Introdução

Pedro Henrique Del Bianco Hokama
6 de Agosto de 2019

Referências:

- Notas de aulas fortemente baseadas no curso: Stanford Algorithms by Tim Roughgarden
  - `https://www.youtube.com/playlist?list=PLEAYkSg4uSQ37A6_NrUnTHEKp6EkAxTMa`
  - Vídeos: 1.1 até 1.8
- CLRS: Cap 1, 2
- REPORT TO THE PRESIDENT AND CONGRESS DESIGNING A DIGITAL FUTURE pag71
  - `https://files.eric.ed.gov/fulltext/ED527261.pdf`

1 Algoritmos

O que é um algoritmo? Algumas respostas são possíveis, mas a ideia é que um algoritmo é um conjunto de regras bem definidas que formam uma receita para resolver algum problema computacional (ou seja, transformar um *Input* em um *Output*.

Exemplos:

- Um conjunto de números que você quer reorganizar para que fiquem em ordem crescente.
- Você tenha um mapa, uma origem e um destino e quer saber qual o caminho mais curto entre eles.
- Você tem um conjunto de tarefas que precisam ser concluídas até um prazo, e você precisa saber em que ordem elas devem ser executadas.

Algoritmos estão em todas as áreas da computação então entender as bases é essencial para fazer um bom trabalho.

Exemplos:

- Roteamento e Redes de Comunicação se baseiam em algoritmos clássicos de caminho mínimo.
- A efetividade de criptografia de chave pública se baseia em algoritmos de teoria dos números.
- Computação gráfica precisa de algoritmos geométricos.
- Banco de dados precisam de algoritmos em árvores de busca balanceadas.
- Biologia computacional usa programação dinâmica para medir a similaridade entre genomas.
- etc. etc.

Texto do relatório do conselho de ciência e tecnologia para a Casa Branca de dezembro de 2010:

Progress in Algorithms Beats Moore's Law

Everyone knows Moore's Law – a prediction made in 1965 by Intel co-founder Gordon Moore that the density of transistors in integrated circuits would continue to double every 1 to 2 years. Fewer people appreciate the extraordinary innovation that is needed to translate increased transistor density into improved system performance. This effort requires new approaches to integrated circuit design, and new supporting design tools, that allow the design of integrated circuits with hundreds of millions or even billions of transistors, compared to the tens of thousands that were the norm 30 years ago. It requires new processor architectures that take advantage of these transistors, and new system architectures that take advantage of these processors. It requires new approaches for the system software, programming languages, and applications that run on top of this hardware. All of this is the work of computer scientists and computer engineers.

Even more remarkable – and even less widely understood – is that in many areas, performance gains due to improvements in algorithms have vastly exceeded even the dramatic performance gains due to increased processor speed.

The algorithms that we use today for speech recognition, for natural language translation, for chess playing, for logistics planning, have evolved remarkably in the past decade. It's difficult to quantify the improvement, though, because it is as much in the realm of quality as of execution time.

In the field of numerical algorithms, however, the improvement can be quantified. Here is just one example, provided by Professor Martin Grötschel of Konrad-Zuse-Zentrum für Informationstechnik Berlin. Grötschel, an expert in optimization, observes that a benchmark production planning model solved using linear programming would have taken 82 years to solve in 1988, using the computers and the linear programming algorithms of the day. Fifteen years later – in 2003 – this same model could be solved in roughly 1 minute, an improvement by a factor of roughly 43 million. Of this, a factor of roughly 1,000 was due to increased processor speed, whereas a factor of roughly 43,000 was due to improvements in algorithms! Grötschel also cites an algorithmic improvement of roughly 30,000 for mixed integer programming between 1991 and 2008.

The design and analysis of algorithms, and the study of the inherent computational complexity of problems, are fundamental subfields of computer science.

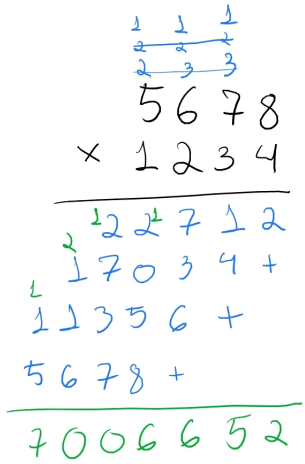
2 Multiplicação de Inteiros

Vamos definir o problema da multiplicação de inteiros:

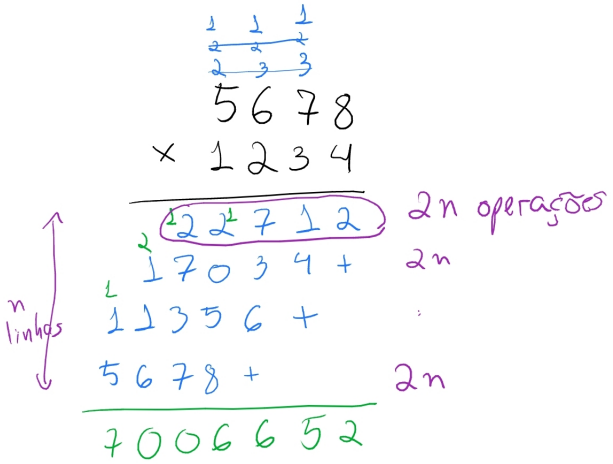
Entrada: Dois números inteiros x e y de n dígitos cada.

Saída: O produto $x.y$

Exemplo: 5678 e 1234



Intuitivamente você deve ter percebido que esse algoritmo está correto, ou seja, dados quaisquer números x e y seguindo adequadamente os passos do algoritmo você terminaria com o produto de x e y. O que talvez você não tenha pensado sobre é o número de operações básicas realizadas. Aqui consideramos operações básicas como somas e multiplicações de dois números, cada um com um único digito.



$2n^2$ operações para formar os produtos parciais. Mais $2n^2$ operações para somar os produtos parciais.

Conclusão: Precisamos aproximadamente de uma constante (aprox. 4) vezes n^2 operações para realizar essa multiplicação. Então o que acontece se você dobrar o número de dígitos? E se você quadruplicar o número de dígitos?

Será que podemos fazer melhor? De fato essa é a pergunta que faremos constantemente. Veremos a seguir um algoritmo diferente (melhor?) para multiplicar inteiros.

> *Perhaps the most important principle for the good algorithm designer is to refuse to be content. (Aho, Hopcroft e Ullman. The Design and Analysis of Computer Algorithms, 1974)*

3 Algoritmo de Karatsuba

A ideia de apresentar esse algoritmo é mostrar o vasto espaço de algoritmos possíveis para resolver diversos problemas (mesmos os mais triviais como a multiplicação de inteiros). Nessa parte ainda não temos as ferramentas necessárias para analisar se esse algoritmo é melhor que o algoritmo que aprendemos na escola.

Recebemos dois inteiros x e y de n dígitos cada e desejamos obter o produto deles. A ideia do Algoritmo de Karatsuba, é dividir x em duas partes, digamos a e b de $n/2$ dígitos e dividir y em duas partes, digamos c e d de $n/2$ dígitos.

1. calcular $a.c$
2. calcular $b.d$
3. calcular $e = (a + b)(c + d)$
4. calcular o resultado do passo 3 menos o passo 2 menos o passo 1 (3 - 2 - 1)
5. somar $a.c.10^n + b.d + e.10^{n/2}$

No nosso exemplo de 5678 e 1234 temos:

1. $a.c = 672$
2. $b.d = 2652$
3. $(a + b)(c + d) = 134.46 = 6164$
4. 3 - 2 - 1 = 2840
5. 6720000 + 2652 + 284000 = 7006652

3.1 Um algoritmo recursivo

Se pensarmos no x como uma combinação de $10^{n/2}a + b$ e $y = 10^{n/2}c + d$ então:

$$x.y = (10^{n/2}a + b).(10^{n/2}c + d) \quad (1)$$
$$= 10^n ac + 10^{n/2}(ad + bc) + bd \quad (2)$$

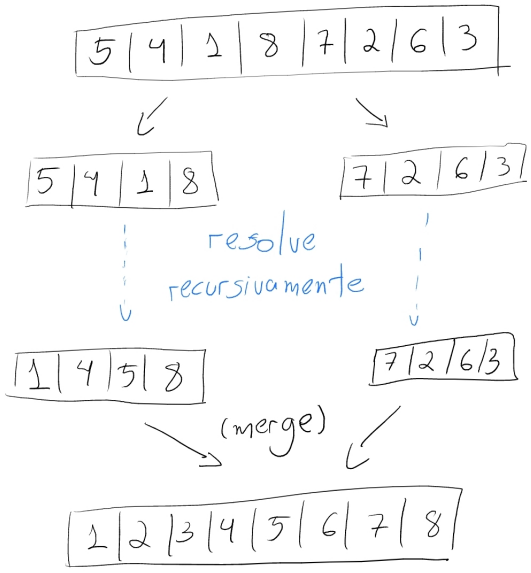
O algoritmo então calcula recursivamente ac, ad, bc e bd e calcula (2) diretamente.

Agora note que em (2) não estamos de fato interessados em obter ad e bc mas apenas na soma deles. Será que podemos fazer apenas 3 chamadas recursivas ao invés de 4? Calculamos recursivamente $(a + b)(c + d) = ac + ad + bc + bd$. E com um truque no inicio do seculo XIX, apresentado pelo matemático Gauss, se fizermos computamos $(a+b)(c+d) - ac - bd$ obtemos exatamente $ad+bc$ que é o valor que queremos.

4 Algoritmo MergeSort

O algoritmo MergeSort que você possivelmente já conhece foi apresentado por John Von Neumann em 1945, então porque estudar um algoritmo tão antigo?

- Ainda é um dos melhores algoritmos.
- Muito usado na prática.
- Melhor que seus concorrentes simples: Selection, Insertion e Bubble Sorts
- Uma boa introdução ao paradigma de divisão e conquista.
- Estabelecer o nível da turma e da disciplina
- Podemos apresentar os princípios de análise de algoritmos (pior-caso e análise assintótica)
- Podemos analisar usando uma Árvore de Recursão que vai permitir generalizar a análise de diversos outros algoritmos de divisão e conquista.



O Problema da Ordenação:
Entrada: Um arranjo de n números, não ordenados.
Saída: Os mesmos números ordenados em ordem não decrescente.
Pseudo-Código para o MergeSort:

Algoritmo 1: MergeSort

Entrada: Um arranjo
Saída: Um arranjo com os mesmos números ordenados
início
    Recursivamente ordenar a primeira metade do arranjo de entrada;
    Recursivamente ordenar a segunda metade do arranjo de entrada;
    Intercalar (Merge) as duas partes ordenadas em uma.;
fim

Pseudo-Código para o Merge:

Algoritmo 2: Merge

Entrada: A e B arranjos ordenados com $m/2$
Saída: Arranjo C de tamanho m com os mesmos elementos de A e B mas ordenados
início
    $i = 1$;
    $j = 1$;
    para k *de 1 até* m **faça**
        se $A[i] < B[j]$ **então**
            $C[k] = A[i]$;
            $i++$;
        senão
            $C[k] = B[i]$;
            $j++$;
        fim
    fim
fim
//verificar finalizações (se A ou B acabarem etc).

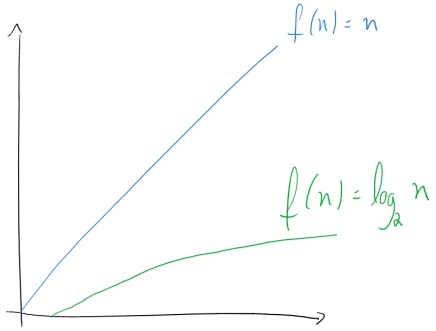
Qual o tempo de execução do MergeSort? Quantas operações básicas faz o MergeSort? Qual o número de linas de código executadas pelo MergeSort?

Primeiro nos perguntaremos qual o tempo de execução do Merge? Podemos ver que $\leq 4m + 2$ e portanto $\leq 6m$

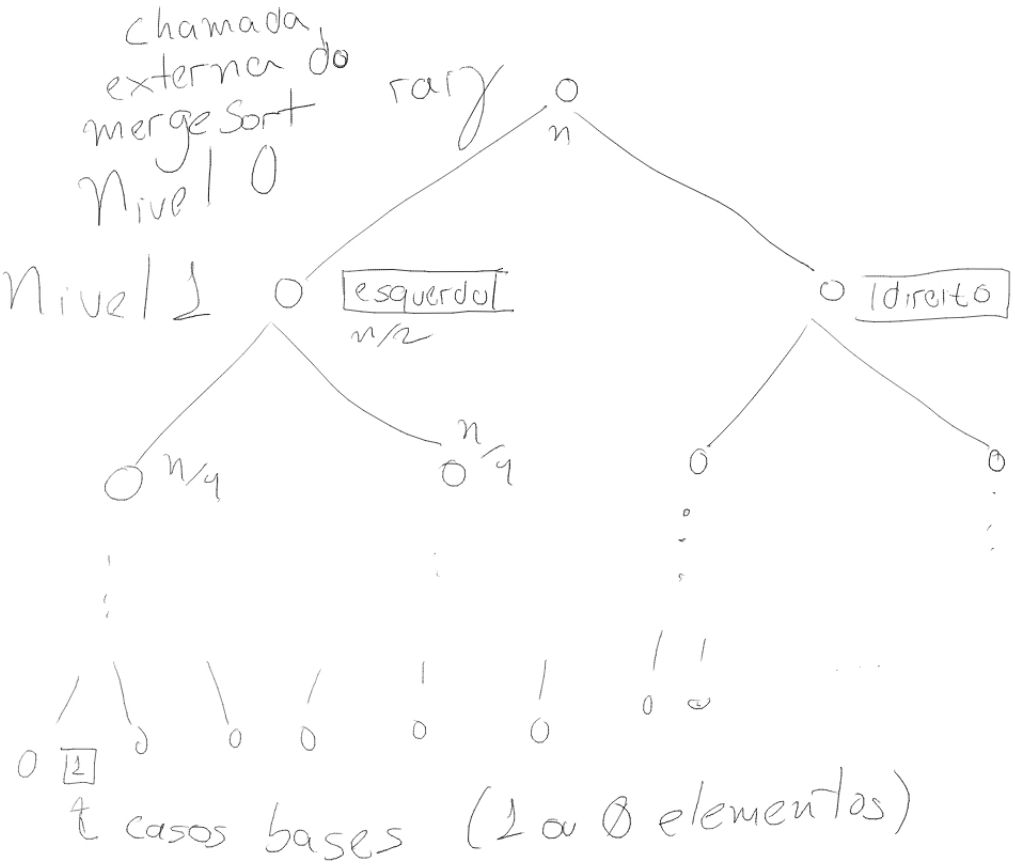
Observe que cada chamada do MergeSort gera 2 outras chamadas recursivas para MergeSort, causando uma explosão de chamadas, porém cada subproblema é menor. É essa analise do número de subproblemas versus o tamanho de cada subproblema que vai ditar a análise do MergeSort (e de outros algoritmos de divisão e conquista). Dessa forma queremos provar o seguinte teorema:

Teorema 4.1: MergeSort exige menos de $6n \log_2 n + 6n$ operações para ordenar n números.

Antes de provar esse teorema, nos perguntamos, será que esse limitante é bom? Relembre que os algoritmos mais triviais exigiam uma contante vezes n^2, enquanto o MergeSort é uma constante vezes $n \log_2 n$. Outra breve lembrança é do que é um $\log_2$, de maneira informal podemos dizer que $\log_2$ de um número, é a quantidade de vezes que você precisa dividir por 2 até chegar em 1. Então o $\log_2 4$ é 2, $\log_2 8 = 3$, $\log_2 16384 = 14$. Ou seja $\log_2 n$ é uma função que cresce devagar.



Para analisar o MergeSort vamos então desenhar uma árvore de recursão,



De maneira geral quantos níveis tem essa árvore de recursão?

Ainda, em cada nível $j = 0, 1, \ldots, \log_2 n$, quantos subproblemas existem e qual o tamanho de cada subproblema?

Agora podemos calcular a quantidade de operações necessárias por nível (sem contar o trabalho feito pelas chamadas recursivas). Para cada nível j o número de operações é $\leq 2^j.6(\frac{n}{2^j}) = 6n$. que você pode notar que não depende de j.

Dessa forma o total de operações é a quantidade de níveis vezes a quantidade de operações por níveis. Ou seja o total de operações é $\leq 6n(\log_2 n + 1) = 6n \log_2 n + 6n$. E portanto demonstramos o teorema.

4.1 Princípios da Análise de Algoritmos

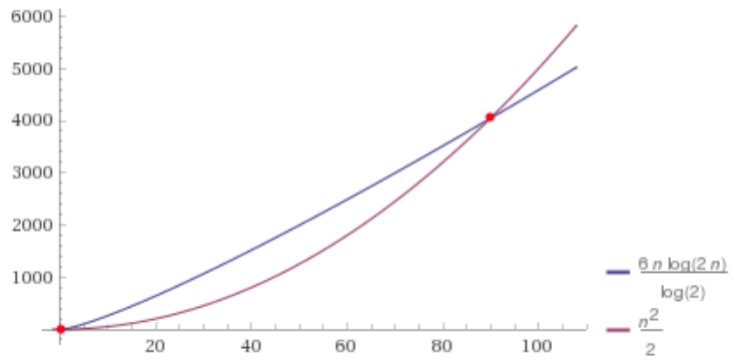
O que fizemos no caso do MergeSort foi uma análise de Pior Caso, ou seja, qualquer que seja a entrada sabemos que o algoritmo executaram em tempo $\leq 6n \log_2 n + 6n$. Esse limite também é aplicado se um adversário tentasse atribuir números de forma a deixar o algoritmo lento. Essa análise é particularmente interessante por não precisar entender a aplicação do problema, padrões de entrada e ela é usualmente mais útil e mais fácil que as alternativas:

- Análise de Caso Médio
- Análise de Melhor Caso
- Desempenho em *Benchmarks*

Outra vantagem dessa análise é que não precisamos nos preocupar com constantes e termos de menor ordem, o que torna a análise geralmente mais simples. Os motivos são que as constantes variam bastante de linguagem/máquina e compilador utilizados, coisas que não estamos nos preocupando aqui. Além disso veremos que termos de menor ordem tem muito pouco impacto na predição do comportamento dos algoritmos.

Também iremos fazer uma Análise Assintótica dos algoritmos, o que significa que estamos interessados no comportamento deles para instâncias **grandes**.

Plot:



Plot: