Análise de Algoritmos

Algoritmos fazem parte do dia-a-dia das pessoas: instruções para o uso de medicamentos, ou indicações de como montar um aparelho eletrodoméstico ou ainda uma receita culinária são exemplos de algoritmos.

Podemos ver um algoritmo como uma seqüência de ações executáveis para a obtenção de uma solução para um determinado tipo de problema. Dijkstra define algoritmo como sendo a descrição de um padrão de comportamento, expresso em termos de um conjunto finito de ações. Ao executarmos a operação a+b percebemos um mesmo padrão de comportamento, mesmo se a operação for realizada para valores diferentes de a e b.

Ainda sobre o conceito de algoritmo, sabe-se que a origem do termo deve-se ao matemático persa *Abu Já'far Mohamed ibn Musa al Khowarizmi* que é, talvez, o autor dos primeiros algoritmos conhecidos, ou registrados, na história (estima-se em torno de 825 D.C.)

De fato, algoritmos existiam muitos anos antes da construção do primeiro computador na década de 1940. No entanto, primordialmente eram dedicados à solução de problemas aritméticos. Atualmente, como é o caso da maioria dos algoritmos que estudaremos, também temos algoritmos para solucionar problemas computacionais que não tem natureza numérica.

Para os propósitos de nossos estudos, um problema será uma pergunta de caráter geral a ser respondida. Normalmente, um problema tem vários parâmetros cujos valores são variáveis, tem uma descrição geral de todos os seus parâmetros, e um enunciado de quais propriedades, ou solução, que deve satisfazer. Uma instância de um problema é obtida ao se fixarem valores particulares de todos os parâmetros do problema.

Por exemplo: para o problema de ordenar uma lista finita de n números $[A_1, A_2,...,A_n]$, a solução consiste nesses mesmos números em ordem crescente (ou decrescente), isto é, uma permutação de índices, p: $(1,...,n) \rightarrow (1,...,n)$, tal que $[A_{i(1)},A_{i(2)},...,A_{i(n)}]$ satisfaça $A_{i(j)} \leq A_{i(j+1)}$, para todo j, $1 \leq j \leq n-1$. Uma instância do problema poderia ser [5, 7, 0, -2] onde n = 4, e a solução: [-2, 0, 5, 7].

Um algoritmo é, em geral, uma descrição passo a passo de como um problema é solucionável. A descrição deve ser finita, e os passos devem ser bem definidos, sem ambigüidades, e executáveis computacionalmente.

Diz-se que um algoritmo resolve um problema P se este algoritmo recebe qualquer instância I de P e sempre produz uma solução para esta instância I. Diz-se, então, que o algoritmo resolve I, e que I é um dado de entrada do algoritmo, e a solução é um *dado de saída* do algoritmo. Enfatizamos que, para qualquer dado de entrada I, o algoritmo deverá ser executável em tempo finito e, produzir uma solução correta para P.

Supondo que se tenha uma descrição passo a passo com todas as propriedades de um algoritmo, exceto unicamente da garantia de que é executável em tempo finito para qualquer instância I. então, tal descrição será chamada de procedimento.

Após o desenvolvimento de um algoritmo, pode-se demonstrar matematicamente que ele calcula a resposta correta para quaisquer dados de entrada que satisfaçam as condições estabelecidas, isto é, pode-se demonstrar a *certificação* do algoritmo.

Algoritmo

Algoritmo é definido como um procedimento, ou função, que consiste de um conjunto de regras não ambíguas que especificam, para cada entrada, uma seqüência finita de operações, terminando com uma saída correspondente.

Um algoritmo resolve um problema quando, para qualquer entrada, produz uma resposta correta, se forem concedidos tempo e memória suficientes para sua execução. O fato de um algoritmo resolver (teoricamente) um problema não significa que seja aceitável na prática.

Os recursos de espaço e tempo requeridos têm grande importância em casos práticos. Às vezes, o algoritmo mais imediato está longe de se razoável em termos de eficiência. Um exemplo é o caso da solução de sistemas de equações lineares. O método de Cramer, calculando o determinante através de sua definição, requer milhões de anos para resolver um sistema 20×20. Um sistema como esse pode ser resolvido em tempo razoável pelo método de Gauss.

Analisar um algoritmo significa prever os recursos de que o algoritmo necessitará. Ocasionalmente, recursos como memória, largura de banda de comunicação ou hardware de computador são a principal preocupação, mas com freqüência é o tempo de computação que desejamos medir.

Em geral, pela análise de vários algoritmos candidatos para resolver um problema, pode-se identificar facilmente um algoritmo mais eficiente. Essa análise pode indicar mais de um candidato viável, mas vários algoritmos de qualidade inferior em geral são descartados no processo.

O crescente avanço tecnológico, permitindo a criação de máquinas cada vez mais rápidas, pode, ingenuamente, parecer ofuscar a importância da complexidade de um algoritmo. Entretanto acontece exatamente o inverso, pois as máquinas ao se tornarem mais rápidas, passam a ter capacidade para solucionar problemas maiores e é a complexidade do algoritmo que determina o novo tamanho máximo de problema resolvível.

Para um algoritmo rápido, qualquer memória na velocidade de execução das operações básicas é sentida, e o conjunto de problemas resolvíveis por ele aumenta sensivelmente. Esse impacto é menor no caso de algoritmos menos eficientes.

Complexidade

A primeira medida da dificuldade de um problema se relaciona com o tempo necessário para resolvê-lo. Para um dado problema, a complexidade de tempo é uma função que relaciona o tamanho de uma entrada (ou instância) ao tempo necessário para resolvê-la.

Outra medida de dificuldade seria a quantidade de espaço necessária ocupada, que não estaremos verificando em nosso resumo. Tipicamente, queremos saber em quanto tempo podemos resolver todas as instâncias de um problema.

Mais especificamente, associamos a cada entrada um valor inteiro chamado *tamanho* que é a medida da quantidade de dados de entrada. Procuramos então uma função do tamanho da instância, que expresse o tempo que o algoritmo necessita para resolver aquela instância. Essa função é chamada complexidade de tempo.

Um exemplo: ... *quanto vale S após a execução do algoritmo abaixo?*

```
1   S ← 0
2   para i ← 2 até n-2 faça
3       para j ← i até n faça
4           S ← S + 1
```

Solução:

Se n ≥ 4, então ao final da execução das linha 1 – 4, temos

$S = (n-1) + (n-2) + (n-3) + ... + 4 + 3 = (n+2)(n-3)/2$

$S = n^2/2 - n/2 - 3$

.... obtemos a resposta com uma função associada ao tamanho da instância *n*.

Para um dado problema, podemos ter vários algoritmos para resolvê-lo, cada um deles com uma complexidade diferente. A menor dessas complexidades é chamada *cota superior de complexidade n* do problema considerado. Esta cota nos diz que, para instâncias arbitrárias de tamanho *n*, podemos resolver o problema em tempo menor ou igual a *cota superior(n)*.

Ou seja, nós não devemos ficar satisfeitos com um algoritmo de complexidade pessimista maior que a *cota superior(n)*, pois outro algoritmo de complexidade pessimista com *cota superior(n)* já existiria.

Por esta razão, algoritmos são analisados para se determinar a sua complexidade, deixando-se em aberto a possibilidade de se reduzir ainda mais a *cota superior(n)*, se for descoberto um novo algoritmo cuja complexidade pessimista seja menor do que qualquer algoritmo já conhecido.

Por outro lado, podemos estar interessados em saber quão rapidamente podem-se resolver todas as instâncias de um dado problema, independentemente do algoritmo que exista ou que se possa descobrir no futuro. Isto é, estamos interessados na complexidade inerente ou intrínseca do problema que é chamada *cota inferior de complexidade n* do problema.

Esta cota nos diz que nenhum algoritmo pode resolver o problema com complexidade pessimista menor do que *cota inferior(n)*, para entradas arbitrárias de tamanho *n*.

A cota é então uma propriedade do problema considerado, e não de um algoritmo particular. Por exemplo, para o problema da ordenação de *n* números inteiros, demonstra-se que sua *cota inferior* é uma função proporcional a *nlgn*, ou seja, qualquer algoritmo para resolver este problema necessitará de um tempo maior ou igual ao valor desta função, para entradas de tamanho *n*, na situação mais desfavorável.

Notamos a distinção entre cotas superior e inferior ao considerarmos que ambas são mínimas sobre a complexidade máxima (pessimista) para entradas de tamanho *n*. Entretanto, *cota inferior(n)* é o mínimo, sobre todos os algoritmos possíveis da complexidade máxima; enquanto que *cota superior(n)* é o mínimo sobre todos os algoritmos existentes, da complexidade máxima.

Aprimorar uma *cota superior* significa descobrir um algoritmo que tenha uma *cota inferior* melhor do que a existente, nós nos concentramos em técnicas que nos permitam aumentar a precisão com a qual o mínimo, sobre todos os algoritmos possíveis, pode ser limitado. Esta distinção nos leva às diferenças nas técnicas desenvolvidas em análise de complexidade.

Embora existam aparentemente duas funções de complexidade de problemas, as *cotas superior* e *inferior*, o objetivo final é fazer estas duas funções coincidirem: *cota inferior(n) = cota superior(n)*, Quando isto ocorre, temos o algoritmo "ótimo", para a maioria dos problemas, este objetivo ainda não foi alcançado.

Caso Pessimista e Caso Médio

Tradicionalmente, a complexidade pessimista tem sido a de maior interesse teórico, pelas razões mencionadas acima. Ocasionalmente, ela é de interesse prático. Por exemplo, um sistema de controle de tráfego de metrô com complexidade média aceitável pode ser considerado inútil, se o caso pessimista puder ocorrer, embora raramente, e causar um desastre.

Entretanto, nos últimos anos, tem havido um interesse maior na análise da complexidade média dos algoritmos, uma vez que ela é mais útil na prática. Por exemplo, o conhecido *algoritmo simplex* para programação linear necessita de um tempo igual a uma função exponencial no tamanho da instância, no caso pessimista, más, para maioria das instâncias encontradas na prática, o algoritmo é extremamente rápido.

Essa abordagem, no entanto, tem as suas dificuldades. Calcular a média sobre muitos casos é uma operação consideravelmente complicada. Além disso, embora a média tenha isoladamente o seu valor, é desejável achar-se a distribuição dos tempos de solução e a variância, mas estes cálculos são adicionais difíceis e freqüentemente não são feitos na prática.

Existe também uma forte objeção no sentido de que as hipóteses de distribuição probabilísticas dos dados de entrada (usualmente simples para tornar a análise tratável matematicamente) são irrealistas. Apesar das objeções, a análise do caso médio tem a sua importância e continua sendo pesquisada por cientistas de computação.

Analisando a execução de um algoritmo

O projeto de algoritmos é fortemente influenciado pelo estudo de seus comportamentos. Depois que um problema é analisado e decisões de projeto são finalizadas, o algoritmo tem que ser implementado em um computador. Neste momento o projetista tem que estudar as várias opções de algoritmos a serem utilizados, onde os aspectos de tempo de execução e espaço ocupado são considerações importantes. Muitos destes algoritmos são encontrados em áreas tais como pesquisa operacional, otimização, teoria dos grafos, estatística, probabilidades, entre outras.

Na área de análise de algoritmos, existem dois tipos de problemas bem distintos, como aponta Knuth:

1. Análise de um algoritmo particular. Qual é o custo de usar dado algoritmo para resolver um problema específico? Neste caso, características importantes do algoritmo em questão devem ser investigadas, geralmente uma análise do número de vezes que cada parte do algoritmo deve ser executada, seguida do estudo da quantidade de memória necessária.
2. Análise de uma classe de algoritmos. Qual é o algoritmo de menor custo possível para resolver um problema particular? Neste caso, toda uma família de algoritmo para resolver um problema específico é investigada com o objetivo de identificar um que seja o melhor possível. Isto significa colocar limites para a complexidade computacional dos algoritmos pertencentes à classe. Por exemplo, é possível estimar o número mínimo de comparações necessárias para ordenar *n* números através de comparações sucessivas.

Antes de pensar na análise de um algoritmo, devemos ter um modelo da tecnologia de implementação que será usada, inclusive um modelo dos recursos dessa tecnologia e seus custos. Nesse resumo, faremos a suposição de um modelo de computação genérico com um único processador, a "RAM" ou *Máquina de Acesso Aleatório* representará nossa tecnologia de implementação e nossos algoritmos serão implementados na forma de um programa de computador. No modelo RAM, as instruções são executadas uma após a outra, sem operações concorrentes. A RAM contem instruções comumente encontradas em computadores reais: instruções aritméticas (soma, subtração, multiplicação, divisão, resto, piso, teto), de movimentação de dados (carregar, armazenar, copiar) e de controle (desvio condicional e incondicional, chamada e retorno de sub-rotina), observando que cada uma dessas instruções demora tempo constante para ser executada. Quanto a tipos de dados, no RAM, serão números inteiros ou reais supondo um limite razoável para o tamanho da palavra de dados. Por exemplo, ao trabalharmos com entradas de tamanho *n*, em geral supomos que os inteiros são representados por Clgn bits para alguma constante C . Exigimos C ≥ 1 para que cada palavra possa conter o valor de *n*, permitindo-nos indexar os elementos de entradas individuais, e limitamos C a uma constante para que o tamanho da palavra não cresça arbitrariamente. Neste modelo não tentaremos modelar a hierarquia de memória (comum em computadores contemporâneos), ou seja, não modelaremos caches ou memória virtual (como na paginação) para simplificar nosso trabalho de análise.

Ao simular o processamento de um algoritmo o conjunto de operações a serem executadas dever ser especificado, assim como o custo associado com a execução de cada operação. O usual é ignorar o custo de algumas das operações envolvidas e considerar apenas as operações mais significativas. Por exemplo, para algoritmos de ordenação consideramos apenas o número de comparações entre os elementos do conjunto a ser ordenado e ignoramos as operações aritméticas, de atribuição e manipulações de índices, caso existam.

Para medir o custo de execução de um algoritmo é comum definir uma função de custo ou função de complexidade *f*, onde *f(n)* é a medida de tempo necessário para executar um algoritmo que resolve um problema de instância *n*. Esta complexidade não representa o tempo diretamente para execução, mas sim o número de vezes que determinada operação relevante é executada.

Como a medida do custo de execução de um algoritmo depende principalmente do tamanho da entrada dos dados (a instância) devemos distinguir três cenários possíveis na execução: o melhor caso, o pior caso e o caso médio. O melhor caso corresponde ao menor tempo de execução sobre todas as instâncias possíveis. O pior caso corresponde ao maior tempo sobre todas as entradas de tamanho *n*. Considerando a complexidade baseada na análise de pior caso então o custo de aplicar o algoritmo nunca é maior do que *f(n)*.

O caso médio (ou caso esperado) corresponde à média dos tempos de execução de todas as entradas de tamanho *n*. Na análise do caso esperado, uma distribuição de probabilidades sobre o conjunto de entradas de tamanho *n* é suposta, e o custo médio é obtido com base nesta distribuição. Por esta razão, a análise do caso médio é muito mais difícil de obter do que as análises do melhor e do pior caso. É comum supor igualmente prováveis, embora, na prática isto nem sempre seja verdade.

Para ilustrar estes conceitos considere "*o problema de acessar registros de um arquivo*". Cada registro contém uma chave única que é utilizada para recuperar registros do arquivo. Dada uma chave qualquer o problema consiste em localizar o registro que contenha esta chave. O algoritmo de pesquisa mais simples é o que faz uma pesquisa seqüencial. Este algoritmo examina os registros na ordem em que eles aparecem no arquivo, até que o registro procurado seja encontrado ou fique determinado que o mesmo não se encontra no arquivo.

Seja *f* uma função de complexidade tal que *f(n)* é o número de registros consultados no arquivo, isto é, o número de vezes que a chave de consulta é comparada com a chave de cada registro. Casos a considerar:

melhor caso: $f(n) = 1$;
pior caso: $f(n) = n$;
caso médio: $f(n) = (n+1)/2$.

O melhor caso ocorre quando o registro procurado é o primeiro consultado. O pior caso ocorre quando o registro procurado é o último consultado, ou então não está presente no arquivo; para tal é necessário realizar n comparações.

Para o estudo do caso médio vamos considerar que toda pesquisa recupera um registro, não existindo pois pesquisa sem sucesso. Se p_j for a probabilidade de que o i-ésimo registro seja procurado, e considerando que para recuperar o *i-ésimo* registro são necessárias i comparações, então $f(n) = 1 \cdot p_1 + 2 \cdot p_2 + 3 \cdot p_3 + \ldots + n \cdot p_n$.

Para calcular $f(n)$ basta conhecer a distribuição de probabilidade p_i.

Se cada registro tiver a mesma probabilidade de ser acessado que todos os outros, então $p_i = 1/n$, $1 \leq i \leq n$. Assim,

$$f(n) = \frac{1}{n}(1 + 2 + 3 + ... + n) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}$$

A análise do caso esperado para a situação descrita revela que uma pesquisa com sucesso examina aproximadamente *metade* dos registros.

Outro exemplo:

Um algoritmo de classificação construído numa abordagem incremental.

Problema: *Rearranjar um vetor A[1..n] de modo que ele fique em ordem crescente*

... uma solução empregando a classificação por inserção:

```
Inserção (A, n)
1   para j ← 2 até n faça
2       x ← A[j]
3       i ← j - 1
4       enquanto i > 0  e  A[i] > x faça
5               A[i+1] ← A[i]
6               i ← i - 1
7       A[i+1] ← x
```

Iniciamos a análise verificando *como* e *porquê* funciona corretamente:

Invariante: no início de cada iteração na linha 1 e antes de executar o bloco de linhas 2-7, o vetor A[1..*j*-1] é crescente:

1		crescente		j-1	j					n
78	79	92	97	99	35	98	35	95	32	99

Vejamos como essa propriedade é válida para a classificação por inserção:

Início: começamos mostrando que o invariante é válido antes da primeira iteração, quando j = 2 na linha 1, então, o sub-vetor A[1..*j*-1] consiste apenas no único elemento A[1], que é de fato o elemento original em A[1]. Além disso, esse sub-vetor é ordenado (de forma trivial), e isso mostra que o invariante é válido antes da primeira iteração da repetição *para...faça...*

Manutenção: em seguida, verificamos que a propriedade de que o invariante é mantido a cada iteração na linha 1. Informalmente, o corpo do laço de repetição *para...faça...* exterior funciona deslocando-se A[*j*-1], A[*j*-2], A[*j*-3] e daí por diante uma posição à direita, até ser encontrada a posição adequada para A[*j*] (linhas 3-6), e nesse ponto o valor de A[*j*] é inserido (linha 7). Para um tratamento mais formal, precisamos mostrar o invariante também no laço interno *enquanto... faça...*

Término: encerrando, examinamos o que ocorre quando o laço *para...faça...* termina. Isso ocorre quando j > n + 1, e aí observamos que o vetor apontado no invariante será A[1..n], porém em seqüência ordenada. Atingimos portando nossa meta, significando que o algoritmo está correto.

Note que essa verificação é semelhante à indução matemática onde procuramos provar uma propriedade válida, demonstrando um caso básico e uma etapa indutiva. Aqui, mostrar que o invariante é válido antes da primeira iteração é equivalente ao caso básico, e mostrar que o invariante é válido de uma iteração para outra equivale à etapa indutiva. Na conclusão da prova, diferimos da indução matemática pois esta é usada indefinidamente enquanto nós "paramos a indução" quando o laço de repetição termina.

E agora analisando sua execução podemos perguntar:

Quanto tempo o algoritmo consome?

A análise de tempo considera a entrada, no caso o tamanho do vetor A que será ordenado pois estamos interessados no pior caso para cada *n*. Para cada "tamanho" *n* da "entrada", escolha um vetor A que force o algoritmo a consumir o máximo de tempo, que podemos identificar com *f(n)*. Em outras palavras, para cada *n* fixo, o algoritmo consome tempo *f(n)* para algum A[1..n].

Como dissemos na introdução consideramos um modelo de máquina, o RAM, para implementar nossos algoritmos, más por um instante considere que o computador é real: *o tempo dependeria do computador em uso*?

Na prática observamos que o tempo não depende tanto do computador. Ao mudar de um computador para outro, o consumo de tempo do algoritmo é, apenas e tão somente, multiplicado por uma constante.

Por exemplo, se $f(n) = 250n^2$ em um computador, então $f(n) = 500n^2$ em um computador duas vezes mais lento e $f(n) = 25n^2$ em um computador dez vezes mais rápido.

Vamos, então, determinar *f(n)* para o algoritmo de inserção, e para isso anotamos, à direita das linhas 1–7, o número de vezes que cada linha é executada *no pior caso*. Suponha também que a execução de qualquer linha do algoritmo consome 1 unidade de tempo.

Inserção (A, n)	**contagem**
1 **para** j ← 2 **até** n **faça**	n
2 x ← A[j]	n-1
3 i ← j - 1	n-1
4 **enquanto** i>0 e A[i]>x **faça**	2+3+ ... +n
5 A[i+1] ← A[i]	1+2+... + n-1
6 i ← i - 1	1+2+... + n-1
7 A[i+1] ← x	n-1

O tempo no pior caso será a *soma* da coluna direita acima: $f(n) = (3/2)n^2+(7/2)n-4$.

Se tivéssemos levado em conta o tempo exato de execução de cada linha, no caso de um computador real, obteríamos coeficientes diferentes de 3/2, 7/2 e –4, mas a expressão de *f(n)* ainda seria da forma:

$$An^2 + Bn + C.$$

O coeficiente 3/2 de n^2 não é importante: ele não depende do algoritmo, mas de nossa hipótese de 1 unidade de tempo por linha.

Já o "n^2" é fundamental: ele é característico do algoritmo em sí e não depende nem do computador nem dos detalhes da implementação do algoritmo.

Em resumo, a única parte importante em *f(n)* é o "n^2".

Todo o resto depende da implementação particular do algoritmo e do particular computador que estivermos usando.

Qual seria o consumo de tempo no melhor caso?

Para essa análise precisamos observar que o vetor de entrada já está classificado, e assim a execução das linhas 4 a 6 será modificada pois o teste A[i]>x na linha 4 evitará a entrada nas linhas 5 e 6, e dessa forma a execução no melhor caso resulta:

Inserção (A, n)	**contagem**
1 **para** j ← 2 **até** n **faça**	n
2 x ← A[j]	n-1
3 i ← j - 1	n-1
4 **enquanto** i>0 e A[i]>x **faça**	1+1+...+1 = n-1
5 A[i+1] ← A[i]	0
6 i ← i - 1	0
7 A[i+1] ← x	n-1

Então o tempo no melhor caso será : $f(n) = 5n-4$.

E o consumo no caso médio?

Para esse exemplo não vamos analisar o caso médio ...

Em geral nos concentramos na analise do tempo de execução de um algoritmo no *pior caso*, ou seja, o tempo de execução mais longo para qualquer entrada de tamanho *n*.

Exercícios de fixação

1. *Quanto vale S após a execução do algoritmo ?*

```
1 S ← 0
2 i ← n
3 enquanto i < 0 faça
4       para j ← 1 até i faça
5               S ← S + 1
6       i ← ⌊i/2⌋
```

2. *Quantas atribuições faz o algoritmo abaixo?*

```
1 S ← 0
2 para i ← 1 até n faça
3       S ← S + i
4 devolva S
```

Escreva um algoritmo melhor que produza o mesmo efeito com menos atribuições

3. Quanto tempo consome o fragmento de algoritmo abaixo que opera sobre uma matriz A[1..n, 1..n]

```
1 para i ← 1 até n-1 faça
2       para j ← i+1 até n faça
3               para k ← i até n faça
4                       A[j,k] ← A[j,k] – A[i,k]*A[j,i]/A[i,i]
```

4. Quanto tempo consome o fragmento de algoritmo abaixo que opera sobre um vetor A[1..n]

```
1  S ← 0
2  para i ← 1 até n faça
3       S ← S + A[i]
4  m ← S/n
5  k ← 1
6  para i ← 2 até n faça
7       se (A[i] – m)^2 < (A[k] – m)^2
8        então k ← i
9  devolve k
```

5. Analise o consumo de tempo e a correção do algoritmo Intercala que recebe dois sub-vetores A[p..q] e A[q+1..r] e rearranja-os devolvendo A[p..r] ordenado:

```
Intercala (A, p, q, r)
 1  para i ← p até q faça
 2       B[i] ← A[i]
 3  para j ← q+1 até r faça
 4       B[r+q+1-j] ← A[j]
 5  i ← p
 6  j ← r
 7  para k ← p até r faça
 8       se B[i] ≤ B[j]
 9        então A[k] ← B[i]
10              i ← i+1
11        senão A[k] ← B[j]
12              j ← j-1
```

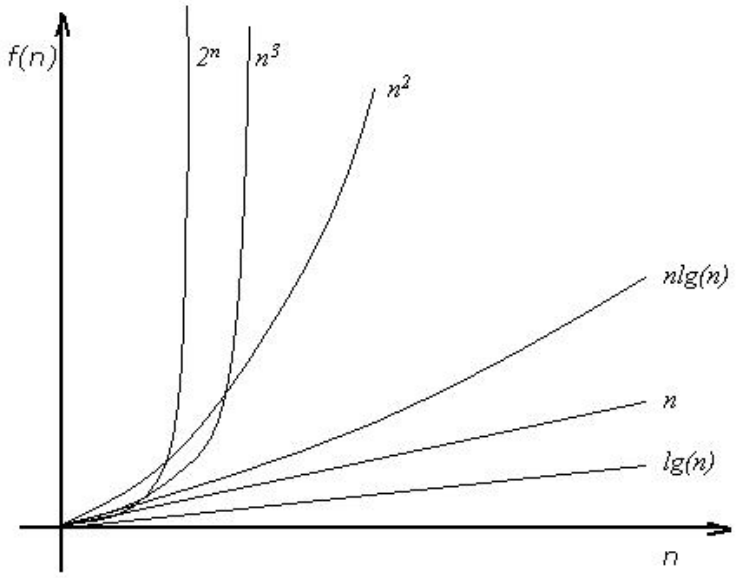
Comportamento Assintótico de Funções

Voltando ao custo de execução de um algoritmo, para pequenas instâncias (*n* pequeno), qualquer algoritmo custa pouco para ser executado, mesmo os ineficientes. Logo, a análise de algoritmos é realizada para valores grandes de *n*, isto é, estuda-se o comportamento assintótico das funções de custo, observe a comparação abaixo onde desprezamos constantes multiplicativas e termos de menor ordem:

	N = 100	**N=1000**	**N=10^4**	**N=10^6**	**N=10^9**
log(n)	2	3	4	6	9
n	100	1000	10000	10^6	10^9
nlog(n)	200	3000	40000	6×10^6	9×10^9
n^2	10^4	10^6	10^8	10^{12}	10^{18}
$100n^2+15n$	$1{,}0015\times10^6$	$1{,}00015\times10^8$	$\approx10^{10}$	$\approx10^{12}$	$\approx10^{20}$
2^n	$\approx1{,}26\times10^{30}$	$\approx1{,}07\times10^{304}$	?	?	?

O comportamento assintótico de *f(n)* representa o limite do comportamento do custo quando *n* cresce.

Observe o comportamento assintótico das funções $f(n) = 2^n, n^3, n^2, nlg(n), n \text{ e } lg(n)$



Knuth sugeriu uma notação para dominação assintótica pois a análise de algoritmos precisa de uma matemática que associa n^2 com: $(3/2)n^2$, $9999n^2$, $n^2/1000$, n^2+100n, etc. Esse tipo de matemática é "assintótico" (pois está interessado somente em valores grandes de n). Nessa matemática, as funções são classificadas em "ordens"; todas as funções de uma mesma ordem são "equivalentes". As cinco funções acima, por exemplo, pertencem à mesma ordem.

Ordem O

Antes de dar uma definição do conceito de ordem, convém restringir a atenção a funções assintoticamente não-negativas, ou seja, funções f tais que $f(n) \geq 0$ para todo n suficientemente grande. (Mais explicitamente: f é assintoticamente não-negativa se existe n_o tal que $f(n) \geq 0$ para todo n maior que n_o.)

Definição 1:

Dadas funções assintoticamente não-negativas f e g, dizemos que f está na *ordem* O de g, e escrevemos $f = O(g)$, se $0 \leq f(n) \leq c \cdot g(n)$ para algum c positivo e para todo n suficientemente grande.

Ou seja, existe um número positivo c e um número n_0 tais que $f(n) \leq c \cdot g(n)$ para todo n maior que n_0.

Exemplos:

i) se $f(n) \leq 999g(n)$ para todo $n \geq 100$ então $f = O(g)$. (a recíproca não é verdadeira)

ii) vamos mostrar que $n^2/2 - 3n = O(n^2)$.

queremos provar que, par algum par de constantes positivas c e n_0, temos:

$$0 \leq n^2/2 - 3n \leq cn^2, \text{ para todo } n \geq n_0$$

dividindo ambos os lados por n^2, procuramos encontrar c e n_0 tais que:

$$0 \leq 1/2 - 3/n \leq c, \text{ para todo } n \geq n_0$$

como a função $1/2 - 3/n$ nunca é maior que 1/2 e é positiva para $n \geq 7$, então $c = 1/2$ e $n_0 = 7$ certificam que $n^2/2 - 3n = O(n^2)$.

Obs.: esse não é o único par de constantes que garante a pertinência de

$n^2/2 - 3n = O(n^2)$; existem infinitos pares.

iii) suponha que $f(n) = (3/2)n^2 + (7/2)n - 4$ e que $g(n) = n^2$.

A tabela a seguir sugere $f(n) \leq 2g(n)$ para $n \geq 6$ e portanto temos $f(n) = O(g(n))$.

n	$f(n)$	$g(n)$
0	-4	0
1	1	1
2	9	4
3	20	9
4	34	16
5	51	25
6	71	36
7	94	49
8	120	64

É fácil verificar que, de fato, $f(n) = O(g(n))$ com um cálculo grosseiro:

$f(n) \leq 2n^2 + 4n^2 = 6n^2 = 6g(n)$ para todo n.

Ordem Ω

A expressão "f = *O*(g)" tem o mesmo sentido que "f $\leq$ g". Agora precisamos de um conceito como "f $\geq$ g".

Definição 2:

Dadas funções assintoticamente não-negativas f e g, dizemos que f *está na ordem Ω de* g, e escrevemos que $f = \Omega(g)$, se $f(n) \geq c \bullet g(n) \geq 0$ para algum *c* positivo e para todo *n* suficientemente grande.

Ou seja, existe um número positivo c e um número n_0 tais que $f(n) \geq c \cdot g(n)$ para todo n maior que n_0.

Exemplos:

i) se $f(n) \geq g(n)/100000$ para todo $n \geq 888$ então $f = \Omega(g)$. (a recíproca não é verdadeira!)

ii) vamos mostrar que $n^2/2 - 3n = \Omega(n^2)$.

queremos provar que, par algum par de constantes positivas c e n_0, temos:

$$0 \leq cn^2 \leq n^2/2 - 3n \quad \text{, para todo } n \geq n_0$$

dividindo ambos os lados por n^2, queremos encontrar c e n_0 tais que:

$$0 \leq c \leq 1/2 - 3/n, \text{ para todo } n \geq n_0$$

note que para $n \leq 6$ temos $1/2 - 3/n \leq 0$.

Como *c* deve ser uma constante positiva teremos $n_0 \geq 7$.

Más, $1/2 - 3/n \leq 0$ cresce monotonicamente para $n \geq 7$, então

$c = 1/2 - 3/7 = 1/14$ e $n_0 = 7$ certificam que $n^2/2 - 3n = O(n^2)$.

Qual a relação entre O e Ω ?

Não é difícil verificar que:

$$f = O(g) \quad \text{se e somente se} \quad g = \Omega(f).$$

Ordem Θ

Além dos conceitos com sentido de "f $\leq$ g" e de "f $\geq$ g", precisamos de um que seja como "f = g".

Definição 3:

Dizemos que *f* e *g* estão na *mesma ordem* Θ e escrevemos $f = \Theta(g)$ se $f = O(g)$ e $f = \Omega(g)$.

Ou seja, $f = \Theta(g)$ significa que $c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ para algum c_1 positivo, algum c_2 positivo, e para todo n suficientemente grande.

Exemplos:

i) as funções abaixo pertencem todas à ordem $\Theta(n^2)$:

$$n^2 \quad , \quad (3/2)n^2 \quad , \quad 9999n^2 \quad , \quad n^2/1000 \quad , \quad n^2+100n \; .$$

ii) finalmente, vamos mostrar que $n^2/2 - 3n = \Theta(n^2)$.

Já mostramos que $n^2/2 - 3n = O(n^2)$ e que $n^2/2 - 3n = \Omega(n^2)$ logo

$$n^2/2 - 3n = \Theta(n^2).$$

Más,

quais seriam as constantes positivas c_1, c_2 e n_0 que garantem que $n^2/2 - 3n = \square(n^2)$?

A resposta é simples:

os valores de c_1 e de c_2 são, respectivamente, os mesmos que garantiram que a função estava em $\Omega(n^2)$ e $O(n^2)$, ou seja, $c_1 = 1/14$ e $c_2 = 1/2$ o valor de n_0 deve ser o maior entre os dois valores de n_0 utilizados na demonstração de pertinência a $\Omega(n^2)$ e $O(n^2)$, ou seja, $n_0 = Max\{7,7\} = 7$.

Classes de comportamento assintótico

A maioria dos algoritmos possui um parâmetro que afeta o tempo de execução de forma mais significativa, usualmente o número de itens a ser processado. Este parâmetro pode ser o número de registros de um arquivo a ser ordenado, ou o número de nós de um grafo.

As principais classes de problemas possuem as funções de complexidade, descritas em notação Θ, como:

1. $f(n) = \Theta(1)$ estes algoritmos são ditos de complexidade constante. O uso do algoritmo independente do tamanho de n. Neste caso as instruções do algoritmo são executadas um número fixo de vezes.
2. $f(n) = \Theta(lgn)$ um algoritmo de complexidade Θ (lgn) é dito ter complexidade logarítmica. Este tempo de execução ocorre tipicamente em algoritmos que resolvem um problema transformando-o em problemas menores. Nestes casos, o tempo de execução pode ser considerado como sendo menor do que uma constante grande. Quando n é mil e a base do logaritmo é 2, $\log_2 n \approx 10$, quando n é um milhão $\log_2 n \approx 20$. Para dobrar o valor de lgn temos que considerar o quadrado de n. A base do logaritmo pouco muda estes valores: compare para n igual a um milhão temos $\log_{10} n \approx 6$.
3. $f(n) = \Theta(n)$ estes algoritmos tem complexidade linear, em geral um pequeno trabalho é realizado sobre cada elemento de entrada. Esta é a melhor situação possível para um algoritmo que tem que processar n elementos de entrada ou produzir n elementos de saída. Cada vez que n dobra de tamanho o tempo de execução dobra.
4. $f(n) = \Theta(nlgn)$ este tempo de execução ocorre tipicamente em algoritmos que resolvem um problema quebrando-o em problemas menores, resolvendo cada um deles independentemente e depois ajuntando as soluções. Quando n é um milhão e a base do logaritmo é 2, $nlgn$ é cerca de 20 milhões. Quando n é dois milhões, $nlgn$ é cerca de 42 milhões, pouco mais que o dobro.
5. $f(n) = \Theta(n^2)$ um algoritmo com esta complexidade é dito ter complexidade quadrática. Os algoritmos dessa ordem ocorrem quando os itens de dados são processados aos pares, muitas vezes em um laço dentro do outro. Quando n é mil, o número de operações é da ordem de 1 milhão. Sempre que n dobra o tempo de execução é multiplicado por 4. Algoritmos deste tipo são úteis para resolver problemas de tamanhos relativamente pequenos.
6. $f(n) = \Theta(n^3)$ um algoritmo de complexidade $\Theta(n^3)$ é dito ter complexidade cúbica. Algoritmos desta ordem de complexidade são úteis para resolver pequenos problemas. Quando n é cem, o número de operações é da ordem de 1 milhão. Sempre que n dobra o tempo de execução fica multiplicado por 8.
7. $f(n) = \Theta(2^n)$ um algoritmo de complexidade $\Theta(2^n)$ é dito ter complexidade exponencial. Algoritmos desta ordem de complexidade geralmente não são úteis sob o ponto de vista prático. Eles ocorrem na solução de problemas quando se usa força bruta para resolvê-los. Quando n é vinte, o tempo de execução é cerca de um milhão. Quando n dobra, o tempo de execução fica elevado ao quadrado.

Exercícios de fixação

1 - Quais das conjecturas abaixo são verdadeiras? Justifique.

a) $10n = O(n)$ b) $10n^2 = O(n)$ c) $10n^{55} = O(2^n)$

2 - É verdade que $n^2 + 200n + 300 = O(n^2)$? Justifique.

3 - É verdade que $n^2 - 200n - 300 = O(n)$? Justifique.

4 - Quais das conjecturas abaixo são verdadeiras? Justifique.

a) $(3/2)n^2 + (7/2)n - 4 = O(n)$ b) $(3/2)n^2 + (7/2)n - 4 = O(n^2)$

5 - É verdade que $n^3-999999n^2-1000000 = O(n^2)$? Justifique.

6 - Quais das conjecturas abaixo são verdadeiras? Justifique.

a) $\log_2 n = O(\log_3 n)$ b) $\log_3 n = O(\log_2 n)$

7 - É verdade que $\lceil \lg(n) \rceil = O(\lg(n))$? Justifique.

8 - Quais das conjecturas abaixo são verdadeiras? Justifique.

a) $\lfloor \lg(n) \rfloor = \Omega(\lg(n))$ b) $2^n = \Omega(3^n)$ c) $(3/2)n^2 + (7/2)n - 4 = \Theta(n^2)$

d) $9999\ n^2 = \Theta(n^2)$ e) $n^2/1000 - 999n = \Theta(n^2)$ f) $\log_2 n + 1 = \Theta(\log_{10} n)$

Algoritmos Iterativos

A análise de algoritmos utiliza técnicas de Matemática Discreta, envolvendo contagem ou enumeração dos elementos de um conjunto que possuam uma propriedade comum. Envolve a manipulação de somas, produtos, permutações, fatoriais, coeficientes binomiais, solução de equações de recorrência, entre outras. Não existe um conjunto completo de regras para analisar programas.

Aho, Hopcroft e Ullman[*] enumeram alguns princípios a serem seguidos:

1. O tempo de execução de um comando de atribuição, leitura ou escrita pode ser considerado constante. Há exceções para as linguagens que permitem a chamada de funções em comandos de atribuição, ou atribuições envolvendo arranjos muito grandes.
2. O tempo de execução de uma seqüência de comandos é determinado pelo maior tempo de execução de qualquer comando da seqüência.
3. O tempo de execução de um comando de decisão é composto pelo tempo de execução dos comandos executados dentro do comando condicional, mais o tempo de avaliar a condição, que é uma constante.
4. O tempo para executar uma estrutura de repetição (um looping) é a soma do tempo de execução do corpo do comando mais o tempo de avaliar a condição para conclusão, em geral constante, multiplicado pelo número de iterações da repetição.
5. Se o programa possui vários procedimentos não recursivos, o tempo de execução de cada procedimento deve ser computado separadamente um a um, iniciando com os procedimentos que não chamam outros procedimentos. Em seguida, devem ser avaliados os procedimentos que chamam os procedimentos que não chamavam outros procedimentos, agregando os tempos já computados dos procedimentos. Este processo é repetido até chegar ao programa principal.
6. Quando o programa possui procedimentos recursivos, para cada procedimento é associada uma função de complexidade *f(n)* desconhecida, onde *n* mede o tamanho dos argumentos para o procedimento.

Exemplos de aplicação desses princípios:

1) Classificação por Inserção:

		pior caso	
Inserção (A, n)		**contagem**	**consumo**
1	**para** j ← 2 **até** n **faça**	n	$O(n)$
2	x ← A[j]	(n-1)	$O(n)$
3	i ← j - 1	(n-1)	$O(n)$
4	**enquanto** i>0 **e** A[i]>x **faça**	n+(n-1)+...+2	$nO(n) = O(n^2)$
5	A[i+1] ← A[i]	(n-1)+(n-2)+...+1	$nO(n) = O(n^2)$
6	i ← i - 1	(n-1)+(n-2)+...+1	$nO(n) = O(n^2)$
7	A[i+1] ← x	(n-1)	$O(n)$
		$(3n^2+7n-8)/2$	$O(3n^2+4n) = O(n^2)$

[*] Aho, A.V., Hopcroft, J. E. e Ullman, J. D.; Data Structure and Algorithms; Addison-Wesley;1983.

Outra análise:

		melhor caso	
Inserção (A, n)		**contagem**	**consumo**
1	**para** j ← 2 **até** n **faça**	n	$\Omega(n)$
2	x ← A[j]	(n-1)	$\Omega(n)$
3	i ← j - 1	(n-1)	$\Omega(n)$
4	**enquanto** i>0 e A[i]>x **faça**	(n-1)	$\Omega(n)$
5	A[i+1] ← A[i]	0	o
6	i ← i - 1	0	o
7	A[i+1] ← x	(n-1)	$\Omega(n)$
		5n-4	$\Omega(5n) = \Omega(n)$

Em nossa análise consideramos o pior caso, onde os elementos do vetor estão em ordem inversa e serão colocados em ordem crescente, e o melhor caso – um vetor ordenado. Os resultados associam o pior caso a $O(n^2)$ e o melhor caso a $\Omega(n)$, correspondendo a um limite assintótico superior e outro inferior. Note que em termos de notação podemos ainda, afirmar que Inserção é $\Omega(n^2)$ no pior caso e $O(n)$ no melhor, para tanto retome a analise trocando as notações.

2) Intercalação de dois vetores:

Supondo A[e..m] e A[m+1..d] em ordem crescente; queremos colocar A[e..d] em ordem crescente:

e					m	m+1			d
11	33	33	55	55	77	22	44	66	88

Empregamos a função Intercala abaixo descrita.

Quanto tempo a função Intercala consome em função de n = d – e + 1?:

Intercala (A, e, m, d)		contagem	consumo
0	**crie** vetor B[e..d]	n=d-e+1	O(n)
1	**para** i ← e **até** m **faça**	2n+2	O(n)
2	B[i] ← A[i]		
3	**para** j ← m+1 **até** d **faça**		
4	B[d+m+1-j] ← A[j]		
5	i ← e	2	O(1)
6	j ← d		
7	**para** k ← e **até** d **faça**	n	nO(1) = O(n)
8	**se** B[i] ≤ B[i]	n	nO(1) = O(n)
9	**então** A[k] ← B[i]	2n	nO(1) = O(n)
10	i ← i+1		
11	**senão** A[k] ← B[j]		
12	j ← j-1		
	Total:	**7n+4**	$O(5n+1) = O(n)$

Observe que funcionará bem mesmo se e = m ou m = d. Para ver que este algoritmo é executado no tempo $O(n)$ observe que nas contagens de execuções das linhas não temos a consideração de melhor ou pior caso como observamos para o algoritmo Inserção.

Portanto, Intercala é $O(n)$ e também é $\Omega(n)$, ou conforme vimos na seção sobre notações, é de ordem $\Theta(n)$.

Uma vez mais, gostaríamos de lembrar que nossa análise procura observar o pior caso, e em algumas circunstâncias procura o melhor. E ainda, o caso médio será pouco citado, pois na maioria dos casos tem desempenho semelhante ao pior caso.

Retornando ao algoritmo Intercala, seria didático provar sua correção e funcionamento.

Execução:

- O algoritmo cria um vetor auxiliar B do tamanho da entrada ($d - e + 1 = n$) e armazena os sub-vetores A[e..m] e A[m+1..d] (em ordem inversa) no vetor auxiliar B.
- A partir da execução do laço *para...faça...* na linha 7 rearranja os elementos armazenando-os em A[e..d] já ordenados mantendo-se invariante:

> No início de cada iteração das linhas 8-12 o sub-vetor A[e..k-1] contém os *k-e* menores elementos dos sub-vetores em B[e..m] e B[m+1..d], em seqüência ordenada.

Prova:

Inicio: antes da primeira iteração desse laço, temos $k = e$, de forma que o sub-vetor *A[e..k-1]* está vazio. Esse sub-vetor contém os $k - e = 0$ menores elementos de *em B[e..m] e B[m+1..d],* e desde que $i = e$ e $j = d$, tanto *B[i]* quanto *B[j]* são os menores elementos dos sub-vetores copiados de *A[e..d].*

Manutenção: para ver que cada iteração mantém o laço invariante, vamos supor primeiro que $B[i] \leq B[j]$. Então *B[i]* é o menor elemento ainda não copiado de volta em A. Como *A[e..k-1]* contém os $k - e$ menores elementos, depois da linha 9 copiar *B[i]* em *A[k]*, o sub-vetor *A[e.. k]* conterá os $k - e + 1$ menores elementos. O incremento de k (na atualização do laço *para... faça...*) e de *i* (linha 10) restabelece o laço para a próxima iteração. Se, em vez disso, temos $B[j] \leq B[i]$, então as linhas 11 e 12 são executam a ação apropriada para manter o invariante.

Término: no término, $k = d+1$. Pelo invariante, o sub-vetor *A[e.. k-1]*, que é *A[e..d]*, contém os $k - e = d - e+1$ menores elementos de *B[e.. m]* e *B[m+1..d]* em seqüência ordenada, todos os *n* elementos foram copiados de volta ordenados.

3) Seqüência de soma máxima

Vamos apresentar mais uma análise de algoritmos ilustrando a solução de um problema em quatro algoritmos diferentes: o "*problema da seqüência de soma máxima*".

Através desses algoritmos podemos identificar a importância da busca de eficiência na solução de problemas. Começamos apresentando o enunciado deste problema:

Dada uma seqüência de números inteiros a_1, a_2,..., An determine a subseqüência a_i,..., a_j , com $0 \leq i \leq j \leq n$, que tem o valor máximo para a soma $\sum_{k=i}^{j} a_k$ *. Como exemplo considere a seqüência: -2, 11, -4, 13, -5, -2. A solução para essa entrada tem valor 20 com soma no segmento: a_2, ..., a_4..*

Se obtivermos um valor negativo na resposta, consideramos o valor nulo para simplificar o processamento.

Esse problema é interessante por apresentar soluções que apresentam variações drásticas no tempo de processamento como ilustra a tabela a seguir, com tempos anotados em segundos para os algoritmos, que apresentaremos a seguir, rodando em uma máquina real (que identificaremos como X):

algoritmo	Somax1	Somax2	Somax3	Somax4
tempo	$O(n^3)$	$O(n^2)$	$O(nlgn)$	$O(n)$
n = 10	0.00103	0.00045	0.00066	0.00034
n = 100	0.47015	0.01112	0.00486	0.00063
n = 1000	448.77	1.1233	0.05843	0.00333
n = 10000	427913.5	111.13	0.68631	0.03042
n = 100000	4×10^9	10994.2	8.0113	0.29832

Há muitas informações nessa tabela, para pequenas instâncias, os algoritmos rodam num piscar de olhos! Assim, podemos considerar que para pequenas entradas, não precisaremos de um grande projeto para resolver nosso problema. Ainda observando a tabela, verificamos que para grandes entradas, o Somax4 é a melhor escolha embora o Somax3 também seja razoável.

Devemos esclarecer que para os tempos acima, não estão computados os tempos para leitura dos dados. Por exemplo, o algoritmo Somax4 teria um tempo total com uma ordem grandeza a mais se considerássemos a leitura em disco de uma grande massa de dados. A leitura da entrada em geral é um "gargalo", porém uma vez feita à leitura, o problema pode ser resolvido rapidamente. Para algoritmos ineficientes isso não ocorre e então grandes recursos computacionais serão necessários, daí a importância de melhorar sempre a eficiência dos algoritmos.

Apresentando agora as soluções para o problema de soma máxima, vamos considerar (por conveniência) que a soma é *nula* para situações onde temos um *valor negativo* como resposta. Podemos nos convencer que o algoritmo Somax1, anotado a seguir, consome um tempo proporcional a $O(n^3)$ para apresentar uma resposta a uma dada instância. Esse tempo é devido inteiramente as linhas 5 e 6, que consistem de um comando que consome um tempo $O(1)$ dentro de três laços " *para ...* " aninhados.

```
Somax1(A, n)
1  max ← 0
2  para i ← 1 até n faça
3     para j ← i até n faça
4          aux  ←  0
5          para k ← i até  j faça
6               aux ← aux + A[k]
7               se aux > max
8                    então max ← aux;
9  devolve max
```

O laço na linha 2 é executado n vezes. O segundo laço tem tamanho $n - i + 1$, que poderá ser menor, ou de tamanho n. *Devemos assumir o pior caso* mesmo sabendo que isso poderá tornar o tempo total uma ordem de grandeza maior. O terceiro laço tem tamanho $j - i + 1$, que também devemos assumir tamanho n.

Assim temos o total $O(1 \cdot n \cdot n \cdot n) = O(n^3)$. A inicialização da variável, na linha 1, toma tempo $O(1)$ enquanto as linhas 7 e 8 tomam tempo $O(n^2)$, pois estão "dentro" dos laços mais externos.

Essa estimativa de tempo total nos apresenta uma resposta de $O(n^3)$ que como veremos a seguir é maior por um fator 6. Para uma analise precisa precisamos calcular a soma abaixo que relata quantas vezes a linha 6 é executada:

$$\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j}1$$

Realizamos o cálculo a partir da soma mais interna (em k): $\sum_{k=i}^{j}1 = j-i+1$

Em seguida, calculamos:

$$\sum_{j=i}^{n}(j-i+1) = \frac{(n-i+1)(n-i+2)}{2}$$

E finalmente:

$$\sum_{i=1}^{n}\frac{(n-i+1)(n-i+2)}{2} = \frac{1}{2}\sum_{i=1}^{n}i^2 - \left(n+\frac{3}{2}\right)\sum_{i=1}^{n}i + \frac{1}{2}\left(n^2+3n+2\right)\sum_{i=1}^{n}1 = \frac{n^3+3n^2+2n}{6}$$

Portanto obtemos para Somax1:

$f(n) = (1/6)n^3 + (1/2)n^2 + (1/3)n$ na ordem $O(n^3)$.

Uma melhora deste algoritmo seria evitar a complexidade cúbica através da remoção de um laço. Obviamente, nem sempre é possível, más nesse caso o algoritmo apresenta algumas computações desnecessárias como a computação nas linhas 5 e 6, observando que: $\sum_{k=i}^{j}a_k = a_j + \sum_{k=i}^{j-1}a_k$

E assim, podemos melhorar nosso algoritmo passando ao algoritmo Somax2, de ordem $O(n^2)$. A análise segue o mesmo raciocínio mostrado para Somax1.

```
Somax2(A, n)
1  max ← 0
2  para i ← 1 até n faça
3      aux ← 0
4      para j ← i até n faça
5          aux ← aux + A[j]
6          se aux > max
7              então max ← aux;
8  devolve max
```

A terceira solução para o problema do segmento de soma máxima é dada em um algoritmo recursivo: Somax3.

Veremos no futuro esse algoritmo, relativamente complicado, observando que se não fosse apresentada a solução que roda em tempo linear (Somax4), apresentada a seguir, teria encontrado não só a melhor resposta como também um exemplo do poder da recursão!

```
Somax4(A, n)
1 max ← aux ← 0
2 para i ← 1 até n faça
3     aux ← aux + A[i]
4     se aux > max
5         então max ← aux
6         senão se aux < 0
7                 então aux ← 0
8 devolve max
```

Uma vantagem extra para esse algoritmo é que ele faz somente uma passagem através dos dados, o que é muito importante, pois após ler e processar um elemento a_k não precisamos mais guardar esse valor.Portanto se o vetor está armazenado em um disco ou fita, pode ser lido seqüencialmente, e não precisamos armazenar qualquer parte dele na memória principal.

Assim em qualquer momento o algoritmo pode dar uma resposta sobre a soma máxima encontrada até aquele momento para os dados lidos. Algoritmos que fazem esse tipo de processamento são identificados como algoritmos "on-line", pois requerem um espaço constante e rodam em tempo linear, tão bom quanto é possível neste caso!

Exercícios de fixação

1 - Escreva e analise um algoritmo que, dada uma matriz quadrada de ordem n, encontre a *inversa* desta (utilize a regra dos cofatores para inverter a matriz).

2 - Problema: dado um inteiro x e um vetor de inteiros $A[e..d]$, em ordem crescente, encontre j tal que $A[j] \leq x < A[j+1]$. Escreva e analise (a correção e o consumo de tempo) de um algoritmo iterativo de uma busca: (i) "linear" (ii)"binária".

3 - Analise a correção e o consumo de tempo do seguinte algoritmo:

```
Inserção-2 (A, e, d)
1  m  ←  ⌊(e + d)/2⌋
2  Inserção-2 (A, e, m)
3  Inserção-2 (A, m+1, d)
4  Intercala(A, e, m, d)
```

4 - Considere o seguinte método para ordenar um vetor A[1 . . n]. Encontre o menor elemento em A e troque-o de posição com A[1]. Então procure o segundo menor elemento de A e troque-o de posição com A[2] (note que para isto basta procurar o menor elemento em A[2 . . n]). Repita este processo para os primeiros n − 1 elementos de A. Escreva o algoritmo para este procedimento que é conhecido como *Seleção* (selection sort). Escreva um invariante de laço que garante que o algoritmo está correto. Por que é suficiente considerar apenas os n−1 primeiros elementos? Faça uma análise de complexidade detalhada como feito em aula para o *Inserção.* Qual é o melhor e o pior caso?

5 - Considere duas matrizes A e B, quadradas de ordem n e, três algoritmos assim descritos:

- O algoritmo X deve fazer a soma das duas matrizes;
- O algoritmo Y deve fazer a multiplicação das duas matrizes;
- O algoritmo Z recebe um valor ***f*** (booleano) como parâmetro e, se ***f*** for verdadeiro, faz uma chamada ao algoritmo X. Caso contrário, executa o algoritmo Y.

Com estas definições, responda as seguintes questões:

a) Escreva os três algoritmos.

b) Analise o melhor caso, o caso médio e o pior caso dos algoritmos X e Y.

c) Analise o melhor e o pior caso do algoritmo Z.

d) Considerando a existência de uma quantidade ***p*** que representa a probabilidade de que a variável ***f*** seja verdadeira, responda qual é o caso médio do algoritmo Z.

Algoritmos Recursivos

Recursão é empregada em computação, tanto como estratégia de programação quanto na definição de expressões matemáticas e estruturas de dados. Como exemplo do uso de recursão em definições, podemos citar o fatorial e os números de Fibonacci. Algumas estruturas de dados também podem ser definidas de forma recursiva, exemplos são as listas generalizadas e as árvores.

Como estratégia de programação, a recursividade é uma ferramenta poderosa que, quando bem empregada, resulta em algoritmos elegantes e legíveis. Um algoritmo recursivo se caracteriza basicamente por conter chamadas a ele mesmo (recursividade direta), ou chamadas para procedimentos que por sua vez invocam o algoritmo original (recursividade indireta).

Há alguma resistência ao uso de técnicas recursivas na elaboração de algoritmos, motivada principalmente pelas dificuldades que surgem na depuração dos algoritmos e na análise de complexidade. As dificuldades na depuração de algoritmos recursivos são conseqüências das estruturas hierárquicas (como pilhas e árvores) que estão associadas às chamadas recursivas. Já a análise de algoritmos recursivos é mais complexa devido ao aparecimento de recorrências que surgem naturalmente da recursividade.

A idéia de qualquer algoritmo recursivo é simples: *Se o problema é pequeno, resolva-o diretamente, como puder. Se o problema é grande, reduza-o a um problema menor do mesmo tipo*. Ou seja, trabalhar com recursão envolve essencialmente o mesmo processo que com a indução matemática.

Vejamos alguns exemplos:

Na definição de funções matemáticas, expressões algébricas, tipos de dados, etc. e geralmente são constituídas de duas partes: *parte base* e *parte recursiva.*

i) Na função fatorial temos:

$Fat(0) = 1$ na parte base, e

$Fat(n) = n * Fat(n-1)$ na parte recursiva

Que resultaria:

```
Fat(n)
1   k ← 1
2   enquanto n > 0 faça
3           k ← k * n
4           n ← n-1
5   devolve k
```

ii) Definição dos números de Fibonacci:

$Fib(0) = 1$ e $Fib(1) = 1$ na parte base, e

$Fib(n) = Fib(n-1) + Fib(n-2),\ n>1$ na parte recursiva

Com o seguinte algoritmo:

```
Fib(n)
1   i ← 0
2   k ← j ← 1
3   enquanto n > 0 faça
4           k ← i + j
5           j ← k
6           i ← j
7           n ← n-1
8   devolve k
```

Uma estratégia de solução de problemas que utiliza recursividade é a técnica de "dividir e conquistar", que veremos futuramente. A idéia básica é dividir um problema em um conjunto de subproblemas menores, que são resolvidos independentemente para depois serem combinados a fim de gerar a solução final.

iii) *Soma recursiva*

Problema:

Escreva um algoritmo recursivo que calcule a soma dos elementos do vetor A[1..n].

1. Qual a *entrada* e qual a *saída* de nosso algoritmo? Nosso algoritmo *recebe* um número *n* e um vetor *A* e *devolve* um único número, que deve ser igual à soma dos elementos de A[1..n].
2. O problema faz sentido para qualquer n $\geq$ 0 e portanto nosso algoritmo aceita n = 0, Quando n = 0, a resposta correta é 0.
3. Feita essa discussão, e fácil escrever o algoritmo:

```
Soma(n, A)
1  se n = 0
2     então S ← 0
3     senão S ← Soma(n-1, A) + A[n]
4  devolve S
```

O algoritmo faz a soma "da esquerda para a direita". Como faríamos para somar "da direita para esquerda"? Para fazer isso é preciso *generalizar o problema*.

O problema generalizado tem dados *n, A e k ≥ 1* e pede a soma *A[k] + ... + A[n]*.

Eis o algoritmo:

```
Soma-dir_esq(k,n,A)
1  se k > n
2     então S ← 0
3     senão S ← A[k] + Soma-dir_esq(k+1,n,A)
4  devolve S
```

iv) Busca em vetor ordenado

Problema: *verificar se x é elemento de um vetor ordenado crescente A[e..d]*

A solução desse problema precisa encontrar *j* tal que *A[j] ≤ x < A[j+1]*, sinalizando com *j = -1* se não for encontrado.

Vejamos a solução com uma versão de "*busca linear*" e outra de "*busca binária*":

```
Busca-linear(x,A,e,d)
1  se e = d+1
2     então devolve -1
3     senão se x = A[d]
4                 então  devolve d
5              se x < A[d]
6                 então devolve Busca-linear(x,A,e,d-1)
7                 senão devolve -1
```

E agora

```
Busca-binária(x,A,e,d)
1  se e = d+1
2     então devolve -1
3     senão  m ← (e+d)/2
4               se x = A[m]
5                  então devolve m
6               se x < A[m]
7                  então devolve Busca-binária(x,A,e,m-1)
8                   senão devolve Busca-binária(x,A,m+1,d)
```

Para analisar a complexidade e correção de algoritmos que contém uma chamada recursiva, como nos exemplos, precisamos descrever o tempo de execução através de uma *recorrência*.

Recorrência é uma equação ou desigualdade que descreve uma função em termos de uma variação dela mesma. Como exemplo, considere a função abaixo:

$$f(n) = \begin{cases} c_1 & \text{se } n = 1 \\ f\left(\frac{n}{2}\right) + c_2 & \text{se } n > 1 \end{cases}$$

Na equação, o termo para *n = 1* é chamado de condição inicial e o termo para *n > 1* é denominado termo geral. Em algumas situações, podem aparecer mais que uma condição inicial e vários termos gerais em uma mesma recorrência.

Para a analise, precisamos *resolver* a recorrência que define uma função, digamos *f(n) que descreve a complexidade,* é obter uma "fórmula fechada" para *f(n)*. Uma "fórmula fechada" é uma expressão que envolve um número fixo de operações aritméticas. Assim, uma "fórmula fechada" não deve conter expressões da forma "1+2+3+...+n".

Resolver recorrências é uma arte, nem sempre fácil, e para tal empregamos os Métodos da Substituição, da Árvore de Recursão ou do Teorema Mestre.

No ***Método da Substituição***:

- devemos pressupor a forma da solução;
- usar indução matemática para mostrar que a solução funciona

Exemplos:

1) Seja a recorrência

T(1) = 1

T(n) = T(n-1)+3n+2 para n = 2,3,4,...

Que define a função T sobre inteiros positivos:

n	1	2	3	4	5	6
T(n)	1	9	20	34	51	71

Nossa hipótese será a função: $T(n) = (3/2)n^2+(7/2)n-4$

Verificando temos:

Se n=1 então $T(n) = 1 = 3/2+7/2-4$

Para $n \geq 2$ suponha que a fórmula está certa para n-1:

Então: $T(n) = T(n-1) + 3n + 2$

$=^{hip} (3/2)(n-1)^2+(7/2)(n-1)-4+3n+2$

$= (3/2)n^2-3n+3/2+(7/2)n-7/2-4+3n+2$

$= (3/2)n^2 + (7/2)n - 4$ *e está correta!*

Ficou parecido com um "truque", pois uma "hipótese" assim só tem essa eficiência se o "resolvedor" for muito experimentado! Podemos então pensar em tentar uma solução a partir dos dados obtidos com a equação, como vemos acima, supondo uma função como $An^2 + Bn + C$ para essa recorrência, e em seguida determinar os valores aproximados para os coeficientes *A*, *B* e *C*.

2) Considere outra recorrência

$G(1) = 1$

$G(n) = 2G(n/2)+7n+2$ para $n = 2,4,...,2^i, ...$

Nossa hipótese agora por ser uma função da forma *nlgn* ou n^2 , vejamos alguns valores de G(n):

n	1	2	4	8	16	32	64	128	256
G(n)	1	18	66	190	494	1214	2878	6654	15102
nlgn	0	2	8	24	64	160	384	896	2048
n^2	1	4	16	64	256	1024	4096	16384	65536

Faltando definir uma constante multiplicativa, observamos que nossa função para a fórmula fechada tem um termo dominante como *nlgn* ao invés de n^2 . Pesquisando empiricamente chegaremos à fórmula fechada:

$$G(n) = 7nlgn + 3n - 2 \qquad para\ n = 1, 2, 4, 8, 16, ...$$

Prova:

Se n=1 então $G(n) = 7(1lg1) + 3(1) - 2 = 1$

Para $n \geq 2$ temos

Então: $G(n) = 2G(n/2) + 7n + 2$

$=^{hip} 2(7(n/2)lg(n/2)+3(n/2)-2) + 7n + 2$

$= 7n(lgn-1) + 3n - 4 + 7n + 2$

$= 7nlgn - 7n + 3n - 2 + 7n$

$= 7nlgn + 3n - 2$ *e está correta!*

Embora o método da substituição possa fornecer uma prova de que uma solução para uma recorrência é correta, torna-se difícil apresentar uma boa suposição.

No ***Método da Árvore de recursão*** onde cada nó representa o custo de um único subproblema em algum lugar do conjunto de chamadas recursivas.

Vamos aplicar o método aos exemplos anteriores:

No exemplo-1 com a recorrência: $T(1) = 1$

$T(n) = T(n-1)+3n+2$ para $n = 2,3,4,...$

Expandindo T(n), temos a árvore:

T(n)	→	3n+2	→	3n+2	→	3n+2	→	...	→	3n+2	nível 0
		|		|		|				|	
		T(n-1)		3(n-1)+2		3(n-1)+2				3n-1	nivel 1
				|		|				|	
				T(n-2)		3(n-2)+2				3n-4	nivel 2
						|				|	
						T(n-3)				...	...
										|	
										8	
										|	
										T(1)	nivel n-1

Portanto:

$$T(n) = [3n+2] + [3(n-1) + 2] + [3(n-2) + 2] + [3(n-3)+2] + ... + 8 + T(1) =$$

$$= 3[n + (n-1) + (n-2) + ... + 2] + 2(n-1) + 1 = \mathbf{(3/2)n^2 + (7/2)n - 4}$$

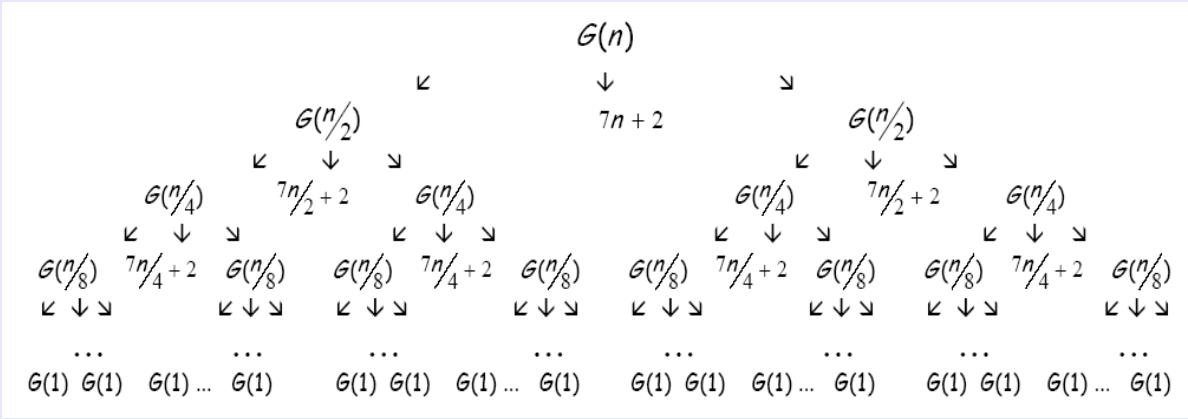
... e obtemos a fórmula fechada!

No exemplo - 2 a recorrência:

$G(1) = 1$

$G(n) = 2G(n/2)+7n+2$ para $n = 2,4,...,2^i, ...$

Apresenta a expansão de G(n):



Temos 1+lgn níveis, com as somas:

nível	soma no nível
0	7n+2
1	7n+4
2	7n+8
...	
k-1	$7n+2^k$
k	$2^k G(1)$

Sendo $n = 2^k$ temos,

$G(n) = 7n+2^1 + 7n+2^2 + ... + 7n + 2^{lgn} + 2^{lgn}\, G(1)$

$= 7nlgn + (2^1 + 2^2 + ... + 2^{lgn}) + 2^{lgn}$

$= 7nlgn + 2 \cdot 2^{lgn} - n$

$= 7nlgn + 3n - 2$

Obs: para resolver usamos o cálculo $x^0+...+x^k = (x^{k+1}-1)/(x-1)$

Os algoritmos recursivos

Retomando a análise de algoritmos, mais especificamente de algoritmos recursivos vamos rever a análise do algoritmo de classificação por inserção trabalhando em sua versão recursiva :

```
Inserção_Rec(A,e,d)
1  se e < d                                  1
2     então Inserção_Rec(A,e,d-1)            f(n-1)
3           x ← A[d]                         1
4           i ← d-1                          1
5  enquanto i ≥ e E A[i] > x faça            n
6        A[i+1] ← A[i]                       n-1
7        i ← i-1                             n-1
8  A[i+1] ← x                                1
```

Trabalhando novamente com a hipótese do pior caso, supomos que o algoritmo consome um tempo *f(n)*, que terá a seguinte expressão obtida com a soma dos tempos de execução de cada linha (anotados à direita das linhas de comando), supondo que em todas as operações temos o consumo de 1 unidade de tempo:

f(1) = 1 se n = 1,

f(n) = f(n-1) + 3n + 2 se temos n = 2, 3, 4, etc.

Precisamos tentar uma fórmula "fechada" como conseguimos na análise para a versão iterativa, e começamos reescrevendo a segunda expressão:

f(n) = f(n-1) + [3n + 2]

f(n) = f(n-2) + [3(n-1) + 2] + [3n + 2]

f(n) = f(n-3) + [3(n-2) + 2] + [3(n-1) + 2] + [3n + 2]

f(n) = f(n-4) + [3(n-3) + 2] + [3(n-2) + 2] + [3(n-1) + 2] + [3n + 2]

...

f(n) = f(2) + [3(3) + 2] + [3(4) + 2] + ... + [3(n-1) + 2] + [3n + 2]

f(n) = f(1) + [3(2) + 2] + [3(3) + 2] + ... + [3(n-1) + 2] + [3n + 2]

assim obtemos:

f(n) = 1 + 2(n-1) + 3[2 + 3+ ... + (n-1) + n], que resulta

f(n) = 1 + 2n – 2 + (3/2)[(n+2)(n-1)]

e finalmente:

f(n) = (3/2)n² + (7/2)n - 4 com consumo de tempo na ordem $O(n^2)$.

Para esse exemplo ficou fácil propor uma recorrência e encontrar a fórmula fechada, más nem sempre é assim.

Suponha que nosso algoritmo resulte em uma recorrência como no último exemplo da seção anterior.

Exemplo (n é potência de 2): G(1) = 1
G(n) = 2G(n/2)+7n+2 para n ≥ 2

... empregando o método da árvore obtivemos a solução exata: $G(n) = 7n\lg n + 3n - 2$

... más poderíamos ter obtido uma solução aproximada, em notação *O*: $G(n)=O(n\lg n)$

Em geral, é mais fácil de obter e provar uma solução aproximada que uma exata, vejamos como ficaria análise do algoritmo **Inserção_Rec** em notação *O*:

Com nosso objetivo de responder a questão:

*Quanto tempo a função **Inserção_Rec** consome en função de* $n = d - e + 1$?

Obtemos o quadro:

linha	consumo na linha
1	$O(1)$
2	$T(n-1)$
3	$O(1)$
4	$O(1)$
5	$O(n)$
6	$O(n)$
7	$O(n)$
8	$O(n1$
	$T(n) = T(n-1) + O(3n+4)$

Ou seja, $T(n) = T(n-1) + O(n)$

Existe uma função $F(n)$ em $O(n)$ tal que $T(n) = T(n-1) + F(n)$ para todo n suficientemente grande...

... ou ainda, existem a, $c > 0$ e $n_0 > 0$ tal que

$T(n_0 - 1) = a$ e $T(n) \leq T(n-1) + cn$ para todo $n \geq n_0$

A solução da recorrência: $T(n) = T(n-1) + O(n)$

Será: $T(n) = O(n^2)$

Vejamos outro exemplo de análise, retomando o "*problema da seqüência de soma máxima*", que apresentamos anteriormente para ilustrar a análise de complexidade de algoritmos iterativos e também a importância da busca de eficiência na solução de problemas.

Das quatro soluções, faltava a terceira solução para o problema do segmento de soma máxima que é dada em um algoritmo recursivo: Somax3.

O Somax3 faz uso da técnica de Divisão e Conquista, que abordaremos oportunamente, onde a idéia central é dividir o problema em dois outros subproblemas com metade do tamanho (no caso com entrada de n/2).

Cada subproblema é resolvido recursivamente então. Essa é a fase da "divisão", enquanto a "conquista" consiste em juntar as duas soluções dos subproblemas em uma solução global para o problema.

Em nosso exemplo, a subseqüência de soma máxima pode ocorrer em um dos seguintes locais:

- estar inteiramente contida na metade esquerda da entrada,
- ou inteiramente contida na metade direita da entrada,
- ou "cruzar" o limite que divide a entrada ao meio.

Os dois primeiros casos podem ser resolvidos recursivamente.

O último caso pode ser obtido encontrando a maior soma na primeira metade (à esquerda) que inclua o último elemento da 1ªmetade.

Em seguida encontra-se a maior soma da segunda metade (à direita) que inclui o primeiro elemento desta.

Finalizando, somam-se as duas somas encontradas obtendo "soma máxima que cruza a divisa" do último caso. Como exemplo considere a seguinte entrada:

1ª metade				2ª metade			
4	-3	5	-2	-1	2	6	2

A subsequência de soma máxima para a 1ªmetade é 6 (do elemento a_1 até a_3) e para a 2ªmetade é 8 (do elemento a_6 até a_7).

A máxima soma na 1ª metade que inclui o último elemento é igual a 4 (do elemento a_1 até a_4), e na 2ªmetade (que inclui o primeiro elemento da 2ª metade, o a_5) obtemos 7 (do elemento a_5 até a_7).

Assim a soma máxima que "cruza" a divisa tem valor: 4 + 7 = 11 (entre a_1 e a_7).

Podemos então, concluir que entre as três somas a última será a resposta para a sequência proposta de oito números: *soma max = 11*.

Apresentamos a seguir o algoritmo Somax3:

```
Somax3 (A, e, d)
 1  E_Lim ← D_Lim ← aux_E ← aux_D ← 0
 2  Lim ← (e + d)/2
 3  se e = d
 4      então se A[e] > 0
 5                então devolve A[e]
 6                senão devolve 0
 7   Esq  ←  Somax3(A, e, Lim)
 8   Dir  ←  Somax3(A, Lim+1, d)
 9   para i ← Lim até e  faça
10        aux_E ← aux_E + A[i]
11        se aux_E > E_Lim
12           então E_Lim ← aux_E
13   para i ← Lim+1 até d  faça
14        aux_D  ←  aux_D + A[i]
15        se aux_D > D_Lim
16           então D_Lim ← aux_D
17  devolve Máximo(Esq, Dir, E_Lim + D_Lim)
```

Nas chamadas recursivas sempre delimitamos o tamanho da entrada indicando os limites indicando os índices "e", esquerdo e "d ", direito, no inicio do processamento de Somax3 temos e = 1 e d = n.

Nas linhas 3 a 6, é manuseado *o caso base*, como na indução matemática. Se $e = d$, há um só elemento, e ele será a soma máxima se não for negativo.

Nas linhas 7 e 8, realizamos duas chamadas recursivas, que dividem o problema inicial em duas partes.

A partir da linha 9 (de 9 a 12 e de 13 a 16) processamos as somas da subseqüência que "cruza o limite" da divisão.

Em seguida com a função **Máximo** selecionamos a maior das somas como listamos anteriormente (à Esquerda, à Direita ou a soma dos máximos "incluindo a divisa").

Chegar à solução com o algoritmo Somax3 requer certamente um esforço muito maior que para os anteriores. Entretanto, soluções menores (ou mais breves) nem sempre implicam no melhor algoritmo. Como vimos na tabela de tempos, esta solução é consideravelmente mais rápida que as anteriores para entradas com grande número de elementos (*n*).

Para analisar o tempo de processamento, considere *f(n)* como o tempo para encontrar a soma máxima de uma seqüência de *n* números. Se *n* = 1, então o tempo de processamento é constante para executar das linhas 3 a 6, que podemos tomar como unitário, portanto: *f(1)* = 1.

Caso contrário, o programa realizará duas chamadas recursivas, os dois laços "*para...*" entre as linhas 9 a 16 e a seleção com **Máximo** na linha 17.

Os dois laços combinados, passam por cada um dos elementos de A[1..n], e há um trabalho constante dentro dos laços de forma que o tempo consumido é *O(n)*.

A inicialização de variáveis, na linha 1, e o trabalho executado na linhas 17, toma tempo constante e pode ser ignorado quando comparado com *O(n)*.

O restante do trabalho é feito nas linhas 7 e 8: nessas linhas são resolvidas duas subsequências de tamanho n/2 (assumindo *n* par). Portanto, essas linhas tomam um tempo *f(n/2)* em cada uma das linhas, com total de *2f(n/2)*.

O tempo total para o algoritmo será então *f(n) = 2f(n/2) + O(n)*. O que nos dá as equações:

$$f(1) = 1$$
$$f(n) = 2f(n/2) + O(n)$$

Para simplificar os cálculos, podemos considerar o termo *O(n)* como n, desde que *f(n)* seja expressa em termos de ordem *O( )*, isso não afetará a resposta e assim temos:

$$f(1) = 1$$
$$f(n) = 2f(n/2) + n$$

Então:

n = 1	f(1) = 1,
n = 2	f(2) = 2*f(2/2) + 2 = 4
n = 4	f(4) = 2*f(4/2) + 4 = 2*f(2) + 4 = 12
...	
$n= 2^k$	f(n) = n * (k+1) = nlgn +n = *O(nlgn)*.

Pela natureza recursiva dessa análise, observamos sua validade somente quando *n* é uma potência de 2, pois de outra forma teríamos um subproblema que não tem tamanho par e a equação não seria válida. Para *n* que não é potência de 2, a análise é mais complicada, porém o resultado não se altera.

Exercícios de fixação

1. Problema: *rearranjar um vetor A[e..d] em ordem decrescente.* Escreva uma versão recursiva do algoritmo "seleção" para o problema. Analise a correção e a eficiência do algoritmo (ou seja, encontre e prove as invariantes apropriadas).

2. Analise a correção e o consumo de tempo do seguinte algoritmo:

```
Inserção-2 (A, e, d)
1  se (d – e) > 0
2     então m ← ⌊(e + d)/2⌋
3           Inserção-2 (A, e, m)
4           Inserção-2 (A, m+1, d)
5           Intercala(A, e, m, d)
```

3. Problema: dado um inteiro x e um vetor de inteiros $A[e..d]$, em ordem crescente, encontre j tal que $A[j] \leq x < A[j+1]$. Escreva e analise (a correção e o consumo de tempo) de um algoritmo recursivo de uma busca: (i) "linear" (ii) "binária"

4. Mostre que a solução da recorrência:

$$T(1) = 1$$

$$T(n) = T(n-1) + 3n + 4 \quad \text{para } n \geq 2$$

Será: $T(n) \leq 4n^2$ para $n \geq 1$

5. Resolva as recorrências:

$T(1) = 1$ $T(2) = 1$ $T(n) = T(n-2) + 2n + 1$ para $n = 3, 4, 5, ..$	$T(1) = 1$ $T(n) = T(\lfloor n/2 \rfloor) + 1$ para $n = 2, 3, 4, 5, ...$