AULA 2

EFICIÊNCIA DE ALGORITMOS E PROGRAMAS

Medir a eficiência de um algoritmo ou programa significa tentar predizer os recursos necessários para seu funcionamento. O recurso que temos mais interesse neste momento é o tempo de execução embora a memória, a comunicação e o uso de portas lógicas também podem ser de interesse. Na análise de algoritmos e/ou programas alternativos para solução de um mesmo problema, aqueles mais eficientes de acordo com algum desses critérios são em geral escolhidos como melhores. Nesta aula faremos uma discussão inicial sobre eficiência de algoritmos e programas tendo como base as referências [1, 2, 8].

2.1 Algoritmos e programas

Nesta aula, usaremos os termos algoritmo e programa como sinônimos. Dessa forma, podemos dizer que um **algoritmo** ou **programa**, como já sabemos, é uma seqüência bem definida de passos descritos em uma linguagem de programação específica que transforma um conjunto de valores, chamado de **entrada**, e produz um conjunto de valores, chamado de **saída**. Assim, um algoritmo ou programa é também uma ferramenta para solucionar um **problema computacional** bem definido.

Suponha que temos um problema computacional bem definido, conhecido como o problema da busca, que certamente já nos deparamos antes. A descrição mais formal do problema é dada a seguir:

> Dado um número inteiro n, com $1 \leqslant n \leqslant 100$, um conjunto C de n números inteiros e um número inteiro x, verificar se x encontra-se no conjunto C.

O problema da busca é um problema computacional básico que surge em diversas aplicações práticas. A busca é uma operação básica em computação e, por isso, vários bons programas que a realizam foram desenvolvidos. A escolha do melhor programa para uma dada aplicação depende de alguns fatores como a quantidade de elementos no conjunto C e da complexidade da estrutura em que C está armazenado.

Se para qualquer entrada um programa pára com a resposta correta, então dizemos que o mesmo está **correto**. Assim, um programa correto **soluciona** o problema computacional associado. Um programa incorreto pode sequer parar, para alguma entrada, ou pode parar mas com uma resposta indesejada.

O programa 2.1 implementa uma busca de um número inteiro x em um conjunto de n números inteiros C armazenado na memória como um vetor. A busca se dá seqüencialmente no vetor C, da esquerda para direita, até que um elemento com índice i no vetor C contenha o valor x, quando o programa responde que o elemento foi encontrado, mostrando sua posição. Caso contrário, o vetor C é todo percorrido e o programa informa que o elemento x não se encontra no vetor C.

Programa 2.1: Busca de x em C.

```
#include <stdio.h>

#define MAX 100

/* Recebe um número inteiro n, com 1 <= n <= 100, um conjunto C de n
   números inteiros e um número inteiro x, e verifica se x está em C */
int main(void)
{
   int n, i, C[MAX], x;

   printf("Informe n: ");
   scanf("%d", &n);
   for (i = 0; i < n; i++) {
      printf("Informe um elemento: ");
      scanf("%d", &C[i]);
   }
   printf("Informe x: ");
   scanf("%d", &x);

   for (i = 0; i < n && C[i] != x; i++)
      ;

   if (i < n)
      printf("%d está na posição %d de C\n", x, i);
   else
      printf("%d não pertence ao conjunto C\n", x);

   return 0;
}
```

2.2 Análise de algoritmos e programas

Antes de analisar um algoritmo ou programa, devemos conhecer o modelo da tecnologia de computação usada na máquina em que implementamos o programa, para estabelecer os custos associados aos recursos que o programa usa. Na análise de algoritmos e programas, consideramos regularmente um modelo de computação genérico chamado de **máquina de acesso aleatório** (do inglês *random acess machine – RAM*) com um processador. Nesse modelo, as instruções são executadas uma após outra, sem concorrência. Modelos de computação paralela e distribuída, que usam concorrência de instruções, são modelos investigativos que vêm se tornando realidade recentemente. No entanto, nosso modelo para análise de algoritmos e programas não leva em conta essas premissas.

A análise de um programa pode ser uma tarefa desafiadora envolvendo diversas ferramentas matemáticas tais como combinatória discreta, teoria das probabilidades, álgebra e etc. Como o comportamento de um programa pode ser diferente para cada entrada possível, precisamos de uma maneira que nos possibilite resumir esse comportamento em fórmulas matemáticas simples e de fácil compreensão.

Mesmo com a convenção de um modelo fixo de computação para analisar um programa, ainda precisamos fazer muitas escolhas para decidir como expressar nossa análise. Um dos principais objetivos é encontrar uma forma de expressão abstrata que é simples de escrever e manipular, que mostra as características mais importantes das necessidades do programa e exclui os detalhes mais tediosos.

Não é difícil notar que a análise do programa 2.1 depende do número de elementos fornecidos na entrada. Isso significa que procurar um elemento x em um conjunto C com milhares de elementos certamente gasta mais tempo que procurá-lo em um conjunto C com apenas três elementos. Além disso, é importante também notar que o programa 2.1 gasta diferentes quantidades de tempo para buscar um elemento em conjuntos de mesmo tamanho, dependendo de como os elementos nesses conjuntos estão dispostos, isto é, da ordem como são fornecidos na entrada. Como é fácil perceber também, em geral o tempo gasto por um programa cresce com o tamanho da entrada e assim é comum descrever o tempo de execução de um programa como uma função do tamanho de sua entrada.

Por **tamanho da entrada** queremos dizer, quase sempre, o número de itens na entrada. Por exemplo, o vetor C com n números inteiros onde a busca de um elemento será realizada tem n itens ou elementos. Em outros casos, como na multiplicação de dois números inteiros, a melhor medida para o tamanho da entrada é o número de bits necessários para representar essa entrada na base binária. O **tempo de execução** de um programa sobre uma entrada particular é o número de operações primitivas, ou passos, executados por ele. Quanto mais independente da máquina é a definição de um passo, mais conveniente para análise de tempo dos algoritmos e programas.

Considere então que uma certa quantidade constante de tempo é necessária para executar cada linha de um programa. Uma linha pode gastar uma quantidade de tempo diferente de outra linha, mas consideramos que cada execução da i-ésima linha gasta tempo c_i, onde c_i é uma constante positiva.

Iniciamos a análise do programa 2.1 destacando que o aspecto mais importante para sua análise é o tempo gasto com a busca do elemento x no vetor C contendo n números inteiros, descrita entre a entrada de dados (leitura) e a saída (impressão dos resultados). As linhas restantes contêm diretivas de pré-processador, cabeçalho da função `main`, a própria entrada de dados, a própria saída e também a sentença `return` que finaliza a função `main`. É fato que a entrada de dados e a saída são, em geral, inerentes aos problemas computacionais e, além disso, sem elas não há sentido em se falar de processamento. Isso quer dizer que, como a entrada e a saída são inerentes ao programa, o que de fato damos mais importância na análise de um programa é no seu tempo gasto no processamento, isto é, na transformação dos dados de entrada nos dados de saída. Isto posto, vamos verificar a seguir o custo de cada linha no programa 2.1 e também o número de vezes que cada linha é executada. A tabela a seguir ilustra esses elementos.

	Custo	***Vezes***
`#include <stdio.h>`	c_1	1
`#define MAX 100`	c_2	1
`int main(void)`	c_3	1
`{`	0	1
`  int n, i, C[MAX], x;`	c_4	1
`  printf("Informe n:  ");`	c_5	1
`  scanf("%d", &n);`	c_6	1
`  for (i = 0; i < n; i++) {`	c_7	$n+1$
`    printf("Informe um elemento:  ");`	c_8	n
`    scanf("%d", &C[i]);`	c_9	n
`  }`	0	n
`  printf("Informe x:  ");`	c_{10}	1
`  scanf("%d", &x);`	c_{11}	1
`  for (i = 0; i < n && C[i] != x; i++)`	c_{12}	t_i
`    ;`	0	$t_i - 1$
`  if (i < n)`	c_{13}	1
`    printf("%d está na posição %d de C\n", x, i);`	c_{14}	1
`  else`	c_{15}	1
`    printf("%d não pertence ao conjunto C\n", x);`	c_{16}	1
`  return 0;`	c_{17}	1
`}`	0	1

O tempo de execução do programa 2.1 é dado pela soma dos tempos para cada sentença executada. Uma sentença que gasta c_i passos e é executada n vezes contribui com $c_i n$ no tempo de execução do programa. Para computar $T(n)$, o tempo de execução do programa 2.1, devemos somar os produtos das colunas ***Custo*** e ***Vezes***, obtendo:

$$T(n) = c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7(n+1) + c_8 n + c_9 n + \\ c_{10} + c_{11} + c_{12} t_i + c_{13} + c_{14} + c_{15} + c_{16} + c_{17} \, .$$

Observe que, mesmo para entradas de um mesmo tamanho, o tempo de execução de um programa pode depender de qual entrada desse tamanho é fornecida. Por exemplo, no programa 2.1, o melhor caso ocorre se o elemento x encontra-se na primeira posição do vetor C. Assim, $t_i = 1$ e o tempo de execução do melhor caso é dado por:

$$\begin{aligned} T(n) &= c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7(n+1) + c_8 n + c_9 n + \\ &\quad c_{10} + c_{11} + c_{12} + c_{13} + c_{14} + c_{15} + c_{16} + c_{17} \\ &= (c_7 + c_8 + c_9)n + \\ &\quad (c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_{10} + c_{11} + c_{12} + c_{13} + c_{14} + c_{15} + c_{16} + c_{17}) \, . \end{aligned}$$

Esse tempo de execução pode ser expresso como uma função linear $an + b$ para constantes a e b, que dependem dos custos c_i das sentenças do programa. Assim, o tempo de execução é dado por uma função linear em n. No entanto, observe que as constantes c_7, c_8 e c_9 que multiplicam n na fórmula de $T(n)$ são aquelas que tratam somente da entrada de dados. Evidentemente que para armazenar n números inteiros em um vetor devemos gastar tempo an, para alguma constante positiva a. Dessa forma, se consideramos apenas a constante c_{12} na fórmula, que trata propriamente da busca, o tempo de execução de melhor caso, quando $t_i = 1$, é

dado por:

$$T(n) = c_{12}t_i = c_{12} \, .$$

Por outro lado, se o elemento x não se encontra no conjunto C, temos então o pior caso do programa. Além de comparar i com n, comparamos também o elemento x com o elemento $C[i]$ para cada i, $0 \leqslant i \leqslant n-1$. Uma última comparação ainda é realizada quando i atinge o valor n. Assim, $t_i = n+1$ e o tempo de execução de pior caso é dado por:

$$T(n) = c_{12}t_i = c_{12}(n+1) = c_{12}n + c_{12} \, .$$

Esse tempo de execução de pior caso pode ser expresso como uma função linear $an+b$ para constantes a e b que dependem somente da constante c_{12}, responsável pelo trecho do programa que realiza o processamento.

Na análise do programa 2.1, estabelecemos os tempos de execução do melhor caso, quando encontramos o elemento procurado logo na primeira posição do vetor que representa o conjunto, e do pior caso, quando não encontramos o elemento no vetor. No entanto, estamos em geral interessados no **tempo de execução de pior caso** de um programa, isto é, o maior tempo de execução para qualquer entrada de tamanho n.

Como o tempo de execução de pior caso de um programa é um limitante superior para seu tempo de execução para qualquer entrada, temos então uma garantia que o programa nunca vai gastar mais tempo que esse estabelecido. Além disso, o pior caso ocorre muito freqüentemente nos programas em geral, como no caso do problema da busca.

2.2.1 Ordem de crescimento de funções matemáticas

Acabamos de usar algumas convenções que simplificam a análise do programa 2.1. A primeira abstração que fizemos foi ignorar o custo real de uma sentença do programa, usando as constantes c_i para representar esses custos. Depois, observamos que mesmo essas constantes nos dão mais detalhes do que realmente precisamos: o tempo de execução de pior caso do programa 2.1 é $an+b$, para constantes a e b que dependem dos custos c_i das sentenças. Dessa forma, ignoramos não apenas o custo real das sentenças mas também os custos abstratos c_i.

Na direção de realizar mais uma simplificação, estamos interessados na **taxa de crescimento**, ou **ordem de crescimento**, da função que descreve o tempo de execução de um algoritmo ou programa. Portanto, consideramos apenas o 'maior' termo da fórmula, como por exemplo an, já que os termos menores são relativamente insignificantes quando n é um número grande. Também ignoramos o coeficiente constante do maior termo, já que fatores constantes são menos significativos que a taxa de crescimento no cálculo da eficiência computacional para entradas grandes. Dessa forma, dizemos que o programa 2.1, por exemplo, tem tempo de execução de pior caso $O(n)$.

Usualmente, consideramos um programa mais eficiente que outro se seu tempo de execução de pior caso tem ordem de crescimento menor. Essa avaliação pode ser errônea para pequenas entradas mas, para entradas suficientemente grandes, um programa que tem tempo de execução de pior caso $O(n)$ executará mais rapidamente no pior caso que um programa que tem tempo de execução de pior caso $O(n^2)$.

Quando olhamos para entradas cujos tamanhos são grandes o suficiente para fazer com que somente a taxa de crescimento da função que descreve o tempo de execução de um programa

seja relevante, estamos estudando na verdade a **eficiência assintótica** de um algoritmo ou programa. Isto é, concentramo-nos em saber como o tempo de execução de um programa cresce com o tamanho da entrada *no limite*, quando o tamanho da entrada cresce ilimitadamente. Usualmente, um programa que é assintoticamente mais eficiente será a melhor escolha para todas as entradas, excluindo talvez algumas entradas pequenas.

As notações que usamos para descrever o tempo de execução assintótico de um programa são definidas em termos de funções matemáticas cujos domínios são o conjunto dos números naturais $\mathbb{N} = \{0, 1, 2, \ldots\}$. Essas notações são convenientes para descrever o tempo de execução de pior caso $T(n)$ que é usualmente definido sobre entradas de tamanhos inteiros.

No início desta seção estabelecemos que o tempo de execução de pior caso do programa 2.1 é $T(n) = O(n)$. Vamos definir formalmente o que essa notação significa. Para uma dada função $g(n)$, denotamos por $O(g(n))$ o *conjunto de funções*

$$O(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tais que } 0 \leqslant f(n) \leqslant cg(n) \text{ para todo } n \geqslant n_0\} .$$

A função $f(n)$ pertence ao conjunto $O(g(n))$ se existe uma constante positiva c tal que $f(n)$ "não seja maior que" $cg(n)$, para n suficientemente grande. Dessa forma, usamos a notação O para fornecer um limitante assintótico superior sobre uma função, dentro de um fator constante. Apesar de $O(g(n))$ ser um conjunto, escrevemos "$f(n) = O(g(n))$" para indicar que $f(n)$ é um elemento de $O(g(n))$, isto é, que $f(n) \in O(g(n))$.

A figura 2.1 mostra a intuição por trás da notação O. Para todos os valores de n à direita de n_0, o valor da função $f(n)$ está sobre ou abaixo do valor da função $g(n)$.

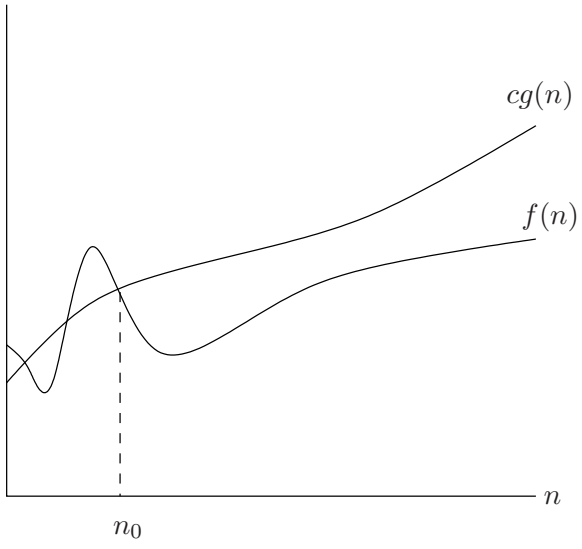

Figura 2.1: $f(n) = O(g(n))$.

Dessa forma, podemos dizer, por exemplo, que $4n + 1 = O(n)$. Isso porque existem constantes positivas c e n_0 tais que

$$4n + 1 \leqslant cn$$

para todo $n \geqslant n_0$. Tomando, por exemplo, $c = 5$ temos

$$\begin{aligned} 4n + 1 &\leqslant 5n \\ 1 &\leqslant n\,, \end{aligned}$$

ou seja, para $n_0 = 1$, a desigualdade $4n + 1 \leqslant 5n$ é satisfeita para todo $n \geqslant n_0$. Certamente, existem outras escolhas para as constantes c e n_0, mas o mais importante é que existe alguma escolha. Observe que as constantes dependem da função $4n + 1$. Uma função diferente que pertence a $O(n)$ provavelmente necessita de outras constantes.

A definição de $O(g(n))$ requer que toda função pertencente a $O(g(n))$ seja assintoticamente não-negativa, isto é, que $f(n)$ seja não-negativa sempre que n seja suficientemente grande. Conseqüentemente, a própria função $g(n)$ deve ser assintoticamente não-negativa, caso contrário o conjunto $O(g(n))$ é vazio. Portanto, vamos considerar que toda função usada na notação O é assintoticamente não-negativa.

A ordem de crescimento do tempo de execução de um programa ou algoritmo pode ser denotada através de outras notações assintóticas, tais como as notações Θ, Ω, o e ω. Essas notações são específicas para análise do tempo de execução de um programa através de outros pontos de vista. Nesta aula, ficaremos restritos apenas à notação O. Em uma disciplina mais avançada, como Análise de Algoritmos, esse estudo se aprofunda e atenção especial é dada a este assunto.

2.3 Análise da ordenação por trocas sucessivas

Vamos fazer a análise agora não de um programa, mas de uma função que já conhecemos bem e que soluciona o problema da ordenação. A função `trocas_sucessivas` faz a ordenação de um vetor v de n números inteiros fornecidos como parâmetros usando o método das trocas sucessivas ou o método da bolha.

```
/* Recebe um número inteiro n > 0 e um vetor v com n números intei-
   ros e rearranja o vetor v em ordem crescente de seus elementos */
void trocas_sucessivas(int n, int v[MAX])
{
   int i, j, aux;

   for (i = n-1; i > 0; i--)
      for (j = 0; j < i; j++)
         if (v[j] > v[j+1]) {
            aux = v[j];
            v[j] = v[j+1];
            v[j+1] = aux;
         }
}
```

Conforme já fizemos antes, a análise linha a linha da função `trocas_sucessivas` é descrita abaixo.

	Custo	*Vezes*
`void trocas_sucessivas(int` n`, int` v`[MAX]`	c_1	1
`{`	0	1
`  int i, j, aux;`	c_2	1
`  for (i =` n`-1; i > 0; i--)`	c_3	n
`    for (j = 0; j < i; j++)`	c_4	$\sum_{i=1}^{n-1}(i+1)$
`      if (`v`[j] >` v`[j+1]) {`	c_5	$\sum_{i=1}^{n-1} i$
`        aux =` v`[j];`	c_6	$\sum_{i=1}^{n-1} t_i$
`        `v`[j] =` v`[j+1];`	c_7	$\sum_{i=1}^{n-1} t_i$
`        `v`[j+1] = aux;`	c_8	$\sum_{i=1}^{n-1} t_i$
`      }`	0	$\sum_{i=1}^{n-1} i$
`}`	0	1

Podemos ir um passo adiante na simplificação da análise de um algoritmo ou programa se consideramos que cada linha tem custo de processamento 1. Na prática, isso não é bem verdade já que há sentenças mais custosas que outras. Por exemplo, o custo para executar uma sentença que contém a multiplicação de dois números de ponto flutuante de precisão dupla é maior que o custo de comparar dois números inteiros. No entanto, ainda assim vamos considerar que o custo para processar qualquer linha de um programa é constante e, mais ainda, que tem custo 1. Dessa forma, a coluna com rótulo ***Vezes*** nas análises acima representam então o custo de execução de cada linha do programa.

O melhor caso para a função `trocas_sucessivas` ocorre quando a seqüência de entrada com n números inteiros é fornecida em ordem crescente. Observe que, nesse caso, uma troca nunca ocorrerá. Ou seja, $t_i = 0$ para todo i. Então, o tempo de execução no melhor caso é dado pela seguinte expressão:

$$\begin{aligned}
T(n) &= n + \sum_{i=1}^{n-1}(i+1) + \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} t_i + \sum_{i=1}^{n-1} t_i + \sum_{i=1}^{n-1} t_i \\
&= n + 2\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 + 3\sum_{i=1}^{n-1} t_i \\
&= n + 2\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 + 3\sum_{i=1}^{n-1} 0 \\
&= n + 2\,\frac{n(n-1)}{2} + n - 1 \\
&= n^2 + n - 1\,.
\end{aligned}$$

Então, sabemos que o tempo de execução da função `trocas_sucessivas` é dado pela expressão $T(n) = n^2 + n - 1$. Para mostrar que $T(n) = O(n^2)$ devemos encontrar duas constantes positivas c e n_0 e mostrar que

$$n^2 + n - 1 \leqslant cn^2\,,$$

para todo $n \geqslant n_0$. Então, escolhendo $c = 2$ temos:

$$\begin{aligned}
n^2 + n - 1 &\leqslant 2n^2 \\
n - 1 &\leqslant n^2
\end{aligned}$$

ou seja,

$$n^2 - n + 1 \geqslant 0\,.$$

Observe então que a inequação $n^2 - n + 1 \geqslant 0$ é sempre válida para todo $n \geqslant 1$. Assim, escolhendo $c = 2$ e $n_0 = 1$, temos que

$$n^2 + n - 1 \leqslant cn^2$$

para todo $n \geqslant n_0$, onde $c = 2$ e $n_0 = 1$. Portanto, $T(n) = O(n^2)$, ou seja, o tempo de execução do melhor caso da função `trocas_sucessivas` é quadrático no tamanho da entrada.

Para definir a entrada que determina o pior caso para a função `trocas_sucessivas` devemos notar que quando o vetor contém os n elementos em ordem decrescente, o maior número possível de trocas é realizado. Assim, $t_i = i$ para todo i e o tempo de execução de pior caso é dado pela seguinte expressão:

$$\begin{aligned}
T(n) &= n + \sum_{i=1}^{n-1}(i+1) + \sum_{i=1}^{n-1} i + 3\sum_{i=1}^{n-1} t_i \\
&= n + \sum_{i=1}^{n-1}(i+1) + \sum_{i=1}^{n-1} i + 3\sum_{i=1}^{n-1} i \\
&= n + 5\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 \\
&= n + 5\,\frac{n(n-1)}{2} + n - 1 \\
&= \frac{5}{2}\,n^2 - \frac{5}{2}\,n + 2n - 1 \\
&= \frac{5}{2}\,n^2 - \frac{1}{2}\,n - 1\,.
\end{aligned}$$

Agora, para mostrar que o tempo de execução de pior caso $T(n) = O(n^2)$, escolhemos $c = 5/2$ e temos então que

$$\begin{aligned}
\frac{5}{2}\,n^2 - \frac{1}{2}\,n - 1 &\leqslant \frac{5}{2}\,n^2 \\
-\frac{1}{2}\,n - 1 &\leqslant 0
\end{aligned}$$

ou seja,

$$\frac{1}{2}\,n + 1 \geqslant 0$$

e, assim, a inequação $(1/2)n + 1 \geqslant 0$ para todo $n \geqslant 1$. Logo, escolhendo $c = 5/2$ e $n_0 = 1$ temos que

$$\frac{5}{2}\,n^2 - \frac{1}{2}\,n - 1 \leqslant cn^2$$

para todo $n \geqslant n_0$, onde $c = 5/2$ e $n_0 = 1$. Assim, $T(n) = O(n^2)$, ou seja, o tempo de execução do pior caso da função `trocas_sucessivas` é quadrático no tamanho da entrada. Note também que ambos os tempos de execução de melhor e de pior caso da função `trocas_sucessivas` têm o mesmo desempenho assintótico.

2.4 Moral da história

Esta seção é basicamente uma cópia traduzida da seção correspondente do livro de Cormen et. al [1] e descreve um breve resumo – a moral da história – sobre eficiência de algortimos e programas.

Um bom algoritmo ou programa é como uma faca afiada: faz o que é suposto fazer com a menor quantidade de esforço aplicada. Usar um programa errado para resolver um problema é como tentar cortar um bife com uma chave de fenda: podemos eventualmente obter um resultado digerível, mas gastaremos muito mais energia que o necessário e o resultado não deverá ser muito agradável esteticamente.

Programas alternativos projetados para resolver um mesmo problema em geral diferem dramaticamente em eficiência. Essas diferenças podem ser muito mais significativas que a diferença de tempos de execução em um supercomputador e em um computador pessoal. Como um exemplo, suponha que temos um supercomputador executando o programa que implementa o método da bolha de ordenação, com tempo de execução de pior caso $O(n^2)$, contra um pequeno computador pessoal executando o método da intercalação. Esse último método de ordenação tem tempo de execução de pior caso $O(n \log n)$ para qualquer entrada de tamanho n e será visto na aula 5. Suponha que cada um desses programas deve ordenar um vetor de um milhão de números. Suponha também que o supercomputador é capaz de executar 100 milhões de operações por segundo enquanto que o computador pessoal é capaz de executar apenas um milhão de operações por segundo. Para tornar a diferença ainda mais dramática, suponha que o mais habilidoso dos programadores do mundo codificou o método da bolha na linguagem de máquina do supercomputador e o programa usa $2n^2$ operações para ordenar n números inteiros. Por outro lado, o método da intercalação foi programado por um programador mediano usando uma linguagem de programação de alto nível com um compilador ineficiente e o código resultante gasta $50n \log n$ operações para ordenar os n números. Assim, para ordenar um milhão de números o supercomputador gasta

$$\frac{2 \cdot (10^6)^2 \text{ operações}}{10^8 \text{ operações/segundo}} = 20.000 \text{ segundos} \approx 5{,}56 \text{ horas},$$

enquanto que o computador pessoal gasta

$$\frac{50 \cdot 10^6 \log 10^6 \text{ operações}}{10^6 \text{ operações/segundo}} \approx 1.000 \text{ segundos} \approx 16{,}67 \text{ minutos}.$$

Usando um programa cujo tempo de execução tem menor taxa de crescimento, mesmo com um compilador pior, o computador pessoal é 20 vezes mais rápido que o supercomputador!

Esse exemplo mostra que os algoritmos e os programas, assim como os computadores, são uma **tecnologia**. O desempenho total do sistema depende da escolha de algoritmos e programas eficientes tanto quanto da escolha de computadores rápidos. Assim como rápidos avanços estão sendo feitos em outras tecnologias computacionais, eles estão sendo feitos em algoritmos e programas também.

Exercícios

2.1 Qual o menor valor de n tal que um programa com tempo de execução $100n^2$ é mais rápido que um programa cujo tempo de execução é 2^n, supondo que os programas foram implementados no mesmo computador?

2.2 Suponha que estamos comparando as implementações dos métodos de ordenação por trocas sucessivas e por intercalação em um mesmo computador. Para entradas de tamanho n, o método das trocas sucessivas gasta $8n^2$ passos enquanto que o método da intercalação gasta $64n \log n$ passos. Para quais valores de n o método das trocas sucessivas é melhor que o método da intercalação?

2.3 Expresse a função $n^3/1000 - 100n^2 - 100n + 3$ na notação O.

2.4 Para cada função $f(n)$ e tempo t na tabela abaixo determine o maior tamanho n de um problema que pode ser resolvido em tempo t, considerando que o programa soluciona o problema em $f(n)$ microssegundos.

	1 segundo	1 minuto	1 hora	1 dia	1 mês	1 ano	1 século
$\log n$							
$\sqrt{n}$							
n							
$n \log n$							
n^2							
n^3							
2^n							
$n!$							

2.5 É verdade que $2^{n+1} = O(2^n)$? E é verdade que $2^{2n} = O(2^n)$?

2.6 Suponha que você tenha algoritmos com os cinco tempos de execução listados abaixo. Quão mais lento cada um dos algoritmos fica quando você (*i*) duplica o tamanho da entrada, ou (*ii*) incrementa em uma unidade o tamanho da entrada?

(a) n^2

(b) n^3

(c) $100n^2$

(d) $n \log_2 n$

(e) 2^n

2.7 Suponha que você tenha algoritmos com os seis tempos de execução listados abaixo. Suponha que você tenha um computador capaz de executar 10^{10} operações por segundo e você precisa computar um resultado em no máximo uma hora de computação. Para cada um dos algoritmos, qual é o maior tamanho da entrada n para o qual você poderia receber um resultado em uma hora?

(a) n^2

(b) n^3

(c) $100n^2$

(d) $n \log_2 n$

(e) 2^n

(f) 2^{2^n}

2.8 Rearranje a seguinte lista de funções em ordem crescente de taxa de crescimento. Isto é, se a função $g(n)$ sucede imediatamente a função $f(n)$ na sua lista, então é verdade que $f(n) = O(g(n))$.

$$f_1(n) = n^{2.5}$$
$$f_2(n) = \sqrt{2n}$$
$$f_3(n) = n + 10$$
$$f_4(n) = 10^n$$
$$f_5(n) = 100^n$$
$$f_6(n) = n^2 \log_2 n$$

2.9 Considere o problema de computar o valor de um polinômio em um ponto. Dados n coeficientes $a_0, a_1, \ldots, a_{n-1}$ e um número real x, queremos computar $\sum_{i=0}^{n-1} a_i x^i$.

(a) Escreva um programa simples com tempo de execução de pior caso $O(n^2)$ para solucionar este problema.

(b) Escreva um programa com tempo de execução de pior caso $O(n)$ para solucionar este problema usando o método chamado de regra de Horner para reescrever o polinômio:

$$\sum_{i=1}^{n-1} a_i x^i = (\cdots (a_{n-1}x + a_{n-2})x + \cdots + a_1)x + a_0 \, .$$

2.10 Seja $A[0..n-1]$ um vetor de n números inteiros distintos dois a dois. Se $i < j$ e $A[i] > A[j]$ então o par (i, j) é chamado uma **inversão** de A.

(a) Liste as cinco inversões do vetor $A = \langle 2, 3, 8, 6, 1 \rangle$.

(b) Qual vetor com elementos no conjunto $\{1, 2, \ldots, n\}$ tem a maior quantidade de inversões? Quantas são?

(c) Escreva um programa que determine o número de inversões em qualquer permutação de n elementos em tempo de execução de pior caso $O(n \log n)$.