```
int main(int argc, char** argv)
{
  char c = 0;
  char* commands = "ads pq"; // key commands: "left,right,rotate,confirm,pause,quit"
  int speed = 2; // sets max moves per row
  int moves_to_go = 2;
  int full = 0; // whether board is full
  init(); // initialize board an tetrominoes


  cur =
  state[
  state[

  system
  printf

  while
    {
      dr

    // process user action
    c = getchar();   // get new action
    if (c == commands[0] && !intersect(cur, state[0]-1, state[1])) state[0]--; // move left
    if (c == commands[1] && !intersect(cur, state[0]+1, state[1])) state[0]++; // move right
    if (c == commands[2] && !intersect(cur->rotated, state[0], state[1])) cur = cur->rotated;
    if (c == commands[3]) moves_to_go=0;

    // scroll down
    if (!moves_to_go--)
      {
        if (intersect(cur,state[0],state[1]+1)) // if tetromino intersected with sth
          {
            cramp_tetromino();
            remove_complete_lines();
            cur = &tetrominoes[rand() % NUM_POSES];
            state[0] = (WIDTH - cur->width)/2;
```


Algoritmos de computador

Algoritmo

Conjunto de etapas que realizam uma tarefa, descritas com precisão suficiente para serem seguidas pelo agente executor

Um algoritmo é uma técnica de resolução de problemas independente de qualquer dispositivo (p.ex. computador digital)

Algoritmo

Conjunto de etapas que realizam uma tarefa, descritas com precisão suficiente para serem seguidas pelo agente executor

Um algoritmo é uma técnica de resolução de problemas independente de qualquer dispositivo (p.ex. computador digital)

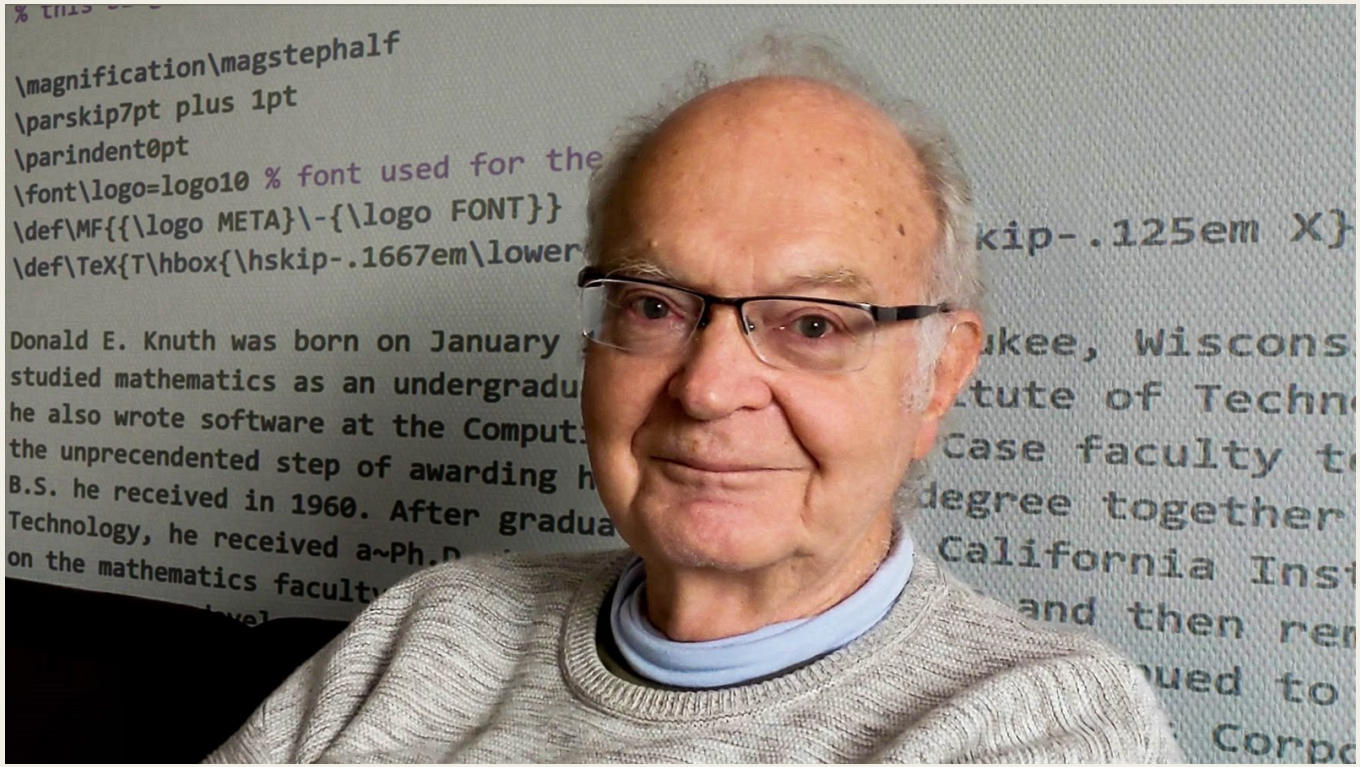
Algoritmo de computador

Conjunto de etapas são descritas em linguagem formal

Linguagem formal não contém ambiguidades

Escrevendo bons algoritmos

"Em vez de imaginar que nossa principal tarefa é instruir o computador sobre o que ele deve fazer, vamos imaginar que nossa principal tarefa é explicar a seres humanos o que queremos que o computador faça." – D. E. Knuth



Corretude

Um algoritmo é correto se ele faz o que deveria fazer

- Todo algoritmo implementa alguma função matemática f
- Um algoritmo correto mapeia toda entrada x no domínio de f no valor desejado $f(x)$

Um algoritmo é correto se ele faz o que deveria fazer

Decomposição em primos

Dado um inteiro, exibir seus fatores primos

Exemplo: Dado 12, exibir 2 e 3

Corretude

Um algoritmo é correto se ele faz o que deveria fazer

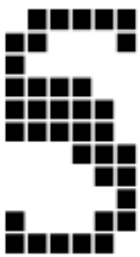
Reconhecimento de caracteres

Dada uma imagem de tamanho 11-por-6 pixels representando uma letra, dígito ou símbolo de pontuação, devolver caractere correspondente (char)

Um algoritmo é correto se ele faz o que deveria fazer

Reconhecimento de caracteres

Dada uma imagem de tamanho 11-por-6 pixels representando uma letra, dígito ou símbolo de pontuação, devolver caractere correspondente (char)

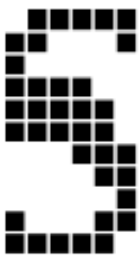


Dada imagem ao lado, devolver S

Um algoritmo é correto se ele faz o que deveria fazer

Reconhecimento de caracteres

Dada uma imagem de tamanho 11-por-6 pixels representando uma letra, dígito ou símbolo de pontuação, devolver caractere correspondente (char)



Dada imagem ao lado, devolver 5

Um algoritmo é correto se ele faz o que deveria fazer

Aplicativo de navegação

Dada uma posição inicial e uma posição final, algoritmo encontra a rota mais rápida de um ponto ao outro

Um algoritmo é correto se ele faz o que deveria fazer

Aplicativo de navegação

Dada uma posição inicial e uma posição final, algoritmo encontra a rota mais rápida de um ponto ao outro segundo o modelo de trânsito do algoritmo

Documentação

Para que possamos analisar a corretude de um algoritmo é necessário que sua especificação (o que se espera) seja bem documentada

- Documentação: o que um algoritmo faz (especificação funcional)
- Código: como o algoritmo faz (especificação operacional, implementação)
- Documentar não é (simplesmente) comentar; comentários demais podem tornam a explicação confusa (e certamente entediante)

Documentação

- Espécie de manual
- Deve conter: entrada e saída esperadas da função, pré-condições de entrada e casos particulares (não triviais)
- Explica passos principais da transformação da entrada na saída

Documentação: Bom exemplo

```c
/* Encontra o maior valor em um vetor de inteiros.
 * Argumentos:
 * - v: ponteiro para vetor de inteiros
 * - n: tamanho do vetor (> 0)
 * Retorno:
 * - int: maior valor em v[0..n-1]
 */
int maximo (int v[], int n) {
  int i, maior;
  maior = v[0]; /* valor inicial */
  for (i = 1; i < n; i++)
    if (v[i] > maior) maior = v[i];
  return maior;
}
```

Documentação: Bom exemplo

```c
/* Devolve o maior valor em vetor de inteiros v[0..n-1]. */
int maximo (int v[], int n) {
  int i, maior;
  maior = v[0]; /* valor inicial */
  for (i = 1; i < n; i++)
    if (v[i] > maior) maior = v[i];
  return maior;
}
```

Documentação: Mau exemplo

```c
/* Devolve o maior valor em um vetor v. */
int maximo (int v[], int n) {
  int i; /* usada para indexar elementos do vetor */
  int maior; /* usada para guardar maior valor */
  maior = v[0];
  for (i = 1; i < n; i++) /* para cada posição de v */
    /* se maior >= v[i], atualiza maior */
    if (v[i] > maior) maior = v[i];
  return maior; /* devolve maior valor visto */
}
```

Tamanho da entrada

inteiro positivo medindo tamanho do “problema”

Tamanho da entrada

inteiro positivo medindo tamanho do “problema”

```
1 /* Encontra maior valor em matriz M */
2 int maximo (int v[], int n) {
3   int i, maior;
4   maior = v[0];
5   for (i = 1; i < n; i++)
6     if (v[i] > maior) maior = v[i];
7   return maior;
8 }
```

tamanho da entrada: tamanho do vetor, n

Tamanho da entrada

inteiro positivo medindo tamanho do “problema”

```c
/* Conta as vogais em string s */
int contavogais (char *s) {
  int n = 0;
  while (*s != '\0') {
    s++; n++;
  }
  return n;
}
```

tamanho da entrada: tamanho da string `s`

Tamanho da entrada

inteiro positivo medindo tamanho do “problema”

```c
/* Encontra maior valor em matriz M */
int maximo (int M[][NCOL], int m, int n) {
  int i, j, maior;
  maior = M[0][0];
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      if (M[i][j] > maior) maior = M[i][j];
  return maior;
}
```

tamanho da entrada: quantidade de elementos da matriz, $m \times n$

Tamanho da entrada

inteiro positivo medindo tamanho do “problema”

```
1 /* Encontra maior valor na j-ésima coluna de matriz M */
2 int maximo (int M[][NCOL], int m, int n, int j) {
3   int i, maior;
4   maior = M[0][j];
5   for (i = 1; i < m; i++)
6       if (M[i][j] > maior) maior = M[i][j];
7   return maior;
8 }
```

tamanho da entrada:

Tamanho da entrada

inteiro positivo medindo tamanho do “problema”

```
1 /* Encontra maior valor na j-ésima coluna de matriz M */
2 int maximo (int M[][NCOL], int m, int n, int j) {
3   int i, maior;
4   maior = M[0][j];
5   for (i = 1; i < m; i++)
6       if (M[i][j] > maior) maior = M[i][j];
7   return maior;
8 }
```

tamanho da entrada: quantidade de linhas da matriz, m

Eficiência

Um algoritmo é eficiente se ele é executado rapidamente relativo ao tamanho da entrada

Eficiência

Um algoritmo é eficiente se ele é executado rapidamente relativo ao tamanho da entrada

Algoritmo A é mais eficiente que algoritmo B se existe um tamanho de entrada mínimo n_0 tal que A roda mais rápido que B em toda entrada maior que n_0

Eficiência

Um algoritmo é eficiente se ele é executado rapidamente relativo ao tamanho da entrada

Algoritmo A é mais eficiente que algoritmo B se existe um tamanho de entrada mínimo n_0 tal que A roda mais rápido que B em toda entrada maior que n_0

Tempo é medido por algum modelo de execução; tempos reais dependem de muitos fatores como arquitetura, sistema operacional, programas concomitantes etc.

Eficiência

Qual algoritmo de busca de palavra em texto é mais eficiente:

A) Busca trivial (testa casamento e desloca 1)

B) Deslocamento pela regra do caractere ruim

C) Deslocamento pela regra do sufixo bom

D) Algoritmo de Boyer-Moore

Eficiência: Algoritmo de Boyer-Moore

Modelo de execução: número de comparações de casamento

```
1  int busca_em_texto_bm (string p, string t)
2  {
3      int i, j, k, m, n, o = 0;    int T1[127], T2[CAP];
4     m = strlen(p); n = strlen(t);
5      preprocessa1 (p, m, T1); // Caractere ruim
6      preprocessa3 (p, m, T2); // Sufixo bom, caso 2
7      preprocessa2 (p, m, T2); // Sufixo bom, caso 1
8      k = 0;
9      while (k <= n-m) {
10         for (i = m-1; i >= 0 && p[i] == t[i+k] ; i--);
11         if (i == -1)  { o++; k += 1; }
12         else k += max(T2[i],T1[t[i+k]]-m+i+1);
13     }  return o;
14 }
15 }
```

Elegância

Um algoritmo é elegante se ele faz o que dele se espera de maneira clara e sucinta, sem desperdiçar recursos (tempo e memória)

Elegância

Um algoritmo é elegante se ele faz o que dele se espera de maneira clara e sucinta, sem desperdiçar recursos (tempo e memória)

Um algoritmo elegante não trata casos especiais desnecessariamente, não faz definições desnecessárias, não realiza construções complicadas sem ganho de eficiência

Qual é mais elegante?

```c
/* Recebe inteiros x e y e retorna o maior deles */
int maior1 (int x, int y) {
  int z;
  if (x==y) return x;
  if (x > y) z = x;
  if (z == x) return x;
  else return y;
}
```

```c
/* Recebe inteiros x e y e retorna o maior deles */
int maior2 (int x, int y) {
  if (x > y) return x;
  return y;
}
```

Qual é mais elegante?

```c
int f1 (int n, int v, int w) {
  int j, i = 0;
  while (i < n) {
    for (j = i; j < n; j++)
      if (v[j] == w[i]) return 1;
  }
  return 0;
}
```

```c
int f2 (int n, int v, int w) {
  int i, j;
  for (i = 0; i < n; i++)
    for (j = i; j < n; j++)
      if (v[j] == w[i]) return 1;
  return 0;
}
```

```
#define _ F-->00 || F-OO--;
long F=00,OO=00;
main(){F_OO();printf("%1.3f\n", 4.*-F/OO/OO);}F_OO()
{
                _-_-_-_
          _-_-_-_-_-_-_-_-_
       _-_-_-_-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
   _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
       _-_-_-_-_-_-_-_-_-_-_-_-_
          _-_-_-_-_-_-_-_-_
                _-_-_-_
}
```

Leiaute

Bons códigos são fáceis de ler

Bons códigos são fáceis de ler

```
1 int maximo(int v[], int n){
2   int i,j; j=v[0];
3   for (i=1;i<n;i++) if (v[i] > j) j=v[i];
4   return j; }
```

```
1 int maximo (int v[], int n) {
2   int i, j;
3   j = v[0];
4   for (i = 1; i < n; i++)
5       if (v[i] > j) j = v[i];
6   return j;
7 }
```

Leiaute

Bons códigos são fáceis de ler

```
1 int maior (int x, int y){
2    if (x > y)
3        return x;
4        return y; }
```

```
1 int maior (int x, int y) {
2   if (x > y) return x;
3   else       return y;
4 }
```

Invariantes de laço

- Asserção sobre estado do programa (valores de variáveis) satisfeita no início de cada passo de uma iteração (`for`, `while`)
- Explicam o processo iterativo e servem de base para provas formais

```
1  /* Recebe um vetor v de inteiros de tamanho n e
2   * retorna o maior valor entre v[0],...,v[n-1] */
3  int maximo(int v[], int n) {
4    int i, maior;
5    maior = v[0];
6    for (i = 1; i < n; i++)
7      /* invariante: maior >= v[j], j=0,...,i-1 */
8      if (v[i] > maior) maior = v[i];
9    return maior;
10 }
```

Invariantes de laço

```
1  /* Recebe um vetor v de inteiros de tamanho n e
2   * retorna o maior valor entre v[0],...,v[n-1] */
3  int maximo(int v[], int n) {
4    int i, maior;
5    maior = v[0];
6    for (i = 1; /* A */ i < n; i++)
7      if (v[i] > maior) maior = v[i];
8    return maior;
9  }
10 /* Em cada iteração, temos que maior >= v[j], j=0,...,i-1
11  * quando a execução se encontra no ponto A */
```

Exercício

Enuncie uma invariante para o código abaixo:

```
1 /* recebe inteiro positivo n e retorna o piso do logaritmo
      de n na base 2 */
2 int lg (int n) {
3   int i = 0; int x = n;
4   while (x /= 2) {
5     /*  x = piso (n/2^i)  */
6     i++;
7   }
8   return i;
9 }
```

n	15	16	31	32	63	64	127	128	255	256	511	512
$\lg(n)$	3	4	4	5	5	6	6	7	7	8	8	9

Corretude

A corretude de um algoritmo (e não de sua implementação) é em geral determinada por uma prova matemática

---

Corretude

A corretude de um algoritmo (e não de sua implementação) é em geral determinada por uma prova matemática

---

Existem muitas maneiras de provar uma afirmação matemática:

- prova por indução matemática
- prova por contradição
- prova por contraposição
- prova por construção
- ...

Indução matemática (finita)

Seja $A(n)$ uma afirmação sobre os números naturais

- Base: Prove que $A(0)$ é verdadeira
- Hipótese indutiva: Prove que se $A(n)$ é verdadeira então $A(n+1)$ também o é, para qualquer inteiro $n \geq 0$
- Conclusão: $A(n)$ é verdadeira para qualquer inteiro $n \geq 0$

Indução matemática: Exemplo

Prove que

$$A(n-1) : 1 + 2 + \cdots + n = n(n+1)/2$$

Indução matemática: Exemplo

Prove que

$$A(n-1) : 1 + 2 + \cdots + n = n(n+1)/2$$

- Base: $A(0) : 1 = 1(1+1)/2$

Indução matemática: Exemplo

Prove que

$$A(n-1) : 1 + 2 + \cdots + n = n(n+1)/2$$

- Base: $A(0) : 1 = 1(1+1)/2$
- Indução: Assuma que $A(n-1)$ é verdadeira para algum $n \geq 0$; então

$$\begin{aligned} A(n) : 1 + \cdots + n + (n+1) &= n(n+1)/2 + (n+1) \\ &= (n+1)(n/2+1) \\ &= (n+1)(n+2)/2 \end{aligned}$$

Exercício

Prove que

$$A(n-1): 1 + 2 + 2^2 + \cdots + 2^{n-1} = 2^n - 1$$

Exercício

Prove que

$$A(n-1): 1 + 2 + 2^2 + \cdots + 2^{n-1} = 2^n - 1$$

- Base: $A(0): 1 = 2^1 - 1$
- Indução: Assumindo que $A(n-1)$ é verdadeira para algum $n \geq 0$:

$$\begin{aligned} A(n): 1 + 2 + 2^2 + \cdots + 2^{n-1} + 2^n &= 2^n - 1 + 2^n \\ &= 2^{n+1} - 1 \end{aligned}$$

Prova de corretude por invariante de laço:

1. Mostrar que afirmação é verdadeira antes da primeira execução do laço
2. Mostrar que se afirmação for verdadeira antes de cada execução do laço então também é verdadeira antes da próxima iteração
3. Usar afirmação após última iteração do laço

Prove que o seguinte código é correto:

```
1  /* recebe inteiro positivo n e rertorna o piso do logaritmo
       de n na base 2 */
2  int lg (int n) {
3    int i = 0;
4    int x = n;
5    while (x /= 2) {
6      /* Invariante: x = piso (n/2^i) */
7      i++;
8    }
9    return i;
10 }
```

Prove que o seguinte código é correto:

```
1  /* recebe inteiro positivo n e rertorna o piso do logaritmo
       de n na base 2 */
2  int lg (int n) {
3    int i = 0;
4    int x = n;
5    while (x /= 2) {
6      /* Invariante: x = piso (n/2^i) */
7      i++;
8    }
9    return i;
10 }
```

- $A(i)$: no começo da `i`-ésima iteração do laço $\mathtt{x} = \lfloor \mathtt{n}/2^{\mathtt{i}} \rfloor$
- O laço termina quando $\lfloor \mathtt{n}/2^{\mathtt{i}} \rfloor < 2 \Rightarrow \mathtt{i} = \lfloor \log_2 \mathtt{n} \rfloor$

Prove que o seguinte algoritmo é correto:

```
1 /* Recebe um vetor de inteiros v[0..n-1] e um inteiro x e
      retorna o maior i t.q. v[i]=x ou -1 se nao existir */
2 int busca_linear (int v[], int n, int x) {
3  for (int i = 0; i < n; i++) {
4      /* invariante? */
5      if (v[i] == x) break;
6  }
7  return i;
8 }
```

Prove que o seguinte algoritmo é correto:

```
1 /* Recebe um vetor de inteiros v[0..n-1] e um inteiro x e
      retorna o maior i t.q. v[i]=x ou -1 se nao existir */
2 int busca_linear (int v[], int n, int x) {
3  for (int i = 0; i < n; i++) {
4      /* v[0..i-1] != x */
5      if (v[i] == x) break;
6  }
7  return i;
8 }
```

- $A(i)$: no início da `i`-ésima iteração `v[0..i-1] != x`
- Após `i`-ésima iteração, temos que `v[i]==x` (com fim de laço) ou `v[i] !=x`, portanto $A(i) \Rightarrow A(i+1)$

Para casa: Prove que o seguinte algoritmo é correto

```
1  /* Rearranja um vetor v[0..n-1] de inteiros para que os
2   * pares ocorram antes dos ímpares */
3   void Rearranja (int n, int v[]) {
4     int x, j = 0;
5     for (int i = 0; i < n; i++) /* Invariante */
6       if (v[i] % 2 == 0) {
7         x = v[i]; v[i] = v[j]; v[j] = x;
8         j++;
9       }
10  }
```

Invariante de laço:

1. `v[0..n-1]` é permutação do vetor original
2. `v[0..j-1]` contém apenas números pares;
3. `v[j..i-1]` contém apenas ímpares.

Resumo

Bons algoritmos são corretos, eficientes e elegantes; bons códigos são claros e bem documentados

- Corretude é (em geral) provada matematicamente (do algoritmo, não da implementação)
- Eficiência é analisada em termos de modelo de execução (mais nas próximas aulas)
- Elegância é em geral qualitativa e subjetiva; requer prática