Zkouškový Test PAL A **Jméno**

V otázkách 13. a 14. vybírejte vždy jedinou správnou variantu odpovědi.
V ostatních otázkách může být správných variant odpovědí více.
Správně zodpovězená otázka je taková, která určí správně všechny varianty odpovědí a je hodnocena jedním bodem. Pokud některá varianta je určena chybně, otázka je hodnocena 0 body.

1.

Četnosti znaků v textu jsou dány tabulkou. Pro statický Huffmanův kód znaků a pro odpovídající Huffmanův strom T platí, že

a	*b*	*c*	*d*	*e*	*f*
22	11	3	50	44	6

a)- kódy všech znaků začínají stejným symbolem,
b)+ hloubka T je 5 (když hloubka kořene je 0),
c)+ dva znaky mají délku kódu rovnou hlubce stromu,
d)+ T má 6 listů,
e)+ jediný znak má kód délky 3.

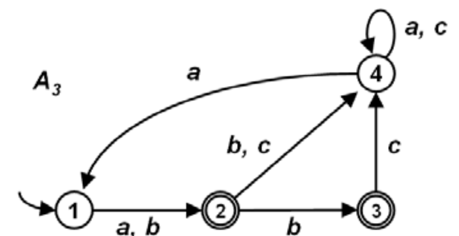
2.

Pro vzorek $P = accac$ nad abecedou $\{a, b, c\}$je sestrojen nedeterministický automat A_1, pomocí nějž lze v textu vyhledávat řetězce, které mají od P Hammingovu vzdálenost rovnu nejvýše 3. Pro A_1 platí

a)- nutně obsahuje ε-přechody,
b)+ má více než 10 stavů,
c)+ má 4 koncové stavy,
d)- v každém stavu, který není koncový, má smyčku (= přechod do téhož stavu),
e)+ po odstranění smyček z přechodového diagramu A_1 vznikne acyklický graf.

3.

K danému automatu A_3 je sestrojen s použitím standardního algoritmu převodu NKA na DKA odpovídající deterministický automat B_3. Přechodová tabulka B_3 obsahuje řádek



a) -	14	124	2	4	F
b)+	14	124	2	4	
c)+	2		34	4	F
d)-	24	1	2	4	
e)-	234	14	34	4	

4.

Jsou dány dva jazyky R_1 a R_2 nad abecedou $A = \{x, y, z\}$. Jazyk R_1 se skládá ze všech slov, která obsahují právě jeden symbol z dvojice (x, y). Jazyk R_2 se skládá ze všech slov, která obsahují symbol z nejvýše jednou. Platí

a)- $R_1 \cap R_2$ je prázdný jazyk,
b)+ R_1 je regulární jazyk,
c)- Doplněk R_2 není regulární jazyk,
d)- $R_1 \subseteq R_2$,
e)+ R_2 není konečný jazyk.

5.

Jazyk L nad abecedou $\{0,1\}$ je generován gramatikou G. G má čtyři pravidla:
$S \rightarrow SA \mid 00S \mid 1$;
$A \rightarrow 1$.
S je počáteční symbol. Platí

a)- žádná dvě slova L nemají stejnou délku,
b)+ L je regulární jazyk,
c)+ L lze rozpoznat deterministickým konečným automatem,
d)- L obsahuje slovo 000111.
e)+ v L existuje slovo $w = w_1w_2...w_n$ $(w_i \in \{0,1\}, n > 1)$, takové, že $w_1 = w_n = 1$.

6.

Pomocí Boyer-Mooreova algoritmu hledáme výskyt vzorku v textu, který se skládá z kombinace znaků *ab* opakované *n* krát (text: *ababab...ab*). Hledaný vzorek je *aaac*. Označme $f(n)$ počet porovnání dvojic znaků, které algoritmus provede nad těmito daty. Pak platí

a)+ $f(n) \in O(n^2)$,
b)- $f(n) \in \Omega(n \cdot \log_2(n))$,
c)- $f(n) \in \Theta(\log_2(n))$,
d)+ $f(n) \in \Theta(n)$,
e)- $f(n) \in O(4)$.

7.

a)- 0, b)- 01, c)- 011, d)+ 00111, e)- 11.	$\delta(q, 0, S) = (q, AS)$ $\delta(q, 0, A) = (q, AA)$ $\delta(q, 1, A) = (r, \varepsilon)$ $\delta(r, 1, A) = (r, \varepsilon)$ $\delta(r, 1, S) = (t, S)$ $\delta(t, 1, S) = (t, \varepsilon)$ $\delta(t, 1, \varepsilon) = (t, \varepsilon)$	Je dán zásobníkový automat P se stavy q, r, t, který přijímá slova prázdným zásobníkem. Vnější abeceda P je $\{0, 1\}$, zásobníková abeceda P je $\{S, A\}$. Na začátku práce je v zásobníku symbol S. Přechody P jsou dány uvedenými vztahy (vrchol zásobníku je vlevo). P přijme slovo

8.

a)+ $M[S][\varepsilon] = \varepsilon$, 3 b)+ M[S][c] = není definováno (= chyba v analyzovaném slově) c)+ $M[A][e] = cBS$, 4 d)+ $M[B][d] = dB$, 6 e)+ $M[A][d] = \varepsilon$, 5.	(1) $S \rightarrow aAB$ (2) $S \rightarrow bBS$ (3) $S \rightarrow \varepsilon$ (4) $A \rightarrow cBS$ (5) $A \rightarrow \varepsilon$ (6) $B \rightarrow dB$ (7) $B \rightarrow e$	Gramatika G má neterminální symboly S, A, B, terminální symboly a, b, c, d, e, startovní symbol S a uvedená pravidla. Označme $M[X][y]$, resp $M[X][\varepsilon]$ prvek rozkladové tabulky gramatiky G, který odpovídá neterminálu X a terminálu y, resp. neterminálu X a prázdnému řetězci ε. Která tvrzení o tabulce M platí?

9.

a) + terminální symbol c b)- neterminální symbol S. d) - neterminální symbol A. e)- řetězec 'aB'. c)+ terminální symbol b.	(1) $S \rightarrow aAbBbS$ (2) $S \rightarrow \varepsilon$ (3) $A \rightarrow aBC$ (4) $A \rightarrow bA$ (5) $B \rightarrow aB$ (6) $B \rightarrow \varepsilon$ (7) $C \rightarrow cC$ (8) $C \rightarrow \varepsilon$	Gramatika G má neterminální symboly S, A, B, C, terminální symboly a, b, c, startovní symbol S a uvedená pravidla. Množina FOLLOW(B) obsahuje

10.

V jaké třídě složitosti je funkce `a` v následujícím programu vzhledem k velikosti pole `p` (budeme ji značit n)? Předpokládejte, že funkce `malloc` (funkce `malloc` naalokuje datovou strukturu velikosti argumentu a vrací na tuto strukturu ukazatel) a `free` (funkce `free` uvolní z paměti naalokovanou datovou strukturu, kterou dostane v argumentu) pracují v konstantním čase vzhledem k velikosti pole `p`. Dále předpokládejte, že velikost proměnné `N` vždy před a po provedení funkce `a` koresponduje s velikosti pole `p`.

a) $O(1)$
b) $\Theta(1)$
c) $O(n^2)$
d) $\Omega(1)$
e) $\Omega(\log(n))$
f) $\Omega(n^2)$
g) $O(n*\log(n))$
h) $\Theta(n^2)$

```
int N = 0;
int * p;
void a (void)
{
      int i=0;
      while ((i < N) && (p[i] == 3)) {
            p[i] = 0;
            i++;
      }
      if (i==N) {
            int m = N+1;
            int j;
            if (N != 0) free(p);
            p = malloc(m*sizeof(int));
            for (j = 0; j<m; j++) p[j]=0;
            N = m;
      }
      p[i] += 1;
}
```

11.

V jaké amortizované třídě složitosti je funkce `a` z předchozího programu vzhledem k velikosti pole `p` (budeme ji značit n)? Předpokládejte, že funkce `malloc` a `free` pracují v konstantním čase vzhledem k velikosti pole `p`.

a) $O(1)$
b) $\Theta(1)$
c) $O(n^2)$
d) $O(n*\log(n))$
e) $\Omega(1)$
f) $\Omega(\log(n))$
g) $\Omega(n^2)$
h) $\Theta(n^2)$

12.

Mějme libovolnou čtvercovou matici A racionálních čísel s rozměry n×n. Rozhodněte, zda platí některá z následujících tvrzení:

a) Matici A lze rozložit pomocí *LUP dekompozice* na matice L, U a P, kde platí, že PA=LU.
b) Pokud je matice A diagonální s hodností n, je po provedení *LUP dekompozice* na A matice P jednotková.
c) Pro matici A takovou, že k ní existuje inverzní matice, lze *LUP dekompozici* provést v čase $O(n^2)$.
d) Pokud má matice A hodnost n, lze k ní najít inverzní matici v čase $O(n^3)$.
e) Předpokládejme, že bychom reprezentovali matici A jako pole čísel s pohyblivou řádovou čárkou dle IEEE 754. Přesnost nalezení inverzní matice k A pomocí *LUP dekompozice* lze zvýšit permutací některých řádků v A před provedením *LUP dekompozice*.

13.

Kolik hran má úplný neorientovaný graf na 13 vrcholech?

a) 42
b) 56
c) 78
d) 120

14.

Jaký je maximální počet hran u libovolného stromu s 1214596 vrcholy?

a) 2314526
b) 362548
c) 1214595
d) 32593623

15.

Uvažujte následující program napsaný v pseudokódu:

Poznámka: metoda `G.neighbors_of(y)` vrací seznam všech vrcholů, které jsou spojeny hranou s vrcholem `y` v grafu `G`.

```
1) Vertices visited = empty;
2) procedure search(Graph G, Vertex start_vertex )
3) {
4)       while (to_visit.number_of_elements() != 0) {
5)             if (v not in visited) then {
6)                   visited.add(v);
7)                   for all Vertex x in G.neighbors_of(v) {
8)                   }
9)             }
10)      }
11) }
```

Rozhodněte, která z následujících tvrzení platí:

a) Přidáním řádku „`Queue to_visit = empty;`“ za řádek 1),
řádku „`to_visit.push(start_vertex);`“ za řádek 3),
řádku „`Vertex v = to_visit.pop();`“ za řádek 4 a
řádku „`to_visit.push(x);`“ za řádek 7)
vznikne následující procedura, která prohledává graf do hloubky.

```
Vertices visited = empty;
Queue to_visit = empty;
procedure search(Graph G, Vertex start_vertex )
{
to_visit.push(start_vertex);
        while (to_visit.number_of_elements() != 0) {
     Vertex v = to_visit.pop();
             if (v not in visited) then {
                  visited.add(v);
                  for all Vertex x in G.neighbors_of(v) {
                       to_visit.push(x);
                  }
             }
   }
}
```

b) Přidáním řádku „`Stack to_visit = empty;`“ za řádek 1),
řádku „`to_visit.push(start_vertex);`“ za řádek 3),
řádku „`Vertex v = to_visit.pop();`“ za řádek 4 a
řádku „`to_visit.push(x);`“ za řádek 7)
vznikne následující procedura, která prohledává graf do hloubky.

```
Vertices visited = empty;
Stack to_visit = empty;
procedure search(Graph G, Vertex start_vertex )
{
to_visit.push(start_vertex);
        while (to_visit.number_of_elements() != 0) {
     Vertex v = to_visit.pop();
             if (v not in visited) then {
                  visited.add(v);
                  for all Vertex x in G.neighbors_of(v) {
                       to_visit.push(x);
                  }
             }
   }
}
```

c) Přidáním řádku „`search(G,x);`“ za řádek 7) a
smazáním řádků 4) a 9) vznikne následující procedura, která prohledává graf do hloubky.

```
Vertices visited = empty;
procedure search(Graph G, Vertex start_vertex )
{
             if (v not in visited) then {
                  visited.add(v);
                  for all Vertex x in G.neighbors_of(v) {
     search(G,x);
                  }
             }
}
```

d) Přidáním řádku „`int a = 0;`“ za řádek 1),
řádku „`Priority_Queue to_visit = empty;`“ za řádek 1),
řádku „`to_visit.push(a++, start_vertex);`“ za řádek 3),
řádku „`Vertex v = to_visit.pop();`“ za řádek 4 a
řádku „`to_visit.push(x);`“ za řádek 7)
vznikne následující procedura, která prohledává celý graf do hloubky.
Poznámka: Metoda `push` má v tomto případě dva argumenty. První je hodnota klíče a druhý jsou data. Metoda `pop` vrací vždy ty data, která byla vložena do prioritní fronty s nejnižší hodnotou klíče. Po provedení operace `pop` je tento klíč z prioritní fronty odstraněn.

```
Vertices visited = empty;
int a = 0;
Priority_Queue to_visit = empty;
procedure search(Graph G, Vertex start_vertex )
{
to_visit.push(a++,start_vertex);
        while (to_visit.number_of_elements() != 0) {
      Vertex v = to_visit.pop();
               if (v not in visited) then {
                     visited.add(v);
                     for all Vertex x in G.neighbors_of(v) {
                           to_visit.push(x);
                     }
               }
   }
}
```

e) Přidáním řádku „`int a = 0;`“ za řádek 1),
řádku „`Priority_Queue to_visit = empty;`“ za řádek 1),
řádku „`to_visit.push(a--, start_vertex);`“ za řádek 3),
řádku „`Vertex v = to_visit.pop();`“ za řádek 4 a
řádku „`to_visit.push(x);`“ za řádek 7)
vznikne procedura, která prohledává celý graf do hloubky.
Poznámka: Metoda `push` má v tomto případě dva argumenty. První je hodnota klíče a druhý jsou data. Metoda `pop` vrací vždy ty data, která byla vložena do prioritní fronty s nejnižší hodnotou klíče. Po provedení operace `pop` je tento klíč z prioritní fronty odstraněn.

```
Vertices visited = empty;
int a = 0;
Priority_Queue to_visit = empty;
procedure search(Graph G, Vertex start_vertex )
{
to_visit.push(a--,start_vertex);
        while (to_visit.number_of_elements() != 0) {
      Vertex v = to_visit.pop();
               if (v not in visited) then {
                     visited.add(v);
                     for all Vertex x in G.neighbors_of(v) {
                           to_visit.push(x);
                     }
               }
   }
}
```