EXERCÍCIOS DE ANÁLISE DE ALGORITMOS
BCC, 1o. SEMESTRE DE 2000

1. (Ex. 23.2-3 de CLR) Considere o algoritmo BFS(G, s), visto em aula. Suponha agora que o grafo é dado por sua matriz de adjacência (e não por listas de adjacência) e que BFS(G, s) seja apropriadamente adaptado para entradas desta forma. Qual é a complexidade de tempo deste novo algoritmo? Justifique. [Aqui, forneça a resposta usando a notação O.] $\{$*Data de entrega*: 10/3/2000$\}$
2. Considere os passos (2) e (8) do algoritmo DFS-VISIT(u). Note que são nestes passos que a variável *time* é incrementada. Prove ou desprove que o intervalo de tempo entre duas execuções de incremento da variável *time* é $O(1)$. $\{$*Data de entrega*: 10/3/2000$\}$
3. Lembre que alteramos DFS-VISIT(u) do Cormen, Leiserson, e Rivest levemente, inserindo a instrução *rotule o arco* (u, v) *com* `'+'` imediatamente antes do **if** do passo (4) e a instrução *rotule o arco* (u, v) *com* `'-'` imediatamente após a chamada recursiva de DFS-VISIT no passo (6). Argumentamos que, em qualquer execução da chamada DFS-VISIT(u) no programa principal DFS(G), o intervalo de tempo entre duas tais instruções quaisquer de rotulação de arco é $O(1)$. Justifique esta asserção. $\{$*Data de entrega*: 10/3/2000$\}$
4. (Ex. 23.3-8 de CLR) Suponha que um vértice x em um grafo orientado G é tal que há tanto arcos entrando em x como arcos saindo de x (isto é, há tanto arcos da forma (y, x) como da forma (x, y')). Suponha, ademais, que o arco (x, x) não existe em G. Suponha agora que executamos DFS(G). Considere a árvore de busca em profundidade que contém x gerada por este algoritmo. É verdade que esta árvore sempre tem mais de um vértice? Justifique. $\{$*Data de entrega*: 10/3/2000$\}$
5. Comentamos informalmente em sala que para transformarmos o algoritmo de busca em largura BFS no algoritmo de busca em profundidade DFS, basta trocarmos a fila Q usada em BFS por uma pilha P. Entretanto, a implementação que vimos de DFS é uma implementação recursiva (a tal pilha fica implícita na implementação, como pilha de recursão). Escreva uma versão *não-recursiva* de DFS(G), usando uma pilha explícita P. $\{$*Data de entrega*: 17/3/2000$\}$
6. Suponha que o comprimento máximo de um caminho dirigido em um grafo orientado G é $\leq R$, isto é, todo caminho orientado de G tem no máximo R arcos. Considere agora sua implementação não-recursiva de DFS. Prove ou desprove que a pilha P nunca terá mais

Date: Versão de 20 de junho de 2000.

que $R+1$ elementos na execução da chamada DFS(G). {*Data de entrega*: 17/3/2000}

7. Seja G um grafo (orientado ou não-orientado) e $s \in V(G)$ um vértice de G. Ponha
$$S_d = S_d(s) = \{x \in V(G)\colon\ \mathrm{dist}(s,x) = d\}$$
para todo inteiro $d \geq 0$. Considere agora a chamada BFS(G, s) do algoritmo de busca em largura BFS. Mostre que a fila Q nunca conterá mais do que
$$\max\{|S_{d-1} \cup S_d|\colon d \geq 1\}$$
elementos. [*Sugestão*: Consulte CLR.] {*Data de entrega*: 17/3/2000}
8. (Continuação do exercício anterior) Prove ou desprove que, para cada $d \geq 0$, o conjunto de vértices S_d está todo contido em Q em algum instante na execução da chamada BFS(G, s). Note que esta asserção, se correta, implica que a fila Q tem, em algum instante, pelo menos
$$\max\{|S_d|\colon d \geq 0\}$$
elementos. {*Data de entrega*: 17/3/2000}
9. (Continuação do exercício anterior) Suponha agora que $\mathrm{dist}(s,x) \leq R$ para todo $x \in V(G)$. Mostre que, para algum d, temos $|S_d| \geq n/(R+1)$. {*Data de entrega*: 17/3/2000}
10. Seja t um inteiro positivo e considere o grafo orientado $G = G_t$ definido da seguinte forma. O conjunto de vértices de G está naturalmente particionado em $t+1$ partes: $V(G) = V_0 \cup V_1 \cup \cdots \cup V_t$. Em V_0, temos a 0-upla $s = ()$. Fixe agora $k \in \{1, \ldots, t\}$. Os vértices de G em V_k são as k-uplas $(i_1, \ldots, i_k)$ de inteiros i_j com $1 \leq i_j \leq t$ para todo j, com a seguinte propriedade adicional: se colocamos k rainhas em um tabuleiro de xadrez $t \times t$, com a j-ésima rainha na coluna j e linha i_j $(1 \leq j \leq k)$, então nenhuma rainha ataca outra. Note que, intuitivamente, os vértices de G em V_k são as soluções parciais do problema das rainhas no tabuleiro $t \times t$ com todas as primeiras k colunas já ocupadas.

    Precisamos agora definir os arcos de G. Todo arco de G sai de um vértice em V_{k-1} e vai para algum vértice em V_k, para algum $k \in \{1, \ldots, t\}$. De fato, pomos um arco do vértice $(i_1, \ldots, i_{k-1}) \in V_{k-1}$ para o vértice $(i'_1, \ldots, i'_k) \in V_k$ se e somente se temos
    $$i_1 = i'_1, i_2 = i'_2, \ldots, i_{k-1} = i'_{k-1}.$$
    (i) Mostre que todo caminho orientado em G tem no máximo $t+1$ vértices.
    (ii) Mostre que G tem pelo menos $(t/4)^{t/4}$ vértices. Calcule o valor desta expressão para $t = 50$ e para $t = 100$.
    (iii) Mostre que existe um d para o qual $S_d = S_d(s)$ satisfaz $|S_d| \geq (t/4)^{t/4}/(t+1)$. O que isto significa para o tamanho da fila Q quando executamos a chamada BFS(G, s)?

(*iv*) Qual é o tamanho máximo da pilha P quando executamos uma chamada da sua versão não-recursiva de DFS(G)? Compare este valor como valor em (*iii*).

(*v*) Naturalmente, quando queremos resolver o problema das rainhas no tabuleiro $t \times t$, não geramos este grafo imenso $G = G_t$. O que fazemos é gerar apenas os vértices relevantes para a busca sendo executada. Agora responda: *por que usamos backtracking (também conhecido como busca em profundidade) ao invés de busca em largura para resolver o probrema das rainhas?*

{*Data de entrega*: 17/3/2000}

11. (Ex. 23.3-1 de CLR) Faça uma tabela 3×3 com as linhas e colunas indexadas pelas cores branca, cinza e preta. Na entrada (i, j) desta tabela, indique se é possível acontecer de termos um arco de um vértice de cor i para um vértice de cor j em uma execução de DFS(G) para um grafo orientado G. Responda a mesma pergunta para o caso em que G é um grafo não-orientado. {*Data de entrega*: 24/3/2000}
12. (Ex. 23.3-4 de CLR) Suponha que (u, v) é um arco de um grafo orientado G e que executamos DFS(G). Mostre que (u, v) é
    (*a*) um arco da floresta de busca ou um arco longitudinal descendente se e só se $d[u] < d[v] < f[v] < f[u]$,
    (*b*) um arco longitudinal ascendente se e só se $d[v] < d[u] < f[u] < f[v]$, e
    (*c*) um arco transversal se e só se $d[v] < f[v] < d[u] < f[u]$.
    {*Data de entrega*: 24/3/2000}
13. (Ex. 23.4-3 de CLR) Dê um algoritmo que recebe como entrada um grafo não-orientado G e determina se G contém um circuito. O seu algoritmo deve ter complexidade de tempo $O(n)$. {*Data de entrega*: 24/3/2000}
14. Mostre que é possível determinar G^{T} em tempo $O(n+m)$, quando G é dado através de listas de adjacências. {*Data de entrega*: 24/3/2000}
15. (Ex. 23.5-4 de CLR) Seja G um grafo orientado. Defina o grafo $G' = (V', E')$ como sendo o seguinte grafo orientado: V' é o conjunto dos componentes fortemente conexos de G. Formalmente,
$$V' = \{[x] \colon x \in V\},$$
onde $[x]$ denota o componente fortemente conexo de x em G. O arco $([x], [y])$ existe em G' se e só se existe um arco $(x', y') \in E(G)$ com $x' \in [x]$ e $y' \in [y]$. Mostre que G' é necessariamente um grafo orientado acíclico, isto é, não contém circuitos orientados. {*Data de entrega*: 31/3/2000}
16. O algoritmo visto em sala para determinar os componentes fortemente conexos de uma grafo orientado descobre estes componentes em uma certa ordem bem definida. Intuitivamente, podemos dizer que o algoritmo 'descobre' os vértices do grafo G' do exercício anterior em uma certa ordem. O que podemos dizer sobre esta ordem em relação

ao grafo G'? Especificamente, é verdade que estes vértices são descobertos em uma ordem topológica de G'? (As fontes são descobertas primeiro, depois os vértices 'intermediários', e por fim os sorvedouros.) Justifique sua resposta. $\{$*Data de entrega*: 31/3/2000$\}$

[**Observação.** A partir do próximo exercício, as datas de entrega são segundas-feiras. Entregue suas soluções na secretaria do MAC. Você pode entregar até a secretaria fechar, o que ocorre usualmente às 18:00.]

17. Seja G um grafo não-orientado e $w\colon E(G) \to \mathbb{R}$ uma função peso definida sobre as arestas de G. Seja ainda T uma árvore geradora de G. Note que, para cada aresta e em T, existe um corte $C_e = (S, V \setminus S)$ naturalmente associado a e: se $e = \{u, w\}$, pomos em S todos os vértices de G que são acessíveis a partir de u por caminhos em T *que não contêm o vértice* w. (Estritamente falando, temos *dois* cortes associados a e, a saber, $(S, V \setminus S)$ e o corte equivalente $(V \setminus S, S)$ com S definido como acima.) Podemos também entender o que é este corte considerando os dois componentes de $T - e$, a floresta obtida de T pela remoção da aresta e. Suponha agora que T seja uma árvore geradora de peso mínimo. Mostre que
$$w(e) = \min\{w(f)\colon f \in C_e\}$$
para toda aresta e em T. $\{$*Data de entrega*: 10/4/2000$\}$
18. Considere os programas `rand` e `progl.1` a `progl.4` discutidos em aula. Lembre que os programas `progl.2` a `progl.4` constroem uma floresta que representa a estrutura de componentes dos grafos de entrada.

    Insira um pequeno código nos programas `progl.2` a `progl.4` para que, ao fim da execução destes programas, eles imprimam a *profundidade média* dos nós da floresta contruída por eles. Por exemplo, se o grafo de entrada não tem nenhuma aresta, então claramente todos os componentes são vértices isolados e a floresta correspondente será composta de árvores com um nó cada, de forma que a profundidade média dos nós será 0. Se tivermos exatamente uma aresta no grafo de entrada, a profundidade média será $1/n$, onde n é o número de vértices no grafo de entrada. Verifique qual é a profundidade média dos nós da árvore gerada por `progl.3` e `progl.4` quando as entradas são as saídas geradas pela execução de
    (i) `rand -n50000 -m270494`
    (ii) `rand -n50000 -m270494 -s271828`
    (iii) `rand -n50000 -m270494 -s31415`
    $\{$*Data de entrega*: 10/4/2000$\}$
19. Construa uma entrada para o `progl.3` que gera uma árvore de altura pelo menos 3. Isto é, um vértice deve estar à distância 3 da raiz da árvore à qual ele pertence. Repita o mesmo para altura 4. Em geral,

quantos elementos são necessários e suficientes para que se gere (no pior caso) uma árvore de altura k? {*Data de entrega*: 8/5/2000}

20. Repita a questão anterior para o programa `prog1.4`. Aqui, você não precisa resolver o caso genérico de altura k, mas você teria alguma observação interessante sobre este caso genérico? {*Data de entrega*: 8/5/2000}
21. Seja $(a_k)_{1\le k\le n}$ uma seqüência de $n \ge 1$ números. Mostre que
$$\sum_{1\le k\le n} a_k = na_n - \sum_{1\le k<n} k(a_{k+1} - a_k).$$
Use este fato para provar que
$$\sum_{1\le k\le n} \lfloor \lg k \rfloor = (n+1)q - 2^{q+1} + 2, \tag{1}$$
onde $q = \lfloor \lg n \rfloor$. Prove que (1) também é válida se tomamos $q = \lfloor \lg(n+1) \rfloor$. Seja $p_{\mathrm{med}}(n)$ a profundidade média de um nó em uma árvore balanceada completa com n nós. Determine
$$\lim_{n\to\infty} \frac{p_{\mathrm{med}}(n)}{\lg n}.$$
Aqui, escrevemos $\lg x$ para $\log_2 x$ (esta é uma notação comum entre computeiros). {*Data de entrega*: 15/5/2000}
22. Nesta questão, assumimos uma pequena familiaridade com o *Stanford GraphBase* (SGB) de Knuth; se você não conhece esta plataforma, consulte algum colega que já fez ou esteja fanzendo MAC328 este semestre. Considere o módulo GB_DIJK do SGB, que contém uma função que implementa o algoritmo de Dijkstra para caminhos mínimos em grafos.
    - (i) Quais implementações de filas de prioridade são feitas neste módulo? (Há duas implementações.)
    - (ii) Vimos que com o uso de *heaps binários*, o algoritmo de Dijkstra tem complexidade de tempo $O((m+n)\log n)$. Qual é a complexidade deste algoritmo como implementado no módulo GB_DIJK? Justifique. (Lembre que há duas implementações para a fila de prioridade.)
    - (iii) Em vista de suas respostas ao item (ii), como você justificaria a escolha do autor deste módulo quanto às implementações da fila de prioridade? {*Data de entrega*: 22/5/2000}
23. Estude os dois conjuntos de programas, dados na página desta disciplina (veja a entrada correspondente à aula do dia 19/5/2000), que implementam o quicksort. Use o programa `sort_drive.c` para verificar/comparar experimentalmente o desempenho destas duas implementações do quicksort com entradas aleatórias. Naturalmente, você também pode escrever outros programas para experimentar/comparar

estas duas implementações (`sort_drive.c` é apenas um exemplo rudimentar para executar tais testes). Faça um pequeno relatório dizendo que testes você fez e quais são suas conclusões. {*Data de entrega*: 29/5/2000}

24. Determine a causa da diferença de desempenho das implementações de quicksort discutidas na questão anterior. Faça uma análise
    (*i*) intuitiva/qualitativa
    (*ii*) quantitativa [este item é um pequeno problema para "pesquisa"].
    Nesta questão, sugiro que você faça vários testes empíricos. Você pode contar o número de acessos a memória (como ilustrado no SGB—consulte alguém que conhece esta plataforma caso você não conheça), ou você pode elaborar estatísticas sobre o número de comparações entre itens ("execuções" da macro `less()`), número de trocas (`exch()`), e número de execuções da função `partition()` (há outros parâmetros de interesse?). A partir destes dados empíricos, você terá boas chances para responder o item (*i*) satisfatoriamente. Para responder (*ii*), você terá de escrever algumas recorrências (como fizemos para C_n, o número esperado de comparações na primeira implementação) para os parâmetros identificados em (*ii*), ou possivelmente outros. {*Data de entrega*: 29/5/2000}
25. Considere a implementação de quicksort dada no diretório `quicksort3`, mencionado na entrada do dia 23/5/2000 da página de nossa disciplina.
    (*i*) Faça alguns testes experimentais que comprovam (ou desprovam!) que esta implementação é de fato mais eficiente que as duas implementações anteriores.
    (*ii*) Nesta implementação, o valor de corte `M` para a recursão é 10. Procure determinar empiricamente o efeito do valor desta constante no desempenho desta implementação de quicksort: faça um conjunto de testes para vários valores de `M` de forma a determinar o melhor valor para esta constante em sua máquina de teste. {*Data de entrega*: 5/6/2000}
26. A implementação de quicksort discutida na questão anterior usa a rotina `insertion()`.
    (*i*) Prove que a complexidade de tempo desta rotina é $O(q)$, onde q é o número de *inversões* no vetor de entrada `a[]`. Uma *inversão* em `a[]` é um par ordenado `(i,j)` com `i<j` e `a[i]>a[j]`.
    (*ii*) Prove que, *após* a execução da chamada de `quicksort()` na rotina `sort()` (veja `quicksort.c` no diretório `quicksort3`), o vetor `a[]` é tal que o número de inversões é no máximo `M`$n/2$ para vetores de entrada `a[]` com n elementos.
    (*iii*) Conclua que a chamada de `insertion()` na rotina `sort()` termina em tempo $O(n)$ para vetores `a[]` com n elementos. (Aqui, consideramos `M` uma constante.) {*Data de entrega*: 5/6/2000}

27. Considere os programas no diretório `quicksort4` (um ponteiro para este diretório é dado na entrada do dia 30/5/2000 de nossa página). A implementação de quicksort dada neste diretório é eficiente no caso em que há muitas chaves repetidas (faça alguns experimentos comparativos; você pode usar os programas do diretório `quicksort3b` para isto). Estude esta implementação de quicksort.
    (i) Diga sucintamente como funciona a rotina de partição nesta implementação de quicksort, prestando especial atenção ao caso em que há vários elementos com a mesma chave que o pivô da partição.
    (ii) Argumente que esta implementação, que é um pouco mais complicada que aquela dada no diretório `quicksort1`, não é muito mais lenta que aquela implementação no caso em que não há chaves repetidas. (Portanto, ela é ainda uma boa implementação para o caso em que não há chaves repetidas.)
    (iii) Prove que para entradas com n elementos e no máximo K chaves distintas (considere K aqui como sendo uma constante), o quicksort dado em `quicksort4` tem complexidade de tempo $O(n)$. A constante implícita na notação $O(\,\cdot\,)$ pode depender de K; o ponto é que se K é considerado uma constante, então o tempo que a nossa rotina de ordenação leva cresce *linearmente com n*. {*Data de entrega*: 5/6/2000}
28. Considere a implementação do algoritmo de ordenação mergesort dado no diretório `mergesort1` (veja a entrada correspondente à aula do dia 26/5/2000). Mostre que esta implementação não é estável. Você pode usar os programas do diretório `mergesort2b` para verificar este fato (veja a entrada correspondente à aula do dia 26/5/2000). Reescreva os programas em `mergesort1` para que tenhamos um mergesort estável. Como se comparam estes programas novos com a implementação em `mergesort1` em termos de eficiência? (Faça alguns testes empíricos para justificar sua resposta.) {*Data de entrega*: 12/6/2000}
29. Como vimos em sala, podemos representar a estrutura das chamadas recursivas do mergesort através de uma árvore binária, que é "quase balanceada". Seja $h(n)$ a altura desta árvore quando a entrada tem n nós. Ademais, seja $k(n)$ a profundidade mínima de uma folha em tal árvore. Mostre que temos, para todo $n > 0$,
    (i) $h(n) = \lceil \lg n \rceil$,
    (ii) $k(n) = \lfloor \lg n \rfloor$.
    Em particular, esta árvore é balanceada se n é uma potência de 2. {*Data de entrega*: 12/6/2000}
30. O objetivo deste exercício é examinar o procedimento de ordenação `quicksortX()`, dado no Programa 10.3 de Sedgewick, que é eficiente para ordenar strings. Você pode encontrar este programa no diretório `radix_string` (veja a entrada do dia 2/6/2000 na página de nossa disciplina). No diretório `quick_string` (veja a mesma entrada em

nossa página), você encontra o quicksort básico usado para ordenar strings.

(*i*) Compare o desempenho destas rotinas de ordenação. Você pode usar os programas dos dois diretórios acima para isto, mas, caso você tenha métodos melhores, use-os! Ademais, nos diretórios acima você encontra um arquivo para testes. Tente encontrar outros arquivos (interessantes) para seus testes.

(*ii*) Como você explica a diferença de desempenho entre estas duas rotinas de ordenação? (Provavelmente, a forma mais fácil de responder este item é estudar a Seção 10.4 de Sedgewick, onde ele explica o funcionamento de `quicksortX()`.)

(*iii*) Incremente os programas dados nos diretórios acima para que `sort_drive` imprima, ao término da execução, o *número de comparações entre caracteres* efetuadas durante a ordenação. Use esta versão incrementada destes programas para justificar (ou refutar!) a sua resposta ao item anterior. {*Data de entrega*: 19/6/2000}

31. Nesta questão, tratamos de árvores de busca binária (ABBs) aleatórias. Para fazer esta questão, você pode considerar os programa no diretório `ABBs` (veja a entrada do dia 13/6/2000 de nossa disciplina).[1]

(*i*) Vimos em sala que o número esperado de comparações que fazemos ao buscarmos, com sucesso, um elemento em uma ABB aleatória é dado por

$$C_n = 2\left(1 + \frac{1}{n}\right) H_n - 3. \tag{2}$$

Verifique (2) experimentalmente.

(*ii*) Lembre que denotamos por $\eta(t)$ a altura de uma ABB t e consideremos agora a variável aleatória $\mathbf{H}_n = \eta(t)$, onde t é uma ABB aleatória com n nós internos. Um teorema de Devroye mencionado em sala brevemente diz que $\mathbf{H}_n/\log n$ converge em probabilidade para uma constante $\lambda > 0$. Isto é, para todo $\epsilon > 0$ temos que

$$\lim_{n\to\infty} \mathbb{P}\left(\left|\frac{\mathbf{H}_n}{\log n} - \lambda\right| > \epsilon\right) = 0. \tag{3}$$

Determine o valor de λ empiricamente.

{*Data de entrega*: 28/6/2000}

[1] Os programas neste diretório ainda precisam ser melhorados. Espero fazer isto em um futuro próximo. A versão atual é, pelo menos minimamente, suficiente para esta questão.