Revisão P2

emanoelim@utfpr.edu.br

Conteúdos

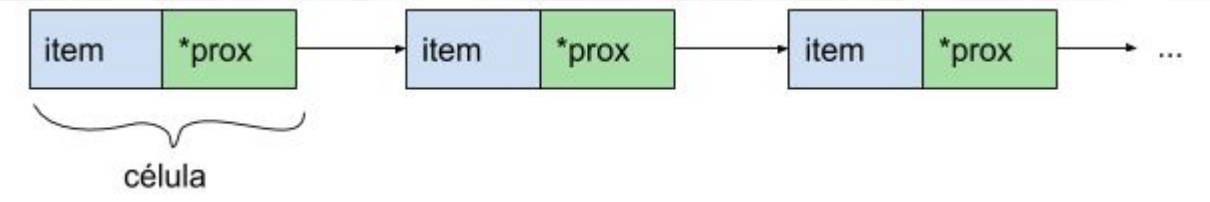
- Listas encadeadas;
- Algoritmos recursivos;
- Algoritmos de ordenação;
- Algoritmos de pesquisa;

Lista encadeadas

- Em listas implementadas usando arranjos/vetores existem alguns inconvenientes:
  - É necessário ter previamente uma ideia do tamanho máximo da lista, para alocar a memória necessária no momento da sua criação.
  - Como os itens da lista ficam em posições contíguas de memória, para adicionar ou remover itens do meio da lista, é preciso deslocar todos os itens após o item adicionado/removido.

Lista encadeadas

- Uma solução a estes problemas, é implementar uma lista encadeada.
- Uma lista encadeada é uma sequência de células: cada célula contém um item e um ponteiro com o endereço da próxima célula.



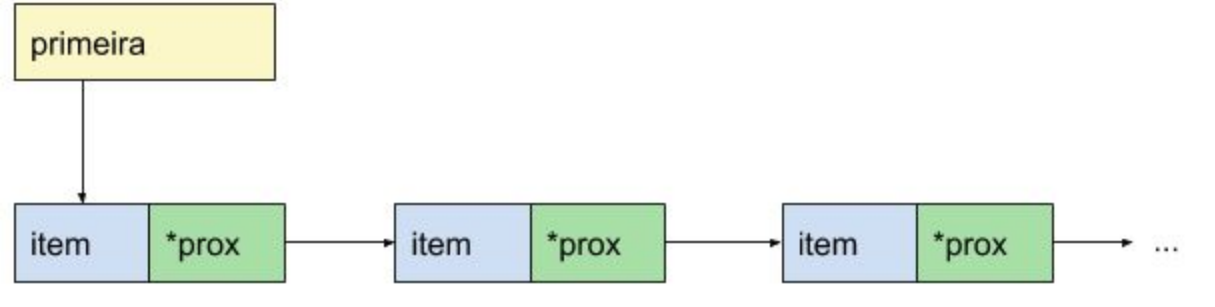
Listas encadeadas

```c
typedef struct item Item;
typedef struct celula Celula;
typedef struct lista Lista;

struct item { // item que vai ser guardado na lista
  int chave;
  // demais campos;
};
struct celula { // guarda um item e um ponteiro para a próxima célula da lista
  Item item;
  Celula *prox;
};
struct lista { // guarda o endereço da 1ª célula de lista. A partir dela, as demais são acessíveis
  Celula *primeira;
};
```

Listas encadeadas

- A estrutura anterior representa a seguinte ideia:

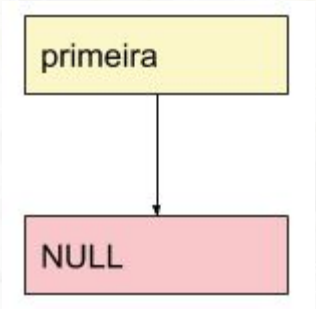


Lista encadeadas

- Operações básicas de um TAD Lista:
  - Criar lista vazia;
  - Inserir item na lista;
  - Excluir item da lista;
  - Buscar item da lista;
  - Imprimir a lista;
  - Liberar a lista.

Lista encadeadas

- Criar lista vazia



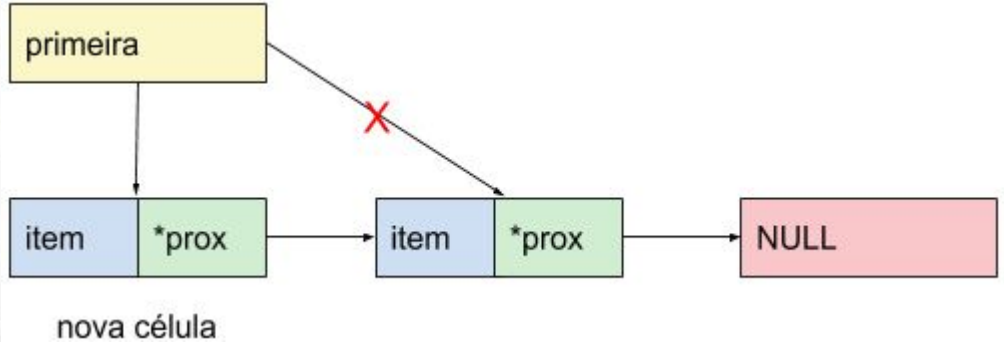

```
Lista      *      cria_lista_vazia()      {
     Lista   *l   =   malloc(sizeof(Lista));
                l->primeira       =       NULL;
                          return              l;
}
```

Lista encadeadas

- Inserir item na lista
  - Em listas encadeadas o conceito de lista cheia não existe, então não é necessário verificar se a lista está cheia ou não antes de inserir um item.
  - A inserção pode ser de 3 tipos:
    - No começo da lista;
    - No meio da lista (posição qualquer);
    - No fim da lista;

Lista encadeadas

- Inserir item na lista - inserção no início:



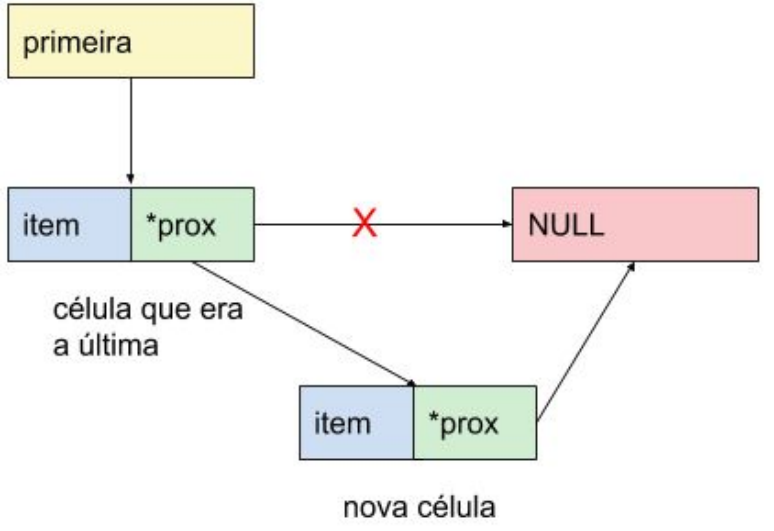
Lista encadeadas

- Inserir item na lista - inserção no início:

```
void insere_inicio_lista(Lista *l, int chave) {
  // criação e valorização do novo item
  Item novo;
  novo.chave = chave;
  // criação da nova célula que vai guardar o item
  Celula *nova = malloc(sizeof(Celula));
  nova->item = novo;
  // inserção - a prox. da nova célula é aquela que era a 1ª
  nova->prox = l->primeira;
  // a 1ª agora é a nova célula
  l->primeira = nova;
}
```

Lista encadeadas

- Inserir item na lista - inserção no final:



Lista encadeadas

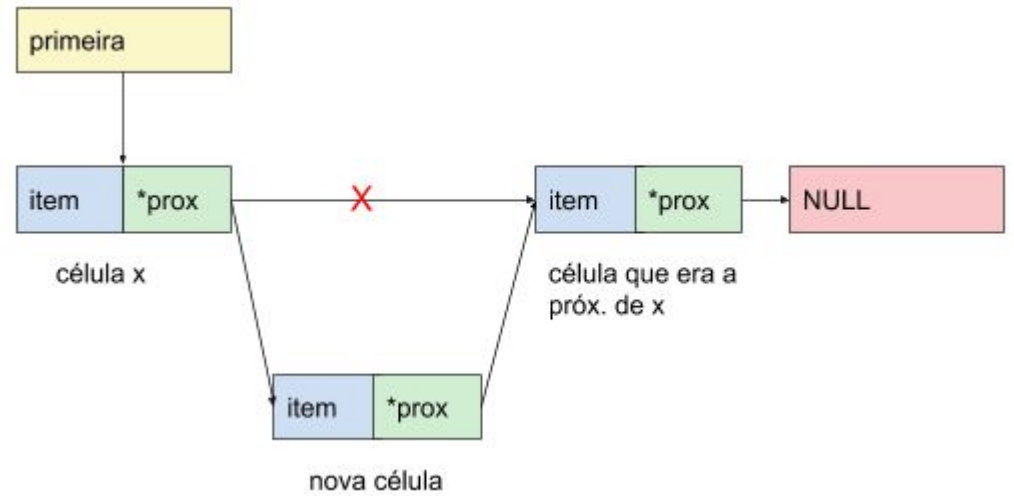
- Inserir item na lista - inserção no final:

```
void insere_fim_lista(Lista *l, int chave) {
  Item novo;
  novo.chave = chave;
  Celula *nova = malloc(sizeof(Celula));
  nova->item = novo;
  nova->prox = NULL; // como a nova célula será a última, sua prox. é NULL
  if(verifica_lista_vazia(l)) // se está vazia, quem vai apontar para a nova é a primeira
      l->primeira = nova;
  else { // se não está vazia, quem vai apontar para a nova é a que era a última
      Celula *ultima = l->primeira;
      while(ultima->prox != NULL) {
            ultima = ultima->prox;
      }
      ultima->prox = nova;
  }
}
```

Partindo da 1ª célula, percorrer até a última para inserir a nova célula após a última.

Lista encadeadas

- Inserir item na lista - inserção no meio:



Inserir uma nova célula após uma célula “x”.

Lista encadeadas

- Inserir item na lista - inserção no meio:

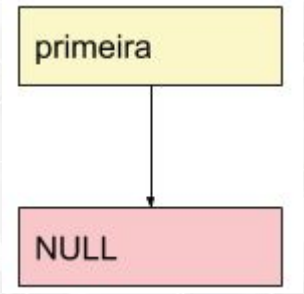
```
void insere_meio_lista(Lista *l, int chave, int x) {
  Item novo;
  novo.chave = chave;
  Celula *nova = malloc(sizeof(Celula));
  nova->item = novo;
  // acha a célula após a qual será feita a inserção
  Celula *aux = busca_por_chave(l, x);
  if(aux != NULL) {
      nova->prox = aux->prox;
      aux->prox = nova;
  }
  else {
      printf("O item informado não existe.\n");
  }
}
```

Lista encadeadas

- Remover item na lista
  - Necessário verificar se a lista não está vazia.
  - A remoção pode ser de 3 tipos:
    - No começo da lista;
    - No meio da lista (posição qualquer);
    - No fim da lista;

Lista encadeadas

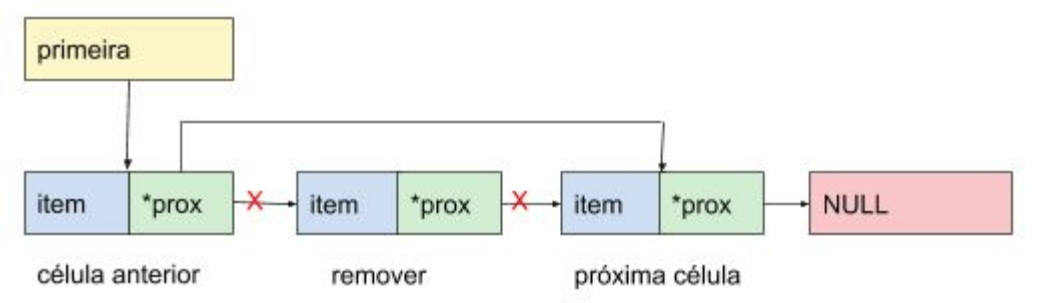
- Remover item na lista
  - Verificação de lista vazia:




```
int verifica_lista_vazia(Lista *l) {
  return l->primeira == NULL;
}
```

Lista encadeadas

- Remover item na lista - remoção do meio



Remover a célula após uma célula “x”.

Lista encadeadas

- Remover item na lista - remoção do meio:

```
void remove_item(Lista *l, int x) {
    int tamanho = tamanho_lista(l);
    Celula *anterior = busca_por_chave(l, x);
    if(verifica_lista_vazia(l) || anterior == NULL || anterior->prox == NULL) {
        printf("Erro: a lista está vazia ou o item não existe.\n");
        return;
    }
    Celula *remover = anterior->prox;
    Celula *proxima = remover->prox;
    anterior->prox = proxima;
    free(remover);
}
```

Lista encadeadas

- Buscar item da lista

```
Celula * busca_por_chave(Lista *l, int chave) {
  int achou = 0;
  Celula *aux = l->primeira;
  while(achou == 0 && aux != NULL) {
      if(aux->item.chave == chave)
         achou = 1;
      else
         aux = aux->prox;
  }
  return aux;
}
```

Lista encadeadas

- Imprimir a lista

```
void imprime(Lista *l) {
      Celula *aux;
      for(aux = l->primeira; aux != NULL; aux = aux->prox)
            printf("chave = %d\n", aux->item.chave);
}
```

Lista encadeadas

- Liberar a lista

```
void libera_lista(Lista *l) {
  Celula *aux = l->primeira;
  Celula *liberar;
  while(aux != NULL) {
      liberar = aux;
      aux = aux->prox;
      free(liberar);
  }
  free(l);
```

Lista encadeadas

- Vantagens da lista encadeada
  - Não é preciso definir tamanho máximo, as células são alocadas conforme a demanda.
  - É possível liberar uma única célula, não é necessário esperar para liberar a lista toda, como na implementação por arranjos/vetores;
  - Para remover ou adicionar uma célula, não é preciso deslocar as outras.

Lista encadeadas

- Desvantagens da lista encadeada
  - Não existe acesso por meio de índices, para encontrar uma célula é preciso percorrer a lista.
  - Usa mais memória pois, além do item, precisa guardar um ponteiro para a próxima célula.

Algoritmos recursivos

- O problema vai sendo dividido em problemas cada vez menores até chegar à menor instância possível do problema (caso base).
- O caso base é resolvido e a partir dele é possível ir resolvendo os demais casos para chegar a uma solução geral.

Algoritmos recursivos

- Por exemplo: calcular o fatorial de 5:
  - O problema pode ser quebrado em partes cada vez menores até encontrar a instância mais simples do problema, que é 1!
    - Resolvendo 1! é possível resolver 2!
    - Resolvendo 2! é possível resolver 3!
    - Resolvendo 3! é possível resolver 4!
    - Resolvendo 4! é possível resolver 5!

Algoritmos recursivos

- Função recursiva para o cálculo do fatorial de 5:

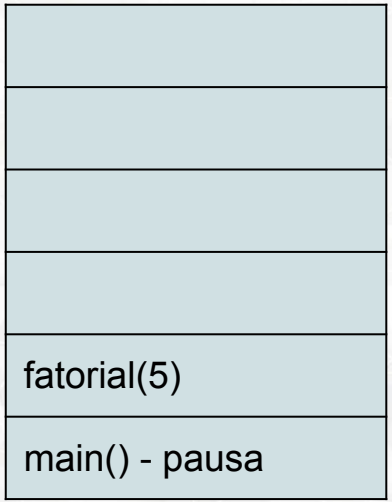
```
int fatorial_rec(int n) {
  if(n == 1) // menor instância do problema - já retorna o resultado
    return 1;
  else // reduz o problema - chama a função novamente para n - 1
    return n * fatorial_rec(n - 1);
}
```

Algoritmos recursivos

- Pilha de execução:

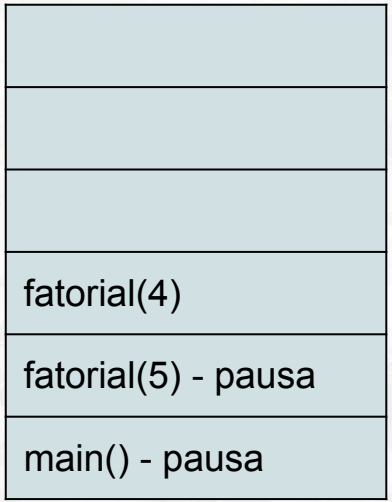
Algoritmos recursivos

- Pilha de execução:



Algoritmos recursivos

- Pilha de execução:



Algoritmos recursivos

- Pilha de execução:

fatorial(3)
fatorial(4) - pausa
fatorial(5) - pausa
main() - pausa

Algoritmos recursivos

- Pilha de execução:

fatorial(2)
fatorial(3) - pausa
fatorial(4) - pausa
fatorial(5) - pausa
main() - pausa

Algoritmos recursivos

- Pilha de execução:

fatorial(1)
fatorial(2) - pausa
fatorial(3) - pausa
fatorial(4) - pausa
fatorial(5) - pausa
main() - pausa

Algoritmos recursivos

- Pilha de execução:

1
fatorial(2) - pausa
fatorial(3) - pausa
fatorial(4) - pausa
fatorial(5) - pausa
main() - pausa

resultado pronto, retorna para fatorial(2)

Algoritmos recursivos

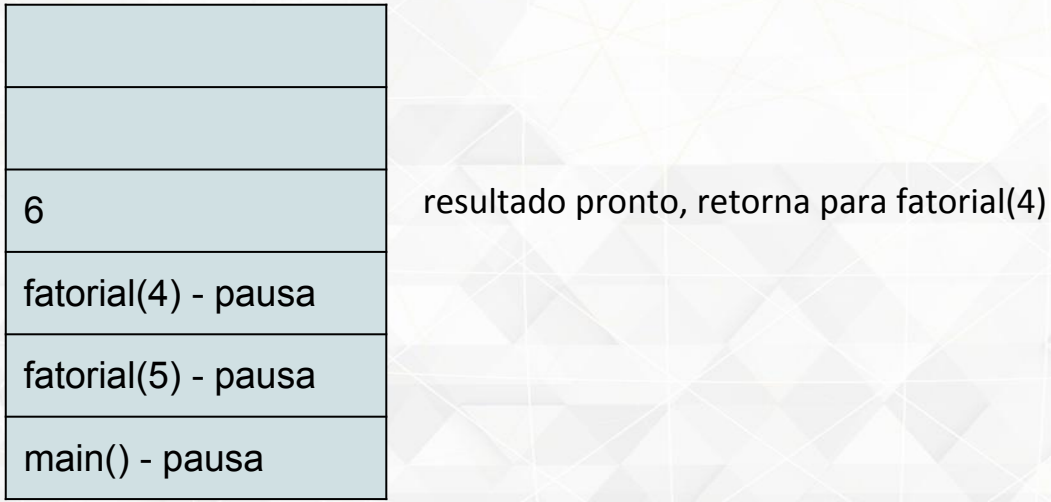
- Pilha de execução:

2
fatorial(3) - pausa
fatorial(4) - pausa
fatorial(5) - pausa
main() - pausa

resultado pronto, retorna para fatorial(3)

Algoritmos recursivos

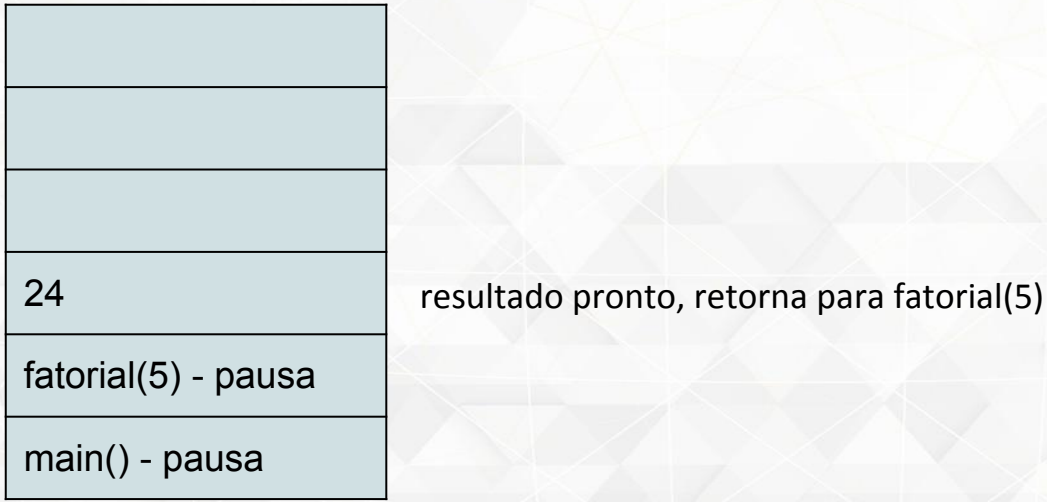
- Pilha de execução:



resultado pronto, retorna para fatorial(4)

Algoritmos recursivos

- Pilha de execução:



resultado pronto, retorna para fatorial(5)

Algoritmos recursivos

- Pilha de execução:

resultado pronto, retorna para main()

Algoritmos recursivos

- Pilha de execução:

Algoritmos recursivos

- Os resultados vão sendo agrupados, do mais simples (caso base) até o resultado final.

Algoritmos recursivos

- Montar a pilha de execução ajuda a entender como o computador irá tratar a função recursiva.
- O caso base é a “condição de parada” da função. Assim como em um laço, ela deve estar bem definida para que a função consiga terminar.
- O código da função recursiva geralmente tem:
  - Um if que trata o caso base (já retorna).
  - Após o if, a lógica que trata os demais casos (divide o problema).

Algoritmos recursivos

- Vantagens:
  - código mais limpo, elegante;
- Desvantagens:
  - geralmente tem maior consumo de memória;
  - por mais limpo que o código fique, às vezes não compensa em termos de eficiência (como no caso do Fibonacci recursivo, por exemplo);
  - pensar em uma solução recursiva é menos intuitivo do que pensar em uma solução sequencial.

Algoritmos de ordenação

- Ordenar - organizar os itens de um conjunto de acordo com uma determinada ordem: crescente ou decrescente, por exemplo.
- Facilita o acesso aos itens posteriormente.

Algoritmos de ordenação

- Tipos de algoritmos de ordenação:
  - Simples / elementares - consumo de tempo de $O(n^2)$.
    - Seleção;
    - Inserção;
    - Bolha.
  - Eficientes - consumo de tempo de $O(n \log n)$.
    - Merge;
    - Quick.

Algoritmos de ordenação

- **Seleção**: dado um vetor de tamanho N:
  - O menor item de v[0, …, N - 1] é selecionado e troca de lugar com o 1º item do vetor.
  - O menor item de v[1, …, N - 1] é selecionado e troca de lugar com o 2º item do vetor.
  - O menor item de v[2, …, N - 1] é selecionado e troca de lugar com o 3º item do vetor, e assim por diante até restar o último item do vetor.

Algoritmos de ordenação

- **Seleção**:

```
ORDENAÇÃO - SELEÇÃO:
para i ← 0 até N - 1
        min ← i
        para j ← i até N
                se vetor[j] < vetor[min], então
                        min ← j
        troca(vetor, min, i)
```

Algoritmos de ordenação

- **Seleção**:
  - Consumo de tempo: não reconhece se o vetor já está ordenado, então em qualquer caso seu custo será $O(n^2)$
  - Consumo de memória: O(1) em qualquer caso.

Algoritmos de ordenação

- **Inserção**: o vetor é percorrido em ordem. Cada item do vetor é selecionado e é movido para sua posição correta.

Algoritmos de ordenação

- **Inserção**:

```
ORDENAÇÃO - INSERÇÃO:
para i ← 1 até N
        atual ← vetor[i]
        j ← i - 1
        enquanto j >= 0 e vetor[j] > atual
                desloca_direita(vetor, j) // para liberar espaço para o atual
                j ← j - 1
        vetor[j + 1] = atual
```

Algoritmos de ordenação

- **Inserção**:
  - Consumo de tempo:
    - Vetor não ordenado: $O(n^2)$
    - Vetor em ordem crescente: O(n), pois se o vetor já estiver em ordem crescente não entra no laço interno.
  - Consumo de memória: O(1) em qualquer caso.

Algoritmos de ordenação

- **Bolha**: o vetor vai sendo percorrido em ordem, comparando pares de elementos. Sempre que o par de elementos estiver desordenado, os elementos trocam de lugar. Ao fim de cada iteração, o último item do vetor já está na posição certa.

Algoritmos de ordenação

- **Bolha**:

```
ORDENAÇÃO - BOLHA:
flag ← 0
para i ← N - 1 até 1
        para j ← 0 até i
                Se vetor[j] > vetor[j + 1], então
                        troca(vetor, j, j + 1) // vai empurrando o maior para o final
                        flag ← 1
        Se flag == 0, então
                termina
```

Algoritmos de ordenação

- **Bolha**:
  - Consumo de tempo:
    - Vetor não ordenado: $O(n^2)$
    - Vetor em ordem crescente: O(n), pois usa uma flag que indica se alguma troca foi necessária. De acordo com o estado da flag, o algoritmo finaliza.
  - Consumo de memória: O(1) em qualquer caso.

Algoritmos de ordenação

- **Merge**:
  - Divide o vetor de entrada em duas metades iguais (ou aprox. iguais) e assim sucessivamente, até chegar em subvetores de tamanho 1 (um vetor com apenas um item já está trivialmente ordenado).
  - Uma vez divididos, os subvetores são agrupados em ordem, usando ideia de **intercalação / fusão** de dois vetores ordenados.
  - Daí vem o nome merge sort: merge = fundir em inglês.

Algoritmos de ordenação

- **Merge**: etapa de intercalação:

Procedimento: MERGE

Entradas:

- *A:* um vetor
- *p*, *q*, *r*: índices dos subvetores *A*[*p*...*q*] e *A*[*q* + 1...*r*] (ambos já ordenados).

Resultado: O subvetor *A*[*p*...*r*] ordenado.

1. Iguale *n1* a *q* - *p* + 1 e *n2* a *r* - *q*.
2. Sejam *B*[0...*n1*] e *C*[0...*n2*] novos vetores.
3. Copie *A*[*p*...*q*] para *B*[0...*n1*] e *A*[*q* + 1...*r*] para *C*[0...*n2*].
4. Enquanto *i* for menor que *n1* e *j* for menor que *n2*:
   a. Se *B*[*i*] ≤ *C*[*j*], então:
      i. iguale *A*[*k*] a *B*[*i*] e incremente *i*.
   b. Caso contrário:
      i. iguale *A*[*k*] a *C*[*j*] e incremente *j*.
   c. Incremente *k*.

Algoritmos de ordenação

- **Merge**: algoritmo recursivo:

Procedimento: MERGE-SORT

Entradas:

- *A:* um vetor.
- *p, r:* índices iniciais e finais de um subvetor de *A*.

Resultado: Os elementos do subvetor A[*p*...*r*] ordenados em ordem crescente.

1. Se *p* < *r*, faça o seguinte:
   a. Iguale *q* a (*p* + *r*) / 2.
   b. Chame recursivamente MERGE-SORT(*A*, *p*, *q*).
   c. Chame recursivamente MERGE-SORT(*A*, *q* + 1, *r*).
   d. Chame MERGE(*A*, *p*, *q*, *r*).
2. Caso contrário, o subvetor A[*p*...*r*] tem só um elemento. Apenas retorne sem fazer nada.

Algoritmos de ordenação

- **Merge**:
  - Consumo de tempo: independente de o vetor estar ordenado ou não, sempre vai dividindo o vetor por 2, então o custo é o mesmo: O(n log n).
  - Consumo de memória: O(n), apesar de precisar ter até log n chamadas na pilha de execução, cada chamada consome memória extra para os vetores auxiliares da função intercala, totalizando custo O(n).

Algoritmos de ordenação

- **Quick**: também vai dividindo o problema em instâncias menores, mas a forma de divisão é diferente. Consiste em:
  - Escolher um pivô, que é um item qualquer do vetor.
  - A partir do pivô, o vetor é separado em duas partes:
    - Todos os itens menores que o pivô irão formar um subvetor;
    - Todos os itens maiores que o pivô irão formar outro subvetor.
  - Esse processo é feito repetidas vezes até encontrar subvetores com um único item.

Algoritmos de ordenação

- **Quick**:
  - Geralmente o pivô é o último item do vetor (método de Lomuto) e todos os itens maiores que o pivô vão sendo jogados para sua direita.

Algoritmos de ordenação

- **Quick** - etapa de separação

```
int separa(int v[], int primeiro, int ultimo) {
   int pivo = v[ultimo];
   int esquerda = primeiro;
   int atual;
   for(atual = primeiro; atual < ultimo; atual++) {
      if (v[atual] <= pivo) {
         troca(&v[atual], &v[esquerda]);
         esquerda++;
      }
   }
   troca(&v[esquerda], &v[ultimo]);
   return esquerda;
}
```

Algoritmos de ordenação

- **Quick** - algoritmo recursivo

```
void quicksort(int v[], int p, int u) {
   if(p < u) {
      int j = separa(v, p, u);
      quicksort(v, p, j - 1);
      quicksort(v, j + 1, u);
   }
   return;
}
```

Algoritmos de ordenação

- **Quick**
  - Consumo de tempo:
    - Vetor ordenado (crescente ou decrescente): caracteriza o pior caso, pois todos os itens ficam amontoados apenas de um lado do pivô. Terá custo $O(n^2)$.
    - Vetor balanceado: o vetor está de tal forma que a mesma quantidade de itens fica de cada lado do pivô, dividindo o problema sempre por 2, como no merge. Caracteriza o melhor caso. Gera custo O(n log n).
    - Vetor desbalanceado com divisão proporcionalmente constante de cada lado do pivô. Caracteriza o caso médio. Gera custo O(n log n).

Algoritmos de ordenação

- **Quick**
  - Consumo de memória:
    - Para melhor caso e caso médio, precisa ter até log n chamadas na pilha de execução, gerando custo O(log n).
    - Para o pior caso, precisa ter até n chamadas na pilha de execução, gerando custo O(n).

Algoritmos de ordenação

- **Comparativo** - consumo de tempo:

	SELEÇÃO	INSERÇÃO	BOLHA	MERGE	QUICK
Pior caso	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$
Caso médio	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Melhor caso	$O(n^2)$	$O(n)$	$O(n)$	$O(n \log n)$	$O(n \log n)$

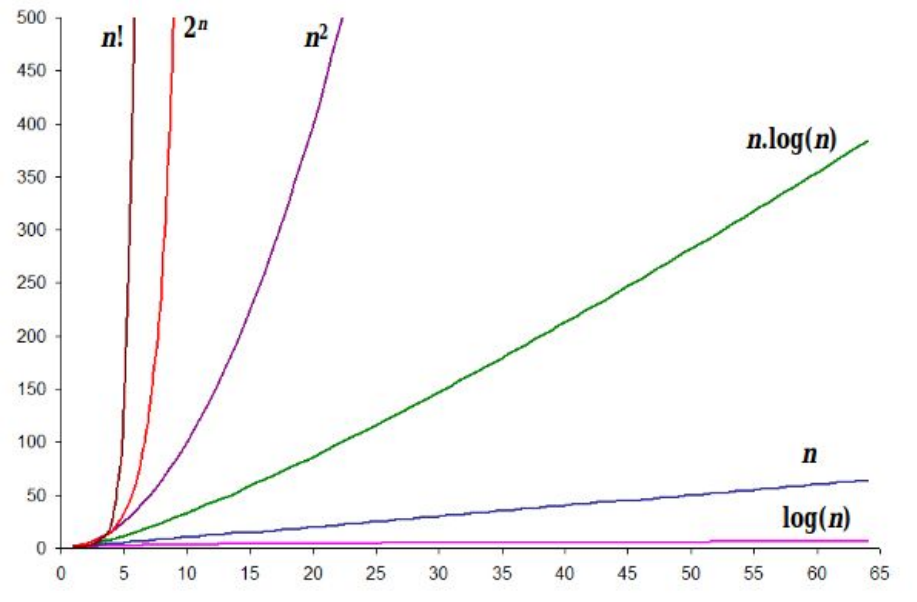
Algoritmos de ordenação

- **Comparativo** - consumo de memória:

	SELEÇÃO	INSERÇÃO	BOLHA	MERGE	QUICK
Pior caso	O(1)	O(1)	O(1)	O(n)	O(n)
Caso médio	O(1)	O(1)	O(1)	O(n)	O(log n)
Melhor caso	O(1)	O(1)	O(1)	O(n)	O(log n)

Algoritmos de ordenação

- Comparativo entre as funções:



Algoritmos de ordenação

- **Observações:**
  - Quick x Merge: “empatam” em consumo de tempo mas o quick é mais eficiente quanto ao uso da memória (o merge cria vetores auxiliares, o quick não), por isso, o quick é geralmente o preferido.
  - Para um vetor suficientemente pequeno, um algoritmo de ordenação simples pode fazer o trabalho mais rápido do que um algoritmo eficiente, pois devido ao código com poucas instruções, os custos constantes são menores.

Algoritmos de ordenação

- **Observações:**
  - Dependendo do caso, um algoritmo eficiente pode ser mais lento que um algoritmo simples. Ex.:
    - Merge para vetor ordenado = O(n log n)
    - Bolha para vetor ordenado = O(n)

Algoritmos de pesquisa

- Têm como objetivo buscar um determinado elemento dentro de um conjunto de dados (array, lista, árvore, etc).
- Principais algoritmos:
  - Pesquisa sequencial;
  - Pesquisa binária.

Algoritmos de pesquisa

- Pesquisa sequencial: percorre um arranjo sequencialmente a partir do primeiro registro até encontrar a chave buscada ou até chegar ao final do array.

```
int busca_sequencial(int v[], int n, int chave) {
  int i;
  for(i = 0; i < n; i++)
    if(v[i] == chave)
      return i;
  return -1;
}
```

Algoritmos de pesquisa

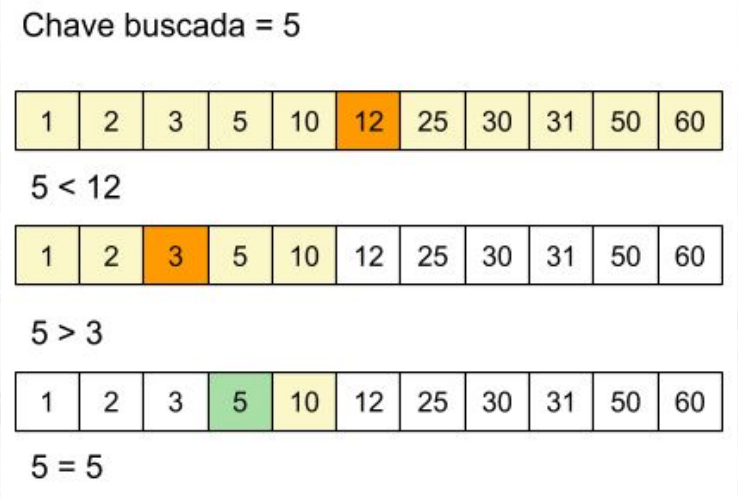
- Pesquisa sequencial:
  - Melhor caso: o item buscado está no primeiro índice - O(1)
  - Pior caso: o item buscado está no último índice ou não existe - O(n)

Algoritmos de pesquisa

- Pesquisa binária: aplica-se quando o array já está ordenado.
  - A chave é comparada com o item que está no meio do array.
  - Se a chave for igual ao item, a busca termina.
  - Se a chave for menor que o item, repete a busca com a metade esquerda do array.
  - Se a chave for maior que o item, repete a busca com a metade direita do array.

Algoritmos de pesquisa

- Pesquisa binária:



Algoritmos de pesquisa

- Pesquisa binária:

```
int busca_binaria(int v[], int n, int chave) {
  int p = 0, r = n - 1, q;
  while(p <= r) {
    q = (p + r) / 2;
    if(v[q] == chave)
      return q;
    else {
      if(v[q] > chave)
        r = q - 1;
      else
        p = q + 1;
    }
  }
  return -1;
}
```

Algoritmos de pesquisa

- Pesquisa binária:
  - Melhor caso: o item buscado está no meio do vetor: O(1)
  - Pior caso: o item buscado não está no vetor: O(log n) - sempre vai dividindo o problema por 2.