# Técnicas para Programação Competitiva

## Problemas com Ordenação e Busca

Samuel da Silva Feitosa

Aula 5

# Problemas com Ordenação

# Algoritmos de Ordenação

- Muitos algoritmos eficientes são baseados na ordenação dos dados de entrada, pois ordenando frequentemente torna o problema mais fácil de resolver.
- A implementação dos algoritmos tradicionais de ordenação já foram estudados em outros componentes curriculares.
  - bubble sort, merge sort, quick sort, counting sort, etc.
- Nesta disciplina, vamos utilizar a implementação presente na STL do C++.
  - Veremos alguns exemplos de como a ordenação pode ser usada como subrotina para criar algoritmos eficientes.

# Ordenação na Prática

- Na prática, praticamente nunca é uma boa ideia implementar um algoritmo de ordenação do zero.
  - Isto porque todas as linguagens modernas possuem bons algoritmos de ordenação em suas STLs.
  - Existem diversas razões para usar uma função presente na biblioteca: ela é certamente correta e eficiente, e também fácil de usar.
- Em C++, a função *sort* é eficiente O(*n* log *n*).

```
vector<int> v = {4,2,5,3,5,8,3};
sort(v.begin(),v.end());
```

```
sort(v.rbegin(),v.rend());
```

# Operadores de Comparação (1)

- A função *sort* usa o operador de comparação do tipo de dado presente no container.
  - A maioria dos tipos de dados do C++ possuem operadores de comparação.
  - Números são ordenados de acordo com seus valores, strings em ordem alfabética, etc.
- Exemplos com outros tipos de dados (pares e tuplas):

```
vector<pair<int,int>> v;
v.push_back({1,5});
v.push_back({2,3});
v.push_back({1,2});
sort(v.begin(), v.end());
// result: [(1,2),(1,5),(2,3)]
```

```
vector<tuple<int,int,int>> v;
v.push_back({2,1,4});
v.push_back({1,5,3});
v.push_back({2,1,3});
sort(v.begin(), v.end());
// result: [(1,5,3),(2,1,3),(2,1,4)]
```

# Operadores de Comparação (2)

- Também é possível utilizar uma função externa para representar a comparação entre os elementos.
  - Por exemplo, a função *comp* ordena as strings primeiramente pelo comprimento e depois pela ordem alfabética.

```
bool comp(string a, string b) {
    if (a.size() == b.size()) return a < b;
    else return a.size() < b.size();
}

sort(v.begin(), v.end(), comp);
```

# Resolvendo Problemas com Ordenação

# Resolvendo Problemas com Ordenação

- Frequentemente é possível resolver problemas em tempo $O(n^2)$ usando algoritmos de força bruta.
  - Mas tal algoritmo é lento se o tamanho da entrada for muito grande.
  - Um objetivo frequente é projetar algoritmos com tempo $O(n)$ ou $O(n \log n)$ para problemas que são resolvidos de forma trivial em $O(n^2)$.
  - Ordenar um conjunto de dados é uma maneira de atingir tal objetivo.

# Uniqueness

- Verificar se todos os elementos de um array são únicos:
  - Algoritmo de força bruta O(n²). Verifica todas as possibilidades.

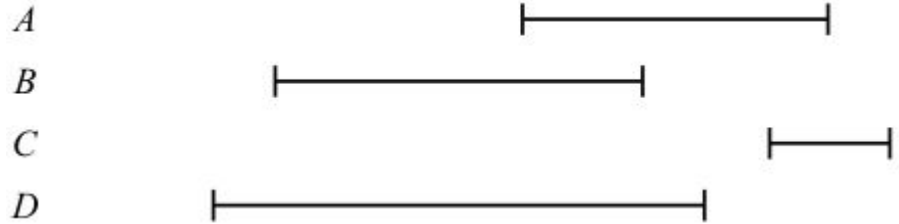
```
bool ok = true;
for (int i = 0; i < n; i++) {
    for (int j = i+1; j < n; j++) {
        if (array[i] == array[j]) ok = false;
    }
}
```

  - A versão mais eficiente primeiro ordena o array em O(n *log* n) e então verifica os pares sequenciais para verificar se são iguais.

```
bool ok = true;
sort(array, array+n);
for (int i = 0; i < n-1; i++) {
    if (array[i] == array[i+1]) ok = false;
}
```

# Algoritmos sweep line

- Modela um problema como um conjunto de eventos que são processados de forma ordenada.
  - Por exemplo, encontrar o número máximo de clientes simultâneos dados os horários de chegada e saída em um determinado dia.

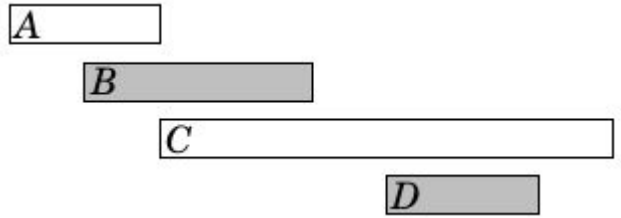



  - Solução: criar dois eventos para cada cliente (chegada e saída). Então, ordenar os eventos e percorrê-los de acordo com seu tempo. Um contador incrementa quando um cliente chega e decrementa quando um cliente sai, mantendo o maior.

# Problema do Agendamento (1)

- Muitos problemas de agendamento podem ser resolvidos usando algoritmos gulosos. Vejamos um problema clássico:
  - Dados *n* eventos com seus horários de início e fim, encontrar uma organização de agendamento que inclua o máximo de eventos possível.

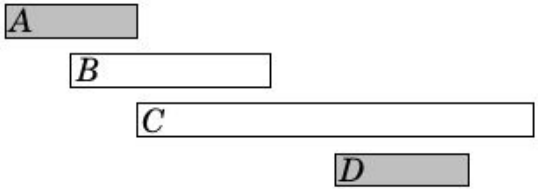
| event | starting time | ending time |
|---|---|---|
| *A* | 1 | 3 |
| *B* | 2 | 5 |
| *C* | 3 | 9 |
| *D* | 6 | 8 |



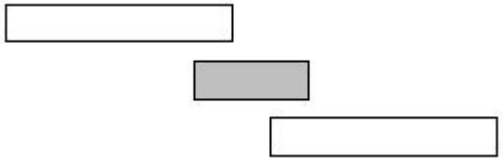

  - É possível utilizar diversos algoritmos gulosos para este problema, mas qual destes resolve o problema para todos os casos?

# Problema do Agendamento (2)

Ideia 1: selecionar os menores eventos.

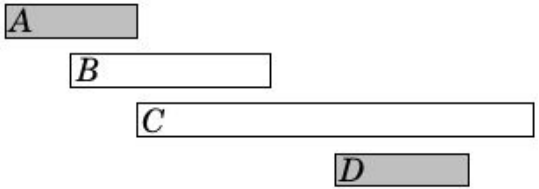



Contra exemplo:

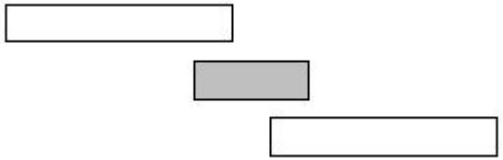


# Problema do Agendamento (2)

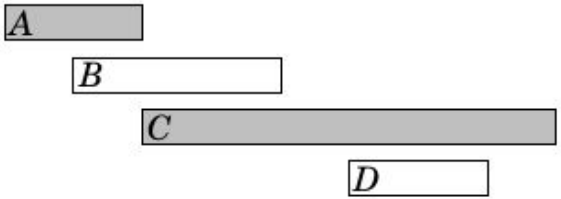
Ideia 1: selecionar os menores eventos.



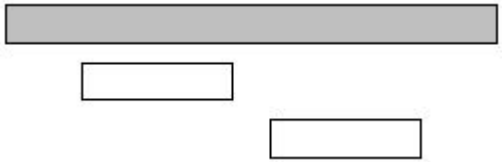

Contra exemplo:



Ideia 2: selecionar próximo que inicia o mais cedo possível.

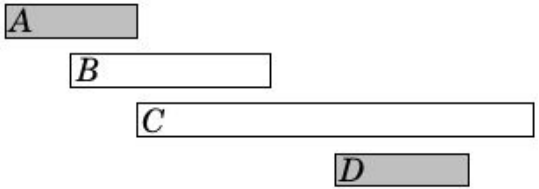



Contra exemplo:

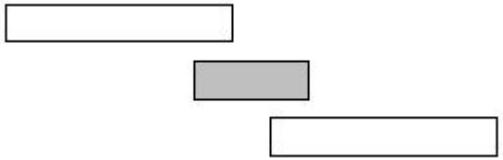


# Problema do Agendamento (2)

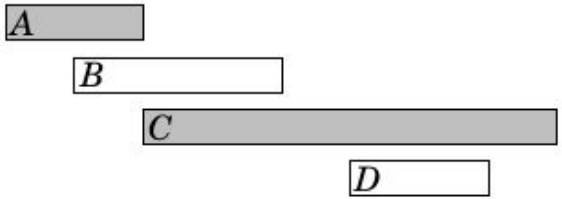
Ideia 1: selecionar os menores eventos.



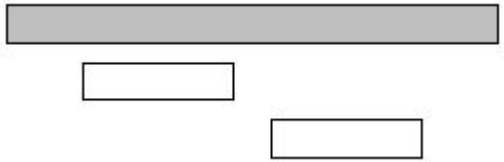

Contra exemplo:



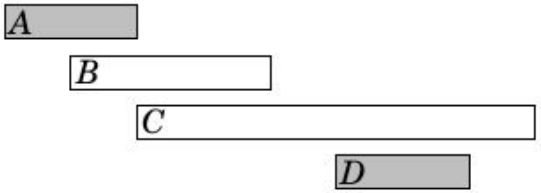
Ideia 2: selecionar próximo que inicia o mais cedo possível.




Contra exemplo:



Ideia 3: selecionar próximo que finaliza o mais cedo possível.



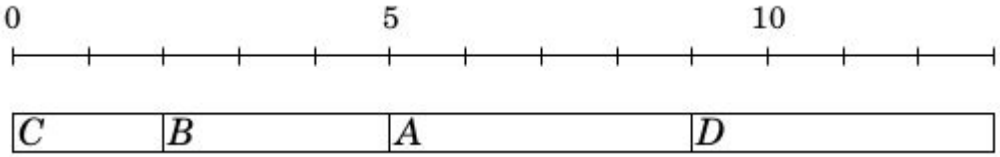

Este algoritmo **sempre** produz uma solução ótima.

# Tarefas e Deadlines

- Este problema recebe *n* tarefas com durações e deadlines e a solução deve encontrar uma ordem para executar as tarefas.
  - Para cada tarefa são registrados *d* - *x* pontos, onde *d* é o deadline e *x* o momento que a tarefa foi finalizada.
  - Qual é o maior valor possível que podemos obter?

| task | duration | deadline |
|---|---|---|
| *A* | 4 | 2 |
| *B* | 3 | 5 |
| *C* | 2 | 7 |
| *D* | 4 | 5 |

**Solução ótima: C = 5 , B = 0 , A = -7 e D = -10 pontos.**

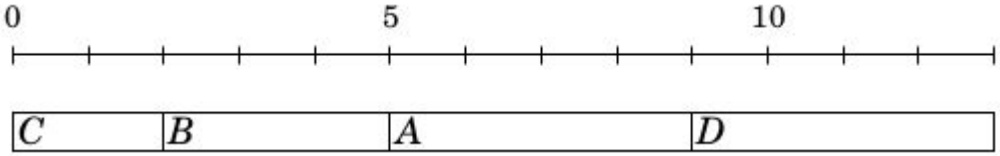


# Tarefas e Deadlines

- Este problema recebe *n* tarefas com durações e deadlines e a solução deve encontrar uma ordem para executar as tarefas.
  - Para cada tarefa são registrados *d* - *x* pontos, onde *d* é o deadline e *x* o momento que a tarefa foi finalizada.
  - Qual é o maior valor possível que podemos obter?

| task | duration | deadline |
|---|---|---|
| *A* | 4 | 2 |
| *B* | 3 | 5 |
| *C* | 2 | 7 |
| *D* | 4 | 5 |

**Solução ótima: C = 5 , B = 0 , A = -7 e D = -10 pontos.**




  - **A solução ótima para o problema não depende dos deadlines. A estratégia correta é simplesmente executar as tarefas ordenadas pela sua duração.**

# Problemas com Busca

# Busca Binária

- A busca binária (binary search) é um algoritmo de tempo O(log *n*) que pode ser usado para eficientemente verificar se um dado elemento está contido em um array ordenado.
  - Implementado através da técnica de **divisão e conquista**.
  - Pode ser implementado através de algoritmo iterativo ou recursivo.
  - Já estudado em outras disciplinas do curso.

# Funções C++

- *binary_search*
  - Retorna *true* se qualquer elemento na faixa informada é equivalente a *x*. Retorna *false*, caso contrário.
- *lower_bound*
  - Retorna um *iterator* (ponteiro) para o primeiro elemento do array com valor de pelo menos *x*.
- *upper_bound*
  - Retorna um *iterator* (ponteiro) para o primeiro elemento do array com valor maior do que *x*.
- *equal_range*
  - Retorna ambos os ponteiros acima.

# Considerações Finais

- Nesta aula vimos alguns exemplos de algoritmos que utilizam as técnicas de divisão e conquista e algoritmos gulosos.
- Os problemas de ordenação e busca constituem base importante para exercícios de programação competitiva.
  - É importante ter contato com diferentes problemas que usam essas técnicas para que seja mais fácil elaborar ou adaptar soluções.