Algoritmos e Estruturas de Dados I - Universidade Fernando Pessoa
José Manuel Torres
Ficha 1: Fundamentos de algoritmos e de análise algorítmica

1. Indique qual é a aproximação til (~) das seguintes expressões função de N:
   1.1. $N + 1$
   1.2. $1 + 1/N$
   1.3. $(1 + 1/N)(1 + 2/N)$
   1.4. $2N^3 - 15N^2 + N$
   1.5. $\lg(2N)/\lg(N)$
   1.6. $\lg(N^2 + 1)/\lg(N)$

2. Obtenha a ordem de crescimento (como função de $N=2^k$, para um inteiro k) dos tempos de execução de cada um dos seguintes fragmentos de código. Considere como modelo de custo a instrução de incremento da variável sum. Construa um cliente de teste que dobra o valor de N, obtenha o gráfico log-log e o declive da regressão linear:
   2.1.
```
int sum = 0;
for (int i = N; i > 0; i /= 2)
   for (int j = 0; j < i; j++)
         sum++;
```
   2.2.
```
int sum = 0;
for (int i = 1; i < N; i *= 2)
   for(int j = 0; j < i; j++)
      sum++;
```
   2.3.
```
int sum = 0;
for (int i = 1; i < N; i *= 2)
   for (int j = 0; j < N; j++)
         sum++;
```

3. Considere o problema 3-*sum*. Dados N inteiros distintos, determinar quantos triplos é que somam exatamente para zero. Considere ainda as soluções ThreeSum e ThreeSumFast fornecidas.
   3.1. Estude, para vários tamanhos de vetor, $N \in \{250, 500, 1K, 2K, 4K \text{ e } 8K\}$, o tempo que demora a resolver o problema para cada uma das implementações. Registe os dados obtidos numa tabela e desenhe as duas curvas.
   3.2. Acrescente colunas à tabela onde deverá transformar o gráfico para log-log aplicando a função logarítmica nas abcissas e nas ordenadas. Aplique regressão linear aos dados obtidos e determine os respetivos coeficientes. O que conclui relativamente à eficiência.

4. Seja $f()$ uma função monotonamente crescente com $f(0) < 0$ e $f(N) > 0$. Encontre o inteiro mais pequeno i tal que $f(i) > 0$. Escreva um algoritmo que faça O(log N) chamadas a $f()$[1]. Avalie empiricamente (por observação) o desempenho temporal da sua solução para arrays com diferentes tamanhos de entrada. Deverá, nesta avaliação, usar contagem de instruções para avaliar o algoritmo. Considere dois casos na avaliação: *average case* e *worst case*.

5. *Floor* e *ceiling*: Dado um conjunto de elementos comparáveis e um elemento x, o *ceiling* de x é o menor valor desse conjunto que é maior ou igual a x, e o *floor* é o maior elemento do conjunto que é menor ou igual a x. Suponha que tem um vetor de N elementos ordenado ascendentemente. Escreva um algoritmo O(log N) que determine o *floor* e o *ceiling* de x.

6. Mostre que o número de diferentes triplos que podem ser escolhidos de N elementos é precisamente $N(N-1)(N-2)$. Dica: pode usar indução matemática.

7. Escreva um programa que, dados dois arrays ordenados de tamanho N, imprima todos os elementos resultantes da reunião de ambos, de um modo ordenado. Cada elemento deve ser impresso apenas uma vez. O tempo de execução do programa deve ser proporcional a N no pior caso.

[1] O modelo de custo (*cost model*) utilizado é o número de acessos à tabela.

8. Escreva um programa para determinar, num ficheiro de entrada (array), o número de pares de valores que são iguais. Se a sua primeira solução for quadrática, tente, usando ordenação, torná-la linear logarítmica.

9. Suponha que num problema algorítmico o tamanho dos dados de entrada, N, é aproximadamente igual a 1 milhão. Quantas vezes é mais rápido um algoritmo que executa em $N \lg N$ versus outro que executa em N^2? Recorde que a função lg é de base 2.
   9.1. 20×
   9.2. 1000×
   9.3. 50000×
   9.4. 1000000×

10. Para cada uma das seguintes funções indique se é: $O(N^3)$, $\Theta(N^3)$, $\Omega(N^3)$.
   10.1. $11N + 15\lg N + 100$
   10.2. $(1/3)N^2$
   10.3. $25000N^3$
   10.4. $100N^4 + 5N^2$

11. Escreva um algoritmo (em notação pseudo-C ou pseudo-Java) que:
   11.1. Dados dois vetores ordenados de tamanho N, imprima todos os elementos que aparecem em ambos os arrays, de um modo ordenado. O tempo de execução do programa deve ser proporcional a N no pior caso.
   11.2. Escreva um algoritmo, em tempo N, que para as mesmas condições de entrada descritas anteriormente, imprima todos os elementos que aparecem apenas num dos arrays.

12. Aplique, ao algoritmo de pesquisa binária na sua versão iterativa, um estudo de desempenho temporal do mesmo através da contagem do número de vezes que o ciclo while é executado.
   12.1. Use vários tamanhos de vetor, $N \in \{250, 500, 1K, 2K, 4K \text{ e } 8K\}$. Registe os dados obtidos numa tabela e desenhe a curva. Faça o estudo respetivo para o caso médio (*average case*) e para o pior caso (*worst case*).
   12.2. Acrescente colunas à tabela onde deverá transformar o gráfico para log-log aplicando a função logarítmica nas abcissas e nas ordenadas. Aplique regressão linear aos dados obtidos e determine os respetivos coeficientes. O que conclui relativamente à eficiência.

13. Escreva um programa que, para um dado valor de d, inteiro, deve gerar uma sequência aleatória de inteiros (extração com repetição) compreendidos entre 0 e d-1 até que seja gerado o primeiro inteiro repetido na extração n. Verifique experimentalmente a hipótese de que o número n é ~ sqrt(2.d.ln(2)). (*generalized birthday problem*[2])

14. Considere um trecho de código em que o modelo de custo considerado na análise temporal do mesmo é a instrução, `s++;`, de incremento de uma variável inteira s. Escreva uma versão possível desse trecho para o caso que a análise temporal é dada por cada uma das seguintes famílias de funções que ignoram os fatores constantes e os termos de menor ordem:
   14.1. $\Theta(N^2)$
   14.2. $\Theta(N)$
   14.3. $\Theta(N.\lg(N))$
   14.4. $\Theta(\lg(N)^2)$

15. Considere um array w[] de inteiros de tamanho M e um segundo array a[] de inteiros de tamanho N. Sabendo que problema algorítmico consiste em imprimir todos os elementos de a[] que estão em w[], discuta abordagens possíveis e a sua implementação para vários cenários possíveis.
   15.1. Considere que o array w[] está ordenado w e o array a[] desordenado. Discuta implementações para vários cenários possíveis como: i) M >> N; ii) N >> M; iii) M e N da mesma ordem de grandeza.
   15.2. Nos cenários anteriores, discuta implementações no caso de w[] e a[] estarem ambos desordenados.

16. Construa um cenário de teste funcional e não funcional ao algoritmo de pesquisa binária, na sua versão iterativa, do seguinte modo: i) faça a geração de vários arrays a[] de tamanhos variáveis $N = \{500, 1000, 2000, 4000, ...\}$, ordenados, e sem repetição de elementos ii) escolha chaves *key* sequencialmente de a[0] até a[N-1] e use-as como chave de teste da pesquisa binária iii) usando pesquisa binária determine qual o índice *idx* do array que corresponde à chave *key* passada para o algoritmo iv) se *key* == a[idx] o teste foi bem sucedido v) acrescente ao teste, em cada iteração, N/2 chaves que não existem para testar a não existência da chave para validar de um modo mais sustentado o algoritmo de pesquisa binária vi) analise o comportamento temporal do teste como um todo através de observação e através de análise de complexidade.

[2] https://en.wikipedia.org/wiki/Birthday_problem#The_generalized_birthday_problem

Algoritmos e Estruturas de Dados I - Universidade Fernando Pessoa
José Manuel Torres
Ficha 2: Casos de estudo de algoritmos

Caso de Estudo Union-Find

1. Implemente o problema de *union-find* usando a estrutura *quick-find* (QF).

2. Qual é o máximo número de entradas no array que poderão mudar durante uma chamada à função *union* usando a estrutura de dados *quick-find* com *N* elementos? (a) 1; (b) lg*N*; (c) *N*-1; (d) *N*

3. Implemente o problema QuickUnionUF usando a estrutura quick-union. Implemente a versão base (QU) e a versão melhorada com weight (WQU).

4.
   4.1. Mostre o conteúdo do array id[] e o número de vezes que o array é acedido para cada par de input quando se usa a estrutura quick-find para a sequência de pares de input: 9-0 3-4 5-8 7-2 2-1 5-7 0-3 4-2.
   4.2. Repita o exercício anterior mas desta feita para a estrutura quick-union. Desenhe, ainda, a floresta de árvores representada pelo array id[] depois de cada par de input ser processado.
   4.3. Repita o exercício anterior, mas use a estrutura weighted quick-union.

5. Calcule qual o array id[] resultante depois de efetuadas as 6 operações de union num conjunto com *N*=10 itens usando o algoritmo quick-find.

   0-7 4-3 2-5 3-9 7-2 6-5

   Nota: a convenção adotada para a operação de *union* p-q na estrutura *quick-find* é alterar id[p] (e eventualmente outras entradas do mesmo componente) mas não id[q].

6. Calcule qual o array id[] resultante depois de efetuadas as 9 operações de union num conjunto com *N*=10 itens usando o algoritmo weighted quick-union.

   6-5 1-0 6-1 4-2 2-8 2-3 8-0 4-7 9-2

   Nota: quando juntar duas árvores de tamanho igual, o weighted quick union (por convenção) coloca a raiz da segunda árvore a apontar para a raiz da primeira árvore.

7. Dos seguintes arrays id[], quais é que poderiam ser o resultado da execução do algoritmo weighted quick union num conjunto de 10 itens?
   7.1. 3 3 2 3 4 5 0 7 8 9
   7.2. 5 2 2 8 6 6 2 2 2 2
   7.3. 7 0 1 0 1 7 1 7 7 5
   7.4. 2 4 2 4 7 9 3 7 4 8
   7.5. 8 9 8 1 8 8 2 9 3 8

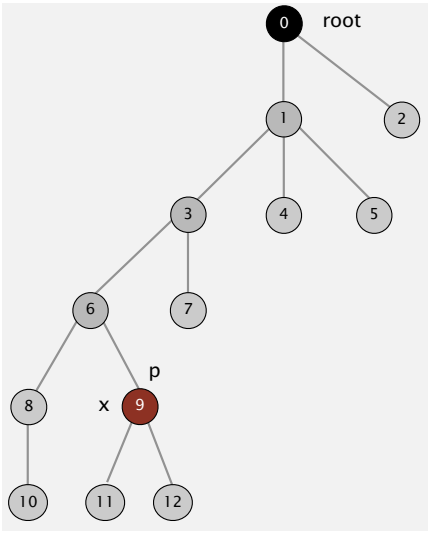
8. Desenhe a árvore correspondente ao array id[] seguinte. Pode este array ser o resultado da execução do algoritmo weighted quick-union? Explique porque é que, caso assim seja, é impossível ou, caso seja possível, indique a sequência de operações que resultam neste array.

i	0	1	2	3	4	5	6	7	8	9
id[i]	1	1	3	1	5	6	1	3	4	5

9. No algoritmo weighted quick-union, suponha que atribuímos *id*[*find*(*p*)] = *q* em vez de *id*[*find*(*p*)] = *id*[*find*(*q*)]. O algoritmo resultante estaria correto?

10. Usando como modelo de custo o número de acessos aos arrays id[] e sz[], crie gráficos de custo das várias soluções implementadas para o problema UF. Use, como ficheiro de teste, o ficheiro de média dimensão (mediumUF.txt) com *N*=625 e M=900 operações de conexão.

11. Considere o conjunto union-find da figura seguinte com N=13 elementos. Admita que o conjunto está representado usando a solução quick-union com weighting.
    11.1. Escreva os arrays id[] e sz[] para esse conjunto.
    11.2. Implemente path compression com duas passagens e use este conjunto como caso de teste para o find(9). (Nota: find com quick-union, weighting e path compression de duas passagens)



Caso de Estudo: maior soma entre as subquências duma sequência (Maximum subarray problem)

12. Implemente vários algoritmos de cálculo da maior soma de entre todas as possíveis subsequências possíveis duma sequência (array) de *N* valores positivos e negativos, não ordenados.

13. Para sequências de entrada de vários tamanhos de *N*, corra os vários algoritmos e crie e analise tabelas e gráficos com os resultados dos tempos de execução obtidos.

Algoritmos e Estruturas de Dados I - Universidade Fernando Pessoa
José Manuel Torres
Ficha 3: Ordenação de Strings

1. Escreva um cliente para testar o algoritmo de *key-indexed counting sort* que considere a seguinte tabela de input:

```
char * strings[] = {"Ana","Carlos","Daniela","Filipe","Hugo","Luis","Manuel","Pedro","Ricardo","Rui"};
int keys[] = {3,2,3,1,2,3,2,3,3,1};
```

   1.1. Escreva uma versão do cliente em que os dados estão programados diretamente no código do programa (*hard-coded*) em constantes.
   1.2. Escreva uma versão do cliente em que os dados estão num ficheiro de dados e o cliente deve carregar esse ficheiro de dados e instanciar a estrutura que é passada para o algoritmo com os dados lidos entretanto.

2. Implemente uma versão do algoritmo key_index_counting que aceite como entrada várias colunas de potenciais chaves para ordenação para comutar de uma para outra em tempo linear de modo a que seja possível fazer a impressão ordenada pela chave/coluna que se desejar. Escreva um cliente para testar a função. A declaração da função deverá ser: void key_index_counting2(StringElementsArray * a, int * keys_cols[], int N_cols, int sorting_col, int R); em que a coluna usada para a ordenação será dada pelo parâmetro sorting_col e o número de colunas por N_cols.

3. Faça o rastreio para o algoritmo de ordenação de strings LSD (*radix sort*) para a seguinte entrada:
   no is th ti fo al go pe to co to th ai of th pa

4. Faça o rastreio para o algoritmo de ordenação de strings MSD para os seguintes casos de entrada. Para cada chamada recursiva do algoritmo de (MSD) sort, com os respectivos parâmetros *lo*, *hi* e *d*, apresente o estado do array *count*[] depois de calculadas as frequências absolutas acumuladas:
   4.1. no is th ti fo al go pe to co to th ai of th pa (N=16, W=2 fixo)
   4.2. verde sol chuva solar tarde sol parar ver verruga parar (N=10, W_i variável, para $0 \leq i < 10$)
   4.3. tarde tanque sapo sal santo salto tanto tio tinto salmo (N=10, W_i variável, para $0 \leq i < 10$)

5. Faça o rastreio para o algoritmo de ordenação de strings MSD para a seguinte entrada:
   5.1. now is the time for all good people to come to the aid of (N=14, W_i variável, para $0 \leq i < 14$)

6. Faça o rastreio para o algoritmo de ordenação de 3-way string (radix) quicksort (quicksort3way) para a seguinte entrada. Considere como elemento pivot (*partitioning item*) o primeiro elemento de cada (sub)tabela a ordenar:
   6.1. now is the time for all good people to come to the aid of (N=14, W_i variável, para $0 \leq i < 14$)

7. Implemente uma versão do algoritmo MSD que considere um parâmetro M de cuttof para utilizar o algoritmo de insertion sort para pequenos arrays.

8. Escreva uma função utilitária que faça a geração aleatória de:
   8.1. Matriculas (fictícias) de automóveis portugueses e que as escreva para um ficheiro de texto.
   8.2. Endereços IP e que os escreva para um ficheiro de texto.

9. Escreva um cliente que faça a ordenação:
   9.1. De matriculas (fictícias) de automóveis portugueses. As matrículas deverão estar guardadas num ficheiro.
   9.2. De endereços IP que deverão estar guardados num ficheiro.

10. Escreva um cliente que faça a ordenação (LSD) de uma lista de inteiros de 32 bits em 4 passagens (W=4) através da divisão do inteiro em 4 bytes em que cada byte corresponde a um dicionário de R=256 símbolos. Leia os inteiros de um ficheiro e escreva o resultado para um ficheiro.

11. Conceba uma representação em string para cartas de jogar (4 naipes e 13 cartas por naipe) que seja apropriada para a ordenação por LSD e por MSD.

12. Considere as seguintes 8 palavras de 6 bits: 010101, 110101, 001101, 111000, 101111, 011110, 111010, 001100. Considere que os símbolos do alfabeto utilizado na ordenação são representados por sequências de 2 bits.
   12.1. Indique o valor de N, W e R. Indique qual é o alfabeto. Justifique.

12.2. Faça o rastreio para o algoritmo de ordenação de strings LSD. Apresente, para iteração do ciclo mais exterior, o estado do array de contagens absolutas e acumuladas.
12.3. Implemente uma função cliente para resolver o problema de ordenação apresentado.

13. Considere as seguintes 8 notas na escala de 0 a 20: 12, 20, 6, 13, 7, 10, 14, 13. Considere duas possibilidades de interpretação dos dados de entrada: i) o alfabeto é constituído pelos algarismos 0...9. ii) o alfabeto é constituído pelas notas de toda a escala 0...20. Para cada uma das possibilidades:
13.1. Indique o valor de N, W e R. Indique qual é o alfabeto. Justifique.
13.2. Faça o rastreio para o algoritmo de ordenação de strings LSD. Apresente, para iteração do ciclo mais exterior, o estado do array de contagens absolutas e acumuladas.
13.3. Implemente uma função cliente para resolver o problema de ordenação apresentado.

14. Implemente uma solução algorítmica, usando MSD sorting, que dado um texto genérico, faça a contagem de frequências absolutas das várias palavras que compõem o texto.

15.
15.1. Descreva um algoritmo que, dados n inteiros na gama 0 a k, pré-processe o seu input e depois responda à pergunta acerca de quantos desses n inteiros caem na gama $[a..b]$ em tempo $O(1)$. O seu algoritmo de pré-processamento deverá usar $\Theta(n + k)$.
15.2. Faça o rastreio desse mesmo algoritmo de pré-processamento para k=3 e n=15 com os seguintes valores v[]={2,1,0,3,3,3,3,2,2,1,0,0,0,0,0}. Usando o algoritmo anterior mostre como poderia calcular, para este caso, quantos dos n inteiros do *array* estão compreendidos entre 1 e 2.

16.
16.1. Mostre como será possível ordenar n inteiros na gama 0 até n^3-1 em tempo $O(n)$.
16.2. Aplique o algoritmo anterior ao *array*, v[] = {20, 350, 500, 900, 450, 342, 682, 1310, 1122, 1700, 1528, 1444}, com n=12 valores. Nota: converta os valores do *array* para base 12 (n=12) e considere os inteiros mais pequenos com zeros à esquerda de modo a que todos os inteiros tenham o mesmo número de dígitos na nova base de destino.

Algoritmos e Estruturas de Dados I - Universidade Fernando Pessoa
José Manuel Torres
Ficha 4: Pesquisa de Substrings em Strings

1.
   1.1. Implemente em C uma função char * strsearch_brute_force(char * txt, char * pat) que pesquisa um padrão de texto ou substring (char * pat) dentro dum texto maior (char * txt). A função deve retornar um apontador para a primeira ocorrência de pat em txt ou 0 caso não seja encontrada nenhuma ocorrência.
   1.2. Implemente uma função que, com base na função anterior, conte o número de ocorrências duma substring numa string.
   1.3. Implemente uma função que, com base na primeira função, imprima todas as ocorrências (posições) duma substring numa string.

2. Faça o rastreio da pesquisa dum padrão pat[] num texto txt[] usando o algoritmo de pesquisa exaustiva (*brute-force*), indicando para cada posição *i* do texto txt[] o índice *j* do caracter do padrão pat[] de pesquisa que será comparado com txt[*i*].
   2.1. pat=" AAAAAAAB", txt=" AAAAAAAAAAAAAAAAAAAAAAAB"
   2.2. pat=" ABABABAB", txt=" ABABABABAABABABABAAAAAAAA"

3.
   3.1. Implemente em C uma função char * strsearch_kmp(char * txt, char * pat) que pesquisa um padrão de texto ou substring (char * pat) dentro dum texto maior (char * txt) usando o algoritmo KMP. A função deve retornar um apontador para a primeira ocorrência de pat em txt ou 0 caso não seja encontrada nenhuma ocorrência.
   3.2. Implemente uma função de criação da tabela com a máquina de estados dfa[R][M] para um dado padrão pat[M], em que R é o tamanho do alfabeto (radix) e M é o tamanho do padrão que se deseja procurar.

4. Determine a tabela de máquina de estados para os seguintes padrões com *radix* R=3, {A,B,C}.
   4.1. pat="ABAAC"
   4.2. pat="AAAAAAAB"
   4.3. pat="AACAAAB"
   4.4. pat="ABABABAB"
   4.5. pat="ABAABAAABAAAB"
   4.6. pat="ABAABCABAABCB"

5. Faça o rastreio da pesqu
6. isa dum padrão pat[] num texto txt[] usando o algoritmo KMP, indicando para cada posição *i* do texto txt[] o índice *j* do caracter do padrão pat[] de pesquisa que será comparado com txt[*i*].
   6.1. pat="ABAAC", txt="ACAABAACAABAC"
   6.2. pat="AAAAAAAB", txt="AAAAAAAAAAAAAAAB"
   6.3. pat="AACAAAB", txt=" AAAAACAAABAAA"

7. Escreva um programa que, dadas duas strings, determine se uma é uma rotação cíclica da outra, tal como as palavras "exemplo" e "emploex".

8. Uma repetição em tandem de uma string base *b* numa string *s* é uma substring de *s* tendo, pelo menos, duas cópias consecutivas de *b* não sobrepostas. Escreva um algoritmo linear que dadas duas strings *b* e *s*, retorne o índice do inicio do tandem mais longo de *b* em *s*. Por exemplo, deve retornar 3 se *b*="abcab" e *s*="abcabcababcababcababcab".

9. Escreva um cliente que aceite os inteiros M, N e T como entradas e corra o seguinte teste T vezes: i) gere um padrão aleatório de comprimento M e um texto aleatório de comprimento N; ii) conte o número de comparações usado pelo KMP para pesquisar o padrão no texto. Para tal deve alterar a função KMP para fornecer o número de comparações e no final imprimir o valor médio de comparações efetuadas nas T vezes.

Algoritmos e Estruturas de Dados I - Universidade Fernando Pessoa
José Manuel Torres
Ficha 5: Estruturas de dados lineares

Pilhas (*Stacks*) e Filas (*Queues*)

1. Implemente uma *stack* de inteiros e as suas principais operações usando:
   1.1. Um *array* estático.
   1.2. Um *array* redimensionável.
   1.3. Escreva procedimentos de teste de cada uma das implementações.

2. Implemente uma *queue* de inteiros e as suas principais operações usando:
   2.1. Um *array* estático.
   2.2. Um *array* redimensionável.
   2.3. Escreva procedimentos de teste de cada uma das implementações.

3. Considere uma *stack* (pilha), inicialmente vazia, e a seguinte sequência de operações de *push* e *pop*: 4, 1, 3, *, 8, * (um inteiro representa uma operação de *push* desse elemento e um asterisco representa uma operação de *pop*).
   3.1. Represente o conteúdo do *array* e estado da *stack*, passo a passo, no caso da implementação com um *array* estático de *size*=6.
   3.2. Repita a alínea anterior para *size*=4 e para a sequência: 5, 10, 15, *, 12, *, 18, *, *, *, 34, 22.
   3.3. Considere uma implementação da *stack* com uma lista ligada. Represente, passo a passo, o estado da lista.
   3.4. Considere uma implementação da *stack* com um *array* redimensionável. Represente, passo a passo, o estado da *stack* (política de redimensionamento: se $N == size$ então $size = 2 \times size$; se $N == size/4$ então $size = size/2$. Inicialmente *size*=2).

4. Considere uma *queue* (fila), inicialmente vazia, e a seguinte sequência de operações de *enqueue* e *dequeue*: 4, 1, 3, *, 8, * (um inteiro representa uma operação de *enqueue* desse elemento e um asterisco representa uma operação de *dequeue*).
   4.1. Represente o conteúdo do *array* e estado da *queue*, passo a passo, no caso da implementação com um *array* estático de *size*=6.
   4.2. Repita a alínea anterior para *size*=4 e para a sequência: 5, 10, 15, *, 12, *, 18, *, *, *, 34, 22.
   4.3. Considere uma implementação da *queue* com uma lista ligada. Represente, passo a passo, o estado da lista.
   4.4. Considere uma implementação da *queue* com um *array* redimensionável. Represente, passo a passo, o estado da *queue* (política de redimensionamento: se $N == size$ então $size = 2 \times size$; se $N == size/4$ então $size = size/2$. Inicialmente *size*=2).

5. Explique como será possível implementar duas *stacks* usando um array estático de *size*=*n* de modo que nenhuma das *stacks* atinja *overflow* a não ser que o número total de elementos em ambas as *stacks* seja *n*. As operações de *push* e *pop* devem ser executadas em tempo constante $O(1)$

6.
   6.1. Explique como será possível implementar uma fila (*queue*) com recurso a duas *stacks* em tempo tempo constante $O(1)$ amortizado[3].
   6.2. Usando esta abordagem simule as operações: 4, 1, 3, *, 8, * (um inteiro representa uma operação de *enqueue* desse elemento e um asterisco representa uma operação de *dequeue*).

7.
   7.1. Explique como será possível implementar uma pilha (*stack*) com recurso a duas filas (*queues*). Deve sugerir duas soluções possíveis, a solução A que privilegia a operação de *push* e a solução B que favorece a solução de *pop*.
   7.2. Usando cada uma das abordagens simule as operações: 4, 1, 3, *, 8, * (um inteiro representa uma operação de *push* desse elemento e um asterisco representa uma operação de *pop*).

8. Dada uma string de texto que representa uma expressão que usa parênteses do tipo "()[]{}", escreva um cliente usando uma stack (pilha) que verifica se os parênteses estão corretamente emparelhados na expressão.

[3] Se a operação for repetida muitas vezes em média será constante.

9. Dada uma expressão numérica representada em notação pós-fixa (RPN - *Reverse Polish notation*) numa string (esta notação dispensa o uso de parênteses), escreva um cliente usando uma stack (pilha) que avalie a expressão. Exemplos: 3 + 4 fica 3 4 +; 2 + 4 * 5 fica 2 4 5 * -

10. Suponha que numa stack são efetuadas uma sequência de operações intercaladas de push e pop de inteiros. Os push são feitos, por ordem, de 0 até 9 e nos pop o valor retornado é impresso no ecrã. Quais das seguintes sequências de output não poderiam ocorrer:

(a) 4 3 2 1 0 9 8 7 6 5
(b) 4 6 8 7 5 3 2 9 0 1
(c) 2 5 6 7 4 8 9 3 1 0
(d) 4 3 2 1 0 5 6 7 8 9
(e) 1 2 3 4 5 6 9 8 7 0
(f) 0 4 6 5 3 8 1 7 2 9
(g) 1 4 7 9 8 6 5 3 0 2
(h) 2 1 4 3 6 5 8 7 9 0

11. Suponha que numa queue são efetuadas uma sequência de operações intercaladas de enqueue e dequeue de inteiros. Os enqueue colocam na fila, por ordem, os inteiros de 0 até 9 e nos dequeue o valor retornado é impresso no ecrã. Quais das seguintes sequências de output não poderiam ocorrer:

(a) 0 1 2 3 4 5 6 7 8 9
(b) 4 6 8 7 5 3 2 9 0 1
(c) 2 5 6 7 4 8 9 3 1 0
(d) 4 3 2 1 0 5 6 7 8 9

Listas Ligadas

12. Considere a seguinte estrutura usada para implementar uma lista simplesmente ligada (*singly linked*):
`struct Node { int item; struct Node * next; };`
    12.1. Crie uma função `struct Node * criaLista(int * v, int n);` que recebe um vector de inteiros com n elementos e o converte para uma lista ligada e que devolve o endereço para o primeiro elemento da lista.
    12.2. Crie uma função `void imprimeLista(struct Node * lista);` que imprime todos os elementos da lista no ecrã.
    12.3. Crie uma função `struct Node * removeElementoLista(struct Node * lista, int k);` que remove o elemento k da lista e que devolve o endereço para a lista. Considere que o primeiro elemento da lista é o elemento zero (k=0).
    12.4. Implemente funções para inserção de um elemento e pesquisa de um elemento.
    12.5. Implemente as funções anteriores usando, desta feita, a seguinte estrutura usada para implementar uma lista duplamente ligada (*doubly linked*): `struct Node { int item; struct Node *prev, *next;};`

13. Implemente uma *stack* usando uma lista simplesmente ligada. Considere a seguinte estrutura de dados:
`struct StackLL { struct Node * first; };`

14. Implemente uma *queue* usando uma lista simplesmente ligada. Considere a seguinte estrutura de dados:
`struct QueueLL { struct Node *first, *last; };`

Algoritmos e Estruturas de Dados I - Universidade Fernando Pessoa
José Manuel Torres
Ficha 6: Introdução à Ordenação

1. Num *array* de chaves parcialmente ordenado ascendentemente, por exemplo num *array* de inteiros, uma inversão é um par de chaves que está fora da ordem. Exemplo, no *array* de letras (A, E, E, L, M, O, T, R, X, P, S), existem seis inversões que correspondem aos pares: T-R T-P T-S R-P X-P X-S. Escreva uma função *int inversions(int v[])* que dado um *array* de inteiros calcula o número de inversões no mesmo usando pesquisa exaustiva.

2. Liste as inversões existentes no *array* (2, 3, 8, 6, 1).

3. Considere o conjunto de permutações possíveis dum *array* com *n* inteiros distintos entre 1 e *n*. Considere o problema da contagem de inversões relativamente ao *array* ordenado ascendentemente: 1, 2, 3, ... , *n*.
   3.1. Qual é o *array* (permutação) que tem menos inversões. Quantas são? Justifique.
   3.2. Qual é o *array* (permutação) que tem mais inversões. Quantas são? Justifique.

4. Escreva uma função que verifique se um *array* de inteiros está ordenado. Escreva um cliente de teste para essa função.

5.
   5.1. Implemente uma versão do algoritmo de ordenação *selection sort*.
   5.2. Faça o rastreio da execução do *selection sort* ao *array* (31, 41, 59, 26, 41, 58)

6.
   6.1. Implemente uma versão do algoritmo de ordenação *insertion sort*.
   6.2. Faça o rastreio da execução do *insertion sort* ao *array* (31, 41, 59, 26, 41, 58)
   6.3. Reescreva uma nova versão do *insertion sort* para fazer a ordenação por ordem decrescente.

7.
   7.1. Implemente uma versão do *insertion sort* que considere um mecanismo de contagem do número trocas efetuadas pelo algoritmo durante a ordenação.
   7.2. Use essa versão para estudar o comportamento do algoritmo para *array* com diversos tamanhos *n* e com diversas configurações de entrada. Para cada tamanho do *array* considere as seguintes quatro configurações: aleatório, quase ordenado, ordenado inversamente, com muitos valores repetidos.

8. Analise um relacionamento entre o número de inversões e o tempo de execução do *insertion sort*.

9. Escreva uma função que implemente o algoritmo de *Knuth shuffle* para baralhar um *array* e criar:
   9.1. Uma permutação aleatória com base num array de *N* elementos distintos.
   9.2. Uma permutação aleatória de *k* elementos com base num *array* com *N* elementos distintos *P*(*n*,*k*).

10. Relativamente ao algoritmo de ordenação *shellsort* que usa, como algoritmo base, o *insertion sort*.
    10.1. Identifique quais os *subarrays* (e quantos) estão contidos no *array* (10, 9, 15, 17, 13, 23, 8, 38, 50, 47, 44, 41, 42, 19, 6, 3) com *n*=16, considerando *h*=7.
    10.2. Identifique quais os *subarrays* (e quantos) estão contidos no *array* (10, 9, 15, 17, 13, 23, 8, 38, 50, 47, 44, 41, 42, 19, 6, 3) com *n*=16, considerando *h*=4.
    10.3. Identifique quais os *subarrays* (e quantos) estão contidos no *array* (10, 9, 15, 17, 13, 23, 8, 38, 50, 47, 44, 41, 42, 19, 6, 3) com *n*=16, considerando *h*=3.

11. Aplique, passo a passo, o algoritmo de ordenação *shellsort*, indicando para cada valor de *h* o estado dos *subarrays*. Considere os seguintes arrays e sequências de *strides h*:
    11.1. (10, 9, 15, 17, 13, 23, 8, 38, 50, 47, 44, 41, 42, 19, 6, 3), *n*=16; *h* = 7, 3, 1.
    11.2. (10, 9, 15, 17, 13, 23, 8, 38, 50, 47, 44, 41, 42, 19, 6, 3), *n*=16; *h* = 13, 4, 1.
    11.3. (10, 9, 15, 17, 13, 23, 8, 38, 50, 47, 44, 41, 42, 19, 6, 3), *n*=16; *h* = 4, 2, 1.

Algoritmos e Estruturas de Dados I - Universidade Fernando Pessoa
José Manuel Torres
Ficha 7: Mergesort e Divide-and-Conquer

1. Dados os seguintes *arrays* aplique, a cada um deles, passo a passo, o algoritmo de ordenação por fusão (*merge sort*) nas versões top-down e bottom-up. Para cada passo indique, claramente, qual a parte do *array* em que o procedimento de fusão está a operar. Na versão top-down desenhe a árvore de recursão e na versão bottom-up indique qual o valor da variável *sz*.
   1.1. *a*[] = 17, 9,22,11,9,23,19,22,24,9,28,5,17,20,16,9 (*n*=16)
   1.2. *a*[] = 10, 35, 60, 50, 24, 12, 5, 2, 20, 72, 21, 14, 41, 44, 7, 6 (*n*=16)
   1.3. *a*[] = 100, 35, 6, 3, 2, 50, 24, 12, 5, 22, 20, 72, 21, 14, 41, 44, 7, 6, 48, 37 (*n*=20)

2.
   2.1. Implemente uma versão base do algoritmo *merge sort* na versão *top-down*.
   2.2. Faça o rastreio, no computador, da execução do *merge sort* ao *array* (17, 9, 22, 11, 9, 23, 19, 22, 24, 9, 28, 5, 17, 20, 16, 9)

3.
   3.1. Implemente uma versão base do algoritmo *merge sort* na versão *bottom-up*.
   3.2. Faça o rastreio, no computador, da execução do *merge sort* ao *array* (17, 9, 22, 11, 9, 23, 19, 22, 24, 9, 28, 5, 17, 20, 16, 9)

4. Implemente uma versão do algoritmo *merge sort* na versão *top-down* com os 3 melhoramentos seguintes:
   4.1. Usar insertion sort para subarrays pequenos (cuttof = 10);
   4.2. Paragem em caso de array já ordenado;
   4.3. Eliminar necessidade de cópia do array auxiliar no processo de fusão

5. Suponha que a versão merge-sort top-down é modificada para saltar a chamada de merge() quando a[mid] <= a[mid+1] (um dos melhoramentos possíveis do top-down). Prove que o número de comparações efetuadas num array de input já ordenado é linear para esse caso.

6. Dado um conjunto com *n* inteiros distintos:
   6.1. Indique quantas sequências distintas com n elementos podem ser obtidas a partir desse conjunto.
   6.2. Considere o conjunto com os três inteiros distintos (n = 3) representados pelas variáveis a, b, c. Usando uma árvore de decisão, em que nos nós internos estão comparações de dois elementos e nos nós terminais sequências desses inteiros, represente o algoritmo de ordenação.
   6.3. Qual a altura máxima duma árvore de decisão de um conjunto com n inteiros?

7. Implemente uma versão do *merge-sort*, designado por *natural merge-sort* que explora a ordem pré-existente no *array* a ordenar através da identificação de subsequências naturalmente ordenadas. Consulte os apontamentos para um exemplo.

8. Escreva um procedimento de fusão (merge) de três arrays a[], b[] e c[], todos com tamanho N.

9. Dê a sequência de tamanhos dos subarrays nas várias fusões executadas pelo merge sort para N=39:
   9.1. na versão top-down.
   9.2. na versão bottom-up.

10. Escreva um procedimento que dado um array a[] de tamanho 2N, contendo N elementos ordenados ascendentemente nas posições 0 até N-1, e um array b[] de tamanho N com N elementos ordenados ascendentemente, faça a fusão de b[] com a[] de modo que no final os 2N elementos fiquem ordenados ascendentemente em a[]. Use O(1) de memória extra (in-place).

11. O processo de contagem das inversões dum *array L*, com tamanho *n*, pode ser efetuado em O(*n*.log(*n*)) usando *divide-and-conquer* baseado em *merge-sort*. Se *L* for composto pelos dois *subarrays* A (*subarray* esquerdo) e B (*subarray* direito), o algoritmo recursivo consiste nos três passos seguintes: 1) ordenar A e contar as suas inversões r_A; 2) ordenar B e contar as suas inversões r_B; 3) fazer o merge ordenado de A com B, obtendo-se o *array* ordenado *L*', e simultaneamente contar as inversões existentes, r_{AB}, entre A e B (o número total de inversões é dado por $r_A + r_B + r_{AB}$).
   11.1. Aplique o algoritmo anterior (considere apenas a primeira chamada do processo recursivo) para calcular o número de inversões em L=(1, 5, 4, 8, 10, 2, 6, 9, 3, 7) dividido nos subarrays ordenados A=(1, 5, 4, 8, 10) e B=(2, 6, 9, 3, 7). Quantifique e identifique as inversões existentes.

11.2. Aplique apenas o passo 3 do algoritmo anterior para calcular o número de inversões entre os *subarrays* ordenados A=(3, 7, 10, 14, 18) e B=(2, 11, 16, 17, 23).
11.3. Implemente o algoritmo descrito em computador e teste com os casos mencionados.

Algoritmos e Estruturas de Dados I - Universidade Fernando Pessoa
José Manuel Torres
Ficha 8: Quicksort e aleatoriedade algorítmica

1. Implemente o algoritmo de ordenação *quick sort* na sua implementação básica. Considere a operação de *shuffle* (usando *knuth shuffle*) do *array* como pré processamento da ordenação.

2. Faça o rastreio da execução completa (ignore a operação de *shuffle*) do *quick sort* a cada um dos *arrays* seguintes:
   2.1. (31, 41, 59, 26, 43, 58), n=6
   2.2. (44, 75, 23, 43, 55, 12, 64, 77, 33), n=9
   2.3. (3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48), n=15
   2.4. (20, 4, 26, 13, 35, 37, 26, 19, 13, 44), n=10

3. Faça o rastreio do método *partition()* do *quick sort* para os seguintes *arrays*. Use o primeiro elemento como *pivot* (*partition item*):
   3.1. (Q, U, I, C, K, S, O, R, T, E, X, A, M, P, L, E)
   3.2. (E, A, S, Y, Q, U, E, S, T, I, O, N)

4. Introduza e implemente os melhoramentos seguintes no algoritmo *quicksort* e analise o seu impacto.
   4.1. Usando uma condição de *cutoff* introduza um segundo método de ordenação mais simples (*insertion sort*) quando o tamanho do *subarray* a ordenar for menor ou igual a M.
   4.2. Modifique o método de *partition* de modo a escolher como elemento *pivot* (*partition item*) o valor mediano de 3 elementos do *subarray* a partir.

5. Um dos melhoramentos possíveis do *quicksort* (função *partition*) é escolher, como elemento pivot, a mediana de k (com k=3,5,7,11,...) elementos escolhidos de posições predeterminadas do *array* a ordenar. Implemente um programa que estude, sistematicamente e empiricamente, a variação (média) da divisão do array em função da mediana para vários arrays de input com diferentes configurações e diferentes tamanhos (pode usar a regra de dobrar N em cada experiência, isto é, N=500,1000,2000,...).

6. Dê um exemplo que mostre que o *quicksort* não é um algoritmo de ordenação estável.

7. Mostre que no pior caso o *quick sort* executa a ordenação dum array de tamanho N em tempo $\Theta(N^2)$. Dê exemplos.

8. Usando, como base, o algoritmo de partição do *quicksort*, implemente um algoritmo para:
   8.1. Calcular a mediana dum conjunto de N elementos.
   8.2. O k menor elemento dum conjunto de N elementos.

9. Implemente um algoritmo que rearrange os elementos dum array de tamanho N de modo que todos os números negativos precedam todos os positivos. O algoritmo deverá ser o mais eficiente possível em espaço e em tempo.

10. Discuta a veracidade da seguinte afirmação, para todo $N > 1$, existe *arrays* de tamanho N que são ordenados mais rapidamente pelo algoritmo de *insertion sort* do que pelo algoritmo de *quicksort*.

11. Estime, justificando, quantas vezes será mais rápido o *quicksort* do que o *insertion sort* num array aleatório com um milhão de números.

Algoritmos e Estruturas de Dados I - Universidade Fernando Pessoa
José Manuel Torres
Ficha 9: Heaps/Filas prioritárias

1. Considere que a sequência P I R O * R * * I * T * Y * * * Q U E * * * U * E (onde uma letra representa uma inserção na fila e um asterisco representa a operação de remoção do valor máximo da fila) é aplicada a uma fila prioritária inicialmente vazia. Indique a sequência de letras retornadas após a aplicação das operação de remoção do valor máximo indicadas.

2. Analise criticamente a seguinte afirmação: para implementar a operação de retornar o valor máximo em tempo constante, basta manter o registo do valor máximo inserido até ao momento atual, e depois retornar esse valor quando for invocada a operação de procura do máximo?

3. Implemente o tipo de dados abstrato fila prioritária, que suporta a operação de inserir e remover o máximo, para cada uma das seguintes estruturas de dados subjacente: vetor não ordenado, vetor ordenado, lista ligada não ordenada e lista ligada ordenada. Construa uma tabela com os valores limites (pior caso) para cada operação e para cada uma das quatro implementações.

4. Acha que um vetor, ordenado por ordem decrescente, pode representar uma *heap* max-orientada.

5. Suponha que a sua aplicação tem um número muito elevado de inserções mas apenas algumas operações de remoção do máximo. Que implementação de fila prioritária acha que poderia ser mais eficaz: heap, vetor não ordenada ou vetor ordenado?

6. Suponha que a sua aplicação tem um número muito elevado de operações de pesquisa do máximo mas um número relativamente baixo de inserções e remoções do máximo. Que implementação de fila prioritária acha que poderia ser mais eficaz: heap, vetor não ordenada ou vetor ordenado?

7. Qual é o número mínimo de itens que têm que ser trocados durante uma remoção do máximo numa *heap* de tamanho N sem chaves duplicadas? Dê um exemplo de heap de tamanho 15 para o qual esse mínimo é atingido. Responda à mesma questão para casos de duas e três remoções sucessivas do máximo da heap.

8. Indique qual o amontoado (*heap*) que resulta quando as chaves E A S Y Q U E S T I O N são inseridas nessa ordem em um *heap* maximamente-orientado inicialmente vazio.

9. Indique a sequência dos amontoados produzidos quando as operações de P R I O * R * * I * T * Y * * * Q U E * * * U * E são executadas em um *heap* max-orientado inicialmente vazio.

10. O maior item num *heap* deve aparecer na posição 1, e o segundo maior deve estar na posição 2 ou a posição 3. Indique a lista de posições num *heap* de tamanho 31 onde o elemento *k* maior: (i) pode aparecer e (ii) não pode aparecer, para $k = 2, 3, 4$ (assumindo que todas as chaves são distintas).

11. Desenhe todos os amontoados diferentes que podem ser feitos de cinco chaves: A B C D E e, em seguida, desenhe todos os amontoados diferentes que podem ser feitos de cinco chaves A A A B B.

12. Verifique se os seguintes arrays de inteiros, que representam árvores binárias a partir da posição 1 em level order, satisfazem a condição de heap order, isto é, são binary heaps max oriented:
    12.1. -, 10, 6, 4, 2, 1
    12.2. -, 10, 12, 16, 18, 22
    12.3. -, 20, 18, 10, 16, 12, 4, 8
    12.4. -, 30, 16, 22, 12, 10, 14, 18, 5