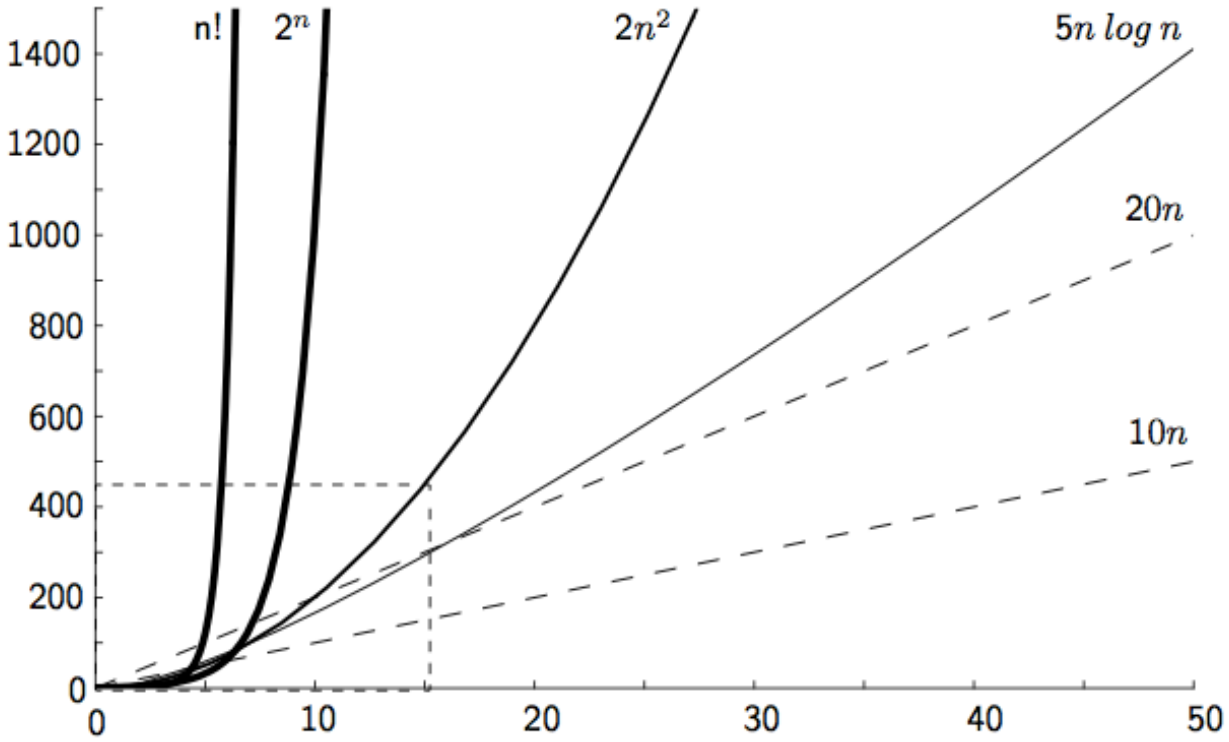


Prof. Raquel C. de Melo-Minardi, Ph.D.
Departamento de Ciência da Computação / UFMG
10 de agosto de 2020

Segundo [Ziviani, 2004], o projeto de algoritmos é fortemente influenciado pelo estudo de seus comportamentos. Após adquirir um sólido entendimento do problema e de possíveis soluções algorítmicas para o mesmo, o algoritmo projetado será enfim implementado em uma linguagem de programação para que possa ser executado um computador. A escolha do algoritmo certamente envolve considerações importantes sobre o tempo de execução e o espaço de memória ocupados. Esse tipo de escolha envolve um processo que chamamos de **análise de algoritmos**.

Há dois tipos de análises bem distintas que podemos realizar:

- **Análise de um algoritmo em particular**: Qual o custo de um determinado algoritmo na solução de um problema? Comumente, analisamos ou contamos número de vezes que cada trecho de um programa irá executar dependendo da entrada. Podemos ainda analisar o espaço em memória requerido também em função da entrada.
- **Análise de uma classe de algoritmos**: Qual o melhor algoritmo ou o algoritmo de menor custo para solução de um problema? Normalmente, analisamos toda uma família de algoritmos para resolver um problema em particular visando escolher o melhor ou mais eficiente deles.

Esse último tipo de análise normalmente envolve a estimativa de **limites** a complexidade computacional da classe de algoritmos. Interessantemente, quando se consegue determinar o menor custo possível para resolver um determinado problema ou problemas de uma certa classe, temos a medida da dificuldade inerente a esse tipo particular de problema.

Quando conseguimos provar que esse custo é o menor possível, dizemos que o algoritmo é **ótimo** para tal medida de custo. Note que comumente temos inúmeros possíveis algoritmos para resolver um mesmo problema e normalmente desejamos utilizar o melhor ou o mais eficiente.

Mas como medir então esse custo computacional?

Esse custo pode ser medido de várias maneiras mas se você disse que mediria através da medição do tempo de execução da implementação desse algoritmo em um programa de computador sendo executado, procure refletir por um instante nos possíveis problemas desse tipo de abordagem empírica.

Algumas das principais objeções apontadas por [Ziviani, 2004] são:

- os resultados dependem do compilador utilizado que pode favorecer certas construções em detrimento de outras
- os resultados dependem de hardware

Tendo em vista os problemas de se utilizar uma avaliação empírica, o que fazemos em Ciência da Computação é utilizar modelos matemáticos baseados em um computador hipotético idealizado. É como se todos os programas rodassem em um mesmo computador pois o mesmo não influenciaria em nossa análise. Define-se assim um conjunto de operações importantes a serem avaliadas e seu custo e ignora-se outro conjunto de instruções que não seriam tão significativas. Exemplo: ao analisarmos a classe de algoritmos que ordena uma lista em ordem ascendente podemos considerar apenas as operações de comparar dois

elementos e trocar esses elementos de posição e desconsiderar outras operações aritméticas, atribuições ou manipulações de índices, caso existam.

Função de complexidade

Esse modelo matemático que nos permite então medir o custo de execução de um algoritmo nada mais é do que uma função matemática que chamamos de **função de complexidade *f(n)*** em que medimos o tempo gasto para executar um algoritmo em um problema de tamanho *n*. Mais especificamente, se *f(n)* é uma medida do tempo gasto para execução, essa função será denominada **função de complexidade de tempo** do algoritmo. Note que essa função não representa o tempo de execução diretamente mas sim o número de vezes que determinadas operações consideradas relevantes são executadas para uma determinada entrada. Por outro lado, se *f(n)* é uma medida da quantidade de memória gasta para execução, essa função será denominada **função de complexidade de espaço** do algoritmo.
Atualmente, considerando a queda no custo do uso de memória, a medida do tempo é mais relevante e, a menos que se explicite o contrário, estaremos usando o termo função de complexidade denotando a função de complexidade de tempo.

Desafio

1. Projete um algoritmo e implemente uma função em Python que receba como entrada uma lista e retorne o maior elemento desse conjunto.
2. Descubra quais as operações mais relevantes em termos de tempo de execução de seu programa.
3. Tente calcular a função de complexidade de tempo *f(n)* de seu programa.
4. Você acha que seu programa é ótimo?

Veja abaixo nossa proposta de solução para a parte (1) desse desafio:

```
def maior(lista):
        maior = lista[0]
        for i in range(1, len(lista)):
                print(i)
                if (maior < lista[i]):
                        maior = lista[i]
        return maior
```

A idéia do algoritmo é ter uma variável “maior” que, a princípio, recebe a primeira posição da lista e vai tendo seu valor modificado a medida que, ao percorrermos a lista da esquerda para a direita, encontramos um valor maior que o atual maior. Ao final desse procedimento, garantimos ter o maior elemento na variável “maior“.

Quanto à parte (2) do desafio, a operação mais relevante e cara a ser executada é a comparação se cada elemento é maior que o maior elemento até o momento, ou seja, o comando condicional. Note que quando tivermos uma lista de tamanho realmente grande, é essa operação que será significativa para o crescimento do tempo de execução já que as outras linhas do programa executarão poucas vezes.
Para obtermos f(n) que é nossa parte (3) do desafio, basta contarmos quantas vezes esse "if" será executado. Como o laço se repete para "i" entre 1 e a última posição válida da lista, podemos afirmar que o "if" será executado *n-1* vezes, ou seja,

$$f(n) = n-1$$

Por fim, a parte (4) do desafio nos pergunta se esse algoritmo é ótimo. Pode-se provar [Ziviani, 2004] que para encontrar o maior elemento em uma lista de n elementos, pelo menos n-1 comparações são necessárias logo, nosso algoritmo é ótimo.

Considere a seguir, um novo desafio:

Desafio

1. Projete um algoritmo e implemente uma função em Python que receba como entrada uma lista e retorne o menor e o maior elemento desse conjunto.
2. Descubra quais as operações mais relevantes em termos de tempo de execução de seu programa.
3. Tente calcular a função de complexidade de tempo *f(n)* de seu programa.
4. Você acha que seu programa é ótimo?

Uma primeira solução para esse desafio é apresentada a seguir:

```
def maiorMenor1(lista):
        menor = maior = lista[0] # Nova variável menor
        for i in range(1, len(lista)):
                if (menor > lista[i]):
                        menor = lista[i]
                if (maior < lista[i]):
                        maior = lista[i]
```

Note que aproveitamos a função anterior e apenas adicionamos a variável “menor” que também foi inicializada com o valor da primeira posição da lista a princípio e posteriormente, dentro do laço, foi tendo seu valor alterado a medida que um novo menor valor era encontrado na lista.
Já sabemos que a operação mais relevante para a análise da complexidade desse algoritmo é o comando condicional (2). Contudo agora termos a execução desse comando duas vezes, o que nos leva a ter como resposta à parte (3) do desafio,

$$f(n)=2(n-1)$$

Seria, de forma análoga à função de complexidade da função “maior“, a função “maiorMenor1“ ótima para resolver esse problema? A resposta à parte (4) do desafio é não. Nesse caso, essa não é uma função ótima pois podemos encontrar formas mais eficientes de obter o menor e o maior elementos de uma lista sem ter de fazer *2(n-1)* comparações.

Mas como isso poderia ser feito? Alguma idéia? Você pode gastar um tempo refletindo a respeito desse código e em possíveis formas de melhorá-lo antes de passar a frente no estudo desse texto se desejar.

Apresentamos a seguir uma nova versão desse código com uma pequena melhoria que trará um certo ganho de desempenho:

```
def maiorMenor2(lista):
        menor = maior = lista[0]
        for i in range(1, len(lista)):
                if (menor > lista[i]):
                        menor = lista[i]
                elif (maior < lista[i]): # Troca de um “if“ por um “elif”
                        maior = lista[i]
        tupla = (menor, maior)
        return tupla
```

O que você observou de diferente em relação às funções “maiorMenor1“ e “maiorMenor2”? Note que elas são quase idênticas mas há uma pequena diferença na codificação que pode evitar inúmeras comparações desnecessárias.
Trocamos um “if” por um “elif” simplesmente porque quando um valor da lista avaliada for maior que o maior valor, ele não tem como ser menor que o menor valor, ou seja, estávamos fazendo inúmeras comparações inúteis. Veja como é importante ter consciência do código a ser escrito e de como pequenas modificações podem ter implicações importantes no desempenho.

Vamos então pensar em qual seria agora nossa função de complexidade. Será que ela mudou muito? Qual seria essa nova função? Pense um pouco sobre ela.

Melhor caso, caso médio e pior caso

Se você passou um tempo refletindo sobre essa nova função, deve ter percebido que ela não é tão simples de ser definida pois ela se tornou probabilística ou passou a depender não só do tamanho da entrada como também dos valores que estarão preenchendo o nossa lista.

Mas e então? Como devemos proceder para analisar a complexidade do nosso algoritmo já que não temos como antever que dados estarão na lista?

Para esse tipo de análise, precisamos distinguir três cenários:

- **Melhor caso**: corresponde ao menor tempo de execução sobre todas as possíveis execuções, ou seja, consiste em identificar em que cenário o seu algoritmo trabalhará menos.
- **Pior caso**: corresponde ao maior tempo de execução sobre todas as possíveis entradas de tamanho *n*, ou seja, identificar quando o algoritmo trabalhará mais. Trata-se de de um limite de tempo superior que nunca será sobrepassado.
- **Caso médio**: corresponde à média dos tempos de execução para todas as possíveis entradas de tamanho *n*, ou seja, seria o mais próximo do caso esperado na realidade.

Qual das três análises você considera mais interessante ou mais apropriada?
Podemos dizer que a análise de melhor caso é excessivamente otimista e que provavelmente não será frequente na prática, o que pode ser verdade. O mesmo valeria então para o pior caso que é a análise mais pessimista, mas também pouco provável. Se seguimos essa linha de raciocínio, diríamos que o caso médio é o mais interessante por ser o mais realista.
Acontece que para se calcular o caso médio, precisamos saber a distribuição de probabilidades sobre o conjunto de possíveis entradas de tamanho *n*, o que normalmente não teremos. É também comum supor uma distribuição de probabilidades em que todas as entradas sejam equiprováveis para possibilitar os cálculos mas note que, dessa forma, o que era mais realista, pode passar a não o ser. Isso torna a análise de caso médio complicada e nem tão realista assim. Normalmente, ela só é realizada em casos que façam sentido essas suposições de equiprobabilidade.
Normalmente, a análise mais importante é a de pior caso pois nos dá um **limite superior** ou uma garantia de que tempo maior não pode ocorrer.

Voltando ao nosso problema de análise de complexidade da função “maiorMenor2“, como proceder então? Devemos analisar o comportamento do algoritmo nos três cenários previamente definidos:

- Melhor caso: precisamos encontrar o tipo de entrada que levaria a nossa construção “if-elif“ a trabalhar o mínimo possível. Note que o “if” sempre será executado, então o melhor caso só poderia ser o caso em que o “elif” nunca fosse executado. Quando isso aconteceria? Quando a lista estivesse em ordem decrescente. Ou seja, a cada iteração obtivéssemos um novo menor valor e, por essa razão, nunca precisaríamos testar o maior valor que já estaria na primeira posição da lista. Nesse caso, qual seria a função de complexidade do nosso algoritmo?

$$f(n) = n-1$$

- Pior caso: de forma análoga, buscar pelo pior caso seria buscar pelo caso em que em todas as iterações tanto o “if“ quanto o “elif” seriam executados. Isso aconteceria no cenário oposto em que a lista estivesse em ordem crescente de forma que a cada iteração um novo maior elemento fosse encontrado. Nesse caso a função de complexidade seria:

$$f(n) = 2(n-1)$$

- Caso médio: para essa análise teríamos que supor que temos 0.5 de probabilidade de o primeiro “if” ser verdadeiro de modo o segundo “elif” executaria em 50% das iterações. Nossa função de complexidade deveria ser então a média da função em melhor caso $f(n) = n-1$ e em pior caso $f(n) = 2(n-1)$, que seria então

$$f(n) = 3n/2 - 3/2$$

Observe que no pior caso a função “maiorMenor2” tem o mesmo desempenho que “maiorMenor2“ mas no melhor caso, seu desempenho é muito melhor e no caso médio também é um pouco melhor.

Seria a função “maiorMenor2” ótima?

A resposta é não e podemos sim, obter um algoritmo ainda melhor. Como? Pense um pouco a respeito de como esse algoritmo poderia ser melhorado.

A idéia é a seguinte:

1. Vamos comparar os elementos aos pares, separando-os sempre em dois subconjuntos: o dos que foram maiores e o dos que foram menores na comparação de que participaram. Note que isso nos renderia no máximo n/2 comparações (considerando um número par de elementos na lista).
2. Obteremos então o maior elemento na metade do vetor que tem os elementos que venceram as comparações como um custo ótimo de no máximo n/2-1 comparações (usando o algoritmo “maior” já apresentado anteriormente que é ótimo)
3. Obteremos o menor elemento de forma semelhante à obtenção do maior elemento a um custo também de no máximo n/2-1.

Somando essas três funções esse algoritmo teria a seguinte função de complexidade:

$$f(n) = 3n/2 - 2$$

Para exercitar, tente implementar esse algoritmo antes de olhar a implementação do mesmo a seguir:

```
def maiorMenor3(lista):
(I)     if (len(lista)%2 == 1):          # Se lista tem tamanho ímpar, faz com
                lista.append(lista[0]) # que se torne par duplicando um valor
                                       # qualquer (no caso o primeiro)

(II)    if (lista[0] < lista[1]):
                menor = lista[0]
                maior = lista[1]
        else:
                menor = lista[1]
                maior = lista[0]
        i=2;
(III)   while (i < len(lista)):
(IV)            if lista[i] > lista[i+1]:
(V)                     if lista[i] > maior:
                                maior = lista[i]
(VI)                    if lista[i+1] < menor:
                                menor = lista[i+1]
                else:
                        if lista[i+1] > maior:
                                maior = lista[i+1]
                        if lista[i] < menor:
                                menor = lista[i]
                i+=2

        tupla = (menor, maior)
        return tupla
```

Como esse código é um pouco mais elaborado e complexo, detalharemos a seguir cada um dos trechos marcados em algoritmos romanos. A função “maiorMenor3”, assim como as funções previamente vistas, recebe uma lista como argumento. A seguir declaramos as variáveis “menor” e “maior" que armazenarão o menor e o maior valores ao longo da varredura da lista. A variável “i" será usada para percorrer a lista.

- A seguir, o comando condicional mostrado em (I) serve para tratar os casos em que a lista tem tamanho ímpar. Isso é necessário pois o algoritmo trabalha comparando elementos aos pares e quando houver um número ímpar de elementos isso não funcionaria. Então o que se faz aqui, por simplicidade, é duplicar arbitrariamente um elemento qualquer da lista. Note que isso não irá interferir no resultado do mesmo pois o menor número continuará sendo o menor e o maior número continuará sendo o maior mesmo que um elemento seja duplicado. Veja o exemplo a seguir onde temos a lista 4 6 8 3 5 que tem 5 elementos. O menor elemento será 3 e o maior o 8. O que fazemos então no trecho (I) é duplicar o elemento “lista[0]” que

nesse exemplo é 4 colocando-o no final da lista através do comando “lista.append(lista[0])“ resultando em 4 6 8 3 5 4. Note que o menor elemento continuará sendo 3 e o maior 8, sem prejuízo à corretude do resultado final. A vantagem de colocar esse tratamento é não ter que tratar nenhuma excepcionalidade no código que se seguirá, o que resultaria na colocação de mais condicionais para verificar se a lista é par ou ímpar, prejudicando a eficiência e mesmo a simplicidade do código.

- O condicional destacado em (II) serve para inicializar de forma inteligente as variáveis “menor” e “maior”. Nas funções que vimos anteriormente, essas variáveis eram sempre inicializadas com o valor da primeira posição da lista. O mesmo poderia ser feito aqui mas, como trabalharemos sempre em pares, optamos por pegar o primeiro par de elementos “lista[0]” e “lista[1]” usando o menor deles para inicializar “menor” e o maior deles para inicializar “maior”.
- O código que se segue e que está destacado em (III) é um laço que inicia com “i=2”, ou seja, avaliando a lista a partir do seu terceiro elemento e a trabalhará em pares. A idéia desse laço é implementar o passo (1) do algoritmo explicado previamente em alto nível "*Vamos comparar os elementos aos pares, separando-os sempre em dois subconjuntos: o dos que foram maiores e o dos que foram menores na comparação de que participaram.*". Perceba que o último comando o laço é “i+=2” que incrementa o indexador em duas posições passando a avaliar o próximo par.
- O condicional (IV) testa qual dos elementos do par (“lista[i]” e “lista[i+1]” é o maior simulando como se a lista estivesse sendo dividida em duas metades como explicado também no passo (1): “*(…) separando-os sempre em dois subconjuntos: o dos que foram maiores e o dos que foram menores na comparação de que participaram.*”. Note que essa divisão da lista não acontece em termos de estrutura de dados mas apenas do código.
- Dentro desse condicional (IV) há o outro condicional (V) que resolve essa questão de avaliar se algum dos elementos do par é o novo menor ou maior elementos conforme explicado nos passos (2) e (3) do algoritmo "*Obteremos então o maior elemento na metade do vetor que tem os elementos que venceram as comparações como um custo ótimo de no máximo n/2-1 comparações (usando o algoritmo “maior” já apresentado anteriormente que é ótimo)*" e "*Obteremos o menor elemento de forma semelhante à obtenção do maior elemento a um custo também de no máximo n/2-1.”*. Note que a divisão da lista não ocorre então na realidade em termos de quebrar uma lista em duas mas sim no código por meio dos “ifs”.
- Por fim, observe em (VII) que essa função retorna uma tupla de duas posições a saber: a primeira é o menor elemento da lista e a segunda, o maior.

Com isso finalizamos a explicação do algoritmo que, como provado em [Ziviani, 2004], é o ótimo para o problema de se obter o menor e o maior elementos em uma lista. Finalizamos também um exemplo prático da necessidade de se avaliar a complexidade de um algoritmo em termos dos seus melhor caso, pior caso e caso médio.

A tabela a seguir resume o comparativo entre as três funções apresentadas:

	f(n)		
Algoritmos	Melhor caso	Pior caso	Caso médio
maiorMenor1	2(n-1)	2(n-1)	2(n-1)
maiorMenor2	n-1	2(n-1)	3n/2 - 3/2
maiorMenor3	3n/2-2	3n/2-2	3n/2-2

Um outro exemplo interessante...

Um outro exemplo interessante que gostaríamos de abordar envolve os tão usados algoritmos de pesquisa. Considere o problema de se pesquisar registros em um arquivo, poderia ser uma uma base de dados em geral, mas em nosso caso, considere que esses registros foram carregados previamente em uma lista estando em memória RAM. Cada um desses registros tem uma chave única, seu identificador, que é utilizada para encontrá-lo no arquivo. Poderia ser um id do UniProt ou mesmo um id do PDB (*Protein Data Bank*). O problema que se deseja resolver algoritmicamente é então: dada uma chave qualquer, localizar, caso exista, o registro que contêm essa chave. Considere implementar esse problema como um desafio.

Desafio

Projete um algoritmo e implemente uma função em Python que receba como entrada uma chave qualquer e uma lista e, localize, caso exista, o registro que contêm essa chave. Sua função deve retornar o índice da lista no qual a chave foi localizada e "-1" em caso de a chave não existir na lista.

Vamos apresentar a solução mais ingênua para esse programa no código a seguir. Veja se ela é semelhante à sua solução:

```
def pesquisa(reg, lista):
        for i in range(0, len(lista)):
                if lista[i] == reg:
                        return i
        return -1
```

Desafio

Com que tipo de lista ocorre e quais as funções de complexidade desse algoritmo no:

1. melhor caso
2. pior caso
3. caso médio

Essa função recebe um registro “reg" e uma lista “lista” e pesquisa através de um laço que percorre a lista desde a posição 0 até a última posição (*n-1*) verificando se encontrou ou não o mesmo na lista.

No melhor caso, o elemento pesquisado seria o primeiro elemento da lista e apenas uma iteração do “for” seria executada visto que há um return dentro do "if" e sua função de complexidade seria:

$$f(n) = 1$$

Obviamente que a probabilidade de tamanha sorte é pequena. No pior caso, o elemento procurado estaria na última posição da lista ou não seria encontrado no mesmo. Nesse caso, o laço seria executado até o final e a função retornaria “return -1;“ resultando na seguinte função de complexidade:

$$f(n) = n$$

Mais uma vez, temos que considerar a baixa probabilidade dessa ocorrência. O que seria então o caso médio ou caso mais esperado? Assim como discutimos no exemplo anterior, para esse tipo de análise seria preciso conhecermos a probabilidade de que cada registro fosse acessado. No contexto de bases de dados biológicas e públicas, é possível que os mantenedores da base de dados tenham estatísticas de acesso a cada registro e isso poderia gerar uma probabilidade de que cada item fosse buscado por um usuário. Seja p_i, a probabilidade de que o *i*-ésimo registro seja procurado e, considerando que para encontrar o *i*-ésimo registro sejam necessárias *i* comparações, teremos

$$f(n) = 1 \times p_1 + 2 \times p_2 + 3 \times p_3 + \ldots + n \times p_n$$

Assim, para calcular f(n), seria necessário conhecer cada p_i. Vamos supor, em casos gerais, que todos os registros tenham a mesma probabilidade de serem procurados, então $p_i = 1/n$ para todo *i* entre 1 e *n:*

$$f(n) = 1/n\ (1 + 2 + 3 + \ldots + n) = (n+1) / 2$$

Interessantemente, podemos notar que o caso médio é o valor médio entre o pior caso e o melhor caso quando considerados todos os registros equiprováveis de serem procurados.

Desafio

- Esse algoritmo é ótimo?
- Se não, como ele poderia ser melhorado?
- Tente implementar melhorias e demonstrar através da nova função de complexidade que ele é mais eficiente.

Uma possível forma de melhorar a eficiência desse código é ordenando a lista de forma que raramente seria necessário percorrê-la até final para garantir que um registro não esteja presente. Por exemplo, se tivermos a lista 2 4 6 8 10, não precisamos percorrê-la até o final para dizer que o número 3 não está presente. Bastariam para isso duas comparações (2 e 4) para saber que o 3 não está presente. Caso você ainda não tenha implementado essa solução, tente fazê-lo usando a função “sort” do Python, vista na unidade anterior.
Veja abaixo esse modificação:

```
def pesquisaOrdenada(reg, lista):
        i = 0
        while i < len(lista) and reg >= lista[i]:
                if lista[i] == reg:
                        return i
                i+=1
        return -1
```

Esse código considera que receberá uma lista em ordem crescente o que pode ser obtido através do seguinte código:

```
lista.sort()
```

A função “pesquisaOrdenado" então varre essa lista da esquerda para a direita (menor elemento para o maior) enquanto o elemento corrente for menor que o registro procurado. Caso um elemento maior ou igual ao registro procurado seja encontrado na lista, o laço é interrompido sem a necessidade de processar até o final. Qual seria a complexidade desse algoritmo?

- No melhor caso, o registro procurado será o primeiro da listae $f(n) = 1$.
- No pior caso, o registro procurado seria maior que todos os registros da lista e $f(n) = n$ assim como na versão anterior da função “pesquisa”. Contudo, esse é um caso bem mais raro de ocorrer. Com a função “pesquisa" TODO registro que não fosse encontrado, demandaria a pesquisa até o final da lista realizando n comparações.
- O caso médio também teria o mesmo custo que o da função “pesquisa" $f(n) = (n+1) / 2$

Então, há ou não vantagem nessa nova função? Embora as funções de custo sejam as mesmas, é claro que o desempenho será sim um pouco melhor visto que o pior caso é significativamente menos freqüente e que probabilidade de se interromper a busca antes é maior. Contudo, há que se considerar o trabalho prévio para se ordenar a lista que usando algoritmos eficientes teria um custo adicional de $f(n) = n \log(n)$. Mais tarde falaremos um pouco desse tipo de algoritmo de forma bem superficial.
Porém, essa abordagem de pesquisa usando uma lista ordenada tem muito mais potencial do que o utilizado até o momento. Os dois algoritmos de pesquisa que vimos até o momento são algoritmos de pesquisa

```
def pesquisaBinaria(reg, lista):
(I)     if len(lista) == 0: # Se a lista está vazia
                return -1
        esq = 0
        dir = len(lista)-1

        i = int((esq+dir)/2)
(II)    while esq <= dir and reg != lista[i]:
(III)           if reg > lista[i]:
                        esq = i + 1
                else:
                        dir = i - 1
                i = int((esq+dir)/2)
        if reg == lista[i]:
                return i
        else:
                return -1
```

sequencial, ou seja, percorrem a lista de forma sequencial do início ao fim de forma bastante ingênua. Como poderíamos fazer melhor uso dessa lista ordenada em nossas pesquisas?

Desafio

- Implemente esse algoritmo de pesquisa binária em uma lista
- Você conseguiria nos dizer qual a função de complexidade da pesquisa binária em uma lista?

Dependendo de sua idade, talvez não tenha chegado a fazer pesquisas em um catálogo telefônico ou mesmo das páginas amarelas. O catálogo é uma lista de todos os assinantes de linhas de telefone de uma cidade (pode ter mais de 1 milhão de pessoas!) e todo o ano recebíamos em nossas casas uma versão atualizada. Como fazíamos a pesquisa nesse livro quando queríamos saber o telefone de alguém? A pesquisa era feita pelo nome completo e só era possível pois a lista era organizada em ordem alfabética.
Mas pense na forma ou no algoritmo que usávamos para realizar a pesquisa. Qual o nosso modelo mental? Basicamente, abríamos a lista tentando abrir ao meio. Caso o nome que procurávamos estivesse na metade inicial, cortávamos essa metade novamente aproximadamente ao meio e repetíamos o mesmo processo até encontrar a inicial do nome que procuramos e assim até encontrar o nome exato. Esse mesmo processo pode ser utilizado para implementar um algoritmo de busca.
Informalmente, esse algoritmo seria como se segue:

- Considere que os registros das listas sejam mantidos em **ordem crescente**

- Compare a chave procurada com a chave da **posição do meio da lista**.
  - Se a chave procurada for **menor** que a chave do meio, o registro estará então **na primeira metade da lista**
  - Se a chave procurada for **maior** que a chave do meio, o registro estará **na segunda metade da lista**
- Repita o processo até que a chave seja **encontrada** ou que reste apenas um registro cuja chave não é a chave procurada, ou seja, a mesma **não se encontra na lista**.

Esse método de busca é clássico e é chamado de pesquisa binária.

Veja a seguir a implementação do algoritmo de pesquisa binária:

- O código do trecho (I) é uma validação do conteúdo da lista que simplesmente encerra a função caso a lista seja vazia.
- O laço (II) executa enquanto o registro procurado não for encontrado e a sub-lista sendo varrida ainda tiver tamanho maior ou igual a 1 (“esq <= dir”).
- Dentro desse laço, há o condicional (III) que é o responsável por fazer a subdivisão virtual da lista de acordo com o local onde o registro poderá estar. Se o registro for maior que a chave atual, "esq" vai valer a posição atual mais 1 e se for menor, “dir" vai valer a posição atual menos 1. Note que dizemos que a divisão da lista é virtual pois, de fato, a lista nunca é dividida ou copiada mas os índices “esq" e “dir" delimitam onde essa lista será varrida e a cada iteração é como se essa lista fosse dividida por dois.
- Por fim, o trecho (IV) apenas é uma verificação de se o registro foi encontrado ou não. Se não for encontrado, retorna o valor “-1” como pré-estabelecido.

Qual será então a função de complexidade desse algoritmo?

Note que o algoritmo sempre divide a entrada ao meio. Esse tipo de divisão nos leva a uma complexidade

$$f(n) = log_2(n)$$

É importante recordar que a função log é uma função que atenua os valores de entrada. Dizer que $f(n) = log_2(n)$ é o mesmo que dizer que o tempo não cresce linearmente com o tamanho da entrada como ocorre na pesquisa sequencial, mas sim cresce proporcionalmente seu logaritmo na base 2 já que a entrada é sempre dividida ao meio, o que é uma função de custo bastante eficiente.

Apenas para exemplificar o funcionamento da pesquisa binária, veja a lista abaixo no qual temos 13 elementos indexados de 0 a 12:

0	1	2	3	4	5	6	7	8	9	10	11	12
0	31	33	35	42	61	78	86	90	91	92	97	98

Suponha que desejemos pesquisar pelo número “16”.

- Na primeira iteração a lista será dividida ao meio e “16" será comparado ao elemento do meio que é “lista[6] = 78”. Como 16 < 78 (em negrito), a segunda interação avaliará a metade da esquerda

0	1	2	3	4	5	6	7	8	9	10	11	12
0	31	33	35	42	61	**78**	86	90	91	92	97	98
0	31	33	35	42	61							

- Na segunda iteração a lista será dividida ao meio novamente e “16" será comparado ao elemento do meio que é “lista[2] = 33”. Como 16 < 33 (em negrito), a segunda interação avaliará a metade da esquerda

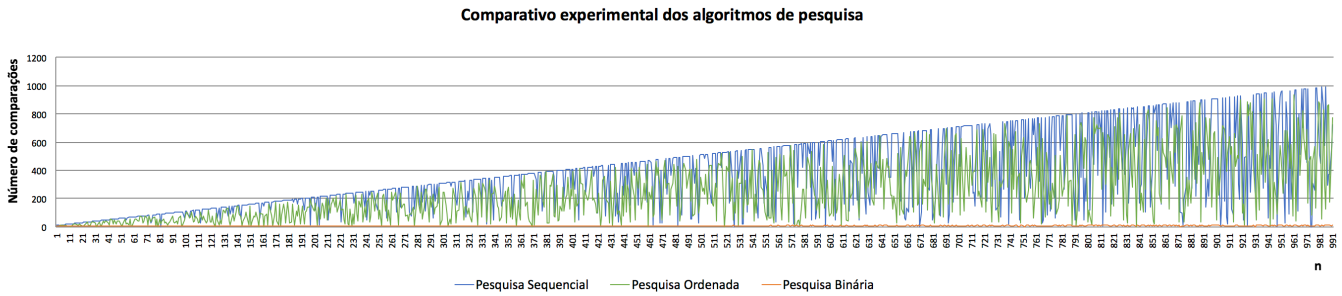
0	1	2	3	4	5	6	7	8	9	10	11	12
0	31	33	35	42	61	**78**	86	90	91	92	97	98
0	31	**33**	35	42	61							
0	31											

- Na terceira iteração a lista será dividida ao meio novamente e “16" será comparado ao elemento do meio que é “lista[0] = 0”. Como 16 != 0 (em negrito) e não há mais como dividir a lista, o processo termina e o registro 16 não foi encontrado.

0	1	2	3	4	5	6	7	8	9	10	11	12
0	31	33	35	42	61	**78**	86	90	91	92	97	98
0	31	**33**	35	42	61							
0	31											

Perceba que ao final do processo apenas 3 comparações (em negrito) foram realizadas das 13 que poderiam ter sido feitas em uma pesquisa sequencial.

A seguir, mostramos um gráfico com um comparativo experimental ilustrando o número de comparações exibido no eixo *y* pelo tamanho da entrada *n* no eixo *x*. Para gerar esse gráfico, geramos lista com números aleatórios entre 0 e 1.000 de tamanhos entre 10 e 1.000 (eixo *x*) e salvamos o número de comparações (eixo *y*) realizadas para cada lista gerada. A busca foi feita por um número fixo em todos as listas. Muitas vezes, esse número poderia não ser encontrado, gerando o pior caso dos algoritmos. Observe pela figura que o algoritmo de **Pesquisa Sequencial** tem um comportamento em pior caso linear. O algoritmo de **Pesquisa Ordenada** também apresenta um comportamento linear porém o seu pior caso (pesquisa na lista por inteiro quando o registro buscado não existe na lista) raramente acontece. Essa é a vantagem do algoritmo de



pesquisa utilizando a lista ordenada tendo em vista que sua classe de complexidade é a mesma da Pesquisa Sequencial O(n). Já a **Pesquisa Binária** é da classe de complexidade $O(\log_2 n)$ apresentando um comportamento muito mais eficiente que os outros.

Apresentamos esse interessante exemplo para introduzi-lo a outro conceito muito importante em análise de algoritmos que são as classes de complexidade. Há diversas classes de complexidade, das quais a classe log(n) é uma de grande interesse por serem problemas que podem ser resolvidos de forma bastante eficiente computacionalmente.

Comportamento assintótico de funções de complexidade

Como vimos, o parâmetro *n* que normalmente é uma medida do tamanho da entrada de um problema é muito importante para que possamos definir a **função de complexidade** que, por sua vez, nos fornece uma medida da dificuldade de se resolver um problema. Ocorre que, para valores suficientemente pequenos de *n*, qualquer algoritmo, por mais simples ou ingênuo que seja, custa pouco para ser executado ou seja executa com rapidez. Dessa forma, dizemos que a escolha de um algoritmo não é um problema crítico para problemas de tamanho pequeno [Ziviani, 2004].

Nos interessa então compreender como se comportam algoritmos para entradas realistas que são comumente muito grandes. Não custa destacar que as bases de dados biológicas são comumente numerosas e enormes. A revista *Nucleic Acids Research* em sua edição especial *Databases* publica todo ano em torno de 200 diferentes bancos de dados biológicos com temas que variam entre ácidos nucleicos (DNA, expressão gênica, genomas, fenótipos, RNS), proteína (sequências, estruturas, modelos teóricos, interações, proteômica) e outros (mutações, carboidratos, caminhos metabólicos, metabolomas, exomas, PCR, taxonomias, etc). Segundo a metabase de dados MetaBase (http://metabase.org), há mais 2.000 bases de dados biológicos comumente utilizadas. Além desse compêndio, encontramos inúmeros outros sobre coleções de bases de dados biológicos. Infelizmente, não encontramos um sumário que apontasse o volume de algumas dessas bases de dados tão utilizadas. Contudo, não é novidade que as bases de dados biológicos tem sofrido um aumento exponencial em seu volume devido aos avanços nas tecnologias de sequenciamento, entre outras tecnologias que tem permitido conhecer mais e mais os seres vivos em nível molecular. Considero que a análise de complexidade de algoritmos é uma teoria de primeira relevância para um estudante que quer se tornar um bioinformata.

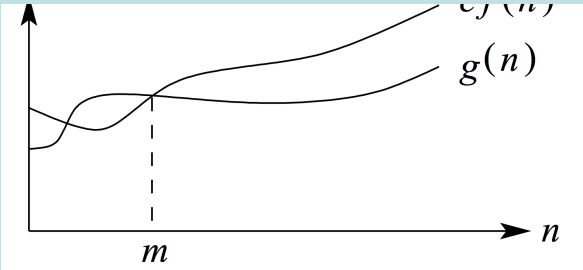
Assim, iniciaremos nossos estudos sobre o **comportamento assintótico de funções de complexidade**.

> O **comportamento assintótico** de um função de complexidade f(n) representa o seu limite quando n cresce e tende a infinito.
>
> —*Nívio Ziviani [Ziviani, 2004]*

Existe uma notação formal para **dominação assintótica** que nos diz que

Escrevemos ***g(n) = O(f(n))*** para expressar que *f(n)* domina assintoticamente *g(n)*. Lê-se *g(n)* é da ordem no máximo *f(n)*.

—Nívio Ziviani [Ziviani, 2004]



Uma função *f(n)* **domina assintoticamente** outra função *g(n)* se existem duas constantes positivas *c* e *m* tais que, para $n \geq m$, temos $|g(n)| \leq c \times |f(n)|$.

—Nívio Ziviani [Ziviani, 2004]

Essa definição formal pode confundir um pouco o aluno proveniente das áreas não exatas mas a informação principal que queremos que você assimile é que cada algoritmo tem sua função de complexidade e que uma função pode dominar outra assintoticamente, ou seja, ser sempre igual ou superior a outra para n tendendo a infinito.

Sumário

Em outras palavras, haverão algoritmos que, para entradas significativamente grandes, serão sempre mais caros ou demorados que outros. Assim, é importante que saibamos encontrar os limites da complexidade de um algoritmo e mais ainda saber escolher o melhor algoritmo para um dado problema em Bioinformática.

Notação *O*

A notação *O* (chamada em inglês de *big-O* ou em português de O) serve para denotar o limite superior de uma função de complexidade. Quando dizemos que um algoritmo é *O(n)* significa que seu tempo de execução crescerá linearmente com a entrada e nunca passará disso, embora possam haver casos em que se comporte melhor. Dizer que um algoritmo tem função de complexidade $f(n) = (n+1)^2$ e que, portanto, será $O(n^2)$ significa dizer que ele tem um limite superior que nunca passará de quadrático. Ou seja, nunca poderá ser cúbico ou exponencial por exemplo. Há outras notações que nos dão o limite inferior (Ω) ou mesmo um limite superior e inferior ou firme (Θ). Há ainda as notações o e ω. Devido ao maior grau de complexidade e menor nível de utilidade em computação e em bioinformática, não nos aprofundaremos nesse conceito. Os alunos mais curiosos e que desejarem conhecer mais sobre esse tema ou mesmo se aprofundarem após esse curso devem buscar informação no livro de Thomas Cormen [Cormen, 2009] considerado um dos mais influentes nesse tópico. Apesar do formalismo requerido por esse tópico, o livro é considerado bastante didático e conta com tradução em português.

Classes de complexidade

Se *f(n)* é uma função de complexidade para um determinado algoritmo, então dizemos que *O(f(n))* é a sua complexidade assintótica. Muitas vezes, diversos algoritmos pertencem a uma mesma **classe de complexidade *O(f(n))*** e podemos dizer que eles são **equivalentes em termos de custo computacional**.

Podemos usar essas classes de equivalência para comparar dois ou mais algoritmos. Por exemplo, suponha que um algoritmo é $f(n) = 2n^2$ e o outro é $g(n) = 100n$. Qual deles é melhor?

A resposta é: depende.

- Para n < 50, o programa quadrático ($f(n) = 2n^2$) se comporta melhor que o linear ($g(n) = 100n$).
- Para problemas com entradas pequenas, esse programa $O(n^2)$ se comporta melhor
- Entretanto, quando os dados de entrada crescem, o programa $O(n)$ leva menos tempo

O que ocorre é que $O(n^2)$ domina assintoticamente $O(n)$ o que nos levaria a escolher o programa da classe $O(n)$ que se comportaria melhor para entradas de tamanhos cada vez maiores. Contudo, para entradas pequenas o programa da classe $O(n^2)$ pode apresentar um comportamento melhor.

Enumeramos a seguir as principais classes de programas e exemplificamos com problemas em Bioinformática:

- ***f(n) = O(1)***: são algoritmos de complexidade **constante**, ou seja, o custo independe do tamanho da entrada pois as instruções são executadas um número fixo de vezes. Na prática, esse tipo de algoritmo não tem grande utilidade pois não realiza nenhum tipo de processamento sobre a entrada. Exemplo: um algoritmo que imprime uma mensagem fixa na tela.
- ***f(n) = O(log n)***: são algoritmos de complexidade **logarítmica**. Tipicamente, esses algoritmos são aqueles que dividem um problema em subproblemas menores. Note que um logaritmo gera um tempo de exceção que pode ser menor que uma constante grande. Por exemplo, quando $n = 1000$, $log_2 n \approx 10$, quando n é 1 milhão, $log_2 n \approx 20$. Para que $log_2 n$ seja dobrado, é preciso tomar n ao quadrado! Exemplo: não conheço nenhum problema em bioinformática que seja resolvido em complexidade logarítmica infelizmente mas um exemplo clássico de algoritmo logarítmico que acabamos de vez é o de pesquisa binária que é usado em diversas áreas, inclusive em bioinformática.
- ***f(n) = O(n)***: são algoritmos de complexidade **linear**. Esse tipo de algoritmo normalmente realiza um pequeno trabalho sobre cada entrada. Exemplo: um algoritmo que processa um sequência para calcular o conteúdo GC ou mesmo a freqüência de cada um dos 4 nucletídeos terá complexidade linear.
- ***f(n) = O(n log(n))***: tipicamente um algoritmo pertence a essa classe quando divide um problema em subproblemas menores e posteriormente ajunta as soluções para gerar a solução final. É bastante eficiente pois está entre $O(n)$ e $O(n^2)$. Quando n é 1 milhão, $n\ log_2 n$ é cerca de 20 milhões. Quando n é 2 milhões, $n\ log_2 n$ é cerca de 42 milhões, pouco mais do que o dobro. Exemplo: também não conheço um problema em bioinformática que pertença a essa classe mas um exemplo clássico de algoritmos dessa classe são os algoritmos de ordenação que recebem uma lista, a dividem em sub-listas menores, ordena-as e ajunta sucessivamente para construir a lista final ordenada. Em caso de interesse por esses algoritmos, procurar no capítulo sobre Ordenação em [Ziviani, 2004].

- ***f(n) = O(n^2)***: é um algoritmo de complexidade **quadrática**. Tipicamente ocorrem quando um algoritmo processa elementos aos pares, muitas vezes um um laço aninhado dentro de outro. Observe que quando n é 1000, *f(n)* será da ordem de 1 milhão. Quando *n* dobra, o tempo é multiplicado por 4. Por esse motivo, são úteis apenas para resolver pequenos problemas. Exemplo: o famoso algoritmo de Smith-Waterman [Waterman et al., 1981] que é base para os algoritmos de alinhamento de sequência par-a-par. Ele é quadrático sobre *n* onde *n* é o tamanho de uma das sequências. Caso as duas sequências alinhadas tenham tamanhos ordens de grandeza diferentes podemos dizer que ele é *O(mn)*, sendo m e n os comprimentos das sequências em termos de nucleotídeos ou aminoácidos. Ele é quadrático pois constrói uma matriz que tenta alinhar cada elemento da primeira sequência contra todos os elementos da segunda, depois o segundo elemento da primeira sequência e assim sucessivamente com todos os outros.

- ***f(n) = O(n^3)***: tem complexidade chamada **cúbica** sendo útil apenas em problemas muito pequenos. Note que quando *n* é 100, o número de operações é da ordem de 1 milhão e que quando *n* dobra, o tempo de execução fica multiplicado por 8. Exemplo: um exemplo clássico de problema cúbico é a multiplicação de matrizes. Mais uma vez, caso as três dimensões envolvidas nas duas matrizes (matrizes de dimensão m x n e n x l resulta em uma matriz de dimensão m x l) sejam ordens de grandeza diferentes podemos dizer que o algoritmo é O(mnl).

    - ***f(n) = O(2^n)***: é chamado **exponencial** e não são úteis na prática. Tipicamente ocorrem quando se usa a força bruta para resolver um problema. Esse jargão **força bruta** é muito utilizado e Ciência da Computação e significa que o algoritmo tenta todas as possibilidades de solução. Note que quando *n* é 20, faz-se cerca de 1 milhão de operações e quando *n* dobra, o tempo é elevando ao quadrado. Exemplo: um problema de bioinformática que pertence a essa classe é o famoso Problema do Enovelamento de Proteínas (PFP, do inglês *Protein Folding Problem*). A base é o número de aminoácidos da proteína e o expoente é o número de conformações que um aminoácido pode assumir. Você pode calcular quão complexo esse problema é? Outro exemplo interessante que pertence a essa classe é o docking de pequenas moléculas se considerar que a base é o número de ligações rotacionáveis da molécula e o expoente, o número de possível graus de liberdade.

    - ***f(n) = O(n!)***: é dito ter complexidade **fatorial**, também ocorre quando se usa força bruta e é ainda pior que 2^n. Quando $n = 20$, $20! = 2.432.902.008.176.640.000$, um número com 19 dígitos. Quando $n = 40$, $40!$ é um número com 48 dígitos! Exemplo: um problema em bioinformática tipicamente fatorial é o problema da montagem de um genoma a partir dos fragmentos onde *n* é o número de fragmentos. A montagem de um genoma nada mais é do que a tentativa de se ordenar os fragmentos da forma como eles deveriam estar encadeados no genoma e um algoritmo que tentasse todas as possibilidades afim de encontrar a melhor ou mais correta tentaria *n!* possibilidades ou todas as permutações possíveis. Todo problema que envolver permutações, combinações ou arranjos (problemas de otimização combinatória) pertencerão a essa classe.

Veja nessa tabela extraída de [Ziviani, 2004], quanto tempo programas das diversas classes de complexidade levariam para executar com entradas de tamanhos que variam entre $n = 10$ e 60:

Função de custo	10	20	30	40	50	60
n	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s	0,00006 s
n^2	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,0035 s	0,0036 s
n^3	0,001 s	0,008 s	0,027 s	0,64 s	0,125 s	0,316 s
n^5	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min	13 min
2^n	0,001 s	1 s	17,9 min	12,7 dias	35,7 anos	366 séc
3^n	0,059 s	58 min	6,5 anos	3.855 séc.	10^8 séc.	10^{13} séc.

É assustador notar que problemas exponenciais são intratáveis mesmo para entradas tão pequena quanto *n* = 50. Na prática, isso nos mostra que o problema do enovelamento de proteínas, por exemplo, não é possível de ser resolvido de forma ótima usando força bruta (ou seja, tentando todas as possibilidades), mesmo para proteínas com poucas dezenas de aminoácidos. Há inúmeros problemas em Bioinformática que são exponenciais ou fatoriais. Na verdade, eu diria que a maioria dos problemas interessantes em boinformática são intratáveis.

Você deve estar se perguntando então se temos boas perspectivas com o desenvolvimento de hardware mais avançado que processe os programas mais rapidamente ou em paralelo. Veja mais uma tabela extraída de [Ziviani, 2004] que faz a estimativa do tamanho dos problemas que poderiam ser resolvidos se tivéssemos computadores mais rápidos:

Função de custo de tempo	Computador atual	Computador 100x mais rápido	Computador 1.000x mais rápido
n	t_1	100 t_1	1.000 t_1
n^2	t_2	10 t_2	31,6 t_2
n^3	t_3	4,6 t_3	10 t_3
2^n	t_4	t_4 + 6,6	t_4 + 10

Note que o problema não é o poder computacional mas, de fato, a complexidade dos problemas. Um problema exponencial não será possível de resolver mesmo com um computador que seja 1.000 vezes mais rápido que o atual ou que tenha 1.000 núcleos e se possa resolver o problema em paralelo (supondo que isso possa ser feito). Note que, no caso de um problema exponencial, mesmo que o computador fosse 1.000 vezes mais rápido, poderíamos resolver um problema quase do mesmo tamanho que t_4 ou um problema de

tamanho 10 unidades maior (t_4 + 10). Pensando no problema do enovelamento de proteínas, se hoje fossemos capazes de enovelar computacionalmente uma proteína de t_4 aminoácidos, um computador 1.000 vezes mais rápido conseguia fazê-lo para uma proteína de t_4 + 10 aminoácidos.

Sumário

Há problemas cujos algoritmos pertencem a classes de funções **polinomiais** e outros que pertencem a classes **exponenciais**. Comumente, dizemos que os fatoriais pertencem à classe exponencial também. A distinção entre essas duas grandes classes é bastante significativa quando o tamanho do problema cresce. Os algoritmos polinomiais são úteis na prática, enquanto os polinomiais não. Assim, um problema é considerado **intratável** quando não existe um algoritmo polinomial para resolvê-lo e **bem resolvido** quando existe.

Referências

[Cormen, 2009] Cormen, Thomas H. Introduction to algorithms. MIT press, 2009.

[Waterman et al., 1981] Smith, Temple F., and Michael S. Waterman. "Identification of common molecular subsequences." Journal of molecular biology147.1 (1981): 195-197.

[Ziviani, 2004] Ziviani, Nivio. Projeto de algoritmos: com implementações em Pascal e C. Vol. 2. Thomson, 2004.