Ano lectivo:	1998/99, semestre par
Docentes:	*Salvador Pinto Abreu* <http://home.di.uevora.pt/~spa/>, *Rui Tavares* <http://home.di.uevora.pt/~rt/>
Horas Teóricas Semanais:	2
Horas Práticas Semanais:	3 (laboratório)
Curso:	*Licenciatura em Engenharia Informática* <http://www.di.uevora.pt/ensino/LEI/>
Semestre do curso:	*4º* <http://www.di.uevora.pt/ensino/LEI/semestres.html#sem4>

Análise e Desenho de Algoritmos

Salvador Pinto Abreu <mailto:spa@di.uevora.pt>

Documento versão 3.5x, modificado em 2001.03.08.

Apontamentos para a disciplina de *Linguagens de Programação*, da Licenciatura em Engenharia Informática da Universidade de Évora.

Contents

1 O Sistema Operativo Unix: uma introdução

O sistema Unix foi desenvolvido nos Bell Labs em meados dos anos 70, com o intuito de permitir usar mini-computadores que estavam disponíveis. A sua característica mais saliente era de ter sido desenvolvido pelos seus principais utilizadores, ie. os programadores que o viriam a utilizar posteriormente.

1.1 Sistema de Ficheiros

Uma das características mais salientes do Unix é a integração de funcionalidades múltiplas no *sistema de ficheiros*, que estão tradicionalmente separadas noutros sistemas operativos. Um exemplo são as formas de aceder a periféricos (os **devices**).

O sistema de ficheiros está organizado como uma árvore, cuja raiz é denominada **root**. Todos os nós da árvore têm um *caminho* (ou **path**), especificado a partir da raiz. Aos nós interiores chama-se *directorias*. Cada nó tem um *nome* aplicável no contexto do seu *pai* (uma directoria). Para especificar um nó dá-se o seu caminho desde a raiz, em que se separam os nomes dos nós intermédios com o símbolo /. Por exemplo:

- **/** representa a raiz.
- **/home/spa** representa o nó **spa** filho do nó **home** (este último sendo forçosamente uma directoria, visto que é um nó interior), ele mesmo filho da raiz.
- **/usr/local/lib/nicks** representa um nó chamado **nicks**, que poderá ser ou não uma directoria.

Note-se que não é possível distinguir sintacticamente um ficheiro normal duma directoria: não há, no Unix, nada no nome dum nó que indique qual o seu tipo.

Cada directoria contém referências para ficheiros (ou outras directorias). Todas as directorias têm duas referências especiais:

- Uma directoria **.** que representa a própria directoria.
- Uma directoria **..** que representa a directoria em que esta está contida. Só no caso especial da raiz (/) é que esta representa a própria directoria.

1.2 Processos

No Unix, o sistema de ficheiro é usado para guardar de forma permanente dados e programas, sendo que alguns ficheiros representam programas *na sua forma latente*.

Os contextos em que os programas executam chamam-se *processos*. Um processo comporta, nomeadamente:

- Um executável activo. É o programa de que o processo se pode considerar como uma instância.
- Uma *directoria corrente*. É uma directoria, portanto um nó interior da árvore de directorias, que serve de base para todas as referências a ficheiros feitas pelo processo.
- Um conjunto de *descritores* de input/output. Estes são a forma como os processos comunicam com o exterior. Normalmente estão associados a ficheiros (sendo relativamente a estes últimos uma espécie de análogo de processo relativamente a programa) mas podem estar associados a dispositivos de comunicação genéricos, nomeadamente canais de comunicação com outros processos (as *pipes*). Um descritor tem associado a forma como pretende fazer operações (se input, output ou as duas).
- Um número, o **UID**, que indica qual o *utilizador* que activou o programa deste processo. Este número serve para dar ao sistema uma pista sobre que direitos relativamente a operações sobre ficheiros atribuir ao processo.

De entre os descritores dum processo, 3 são distinguidos no sentido em que vêm habitualmente abertos, são estes:

- Descritor 0: **stdin**. É um descritor que está implícitamente aberto para leitura.
- Descritor 1: **stdout**. É um descritor que está implícitamente aberto para escrita.
- Descritor 2: **stderr**. É um descritor que está implícitamente aberto para escrita, mas diferente do anterior no sentido em que se espera que esteja um humano a ver directamente o que fôr escrito para aqui. É típicamente usado pelos programas para assinalar erros.

1.3 Filtros

A forma fundamental como os processos comunicam no Unix é como *filtros*. Um filtro lê um stream como entrada e produz outro stream como saída, usando para tal os descritores 0 e 1 préviamente descritos.

Um filtro é normalmente preparado pelo *shell* (ver abaixo), em que o descritor 0 dum processo fica ligado ao descritor 1 doutro, sendo assim possível ao primeiro lêr directamente aquilo que o segundo escreve, sem ter de recorrer a ficheiros temporários.

1.4 Utilitários para Desenvolvimento

O sucesso do Unix deve-se em grande parte à multiplicidade e flexibilidade de pequenos utilitários (programas de sistema) que se combinam e integram bem. Iremos aqui descrever alguns destes utilitários.

1.4.1 O shell

O *shell* é o utilitário mais visível dum sistema Unix, pois é o que permite ao utilizador passear pela hierquia de ficheiros e executar outros programas. Chama-se desta maneira pois permite controlar o ambiente de execução dos outros programas.

Existem vários shells para o Unix, por exemplo o tradicional *Bourne Shell* (`/bin/sh`), o *C Shell* (`/bin/csh`) e o *bash* (que é uma versão melhorada inspirada no Bourne Shell, e parte do projecto GNU). Também correspondem ao conceito de shell algumas interfaces gráficas que permitam ao utilizador navegar pelo sistema de ficheiros e activar programas (o `kfm`, parte do K Desktop Environment, é um exemplo).

1.4.2 Editores

Um *editor de texto* é um programa que permite visualizar e modificar um ficheiro. Existem no Unix muitos editores diferentes, sendo os mais tradicionais (e vulgarmente disponíveis) o **ed** (um editor de linha), o **vi** (um editor visual, com uma filosofia derivada da do **ed**) e o **Emacs** (um editor programável e extensível, que integra um sistema de programação completo para uma linguagem de uso geral, bem adaptada à manipulação de texto).

1.4.3 Compiladores

Um *compilador* é um programa que serve para traduzir programas escritos numa linguagem para outra. O uso mais frequente será a tradução (compilação) duma linguagem de alto nível para outra, de baixo nível. Por exemplo, o programa **gcc** é um compilador da linguagem **C** para assembler da máquina subjacente. O **as** é um compilador de assembler para linguagem máquina (ficheiros `.o`, ou *objectos*), em que os programas resultantes ainda são susceptíveis de ser integrados com outros, antes de formar um programa directamente executável.

A documentação dos editores normalmente encontra-se disponível sob a forma de páginas de manual (ver 1.5.1). Uma excepção notável é o Emacs em que a maior parte da documentação é acedida dentro do próprio editor, nomeadamente pelo subsistema `Info`.

1.4.4 Make

O **make** é um programa que, mediante uma descrição, permite controlar *outros programas* com a intenção de combinar a sua execução por forma a obter um resultado final a partir dos seus componentes. Por exemplo, poderiamos usar o **make** para definir como é que se produzem ficheiros `.o` (objectos) a partir dos ficheirso `.c` (ficheiros fonte, neste caso programas na linguagem **C**).

O **make** usa um ficheiro de descrição de *objectivos*, que se chama habitualmente `Makefile`. Neste ficheiro estão descritas as formas como se podem realizar objectivos, tendo satisfeitos outros objectivos (ditos *intermédios*). Um exemplo de `Makefile` para produzir um executável `teste` a partir do resultado da compilação dos ficheiros `init.c`, `teste.c` e `util.c`, poderia ser o seguinte:

```
teste: init.o teste.o util.o
  cc -o teste init.o teste.o util.o

%.o: %.c
  cc -c $<
```

Em que se diz que:

1. O ficheiro `teste` pode ser criado com o comando `cc -o teste init.o teste.o util.o`, desde que os ficheiros `init.o`, `teste.o` e `util.o` estejam disponíveis.
2. Pode-se produzir um ficheiro `XPTO.o` a partir dum ficheiro correspondentemente chamado `XPTO.c` usando o comando `cc -c XPTO.c`.

O **make** é tradicionalmente usado para indicar quais os ficheiros que devem ser compilados para produzir um resultado. No entanto a sua utilidado não se limita à compilação, mas poderá ser posta em prática sempre que seja necessário produzir alguma coisa a partir de outra coisa.

1.5 A documentação on-line

O sistema Unix foi dos primeiros a dispor de documentação acessível dentro do próprio sistema. Esta existe sob várias formas, sendo as *manpages* a mais tradicional e generalizada.

1.5.1 O man

As *manpages* são documentos (habitualmente relativamente compactos, uma ou poucas páginas) que documentam um programa, uma função, um tipo ou outra entidade. As *manpages* estão estruturadas em secções, sendo usada a seguinte convenção:

- **1: Comandos para os utilizadores**. Encontram-se aqui informações sobre os comandos habitualmente usados por todos os utilizadores do sistema.
- **2: Chamadas ao sistema**. Esta secção documenta as system calls.
- **3: Funções de biblioteca**. Aqui encontra-se documentação sobre funções de biblioteca, para uso com qualquer linguagem, nomeadamente o C.
- **4: Ficheiros especiais**. É a documentação sobre os ficheiros que se encontram na directoria `/dev`, que servem primáriamente para aceder a periféricos.
- **5: Formatos de ficheiros**. É este o local onde se documentam os formatos dos diversos ficheiros existentes no sistema, nomeadamente ficheiros de configuração de programas e formatos de ficheiros (por exemplo executáveis).
- **6: Jogos**.
- **7: Diversos**.
- **8: Comandos para administração do sistema**. É uma secção como a **1**, mas para programas de administração dos sistema, por exemplo demónios.

O comando usado para visualizar as *manpages* é o `man` (de manual), e permite fazer algumas outras funções, como pesquisar no índice de manpages para determinar quais as páginas que versam um determinado assunto. O `man` tem uma página sobre ele próprio.

1.5.2 O info

O `info` foi dos primeiros sistemas de hipertexto desenvolvidos (o *hipertexto* é texto em que certas palavras são selecionáveis, estabelecendo ligações a outras partes do texto). Foi inicialmente implementado no `Emacs`, sendo particularmente utilizado nos programas do projecto GNU. Actualmente existem diversos visualizadores para o formato info que não recorrem ao Emacs.

1.5.3 Documentação em HTML

1.6 Exercícios

1. Usando um editor de texto, crie um ficheiro chamado `exercicio` em que relata o que fêz ontem.
2. Traduza o seu ficheiro todo para maiúsculas. Use para isso o comando `man -k translate` para saber qual o programa que poderá usar.

3. Generalize o procedimento anterior, fazendo uma Makefile, por forma que, a partir dum ficheiro `XPTO` possa obter dois ficheiros: `XPTO.uc` e `XPTO.lc`, em que o primeiro tem o seu conteúdo todo passado para maiúsculas, e o segundo todo para minúsculas. O nome `XPTO` não deve ser obrigatório, mas sim um padrão; por exemplo dever-se-á poder dizer `make pois.uc` para gerar `pois.uc` a partir dum ficheiro chamado `pois`.

2 A Linguagem C: uma introdução

A linguagem C vem na linha da linguagens BCPL, sendo ambas as linguagens ditas de *programação de sistemas*, embora de alto nível. O desenvolvimento do C está íntimamente ligado ao do sistema Unix, tendo havido uma grande influencia mútua: a principal aplicação da linguagem C foi o Unix em si, enquanto que todo o Unix tinha um cheiro a C, dado que este foi usado para implementar a quase totalidade do sistema.

2.1 Nomes e Visibilidade

No C iremos chamar aos símbolos *nomes*, estes designam sempre um *valor* e terão um *tipo* associado. Um nome tem uma *visibilidade*, ou *âmbito* (do inglês *scope*), que indica em que partes do programa a associação nome/tipo/valor é conhecida.

2.1.1 Blocos

O âmbito dum nome inserido num bloco é esse bloco. Um bloco é o fragmento de código fonte compreendido entre uma marca de início de bloco { e a marca de fim correspondente }.

2.1.2 Âmbito dum nome

O âmbito (*scope*) dos nomes é normalmente a *unidade de compilação*, ou *módulo*. Um módulo é o resultado da aplicação do pré-processador (ver 2.7) a um ficheiro *fonte*.

2.1.3 Definições e Declarações

Cada módulo é composto por uma sequência de *definições* e *declarações*. Uma declaração introduz um *nome* e associa-lhe um *tipo*. Uma definição é uma declaração que também associa um *valor* ao nome.

2.2 Instruções

Expressão. Deverá ter um efeito secundário (afectação ou chamada de função).

Sequenciação.

Instrução composta. (bloco)

Condicional. (if-else)

Ciclos. (while, do...while, for)

Condicional múltiplo. (switch)

Saída. (break; continue; return)

2.3 Tipos

Todo o nome tem um *tipo* associado. Os tipos no C não são tão elaborados como no ML (não exitem tipos paramétricos), mas podem ser encarados de forma semelhante.

2.3.1 Tipos básicos

Tipos inteiros: `int`, `char`, `short`, `long`, `unsigned`.

Tipos de vírgula flutuante: `float`, `double`.

2.3.2 Constructores de tipos

- Arrays
- Estruturas (records) (`struct`s e `union`s)
- Funções

2.4 Funções

Blocos com nome.

2.5 Aritmética de Endereços

2.5.1 Um problema: passagem de parâmetros

A linguagem C só dispõe dum mecanismo para passagem de parâmetros: a passagem *por valor*. Contraste-se isto com a situação noutras linguagens, nomeadamente o Pascal, em que existem (pelo menos) duas formas de passagem de parâmetros: por valor e *por referência*. Algumas outras linguagens também permitem a chamada passagem de parâmetros *por nome*, que é semelhante à abstracção do CAML ou aos blocos do Smalltalk.

Dado que o C não dispõe de mais nada que não a passagem por valor, seria impossível fazer funções como um `swap(a,b)`.

2.5.2 Os operadores & e * e o constructor de tipos *

Estes vêm solucionar o problema referido, permitindo que o programador *explícitamente* possa encarar o *endereço* duma expressão como um valor manipulável.

2.5.3 Operações sobre valores to tipo `T *`

Há essencialmente duas operações relativas a valores dum tipo `T *`: a obtenção do endereço (`&`), que toma um valor do tipo `T` e retorna o seu endereço (do tipo `T *`), e a *desreferenciação* (`*`) que toma um valor do tipo `T *` e retorna o valor do tipo `T` a que se refere o anterior.

2.5.4 Arrays

Declaração de array: `TIPO NOME[TAMANHO]`. `TAMANHO` é o número de elementos do tipo `TIPO` consecutivos que se pretende associar a `NOME`.

Se usarmos `NOME` numa expressão, este será encarado como sendo do tipo `TIPO *`, pelo que poderemos dizer coisas como:

```
int x[10];

pois (*x);
pois (*(x+3));
```

Os valores que serão passados à função `pois` são respectivamente o `int` que se encontra na primeira posição reservada para `x` e o `int` que está na *quarta* posição.

2.5.5 Arrays de `char` e literais de `char*`

Os strings no C não são mais que arrays de `char`. Para representar strings de comprimento variável (mas limitado a, digamos `N`), convenciona-se que é necessário manter espaço para `N+1` elementos. O string pretendido é constituído pelos elementos correspondentes ao texto, **terminado por um caracter zero**.

Sintaxe dos literais de `char *`. Convenções sobre os escapes dentro de strings.

2.5.6 Estruturas

No C o constructor de tipos para colecções heterogéneas é a *estrutura*, dada pela sintaxe do `struct`.

Os acessos a elementos duma estrutura, denominados *membros*, é feita com a notação `ESTRUTURA.MEMBRO`. Há também a notação de apontador, `APONT->MEMBRO`, equivalente a `(*APONT).MEMBRO`, cuja utilidade é justificada pelas precedências relativas de `*` e `.`.

2.6 Organização da Memória

No C há uma noção explícita da organização de memória, em 3 espaços disponíveis para ter os dados:

2.6.1 Espaço estático

É o espaço usado para as variáveis globais assim como para as locais (em termos de visibilidade) declaradas com o atributo `static`. Estas variáveis não são alteradas entre chamadas a funções e só são inicializadas uma vez.

Também é aqui que ficam as constantes (por exemplos os strings).

2.6.2 Espaço da pilha

São as variáveis típicas locais a uma função. São alocadas com uma disciplina de pilha (*stack*) e portanto têm como tempo de vida útil a duração da invocação duma função.

2.6.3 Espaço dinâmico

É o espaço dito de heap, em que o programa pode solicitar alocação (e desalocação) de memória. Este espaço é contíguo com o resto do programa, pelo que também se comporta como uma pilha (só pode crescer num sentido e deverá encolher em consequência).

Há no entanto na biblioteca standard do C funções que permitem alocar e libertar blocos de memória de tamanho arbitrário (especificado) neste espaço (`malloc`, `free`). Esta memória é gerida duma forma complexa pelo que deverá ser usada *com parcimónia*.

2.7 O Pré-processador

A linguagem de programação C já foi descrita. No entanto, é conveniente numa linguagem de programação de sistema permitir que algumas operações sejam efectuadas em tempo de compilação: tal pretensão requer um *processador de macros*, conceito próximo daquilo que se encontra nos assemblers.

2.7.1 Directivas do pré-processador

O CPP define várias directivas, dentre as quais se destacam:

- `#include "FICHEIRO"`
- `#include <FICHEIRO>`
- `#define NOME DEFINIÇÃO`

  Esta directiva é frequentemente usada para dar nomes a constantes numéricas. Tem um uso semelhante à declaração **const** do Pascal.
- `#define NOME(ARGS) DEFINIÇÃO`

  Esta directiva serve para definir *macros*, ie. sequências de instruções parametrizáveis.
- `#ifdef NOME` ... `#else` ... `#endif`

  Esta directiva é o fundamento da *compilação condicional*, pois irá fazer com que o compilador C só veja um dos lados (o do `#if` ou o do `#else`).

  Frequentemente usa-se esta directiva em conjunto com a `#define` para permitir que um ficheiro `.h` possa ser incluído mais de uma vez, sem efeitos nefastos.

Há mais directivas do CPP, mas por agora estas serão suficientes.

2.7.2 Organização em ficheiros

Os ficheiros `.c` contêm os módulos nos quais há definições, e os `.h` só devem conter *declarações*.

É habitual, quando se constrói um programa mais estruturado, fazer vários ficheiros com código C, em que a cada conjunto de funções relacionadas (por exemplo as que sirvam para implementar um determinado TAD) se faz corresponder um ficheiro `NOME.c` onde está o código das funções e outro ficheiro, `NOME.h`, onde se encontram as declarações todas, juntamente com quaisquer directivas do CPP que se pretenda facultar para facilitar o uso das funções.

2.8 Convenções no Unix

O programa principal: a função `main` e os seus argumentos.

Boa prática de modularização.

2.9 A biblioteca normalizada de I/O

A linguagem C não define nenhuma instrução para fazer input/output. Para tal recorre a bibliotecas fornecidas com o sistema.

Uma destas é o `stdio`, cujas declarações podem ser efectuadas incluindo o ficheiro `<stdio.h>`.

Algumas funções e características a destacar:

- `getchar()` e `putchar()`.
- A constante `EOF`.
- `printf`.
- `atoi`.
- `gets`.

2.10 Exercícios

1. Escreva um programa que determite o tamanho em bytes (usando o operador `sizeof`) dos tipos de base do compilador C que está a usar (pelo menos `char`, `short`, `int`, `long`, `long long`, `float`, `double`).
2. Faça um programa que, dada a seguinte declaração:

```
int vec[] = { 12, 31, 54, 12, 43, 65, 9, 19 };
```

   Calcule a soma dos elementos de `vec`, indique o número de elementos de `vec`, e calcule a média (com arredondamento) de `vec`.
3. Faça um programa que, dada a seguinte declaração:

```
int vec[][2] = { {1,2}, {3,4}, {5,6}, {7,8}, {0,0} };
```

   Calcule a soma dos produtos dos pares. O seu programa deverá terminar quando encontrar o par $\{0,0\}$.
4. Escreva um programa que leia todo o seu input (via a função `getchar()`) e escreva o resultado para o output (via a função `putchar()`), fazendo uma de duas operações, conforme o primeiro argumento da linha de comando seja respectivamente `"upper"` ou `"lower"`:
   - Converter todas as letras para maiúsculas
   - Converter todas as letras para minúsculas
5. Faça um programa em C que leia todo o seu input (via a função `getchar()`) e, quando terminar, escreva no output o número de caracteres lidos, o número de palavras vistas (define-se como palavra uma sequência de letras separadas por qualquer número de não-letras) e o número de linhas vistas (em que cada linha é terminada pelo caracter de código 12, `'\n'`).
6. Escreva um conjunto de funções C para fazer operações sobre strings, usando para tal um espaço alocado pelo utilizador. Convenciona-se que os strings são numerados de 1 em diante (preveja um número máximo de strings, digamos MAX_STRING, e defina-o como constante numérica usando o CPP). Os strings existentes são sempre referidos com este índice.

   Organize a sua biblioteca como dois ficheiros: um `"strings.h"` em que estão as *declarações* e outro `"strings.c"` em que estão as *definições*.

   A sua biblioteca deve implementar as funções seguintes:

- `void sb_init(char *espaço, int tamanho)`
- `int sb_insert(char *string)`
- `int sb_numstrings()`
- `char *sb_string(int indice)`
- `void sb_remove(int indice)`
- `int sb_search(char *substring)`
- `void sb_compact()`

Escreva um ou dois programas que usem a sua biblioteca. Estes programas deverão incluir "`string.h`" e ser linkados com `string.o`, o objecto correspondente à compilação do módulo de strings.

Sugestão: implemente uma agenda de contactos.

3 Análise de Complexidade

Para resolver um problema que tenha uma componente repetitiva, é importante poder caracterizar quais os recursos que irão ser dispendidos na resolução do mesmo. Estamos aqui a falar do tempo que demorará a resolver, assim como da quantidade de memória necessária para tornar a resolução possível.

Estas duas grandezas (tempo e espaço) constituem a medida do desempenho dum algoritmo. Ambas deverão ser apresentadas como função duma métrica do problema, que representará de alguma forma a dimensão do problema que o algoritmo pretende solucionar.

3.1 Contagem

A determinação da complexidade dum algoritmo passa pela contagem duma grandeza (por exemplo, as *operações*) que considerarmos relevante, em função dum parâmetro que se considere ser uma medida do tamanho do problema, que designaremos por N.

Há típicamente duas grandezas que se pretende analisar desta forma: o tempo gasto na resolução do problema ou *complexidade temporal*, designado por $T(N)$, e o espaço ocupado na resolução ou *complexidade espacial* designado por $E(N)$. Iremos focar a atenção mais na complexidade temporal.

Há várias formas de caracterizar o tempo gasto para resolver um problema:

- Pelo pior caso, em que se procura definir um limite superior da complexidade.
- Pelo caso médio, em que o que se pretende é uma caracterização da complexidade para situações *típicas*. Frequentemente a caracterização do caso médio é problemática e necessita duma análise estatística.

3.2 Notações Theta-grande, O-grande, Omega-grande

3.2.1 Caracterização

Para caracterizar a taxa de crescimento de algumas funções, utilizaremos 3 notações:

Diremos que $g(N)$ é $\Theta(f(N))$ se existirem constantes c_1 e c_2 tais que para N suficientemente grande, teremos $c_1 f(N) \leq g(N) \leq c_2 f(N)$.

Θ designa um limite duplo (superior e inferior).

O designa um limite superior, representa o *pior caso* dos algoritmos. É esta a forma habitualmente usada.

Ω designa um limite inferior.

Diremos que uma grandeza é $\Theta(f(N))$ se fôr simultâneamente $O(f(N))$ e $\Omega(f(N))$, embora esta situação seja raramente usada para caracterizar algoritmos: estes podem ter casos (correspondentes a situações de input) muito diferentes, pelo que se usa habitualmente só $O(f(N))$

3.2.2 Operações

Como fazer operações aritméticas sobre $O()$. Seja $T_1(n)$ e $T_2(n)$ os tempos de execuçao de dois fragmentos de programa P_1 e P_2, sendo $T_1(n) = O(f(n))$ e $T_2(n) = O(g(n))$, teremos:

- **Soma**: o tempo de execução da sequência P_1, P_2 é dado por $T_1(n) + T_2(n)$ que é $O(max(f(n), g(n)))$. *(Dominação pelo mais forte).*
- **Produto**: $T_1(n)T_2(n)$ é $O(f(n)g(n))$.

3.2.3 Procedimento

Por exemplo, tomemos o código:

```
  bubble (int a[], int n)
  {
    int i, j, t;

1   for (i=0; i<n-1; i++) {
2     for (j=n-1; j>i; j--) {
3       if (a[j-1] > a[j]) {
4         t = a[j-1];
5         a[j-1] = a[j];
6         a[j] = t;
        }
      }
    }
  }
```

Que ordena um vector. Cada passagem pelo ciclo interior (3-6) puxa o elemento mínimo para a frente.

A medida adequada para este programa é `n`, o número de elementos a ordenar. Observamos primeiro que cada afectação consome tempo constante, sendo portanto $O(1)$. Pela regra da soma, a complexidade do ciclo interior é $O(max(1,1,1)) = O(1)$.

Temos agora de considerar a instrução condicional e o ciclo interior. Como o `if` está incluído no `for`, iremos calcular isto de dentro para fora: para o `if`, o teste demora $O(1)$, e não sabemos se se deve ou não contabilizar as linhas 4-6. Como estamos a considerar o *pior caso*, vamos considerar que sim; desta forma as instruções nas linhas 3-6 demoram $T = O(1)$.

Para o ciclo interior (linhas 2-6), seguimos este procedimento geral para ciclos: o tempo consumido é a soma dos tempos consumidos em cada iteração, à qual se deve adicionar pelo menos $O(1)$ para o controle do ciclo. Neste caso, cada ciclo é $O(1)$ e o ciclo é executado $n-i$ vezes, pelo que a complexidade do ciclo é $O(n-i)O(1)$, ou seja $O(n-i)$.

Para o ciclo exterior (linhas 1-6), iremos seguir o esquema geral enunciado anteriormente: será portanto $\sum_{i=1}^{n-1}(n-i) = n(n-1)/2 = n^2/2 - n/2$, que é $O(n^2)$.

Assim pode-se afirmar que este programa ordena n elementos em tempo $O(n^2)$, o que é muito pior que outros algoritmos conhecidos, que o fazem em $T = O(nlogn)$.

Regras gerais:

1. As afectações, leituras e escritas terão $T = O(1)$.
2. O tempo duma sequência é dado pela regra da soma.
3. O tempo dum `if-then` é o tempo das instruções executadas mais o tempo de avaliação da condição, geralmente $O(1)$.

   No caso dum `if-then-else`, o *tempo das instruções* será considerado como o maior dos dois ramos.
4. O tempo de execução dum ciclo é dado pela soma dos tempos de execução de cada passagem pelo ciclo, ao qual deve ser somado o tempo do controle do ciclo.

   Em certas circunstâncias o número de passagens não pode ser determinado com precisão por inspecção do programa.

3.3 Chamadas a procedimentos e Relações de recorrência

Sempre que tenhamos uma chamada a um procedimento (recursivo ou não), temos de equacionar o tempo do procedimento como uma expressão, por exemplo $T(n) = c + T(n-1)$, ou outra semelhante.

3 formas de estabelecer a complexidade (O ou Θ):

1. O método da **substituição**, em que testamos uma hipótese colocada por nós próprios.

   Problema de "adivinhar" uma boa função, improvável no caso geral mas para o qual existem algumas heurísticas.
2. O método da **iteração** em que se converte a recorrência num somatório, que é depois resolvido usando métodos algébricos.

   Uma forma de ajudar a não se perder é fazer uma *árvore de recursão*.
3. O método **mestre** que indica valores para recorrências da forma $T(n) = aT(n/b) + f(n)$ onde $a \geq 1$ e $b > 1$.

   3 casos, em função de $f(n)$, e seja $\varepsilon > 0$ uma constante:

   (a) Se $f(n) = O(n^{log_b a - \varepsilon})$ teremos $T(n) = \Theta(n^{log_b a})$.

   (b) Se $f(n) = \Theta(n^{log_b a})$ teremos $T(n) = \Theta(n^{log_b a} log_2 n)$.

   (c) Se $f(n) = \Omega(n^{log_b a + \varepsilon})$ e que $af(n/b) \leq cf(n)$ para uma constante $c < 1$ e n suficientemente grande, teremos $T(n) = \Theta(f(n))$.

   Este método não cobre todas as situações, mas caso sirva permite obter directamente uma expressão para a complexidade.

Exemplo com a funcão factorial, sob a forma recursiva ($O(n)$), e com o mergesort ($O(NlogN)$).

3.4 Complexidade assimptótica e observada

A complexidade dum programa pode ser caracterizada duma forma muito precisa se tivermos dados para cada instrução individual. No entanto, esta determinação habitualmente só é possível *experimentalmente*.

Importância da complexidade assimptótica quando a velocidade dos sistemas aumenta: aproximação do comportamento assimptótico, logo maior relevância dum algoritmo com melhor desempenho assimptótico.

3.5 Computabilidade e complexidade: a classe P e a classe NP

A classe de programas cuja complexidade é *polinomial* chama-se P. Chamam-se a estes problemas *tratáveis* pois é viável construir para estes um algoritmo que encontre uma solução em tempo polinomial, por oposição a um programa que gere todas as soluções e determine qual a válida (este teria complexidade exponencial).

A classe NP é a dos programas com complexidade *não-determinísticamente polinomial*. Esta última inclui a subclasse NP-completa, sendo de notar que os problemas na classe NP-completa têm complexidade equivalente entre si (daí a classe), mas não foi demonstrada a não-existência de algoritmos polinomiais.

3.6 Exercícios

1. Considere o código para a pesquisa binária (vector ordenado). Caracterize a complexidade das operações de **inserção**, **remoção** e **busca**.
2. Os números de fibonacci são dados por $fib(N) = fib(N-1) + fib(N-2)$. A base da recursão é dada por $fib(1) = fib(2) = 1$.
   (a) Escreva uma implementação ingénua directa deste algoritmo, chame-lhe `int fib(int a)`.
   (b) Faça a análise do *tempo de execução* da sua implementação. Caracterize a complexidade temporal.
   (c) Assumindo que cada activação da função `fib()` requer K bytes para armazenar as variáveis locais e outra informação, caracterize a complexidade espacial da função `fib()`.
   (d) Sugira formas como se poderia melhorar o desempenho da função `fib()`. Procure fazer com que a melhoria incida na complexidade temporal para uma sequência de *K* invocações da função.

4 Estruturas de dados para Dicionários

O tipo abstracto de dados (TAD) *Dicionário* (às vezes referido como tabela de símbolos) é um dos mais universais na sua aplicabilidade, sendo dos que mais variações tem. Iremos explorar algumas, para além das que já foram apresentadas na disciplina de Estruturas de Dados.

Frequentemente é necessário representar e manipular conjuntos. As operações em questão poderão variar conforme o problema.

Operações primárias sobre o TAD *Dicionário*:

- **insere**: Inserção dum par (*chave*, *valor*).
- **membro**: Localização dum par (*chave*, *valor*) dada a *chave*.
- **remove**: Remoção dum par (*chave*, *valor*) dada a *chave*.

Adicionalmente, poderemos ainda especificar uma ou mais das seguintes operações:

- **novo**: Criação duma nova instância dum dicionário.
- **travessia**: Travessia sequencial de todos os pares (*chave*, *valor*), possívelmente ordenados por *chave*.
- **próximo** e **anterior**: Obtenção do próximo (ou anterior) par (*chave*, *valor*) dada uma *chave*. É assumida uma ordem total sobre as chaves.
- **min**, **max**: retorna o par com a *chave* mais baixa (ou mais alta, respectivamente).

Poderemos exprimir os algoritmos exclusivamente em termos da *chave*, pois o *valor* é simplesmente uma informação periférica, uma espécie de anotação associada à chave, que não contribui para as operações.

As estruturas de dados que vamos abordar são representativas de duas situações distintas: os dicionários ditos ”internos”, que são adequados para uma representação em memória principal e os dicionários ditos ”externos” em que uma representação em memória secundária é natural. Esta característica não é intrínseca dos algoritmos: reflecte antes o custo esperado do acesso a um elemento.

4.1 Red-Black Trees

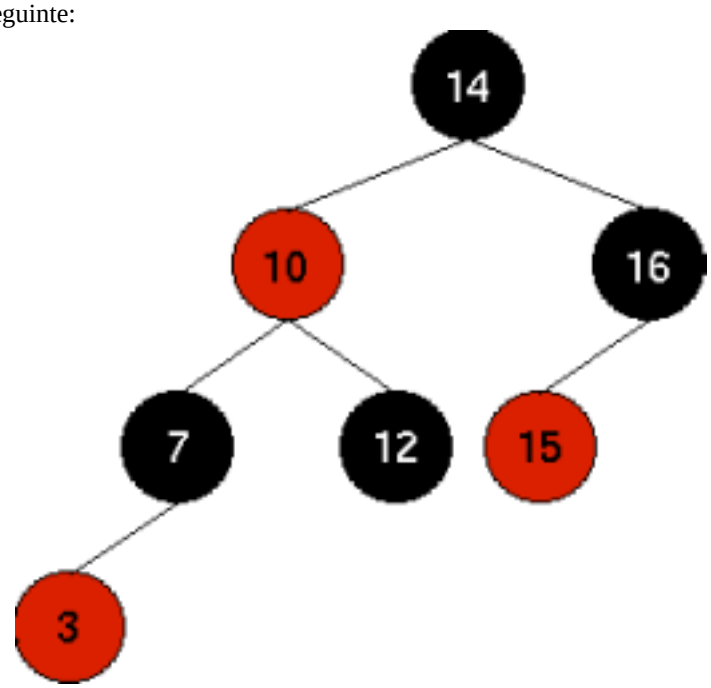
Iremos ver estruturas de dados susceptíveis de serem utilizadas para dicionários em que a quantidade de informação (e outras características, como a persistência) tornam viável uma organização totalmente em memória, i.e. não é necessário efectuar operaçoes de I/O para aceder à informação.

Árvores semi-equilibradas: em vez de ter árvores totalmente equilibradas (como é o caso nas árvores AVL), definem-se critérios que garantam que a árvore nunca fica ”muito” desequibrada. A estratégia será ”pintar” cada nó a vermelho ou preto e colocar restrições sobre as cores dos nós por forma a não haver caminhos de nenhum nó para as folhas cujo comprimento seja mais que o dobro de qualquer outro (partindo do mesmo nó inicial). O algoritmo resultante é mais simples que o das árvores AVL já nossas conhecidas.

Um nó será composto pelos seguintes campos: *cor* (vermelho ou preto), *valor* (os ”dados” do nó), *fe* (filho esquerdo, outro nó), *fd* (filho direito, outro nó), *pai* (outro nó). Se um filho ou o pai dum nó não existir, será representado pelo *apontador vazio*, nas linguagens que suportarem o conceito. Define-se a propriedade RBT da seguinte maneira:

1. Cada nó é *preto* ou *vermelho*.
2. *Todas as folhas (apontadores vazios) são consideradas pretas*.
3. Se um nó é vermelho, *ambos os seus filhos serão pretos*.
4. Qualquer caminho desde um nó até qualquer folha contém o *mesmo número de nós pretos*.

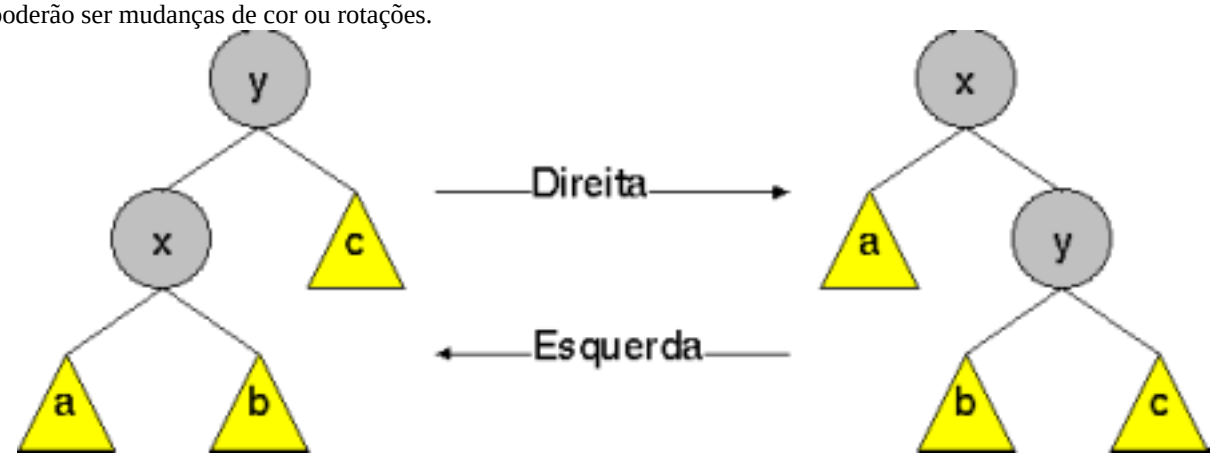
Se estas propriedades forem respeitadas, a altura duma RBT será no máximo $2log(N+1)$ em que N é o número de nós da árvore. Esta característica garante que a operação **membro** pode sempre ser feita em $T = O(logN)$. Um exemplo de Red-Black Tree é a seguinte:



Tal como é o caso com as árvores AVL, o problema surge quando se pretende aplicar a função **insere**, porque podemos ficar numa situação em que as propriedades RBT são violadas. As operações que iremos usar são as rotações e as mudanças de cor.

Definição: Seja a $hrb(X)$ *altura Red-Black* dum nó X o número de nós pretos de qualquer caminho de X até uma folha. Pela propriedade 4 este valor existe e é único. Pode-se demonstrar que uma RBT com N nós internos terá a altura máxima $2log(N+1)$.

Na aplicação da função de inserção (o mesmo é verdade para a remoção), há a possibilidade de a árvore resultante violar as propriedades RBT. À partida a operação **insere** é feita em $T = O(logN)$ (por a árvore inicial estar equilibrada). Se a árvore resultante não respeitar as condições RBT, teremos de a alterar para "repôr a normalidade". As alterações poderão ser mudanças de cor ou rotações.



A título de exemplo, eis código C para a rotação esquerda (a rotação direita pode ser fácilmente deduzida a partir desta).

```
 1  rot_esquerda (struct rbt **T, *x)        /* T é a raiz */
 2  {                                        /* Nota: assume x->fd != NULL */
 3    struct rbt *y = x->fd;                 /* Determina y em função de x */
 4    x->fd = y->fe;                         /* Passa o FE de y para FD de x */
 5    if (y->fe) y->fe->pai = x;
 6    y->pai = x->pai;                       /* y passa a filho do pai de x */
 7    if (! x->pai)
 8      *T = y;
 9    else if (x == x->pai->fe)
10      x->pai->fe = y;
11    else
12      x->pai->fd = y;
13    y->fe = x;                             /* Passa x para FE de y */
14    x->pai = y;
15  }
```

Note-se que o parâmetro correspondente à árvore é passado *por referência* (o `**` em vez de `*`), para permitir que a raiz seja modificada. Uma formulação mais ortodoxa, em termos de C, poderia ser:

```
1  struct rbt *rot_esquerda (struct rbt *T, *x)    /* T é a raiz */
2  {                                        /* Nota: assume x->fd != NULL */
3    struct rbt *y = x->fd;                 /* Determina y em função de x */
4    x->fd = y->fe;                         /* Passa o FE de y para FD de x */
5    if (y->fe) y->fe->pai = x;
```

```
 6    y->pai = x->pai;                      /* y passa a filho do pai de x */
 7    if (! x->pai)
 8      T = y;
 9    else if (x == x->pai->fe)
10      x->pai->fe = y;
11    else
12      x->pai->fd = y;
13    y->fe = x;                            /* Passa x para FE de y */
14    x->pai = y;
15    return T;
16  }
```

Que deverá ser utilizado tendo em atenção que o resultado da função é a nova raiz da árvore (possívelmente inalterada).

Portanto, o procedimento a seguir para a inserção dum nó X numa RBT é dado por:

1. Insere-se X como numa árvore binária de pesquisa normal. X irá forçosamente substituir uma folha existente.
2. Dá-se a X a cor vermelha.
3. Caso seja necessário (ie. se qualquer uma das propriedades RBT fôr violada), repõe-se as propriedades das RBTs, para tal:
   (a) Alteram-se cores de nós.
   (b) Fazem-se rotações.

Correspondente a esta descrição pode-se usar o código:

```
 1  insere_rbt (struct rbt **T, *x)
 2  {
 3    struct rbt *y;
 4    insere_abp (T, x);            /* Como numa árvore binária normal */
 5    x->cor = VERMELHO;

 6    while (x != *T && x->pai->cor == VERMELHO) {
 7      if (x->pai == x->pai->pai->fe) {
 8        y = x->pai->pai->fd;
 9        if (y->cor == VERMELHO) { /* ------------------------ CASO 1 */
10          x->pai->cor = y->cor = PRETO;
11          (x = x->pai->pai) -> cor = VERMELHO;
12        }
13        else {
14          if (x == x->pai->fd)    /* ------------------------ CASO 2 */
15            rot_esquerda (T, (x = x->pai));
16          x->pai->cor = PRETO;    /* ------------------------ CASO 3 */
17          x->pai->pai->cor = VERMELHO;
18          rot_direita (T, x->pai->pai);
19        }
20      }
21      else {              /* igual, trocando "direita" com "esquerda" */
22        ...;              /* e "fe" com "fd"... */
23      }
24  }
```

Comentários ao código:

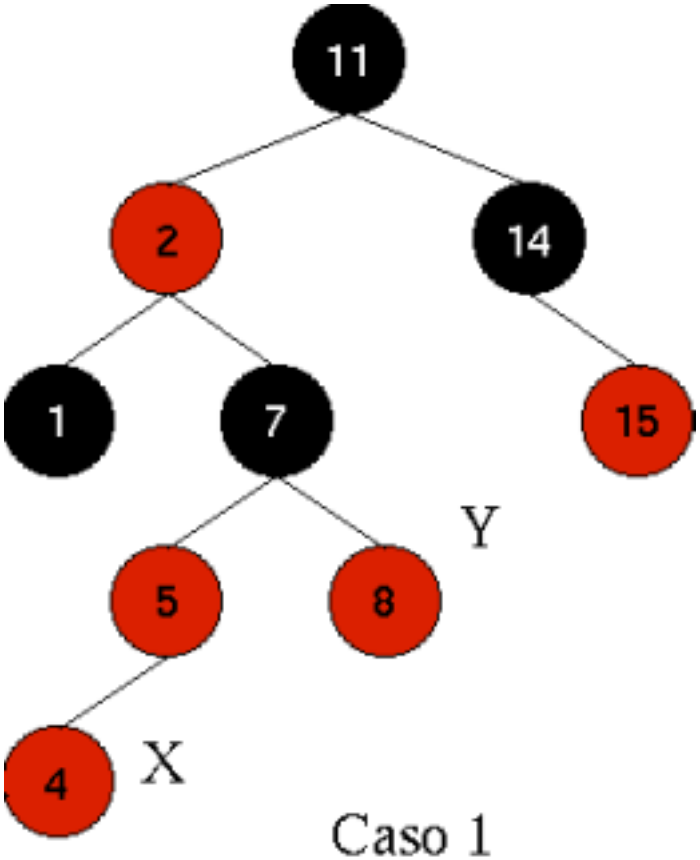
- Há 3 aspectos a ter em conta: (1) determinação precisa de que formas de violação das propriedades RBT podem ser introduzidas na inserção com o algoritmo da ABP normal, (2) explicação do objectivo do ciclo while e (3) ver como é que os 3 casos dentro do ciclo satisfazem o que se pretende (a manutenção das propriedades RBT).
- As propriedades 1 e 2 nunca poderão ser violadas. A propriedade 4 também não, porque substitui um nó preto (uma "folha", na definição 2) por uma cadeia composta por um nó vermelho (o novo) e outro preto: mantém-se assim o número de nós pretos. Já o mesmo não se pode dizer da propriedade 3, que pode ser violada: esta situação ocorre quando o pai de X é vermelho.
- O objectivo do ciclo while é, sabendo que há uma violação da propriedade 3, o de "empurrar" a violação para cima, na árvore RBT. Para interpretar o código, Deve-se procurar manter a propriedade 4 como invariante. Assim, no início de cada passagem pelo ciclo, X designa um nó vermelho com um pai vermelho e esta constitui a única violação das propriedades RBT em toda a árvore. Y designará o (único) "tio" de X (ie. o outro filho do avô de X).
- Consequências possíveis duma passagem pelo ciclo: ou o apontador X sobe na árvore ou são feitas rotações que resultam na conformidade com as propriedades RBT e portanto pode-se sair do ciclo.
- Situações dentro do ciclo: na realidade há 6 casos, mas 3 são simétricos aos outros 3, pelo que foram omitidos do código apresentado. O código é válido porque há o pressuposto que a raiz é preta: esta propriedade é garantida pela última instrução do código. Esta situação permite afirmar que `x->pai` nunca é a raiz, e portanto existe sempre `x->pai->pai`.
- O `CASO 1` difere dos 2 e 3 pela cor do "tio" de `X`. Em todos os casos, o "avô" de `X` é sempre garantidamente preto.

Assim, em cada iteração, resolve-se o problema que possa existir em `X`, passando-o para cima. A iteração seguinte irá aplicar-se ao nó acima, ie. `X` irá designar outro nó.

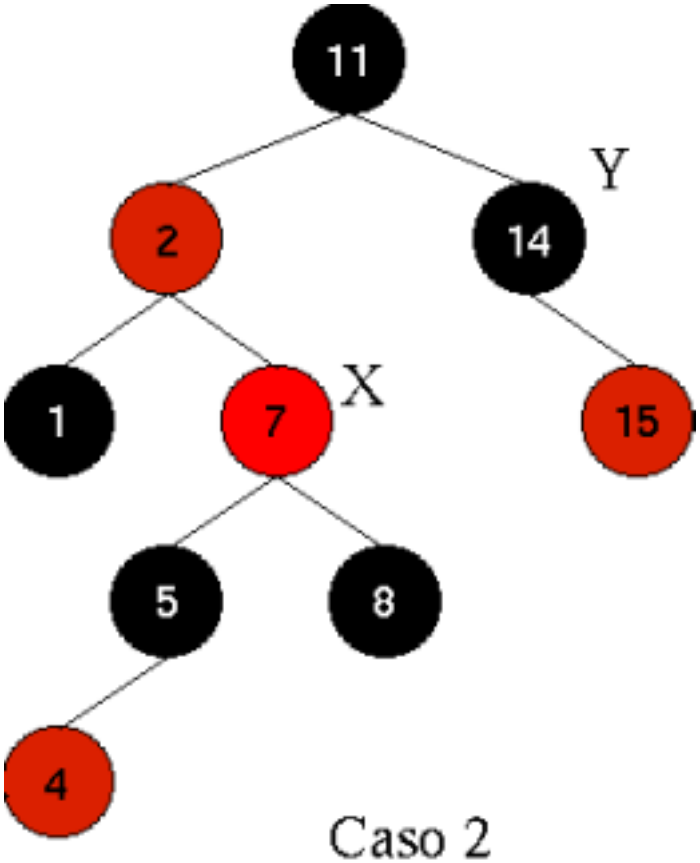
Em todos os casos, podemos afirmar que o avô de `X` é preto (porque a propriedade 3 só é violada entre `X` e `X->pai`).

- No caso 1, passa-se o "excesso de vermelho" para cima, pinta-se o pai de `X` e `Y` (o "tio") de preto e pinta-se o avô de `X` de vermelho. Na situação resultante só pode haver problemas entre o avô e o bisavô de `X`. `X` passa a designar o seu anterior avô.
- No caso 2, se `X` fôr o filho direito do seu pai, `X` passará a designar o pai e faz-se uma rotação esquerda, colocando-nos na situação do caso 3.
- No caso 3, que pode ser a continuação do caso 2, o tio de `X` é preto e o pai é vermelho. Muda-se a cor do pai e do avô de `X` (passam a ser respectivamente preto e vermelho). Faz-se uma rotação direita sobre o avô de `X`. O caso 3 termina o ciclo (o pai passa a ser preto, pelo que já não há conflito com as propriedades RBT).

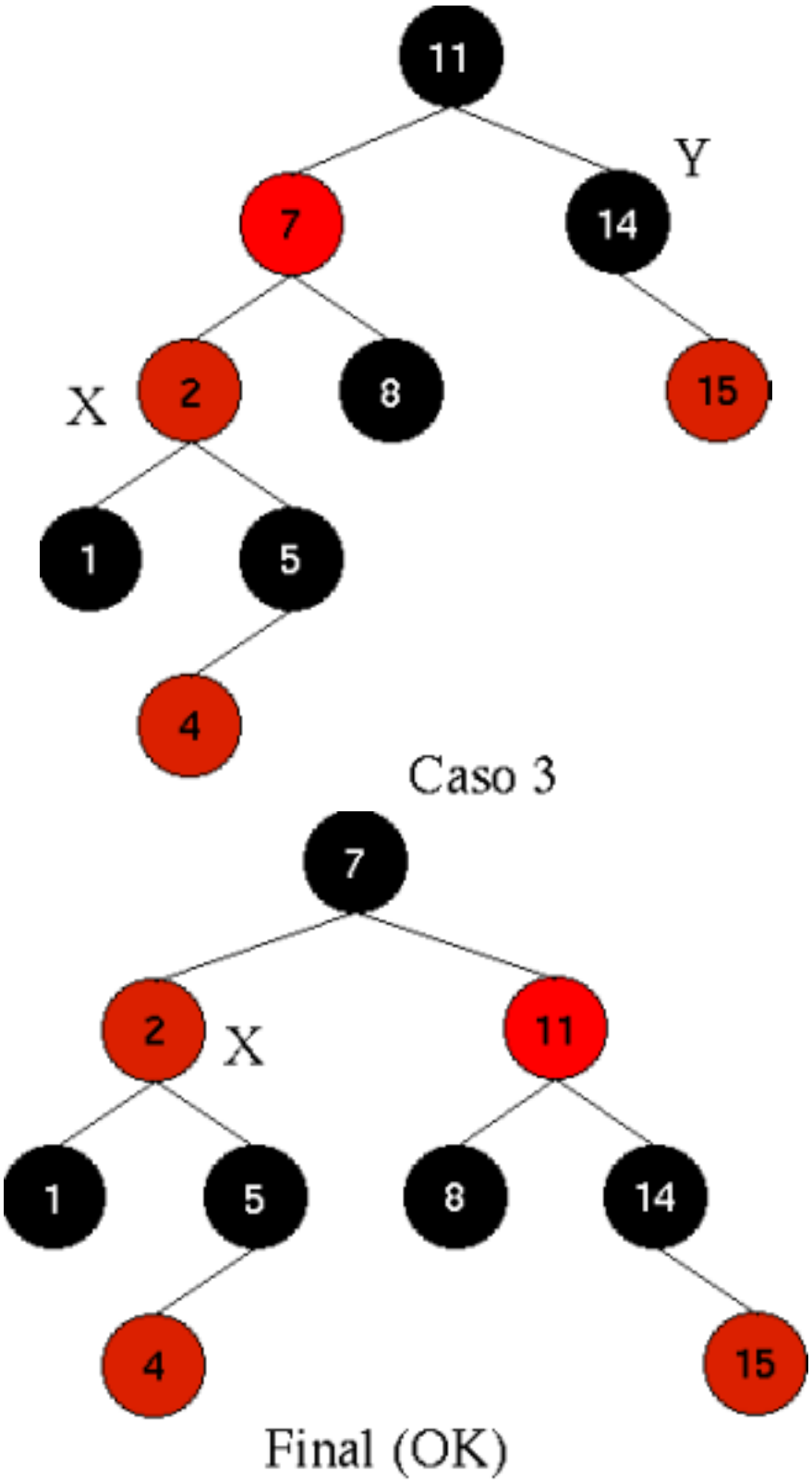
Exemplo: inserção do valor 4 na árvore (11, 2, 14, 15, 1, 7, 5, 8). Teremos a sequência de árvores e situações indicado nas figuras. É de observar os nós representados por `X` e `Y`.



Caso 1



Caso 2



Análise para **insere**:

- Dado que a altura da árvore é conhecida como sendo $O(logN)$, o tempo atribuível à inserção como árvore binária normal é $T = O(logN)$.
- Resta avaliar o tempo de execução do ciclo: cada passagem pelo ciclo é $O(1)$ dado que nenhuma das operações em questão (rotações e alterações de cor) depende de alguma forma de N. O ciclo só continua a ser executado

enquanto estivermos no caso 1, tendo cada iteração $T = O(1)$. Como se sobe na árvore em cada passagem no ciclo, e a profundidade desta é `O(log N)`, este será executado no máximo $K = O(logN)$ vezes.

Temos duas partes sequenciais que são cada uma $T = O(logN)$, pelo que a complexidade total para a operação **insere** no caso das Red-Black Trees é $T = O(logN)$.

Comentário: este resultado era de esperar dado o enunciado da propriedade RBT, que garantia a altura máxima da árvore. No entanto restava uma dúvida acerca do custo da manutenção desta propriedade: a análise revelou que esta também pode ser feita em $T = O(logN)$, pelo que as árvores Red-Black são uma boa estrutura de dados para inserção sucessiva de valores, independentemente da sua ordem, porque não tem casos patológicos.

4.2 B-Trees

4.2.1 Definição

uma B-Tree de ordem M é uma árvore em que:

- Os nós interiores têm entre $ceil(M/2)$ e M filhos. Um nó interior contém referências qualificadas aos seus filhos. A estrutura dum nó interior pode ser encarada como:
  1. Uma sequência de $K \leq M$ referências aos filhos.
  2. Uma sequência $m_1..m_{K-1}$ de $K-1$ valores, em que o valor de m_I representa o *mínimo* dos valores da sub-árvore I.
- A raiz é uma folha ou tem entre 2 e M filhos. No segundo caso, a raiz é considerada como um nó interior e tem as mesmas características que estes.
- As folhas estão todas à mesma profundidade. E servem para guardar os dados.[1]

Às vezes designam-se as B-Trees de ordem M como árvores $(M-1)-M$, por exemplo uma B-Tree de ordem 3 é uma árvore 2-3.

4.2.2 Operações

- **Membro.** Este processo é simples e equivalente a fazer uma pesquisa (binária) em cada nó, descendo para a sub-árvore em que se encontra o mais alto mínimo superior ou igual à chave de pesquisa.
- **Inserção.** Inicialmente igual à operação membro. Obtemos assim a folha onde seria para inserir. Caso não haja violação da propriedade de definição (ie, há menos de M valores na folha), inserimos e a operação fica completa. Possívelmente teremos de ajustar os valores do mínimo (m_i) em todos os nós no caminho até à folha.

  Caso a nova chave não caiba na folha selecionada: temos de repôr a conformidade com as características da B-Tree:

  1. Dá-se mais um irmão à folha onde se iria inserir, redistribuindo os valores entre as novas folhas. Ajustam-se os índices m_i do pai.
  2. Caso não seja possível (porque o pai já tem M filhos):

     Repetir a operação anterior ao nível superior, ou seja: dividir o pai em dois, cada um com a metade dos filhos. Inserir o novo 'tio' no 'avô'.
  3. Caso não seja possível, repetir operação ao nível do avô. Chegando à raiz, e esta estando cheia...

[1] A variante aqui descrita é geralmente conhecida por B+-Tree. A diferença para as B-Trees originais é que nestas os nós interiores também contêm dados.

4. ... Mantém-se os filhos e refaz-se uma árvore mais profunda: parte-se o conjunto dos actuais elementos da raiz em dois e criam-se dois novos nós para albergar esses elementos.

- **Remoção.**

  Esta operação é Semelhante à inserção, excepto:

  1. Se número de elementos da folha resultante fôr maior que 1, não é preciso fazer mais nada.
  2. Se número de elementos da folha resultante fôr 1, combina-se o elemento restante com um irmão.
  3. Se passou a ser filho único, repetir a operação ao nível superior.

4.2.3 Análise

- A profundidade máxima duma B-Tree de ordem M na qual foram inseridos N elementos é $ceil(log_{floor(M/2)}N)$, portanto trata-se duma grandeza $O(logN)$.
- Na operação **membro**, em cada nó fazemos $O(logM)$ operações para determinar por onde vamos (usando o algoritmo de pesquisa binária sobre os m_i). Como a altura da árvore é $O(logN)$ teremos como caso médio $T = O(logN)$, por termos fixado M.
- Na **inserção e remoção**, estamos na mesma situação do que na operação **membro**, mas podemos ter de fazer $O(M)$ operações para repôr as condições. Se ignorarmos M por este estar fixado (em geral sê-lo-á), estas operações são efectuadas em $T = O(logN)$. O pior caso tem a mesma complexidade pois a operação de partir a raiz em 2 é feita em $T = O(1)$.

4.2.4 Uso

Usos das B-Trees: em bases de dados, em que a árvore é mantida em disco e não em memória principal. Se o número de acessos a nós fôr o número de acessos a disco, e que esta operação (acesso a disco) fôr muito lenta, convém evitá-los.

Assim, guarda-se a estrutura da árvore em memória (com acessos rápidos) e as folhas (que contém os dados) em disco. O tamanho das folhas é escolhido por forma a caber num bloco em disco. Os valores de M costumam ser relativamente grandes, sendo habitual valores entre 32 e 256, por forma a ter árvores relativamente pouco profundas.

4.3 Tries

São aos restantes algoritmos de dicionário o que o 'bucket sort' e 'radix sort' são aos algoritmos de ordenação.

Aplicam-se a conjuntos de dados em que as chaves sejam sequências de elementos sobre um conjunto finito, de tamanho reduzido. Por exemplo, uma sequência de cadeias de caracteres qualifica-se como uma sequência de chaves às quais se podem aplicar tries.

Definição: uma Trie é uma estrutura de dados de dicionário que se aplica a valores que sejam cadeias sobre um alfabeto Σ finito. Seja $K = |\Sigma|$ o cardinal de Σ, uma trie T poderá ser:

- A trie vazia, que não contém nenhum elemento.
- Um conjunto de triplos (C_i, S_i, T_i) em que C_i é um membro de Σ, S_i é 1 se C_i estiver no dicionário T ou 0 caso contrário e T_i é outra trie.

  Neste caso diz-se que os elementos da trie T são as cadeias de comprimento 1 C_i tais que S_i é 1 assim como todas as cadeias com prefixo C_i em que o sufixo está na trie T_i.

4.3.1 Implementação

Dependendo da natureza dos dados, uma trie poderá ser implementada como uma estrutura *densa* ou uma estrutura *esparsa*.

- Numa estrutura *densa*, cada nó da trie é um vector indexado pelos membros de Σ, ou seja os C_i. Nesta situação acede-se aos elementos
- Numa estrutura *esparsa*, cada nó da trie é representado directamente como pela definição.

4.3.2 Análise

Essencialmente, para inserir uma cadeia de M símbolos teremos de fazer $O(M)$ comparações. Se estivermos a inserir $N >> M$ cadeias, teremos que o tempo médio de inserção será $O(N)$ pelo que uma trie tem complexidade temporal linear.

O problema vem do potencial desperdício de espaço.

4.4 Exercícios

1. Complete o código do TAD Red-Black Tree: defina o tipo e implemente as funções primárias (ver 4).
2. Desenhe a sequência de Red-Black Trees correspondentes à inserção da sequência de números 27 28 16 7 16 15 22 17 5 21 8 29 3 8 9 25 12 1 7 6 .
3. Repita o exercício anterior com a sequência de números 13 27 21 6 27 24 23 26 5 21 23 2 19 9 5 24 2 22 29 .
4. Indique como é que procederia para acrescentar a operação **remove** a uma implementação do TAD Red-Black Tree.
5. As Red-Black trees pretendem oferecer uma estrutura de dados Dicionário que não sofra dos problemas das árvores binárias de pesquisa simples, garantindo que as árvores nunca ficam *muito* desequibradas. As árvores AVL *garantem* que as árvores ficam *sempre* equilibradas.
   (a) Caracterize as situações em que as árvores AVL são preferíveis às RBTs e vice-versa.
   (b) **(*)** Verifique experimentalmente as suas conclusões, implementando duas livrarias com a mesma interface (um tipo ”opaco” `dic`, e as operações primárias sobre esse tipo (4)). Essas duas livrarias deverão ser usadas com *um mesmo programa* de teste, que será aplicado a vários conjuntos de dados.
6. **(*)** A implementação descrita para o TAD Red-Black Tree utiliza nós com apontadores para o pai: esta abordagem pode ser encarada como desnecessária pois desperdiça espaço. Descreva uma forma de evitar este problema, guardando o mínimo de informação possível num nó.
7. Considere uma B-Tree de ordem 4. Desenhe todos os passos da inserção dos valores 27 28 16 7 16 15 22 17 5 21 8 29 3 8 9 25 12 1 7 6 .
8. **(*)** Descreva um algoritmo para converter uma B-Tree numa Árvore Binária de Pesquisa. Procure obter um processo que não consista numa simples travessia sequencial da B-Tree em que cada elemento seria inserido na ABP. Explique porque é que este método (trivial) seria indesejável. Caracterize a complexidade temporal do seu algoritmo.
9. Desenhe uma Trie correspondente à inserção da sequência de números `27918 49125 30501 18508 59307 6348 47938 8068 52311 10478 23987 10924 19618 33653 37491 56002`. Utilize uma chave de indexação com 10 valores distintos.

10. Suponha que pretendemos usar tries para encontrar *frases* e não simplesmente palavras: as chaves de indexação passam a ter um número infinito de valores possíveis. Proponha uma variante da estrutura de dados das Tries apresentada nas aulas teóricas que suporte a inserção de valores sobre domínios muito grandes. Caracterize a sua abordagem em termos de complexidade temporal no pior caso.

5 Algoritmos de Pesquisa de Cadeias de Caracteres

O problema de localizar ocorrências duma cadeia de caracteres (*strings*) num texto é muito frequente. Existem muitos algoritmos com qualidades diferentes para satisfazer esta necessidade. Iremos explorar alguns dos principais algoritmos de pesquisa de strings.

5.1 Pesquisa 'ingénua'

O problema pode ser formulado como: dado um conjunto de símbolos Σ, que designaremos por *alfabeto*, designaremos as sequências de zero ou mais símbolos pertencentes a Σ por *palavras* ou *strings*. Designaremos por T (de *text*) e P (de *pattern*) dois strings sobre um mesmo alfabeto Σ com comprimentos respectivos N e M.

Definições:

- Operações sobre strings: igualdade, catenação, prefixo, sufixo, substring.
- Comprimento dum string. Substring numa posição I.
- Ocorrência numa posição I.

Formulação: encontrar o índice I da primeira ocorrência de P em T. Significa encontrar o mais pequeno I tal que exista um string R com $comprimento(R) = I$ e RP é um prefixo de T.

```
 1  int naive_search (char *T, char *P) {
 2       char *t;

 3       for (t = T; t < T + N - M; ++t) {
 4            char *tt = t, *p = P;

 5            while (*p && *p==*tt)
 6                 ++tt, ++p;
 7            if (*p == '\0') return (t - T);
 8       }
 9       return -1;
10  }
```

Descrição do funcionamento do algoritmo. Caracterização em termos de complexidade, se $N = comprimento(T)$ e $M = comprimento(P)$, teremos $T = O(NM)$.

5.2 Autómatos finitos

5.2.1 Linguagens

- Alfabeto: conjunto de símbolos possíveis (habitualmente designado por Σ)
- Frase (ou 'string'): sequência de símbolos (habitualmente S) sobre Σ.

- Frase vazia: λ ou ε.
- Operações sobre frases:
  - Catenação.
  - λ elemento neutro da catenação.
  - Potência: designa aplicação repetida da catenação.
- Linguagem finita: número de frases distintas finito. Em particular uma linguagem constituída por um único string é uma linguagem finita.

5.2.2 Reconhecedores

Reconhecedor para uma linguagem: programa que tem como input uma frase e como output um booleano, que será verdade se a frase estiver na linguagem. Há que considerar a terminação, i.e. determinar se o reconhecedor termina.

Objectivo: Construção dum reconhecedor para uma linguagem finita, em particular para uma linguagem constituída por um único string.

Dispositivo: autómato finito (AF). Seja A um AF, teremos $A = (\Sigma, S, \delta, s_0, F)$, em que:

- S é um conjunto de estados.
- Σ é um conjunto de símbolos
- δ é uma função de S e Σ em S.
- $s_0 \in S$ designa o estado *inicial*.
- F designa um subconjunto de S, em que os estados membros são designados *finais* ou *de aceitação*.

Autómatos Finitos: um AF pode ser representado como um grafo dirigido, em que:

- os nós são os estados.
- os arcos são as transições (a extensão da função trans).

Num AF Não-Determinístico (NFA) as transições também podem ser em ε e pode haver mais duma transição sobre o mesmo símbolo a partir dum dado estado. Para um AF poder ser dito *determinístico* (DFA), as condições acima não se podem verificar. Os DFAs são fáceis de simular.

Simulação dum DFA:

- Autómato extendido, Igual ao anterior Junta-se um novo símbolo, $ para designar o fim do input
- Representação da função de transição: como árvore n-ária (ramos etiquetados com a transição) ou como vector indexado com a transição Como hash table indexada com a transição.

Aplicação ao problema da pesquisa de strings.

```
1  int dfa_search (char *T, char *P) {
2       char *t = T;
3       state *S = make_dfa (P);

4       for (; S; S = transition (S, *t++)) {
5            if (accepting (S))
```

```
6                 return (t - T - 1);
7       }
8       return -1;
9   }
```

Resta a definição das funções auxiliares utilizadas em `search_dfa`, nomeadamente `make_dfa` e `transition`.

Análise: a função `make_dfa` pode ser efectuada em $T = O(M)$ se usarmos uma *Trie* para representar o DFA. A função `transition` pode ser feita em `T=O(1)`. O ciclo é efectuado no máximo N (comprimento de T) vezes, pelo que a complexidade temporal deste algoritmo é $T = O(M+N)$, o que o torna claramente preferível à pesquisa ingénua.

5.2.3 Extensão

Extensão: o algoritmo de pesquisa de strings usando DFAs pode ser extendido, sem modificações ao ciclo, para efectuar a pesquisa simultâneamente em vários strings.

5.3 Algoritmo de Rabin-Karp

Descrição: utilizar uma função de hash para comparar P com sub-strings de T. A função de hash deverá permitir encarar sequências de símbolos sobre Σ como se fossem números na base K, em que K é o número de símbolos distintos em Σ.

```
1   const unsigned int Q = ...;
2     /* Q é PRIMO e Q < 256^sizeof(unsigned int) */

3   int search_rabin_karp (char *T, char *P) {
4       int N = strlen (T);
5       int M = strlen (P);
6       unsigned int ph = rk_init (P, M, Q);
7       unsigned int th = rk_init (T, M, Q);
8       char *t = T;

9       while (t <= T+N-M) {
10          if (th == ph && streq (t, P))
11              return (t - T);
12          th = rk_shift (th, M, t);
13          ++t;
14      }
15      return -1;
16  }
```

Explicação: trata-se de comparar "assinaturas" de sub-strings de T com a assinatura de P, em que:

- Uma *assinatura* é o resto da divisão dos números correspondentes a P e T_i, ambos correspondentes a strings de comprimento M, por um número Q, devidamente escolhido de forma que:
  - Q seja primo.
  - KQ seja tão grande quanto possível mas ainda caiba numa palavra do computador utilizado.
- `rk_init` é a função que determina os "números" correspondentes ao resto da divisão por Q de P e T_0.
- `tk_shift` é a função que "avança" o resto da divisão por Q de T_i para obter o de T_{i+1}.

Análise: `rk_init` tem $T = O(M)$. `rk_shift` tem $T = O(1)$. A comparação `streq` tem $T = O(M)$ mas não se espera que seja invocada excepto na ocorrência de P.

5.4 Algoritmo de Boyer-Moore

Ideia fundamental: analisar o padrão para poder percorrer o texto mais depressa (ie. com menos comparações).

5.4.1 Introdução

Consideram-se sufixos do padrão:

1. Comparar o símbolo actual do padrão (de trás para a frente) e o correspondente no texto.
2. Se coincidir, recuar para o símbolo anterior (no padrão e no texto).
3. Senão, usar (1) o número de caracteres já identificados e (2) o código do caracter *do texto* que não coincidiu com o padrão para obter o número de posições a avançar o padrão, no texto.

5.4.2 Heurísticas

A questão (quando não se encontrou uma ocorrência) é portanto saber em *quantas* posições é que se desloca o padrão para a direita, ie. que salto é que se faz. Aqui podem ser aplicadas várias estratégias (heurísticas):

- A heurística das (más) *ocorrências*, em que se coloca o caracter (do texto) que diferiu alinhado com a sua *última* ocorrência no padrão. A tabela referida é construída colocando em `do[b]` (**D**eslocamento por **O**corrências) o índice da última ocorrência do símbolo `a`. Considera-se que todos os símbolos (que não os do prefixo) ocorrem simultâneamente antes de `P`, pelo que `do[b]` em que `b` não é um símbolo do padrão é igual ao comprimento do string (implica que só pode haver coincidência do padrão *à direita* do símbolo `b`).
- A heurística das (boas) *coincidências* ('matches') resulta de observar que, quando se desloca o padrão para a direita, na nova posição este deve:
  1. Coincidir com *todos* os caracteres já encontrados. Isto significa encontrar um substring de `P` que coincida com um sufixo do mesmo `P`.
  2. Colocar um caracter *diferente* (no padrão) do que não coincidiu com o texto, na posição em questão. Esta condição implica encontrar a *última* ocorrência do substring de `P` igual ao sufixo de `P` do comprimento verificado, em que o caracter *anterior* a esse substring seja *diferente* do caracter anterior ao sufixo.

  A implementação desta heurística implica construir a tabela `dc` (**D**eslocamento por **C**oincidência) em que, para cada comprimento dum *sufixo*, se indica de quantas posições se deve descolar o padrão por forma a que a coincidência do sufixo já verificada coincida com um prefixo de comprimento máximo (deverá ser um prefixo do sufixo já visto).

Assim, um código possível para a pesquisa usando o algoritmo de Boyer-Moore é dado por:

```
1  int search_boyer_moore (char *T, char *P) {
2    int N = strlen (T);           /* opcional */
3    int M = strlen (P);
4    char *t = T+M;
5    char *p = P+M;
```

```
 6    do = bm_init_do (T, P);
 7    dc = bm_init_dc (T, P);

 8    while (p > P) {
 9      if (*--p != *--t) {
10        t += max (do[*t], dc[p-P]);
11        p = P+M;
12      }
13    }
14    if (p == P) return t-T; else return -1;
15  }
```

Resta definir as tabelas `do` e `dc`, que podem ser calculadas com a aplicação, respectivamente, das duas heurísticas mencionadas anteriormente.

5.4.3 Exemplo

Queremos encontrar o padrão '`abaldraba`' (comprimento 9) num texto qualquer. Temos de inicializar as tabelas `do` e `dc`, que serão:

- `do` (indexado pelas letras): `[a]:0`, `[b]:1`, `[d]:4`, `[l]:5`, `[r]:3`. Para todos os outros valores do índice teremos `[X]:9`, que corresponde a saltar tudo.
- `dc` (indexado pelo primeiro índice dentro de `P` do sufixo já identificado, omitindo portanto o caracter que não coincidiu; este é só feito para sufixos não-vazios): `[7 (b/a)]:2 ()`, `[6 (a/ba)]:9`, `[5 (r/aba)]:6`, etc.

5.4.4 Algoritmo de Boyer-Moore simplificado.

Baseado na observação que as ocorrências múltiplas dum substring nos padrões são raras, pode-se simplesmente registar a última ocorrências de cada símbolo, pelo que se utiliza só uma heurística (a das *ocorrências*, portanto o índice `do`) em vez das duas.

A versão simplificada do algoritmo de Boyer-Moore ocupa menos espaço, demora menos tempo a preparar e, na prática, é tão eficiente como o completo.

5.4.5 Comparação de desempenho

Ao comparar os diversos algoritmos em termos de tempo de processador, para a busca dum padrão cujo comprimento é o parâmetro num texto fixo, observa-se que:

- O algoritmo ingénuo demora um tempo constante, indiferentemente do comprimento do padrão.
- O algoritmo de Rabin-Karp exibe um comportamento semelhante, embora efectue menos comparações.
- O algoritmo de Autómatos Finitos também tem o mesmo comportamento. A sua relação com o de Rabin-Karp depende muito da implementação e das características do alfabeto.
- O algoritmo de Boyer-Moore demora um tempo que decresce quando o comprimento do padrão aumenta.

Dependendo da aplicação, a situação é: o Rabin-Karp é preferível até padrões dum certo comprimento (normalmente na ordem de 5 a 10 caracteres), depois disto o Boyer-Moore torna-se mais rápido pois potencialmente efectua saltos que podem ir até ao comprimento do padrão.

5.5 Pesquisa de vários padrões simultâneos e de expressões regulares

Vários dos algoritmos apresentados extendem-se naturalmente à pesquisa simultânea de vários padrões, tal é o caso do algoritmo de pesquisa por Autómatos Finitos.

5.6 Exercícios

1. Implemente o algoritmo de pesquisa por DFAs.
2. Faça uma implementação do algoritmo de Rabin-Karp.
3. Complete a definição do algoritmo de Boyer-Moore simplificado, definindo uma implementação das funções de inicialização da tabela `do`.
4. (*) Compare os desempenhos das implementações dos algoritmos anteriores, fazendo variar os comprimentos do texto e do padrão.
5. Construa a tabela `do` para procurar o string . . .

6 Algoritmos sobre Grafos

Frequentemente é necessário representar estruturas de dados que são implementações dum modelo formal assente na teoria de grafos. Exemplos destas situações ocorrem em sistemas operativos, em compiladores, em bases de dados, etc. Pode-se dizer que o *grafo* é genéricamente o tipo abstracto de dados mais recorrente.

A totalidade dos TADs com estruturas ligadas (ex. listas, pilhas, filas, árvores, etc.) podem ser encaradas como especializações de grafos.

A matéria sobre este assunto (grafos) acompanha os capítulos 23 a 25 do Cormen.

6.1 Formulação e Representação

Definição dum grafo G como um par $G = (V,E)$ em que V é o conjunto de nós (vértices) e E é o conjunto de arcos (*edges*), que são pares de elementos de V.

Como em todos os problemas que se pretende resolver, a representação é crucial no seu efeito sobre o desempenho da solução. No caso dos grafos temos muitas formas de os representar, mas importa mencionar uma propriedade que influencia muito a forma como estes são representados: o número de nós e arcos, assim como a distribuição relativa destes pode levar a duas formas principais de implementação:

1. Grafos esparsos, caracterizados por $|E|$ ser muito menor que $|V|^2$. Um grafo com estas características será provávelmente melhor representado como uma lista de nós adjacentes a um determinado nó.
2. Grafos densos, caracterizados por $|E|$ ser próximo de $|V|^2$. Neste caso, a representação preferencial será uma *matriz de adjacências*, em que se indexa por quaisquer dois nós do grafo, e o valor é 1 se o arco existir ou 0 caso contrário.

   No caso de estarmos a representar um grafo não-dirigido podemos reduzir a matriz de adjacências a pouco mais de metade, explorando a simetria.

Grafos *ponderados*, em que a cada arco está associado um *peso* podem fácilmente ser descritos com ligeiras variantes das duas representações. Na representação de matriz de adjacências, pode-se optar por denotar a ausência de arco por um peso infinito.

6.2 Busca em Largura (*breadth-first*)

É um algoritmo que está na base de muitos algoritmos para grafos. Definição:

- Input: um grafo $G = (V,E)$ e um nó distinguido de V, designado por s.
- Output: conjunto de nós atingíveis de s, a distância entre s e todos os elementos desse conjunto, uma *árvore de percurso em largura*.
- Algoritmo: anota-se os nós do grafo para indicar em que medida é que já foram *visitados*. Para tal começa-se por marcar todos os nós como não tendo sido visitados (*brancos*) . Os nós que já foram vistos dividem-se em duas categorias: os que já foram inteiramente vistos (*pretos*) e os que estão "em curso" (*cinzentos*) .

```
 1  void bfs (grafo G, vertice s)
 2  {
 3    queue Q;
 4    for (all v in G; v != s) {
 5      v->color = WHITE;
 6      v->d = INF;
 7      v->p = NULL;
 8    }
 9    s->color = GRAY;
10    s->d = 0;
11    s->p = NULL;
12    insert (Q, s);
13    while (! empty(Q)) {
14      u = head (Q);
15      for (all v in u->adj) {
16        if (v->color == WHITE) {
17          v->color = GRAY;
18          v->d = u->d + 1;
19          v->p = u;
20          insert (Q, v);
21        }
22      }
23      delete (Q);
24      u->color = BLACK;
25    }
26  }
```

A intenção do campo `p` tornar-se-á evidente posteriormente.

6.2.1 Análise

Análise de complexidade: resulta em $T = O(V+E)$.

6.2.2 Caminho mais curto

Seja a *menor distância* $d(s,v)$ o menor número de arcos num caminho de s para v.

Lema 1 *Seja (u,v) um arco de G, teremos que $d(s,v) \leq d(s,u)+1$.*

Lema 2 *Ao terminar a execução do algoritmo `bfs`, teremos `v->d`$\geq d(s,v)$.*

Lema 3 *Supondo que, na execução do algoritmo `bfs` temos os vértices $(v_1,v_2,...,v_r)$ em que v_1 é a cabeça da fila. Teremos então $v_r->d \leq v_1+1$ e $v_i->d \leq v_{i+1}$ para todo i.*

Teorema 1 *Corre-se `bfs` num grafo $G=(V,E)$ com um vértice inicial $s \in V$. No fim, `bfs` encontra todos os vértices atingíveis de s e calcula a distância de s, dada por $d(s,v)=v->d$. O caminho mais curto de s até v é dado pelo caminho mais curto de s até $v->p$ seguido de v.*

6.2.3 Árvore *breadth-first*

Define-se o *grafo de antepassados* de G formado por (V_p,E_p), com: (defs para ...)

O grafo de antepassados é uma árvore breadth-first. Se houver um caminho na árvore breadth-first de *s* para *u* então esse caminho também existe em *G* e é o mais curto.

O grafo de antepassados é uma árvore porque é conexo e tem $|E|=|V|-1$.

6.3 Busca em Profundidade

A ideia aqui é, ao detectar um sucessor dum nó, procurar explorar os sucessores desse nó *antes* de fazer o mesmo para os restantes sucessores do antecessor do nó em questão.

Pode-se usar o código:

```
 1  void dfs (grafo G) {
 2    for (all v in G) {
 3      v->color = WHITE;
 4      v->p = NULL;
 5    }
 6    time = 0;
 7    for (all v in G)
 8      if (v->color == WHILE)
 9        dfs_visit (v);
10  }

11  void dfs_visit (vértice v) {
12    time = time+1;
13    v->color = GRAY;
14    v->d = time;
15    for (all u in v->adj)
16      if (u->color == WHITE) {
17        u->p = v;
18        dfs_visit (u);
19    }
20    time = time+1;
21    v->color = BLACK;
22    v->f = time;
23  }
```

O atributo `d` representa o *tempo* em que um nó foi visto pela primeira vez, i.e. a altura em que foi marcado cinzento. O atributo `f` representa o *tempo* em que o nó foi completamente visitado, neste caso isto significa que todos os seus descendentes foram vistos. Coincide com a altura em que é marcado preto.

6.3.1 Propriedades da busca em profundidade

Teorema 2 *(Parentesis.)*

Numa travessia em profundidade dum grafo $G = (V,E)$ e para quaisquer dois nós u e v, dá-se exactamente uma das seguintes situações:

- *Os intervalos $[d(u), f(u)]$ e $[d(v), f(v)]$ são dijuntos.*
- *O intervalo $[d(u), f(u)]$ está contido em $[d(v), f(v)]$ e u é um descendente de v na árvore em profundidade de G.*
- *O intervalo $[d(v), f(v)]$ está contido em $[d(u), f(u)]$ e v é um descendente de u na árvore em profundidade de G.*

Corolário 1 *Imbricação dos intervalos dos descendentes.*

Consequência do teorema dos parentesis.

Teorema 3 *Caminho branco.*

Numa floresta depth-first dum grafo $G = (V,E)$, um nó v será um descendente dum nó u sse, na altura da descoberta de u, v pode ser atingido de u por um caminho que passe exclusivamente por nós brancos.

6.3.2 Classificação dos arcos

Os arcos dum grafo G podem ser classificados em função da sua relação com os da floresta *depth-first*. Esta informação pode ser útil para caracterizar propriedades do grafo. Os tipos de arco que nos interessam são:

- Arcos de árvore (*tree edges*).
- Arcos para trás (*back edges*): ligam um descendente ao seu antecessor. Esta classificação inclui os ciclos de comprimento 1 (imediatos).
- Arcos para a frente (*forward edges*): saltos. Ligam um nó a um seu descendente não imediato.
- Arcos cruzados (*cross edges*): todos os outros tipos de arcos.

Em termos do algoritmo `dfs` esta classificação pode ser determinada pela cor dum vértice quando é encontrado: quando se tenta o arco (u,v), a cor de *v* indica:

- Branco: um arco de árvore.
- Cinzento: um arco para trás.
- Preto: um arco para a frente ou cruzado.

Teorema 4 *Numa pesquisa em profundidade dum grafo não-dirigido G, qualquer arco de G é ou de árvore ou para trás.*

6.4 Ordenação Topológica

Aplica-se a DAGs, ie. grafos dirigidos acíclicos. Uma *ordenação topológica* dum DAG $G = (V,E)$ é uma permutação de V tal que u aparece antes de v sse tivermos um arco (u,v) em E.

O algoritmo `topsort` consiste no uso de `dfs`, modificado no sentido de acrescentar cada nó, quando é marcado como preto, na cabeça duma lista ligada. A permutação pretendida é essa lista.

Lema 4 *Um grafo dirigido $G = (V,E)$ é acíclico sse uma travessia em profundidade de G não indica nenhum arco para trás.*

Teorema 5 *O procedimento `topsort` produz uma ordenação topológica dum grafo acíclico G.*

6.5 Componentes fortemente conexos

Definição de SCC.

Usa-se a *transposta* dum grafo $G = (V,E)$, dada por $G^T = (V,E^T)$ em que E^T é o conjunto dos arcos formados pelo par com a ordem inversa dos elementos de E.

Este método pode ser expresso pelo algoritmo:

```
1  scc (grafo G) {
2    bfs (G);
3    calcular GT /* transposta */;
4    bfs* (GT)   /* ciclo principal: v->f decrescente */;
5    /* cada árvore de bfs* é um SCC */
6  }
```

Lema 5 *Se dois nós estão no mesmo SCC, então não existe nenhum caminho entre eles que saia do SCC.*

Lema 6 *Numa busca em profundidade, todos os nós dum mesmo SCC ficam na mesma árvore depth-first.*

6.6 Exercícios

1. (TBD)

7 Algoritmos de Compressão

Os algoritmos de compressão são muito frequentemente usados, por permitirem uma melhor utilização de recursos como o espaço e a largura de banda. Embora sem fazermos uma análise em termos de *Teoria da Informação* (esta matéria será abordada noutro contexto), iremos aqui descrever alguns dos algoritmos mais simples para resolver este problema.

Conceitos relativos a uma mensagem: Elemento Significativo, Informação, Redundância, Entropia, Previsibilidade. Trabalhos pioneiros de Shannon (circa 1940-50) e Nyquist. Motivação original: transmissões militares. Ligação com a cifra.

7.1 Avaliação da compressão

Antes de descrever formas de comprimir mensagens, temos de estabelecer formas de *avaliação* do desempenho das técnicas descritas. Utilizaremos a definição de *quociente de compressão* dado por: $Q_{comp} = T_{inicial}/T_{comprimido}$ em que $T_{inicial}$ é o tamanho da mensagem original e $T_{comprimido}$ é o tamanho da mensagem comprimida.

A partir desta definição introduzimos a *taxa de compressão* dada por: $T_{comp} = 1/Q_{comp}$. Geralmente esta grandeza é expressa como uma percentagem.

O *ganho de compressão*, também expresso como uma percentagem, pode ser definido como $G_{comp} = 1 - T_{comp}$.

Por exemplo, se tivermos um ficheiro de 1000 bytes que, depois de comprimido, fica reduzido a 400 bytes, teremos um quociente de compressão de 2.5, uma taxa de compressão de 40% e um ganho de compressão de 60%.

7.2 Codificação estatística com comprimento variável

Ideia fundamental: atribuir aos símbolos mais frequentes as codificações mais curtas. A frequência de ocorrência pode ser conhecida de antemão mas frequentemente não será este o caso, pelo que teremos de determinar numa base casuística qual essa distribuição. Esta imposição leva a percorrer duas vezes a informação original (uma para analisar e outra para codificar/comprimir).

Para evitar esta situação, a análise pode ser feita sobre uma parte dos dados originais e não sobre a totalidade. O inconveniente maior desta abordagem é uma potencial desadequação das codificações utilizadas pois a *amostragem* pode não ser representativa do todo.

Aplicações em 'tempo real' destes métodos (por exemplo modems ou FAXs) fazem a análise sobre parte dos dados, repetindo (ajustando) a informação periódicamente.

Os códigos de Huffmann são os mais representativos destas técnicas de codificação.

7.2.1 Códigos de Huffmann

Procedimento: efectua-se uma análise estatística (frequência de ocorrência de cada símbolo) sobre o input. A ideia será atribuir aos símbolos mais frequentes as codificações mais curtas. Assim, e como pretendemos obter codificações construídas com dois símbolos (os valores 0 e 1 correspondentes a um bit) vamos construir uma árvore *binária* em que os nós terão como etiqueta a frequência de ocorrência de todos os símbolos correspondentes à sub-árvore aí enraizada. Só as folhas é que representam símbolos.

A construção desta árvore é feita iterativamente, como uma floresta que se vai reduzindo até ficar com um só nó.

1. A floresta consiste inicialmente nas futuras folhas da árvore, etiquetadas com o símbolo e a sua frequência de ocorrência.
2. Em cada passo são retirados os dois elementos com menor frequência. Estes são substituídos por um novo elemento, uma árvore, cuja raiz tem como frequência a soma das frequências dos nós removidos, e cujos filhos são os nós anteriormente removidos. Este novo elemento é inserido (ordenado por frequência) na lista.
3. O algoritmo termina quando há um só nó (será este a raiz da árvore de Huffmann).

A codificação de cada símbolo será a representação (em binário) do caminho da raiz até ao símbolo. Assim, e tomando como exemplo a palavra 'correspondentes' teremos:

```
1 Palavra: correspondentes

2 Letra:  c d e n o p r s t
```

```
 3  Freq:   1 1 3 2 2 1 2 2 1

 4  Arvore: c:1 d:1 e:3 n:2 o:2 p:1 r:2 s:2 t:1
 5          (c,d):2 e:3 n:2 o:2 p:1 r:2 s:2 t:1
 6          (c,d):2 e:3 n:2 o:2 (p,t):2 r:2 s:2
 7          ((c,d),n):4 e:3 o:2 (p,t):2 r:2 s:2
 8          ((c,d),n):4 e:3 (o,(p,t)):4 r:2 s:2
 9          ((c,d),n):4 e:3 (o,(p,t)):4 (r,s):4
10          (((c,d),n),e):7 (o,(p,t)):4 (r,s):4
11          (((c,d),n),e):7 ((o,(p,t)),(r,s)):8
12          ((((c,d),n),e),((o,(p,t)),(r,s))):15

13  Codigos: (0,1)
14          e = 01
15          n = 001
16          c = 0000
17          d = 0001
18          o = 100
19          p = 1010
20          t = 1011
21          r = 110
22          s = 111

23  Tinicial = 15*8 = 120 bits
24  Tcomp = 2*3 + 3*2 + 4*1 + 4*1 + 3*2 + 4*1 + 4*2 + 3*2 + 3*2
25        = 50 bits
26  Tcomp' = 5*15 = 75 bits
```

Neste caso temos um tamanho de mensagem original de 120 bits (supondo a codificação ASCII). Esta pode ser reduzida a 75 bits se usarmos 5 bits por letra. Com o códigos de Huffmann passa a 50 bits, pelo que temos um quociente de compressão de $Q_{comp} = 120/50 = 2.4$, ou seja uma taxa de compressão de 47%, ou ainda um ganho de compressão de 53%.

Estes ganhos devem ser moderados porque, para que a descompressão possa ser efectuada, é necessário transmitir a árvore (ou a tabela de frequências, embora esta necessite de maior coordenação entre compressão e descompressão). Quando os dados são poucos esta informação é significativa e afecta a compressão efectiva que se pode obter, no entanto este problema torna-se insignificante quando a quantidade de informação aumenta. Estudos sobre a língua inglesa apontam para um ganho de compressão médio entre 30 e 40%.

Em termos de implementação, a construção da árvore sugere que se utilize um heap, pois em cada iteração pretendemos aceder aos dois elementos com menor peso. Na descompressão o processo é mais simples e um autómato acíclico é suficiente.

Exemplos reais de aplicação: compressão MNP (modems), FAX, Photo-CD (para as imagens).

Discussão das características de complexidade destes algoritmos.

7.3 Codificação com dicionários

Estas abordagens diferem das anteriores por não procurar representações para os símbolos individuais mas sim para sequências destes, formando *palavras*. A representação duma ocorrência duma palavra será assim um *índice* num dicionário. Este tipo de método pode, com dicionários fixos e se o texto for tal que todas as 'palavras' ocorram no dicionário, resultar em boas taxas de compressão.

Colocam-se problemas com a abordagem descrita acima: é necessário haver concordância quanto ao dicionário, o tratamento de palavras não incluídas no dicionário não é passível de compressão, etc. Para endereçar estes problemas foram feitas várias propostas, uma primeira consiste numa análise prévia do texto para construção dum dicionário, seguida do envio do dicionário e da codificação do texto. Esta abordagem sofre do mesmo tipo de problemas que os códigos de Huffmann, embora em maior escala.

Lempel e Ziv propuseram algoritmos de compressão de dados em que o dicionário não precisa ser transmitido nem préviamente calculado, pois é construído incrementalmente quer na compressão quer na leitura.

7.3.1 Algoritmos de Lempel-Ziv

Lempel e Ziv propuseram um primeiro algoritmo em 1977 (conhecido como LZ77) em que o dicionário é representado por uma *janela* sobre o texto não comprimido. Ocorrências que não a primeira dum string na janela são substituídas por uma referência à posição do 'dicionário' e pelo comprimento da mesma. Esta posição é relativa à janela. Depois de processado todo o conteúdo da janela, esta é avançada duma quantidade não muito grande (relativamente ao seu tamanho) por forma a melhor explorar repetições.

O algoritmo LZ77 tem problemas pois é muito sensível ao tamanho da janela: para ser eficaz em termos de compressão é necessária uma janela grande, sendo que nesse caso o algoritmo torna-se muito ineficaz em termos de recursos computacionais (tempo). Assim, em 1978 os mesmos Lempel e Ziv apresentaram uma versão melhorada do seu algoritmo, este conhecido como LZ78: deixa-se de parte o conceito de janela deslizante e o dicionário é construído dinâmicamente, sendo persistente (ao contrário da janela). O processo de codificação consiste na emissão dum índice do dicionário e do próximo símbolo. A catenação do conteúdo do dicionário no índice indicado com o próximo símbolo também será inserida no dicionário, podendo assim ser reutilizada.

Esta forma de codificação tem algumas dificuldades, nomeadamente quando se ultrapassa o número de entradas no dicionário implicado pelo tamanho dos índices usados para codificar as ocorrências: dá-se a situação do dicionário estar *cheio*. Aqui pode-se seguir vários caminhos:

- Ignora-se a situação e não se insere a nova palavra. Esta abordagem, embora simples, tem o inconveniente de conduzir à degradação do desempenho pois novos strings que venham a repetir-se não vão poder ser codificados. Esta situação é notável quando os dados mudam de natureza a meio (exemplo: um ficheiro .tar.gz).
- Deita-se fora o dicionário existente e recomeca-se com um novo. Esta abordagem pode ser eficaz mas tem o inconveniente de perder toda a informação préviamente colecionada.
- Uma abordagem possível é um compromisso entre as anteriores: não se faz nada enquanto não fôr observável uma degradação da compressão, altura em que se re-inicializa o dicionário.

Em 1984, o Terry Welch melhorou o algoritmo LZ78 para produzir o conhecido algoritmo LZW: o dicionário, em vez de ser inicialmente vazio, é pré-carregado com todos os strings de comprimento 1; esta operação melhora o desempenho nos casos em que os dados a comprimir não têm muitas repetições. No algoritmo LZW há ainda melhorias na comunicação entre compressor e descompressor, nomeadamente no tocante à modulação do tamanho do dicionário (implicações ao nível do tamanho dos índices) ou à purga parcial ou total do dicionário, que passam a ser acções possíveis para além da que corresponde à codificaçao de texto própriamente dita.

É frequente a utilização desta família de algoritmos, sendo que o LZW, por ser patenteado, práticamente não é utilizado.

Desempenho: estes algoritmos são os que dão melhores ganhos de compressão em média. Os mesmos estudos que foram feitos para os códigos de Huffmann apontam no caso dos algoritmos LZ para ganhos de compressão na ordem de 45 a 50%.

Discussão das características de complexidade destes algoritmos.

7.4 Outros métodos de compressão

7.4.1 Codificação por repetição: Run-length Encoding (RLE)

Métodos simples frequentemente usados para informação com forte correlação entre símbolos adjacentes. É o caso de gráficos, especialmente se de síntese ou se representados com poucos valores distintos possíveis para as cores.

O RLE consiste na emissão dum par (número de ocorrências, valor) para cada símbolo. Tem a vantagem de ser muito simples de codificar e descodificar, sendo no entanto susceptível de gastar muito espaço se (por exemplo) os dados consistirem numa sequência alternada de dois símbolos diferentes.

O RLE pode ser melhorado pontualmente, e é, resultando em formatos relacionados com grafismos. Exemplos são os formatos PCX e TIFF, o FAX ou os Photo-CD e CD-I. Alguns destes formatos combinam o RLE com outro algoritmo, nomeadamente os códigos de Huffmann ou um LZ.

7.4.2 Compressão com degradação

Estas formas de compressão são normalmente usadas em contextos 'multimedia', onde se pode *perder* informação para ganhar no tamanho; é esta a situação nos formatos JPEG e MPEG.

Estes algoritmos não serão aqui descritos pois são do âmbito doutras disciplinas.

7.4.3 Compressão predictiva

Quando os dados a codificar são valores numéricos que representam uma grandeza que evolui no tempo, podem-se utilizar formas de codificação que se assemelham a métodos de modulação de sinal (estas técnicas tiveram como base a teoria dos sinais, uma disciplina de telecomunicações). Assim, partindo do princípio que se está a codificar uma *amostragem* dum sinal, pode-se representar o sinal não como o seu valor mas sim como um *incremento* relativamente ao valor anterior. Teremos assim a modulação diferencial ou delta-modulação.

Novamente, o resultado dum sinal codificado nesta modalidade pode ser sujeito a compressão usando um dos outros algoritmos.

Estes algoritmos não serão aqui descritos pois são do âmbito doutras *licenciaturas*.

7.5 Exercícios

8 Enunciados dos Trabalhos

Os trabalhos devem ser efectuados por grupos de no máximo 3 elementos e consistem num programa (código fonte) que resolva o problema escolhido, **obrigatóriamente** acompanhado de:

1. Um relatório no formato HTML (também se aceita PDF), com um máximo de 1500 palavras. Para contar as palavras dum documento HTML podem-se usar os comandos `unhtml` e `wc -w` (disponíveis na `alunos.uevora.pt`).
2. Um ou mais ficheiros de dados para exemplificar o funcionamento do programa.
3. Uma `Makefile` com os seguintes *targets*:
   - `all` para compilar e linkar o programa,
   - `test` para executar o programa sobre um conjunto de dados disponibilizado pelo grupo,
   - `clean` para remover todos os ficheiros objecto, executáveis e outros ficheiros gerados ou temporários

Os programas e o relatório devem ficar disponíveis no Web, **cifrados** pelo PGP (ver <http://www.pgp.net/> com a chave pública do responsável da disciplina (usar o ID `spa@di.uevora.pt`, a chave pública pode ser obtida em <http://home.di.uevora.pt/~spa/pubkey.asc>, na máquina `alunos.uevora.pt`, com um URL da forma `http://alunos.uevora.pt/~lXXXXX/trab-lp.tar.gz.pgp` em que XXXXX é o número dum dos alunos de cada grupo.

1. **Diagrama de Gantt.**

   Pretende-se fazar a análise da execução de um projecto, constituido por várias tarefas, recorrendo a um diagrama de Gantt. Em particular, o programa deve produzir informação sobre:

   - Prazos mínimos e máximos de execução (para o projecto na sua totalidade).
   - Caminho crítico (uma lista de tarefas).
   - Folgas de cada tarefa (por tarefa).

   Para isso deve-se construir um programa que aceite como input (stdin) linhas do tipo: `tarefa duracao precedencias`

   Onde `tarefa` é um identificador para uma tarefa (uma sequência não vazia de caracteres alfanuméricos ou o símbolo `'-'`), `duracao` é um numero inteiro que indica a duração da tarefa (em dias, ou meses se o nome da tarefa começar por "ein-") e `precedencias` é uma lista com zero ou mais identificadores de tarefas, separados por espaços em branco, que antecedem (ie. têm de estar completadas antes) a tarefa definida.

   O output deverá poder apresentar toda a informação solicitada, sendo a natureza do resultado determinada por argumentos de linha de comando: `--time` resulta nos prazos mínimos e máximos. `--cpath` resulta no caminho crítico. `--slack` resulta nas folgas. Estes argumentos podem ser múltiplos, caso em que vários outputs devem ser produzidos.

2. **Grafo de chamadas.**

   Um *grafo de chamadas* (call-graph) é uma relação binária representada por um grafo dirigido em que os nós representam procedimentos e um arco do nó A para o nó B representa o facto que o procedimento A *chama* o procedimento B. Pretende-se fazer um programa que, dada a relação de chamada, representada no stdin como uma sequência de linhas com dois identificadores (de procedimento) em cada uma, sendo o primeiro o A e o segundo o B, escreva no stdout a seguinte informação:

   - Para cada identificador, o conjunto de procedimentos que este invoca directa ou indirectamente (aqueles de que depende).
   - Para cada identificador, o conjunto de procedimentos que o invocam directa ou indirectamente (os "chamadores").
   - Uma classificação de cada procedimento com uma indicação de qual a *profundidade mínima* a que este se encontrará numa árvore de chamadas. Assuma que *tem de existir* um procedimento de nome `main`, cuja profundidade é zero.
   - Uma relação dos procedimentos que sejam directa ou indirectamente recursivos.
   - Uma relação dos procedimentos que não sejam atingíveis desde o `main`.
   - Uma lista dos procedimentos, por uma ordem que garanta que, se um procedimento A chama um procedimento B, teremos B listado *antes* de A.

3. **Edição sequencial.**

   Pretende-se fazer um programa que leia do stdin e escreva no stdout um texto, efectuando as substituições indicadas pelos argumentos da linha de comando. O seu programa deverá:

   - Ser utilizado com uma linha de comando `PROG orig1 subst1 orig2 subst2 ... origN substN`. Em que a leitura é "substituir todas as ocorrências de `orig1` por `subst1`", etc...

- Deve permitir substituições como `A B B A`, com o entendimento que se quer trocar os `A`s com os `B`s, pelo que as substituições deverão ser feitas em paralelo.

Implemente *pelo menos duas* versões do referido programa (usando para tal algoritmos diferentes) e compare-as entre si, quer do ponto de vista formal (análise de complexidade assimptótica) quer experimental (escolha várias situações de teste e analise o desempenho verificado).

4. **Códigos de Huffmann.**

   Implemente um programa `huff` que leia o stdin e o comprima usando códigos de Huffmann (o resultado deverá ser escrito no stdout). Complemente o seu programa com outro que efectue a operação inversa. Compare o tempo de execução e a taxa de compressão obtida pelo seu programa com os do programa `gzip` disponível no Linux.

   O programa `PKZIP` no Windows/DOS cria um arquivo de ficheiros individualmente comprimidos. Compare esta abordagem com a habitual no Unix de comprimir um arquivo como um todo: diga quais as vantagens e inconvenientes de cada um.

5. **Árvore Genealógica.**:

   Deverá criar procedimentos que gerem a árvore genealógica, isto é, o indivíduo A é filho de B, o indivíduo C casa com D, etc. Após a geração da árvore, terá de criar funções base de pesquisa na árvore criada com por exemplo PaiDe, IrmaoDe, CasadoCom, que devolvem conjuntos de elementos na árvore que satisfazem as referidas condições. Assim PaisDe devolve o conjunto de duas pessoas Pai, Mãe. IrmaoDe devolve o conjunto de todos os irmãos do argumento.

   O utilizador poderá a partir de então criar novas relações entre pessoas através da sua definição partindo das funções anteriormente definidas. CunhadoDe(X) é conjunto dos irmaos de EsposaDe(X), etc.

6. **Diccionário de palavras.**

   Parte 1: Deverá fazer um diccionário em que sempre que possível, partes de cada palavra que possam ser partilhadas, tenham o mesmo apontador. Por exemplo as palavras que terminam em -mente, deverão sempre que possível evitar a duplicação das sub-árvores com o mesmo sufixo.

   Parte 2: Em seguida deverá fazer uma função de pesquisa que no caso de não encontrar uma palavra no diccionário, devolva um conjunto de palavras semelhantes à palavra não encontrada. (Refs: Aho, Hopcroft, Ullman "Data Structures and Algorithms", Addison-Wesley pp163-169.)

7. **Simplificação duma expressão.**

   Dada uma expressão aritmética com inteiros, variáveis de uma só letra e operadores unários (-), e binários (+, -, *, e /), escrever uma nova expressão simplificada (ou representação) nos quais foram aplicadas algumas regras de simplificação, como:

   (a) `x * 0 = 0`

   (b) `--x = x`

   (c) `4*5+3=23`

   (d) `(a/b)*(1/c)=a/(b*c)`

   O objectivo final é obter o menor número de nós na árvore possível. (Ref: Barret and Couch "Compiler Construction: theory and practice", SRA 1979, pp553-562.)

8. **Construção dum DAG.**

   Dada uma expressão aritmética, construir um DAG, no qual a representação de sub-expressões comuns partilhem as mesmas estruturas de dados. (Ref: Barret and Couch, mesma que em 8; Aho and Ullman "Principles of compiler design", Addison-Wesley 1979, pp418-428, pp537-540, pp547-548.)

9 Bibliografia

1. Salvador Pinto Abreu, *Apontamentos para Análise e Desenho de Algoritmos*, Universidade de Évora, 1999, <http://home.di.uevora.pt/~spa/aulas/1998-99/s2/ada/>. **(Referência Principal)**
2. Thomas Cormen, Charles Leiserson and Ronald Rivest, *Introduction to Algorithms*, MIT Press, 1990, ISBN 0-262-53091-0.
3. Gilles Brassard and Paul Bratley, *Fundamentals of Algorithmics*, Prentice Hall, 1996, ISBN 0-13-073487-X.
4. Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, 1974, Addison-Wesley, ISBN 0-201-00029-6.
5. Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, *Data Structures and Algorithms*, 1987, Addison-Wesley, ISBN 0-201-00023-7.
6. Phil Cornes, *The Linux A-Z*, Prentice Hall, 1997, ISBN 0-13-234709-1.
7. Kernighan and Ritchie, *The C Programming Language*, Prentice-Hall, 1978, ISBN XXX.
8. Ellis Horowitz, Sartaj Sahni, Susan Anderson-Freed, *Fundamentals of Data Structures in C*, W.H.Freeman, 1993, ISBN 0-7167-8250-2.

10 Avaliação

Duas frequências (primeira com peso 1 e segunda com peso 2) ou um exame (peso 3). Trabalho de grupo obrigatório (peso 1).

Cada grupo terá o máximo de dois elementos. Os enunciados serão divulgados a meio do semestre. A data limite de entrega do trabalho é a da segunda frequência (coincide com o exame). Não se aceitam trabalhos entregues depois dessa data. A nota de trabalho será contada no caso de o estudante se submeter a exame em época de recurso.