Abstraktní datové typy (ADT)

- **množina:** žádný prvek v množině se nesmí opakovat

Následující tabulka ukazuje asymptotické složitosti pro různé implementace. Symbol *n* označuje počet prvků, symbol *x* označuje mohutnost univerza - počet všech různých prvků, které lze do množiny vkládat.

Operace	Pole	Usp. pole	Sp. seznam	Usp. sp. seznam	Bit. pole
vypsání všech prvků	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(x)$
vložení prvku, bez kontroly přítomnosti prvku	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$
vložení prvku, s kontrolou přítomnosti prvku	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$
odebrání prvku	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$
zjištění, zda je prvek v množině	$O(n)$	$O(log(n))$	$O(n)$	$O(n)$	$O(1)$
sjednocení dvou množin	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n)$	$O(x)$
průnik dvou množin	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n)$	$O(x)$
rozdíl dvou množin	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n)$	$O(x)$
symetrická diference dvou množin	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n)$	$O(x)$
doplněk množiny	$O(x*n)$	$O(x)$	$O(x*n)$	$O(x)$	$O(x)$

(např. můžeme do množiny vkládat čísla od 1 – 1000, ale máme tam aktuálně jen čísla 1 – 20; n tedy bude 1 – 20, zatímco x bude 1 – 1000)

- **seznam:** prvky se mohou opakovat a pořadí prvků je důležité
  - **implementace pomocí pole**: při realokaci po každém přidání nového prvku má vložení nového prvku složitost **$O(n^2)$**; při inteligentní realokaci (na cca 1,5 násobek původní velikosti) má vložení složitost **$O(1)$**
  - **implementace pomocí zřetězeného seznamu:** vložení prvku má konstantní složitost **$O(1)$**; vyhledávání pozice je lineární **$O(n)$**
- zásobník: LIFO; veškeré operace má konstantní – teda **$O(1)$**
- **fronta:** FIFO
- **prioritní fronta:** fronta, ve které má každý prvek definovanou hodnotu priority (jsou následně vybírány dle priority a pokud mají stejnou prioritu, tak podle toho který z nich je vložen byl vložen jako první)
- **asociativní pole:** „obdoba pole“, kdy k prvků nepřistupujeme pomocí indexu, ale klíče

„Dobré vědět“

char* - char pointer, chová se jako pole

char *str1 – str1 v podstatě referuje na první prvek, je tam uloženo intovsky 1 číslo
strlen – délka řetězce
sizeof – počet bajtů

```
1  char *str1 = "abcd";
2  char *str2 = "xyz";
3
4  sizeof(str1) - sizeof(str2);
5  strlen(str1) - strlen(str2);
```

Řádek 4: 1 – 1 = 0 (*ve str1 je uloženo jedno intové číslo, tzn. velikost v bajtech bude stejná*)
Řádek 5: 4 – 5 = 1

```
1  char str1[20] = "abcd";
2  char str2[10] = "xyz";
3
4  sizeof(str1) - sizeof(str2);
```

Řádek 4: 20 – 10 = 10 (*v daném případě sizeof vrací velikost celého alokovaného pole*)

a != b – „a se nerovná b“
a =! b – „a se rovná !b“, v zásadě se jedná o ekvivalentní zápis jako a = (!b)

Příklady ze zkouškových písemek 2015 (teoretická část)

1.

```
1  ...
2  int main()
3  {
4    char s[100]={0};
5    fgets(s,sizeof(s), stdin);
6    printf("%s", s);
7    fputs(s,stdout);
8    return 0;
9  }
10
```

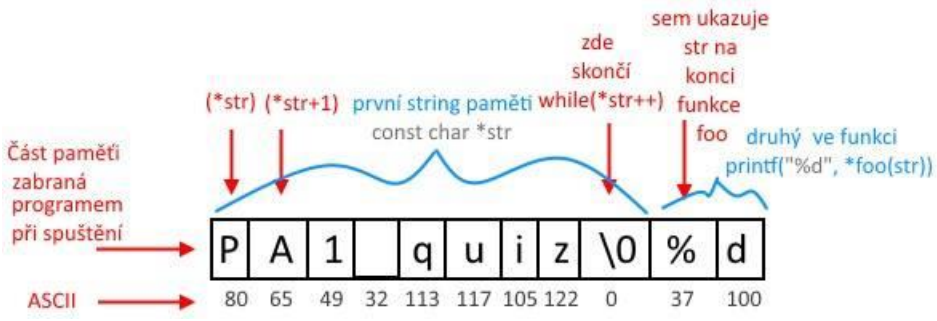
- Jestliže přijde „a“ 100x na vstup, program může přepsat paměť a spatnout.
- **Při zobrazení mohou být obě řádky stejně dlouhé.** ✔
- Pokud program spustíme vícekrát a zadáme stejný vstup, může být zobrazený výstup pokaždé jiný.
- **Program vždy zobrazí ten samý text 2x.** ✔
- Program nikdy nezobrazí ten samý text 2x.

2.

```
1  const char * foo(const char * str)
2  {
3    while(*str++) {}
4    return str;
5  }
6  int main()
7  {
8    const char *str = "PA1 quiz";
9    printf("%d", *foo(str));
10   /* ASCII  A=64,...,Z=90
11             a=97,...,z=122  */
12   return 0;
13 }
```

- Program je správně a zobrazí: ____
- Program neuvolňuje paměť.
- **Program může spadnout / výsledek není definovaný.** ✔

/ Ač mnohým ten kód zobrazí 37, dle fit-wiki je toto 100% ověřená odpověď; vysvětlení: (konkrétní hodnota '%' vychází proto, protože je řetězec "PA1 quiz" uložen v paměti hned před tisknutym řetězcem, který začíná '%') Na to se ale nemůžeme spoléhat, takže nedefinované chování. */*



3.

```
1  int main()
2  {
3    unsigned short a[27][10];
4    /* zde doplnit */
5    foo(a);
6    return 0;
7  }
8
```

Jaké rozhraní může mít fce foo, aby bylo její volání správné:

- void foo(unsigned short (*x)[27])
- **void foo(unsigned short x[10][10]) ✔**
- void foo(unsigned short ***x)
- **void foo(unsigned short x[27][10]) ✔**
- **void foo(unsigned short x[][10]) ✔**

4..

T(x,y,z) je časová složitost algoritmu pro hledání největšího společného dělitele tří čísel: GCD(x,y,z). Aritmetická operace s čísly mají složitost O(1). Co platí:

- **T(x,y,z) ∈ O(xyz) ✔**
- **T(x,y,z) ∈ O(x^2 + y^2 + z^2) ✔**
- T(x,y,z) ∈ O(log(log(x + y + z)))
- **T(x,y,z) ∈ O(xy + yz + xz) ✔**
- **T(x,y,z) ∈ O(x + y + z) ✔**
- **T(x,y,z) ∈ O(log(xyz)) ✔**

5.

```
...
int i;
for (i = -8191; i < 8191; i++)
  if (i != -i)
    printf("*");

```

Na počítači jsou celá čísla int reprezentována v doplňkovém kódu. Kolik hvězdiček vypíše následující kus kódu? (celé desítkové číslo)

- **16 381 ✔**

/ Reprezentace je v doplňkovém kódu, tzn. 0 je to samé jako -0 (kdyby to bylo v přímém kódu, tak by se -0 odlišovala od 0 a pak by to bylo 8191+8191), proto se vypíše 8191 + 8190 = 16 381 hvězdiček */*

6.

```
for (i = 1; i < n; i *= 2, n -= i)

```

Určete složitost T(n):

- **T(n) ∈ O(n^3)** ✔
- **T(n) ∈ O(log(n))** ✔
- T(n) ∉ O(3^n)
- T(n) ∉ O(n)
- T(n) ∉ O(n^2(log(n)))

/ když si rozepíšeme cykly pro n = 4, n = 6, n = 8, n = 12, n = 16, ... tak z toho vypozorujeme, že při $n = 2^x$ bude počet cyklů (x-1); konstantu můžeme v zásadě zanedbat, takže při 2^x bude počet cyklů x, tzn. O(x). Jak tam dostaneme „n“? $n = 2^x$ => log(n) = x */*

7.

```
for (i = 1; i < n; i *= 2)
  for (j = 0; j < i; j++)
    doIt();

```

Určete složitost T(n):

- T(n) ∈ O(log(log(log(n))))
- **T(n) ∈ O(n)** ✔
- T(n) ∉ O(n^2(log(n)))
- **T(n) ∈ O(n(log(n)))** ✔
- **T(n) ∈ O(n^√2)** ✔

/ vnější cyklus se provádí pro i = 1, i = 2, i = 4, i = 8, i = 16, ... vidíme z toho, že pro nějaké „n“ se vnější cyklus provede sqrt(n); při rozepsání vnitřního cyklu vidíme, že v nejhorší variantě se to provede o jedna méně než vnější cyklus; konstantu (-1) můžeme zanedbat, tzn. sqrt(n)*sqrt(n) dává složitost O(n) */*

8.

```
int main()
{
  char x = 'Z';
  int y = 'Z';
  printf("%c = %c", x, y);
  return 0;
}

```

Co platí pro následující kód:

- Program půjde zkompilovat, ale může spadnout nebo bude mít nedefinovaný výsledek.
- **Program zobrazí: (doplnit) ✔ Z = Z**
- Program nepůjde zkompilovat.

9.

```c
#include <stdio.h>
#include <limits.h>
int main()
{
  int a = 35, b = 35, c = 35;
  printf("%s", a==b==c ? "true" : "false");
  return 0;
}
```

Co platí pro následující kód:

- Program půjde zkompilovat, ale po spuštění spadne.
- **Program půjde zkompilovat, nespadne a zobrazí: (doplnit) ✔ false**
- Program půjde zkompilovat, nespadne a ukončí se, ale výsledek není definovaný.
- Program půjde zkompilovat, nespadne, ale zacyklí se.
- Program nepůjde zkompilovat.

/ Nejprve se provede a == b, to je 1, tzn. teď se provede 1 == c, což je 0, tzn. vypíše se pravá strana, tzn. „false“ */*

10.

text: slova_v_azbuce slova_v_češtině

Vyberte vhodné kódování textu tak, aby: - šlo tento text reprezentovat - jeho editace byla co nejjednodušší:

- UTF-8
- **UCS-2 ✔**
- ISO-8859-2(Central European)
- **UTF-32 ✔**
- US-ASCII

/ UCS-2 je obdoba UTF-16, ALE! UCS-2 je na úpravu stejně jednoduchá jako UTF-32, protože je to zastaralé kódování, které mělo fixních 16 bitů (stejně jako UTF-32 má fix. 32 bitů) | UTF-16 je sice podobná UCS-2, ale nemá fix. na 16 bitech, pro složitější znaky používá i 32, proto na úpravu je složitější než UCS-2 a UTF-32 | zároveň ale když se ptáme na kodování, které zabírá nejméně místa, tak UTF-16 taky není správná odpověď, protože je tu ještě UTF-8, které zabírá ještě méně místa než UTF-16 */*

11.

```
1  #include <stdio.h>
2  #include <limits.h>
3  int main()
4  {
5    #define MIN(X,Y) ((X) < (Y) ? (X) : (Y))
6    int a = 6, b = 5;
7    printf("%d %d %d", MIN(a++,b++), a, b);
8    return 0;
9  }
```

Co platí pro následující kód?

- Program půjde zkompilovat, ale po spuštění spadne.
- Program půjde zkompilovat, nespadne a zobrazí: doplnit
- **Program půjde zkompilovat, nespadne a ukončí se, ale výsledek není definovaný.** ✔
- Program půjde zkompilovat, nespadne, ale zacyklí se.
- Program nepůjde zkompilovat.

/ define je sice uprostřed main, ale to vůbec nevadí – takto napsaný platí od řádku 5 až do konce programu, tzn. se provede v pořádku; jedná se o post increment, tzn. MIN příjme 6++ a 5++, porovná 6 < 5 (nicméně ihned po porovnání se do „a“ přiřadí 7 a do „b“ se přiřadí 6); zjistí, že to není pravda, takže vrátí Y, tedy b++ (b++ je ale postincrement zase, takže návratová hodnota bude 6, ale v „b“ po návratu bude uložena hodnota 7); levá strana (tedy X) se vůbec neprovede, takže v „a“ bude stále uloženo 7, takže se vypíše návratová hodnota 6 a pak „a“, což je 7 a „b“ což je taky 7 */*

12.

```
1  unsigned safeSizeOf( unsigned int * * x )
2  {
3    if ( !x ) return 0;
4    return sizeof(x);
5  }
6
7  int main()
8  {
9    unsigned int * foo[199];
10   printf("%u %u", (unsigned)sizeof(foo), safeSizeOf(foo));
11   return 0;
12 }
```

Co zobrazí následující kód?

- Zobrazí: **1592 8** ✔

/ první se ptá na velikost pole ukazatelů, což je 199*velikost_ukazatele, tzn. 199*8 = 1592; to druhé (safeSizeOf) se ptá jen na velikost ukazatele, v daném případě 8B, tzn. vrátí 8 ; pokud by tam bylo sizeof(*foo), tak to vrátí velikost unsigned intu */*

13.

```
...
int i;
for (i = -256; i < 255; i++)
  if (i != -i)
    printf("*");
```

Na počítači jsou celá čísla int reprezentována v doplňkovém kódu. Otázka na doplnění: Kolik hvězdiček vypíše následující kus kódu? (celé desítkové číslo)

- **510** ✔

/ Reprezentace je v doplňkovém kódu, tzn. 0 je to samé jako -0 (kdyby to bylo v přímém kódu, tak by se -0 odlišovala od 0 a pak by to bylo 256+255), proto se vypíše 256 + 255 = 510 hvězdiček */*

14.

```
#include <limits.h>

int main(void){
int a=5, b=5, c=5;
printf("%s", a==b==c ? "true" : "false");
return 0;
}
```

- **Program půjde zkompilovat, výsledek: ___** ✔ **false**
- Program nepůjde zkompilovat.

/ vykoná se a == b, což bude 1 a pak se vykoná 1 == c, což bude 0, tzn. se vypíše „false“ */*

15.

Máme množinu n bodů v rovině (x,y) a máme zjistit duplicity. Jakou bude mít algoritmus složitost?

- $\sqrt{n}$
- $\boldsymbol{n^2 \log(n)}$ ✔
- $\log(n)$
- $\boldsymbol{n^{\sqrt{2}}}$ ✔
- 1
- $\boldsymbol{n * \log(n^3)}$ ✔

/ můžeme si množinu seřadit n*log(n) a pak jí projít (n), tzn. n*log(n)+n, tedy složitost O(log(n)) */*

16.

Máme dvourozměrný zřetězený spojový seznam. Jaká bude časová složitost testu prázdnosti seznamu?

- **∈ O(1)** ✔

17.

```
1  short a[99000];
2  int *b[90000];
3
4  int foo(void) {
5      short a[30000],b[30000];
6      int c;
7      /* neco s mallocem */
8      return c;
9  }
10
11 int main(void) {
12     char *c;
13     ...
14 }
```

Kolik paměti v zásobníku zabere tento program?

- **sizeof(char*) + 60000*sizeof(short) + sizeof(int)** ✔

18.

Spočtěte složitost přidání prvku do obousměrně zřetězeného spojového seznamu (ideální řešení).

- T(n) ∈ O(log(log(log(n))))
- **T(n) ∈ O(n)** ✔
- **T(n) ∉ O($\sqrt{n}$)** ✔
- **T(n) ∈ O(n(log(n)))** ✔
- **T(n) ∈ O(2^n)** ✔

/ máme seznam, potřebujeme přidat prvek, tzn. nejhorší možná možnost je, že budeme přidávat prvek do prostředka, tzn. n/2, což znamená, že složitost je v daném případě O(n), tzn. musíme zaškrtnout i všechno co je větší a zároveň nezapomenout zaškrtnout i odpovědi, kde se složitost nerovná menšímu (jako třeba odpověď c v daném případě) */*

19.

```
int sum  0;
switch(a) {
    default: sum += 3;
    case 0: sum += 2;
    case 1: sum += 2;
    case 3: sum+= 2;
}

printf("%d", sum);
```

Je tento switch validní? Co vypíše pokud a = 0?

- **ANO – vypíše 6** ✔

/ chybí mu sice break, ale to nevadí, fungovat bude i tak – naskočí na validní řádek (tzn. pokud např. a = 0, tak na case 0) a pak propadne až úplně nakonec, přičemž vykoná po cestě všechny příkazy (tzn. pokaždé přičte 2); pokud by „a“ bylo např. 4 (tzn. ani jedno z těch čísel), skočí to na default a projede všechno až dolů, tzn. by to vypsalo 9 */*

20.

Spočtěte složitost operace vyzvednutí v případě datového typu fronta realizovaný pomocí dynamicky alokovaného pole (kruhový buffer):

- **O(1)** ✔

/ Kruhový buffer: pole se 2 ukazateli, kde jeden ukazuje na začátek a druhý na konec; složitost bude konstantní, tedy O(1), protože jediné, co v daném případě děláme je, že vrátíme prvek, na který ukazuje ukazatel začátku; jinými slovy – jedná se o frontu, tzn. můžeme buď přidávat tam kde je jeden ukazatel (konec fronty) a nebo odebírat odtud, kde je druhý ukazatel (začátek fronty), v obou případech je tedy složitost O(1) */*

21.

```
char x[50];
/* nactu do ni dle jeji
 * velikosti ze stdin */
printf("%s", x);
printf(x);
```

Jak se zachová tento kód?

- **První řádka může být delší, ale také stejně dlouhá.** ✔

/ printf(x) může řetězec useknout v místě, kde se mu něco nebude zdát, například pokud budeme mít „30%slevy“, může dojít k tomu, že se to zasekne po 30 a vypíše to tudíž jenom 30; naproti tomu printf(„%s“, x) vypše „30%slevy“ a nebude s tím mít problém, tzn. pokud printf(x) zrovna vypíše řetězec správně, tak první řádka bude stejně dlouhá a pokud ne, tak bude delší */*

22.

Co se stane s preprocesorovou funkcí (makrem), která je definována v těle mainu?

- **Bude v pořádku fungovat, ale bude platit jen pro main a funkce pod ním.** ✔

23.

Máme dvě neseřazená dynamicky alokovaná pole, jakou složitost má porovnání těchto polí?

- **O(n^2)** ✔

/ pro každý prvek projíždíme prvky druhého pole; pokud by se jednalo o seřazená pole, složitost by byla O(n) */*

24.

Jaká je složitost výběru mediánu seřazeného pole (nej. algoritmus)?

- **O(1)** ✔

/ medián v seřazeném poli je hodnota v prostředku pole a protože máme pole, tak můžeme rovnou přistoupit k prostřednímu indexu a vytáhnout medián, tzn. se jedná o 0(1) složitost */*

25.

```c
#include<stdio.h>
#define MIN(a,b) (a>b) ? b:a

int main() {
    int a = 8;
    int b = 7;
    printf("%d", MIN(a++,b++));
    printf("%d %d\n", a, b);
    return 0;
}
```

Co vypíše tento kód?

- **8 9 9** ✔

/ jedná se o post increment, tzn. MIN příjme 8++ a 7++, porovná 8 > 7 (nicméně ihned po porovnání se do „a“ přiřadí 9 a do „b“ se přiřadí 8); zjistí, že je to pravda, takže vrátí b, (ale protože tam dochází v podstatě ke copy-paste, tak na místě „b“ je ve skutečnosti b++, ale jde o postincrement, takže návratová hodnota bude 8, ale v „b“ po návratu bude uložena hodnota 9); pravá strana (tedy „a“) se vůbec neprovede, takže v „a“ bude stále uloženo 9, takže se vypíše návratová hodnota 8 a pak „a“, což je 9 a „b“ což je taky 9 */

26.

```
1  void swap(int* a, int* b) {
2      int *c = a;
3      a = b;
4      b = c;
5  }
6
7  int main() {
8      int a = 1;
9      int b = 2;
10     swap(&a,&b);
11     printf("%d %d\n", a, b);
12     return 0;
13 }
14
```

Co vypíše tento kód?

- **1 2** ✔

/ funkci swap předávám adresy, funkce swap si vytvoří kopii těch adres, které si sama v sobě přehazuje; přes tyto adresy samozřejmě může přistoupit na reálné prvky, ale nikde to nedělá, pouze přehazuje ty svoje lokální kopie adres, původní hodnoty tedy zůstanou beze změny */

27.

Předpokládáme celá čísla bez znaménka o velikosti 12 bitů. Jakou hodnotu musí mít proměnná b, aby byl výsledek programu 762?

```
1  unsigned int a = 502, b = ...;
2
3  a = a + 31 * b;
4  printf("%d\n", a);
5
```

- **3708** ✔

/ a = (502 + 31 * b) mod 4096 = 762 ; proč mod 4096? Ze zadání víme, že předpokládáme čísla o velikosti 12 bitů, tzn. $2^{12} = 4096$; rovnici vyřešíme například pomocí Euklidova algoritmu... */*

28.

```c
int main ( void )
{
unsigned char y = "g";

printf("%c",y);
return 0;
}
```

Jak se zachovají jednotlivé kompilátory?

- **Program nelze zkompilovat (překladač hlásí chyby nebo varování).** ✔

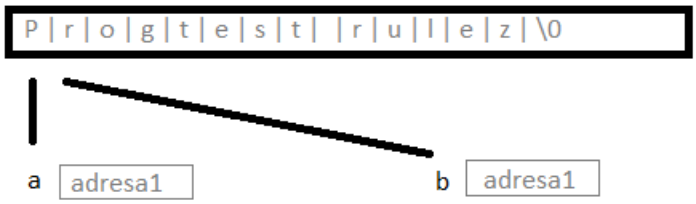
/ problém je v tom, že na levé straně máme unsigned char a na pravé straně máme const char* (jinými slovy jsou tam špatné uvozovky, měly by tam být jednoduché uvozovky) */*

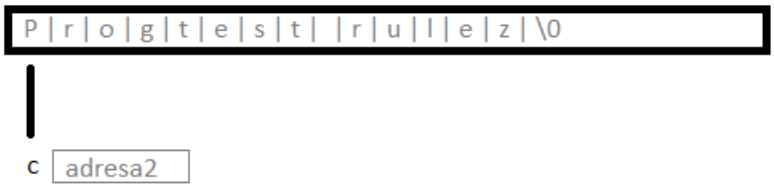
29.

```c
int main(void) {
    const char* a = "Progtest rulez!";
    const char* b = a;
    char *c = strcpy((char*)malloc(28),b);
    printf("%d %d %d",!strcmp(a,c),a == b, b == c);
    free(c);
    return 0;
}
```

- Program půjde zkompilovat, ale pracuje špatně s pamětí
- Program nepůjde zkompilovat
- **Program pracuje dobře, zobrazí: _____** ✔ **1 1 0**

/ vytvoří se „a“ (řádek 2); vytvoří se „b“ a to se namapuje na stejné místo jako „a“, protože v „a“ je uložena adresa na první prvek, tzn. přiřadíme jí do „b“ a tím ho namapujeme na stejnou adresu (řádek 3); strcpy(cíl,zdroj) – vezme „b“ a překopíruje to komplet do „c“ – nenamapuje to, překopíruje to celé na novou adresu (řádek 4); strcmp(a,c) – porovná řetězce „a“ a „c“, přičemž pokud jsou stejné, vrátí 0; my víme, že jsme do „c“ překopírovali celý řetězec na novou adresu, strcmp porovnává do první binární nuly, tzn. „a“ a „c“ budou ekvivalentní, tzn. strcmp vrátí 0 a !0 je **1**; dále porovnáme a == b, což se rovná (míří na stejnou adresu, tzn. jsou stejné, tzn. vrátí **1**; no a nakonec porovnáme b == c, to ale není rovno, protože „b“ míří na jinou adresu než „c“, tzn. vrátí **0** */*





30.

```c
#include <stdio.h>
#include <limit.s>

int main(void) {
    double d;
    int cut = 0;
    for (d = 0; d != 19; d+= 0.2) {
        cut++;
        printf("%d", cut);
        return 0;
    }
}

```

- Program půjde zkompilovat, ale bude nekonečný
- Program nepůjde zkompilovat
- **Program pracuje dobře, zobrazí: _____ ✔ 1**

/ vzhledem k tomu, že zde je na 10. řádku return, vypíše to 1 a ukončí se | kdyby tam ale ten return nebyl, byl by cyklus nekonečný protože jde o double, může se nám stát, že nebudeme schopni v doublu zcela přesně reprezentovat 19 (nejspíš to bude něco jako 18.9999 nebo tak něco), tím pádem to s velkou pravděpodobností tu podmínku vůbec nesplní a pojede do nekonečna => ponaučení: NIKDY nepoužívat double jako řídící proměnnou ☺ */*

31.

```c
int rec(int n, int *a, int *b) {
    return n == 1 ? *a = *b : rec(n/2, b, a) + rec((n+1)/2, a + n
    /2, b + n/2);
}

int main (void) {
    int a[] = {5,9,9,6,4,0};
    int b[] = {1,7,2,9,2,0};

    printf("%d %d", rec(5,a,b), rec(6,b,a);
    return 0;
}
```

- Program půjde zkompilovat, ale pracuje špatně s pamětí
- Program nepůjde zkompilovat
- **Program pracuje dobře, zobrazí: _____ ✔ 27 27**

/ můžeme si to rozkreslit jako stromeček... */*

32.

```c
int main(void) {
    int i = 0, sum = 0;

    switch(i) {
        case 0: sum = 12;
        case 1: sum = 7;
        case 2: sum = 11;
        default: sum = 5;
    }

    printf("%d", sum);
    return 0;
}
```

- Program nepůjde zkompilovat
- **Program pracuje dobře, zobrazí: _____ ✔ 5**

/ switchi sice chybí break, ale to nevadí, fungovat bude i tak – naskočí na validní řádek (tzn. case 0) a pak propadne až úplně nakonec (tzn. v sum bude 5) */*

33.

Uvažujme datový typ fronta realizovaný jako dvousměrný zřetězený spojový seznam. Jaká je časová složitost složitost T(n) operace test prázdnosti?

- **$T(n) \in O(\log^2(n))$** ✔
- **$T(n) \in O(1)$** ✔
- $T(n) \notin O(n\sqrt{n})$
- $T(n) \notin O(n!)$
- **$T(n) \in O(n)$** ✔

/ jedná se o frontu, tzn. buď přidáváme nebo odebíráme s tím, že máme ukazatel na začátek a na konec; jde o typ FIFO – tedy buď přidáváme z jedné strany nebo odebíráme z jedné strany, složitost je tedy konstantní – O(1); test prázdnosti v daném případě znamená, že se stačí podívat na první prvek a vidím – tzn. 0(1) */*

34.

```
1  int foo(int* p, int n) {
2      static int a[60];
3      int i, v;
4      for (i = 0; i < n; i ++)
5          a[i] = p[i];
6      if(n == 1) return a[0];
7      v = foo(a, n-1);
8      if(v > a[n-1])
9          return v;
10     else
11         return a[n-1];
12 }
13
14 int main(void) {
15     int b[60], i;
16     for (i = 0; i < 60; i ++)
17     b[i] = rand();
18     foo(b,60);
19     return 0;
20 }
21
```

Kolik instancí proměnné „a“ bude nejvýše existovat?

- **1** ✔

/ protože je proměnná static int, zachová se i do dalších volání a proto bude existovat 1 instance */*

35.

Uvažujme následující deklaraci. Které z výrazů jsou syntakticky správné l-value (mohou se vypisovat na levé straně přiřazení)?

unsigned char *A[57];
unsigned char *B[57][57];
unsigned char *C[(57,57)];
unsigned char (*D)[57];

- **C[53]**
- **D[53][34]**
- **D**
- **A[53][34]**
- A

/ pointou tohoto příkladu je, že máme v zadání uvedeno tak, jak je to zapsáno na začátku kódu; my máme říct, které z nabízených odpovědí mohou být na levé straně dále v kódu, tzn. jako kdybychom chtěli pokračovat v zadaném kódu a napsat např. D = NULL; neřešíme kde to přeteče nebo nepřeteče, řešíme pouze syntaxi */*

36.

Vstupem je seřazený dvousměrně zřetězený spojový seznam. Najděte medián. Algoritmus může při zpracovávání dat změnit (např. procházet), má k dispozici pouze 0(1) paměti. Jaká je jeho časová složitost T(n):

- **$T(n) \in O(n^2)$**
- $T(n) \in O(1)$
- $T(n) \notin O(n * \log(\log(n)))$
- $T(n) \notin O(3^n)$
- **$T(n) \in O(n!)$**

/ jde o medián, tzn. prostřední hodnotu; máme ukazatel na začátek a na konec, musíme projít spojový seznam až do půlky, abychom našli medián, tzn. n/2, což je 0(n) */*

37.

```
1  void swapvalues(int a, int b) {
2      int c = a;
3      a = b;
4      b = a;
5  }
6
7  int main(void) {
8      int x = 86; y = 94;
9      swapvalues(x,y);
10     printf("%d %d", x, y);
11     return 0;
12 }
13
```

- Program půjde zkompilovat, ale pracuje špatně s pamětí
- Program nepůjde zkompilovat
- **Program pracuje dobře, zobrazí: _____ ✔ 86 94**

/ do swapvalues se hodnoty „x“ a „y“ zkopírují, tzn. swapvalues si dále hraje se svýma lokálníma kopiema a nijak neovlivní původní hodnoty */*

38.

Unsigned short int je reprezentace pomocí 10 bitů v binárním kodu. Jak bude vypadat číslo 884?

- **1101110100 ✔**

/ 884:2 = 442 (zbytek 0) -> 0*
442:2 = 221 (zbytek 0) -> 0
221:2 = 110 (zbytek 1) -> 1
110:2 = 55 (zbytek 0) -> 0
55:2 = 27 (zbytek 1) -> 1
27:2 = 13 (zbytek 1) -> 1
13:2 = 6 (zbytek 1) -> 1
6:2 = 3 (zbytek 0) -> 0
3:2 = 1 (zbytek 1) -> 1
1:2 = 0 (zbytek 1) -> 1

tzn. ***1101110100***

*a nebo si otevřete bc –l a pište: ibase=A;obase=2;884 a klikejte enter ☺ */*

39.

Určete časovou složitost T(n) volání funkce doIt(), která má časovou složitost 0(1).

```
int i, j;
for (i = 0; i < n*n; i++)
    for (j = i*i; j; j/=2)
        doIt();
...
```

- **T(n) ∈ O(n^2log(n))** ✔
- T(n) ∈ O($\sqrt{n}$)
- **T(n) ∉ O(log(n))** ✔
- **T(n) ∈ O(n^4)** ✔
- T(n) ∈ O(n$\sqrt{n}$)

/ vnější cyklus se provede n*n krát; vnitřní cyklus je i, které do kola děleno 2, tzn. log(i); i je v daném případě n^4, protože pro nejhorší variantu (tedy poslední cyklus) je n^2 a j = i*i, tzn. n^2 * n^2 = n^4 => z toho nám vychází O(n^2 * log(n^4)) => log(n^4) je to samé jako 4 * log(n) a konstatu můžeme zanedbat, tzn. je to O(n^2 * log(n)), takže musíme zaškrtnout tu a všechny větší */*

40.

```
int main(void) {
    char x = 'Z';
    int y = 'Z';
    printf("%c=%c", x, y);
    return 0;
}
```

- Program nepůjde zkompilovat
- **Program pracuje dobře, zobrazí: _____** ✔ **Z=Z**

41.

```
unsigned safesizeof(unsigned int ***x) {
    if(!x) return 0;
    return sizeof(x);
}

int main(void) {
    unsigned int **foo[50];
    printf("%u %u", (unsigned)sizeof(foo),safesizeof(foo));
    return 0;
}
```

Co zobrazí následující kód? Velikosti dat: char 1B, short 2B, int 4B, long long 8B, ukazatel 4B.

- **200 4** ✔

/ stejně jako v předchozí verzi; sizeof(foo) vrací 50*velikost_ukazatele, tzn. 50*4 = 200; a safesizeof(foo) vrátí velikost ukazatele, tzn. 4 */*

42.

```
int i;
for(i = -4095; i < 4095; i++)
    if(abs(i) < 0)
        printf("x");
...
```

Na počítači jsou celá čísla int reprezentována v přímém kódu pomocí 13 bitů. Kolik „x“ vypíše tento kód?

- **0** ✔

/ abs(i) udělá absolutní hodnotu z „i“, tzn. nezávisle na čísle NIKDY nemůže být menší než 0 */*

43.

```
int i = 0, j = 1;
for(i = 0; i < n*n; i+=j) {
    doIt();
    j+=2;
}
...
```

Určete časovou složitost T(n) volání funkce doIt(), která má časovou složitost 0(1).

- T(n) ∉ O(n^2log(n))
- T(n) ∈ O($\sqrt{n}$)
- T(n) ∈ O($log^2(n)$)
- **T(n) ∈ O(n^3)** ✔
- **T(n) ∈ O($n\sqrt{n}$)** ✔

/ vyzkoušíme si pár možností a zjistíme, že pro cyklus 0 ... n^2 to proběhne n-krát – pro cyklus 0 ... 16 proběhne jen 4x [1. i = 0+3; 2. i = 3+5; 3. i = 8+7; 4. i = 15 + 9 <= a dál už to nepojede, protože 15+9 už je větší než 16] tzn. se jedná o složitost O(n) */*

44.

Kolik bajtů paměti bude potřebovat následující program v datovém segmentu? Povolá tolerance je 1 promile. Neuvažuj runtime a OS. Velikost typu: char 1B, short 2B, int 4B, long long 8B, ukazatel 8b.

```
1  int * a[94000];
2  static int b[25000];
3
4  void test(void){
5      int i;
6      int *ptr[6];
7      for(i = 0; i < 6; i++) {
8          int a[35000];
9          static int b[65000];
10         ptr[i] =(int*)malloc(22000 * sizeof(int));
11     }
12     for(i = 0; i < 6; i++){
13         free(ptr[i]);
14     }
15 }
16
17 int main(void) {
18     int *c = (int*)malloc(89000*sizeof(*c));
19     int a[28000];
20     static int b[14000];
21     free(c);
22
23     c = (int*)malloc(89000*sizeof(*c));
24     test();
25     free(c);
26     return 0;
27 }
28
```

45.

Ekvivalentní výrazy pro a[11][17][29]:

- ***(*(*(a+11)+17)+29)** ✔
- **(*(a+11))[17][29]** ✔
- **(*(*(a+11)+17))[29]** ✔
- **(*(a[11]+17))[29]** ✔

46.

```c
int main() {
    char a[] = "abcde";
    char *b = a;
    strcpy(a,"aaaa");
    printf("%s\n",a);
    return 0;
}
```

- **Program funguje správně** ✔
- Program nepůjde zkompilovat
- Program neuvolňuje paměť
- Program pracuje špatně s pamětí, může spadnout

47.

```c
int main() {
    char a[100];
    fgets(a,sizeof(a),stdin);
    printf("%s\n",a);
    return 0;
}
```

- Program funguje správně
- Program nepůjde zkompilovat
- Program neuvolňuje paměť
- Program může spadnout při zadání 100xA

48.

```c
#define <stdio.h>
#define <math.h>

int main() {
    int i;
    for(i = -256; i < 255; i++) {
        if(fabs(i) > 0) {
            printf("*");
        }
    }

    return 0;
}
```

Na počítači jsou celá čísla int reprezentována v doplňkovém kódu pomocí 9 bitů. Kolik * vypíše tento kód?

- **Program funguje správně a vypíše _____ hvězdiček ✔ 510**
- Program nepůjde zkompilovat
- Program neuvolňuje paměť
- Program může spadnout

/ logicky se nevytiskne pro i = 0, protože fabs(0) není větší než 0; nicméně někoho by mohlo napadnout, že když máme 9 bitovou reprezentaci (která má max. 255), tak budeme mít problém s -256; nicméně protože jsme v doplňkovém kódu, ten má rozsah skutečně -256 až 255, tzn. do „i" se to bez problémů přiřadí; abs(-256) by následně ale přetekla a vrátila -256, což by způsobilo výpis 509 hvězdiček – my tu ale máme, díky kterému to nepřeteče, vrátí to nějaké kladné číslo (protože to vrací double), tzn. i ta možnost fabs(-256) bude > 0, tzn. se vypíše 510 hvězdiček */*

49.

```c
#include <stdio.h>

int main() {
    int a[] = {0,1,2,3,4,5};
    printf("%d",(a+7)[-3]);
    return 0;
}
```

- **Program funguje správně a vypíše _____ ✔ 4**
- Program nepůjde zkompilovat
- Program může spadnout

/ můžeme si to představit skutečně jako a+7-3 => tedy v daném případě je to stejné jako kdybychom napsali a[4], tedy 5. prvek */*

50.

```c
#include <stdio.h>

int rec(int* a, int b) {
    if(!--b)
        return a[0]&0x1;
        return(a[b]&0x1) + rec(a,b);
}

int main() {
    int a[] = {3,2,6,1,7,5};
    printf("%d %d",rec(a,5),rec(a,4));
    return 0;
}
```

- **Program funguje správně a vypíše _____ ✔ 3 2**
- Program nepůjde zkompilovat
- Program může spadnout

/ můžeme si to zase rozkreslit do stromku, ze kterého nám výjde, že v zásadě počítáme počet lichých prvků v rozsahu a[0] – a[b-1], tedy v prvním případě počítáme lichá čísla na pozicích a[0] – a[4] a ve druhém případě na pozicích a[0] – a[3] */*

51.

```c
void foo(int n) {
    int i;
    int j;
    for(i = 0; i < n; i++) {
        for(j = i; j < n; j+=2) {
            if(i==j) { x(); }
        }
    }
}
```

Určete časovou složitost T(n) volání funkce x(), která má časovou složitost 0(1).

- T(n) ∉ O(n^2log(n))
- T(n) ∈ O($\sqrt{n}$)
- T(n) ∈ O($log^2(n)$)
- **T(n) ∈ O(n^3)** ✔
- **T(n) ∈ O($n\sqrt{n}$)** ✔

/ vzhledem k podmínce na řádku 6 se to provede pouze n-krát, tzn. O(n) */*

52.

Short reprezentovaný 11 bity v doplňkovém kódu 01001110000. Jeho dekadická hodnota je?

- **624** ✔

/ otevřete si bc –l a pište: ibase=2;obase=A;0100111000 a klikejte enter ☺*

toto nám bude bez problémů fungovat při kladném čísle -> s nulou na začátku, pokud se však jedná o doplňkový kód a máme na začátku jedničku (tedy je to záporné číslo), musíme nejprve výraz znegovat, pak se stejnou syntaxí zapíšeme výraz do bc, k výsledku přičteme jedničku a před číslo dáme 0

např.: 10101000 -> (znegujeme a dáme do bc) -> ibase=2;obase=A;01010111 -> 87 -> (přičteme 1) -> 88 -> (a na začátek dáme mínus) -> -88/*

53.

```
1  int rec(unsigned int x, unsigned int x) {
2      if(!x && !y) return 0;
3      return rec(y>>1,x>>1)+(x&1)+(y&1);
4  }
5
6  int main(void) {
7      printf("%d %d", rec(26,24), rec(26,28));
8      return 0;
9  }
```

- **Program funguje správně a vypíše _____** ✔ **5 6**
- Program nepůjde zkompilovat
- Program lze zkompilovat, ale spadne nebo je jeho chování nedefinované.

/ >> funguje jako bitový posun s tím, že pokud jde o unsigned int, tak „nově vzniklá místa“ nahrazuje 0; & je logický AND */*

54.

```
int main(void) {
    char a[]={3,10,4,8,0};
    char * b;
    for(b = a; b < a+5; b++) {
        if(b>a) printf(" ");
        printf("%d", *b);
    }
    return 0;
}
```

- **Program funguje správně a vypíše _____ ✔ 3 10 4 8 0**
- Program nepůjde zkompilovat
- Program lze zkompilovat, ale spadne nebo je jeho chování nedefinované.

55.

```
...
int a = 11;
for(i = INT_MIN; i <= INT_MAX; i++) {
    a^=6;
}
printf("%d", a);

```

- **Program funguje správně a vypíše _____ ✔ 11**
- Program nepůjde zkompilovat
- Program se zacyklí
- Výstup není definovaný

/ a^=6 je XOR, tzn. že 1 je tam kde je 1 a 0, zatímco 0 a 0 nebo 1 a 1 dá dohromady 0*
1011 – číslo 11
0110 – číslo 6

1101 – lichý cyklus
0110

1011 – sudý cyklus
0110

1101 – lichý cyklus

Tímto velice snadno zjistíme, že každý lichý cyklus bude výsledek 1101 a každý sudý cyklus bude výsledek 1011, zbývá nám tedy určit jestli počet cyklů ve foru je lichý nebo sudý.

*Buď máme představu o hodnotách a víme, že INT_MIN je sudé číslo (reprezentace 100000...) a INT_MAX liché číslo (reprezentace 0111111...) a víme, že i = sudé; i <= liché je ve skutečnosti **sudý** počet cyklů. A nebo si to vyzkoušíme na konkrétních číslech z jiných příkladů – např. 9 bitový int (viděli jsme ho v dřívějších příkladech) má rozsah -256 až 255, na kterém krásně vidíme, že je to –sudé_číslo až +liché_číslo; i = -256; i <= 255 je 256+256 (kvůli <=) což je **sudé** číslo, tzn. to vypíše 1011, tedy číslo **11**, protože se ptáme na intovskou hodnotu uloženou v „a“ */*

56.

text: slova_v_azbuce slova_v_češtině

Vyberte vhodné kódování textu tak, aby v něm byl správně interpretován uvedený text a aby jeho uložení zabíralo nejméně místa:

- **UTF-8** ✔
- UCS-2
- ISO-8859-1
- UTF-32
- US_ASCII

57.

Uvažujme dvousměrně zřetězený spojový seznam. Jaká bude časová složitost T(n) testu prázdnosti seznamu. Neuvažujme režii OS a podobně.

- $T(n) \notin O(2^n)$
- **$T(n) \in O(1)$** ✔
- **$T(n) \in O(\log(n))$** ✔
- **$T(n) \in O(n^3)$** ✔
- $T(n) \notin O(n)$

/ máme zřetězený spojový seznam, kde se nám stačí podívat na první prvek (tam kde máme ukazatel), tzn. složitost je O(1) */*

58.

```c
int foo(int a[][13], int b[][7]) {
    return sizeof(*a) - sizeof(*b);
}

int main(void) {
    int a[23][13];
    int b[23][7];
    printf("%d", foo(a,b));
}
```

- **Program funguje správně a vypíše _____ ✔ 6*sizeof(int)**
- Program nepůjde zkompilovat
- Program se zacyklí
- Výstup není definovaný

/ řádek 2 => tam jsou v podstatě jen ukazatele na jednotlivé řádky, tzn. odečítám velikost řádku a(13) – velikost řádku b(7), tzn. 13 – 7 = 6, nicméně je to velikost řádku intového pole, proto výsledek bude 6*sizeof(int) | pozor!! pokud by na řádcích 6. a 7. byly prohozené hodnoty ve druhé závorce (tzn. bylo by to např. int a[23][7]; int b[23][13]; tak by program nešel zkompilovat, protože ta čísla neodpovídají těm, která jsou nastavena ve funkci foo() */*

59.

```
1  int rec(int n, int *a, int *b) {
2      return n == 1 ? *a == *b : rec(n/2, b, a) + rec((n+1)/2, a +
       n/2, b + n/2);
3  }
4
5  int main (void) {
6      int a[] = {5,9,9,6,4,0};
7      int b[] = {1,7,2,9,2,0};
8
9      printf("%d %d", rec(5,a,b), rec(6,b,a));
10     return 0;
11 }
```

- Program půjde zkompilovat, ale pracuje špatně s pamětí
- Program nepůjde zkompilovat
- **Program pracuje dobře, zobrazí: _____ ✔ 0 1**

/ můžeme si to rozkreslit jako stromeček... daný případ se od toho předchozího liší v 2. řádku o *a == *b (v tom původním bylo jen jedno rovnítko) -> v tomto případě v zásadě jde jenom o to, kolik stejných číslic je obsaženo v množinách. rec(5,a,b) se dostane nejdál na a+4 a b+4, tzn. nebude mít žádnou stejnou; rec(6,b,a) se ale dostane až k a+5 a b+5, kde je uložená 0 v obou množinách, proto výsledek je 1 */*