Notas de aula: Análise de Algoritmos
Centro de Matemática, Computação e Cognição
Universidade Federal do ABC
Profa. Carla Negri Lintzmayer

Divisão e conquista e recorrências
(*Última atualização: 29 de março de 2019*)

Referências e materiais complementares desse tópico

Livro Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C.. Introduction to Algorithms. 2nd ed. MIT Press. 2002.
Capítulo 2.3, 4 (intro).

Livro Dasgupta, S.; Papadimitriou, C.; Vazirani, U.. **Algorithms**. Boston: McGraw-Hill. 2008.
Capítulo 2.3.

Vídeos Vídeo aulas do prof. Tim Roughgarden, de Stanford, em inglês (com legendas)
Aulas **1.5**, **1.6**, **1.7** e **4.1**.

Sumário

1 Divisão e conquista

- É uma técnica de projeto de algoritmos que faz uso de recursão.
- Consiste de 3 partes principais (em cada chamada recursiva):
  - Dividir o problema em subproblemas menores do mesmo tipo.
  - Conquistar os subproblemas por meio de recursão.
  - Combinar as soluções dos subproblemas em uma única para o problema original.
- Os dois algoritmos recursivos que vimos para resolver o problema da multiplicação de dois inteiros seguem o paradigma de divisão e conquista.

2 Voltando ao problema de ordenação

- Vimos um algoritmo de ordenação por inserção e agora veremos um por intercalação.

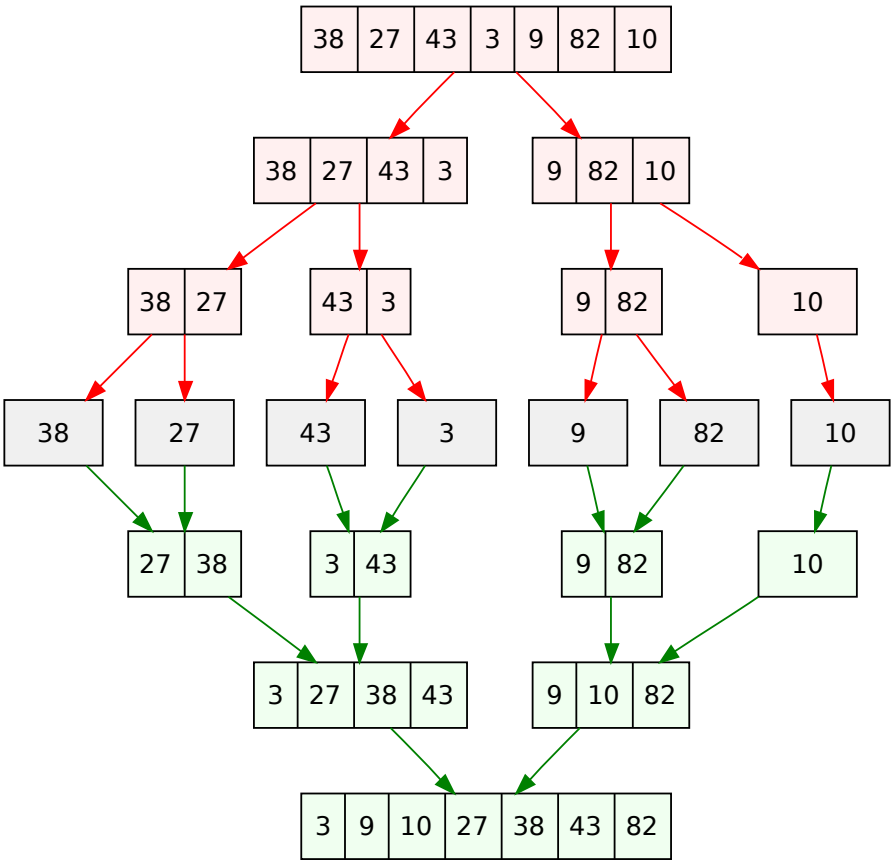
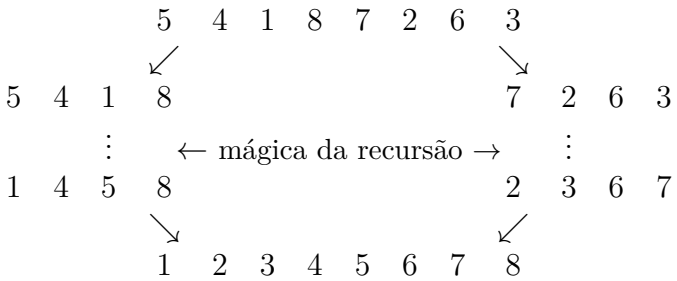


Figura 1: Exemplo de execução completa do MergeSort. Fonte: https://en.wikipedia.org/wiki/Merge_sort.

- A ideia do MergeSort (Algoritmo 1) é dividir o vetor em duas partes com metade do tamanho da entrada inicial, ordenar cada metade recursivamente e combinar os resultados com o procedimento Merge (Algoritmo 2).



- Veja uma **animação** desse algoritmo.

Algoritmo 1 Algoritmo de intercalação para o problema de ordenação.

```
1: função MergeSort(A, ini, fim)
2:     se ini < fim então
3:         meio ← ⌊(ini + fim)/2⌋
4:         MergeSort(A, ini, meio)
5:         MergeSort(A, meio + 1, fim)
6:         Merge(A, ini, meio, fim)
```

- A operação chave do MergeSort é a intercalação dos dois subvetores já ordenados em um único.
- Como implementar o Merge? Isto é, dados dois vetores B e C de tamanho $n/2$ já ordenados, como combiná-los de modo a gerar um único vetor ordenado A de tamanho n, contendo todos os elementos dos dois?
- Seja $B = (1, 4, 5, 8)$ e $C = (2, 3, 6, 7)$. Qual elemento vai ficar em $A[1]$? E em $A[2]$?
  - Em $A[1]$ só podemos ter $B[1]$ ou $C[1]$, o que for menor dentre os dois. Se $A[1] = B[1]$, então $A[2]$ só pode ter $B[2]$ ou $C[1]$, o que for menor dentre esses dois. Mas se $A[1] = C[1]$, então $A[2]$ só pode ter $B[1]$ ou $C[2]$, o que for menor dentre esses dois.
  - Note que uma vez que um elemento $B[i]$ é copiado para A, esse elemento não deve mais ser considerado. Da mesma forma, uma vez que um elemento $C[j]$ é copiado para A, ele não deve mais ser considerado.
- Precisamos manter um índice i para percorrer o vetor B, um índice j para percorrer o vetor C e um índice k para o vetor A. A cada iteração, precisamos colocar um elemento em $A[k]$: se $B[i] < C[j]$, então $A[k] \leftarrow B[i]$ e i deve ser incrementado para que $B[i]$ não seja considerado novamente (j não deve ser incrementado); mas se $C[j] \leq B[i]$, então $A[k] \leftarrow C[j]$ e j deve ser incrementado para que $C[j]$ não seja considerado novamente.
- Note que quando colocamos um elemento na posição k de A, sabemos que $A[1..k-1]$ está preenchido com os elementos de $B[1..i-1]$ e $C[1..j-1]$ e já está ordenado. Como $B[i]$ e $C[j]$ são menores do que todos os elementos de $B[i+1..n/2]$ e $C[j+1..n/2]$ e são maiores do que todos os elementos de $A[1..k-1]$, então o menor dos dois é de fato o elemento que deve ser colocado em $A[k]$.

Algoritmo 2 Algoritmo auxiliar para o MergeSort.

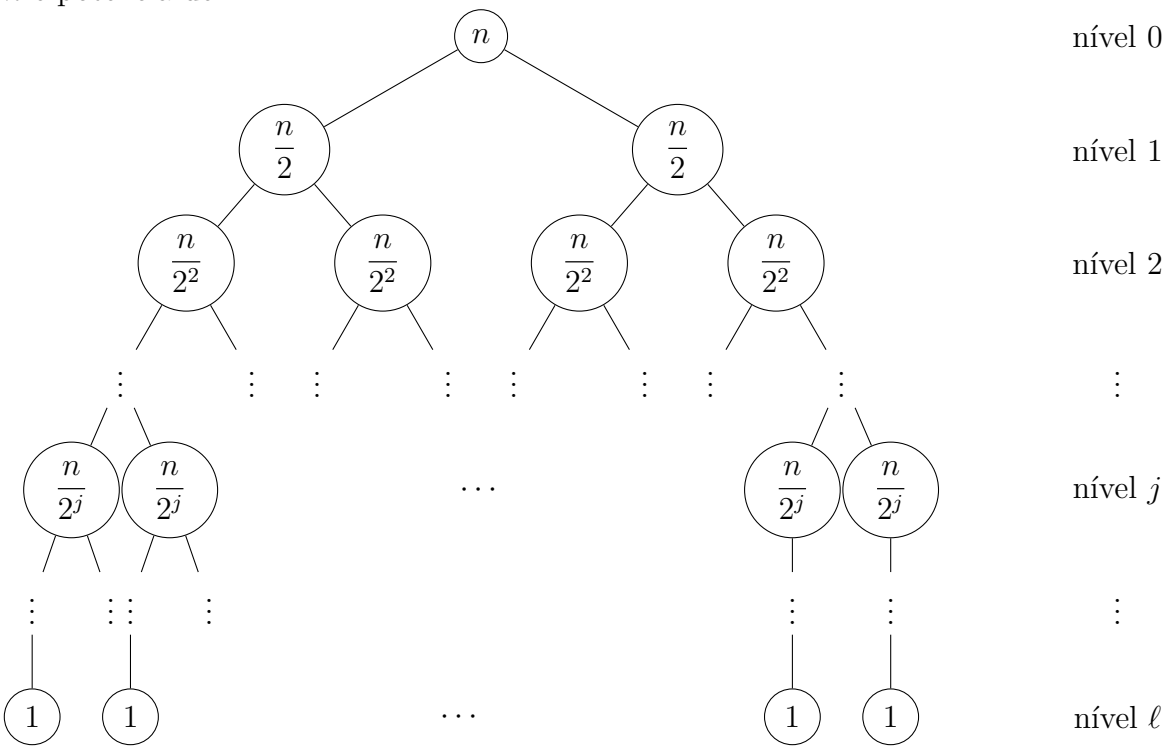
```
1:  função MERGE(A, ini, meio, fim)
2:      Seja B uma cópia de A[ini..meio] e C uma cópia de A[meio + 1..fim]
3:      i ← 1, j ← 1, k ← ini
4:      enquanto i < meio − ini + 1 e j < fim − meio faça
5:          se B[i] ≤ C[j] então
6:              A[k] ← B[i], i ← i + 1
7:          senão
8:              A[k] ← C[j], j ← j + 1
9:          k ← k + 1
10:     enquanto i < meio − ini + 1 faça
11:         A[k] ← B[i], i ← i + 1, k ← k + 1
12:     enquanto j < fim − meio faça
13:         A[k] ← C[j], j ← j + 1, k ← k + 1
```

- O MergeSort está correto?
  - Exercício: Mostre que o Merge está correto por meio de uma invariante de laço. Considere a notação do Algoritmo 2.
  - Por indução no tamanho n do vetor de entrada, vamos mostrar que o MergeSort está correto. Considere a notação do Algoritmo 1 (temos então que $n = fim - ini + 1$). Caso base: Quando o vetor tem tamanho 1 ou menos (caso em que $ini \geq fim$)

ele já está ordenado e o algoritmo de fato não faz nada. HIPÓTESE DE INDUÇÃO: Suponha que MERGESORT corretamente ordena qualquer vetor de tamanho menor do que n.

PASSO: Seja $A = [ini..fim]$ um vetor de tamanho $n > 1$. O Algoritmo 1 faz duas chamadas recursivas a vetores de tamanho claramente menor do que n. Por hipótese, esses dois vetores ($A[ini..meio]$ e $A[meio+1..fim]$) são ordenados corretamente. Pelo exercício anterior, a chamada à MERGE intercala corretamente esses dois vetores, deixando $A[ini..fim]$ totalmente ordenado.

- Quando tempo o MERGESORT leva?
  - Esse é um algoritmo recursivo. Como saber o número de passos básicos em todas as chamadas?
  - Antes, uma pergunta mais simples que conseguimos responder: quanto tempo o MERGE leva?
    - Se o vetor de entrada tem m números, então ele claramente leva tempo $\Theta(m)$.
  - Qual o comportamento do MERGESORT?
    - Quando ele faz uma chamada recursiva, é sobre uma entrada bem menor (metade do tamanho original).
    - Vamos ver a **árvore de recursão** desse algoritmo. Para facilitar, suponha que n é potência de 2[1]:



    - Note que o último nível, ℓ, tem problemas de tamanho 1 e nós começamos com um problema de tamanho n, com cada um sendo subdividido em 2 partes. Portanto, a árvore tem $\log n + 1$ níveis ao todo ($\ell = \log n$).
    - Para saber o tempo total do algoritmo, a ideia é contar o tempo gasto em cada nível (fora das chamadas recursivas).

[1] É possível mostrar que isso não modifica assintoticamente a análise para qualquer n.

* Note que o único trabalho feito fora da recursão é uma chamada ao MERGE. Assim, o tempo gasto em um nível j qualquer é a quantidade de problemas que ele tem (2^j) vezes o tempo gasto pelo MERGE para resolver cada problema ($\Theta(n/2^j)$, pois é proporcional ao tamanho do problema), ou seja:

$$= 2^j c \frac{n}{2^j} = cn \ ,$$

independente de j!

* O tempo total do algoritmo é, portanto, a soma do tempo gasto em cada nível:

$$= \sum_{j=0}^{\log n} cn = cn(\log n + 1) = cn \log n + cn = \Theta(n \log n) \ .$$

– O MERGESORT é melhor do que **o** INSERTIONSORT?

* Depende.
* Podemos dizer que o MERGESORT leva tempo $\Theta(n \log n)$, mas só podemos dizer que o INSERTIONSORT leva tempo $\Theta(n^2)$ *no pior caso*. O tempo do INSERTIONSORT é $O(n^2)$ mas é $\Omega(n)$.
* Como $n \log n \leq n^2$, então no pior caso sim, o MERGESORT é melhor.

– Dá para fazer melhor do que o MERGESORT?

* Se o algoritmo se baseia em comparações dos itens do vetor para fazer a ordenação, então não é possível conseguir tempo melhor do que $\Theta(n \log n)$.

  Teorema 1. *Todo algoritmo baseado em comparação tem tempo de execução no pior caso de* $\Omega(n \log n)$.

– Veja uma **animação** comparando vários algoritmos de ordenação.

3 Recorrências

- Uma recorrência é uma equação/inequação que descreve uma função em termos dela mesma sobre entradas menores.
- Muito comuns para descrever tempo de algoritmos recursivos.
- Portanto, tem duas partes:
  - Tempo quando não há recursão (caso base)
  - Tempo quando tem recursão = tempo das chamadas recursivas + tempo na chamada atual.
- MergeSort: $T(n) = 2T(\frac{n}{2}) + \Theta(n)$
- Multiplicação com recursão simples: $T(n) = 4T(\frac{n}{2}) + \Theta(n)$
- Multiplicação com Karatsuba: $T(n) = 3T(\frac{n}{2}) + \Theta(n)$
- Fibonacci recursivo: $T(n) = T(n-1) + T(n-2) + \Theta(1)$

- EXERCÍCIO: O problema da busca é descrito a seguir: dados um vetor A com n elementos e um inteiro x, devolva i caso exista algum i tal que $A[i] = x$ e devolva -1 caso contrário. Note que se A estiver ordenado, podemos comparar o elemento armazenado na posição média ($A[n/2]$) com x e eliminar metade do vetor do espaço de busca: se $A[n/2] = x$, então a busca está encerrada; caso contrário, se $x < A[n/2]$, então temos a certeza de que, se x estiver em A, então x está na primeira metade de A, i.e., x está em $A[1..n/2-1]$ (isso segue do fato de A estar ordenado); caso $x > A[n/2]$, então sabemos que, se x estiver em A, então x está no vetor $A[n/2+1..n]$. A *busca binária* é um algoritmo que repete essa ideia a cada passo, sempre dividindo ao meio o vetor restante. Escreva um pseudocódigo recursivo da busca binária e uma recorrência para seu tempo de execução. Prove que seu algoritmo está correto por indução no tamanho do vetor.

4 Resolvendo recorrências

- Encontrar uma "fórmula fechada" para a recorrência: uma expressão que não depende da própria função.
- Vimos que o MergeSort tem tempo $\Theta(n \log n)$
  - Será que $T(n) = 2T(\frac{n}{2}) + \Theta(n)$ tem solução $T(n) = \Theta(n \log n)$?
- Existem basicamente quatro métodos de resolução de recorrências:
  - Iteração
  - Substituição
  - Árvore de recursão
  - Teorema Mestre