it);
    printf("\n");

    return 0;
}

```

Container: **list**

São **listas duplamente encadeadas**, que permitem operações de inserção e remoção de itens em tempo constante em qualquer posição da sequência e iteração nos dois sentidos.

- Armazenam elementos de maneira não contígua, mantendo a ordem internamente através de uma associação: cada elemento possui uma ligação com seu antecessor e sucessor na lista. Não é possível utilizar o operador [].
- Em comparação com deque e vector, a list se sai melhor nas operações de inserção, remoção e movimentação de elementos em qualquer posição do container para a qual um iterador já tenha sido obtido, e, portanto, se destaca também em algoritmos que fazem uso intensivo destas operações, como algoritmos de ordenação.
- Possui todas as operações já citadas. Possui uma operação **remove**, que remove elementos com um valor específico.
- É necessário incluir o cabeçalho **<list>**

Container: **list**

- A maior desvantagem das lists é que, comparadas com o vector e deque, elas **não possuem acesso direto aos elemento** pela simples especificação de sua posição, seja utilizando a função **at()** ou o **operador []**.
- Para acessar, por exemplo, o décimo elemento da lista, é necessário iterar a partir de uma posição conhecida (como seu início ou fim) até aquela posição, o que leva um tempo linear na distância entre eles.
- Vejamos um exemplo da utilização de *lists*:

```

#include <iostream>
#include <list>
#include <string>

using namespace std;

int main()
{
    list<string> nomes;
    list<string>::iterator it;

    // Inserindo dados(back, front, random)
    nomes.push_back("Joao");
    nomes.push_back("Paulo");
    nomes.push_front("Roberto");

    it = nomes.begin();
    nomes.insert(it, "Roberto");
    ++it; ++it;
    nomes.insert(it, "Edmilson");

    cout << "Lista de Nomes: " << endl;
    for(it = nomes.begin(); it != nomes.end(); ++it)
        cout << *it << endl;

    // Removendo todos os Robertos
    nomes.remove("Roberto");
    cout << "\nLista de Nomes: " << endl;
    for(it = nomes.begin(); it != nomes.end(); ++it)
        cout << *it << endl;

    return 0;
}

```


Adaptadores de Container

- Adaptadores de Container são classes que **encapsulam um container** específico e nos proveem uma interface pública apenas com um conjunto de funções membro, adaptadas para aquele tipo de estrutura que se quer simular.
- Não suportam iteradores e, portanto, não podem ser utilizados com os algoritmos da STL.
- Não permitem acesso aleatório a seus elementos.
- Os adaptadores de container são:
 - **Pilha** (stack)
 - **Fila** (queue)
 - **Fila com Prioridade** (priority_queue)
 - Vou falar dela um pouco mais pra frente. É uma estrutura não-linear – uma heap.

Adaptadores: **stack**

- **stack** é um adaptador de container projetado especificamente para operar em um contexto **LIFO** (*last in, first out*), onde elementos são SEMPRE inseridos e removidos apenas do final (topo) do container.
- Trabalha como uma pilha da vida real.
 - Último a entrar, primeiro a sair.
- Cabeçalho: **<stack>**
- Operações básicas:
 - **empty**
 - **size**
 - **top**
 - **push** (push_back)
 - **pop** (pop_back)



Adaptadores: **queue**

- **queue** é um adaptador de container projetado especificamente para operar em um contexto FIFO (first in, first out), onde elementos são SEMPRE inseridos no fim e removidos apenas do início do container.
- Funciona exatamente como uma fila da vida real.
 - Primeiro a entrar, primeiro a sair.
- Cabeçalho: **<queue>**
- Operações básicas:
 - **empty**
 - **size**
 - **front**
 - **back**
 - **push** (push_back)
 - **pop** (pop_front)



```

#include <iostream>
#include <stack>
#include <queue>

using namespace std;

int main()
{
    stack<int> pilha;
    queue<int> fila;

    // insere na pilha e na fila
    for(int i = 0; i < 5; i++)
    {
        pilha.push(i);
        fila.push(i);
    }

    // imprime pilha
    cout << "Pilha: ";
    while(!pilha.empty())
    {
        cout << pilha.top() << " ";
        pilha.pop();
    }

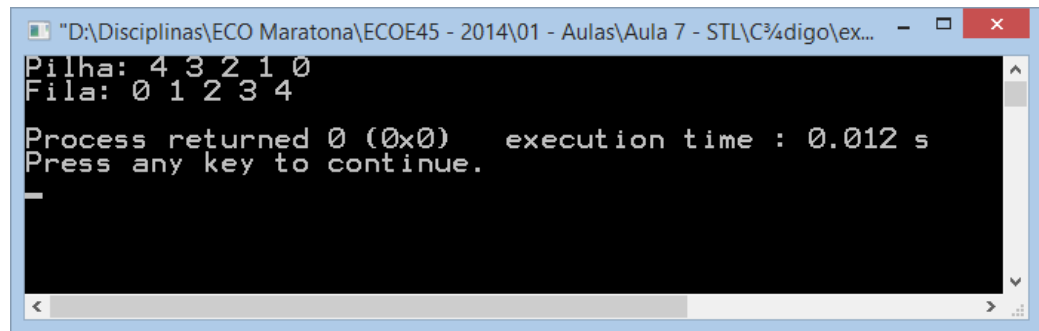
    cout << endl;

    // imprime fila
    cout << "Fila: ";
    while(!fila.empty())
    {
        cout << fila.front() << " ";
        fila.pop();
    }

    cout << endl;

    return 0;
}

```



```

"D:\Disciplinas\ECO Maratona\ECOE45 - 2014\01 - Aulas\Aula 7 - STL\C%4digo\ex...
Pilha: 4 3 2 1 0
Fila: 0 1 2 3 4

Process returned 0 (0x0)   execution time : 0.012 s
Press any key to continue.

```

Sobre complexidade

- Existem, basicamente, duas operações que podem ser realizadas em estruturas sequenciais:
Ordenação e busca.
- Ordenação
 - Existe uma série de algoritmos de ordenação baseados em comparação, que tomam o tempo $O(n^2)$. Precisam ser compreendidos, porém evitados.
Ex.: Bubble, Selection/Insertion sort.
 - Existem outros algoritmos, também baseados em comparação, que possuem complexidade $O(n \log n)$. Estes devem ser utilizados quando necessário.
Ex.: Merge, Heap, quick Sort.
 - Implementações em STL: `sort`, `partial_sort`, `stable_sort`.

Sobre complexidade

- Existem, ainda, outros algoritmos de ordenação, mas eles possuem aplicações bem específicas: *Counting*, *Radix*, *Bucket sort*. É interessante conhecê-los, mas eles **não são frequentes em competições** (Veja CCO005).
- Busca
 - **Busca linear**, $O(n)$, quando buscamos um elemento percorrendo totalmente o vetor, do índice 0 a $n-1$. Deve ser evitada em competições.
 - **Busca Binária**, $O(\log n)$, implementada como `lower_bound` (ou `binary_search`) em STL. Necessita que a estrutura esteja ordenada – portanto, é interessante que ordenemos a estrutura apenas uma vez, utilizando um algoritmo de $O(n \log n)$, se formos utilizar a busca binária, que é $O(\log n)$, muitas vezes.

Estruturas não-lineares

- Para alguns problemas, existem maneiras melhores de se representar os dados do que uma simples sequência. Com as implementações da STL das estruturas não-lineares que iremos discutir a seguir, podemos realizar buscar muito mais rápidas e acelerar nossos algoritmos, quando as condições forem favoráveis.
- De quais estruturas estamos falando?
 - **Balanced Binary Search Tree (BST)**
 - STL: `<map>/<set>`
 - **Heap**
 - STL: `<queue>` : `priority_queue`

Árvores Binárias

- Maneira de se organizar dados em forma de árvore.
- Em cada sub-árvore com raiz em x , os itens na sub-árvore à **esquerda** de x são **menores** x e os itens na sub-árvore à **direita** de x são **maiores ou iguais** a x .
- Este tipo de organização permite inserção, busca e remoção em $O(\log n)$, mas somente funciona se a árvore for balanceada (AVL, RB-Tree).
- STL **<map>** e **<set>** são implementações de um tipo de árvore binária balanceada chamada **Red-Black Tree**. Portanto, nelas, todas as operações são realizadas em $O(\log n)$.
- **Qual a diferença?**
 - **<map>** armazena pares (key, data)
 - **<set>** armazena apenas (key)

set

```
#include <iostream>
#include <set>

using namespace std;

int main()
{
    set<int> conjunto;
    set<int>::iterator it;

    // Em um set, as chaves não podem ser duplicadas.
    // para isso, utilize um multiset (também de <set>
    conjunto.insert(30);
    conjunto.insert(20);
    conjunto.insert(10);
    conjunto.insert(10); // não será inserido
    conjunto.insert(20); // não será inserido
    conjunto.insert(40);
    conjunto.erase(10); // apaga um item, baseado no valor.

    cout << "Quantidade de elementos: " << conjunto.size() << endl;
    cout << "Elementos: ";
    // perceba que os elementos em um set estão sempre ordenados
    for(it = conjunto.begin(); it != conjunto.end(); ++it)
        cout << *it << " ";
    cout << endl;

    return 0;
}
```

ex6

map

```
#include <map>
#include <iostream>

using namespace std;

int main()
{
    map<int, string> alunos;
    map<int, string>::iterator it;

    // inserindo quatro alunos
    alunos.insert(make_pair(11984, "Joao Paulo"));
    alunos.insert(make_pair(23456, "Jose"));
    alunos.insert(make_pair(8541, "Carlos"));
    alunos.insert(make_pair(8541, "Edmilson")); // Não é inserido (multimap)
    alunos.insert(pair<int, string> (29546, "Maria"));

    // Removendo José
    alunos.erase(23456);

    // Imprimindo lista de alunos
    // Repare que o map está ordenado de acordo com a chave
    cout << "Lista de ALunos: " << endl;
    for(it = alunos.begin(); it != alunos.end(); ++it)
        cout << it->first << " - " << it->second << endl;

    return 0;
}
```

ex7

Heap

- Heap é outra maneira de organizar dados em forma de árvore. No entanto, possui propriedades diferentes da BST.
- Para cada sub-árvore com raiz em x , itens das sub-árvores esquerda e direita são menores do que x . Esta propriedade garante que o topo da heap sempre terá o elemento de valor máximo.
- A árvore precisa ser completa pelo menos até seu penúltimo nível. No último nível, os itens estão “justificados” à esquerda.
- Normalmente não trabalhamos com busca na heap, mas inserção e remoção podem ser feitas em $O(\log n)$.
- Pode ser modelada como uma fila de prioridade.
- Em STL, está em `<queue>` implementada como `priority_queue`.
- É importante em uma série de algoritmos como **Dijkstra**, **Kruskal** e na ordenação **heap sort**, implementada em **partial_sort** e realizada em $O(k \log n)$ quando ordenamos k elementos.

Heap

Priority queue (STL)

Filas de prioridade são um tipo de adaptador de container projetado especificamente para que seu **primeiro elemento seja sempre o maior entre todos os elementos**, de acordo com um critério de ordenação.

O contexto é, portanto, similar ao de uma heap, onde elementos podem ser inseridos a qualquer momento, e somente o elemento máximo da heap pode ser obtido (aquele no topo da fila de prioridade, `pop_back`).

Funções membro:

- **empty**
- **size**
- **top**
- **push** (`push_back`)
- **pop** (`pop_back`)

```
#include <iostream>
#include <queue>

using namespace std;

int main()
{
    priority_queue<float> distancias;

    // insere valores de distancias
    distancias.push(1000.0);
    distancias.push(100.0);
    distancias.push(10.0);
    distancias.push(1001.0);
    distancias.push(900.0);

    cout << "Imprimindo na ordem de prioridade: " << endl;
    while(!distancias.empty())
    {
        cout << distancias.top() << endl;
        distancias.pop();
    }

    return 0;
}
```

Alguns Algoritmos...

Ordenação

Para competições, não se preocupe em conhecer TODOS os métodos passo a passo. Em geral, o que precisamos é apenas utilizar a função de ordenação $O(n \log n)$ presente na STL.

Na Maratona, ordenação normalmente é apenas um **passo preliminar** para um algoritmo mais complexo ou um **último passo**, para formatar corretamente a saída. **Difilmente será o objetivo do programa.**

Familiaridade com biblioteca de ordenação é OBRIGATÓRIA.



Alguns Algoritmos...

Na STL,

Temos três algoritmos prontos (biblioteca `<algorithm>`):

- **sort:** O algoritmo específico não é fixo e pode variar dependendo da implementação. No entanto, a complexidade no pior caso é, obrigatoriamente, $O(n \log n)$.
 - Bastante **rápido**;
 - Ordena tanto dados **básicos** quanto tipos **definidos pelo usuário**.
- **partial_sort:** Implementa a heap sort e pode ser utilizado para ordenar apenas uma parte da estrutura. Se for necessário ordenar k itens, sua complexidade no tempo será de $O(k \log n)$.
- **stable_sort:** Preserva ordem e elementos com o mesmo valor, se necessário.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <ctime>
#include <cstdlib>

using namespace std;

// Função de comparação
bool compare(int i, int j)
{
    return (i < j);
}

int main()
{
```

```
    vector<int> numeros;
    srand(time(NULL));
```

```
    // Sorteando numeros aleatorios e imprimindo
```

```
    cout << "Vetor sorteado: ";
    for(int i = 0; i < 10; i++)
    {
        int x = rand()%100;
        numeros.push_back(x);
        cout << x << " ";
    }
    cout << endl;
```

```
    // Ordenando primeira metade (heap sort)
```

```
    partial_sort(numeros.begin(), numeros.begin()+5, numeros.end());
    cout << "Primeira metade ordenada: ";
    for(int i = 0; i < 10; i++)
        cout << numeros[i] << " ";
    cout << endl;
```

```
    // Ordenando segunda metade (sort normal)
    sort(numeros.begin() + 5, numeros.end());
    cout << "Segunda metade ordenada: ";
    for(int i = 0; i < 10; i++)
        cout << numeros[i] << " ";
    cout << endl;
```

```
    // Ordenando tudo (com função de comparação e ordenação estável)
    stable_sort(numeros.begin(), numeros.end());
    cout << "Vetor ordenado: ";
    for(int i = 0; i < 10; i++)
        cout << numeros[i] << " ";
    cout << endl;

    return 0;
}
```

```
"D:\Disciplinas\ECO Maratona\ECOE45 - 2014\01 - Aulas\Aula 7 - STL\C%4digo\ex... - [X]
Vetor sorteado: 63 30 61 69 40 6 6 77 55 24
Primeira metade ordenada: 6 6 24 30 40 69 63 77 61 55
Segunda metade ordenada: 6 6 24 30 40 55 61 63 69 77
Vetor ordenado: 6 6 24 30 40 55 61 63 69 77

Process returned 0 (0x0)   execution time : 0.018 s
Press any key to continue.
```


Alguns Algoritmos...

Busca

Existem dois casos:

1. Quando o vetor já se encontra ordenado (linear)
2. Caso contrário (logarítmica)

A busca pode ser feita **linearmente** quando o vetor não estiver ordenado. Trivial, com complexidade linear $O(n)$.

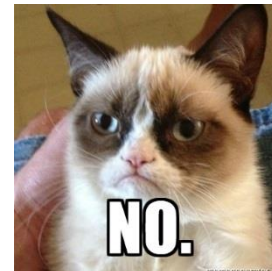
Quando ordenado, podemos utilizar a **busca binária**, que possui complexidade $O(\log n)$. Não é necessário implementar do zero, há **implementação na STL**.

Compensa ordenar primeiro e depois aplicar a busca binária?

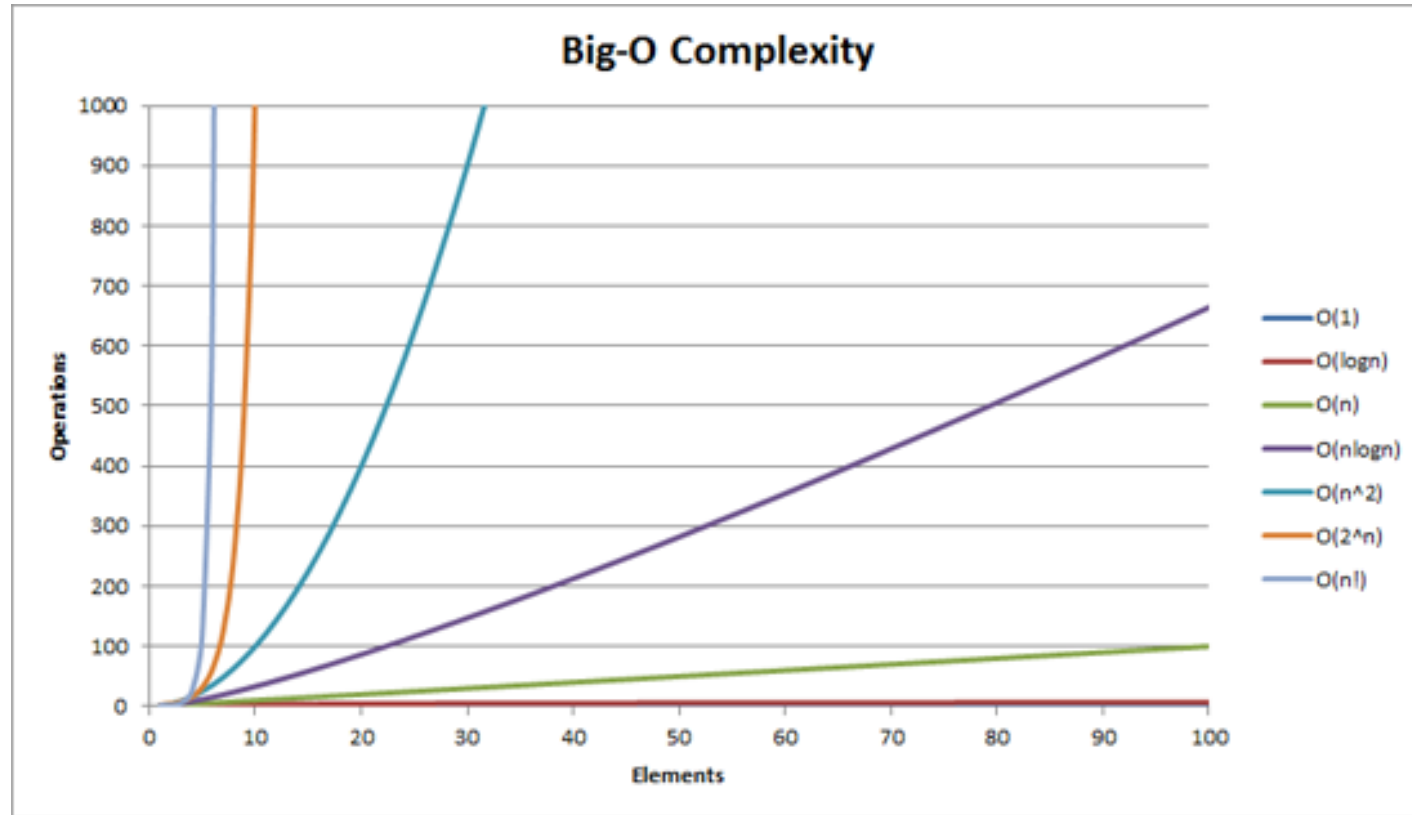
Ordenação + busca = $O(n \log n) + O(\log n) = O(n \log n)$

(princípio da **absorção**)

Mas $O(n \log n)$ não é melhor que $O(n)$? **Não.**



Alguns Algoritmos...



$O(n \log n)$ é pior que $O(n)$. **Caso encerrado!**

Alguns Algoritmos...

Na STL,

Temos dois algoritmos prontos (biblioteca **<algorithm>**). Nos dois casos, é necessário que o vetor esteja previamente ordenado. Veja:

- **binary_search**: Recebe o intervalo de busca e o valor a ser buscado. Retorna *true* se encontrar e *false* caso contrário.
Desvantagem: Não retorna uma referência para o elemento encontrado (iterator).
- **lower_bound**: Recebe os mesmos parâmetros da `binary_search`. No entanto, retorna um iterador apontando para o primeiro elemento no intervalo que não é menor do que o valor buscado.
Vantagem: Retorna o próprio elemento, caso necessário processá-lo.
- Veja o exemplo:

```

#include <iostream>
#include <algorithm>
#include <vector>
#include <ctime>
#include <cstdlib>

using namespace std;

int main()
{
    vector<int> numeros;
    srand(time(NULL));

    // Sorteando numeros aleatorios e ordenando vetor
    for(int i = 0; i < 30; i++)
    {
        int x = rand()%100;
        numeros.push_back(x);
    }
    sort(numeros.begin(), numeros.end());

    int valor = -1;
    while(true)
    {
        cout << "Entre com um valor a ser buscado: ";
        cin >> valor;

        if(binary_search(numeros.begin(), numeros.end(), valor))
        {
            vector<int>::iterator it = lower_bound(numeros.begin(), numeros.end(), valor);
            cout << "Numero encontrado: " << *it << endl;
            break;
        }
        else
            cout << "Nao encontrado." << endl;
    }

    return 0;
}

```

Alguns Exercícios...



Lista de Exercícios 1 – URI Academic

Enviar todos os exercícios até 10/03

Pontos extras.



UVa 482 – Permutation Array (arrays)

UVa 514 – Rails (stack)

UVa 336 – A Node Too Far (queue, BFS)

UVa 10226 – Hardwood Species (map)

UVa 11492 – Babel (priority_queue, dijkstra)

Para seu estudo em casa...

Referência STL (C++ reference)

<http://www.cplusplus.com/reference/stl/>

Árvore Rubro Negra – Red Black tree

http://pt.wikipedia.org/wiki/%C3%81rvore_rubro-negra

Heaps

<http://pt.wikipedia.org/wiki/Heap>

Métodos de Ordenação

- Bubble, selection, insertion, merge, quick, heap, counting, radix e bucket sort.

**FINALLY CLASS IS
OVER**

**NOW I CAN GO HOME AND
STUDY**