Estruturas de Dados II

Problemas e suas características

Prof. Bruno Azevedo

Instituto Federal de São Paulo

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus Catanduva

Sobre as Avaliações

- Como conversado na última aula, a nota final da disciplina será composta por três avaliações: Prova, Seminário e Exercícios.
- Os exercícios serão todos os exercícios passados nos slides até o fim do semestre, portanto, **façam** os exercícios apresentados nos slides.
- Os pesos das avaliações serão definidos futuramente.

Problemas

- Entenda a estrutura do problema para que possa encontrar uma solução eficiente.
- Por exemplo, suponha que queremos encontrar um número em uma lista de números (vetor).
- Qual é a característica dessa lista? Lista de números em posicoes arbitrárias? Está ordenada?
- Vamos assumir o primeiro caso. Se ela não está ordenada, faremos uma busca linear e temos um tempo de $O(n)$.
- "Professor, mas nao seria melhor ordenar antes?" Não. Se formos ordenar, temos já teremos um tempo de $O(n \log_2 n)$, **que é pior que apenas buscar linearmente**.

Problemas

- Mas e se a lista está ordenada? Então temos um problema diferente.
- Para esta característica particular, temos uma solucao melhor. Um **algoritmo** melhor.
- Vamos conhecer o algoritmo de **Busca Binária**.

- A Busca Binária possui tempo de execução de $O(log_2 n)$ (por quê?).

Problemas

- Ou seja, a solucao proposta deve vir do **entendimento das características do problema**.
- Quanto melhor entendermos as propriedades e a estrutura do nosso problema, mais capazes seremos de resolvê-lo de forma eficiente. Tenham isso em mente.
- E muitos problemas já foram estudados, portanto, é de seu interesse conhecer o maior número possível de problemas e as soluções já propostas.
- Não apenas para evitarem a "reinvenção da roda".
- Ao conhecerem soluções para problemas já bem estudados, irão encontrar ideias interessantes que podem ser aplicadas futuramente na resolução de outros desafios.

Exercícios (1)

- (1) Implemente em C/C++ o algoritmo da Busca Binária de acordo com a explicação anterior.
- O algoritmo tradicional verifica se o elemento central é igual ao elemento buscado.
- Mas teremos um algoritmo mais eficiente (não no sentido assintótico) se evitarmos essa comparação até o fim.
- (2) Implemente em C/C++ o algoritmo da Busca Binária realizando essa verificação apenas quando sobrar um único elemento.
- (3) Demonstre que a Busca Binária executa em $O(log_2 n)$.

Solução do Exercício (3)

- Começamos com n elementos e, após cada iteração, a lista de números possui metade do tamanho da iteração anterior.
  - Primeira iteração: lista de n elementos.
  - Segunda iteração: lista de $\frac{n}{2}$ elementos.
  - Terceira iteração: lista de $\frac{n}{4}$ elementos.
- E assim por diante, até termos apenas um elemento.

Solução do Exercício (3)

- Portanto, o número de iterações necessárias para reduzir o tamanho da lista a um elemento é **o número de vezes que podemos dividir a lista por dois**.
- Supondo uma lista de n elementos, isso pode ser descrito pela equação:

$$\frac{n}{2^k} = 1$$

- Onde k é o número de iterações, que é o que queremos encontrar.

Solução do Exercício (3)

- Desejamos encontrar k, então vamos isolar este termo.
- Multiplicamos ambos os lados por 2^k.

$$\frac{n}{2^k} \cdot 2^k = 1 \cdot 2^k \quad (1)$$

$$n = 2^k \quad (2)$$

- Aplicamos logaritmo base 2 em ambos os lados (para quem não lembra, $log_a(b) = c \Leftrightarrow a^c = b$):

$$\log_2(n) = k$$

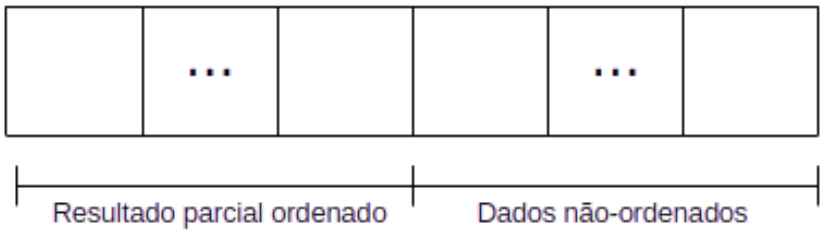
- Ou seja, se temos n elementos na lista, o número de iterações (número de comparações) no pior caso, é $k = \log_2(n)$.

Problema de Ordenação

- No "mundo real", quando queremos ordenar números geralmente é porque eles são chaves associadas a outros dados.
- Suponha que você precise ordenar uma sequência de números em ordem estritamente crescente.
- Este problema pode ser formalizado como um problema computacional da seguinte forma:
- Entrada: Uma sequência de n números $\langle a_1, a_2, \ldots, a_n \rangle$.
- Saída: Uma permutação $\langle a'_1, a'_2, \ldots, a'_n \rangle$ da sequência de entrada de forma que $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Um Primeiro Algoritmo: Insertion Sort

- O algoritmo Insertion Sort é um método simples de ordenação que constrói a solução final de forma incremental.
- Intuitivo e eficiente no sentido apresentado na primeira aula (alguém lembra?).
- Funcionamento:
  1. Iteração Inicial: Começa com o segundo elemento do vetor, considerando que o primeiro elemento está ordenado, dado que está sozinho.
  2. Comparação e Inserção: Compara o elemento atual com os elementos anteriores, deslocando-os para a direita, até encontrar a posição correta onde o elemento deve ser inserido.
  3. Repetição: Repete o processo para todos os elementos do vetor.
- Podemos visualizar dois subvetores durante a execução do algoritmo: um ordenado (mais à esquerda) e um não-ordenado (mais à direita).



Um Primeiro Algoritmo: Insertion Sort

```
void insertionSort(int V[], int n) {
    int j, chave;
    for(int i = 1; i < n; i++) {
        chave = V[i];
        j = i - 1;
        while(j >= 0 && V[j] > chave) {
            V[j + 1] = V[j];
            j = j - 1;
        }
        V[j + 1] = chave;
    }
}
```

Um Primeiro Algoritmo: Insertion Sort

- ```
  void insertionSort(int V[], int n) {
      int j, chave;
      for(int i = 1; i < n; i++) {
          chave = V[i];
          j = i - 1;
          while(j >= 0 && V[j] > chave) {
              V[j + 1] = V[j];
              j = j - 1;
          }
          V[j + 1] = chave;
      }
  }
  ```
- Exemplo: $\{6, 5, 3, 1, 8, 7, 2, 4\}$.


  ```

Exercícios (2)

- Considerando as comparações:
- (1) Determine a sua complexidade de tempo no pior caso utilizando a notação-O.
- (1.1) Qual seria a propriedade da instância onde teríamos o pior caso?
- (2) Determine a sua complexidade de tempo no **melhor** caso utilizando a notação-O.
- (2.1) Qual seria a propriedade da instância onde teríamos o melhor caso?

Analisando o Pior Caso

```
void insertionSort(int V[], int n) {
    int j, chave;
    for(int i = 1; i < n; i++) {
        chave = V[i];
        j = i - 1;
        while(j >= 0 && V[j] > chave) {
            V[j + 1] = V[j];
            j = j - 1;
        }
        V[j + 1] = chave;
    }
}
```

- O primeiro laço executa n - 1 vezes.
- O segundo laço executa um número variável de vezes, dependendo da instância.
- Mas no **pior** caso, teríamos que inserir o número na primeira posição do vetor, ou seja, teremos que percorrer **i** vezes.
- Se percorremos *i* vezes para cada iteração do laço externo, teremos então a soma dos primeiros $n - 1$ números inteiros.

Analisando o Pior Caso

- $S = 1 + 2 + 3 + \ldots + (n-1) = \frac{(n-1)n}{2} = \frac{1}{2} \cdot n^2 - \frac{1}{2} \cdot n$
- Para utilizar a notação-O, iremos ignorar constantes multiplicativas e aditivas e focar no termo de maior ordem.
- Portanto, a complexidade computacional no pior caso do algoritmo Insertion Sort é $O(n^2)$.

Analisando o Pior Caso

- Qual a característica desta instância? É uma instância em ordem decrescente.
- Deste modo, cada vez o laço interno executará i vezes.

Analisando o Melhor Caso

```
void insertionSort(int V[], int n) {
    int j, chave;
    for(int i = 1; i < n; i++) {
        chave = V[i];
        j = i - 1;
        while(j >= 0 && V[j] > chave) {
            V[j + 1] = V[j];
            j = j - 1;
        }
        V[j + 1] = chave;
    }
}
```

- O primeiro laço executará n - 1 vezes.
- O laço interno nunca executará, com a condição V[j] $>$ chave sendo falsa em todas as vezes.
- Portanto, teremos n - 1 comparações, ou seja, a complexidade computacional no melhor caso do algoritmo Insertion Sort é $O(n)$

Analisando o Melhor Caso

- Qual a característica desta instância? É uma instância já ordenada em ordem crescente.
- Deste modo, a segunda condição do laço for nunca será verdadeira.

Divisão e Conquista

- Existem diversas técnicas para o design de algoritmos. O Insertion Sort utiliza uma abordagem incremental.
- Vamos conhecer outro método: **Divisão e Conquista**.
- Muitos algoritmos possuem uma estrutura recursiva: para resolver um problema, eles se chamam a si mesmos uma ou mais vezes para lidar com subproblemas relacionados.
- Esses algoritmos (geralmente) seguem o método de Divisão e Conquista.
- Dividem o problema em vários subproblemas que são semelhantes ao problema original, mas menores em tamanho, resolvem os subproblemas recursivamente.
- Em seguida, combinam essas soluções para criar a solução do problema original.

Divisão e Conquista

- Vocês devem lembrar das abordagens recursivas para encontrar o enésimo número da sequência de fibonacci e para calcular o fatorial de um número.
- Esses algoritmos utilizavam a estratégia de Divisão e Conquista.
- A ideia geral é:
  1. Divida o problema em um ou mais subproblemas que são instâncias menores do mesmo problema.
  2. Conquiste os subproblemas resolvendo-os recursivamente.
  3. Combine as soluções dos subproblemas para formar uma solução para o problema original.

Merge Sort

- O algoritmo para ordenação **Merge Sort** utiliza a estratégia de Divisão e Conquista.
- Em cada passo, ele ordena um subvetor, começando com o vetor inteiro e recursivamente dividindo em subvetores menores e menores.
  1. Divida o subvetor $V[p \ldots r]$ a ser ordenado em dois subvetores adjacentes, cada um com metade do tamanho. Para fazer isso, calcule o ponto médio q de $V[p \ldots r]$ (tirando a média de p e r), e divida $V[p \ldots r]$ em subvetores $V[p \ldots q]$ e $V[q+1 \ldots r]$.
  2. Conquiste, ordenando cada um dos dois subvetores $V[p \ldots q]$ e $V[q+1 \ldots r]$ recursivamente usando o Merge Sort.
  3. Combine, mesclando os dois subvetores ordenados $V[p \ldots q]$ e $V[q+1 \ldots r]$ de volta em $V[p \ldots r]$, produzindo a resposta ordenada.
- Pode ser implementado de diversas formas diferentes, e encontrarão diferentes implementações dependendo da fonte utilizada.
- Veremos uma implementação simples à seguir.

Merge Sort

- Nosso algoritmo conterá duas funções: MergeSort e Merge.
- MergeSort: Divide recursivamente o vetor em subvetores menores e chama a função Merge para mesclá-los.
- Merge: Mescla dois subvetores ordenados em um único vetor ordenado. Ou seja, efetua as comparações e a mesclagem.

Merge Sort

```
void MergeSort(int V[], int p, int r) {
    if(p < r) {                  // Se há mais de um elemento
        int q = (p + r) / 2;     // Calcula o ponto médio
        MergeSort(V, p, q);      // Ordena recursivamente o primeiro subvetor
        MergeSort(V, q + 1, r); // Ordena recursivamente o segundo subvetor
        Merge(V, p, q, r);       // Combina os subvetores ordenados
    }
}
```

Merge Sort

```
void Merge(int V[], int p, int q, int r) {
    int tamEsq = q - p + 1;          // Comprimento de V[p ... q]
    int tamDir = r - q;              // Comprimento de V[q + 1 ... r]
    int E[tamEsq], D[tamDir];        // Cria novos vetores E[] e D[]
    for(int i = 0; i < tamEsq; i++) // Copia dados para os vetores temporários E[] e D[] */
        E[i] = V[p + i];
    for(int j = 0; j < tamDir; j++) // Copia V[q + 1 ... r] para D[0 ... tamDir - 1]
        D[j] = V[q + 1 + j];
    int i = 0, j = 0;                // Índices iniciais de L e R
    int k = p;                       // Índice inicial do subvetor mesclado
    while(i < tamEsq && j < tamDir) {   // Enquanto L e R tiverem elementos não mesclados
        if(E[i] <= D[j]) {
            V[k] = E[i];
            i++;
        } else {
            V[k] = D[j];
            j++;
        }
        k++;
    }
    while(i < tamEsq) {              // Copia o restante de L, se houver
        V[k] = E[i];
        i++;
        k++;
    }
    while(j < tamDir) {              // Copia o restante de R, se houver
        V[k] = D[j];
        j++;
        k++;
    }
}
```

Merge Sort

- Um exemplo visual para facilitar a compreensão do algoritmo Merge Sort:

Merge Sort

- Para analisar algoritmos recursivos de divisão e conquista, são necessárias ferramentas matemáticas.
- Uma recorrência é uma equação que descreve uma função em termos de seu valor em outros argumentos.
- Recorrências fornecem uma maneira natural de caracterizar os tempos de execução de algoritmos recursivos.
- Entretanto, é necessário um conhecimento matemático sólido para resolver recorrências.

Merge Sort

- Existem diferentes métodos que podemos utilizar para resolver recorrências, tais como:
  - Método da substituição: Você supõe um limitante e prova tal limitante utilizando indução.
  - Árvore de recursão: Você determina os custos em cada nível e os soma. Pode ser utilizado para ajudar na descoberta de um limitante para o método da substituição.
  - Método Mestre: Pode ser utilizado para resolver um tipo de recorrência e é necessário memorizar três casos específicos para ser aplicado.
- Mas não entraremos neste detalhe na disciplina, focaremos em analisar algoritmos que não utilizem recorrências.
- Caso tenham interesse em aprender tal análise, a maioria dos livros de análise de algoritmos possuem essas e outras técnicas para a resolução de recorrências.

Merge Sort

- O Merge Sort possui a mesma complexidade de tempo assintótica para o pior caso e o melhor caso.
- Afinal, mesmo que o vetor já esteja ordenado, o Merge Sort executa os passos de divisão e combinação.
- A complexidade é de $O(n \log_2 n)$ para ambos os casos.
- Sua principal vantagem é a previsibilidade do tempo de execução, o que é especialmente útil em situações onde a consistência do desempenho é crucial.
- No entanto, ele requer espaço adicional para armazenar os subvetores temporários durante o processo de mesclagem.

Merge Sort

- Ele pode ser implementado utilizando os benefícios do paralelismo, dividindo a tarefa de ordenação em várias threads ou processos que trabalham simultaneamente.
- Isso pode acelerar significativamente o tempo de execução em sistemas com múltiplos núcleos ou processadores.
- Entretanto, não mudará sua complexidade de tempo de execução.
- Afinal, a paralelização não altera o número total de operações necessárias para ordenar os dados, apenas reduz o tempo necessário para concluir essas operações distribuindo o trabalho entre vários processadores.

Exercícios (3)

- (1) A animação representa perfeitamente o algoritmo apresentado do Merge Sort? Justifique sua resposta.
- (2) Explique **em muito detalhe** o funcionamento da implementação fornecida do Merge Sort.
- (3) O Bubble Sort é um algoritmo simples de ordenação que percorre uma lista, comparando pares de elementos adjacentes e trocando-os se estiverem na ordem errada. O processo é repetido até que a lista esteja ordenada. Implemente o algoritmo Bubble Sort. Mais detalhes do algoritmo no último slide.
- (4) Determine a complexidade de tempo do Bubble Sort no pior caso e no melhor caso utilizando a notação-O.
- (5) Qual o melhor caso para o Bubble Sort? Por quê? Qual o pior caso? Por quê?
- (6) Por que o Bubble Sort geralmente é considerado menos eficiente que o Insertion Sort?
- (7) Em quais situações o Merge Sort é preferível ao Insertion Sort e ao Bubble Sort?

Exercícios (4)

- (8) Considere as listas de números abaixo. Para cada lista, aplique o Insertion Sort, Bubble Sort e Merge Sort e conte o número de comparações realizadas por cada algoritmo.

```
1. [5, 2, 9, 1, 5, 6]
2. [1, 2, 3, 4, 5, 6] (Lista já ordenada)
3. [6, 5, 4, 3, 2, 1] (Lista em ordem inversa)
```

  - Qual dos três algoritmos realizou o maior número de comparações em cada caso?
  - Explique por que alguns algoritmos realizaram menos comparações em vetores já ordenados ou quase ordenadas.
- (9) Considere um algoritmo que precisa ordenar uma lista onde os primeiros 50% dos elementos já estão ordenados, mas os últimos 50% estão em ordem inversa. Qual dos três algoritmos (Insertion Sort, Bubble Sort ou Merge Sort) você escolheria para esse caso? Justifique sua escolha baseando-se no comportamento de cada algoritmo.
- (10) Implemente os três algoritmos (Insertion Sort, Merge Sort e Bubble Sort) e modifique o código para contar o número total de trocas e comparações realizadas durante a execução. Execute-os em vetores de diferentes tamanhos e ordens (aleatória, já ordenada, inversa, etc.).
- (10.1) Como esses números variam entre os algoritmos e diferentes tipos de listas?

Bubble Sort

- Bubble Sort é um algoritmo de ordenação que funciona comparando repetidamente pares adjacentes de elementos em uma lista e trocando-os de posição se estiverem na ordem errada.
  1. Comece no início da lista.
  2. Compare o primeiro elemento com o segundo. Se o primeiro for maior que o segundo, troque-os de lugar. Se não, mantenha-os como estão.
  3. Mova-se para o próximo par de elementos (o segundo e o terceiro) e repita o processo.
  4. Continue comparando e trocando os elementos adjacentes até chegar ao final da lista. Ao final dessa primeira iteração, o maior elemento estará no final da lista, na posição correta.
  5. Repita o processo para o restante da lista (ignorando o último elemento já ordenado). A cada iteração, um elemento a menos precisa ser comparado, pois os maiores elementos já foram posicionados no final da lista.
  6. Continue repetindo o processo até que a lista esteja completamente ordenada (quando nenhuma troca for necessária).