Departamento de Ciência de Computadores FCUP

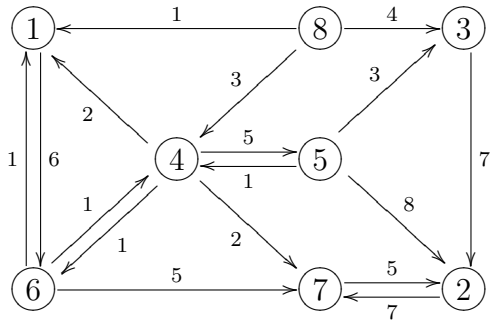
Desenho e Análise de Algoritmos (CC2001) 2017/18

1º Teste (11.11.2017) *duração: 2h30m*

Resolução de questões selecionadas Cotação: $4 \times 2 + 2 + 5 + 5$

Das perguntas 1, 2 e 3, deve responder a DUAS (se responder a mais, **3** não será cotada.)

1. Descreva os passos principais do algoritmo de Kosaraju-Sharir e aplique-o para obter o conjunto de nós que definem cada uma das componentes fortemente conexas do grafo G representado. Assuma que a lista dos adjacentes de cada nó em G está ordenada por ordem crescente de identificador. Apresente o conteúdo das estruturas de dados (usadas no algoritmo) no final de cada um dos passos principais.



Grelha de correção:

(1.5) Indicar os passos principais (ver slides)

(1.5) Apresentar o conteúdo da *stack* depois da pesquisa DFS de G (ver exemplo nos slides)

(0.5) Explicar ou indicar informação sobre os nós que já se encontram visitados sempre que, após efetuar "pop" da stack, inicia a pesquisa DFS a partir do nó v que saiu (v está por visitar em G^T).

(0.5) Apresentar os conjuntos de nós que definem as componentes fortemente conexas.

2. Aplique o algoritmo de Dijkstra ao grafo representado em **1.**, com origem $s = \mathbf{6}$, para obter um vetor p em que $p[v]$ indica o nó que precede o nó v no caminho mínimo de s para v que se encontrou, para cada $v \neq s$. Deve determinar também as distâncias correspondentes. Em casos de empate, opte pelo nó que tiver o identificador **menor**. Indique todos **os valores intermédios** e a ordem pela qual os nós saem da fila de prioridade (mas não desenhe a fila). Se não existir caminho, o valor de $p[v]$ é -1.

Grelha de correção:

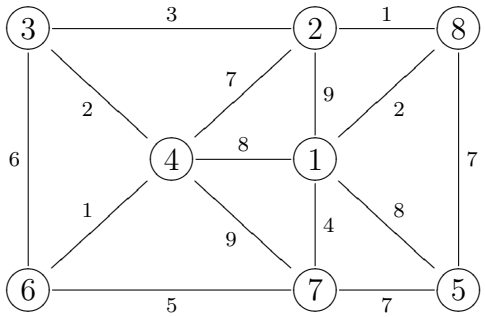
(0.5) Inicializar $pai[\cdot]$ e $dist[.]$ (pode ser por anotação do grafo, como foi feito nas aulas teóricas).

(1.0) Indicar a ordem pela qual os nós saem da fila (no método básico dado nas aulas, todos os nós são colocados na fila inicialmente e, no fim, a fila fica vazia; assim, os nós não acessíveis de s serão indicados também).

(1.5) Atualizar os valores das estimativas das distâncias e dos pais, em cada iteração (pode ser por correção dos valores no grafo, como foi feito nas aulas teóricas).

(1.0) Indicar o contéudo de $p[\cdot]$ no fim (idêntico ao de $pai[.]$) ou usar $p[\cdot]$ em vez de $pai[\cdot]$ no método.

3. Pretendemos obter uma árvore de suporte com peso mínimo para o grafo representado. Ilustre a aplicação do algoritmo de Prim.



- Deve tomar o **nó 5** como raiz da árvore de suporte e apresentar todos os **valores intermédios** para a distância e o pai de cada nó.
- Deve indicar também o **estado da fila de prioridade** (*heap* de mínimo) no início de cada iteração, desenhando a árvore correspondente com nós etiquetados por pares "identificador-valor". Em casos de empate, opte pelo nó que tem o identificador **menor**.

Grelha de correção:

NB: Muito mais trabalhoso do que os outros dois problemas...

(2.5) Aplicar o algoritmo de Prim, mostrando a inicialização de $dist[\cdot]$ e $pai[\cdot]$, a ordem pela qual os nós saem da fila (isto é, da *heap-min*) e a atualização de $dist[\cdot]$ e $pai[\cdot]$, em cada iteração. Esses valores podem ser anotados no grafo, como foi exemplificado nas aulas teóricas.

(1.5) Apresentar o estado da *heap-min* após a inicialização feita por MK_HEAPMIN, depois de cada operação EXTRACTMIN e de cada operação DECREASEKEY. Deverão ser aplicados os algoritmos descritos nas aulas para suportar essas operações (como no "package (C/Java)" disponibilizado, mas sem indicar o conteúdo do vetor que tem informação sobre a posição de cada nó do grafo na *heap-min*).

---

4. Usando **diretamente** a definição das classes $\Theta(n^2)$, $O(n\log_2 n)$ e $\Omega(3n+n\log_2 n)$, prove que:

a) $2n^2-7n+2 \in \Theta(n^2)$. Pode assumir que n é suficientemente grande para que $2n^2-7n+2>0$.

Resposta:

(1.0) Para provar que $2n^2-7n+2 \in \Theta(n^2)$ temos que mostrar que existem **três constantes positivas** c_1, c_2 e n_0, com $n_0 \in \mathbb{Z}^+$, tais que $c_1n^2 \leq 2n^2-7n+2 \leq c_2n^2$, para todo $n \geq n_0$. Tem-se:

- $2n^2-7n+2 \leq 2n^2+2 \leq 4n^2$, para todo $n \geq 1$, e
- $2n^2-7n+2 \geq n^2$, para todo n maior ou igual que a maior raiz de $2n^2-7n+2=n^2$ (ou seja, para $n \geq \frac{7+\sqrt{41}}{2}$).

Concluimos que $n^2 \leq 2n^2-7n+2 \leq 4n^2$, para todo $n \geq 7$. Portanto, as constantes que procuramos podem ser $c_1=1$, $c_2=4$ e $n_0=7$. (notar que c_1 e c_2 não têm de ser inteiros, mas têm de pertencer a $\mathbb{R}^+$)

b) $O(n\log_2 n) \cap \Omega(3n+n\log_2 n) \neq \emptyset$.

Resposta:

(1.0) A interseção não é vazia porque, por exemplo, a função $f(n)=3n+n\log_2 n$ pertence à interseção dos **dois conjuntos**. É trivial que $f(n) \in \Omega(3n+n\log_2 n)$, pois $3n+n\log_2 n \geq c'(3n+n\log_2 n)$ para $c'=1$, para todo $n \geq 1$. Para concluir que $f(n) \in O(n\log_2 n)$, basta observar que para $c=4$, se tem $3n+n\log_2 n \leq cn\log_2 n$, para para todo $n \geq 2$.

(Continua, v.p.f.)

Das perguntas 5 e 6 deve responder a UMA (se responder a mais, **6** não será cotada.)

5. Seja $G = (V, E, d)$ um grafo dirigido com valores nos arcos dados pela função $d : E \to \{\mathtt{a}, \mathtt{b}, \varepsilon\}$. A representação do grafo é baseada em *listas de adjacências* (suponha que ε é denotado pelo carater '#'). Os vértices são identificados por inteiros consecutivos, de 1 a $|V|$. Considere um tipo abstrato de dados (ADT) para representação de subconjuntos de $\mathcal{U}_n = \{1, 2, \ldots, n\}$, com funções: MKEMPTYSET(n), que retorna um conjunto vazio, INSERT(x, k), que insere k no conjunto x, UNION(x, y), que obtém em x a união de dois conjuntos x e y, e SET2LIST(x) que retorna a lista (*linked list*) dos elementos do conjunto x.

Dado um conjunto $S \subseteq V$ e dado $k \in \{\mathtt{a}, \mathtt{b}\}$, queremos determinar o subconjunto de V formado por todos os nós que são acessíveis de nós em S por percursos com exatamente um símbolo k, sendo os restantes ramos do percurso etiquetados por ε, se existirem. Usando pseudocódigo, apresente um **algoritmo** que resolva o problema e que aplique pesquisa em largura (ou em profundidade) a partir de cada nó $s \in S$. Justifique sucintamente a sua **correção** e **complexidade** temporal (use $|S|$, $|V|$ e $|E|$ para a definir). Assuma que as operações referidas acima têm complexidade $\Theta(n)$, $O(1)$, $\Theta(|y|)$ e $O(n)$, respetivamente.

Grelha de correção: **(3.0)** Algoritmo em pseudocódigo. **(1.3)** Justificar sucintamente a correção. **(0.7)** Indicar a complexidade e justificar sucintamente.

Ideias para a resolução:

Algoritmo mais simples: Para cada nó v em V, adaptar pesquisa em largura (ou profundidade) para determinar os nós que são acessíveis de s por percursos só com ε. Depois, usar essa informação para, na pesquisa em largura (ou profundidade) a partir de $s \in S$, determinar os nós acessíveis por percursos-k (isto, é que têm um k exatamente (e de resto apenas ε)).

Algoritmo que integra os dois passos: Efetuar pesquisa em profundidade a partir de cada nó $s \in S$. Na visita a partir de s, para cada nó $v \in V$, mantém informação sobre se já foi visitado num percurso-ε e se já foi visitado num percurso-k. Essa informação é usada na chamada recursiva a partir de v para restringir o tipo de arcos admissíveis no percurso a partir de v. Cada nó pode ser descoberto **duas vezes** (uma se for visitado por um percurso-ε e outra se for visitado por um percurso-k) e dar origem a duas chamadas recursivas. No início, DFS_VISIT é chamada a partir de s com indicação ε.

6. Pretendemos atribuir postos de trabalho a candidatos. Cada candidato está **identificado** por um inteiro (igual ao número de ordem da sua candidatura) e tem **uma classificação** (que é um inteiro positivo que resultou da ponderação de alguns critérios). Os candidatos são colocados nos postos por ordem decrescente de classificação. Não há dois candidatos com a mesma classificação. Cada candidato indicou as suas preferências por postos de trabalho, por ordem decrescente de preferência. Cada posto de trabalho tem disponíveis um certo número de vagas. Há que colocar os candidatos respeitando as suas preferências e classificações. Escreva, em pseudocódigo, uma função para resolver o problema. Suponha que a informação sobre os candidatos está disponível num *array* P, em que $P[i].nota$ dá o valor da classificação do candidato i e $P[i].prefs$ dá a sua lista de preferências (ordenada). O vetor $vagas$ tem em $vagas[j]$ o número de vagas do posto j, para $1 \leq j \leq m$. Existem n candidatos e m postos. O algoritmo deve ser suportado por uma "***heap* de máximo**" de onde se extrai o identificador do candidato que é colocado em cada iteração. A função produz um vetor c em que $c[i]$ indica o posto em que i ficou (e tem -1 se o candidato i não ficar colocado). Justifique sucintamente a **correção** e a **complexidade** temporal do algoritmo que escreveu. Use m, n, e o comprimento total das listas de preferências para definir a complexidade.

Grelha de correção: **(3.0)** Algoritmo em pseudocódigo. **(0.7)** Justificar sucintamente a correção. **(1.3)** Indicar a complexidade e justificar sucintamente.

Ideias para a resolução:

Inspirado em "Sentar ou não sentar" mas sem os candidatos ("habitantes do reino") estarem ordenados nos dados e, além disso, seria preciso indicar onde ficam colocados. Em vez de começar por ordenar os candidatos, deverá construir uma fila de prioridade (que será suportada por uma "heap de máximo" em que as chaves são as classificações dos candidatos) e em cada iteração usar EXTRACTMAX (como no "package" disponibilizado) para identificar o candidato a colocar nessa iteração.

7. A função INVERTER(z, k, j) inverte o segmento $z[k], z[k+1], \ldots, z[j]$ de um vetor z de n elementos, **quando** $0 \leq k \leq j < n$. Se o estado inicial desse segmento for $(c_k, c_{k+1}, c_{k+2}, \ldots, c_{j-2}, c_{j-1}, c_j)$ então, após a chamada da função, será $(c_j, c_{j-1}, c_{j-2}, \ldots, c_{k+2}, c_{k+1}, c_k)$.

```
INVERTER(z, k, j)
1. | Enquanto (k < j) fazer
2. |     t ← z[k];   z[k] ← z[j];   z[j] ← t;
3. |     k ← k + 1;   j ← j − 1;
```

a) Indique uma condição sobre o estado das variáveis (quando se está a testar a condição de paragem do ciclo pela i-ésima vez, para $i \geq 1$) que permite concluir que INVERTER(z, k, j) está correta.

Resposta:

(1.0) Sendo k_0 e j_0 os valores de k e j à entrada da função, então, quando se está a testar a condição de ciclo pela i-ésima vez, o valor de k é $k_0 + (i-1)$, o valor de j é $j_0 - (i-1)$, e $z[k_0 + t] = c_{j_0 - t}$ e $z[j_0 - t] = c_{k_0 + t}$, para $0 \leq t \leq i-2$, e $z[t] = c_t$, para $k_0 + (i-1) \leq t \leq j_0 - (i-1)$.

b) Usando indução matemática, demonstre a condição que enunciou.

Resposta:

(0.5) (i) Caso de base $i = 1$

O valor de k é k_0 e o valor de j é j_0. Como o intervalo $0 \leq t \leq i-2$ é vazio, resta analisar $k_0 + (i-1) \leq t \leq j_0 - (i-1)$, o qual é $k_0 \leq t \leq j_0$. Os valores de $z[t]$ para $k_0 \leq t \leq j_0$ são os valores iniciais, pelo que $z[t] = c_t$, como se afirma.

(1.0) (ii) Hereditariedade

Suponhamos (como hipótese de indução) que a condição enunciada se verifica para um certo i e que $k < j$ quando testa a condição de paragem. Então, o bloco do ciclo será executado novamente, fazendo a troca $z[k]$ com $z[j]$ (na linha 2), ou seja, $z[k_0 + (i-1)]$ com $z[j_0 - (i-1)]$ (dado que k e j têm por hipótese esses valores). A seguir, incrementa k e decrementa j, e testa a condição de ciclo pela $(i+1)$-ésima vez. Nesse teste:

- $k = k_0 + (i-1) + 1 = k_0 + (i+1-1)$ e $j = j_0 - (i-1) - 1 = j_0 - (i+1-1)$;
- $z[k_0 + t] = c_{j_0 - t}$ e $z[j_0 - t] = c_{k_0 + t}$, para $0 \leq t \leq i-1$ (ou seja para $0 \leq t \leq (i+1)-2$), pela hipótese de indução e a troca de $z[k_0 + (i-1)]$ com $z[j_0 - (i-1)]$ efetuada na iteração i;
- $z[t] = c_t$, para $k_0 + (i+1-1) \leq t \leq j_0 - (i+1-1)$ pela hipótese de indução (na iteração i, manteve esses valores).

Portanto, o invariante de ciclo verifica-se quando testa a condição de ciclo pela $(i+1)$-ésima vez. Pelo princípio de indução, de (i) e (ii) pode-se concluir que a condição enunciada se verifica em todas as iterações i, até $k \geq j$.

c) Explique como é que, usando a condição que enunciou, se conclui que a função está correta.

Resposta:

(0.5) O ciclo pára na iteração i em que $k \geq j$ pela primeira vez. Do invariante enunciado conclui-se que $k = k_0 + (i-1) \geq j_0 - (i-1) = j$ e, portanto, o segmento $z[t]$ para $k_0 + (i-1) \leq t \leq j_0 - (i-1)$ ou não tem nenhum (se $k > j$) ou tem apenas um elemento (se $k = j$), sendo esse elemento $c_{j_0 - (i-1)}$.

Como, da condição provada, resulta que $z[k_0 + t] = c_{j_0 - t}$ e $z[j_0 - t] = c_{k_0 + t}$, para $0 \leq t \leq i-2$, então, se $k = j$, o estado de z é $(c_{j_0}, c_{j_0-1}, c_{j_0-2}, \ldots, c_{j_0-(i-2)}, c_{j_0-(i-1)}, c_{k_0+(i-2)}, \ldots c_{k_0+2}, c_{k_0+1}, c_{k_0})$. Se $k > j$, o estado é $(c_{j_0}, c_{j_0-1}, c_{j_0-2}, \ldots, c_{j_0-(i-2)}, c_{k_0+(i-2)}, \ldots c_{k_0+2}, c_{k_0+1}, c_{k_0})$. Em ambos os casos, esse estado corresponde à inversão pretendida.

d) Assuma que n é par e $n \geq 2$. Considere o bloco de código seguinte.

```
k ← 0;
Enquanto (k < n − 1 − k ∧ z[k] > z[n − 1 − k]) fazer
    INVERTER(z, k, n − 1 − k);   falta k ← k + 1;
```

Justifique que a complexidade temporal assintótica do bloco é $O(1)$ no melhor caso e $\Omega(n^2)$ no pior caso. O que é correto afirmar sobre a complexidade desse bloco: que é $\Theta(n^2)$? $O(n^2)$? $\Omega(n^2)$? Explique detalhadamente, começando por definir um modelo de custos para as instruções elementares e por provar que a complexidade de INVERTER(z, k, j) é $\Theta(j - k + 1)$, para $k \leq j$.

Resposta:

(1.5) Para provar que a complexidade de INVERTER(z, k, j) é $\Theta(j - k + 1)$, para $k \leq j$. definimos:

c_1: custo do teste da condição de paragem do ciclo (linha 1) e transferências de controlo
c_2: custo das três atribuições na linha 2
c_3: custo das duas atribuições na linha 3

Se na iteração i se tiver $k_0 + (i-1) \geq j_0 - (i-1)$, então $2i \geq j_0 - k_0 + 2$. Logo, $i \geq (j_0 - k_0)/2 + 1$. Se $j_0 - k_0$ for par, o ciclo termina quando $i = (j_0 - k_0)/2 + 1$. Se não, termina quando $i = (j_0 - k_0 + 1)/2 + 1$. Logo, em ambos os casos, termina quando $i = \lceil (j_0 - k_0)/2 \rceil + 1$. Assim, o tempo total $T(k_0, j_0)$ seria dado por:

$$T(k_0, j_0) = \sum_{i=1}^{\lceil (j_0-k_0)/2 \rceil} (c_1 + c_2 + c_3) + c_1 = (c_1 + c_2 + c_3)\lceil \frac{j_0 - k_0}{2} \rceil + c_1.$$

Como $x \leq \lceil x \rceil \leq 2x$, para $x \geq 0$, verifica-se $c_1 \frac{j_0-k_0+1}{2} \leq T(k_0, j_0) \leq 2(c_1 + c_2 + c_3)(j_0 - k_0 + 1)$.

Mas, j_0 e k_0 denotam os valores iniciais de j e k. Portanto, o tempo de execução de INVERTER(z, k, j) satisfaz $T(k, j) \in \Omega(j - k + 1)$, pois $C''(j - k + 1) \leq T(k, j)$, para $C'' = c_1/2$ e $T(k, j) \in O(j - k + 1)$, pois $T(k, j) \leq 2(c_1 + c_2 + c_3)(j - k + 1) \leq C'(j - k + 1)$, para $C' = 2(c_1 + c_2 + c_3)$. Portanto, $T(k, j) \in \Theta(j - k + 1)$.

Nesta análise não contabilizámos o tempo de passagem dos parâmetros, mas não altera a ordem de grandeza pois é constante, dado que se assume que o *vetor é passado por referência*.

(0.5)- só foi cotado "Melhor caso" devido à deteção tardia da gralha indicada. Para a complexidade do bloco dado em **d)**, note-se que a instrução $k \leftarrow k + 1;$ é necessária para se ter $\Omega(n^2)$ no pior caso. Se não, o bloco teria complexidade $O(n)$ para todas as instâncias.

Com essa instrução, a resposta seria:

- Melhor caso: por exemplo, para instâncias em que $z[k] \leq z[n - 1 - k]$, o tempo é $O(1)$ porque não entra no ciclo.
- Sendo $O(1)$ no melhor caso, não seria correto afirmar que a complexidade do bloco é $\Theta(n^2)$ nem que é $\Omega(n^2)$. Só $O(n^2)$ poderia estar correto, mas tal obriga a mostrar que é $O(n^2)$ no pior caso.
- Como a complexidade de INVERTER$(z, k, n-1-k)$ é $\Theta((n-1-k)-k+1)$, ou seja, $\Theta(n-2k)$, podemos concluir que o tempo de execução de $k \leftarrow 0$ e do ciclo "Enquanto" seria majorado por

$$2c + \sum_{k=0}^{\frac{n}{2}-1} c(n - 2k)$$

para algum $c > 0$, pois, no pior caso, o ciclo termina quando $k = \frac{n}{2}$, porque $k \geq n - 1 - k$. Como

$$2c + \sum_{k=0}^{\frac{n}{2}-1} c(n - 2k) = 2c + cn \sum_{k=0}^{\frac{n}{2}-1} 1 - 2c \sum_{k=0}^{\frac{n}{2}-1} k = 2c + \frac{cn^2}{4} + \frac{cn}{2} \leq 4cn^2$$

concluimos que a complexidade do bloco é $O(n^2)$.

Para as instâncias em que $z[k] \leq z[n - 1 - k]$, para todo $k < n/2$, o tempo é minorado por $2c' + \sum_{k=0}^{\frac{n}{2}-1} c'(n - 2k)$, para algum $c' > 0$, e portanto, é $\Omega(n^2)$, pois é maior ou igual que $\frac{c'}{4}n^2$.

Nesta versão (C), a opção mais simples (ou que requer menos trabalho) é: 1, 2, 4, 6, 7 **(Fim)**

Resolução de 6.

Algoritmo:

COLOCAR$(P, vagas, n, c)$

```
1.  | Para i ← 1 até n fazer
2.  |     c[i] ← −1;
3.  |     keys[i] ← P[i].nota;      /* keys será um array auxiliar */
4.  | Q ← PQ_MK_HEAPMAX(keys, n);
5.  | Enquanto (PQ_NOTEMPTY(Q)) fazer
6.  |     i ← PQ_EXTRACTMAX(Q);
7.  |     lista ← P[i].prefs;
8.  |     Enquanto (c[i] = −1 ∧ NOTEMPTY(lista)) fazer
9.  |         j ← VALORNO(lista);
10. |         Se vagas[j] > 0 então
11. |             vagas[j] ← vagas[j] − 1;
12. |             c[i] ← j;
13. |         senão lista ← PROX(lista);
```

Correção:

O bloco 1-3 inicializa $c[i]$ com -1, para todo i, pelo que se o candidato i não puder ser colocado (linha 12), então o resultado em $c[i]$ está correto. Nesse bloco, cria o array de chaves ($keys$) necessário para a construção da heap (na chamada do construtor na linha 4), de acordo com a implementação descrita nas aulas. No ciclo 5-13, efetua as colocações, por ordem crescente de prioridade, pois PQ_EXTRACTMAX(Q) retorna o identificador do candidato com chave máxima ainda em Q e remove o no correspondente da *heap* (aplicando o algoritmo descrito nas aulas). Quando chega a vez do candidato i, analisa a sua lista de preferências (que se assume estar ordenada) e coloca-o na primeira vaga j disponível (Linhas 10-12) decrementando $vagas[j]$ de uma unidade. Se o conseguir colocar, a condição na linha 8 irá naturalmente falhar (se não o conseguir colocar, a lista ficará eventualmente vazia, dado que vai descendo na lista (linha 13)). Assume-se que VALORNO$(lista)$ e PROX$(lista)$ traduzem as operações elementares sobre os nós de uma lista: acesso a valor e passagem ao nó seguinte (tempo de execução $O(1)$).

Complexidade:

Ciclo 1-3: $\Theta(n)$. Linha 4: $\Theta(n)$. Linha 5: teste da condição $O(1)$, sendo testada $n+1$ vezes. Linha 6: $O(\log_2(n))$. Linha 7: $O(1)$. Linha 8: teste da condição $O(1)$, sendo testada no máximo $O(1 + |P[i].prefs|)$. Bloco 9-13: complexidade $O(1)$.

Portanto, o ciclo 8-13 tem complexidade $O(1+|P[i].prefs|)$ e o bloco 6-13 tem complexidade $O(\log_2(n)+1+|P[i].prefs|)$.

O ciclo 6-13 tem complexidade $O(n \log_2 n + T)$, em que $T = \sum_{i=1}^{n} |P[i].prefs|$.

A complexidade do algoritmo é $O(n + n \log_2 n + T) = O(n \log_2 n + T)$.

Na linha 6, a complexidade é $O(\log_2 k)$ sendo k o número de elementos ainda na fila (se $k = 1$ seria $O(1)$). Contudo, essa análise mais fina não é relevante pois

$$O(\sum_{k=1}^{n} \log_2 k) = O(\int_2^{n+1} \log_2 x \; dx) = O(\int_2^{n+1} \log_e x \; dx) = O(n \log_2 n).$$

Resolução de 5.

Algoritmo:

```
DFS_VISIT(v, k, tipo, R)
 1.  Para cada w ∈ G.Adjs[v] fazer
 2.      Se G.d(v, w) = '#' então
 3.          Se tipo = '#' então
 4.              Se visitadoE[w] = false então
 5.                  visitadoE[w] ← true;
 6.                  DFS_VISIT(w, k, '#', R);
 7.          senão    /* tipo = k */
 8.              se visitadoK[w] = false então
 9.                  visitadoK[w] ← true;
10.                  INSERT(R, w);
11.                  DFS_VISIT(w, k, k, R);
12.      senão se G.d(v, w) = k ∧ tipo = '#' então
13.          se visitadoK[w] = false então
14.              visitadoK[w] ← true;
15.              INSERT(R, w);
16.              DFS_VISIT(w, k, k, R);

PESQUISA(S, k)
17.  R ← MKEMPTYSET(n);
18.  L ← SET2LIST(S);
19.  Para s ∈ L fazer
20.      Para i ← 1 até n fazer
21.          visitadoK[i] ← false;
21.          visitadoE[i] ← false;
22.      visitadoE[s] ← true;
23.      DFS_VISIT(s, k, '#', R);
24.  retorna R;
```

Correção:

Assumimos que o grafo G e o número de nós n são variáveis globais, bem como os dois vetores de booleanos $visitadoE[\cdot]$ e $visitadoK[\cdot]$. Esses vetores indicam, para cada nó, se foi já visitado por um percurso com origem em s do tipo "percurso-ε" e do tipo "percurso-k". A variável R é passada por referência nas chamadas de DFS_VISIT. No fim, R tem o resultado pedido (que se retorna na linha 24). Em DFS_VISIT, o parâmetro $tipo$ indica o tipo de percurso até v. No bloco 1-16, analisa-se a extensão desse percurso para cada adjacente w de v, adaptando o método de pesquisa em profundidade. Se $tipo = k$, só os arcos com $d(v, w) =$ '#' podem ser usados e o percurso que chega a w é um percurso-k (bloco 8–11). A chamada de DFS_VISIT a partir de w (com tipo k), faz-se se $visitadoK[w] =$ `false`. Se $tipo =$ '#' então, se $d(v, w) = k$, o percurso que chega a w é um percurso-k (bloco 12-16) e se $d(v, w) =$ '#', o percurso que chega a w é um percurso-ε (bloco 2-6). Em cada caso, DFS_VISIT é chamada a partir de w, com essa indicação, se não tiver sido chamada antes nessas condições (o que se conclui da análise de $visitadoK[w]$ e $visitadoE[w]$). Os nós que são visitados por percursos-k são inseridos no conjunto R (linhas 10 e 15). Esse conjunto foi criado na linha 18 e acumulará o resultado de todas as chamadas (isto é, das chamadas para $s \in S$). Antes da pesquisa em profundidade a partir de cada nó $s \in S$ (linha 23), definimos todos os nós de V como não visitados (linhas 20-21) e apenas $visitadoE[s] =$ `true` (linha 22), permitindo que todos os nós acessíveis por percursos-k que partem de s sejam visitados nessa chamada.

Complexidade:

A complexidade da chamada DFS_VISIT$(s, k,$ '#', $R)$ (linha 23) é $O(|V| + |E|)$, assintoticamente igual à da pesquisa em profundidade, porque cada nó de V pode ser dar origem no máximo a duas chamadas recursivas de DFS_VISIT e a complexidade de INSERT é $O(1)$. Assim, a complexidade do bloco 20-23 é $O(|V| + |E|)$ e como esse bloco é executado $|S|$ vezes, a complexidade da função é $O(|V| + |S|(|V| + |E|))$, ou seja, $O(|S|(|V| + |E|))$, se $|S| \neq 0$.

```
PESQUISA(S, k)   /* versão alternativa mas cuja correção é menos óbvia */
    R ← MKEMPTYSET(n);
    L ← SET2LIST(S);
    Para i ← 1 até n fazer
        visitadoK[i] ← false;
        visitadoE[i] ← false;
    Para s ∈ L fazer
        Se visitadoE[s] = false então
            visitadoE[s] ← true;
            DFS_VISIT(s, k, '#', R);
    retorna R;
```