Knowledge Sharing - zadania

Michał Kukowski

Kwiecień 2020

> Nie mogę nikogo niczego nauczyć.
> Mogę tylko sprawić, że zaczną myśleć.
>
> *Sokrates*

1 Wprowadzenie

Ten dokument zawiera listę zadań (podobną do tych, które można spotkać na studiach), które zostały uznane za wartościowe i takie, które polepszą umiejętności programistyczne.

Zadania podzielone są na 3 kategorie:

- **inżynierskie** - polegają na zaimplementowaniu małej części systemu np. alokatora. Są to zadania najbliższe temu co robimy w pracy.
- **eksperymentalne** - polegają na zaimplementowaniu kilku różnych wariantów danego algorytmu lub systemu, a następnie przeprowadzenia eksperymentów pokazujących wady i zalety wykorzystanych metod.
- **algorytmiczne** - polegają na rozwiązaniu zadania tekstowego skrywającego podstawowy problem informatyczny, np. Vertex Cover.

Rozwiązanie zadania powinno zawierać następujące elementy:

- **Opis problemu** - należy przedstawić problemy (informatyki) jakie należy rozwiązać aby uzyskać działające rozwiązanie zadania. Np: Alokator - fragmentacja, bin packing problem. Głównie tyczy się to zadań algorytmicznych.
- **Kod źródłowy** - należy udostępnić wszystkim swój kod źródłowy i krótko go omówić na prezentacji zadania. Projekt powinien zawierać czytelną strukturę plików i katalogów, a także automatyczny system budowania: make, cmake. Przykładowy Makefile można znaleźć w dodatku 3.2.
- **Działający przykład** - autor implementacji powinien przygotować przykład, który odpali przy wszystkich podczas prezentacji. Wyjątkiem są eksperymenty, które czasami trzeba przeprowadzić offline. Wtedy przykład możemy sobie darować.
- **Schemat działania** - podczas prezentacji należy przedstawić schemat działania wybranego algorytmu, najlepiej wykonać na sucho (bez kodu) swój algorytm na wybranym przykładzie. Ten punkt najlepiej wykonać w formie prezentacji i rysunków.
- **Benchmarki** - jeśli zadanie prosi o eksperymenty należy przygotować i udostępnić kod do przeprowadzania eksperymentów i generowania wyników. Tak aby każdy mógł pobawić się implementacją i sprawdzić wpływ zmian na zachowanie się algorytmu. Wyniki eksperymentów mogą być przedstawione w formie wydruku na konsoli lub wykresów. Spróbuj napisać skrypt do generowania wykresów. Możesz użyć do tego dowolnego języka jak python czy julia lub narzędzia gnuplot. Do zmierzenia czasu wykonywania się kodu w języku C możesz użyć makra umieszczonego w dodatku 3.1.

2 Lista zadań

Zadanie 1: Prosty alokator (inżynierskie)

Napisz prosty alokator. Przedstaw problemy jakie należy rozwiązać podczas tworzenia alokatora. Przygotuj system debugowy, który przedstawi statystyki dotyczące fragmentacji pamięci (np średni rozmiar bloku, liczba niezależnych bloków pamięci).
Alokator powinien korzystać ze statycznej pamięci (tablicy) jako pamięci dostępnej dla programu.

```
1 #include <stdint.h>
2
3 /* default value, MEMORY_SIZE should be passed in compile time by
      -D option */
4 #ifndef MEMORY_SIZE
5 #define MEMORY_SIZE (1 << 20) /* 1MB */
6 #endif
7
8 static uint8_t memory[MEMORY_SIZE];
```

Zadanie 2: Model LDM-CM (inżynierskie)

Zadanie polega na zaimplementowaniu systemu, który ukaże charakterystykę modelu local-common memory.
Należy zaimplementować system mierzenia czasu potrzebnego na dostęp do pamięci. Należy uwzględnić latencje czyli czas potrzebny na ustawienie kontrolera oraz przepustowość czyli czas poświęcony na kopiowanie.
Możemy przyjąć następujące wartości:

```
1 #define LDM_LATENCY_CC      0
2 #define LDM_THP_PER_BYTE_CC 1
3
4 #define CM_LATENCY_CC       30
5 #define CM_THP_PER_BYTE_CC  1
```

W zadaniu należy przygotować:

1. System mierzenia cykli (jako cycles += 30)
2. Funkcje (makro) readFromCm - czytającą jedną zmienną z CM do LDM
3. Funkcje (makro) writeToCm - zapisującą jedną zmienną z LDM do CM
4. Funkcje (makro) ldmToCmCopy - kopiującą dowolną liczbę bajtów z LDM do CM
5. Funkcje (makro) cmToLdmCopy - kopiujacą dowolną liczbę bajtów z CM do LDM
6. Różne przykłady pokazujące charakterystykę pamięci (np. opłaca się buforować całą strukturę chociaż czytamy z niej 2 pola)

Aby zadbać o poprawność kodu, możemy użyć narzędzia *sparse* jest to semantyczny parser języka C, który rozróżnia przestrzeń adresową. Przestrzeń tę trzeba zadeklarować. Poniższy przykład powinien zostać użyty w zadaniu aby zaimplementować przestrzeń ldm jak i cm.

```c
#include <stdio.h>
#include <stdlib.h>

/* to check address space run: sparse -Waddress-space -Wcast-to-as *.c */

/* for sparse only */
#ifdef __CHECKER__
#define __cm __attribute__((address_space(1)))
#define __force_cast __attribute__((force))
#else /* in gcc mode do nothing */
#define __cm
#define __force_cast
#endif
/* this macro should be used only to trick compiler in special cases like below */
#define __force_cast_to_ldm  __force_cast
#define __force_cast_to_cm  __force_cast __cm

#define cmMalloc(size) (__force_cast_to_cm void*)malloc(size)
#define cmFree(ptr) free((__force_cast_to_ldm void*)ptr)

void f(__cm int *a);
void f(__cm int *a)
{
        printf("A = %d\n", *a);
}

int main(void)
{
        __cm int buf = 17;
        __cm int *b = &buf;
        f(b);

        __cm int *t = cmMalloc(100 * sizeof(*t));
        cmFree(t);
        return 0;
}
```

Zadanie 3: Hardware z LDM-CM (eksperymentalne)

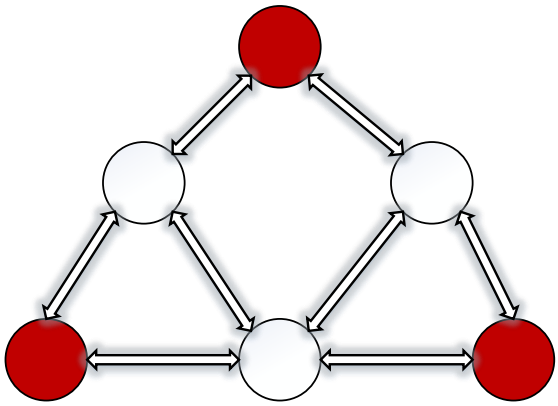
Postaw hipotezy na temat działania modelu LDM-CM. Możesz do tego użyć dokumentacji tego modelu.
Zweryfikuj eksperymentalnie postawioną hipotezę. Użyj do tego pomiarów LAT.
Przygotuj różne przykłady pokazujące charakterystykę pamięci (np. opłaca się buforować całą strukturę chociaż czytamy z niej 2 pola).

Możesz wzorować się na benchmarkach dysków twardych. (Seq read (read array), Seq Write(write array), Random Read (read int), Random Write (write int)).

Zadanie 4: Rozmieszczenie anten (algorytmiczne)

Właściciel dużego domu na przedmieściach miasta wpadł na niecodzienny pomysł. Chce stworzyć serwerownie w swoim domu, jednak ma dostęp tylko do internetu LTE. Kupił więc bardzo dużo kart od różnych dostawców internetu, ustawił routery tak, aby łączyły się z różnymi przekaźnikami internetowymi, jednak internet nadal nie był szybki i stabilny. Postanowił więc kupić do każdego routera anteny. Przygotował na dachu maszty dla każdej z nich. Okazało się jednak że anteny zakłócały się wzajemnie. Właściciel zmierzył więc bezpieczną odległość od anten i zastanawia się właśnie ile anten może umieścić na masztach aby się nie zakłócały.

Całość da się zamodelować jako graf $G(V, E)$. Wierzchołkami $v \in V$ niech będą maszty, na których można umieścić anteny. Wierzchołki posiadają krawędź $e = (v, v') \in E$ jeśli umieszczone na masztach anteny będą się zakłócać. Należy znaleźć największy zbiór masztów, który nie będzie powodować zakłóceń.
Przykład:



Na powyższym rysunku białym kolorem zaznaczone są puste maszty, czerwonym maszty z antenami. Jak widać, żadne dwie anteny nie zakłócają się (czerwone wierzchołki nie są połączone ze sobą krawędziami).

Jaką złożoność ma Twój algorytm? Dla jak dużych instancji problemów (wielkość zbioru wierzchołków i krawędzi) Twój algorytm kończy pracę w zadowalającym czasie? Czy znasz jakieś inne szybsze rozwiązania? Poczytaj o aproksymacjach problemu, zaimplementuj jedną z nich. Podaj górne oszacowanie na błąd wybranej aproksymacji.

Zadanie 5: Sokrates i Platon (algorytmiczne)

Zadanie polega na rozwiązaniu następującej zagadki:
M i N są liczbami naturalnymi większymi od 1 i mniejszymi niż 100. Sokrates zna jedynie sumę S tych liczb, a Platon zna jedynie ich iloczyn P.

Platon: Nie wiem, o jakie liczby chodzi.
Sokrates: Wiedziałem, że nie będziesz wiedział jakie to liczby. Ja również nie wiem, jakie to liczby.
Platon: Teraz już wiem, jakie to liczby.
Sokrates: Ja też już wiem.

Napisz program, który rozwiązuje tę zagadkę. Omów własności liczb z których korzystasz. Omów dokładnie każdy krok Twojego rozumowania jak i implementacji.

Zadanie 6: Ranking trywialnych sortowań (eksperymentalne)

Zaimplementuj podstawowe wersje algorytmów sortowania:

1. bąbelkowe (bubble sort)
2. przez wstawianie (insertion sort)
3. przez wybór (selection sort)

Przygotuj odpowiednie benchmarki ze względu na stopień posortowania. Możesz wykorzystać do tego metodę shuffle Knutha. . Np pętla N / 10 odpowiada 10% nieposortowaniu.

Przeprowadź eksperymenty na hoście (linux) jak i realnym HW.
Spróbuj wyłonić najlepsze sortowanie dla HW.

Zadanie 7: Gramatyka bezkontekstowa (algorytmiczne)

Dane są nawiasy: (,), {, }, [,]. Napisz program, który sprawdzi czy nawiasy są dobrze dopasowane matematycznie.

Przykłady pozytywne: ({ } [ ]), ((([ ] { ( ) }))).
Przykłady negatywne: (( ), (([ ))].

Przygotuj naiwny algorytm oraz bardziej wyrafinowany. Przeprowadź testy obu algorytmów i porównaj ich szybkość działania.
Postaraj się wymyślić elegancki design kodu. Unikaj morza if-else.

Zadanie 8: Dopasowanie tekstu (algorytmiczne)

Istnieje wiele efektywnych algorytmów znajdujących wzór w tekście, np. algorytm KMP. Niestety takie algorytmy działają tylko dla znaków deterministycznych (czyli takich o stałym znaczeniu jak litery w alfabecie). Nie trudno się domyślić, że takie algorytmy nie wystarczają aby obsłużyć wyrażenia regularne, które są podstawą analizy tekstów!

Zaprojektuj algorytm, który sprawdza czy da się dopasować wzór do całego tekstu. Algorytm powinien obsługiwać 2 znaki specjalne:

1. * - zastępuje dowolną liczbę(w tym 0 znaków) dowolnych znaków
2. ? - zastępuje 1 dowolny znak

Przykłady:

```
1 const char* pattern1 = "xyxzzxy";
2 const char* text1 = "x***y";
3
4 /* output should be: Match */
5
6 const char* pattern2 = "xyzzxy";
7 const char* text2 = "x*zz?";
8
9 /* output should be: No match */
```

Jaka jest złożoność Twojego algorytmu? Użyj notacji duże O.

Zadanie 9: Listewki (algorytmiczne)

Po remoncie mieszkania, zostały Ci tzw ścinki listewek. Czyli jakieś niepotrzebne ucięte części listewek o różnej długości. Aby nie marnować pieniędzy chcesz wykorzystać te części do położenia listewki przy ostatniej ścianie. Niestety zapomniałeś, że oddałeś piłkę sąsiadowi i nie jesteś w stanie zmniejszyć już pozostałych części.

Zaprojektuj algorytm, który sprawdzi czy da się wykorzystać listewki pozostałe po remoncie.
Przykład:

```
1 int expectedLen = 14;
2 int lengths[] = {7, 3, 2, 5, 8, 12};
3
4 /* output should be: 2, 5, 7 */
```

Spróbuj zmodyfikować swój algorytm, tak aby jako drugie kryterium brał pod uwagę najmniejszą liczbę użytych listewek. Czyli powyższy przykład powinien wyglądać tak:

```
1 int expectedLen = 14;
2 int lengths[] = {7, 3, 2, 5, 8, 12};
3
4 /* output should be: 2, 12 */
```

Spróbuj zaprojektować efektywny algorytm. Porównaj go z wersja naiwną.

Zadanie 10: Podział zawodników (algorytmiczne)

W grach zespołowych, często występuje problem zbalansowania rozgrywki. Tzn takiego doboru drużyn aby gra była wyrównana. Załóżmy, że chcemy wprowadzić własną ligę Twojej ulubionej gry zespołowej, w której liczba zawodników może być różna. Ważne aby umiejętności zespołów były porównywalne.

Zaprojektuj algorytm, który sprawdza, czy z podanego zbioru zawodników, da się złożyć 2 drużyny o takich samych umiejętnościach. Jeśli się da wydrukuj na ekranie te drużyny.
Przykład:

```
1 int skills[] = {3, 1, 1, 2, 1, 2};
2
3 /* output should be: {3, 2}, {1, 1, 1, 2} */
```

Spróbuj ulepszyć swój algorytm, tak aby również wybierał jako drugie kryterium podobną liczbę zawodników. Czyli poprzedni przykład powinien wyglądać tak:

```
1 int skills[] = {3, 1, 1, 2, 1, 2};
2
3 /* output should be: {3, 1, 1}, {1, 2, 2} */
```

Porównaj swój algorytm z naiwnym podejściem.

Zadanie 11: Kombinacje z sumą (algorytmiczne)

Dana jest tablica t o rozmiarze n oraz stała S. Liczby w tej tablicy są losowe z całego dostępnego przedziału. Aby to uzyskać należy użyć funkcji random zamiast rand. Poniższy przykład pokazuje jak wygenerować losowy ciąg do zadania.

```c
#include <stdlib.h> /* random */
#include <time.h> /* time */

/* Example for S << N */
#define S 100
#define N 1000000

static unsigned int t[N];

int main(void)
{
  srandom((unsigned long)time(NULL));

  for (size_t i = 0; i < N; ++i)
    t[i] = (unsigned int)random();

  return 0;
}
```

Znajdź liczbę kombinacji takich i j, że $t[i] + t[j] < S$.
Przygotuj naiwny algorytm i wersje optymalną. Przeprowadź testy obu algorytmów i porównaj ich szybkość działania.
Dobierz $S << N$, $S = N$, $S >> N$. (symbol $<<$ oznacza istotnie mniejsze).

Przykład:

```c
#define S 100
#define N 4

static unsigned int t[N] = {0, 99, 0, 2};

/*
Result = 5
because:
t[0] + t[1] = 99
t[0] + t[2] = 0
t[0] + t[3] = 0
t[1] + t[2] = 99
t[2] + t[3] = 2
*/
```

Spróbuj pokazać, że Twój algorytm jest optymalny (nie ma lepszego według kryterium relacji duże O).
Kiedy warto jest używać Twojego algorytmu? Czy zawsze jest szybszy od naiwnego podejścia?

Omów dokładnie Twoje rozwiązanie. Podaj nazwy algorytmów, z których korzystasz.

Uważaj na pułapkę z overflowem na typie unsigned!

Zadanie 12: Równoległe wyszukiwania (eksperymentalne)

Zaimplementuj używając dowolnej znanej Ci biblioteki do zrównoleglania kodu (np pthreads albo omp) wyszukiwanie liniowe oraz wyszukiwanie binarne.
Porównaj wyniki z sekwencyjnymi algorytmami. Spróbuj odpowiedzieć na następujące pytania:

1. Który algorytm idealnie nadaje się do zrównoleglania?
2. Który algorytm posiada największy zysk po zrównolegleniu?
3. Czy opłaca się wyszukiwać klucze równolegle?

Zadanie 13: Sumy prefiksowe (inżynierskie)

Poczytaj o problemie sum prefiksowych np. na wikipedi i ważniaku.
Zaimplementuj algorytm równoległych sum prefiksowych. Możesz użyć do tego pthreads albo omp.
Pokaż zysk nad algorytmem sekwencyjnym.
Gdzie można wykorzystać ten algorytm?

Zadanie 14: Kodowanie liczb (algorytmiczne)

Stwórz algorytm do zakodowania kombinacji i permutacji liczb w jednej zmiennej całkowitoliczbowej (np. uint64_t). Zmierz liczbę bitów potrzebną na zakodowanie tych obiektów. Porównaj swój algorytm z naiwną metodą kodowania liczb jak pól bitowych.

```
1 #define N 5
2
3 /* Permutations examples */
4 int perm1[N] = {1, 2, 3, 5, 4};
5 int perm2[N] = {5, 2, 1, 3, 4};
6
7 /* Combinations examples */
8 int comb1[N] = {1, 4, 4, 4, 5};
9 int comb2[N] = {5, 2, 1, 3, 4};
```

Kodując permutacje możesz wzorować się na tym artykule.

Wykaż empirycznie poprawność swojego kodowania. Wygeneruj wszystkie możliwe kombinacje i permutacje np 5 elementowe. Pokaż że $decode(encode(X)) = X$. Sprawdź również czy nie ma kolizji. Tzn pokaż że jeśli $encode(X) = encode(Y)$ to $X = Y$.

Kiedy warto używać Twojego kodowania liczb, a kiedy lepsze są pola bitowe?
Zmierz narzut wynikający z używania pól bitowych i z kodowania / dekodowania.

Zadanie 15: Przewidywanie liczb losowych (algorytmiczne)

Pobaw się w hackera. Spróbuj przewidzieć liczby losowe generowane przez funkcję random() z języka C. Czy Twoja metoda ma jakieś założenia? Czy działa w 100%?

Wskazówka: Poczytaj o LCG (Linear congruential generator). Przeczytaj implementację funkcji random. Znajdź podobieństwa i różnice pomiędzy random a zwykłym LCG.

Zadanie 16: Szybkie sortowanie (eksperymentalne)

Zaimplementuj podstawową wersję algorytmu sortowania szybkiego oraz wybierz jakieś sortowanie naiwne (np. sortowanie przez wstawianie).
Przedstaw eksperymenty pokazujące zysk wynikający z używania quicksorta.
Czy zawsze quicksort jest szybszy od wybranego przez Ciebie sortowania?
Porównaj wyniki z funkcją biblioteczną qsort.

Poczytaj o lepszej wersji quicksorta: dual pivot quicksort, spróbuj zaimplementować tę wersję i przeprowadzić te same eksperymenty.
Czy w kwestii przydatności sortowań naiwnych coś się zmieniło?

Zadanie 17: Printf i scanf (inżynierskie)

Stwórz własną implementację funkcji printf oraz scanf.
Zacznij od określenia funkcjonalności. Nie musisz implementować wszystkiego, ale postaraj się obsłużyć najważniejsze zadania tych funkcji.
Do przesyłania danych pomiędzy programem a konsolą używaj tylko funkcji read i write.
Poczytaj o variadic arguments. Do obsługi parametrów funkcji możesz wzorować się na tym przykładzie:

```
#include <stdarg.h>

/* sum takes Int, Long, Int, Long, Int ... */
long sumIntLongInt(int numArgs, ...)
{
    va_list args;
    long sum = 0;

    /* init args starting from parameter numArgs */
    va_start(args, numArgs);

    for(int i = 0; i < num_args; ++i)
        if (i % 2 == 0) /* Even pos is Int: 0, 2, 4 ... */
            sum += (long)va_arg(args, int);
        else /* odd pos is Long: 1, 3, 5 ... */
            sum += (long)va_arg(args, long);

    va_end(args);
    return sum;
}
```

Zadanie 18: Przenikanie obrazu (inżynierskie)

Poczytaj o problemie blending alpha. Zaimplementuj algorytm w języku C oraz w asemblerze przy użyciu instrukcji SIMD.
W szczególności zwróć uwagę na instrukcje: punpcklbw, pmullw, paddw, psrlw i packuswb.
Przygotuj jakieś benchmarki aby porównać performace.

Aby zaimplementować algorytm, potrzebujesz biblioteki graficznej. Możesz użyć SDL jest bardzo prosta i przejrzysta.

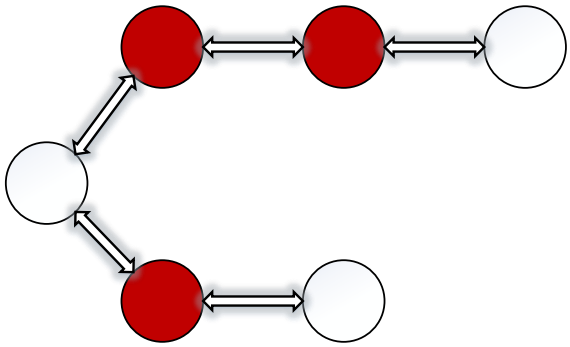
Czy uważasz, że warto implementować takie algorytmy w asemblerze?

Zadanie 19: Stacje pociągów (algorytmiczne)

Władze regionu chcą zmodernizować swoją sieć pociągów regionalnych. Przeprowadzili ankiety, wywiady środowiskowe oraz analizy, dzięki czemu znając potrzeby ludzi.

Stworzyli graf $G(V, E)$ modelujący te potrzeby. Wierzchołkami $v \in V$ są miasta regionu, które znalazły się w ankiecie jako miejsce startowe lub końcowe trasy pociągu, krawędziami $e = (v, v') \in E$ są relację między miastami, do których muszą jeździć pociągi.

Zaprojektuj algorytm, który wybierze miasta, w których władze regionu muszą wybudować większe perony (na stacje końcowe), tak aby każda relacja między miastami była incydentna do przynajmniej jednej stacji końcowej.
Przykład:



Powyższy rysunek przedstawia przykładowy wynik ankiet. Wybrane wierzchołki zaznaczone są kolorem czerwonym. Jak widać każda krawędź sąsiaduje z wybranym wierzchołkiem, zatem jest to prawidłowe rozwiązanie. Nie trudno również wykazać, że nie da się wybrać tylko 2 wierzchołków (mamy 5 krawędzi, każdy wierzchołek ma stopień 2, więc potrzeba przynajmniej $\lceil \frac{5}{2} \rceil$ wierzchołków by spełnić wymogi zadania).

Jaką złożoność ma Twój algorytm? Dla jak dużych instancji problemów (wielkość zbioru wierzchołków i krawędzi) Twój algorytm kończy pracę w zadowalającym czasie? Czy znasz jakieś inne szybsze rozwiązania? Poczytaj o aproksymacjach problemu, zaimplementuj jedną z nich. Podaj górne oszacowanie na błąd wybranej aproksymacji.

Zadanie 20: Logarytm dyskretny (eksperymentalne)

Poczytaj o protokole Diffiego-Hellmana. Próba złamania tego protokołu redukuje się do próby obliczenia logarytmu dyskretnego.

Zaimplementuj algorytmy Rho, kangaroo oraz Pohlinga.

Przygotuj benchmarki pokazujące czas potrzebny na złamanie protokołu w zależności od wielkości liczby pierwszej. Do przetrzymywania dużych liczb pierwszych możesz użyć biblioteki gmp.

Czy możliwe jest złamanie protokołów z użyciem 4000 bitowych liczb pierwszych?

Zadanie 21: Różnice w wyrazach (algorytmiczne)

Zaprojektuj prosty difftool. Który będzie porównywał 2 linie tekstu a następnie wydrukuje na ekranie najmniejszą liczbę zmian jaka powinna zostać dokonana aby zmienić jedną linie w drugą.
Przykład:

```
const char* line1 = "XMJYAUZ";
const char* line2 = "XMJAATZ";

/*
    Output:
    X M J -Y A -U +A +T Z

    Let's check this:
    XMJY(-Y)AU(-U)(+A)(+T)Z = XMJAATZ
    So output was correct, because we have made 2nd line
    from 1st line using output.
*/
```

Spróbuj zaprojektować efektywny algorytm. Porównaj go z wersja naiwną.

Zadanie 22: Operacje na pamięci (eksperymentalne)

Stwórz własne implementacje funkcji memset i memcpy w języku C.
Porównaj szybkość wykonywania z funkcjami bibliotecznymi ze string.h.
Porównaj również zerowanie jak i kopiowanie wbudowane w język C.

```
#include <string.h>

void *my_memset(void *s, int c, size_t n)
{
    /* simple loop maybe with some tricks */
}

void *memcpy(void *dest, const void *src, size_t n)
{
    /* simple loop maybe with some tricks */
}

/* 1st benchmark */
struct S s = {0};
memset(&s, 0, sizeof(s));
my_memset(&s, 0, sizeof(s));

/* 2nd benchmark */
struct S s1, s2;
s2 = s1;
memcpy(&s2, &s1, sizeof(s2));
my_memcpy(&s2, &s1, sizeof(s2);
```

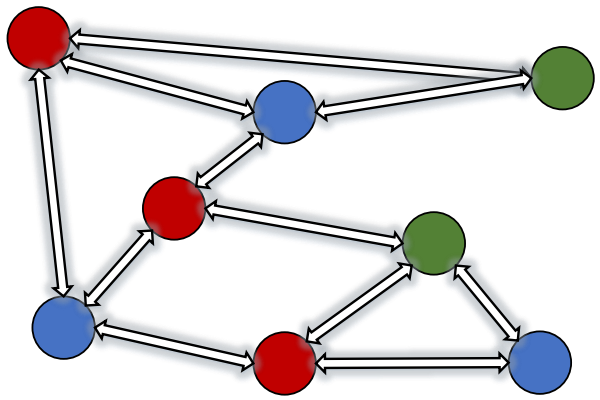
Przeanalizuj kod asemblera Twojej memset oraz tej bibliotecznej z glibc. Czy potrafisz wytłumaczyć wyniki eksperymentów?

Zadanie 23: Wielkie korporacje (algorytmiczne)

Od wielu lat trwa wojna gospodarcza pomiędzy 3 wielkimi sieciami handlowymi. Każda z nich chce sklep w galeriach handlowych pewnego dużego miasta. Naszym zadaniem jest przygotować rozmieszczenie sklepów 3 korporacji tak aby przynosiły duże zyski. Według analityków 2 sklepy w galeriach handlowych, które są blisko siebie nie sprzyjają maksymalizacji zysków. Zatem musimy to zapewnić.

Problem da się zamodelować w teorii grafów. Tworzymy graf $G(V, E)$. Wierzchołkami $v \in V$ niech będą galerie handlowe, w których będzie można wynająć lokal na sklep. Jeśli galerie są zbyt blisko siebie to tworzymy krawędź $e = (v, v') \in E$. Zatem musimy przyporządkować wierzchołki $v \in V$ 3 zbiorom tak aby żaden wierzchołek nie miał sąsiada należącego do tego samego zbioru.

Przykład:



Powyższy rysunek przedstawia przykładową sytuację. Każdy kolor oznacza zbiór innej sieci handlowej. Zauważmy, że żaden z wierzchołków tego samego koloru nie jest połączony, czyli korporacja nie jest właścicielem 2 sklepów w pobliskich galeriach.

Jaką złożoność ma Twój algorytm? Dla jak dużych instancji problemów (wielkość zbioru wierzchołków i krawędzi) Twój algorytm kończy pracę w zadowalającym czasie? Czy znasz jakieś inne szybsze rozwiązania? Poczytaj o aproksymacjach problemu, zaimplementuj jedną z nich. Podaj górne oszacowanie na błąd wybranej aproksymacji.

Zadanie 24: Prosta powłoka systemowa (inżynierskie)

Zaimplementuj w języku C prostą wersję powłoki systemowej. Niech zachowuje się jak prawdziwa powłoka, tzn niech odczytuje linię ze standardowego wejścia i wykonuje podany program. Pamiętaj o ustawieniu argumentów wykonywanej komendy. Jeśli linia kończy się znakiem (&), wtedy powłoka nie powinna czekać aż komenda zostanie skończona i od razu wrócić do trybu oczekiwania na rozkazy. W innym przypadku powłoka powinna zaczekać, aż program się wykona. Jeśli czujesz się na siłach zaimplementuj także pipe |.

Zadanie 25: Prosty debugger (inżynierskie)

Napisz prosty debugger. Wykorzystaj do tego funkcję ptrace.
Zaimplementuj przynajmniej te funkcje:

1. Zatrzymanie wykonywania kodu programu i pobieranie instrukcji z konsoli (aby wprowadzić komendy)
2. Wykonywanie pojedynczej instrukcji w asemblerze za pomocą polecenia step
3. Wydrukowanie zawartości rejestrów za pomocą polecenia show regs
4. Zmiana zawartości pojedynczego rejestry za pomocą polecenia set reg (nazwa) (value) np. set reg ebx 41
5. Kontynuacja programu bez trybu debuggowego za pomocą polecenia continue

Możesz skorzystać z tego artykułu.
Przetestuj swój debugger na prostym programie w języku C.

Zadanie 26: Maksymalny ciąg podskoków (algorytmiczne)

Przypomnij sobie grę flappy birds.
Załóżmy, że chcemy napisać dodatkową statystkę wyświetlaną na końcu gry. Statystyka ma pokazywać skoki w górę, które dają największą sumę wysokości.

Przykład:
Wysokości skoków: 0, 8, 4, 12
Czyli przy uruchomieniu gry jesteśmy na wysokości 0. Użytkownik wykonał dobry ruch i znalazł się na wysokości 8. Lecz później aby uniknąć przeszkody musiał zlecieć na wysokość 4 aby zakończyć grę na wysokości 12.
Po takiej grze statystyka powinna wyświetlić 0, 8, 12. Ponieważ to ciąg skoków w górę, która ma maksymalną sumę wyokości.

Poniżej mamy o wiele trudniejszy przykład:

```
1  int jumps[] = {0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11};
2
3  /*
4      output should be: 0, 8, 12, 14
5      because sum of those 3 jumps are equal 34,
6      i.e others sequences like
7      {0, 4, 12, 14} = 30,
8      {0, 2, 10, 14} = 26,
9      {0, 2, 6, 13} = 21
10     have lower sum
11 */
```

Zadanie polega na zaprojektowaniu efektywnego algorytmu liczącego tę statystykę. Porównaj go z wersją naiwną.

Zadanie 27: Problemy z hashowaniem (eksperymentalne)

Jednym z największych problemów wynikających z używania funkcji hashującej są kolizje. Wybierz 2-3 niekryptograficzne funkcje hashujące (możesz je skopiować stąd).

Przygotuj benchmark pokazujący liczbę kolizji.
Wykorzystaj do tego duży zbiór danych np. 100tys.
Użyj różnej długości danych, np int, long, char[32].

Czy jeśli zmienisz rozmiary dziedzin zachowując ich stosunek to wyniki się zmienią (pogorszą / poprawią?).
Normalnie stosunek był następujący: $\frac{2^{32}-1}{100000} = 42950$, przygotuj więc kolejny benchmark gdzie danych będzie 1000. A wynik funkcji hashującej umieść w pierścieniu $Z_{42950000}$ czyli wykonaj działanie $H(x) = H(x) \; mod \; 42950000$.

Czy wyniki się pogorszą gdy zmniejszymy stosunek? Np. Liczba danych równa 1000. Pierścień Z_{100000}?

Zadanie 28: Porównywanie plików (algorytmiczne)

Napisz algorytm do przeszukiwania obecnego katalogu w celu znalezienia pliku z dokładnie tą samą zawartością co podany jako argument plik.

Przykład:

```
$ls
f1.txt
f2.txt
f3.txt
$cat f1.txt
Ala ma kota
$cat f2.txt
Kot ma Ale
$cat f3.txt
Ala ma kota
$findTheSameFile.out f1.txt
Found 1 file: f3.txt
$findTheSameFile.out f2.txt
Found 0 file:
```

Jeśli masz pomysł na kilka rozwiązań zaimplementuj przynajmniej 2 z nich i przygotuj benchmarki pokazujące przydatność Twoich algorytmów.

Zadanie 29: Obliczanie długości słowa (eksperymentalne)

Napisz prostą implementację strlen.
Przeprowadź eksperymenty (na różnych długościach wyrazów, sięgających nawet 10000 znaków) pokazujące wyższość funkcji bibliotecznej nad Twoją.

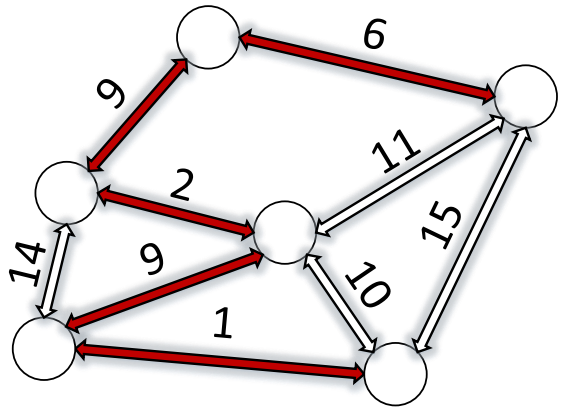
Zapoznaj się z implementacją glibc. Spróbuj zrozumieć i wytłumaczyć optymalizacje wykonywane w ich kodzie.

Wybierz 1-2 proste tricki, które poznałeś podczas studiowania ich implementacji. Dodaj je do swojej funkcji i powtórz eksperymenty.

Zadanie 30: Internet światłowodowy (algorytmiczne)

Pewna firma chce zapewnić dostęp do światłowodu wszystkim mieszkańcom w mieście. Jak zawsze firma chce to wykonać najmniejszym możliwym kosztem. Analitycy przygotowali mapę miasta z zaznaczonymi rozdzielnicami internetu oraz kosztem instalacji połączenia pomiędzy nimi. Musimy zatem połączyć wszystkie miejsca najmniejszym kosztem.

Całość można przedstawić jako graf $G(V, E)$. Wierzchołkami grafu $v \in V$ są miejsca, do których trzeba doprowadzić internet. Krawędziami $e = (v, v') \in E$ są drogi, którymi można pociągnąć światłowód. Każda krawędź $e \in E$ ma przypisany koszt stworzenia takiego połączenia.
Przykład:



Powyższy rysunek przedstawia przykładową sytuację. Jak widać udało stworzyć się połączenie pomiędzy wszystkimi miejscami kosztem 27. Jest to najmniejszy koszt jaki można uzyskać.

Rozwiąż ten problem! Jaką złożoność ma Twój algorytm?

Zadanie 31: Zasięg postów w sieciach społecznościowych (algorytmiczne)

Sieci społecznościowe jak facebook czy twitter są bardzo ważnymi środkami przekazu informacji. Dlatego zasięg postu jaki umieszcza dana osoba to podstawowy wyznacznik popularności oraz zarobków tzw. influencerów.

Naszym zadaniem jest policzenie takiej statystyki i wyłonienie lidera grupy. Czyli osoby, która ma największy zasięg swoich postów. Z pomocą przychodzi do nas teoria grafów! Sieć znajomych to graf $G(V, E)$. Wierzchołkami w tym grafie $v \in V$ są użytkownicy. Gdy 2 użytkowników są znajomymi to dodajemy krawędź $e = (v, v') \in E$ do grafu.
Relację znajomości możemy uznać za przechodnią (tak właśnie działa facebook). Tzn. znajomy Twojego znajomego jest moim znajomym jeśli i my się znamy. W rzeczywistości jest pewien limit na przechodniość tej relacji np. 2. Co oznacza, że łańcuch nie dosłownej znajomości może mieć tylko długość 3. (znajomy znajomego mojego znajomego nie jest już moim znajomym).

Rozwiąż ten problem! Pobaw się omówionym parametrem, zobacz jak mocno to wpływa na zasięg postów i długość trwania Twojego algorytmu.

Zadanie 32: Xor Lista (inżynierskie)

Xor Lista ($\oplus$-Lista) była wykorzystywana do zapisywania listy kontaktów w urządzeniach, które miały bardzo mało pamięci. Lista ta nie różni się zbytni od listy dwukierunkowej. Korzysta ona z własności funkcji xor ($\oplus$) aby zakodować 2 wskaźniki w jednym. Jeśli chcesz poczytać o niej więcej to polecam te źródła: linux, wikipedia.

Definicja (nazywana w C deklaracją) struktury powinna wyglądać następująco.

```
1  /* Normal Double Linked List  */
2  typedef struct DLLNode
3  {
4      struct DLLNode* prev;
5      struct DLLNode* next;
6      int data;
7  } DLLNode;
8
9  typedef struct DoubleList
10 {
11     DLLNode* head;
12     DLLNode* tail;
13     size_t len;
14 } DoubleList;
15
16 /* Xor List */
17 typedef struct XLNode
18 {
19     struct XLNode* prev_next;
20     int data;
21 } XLNode;
22
23 typedef struct XorList
24 {
25     XLNode* head;
26     XLNode* tail;
27     size_t len;
28 } XorList;
```

Skorzystaj z własności funkcji $\oplus$:

- Istnienie elementu neutralnego: $X \oplus 0 = X$
- Odwrotność: $X \oplus X = 0$
- Przemienność: $X \oplus Y = Y \oplus X$
- Łączność: $X \oplus (Y \oplus Z) = (X \oplus Y) \oplus Z$

Zaimplementuj najważniejsze funkcje listy. Pokaż jak wykonać wszystkie podstawowe operacje na liście dwukierunkowej mając tylko jeden wskaźnik.

Czy opłaca się używać xor Listy?
Czego nie da się zrobić mając xor listę a da się mając zwykłą listę dwukierunkową?

Zadanie 33: Lista online (eksperymentalne)

Struktury danych noszące miano online (lub samoorganizujących się) to struktury, które potrafią się dostosować do kolejnych kwerend użytkownika bez jego ingerencji (wywołania jakieś funkcji, która przereorganizuje strukturę). Czyli należy przygotować taki algorytm, który potrafi przeanalizować wcześniejsze zapytania i dostosować strukturę do następnych. Jeśli chcesz poczytać więcej o takich algorytmach to polecam tę książkę.
W tym zadaniu należy przygotować 4 różne warianty listy.

1. **Normal** - normalna lista jednokierunkowa
2. **Move to Front Method** - element wyszukany staje się elementem początkowym listy
3. **Count Method** - elementy są sortowane ze względu na częstotliwość wyszukiwań.
4. **Transpose Method** - elementy wyszukany jest zamieniany ze swoim poprzednikiem.

Struktura powinna wyglądać następująco:

```
1  typedef enum listType_t
2  {
3      LIST_TYPE_NORMAL,
4      LIST_TYPE_MOVE_TO_FRONT_METHOD,
5      LIST_TYPE_COUNT_METHOD,
6      LIST_TYPE_TRASNPOSE_METHOD,
7  } listType_t;
8
9  typedef struct ListNode
10 {
11     struct ListNode* next;
12     int data;
13 } ListNode;
14
15 typedef struct List
16 {
17     ListNode* head;
18     ListNode* tail;
19     size_t len;
20
21     listType_t type;
22 } List;
```

Aby lepiej zrozumieć powyższe metody możesz poczytać o nich na wikipedii lub w tej prezentacji.

Opowiedz krótko o tych metodach. Spróbuj znaleźć ich wady i zalety.
Przygotuj benchmarki porównujące te 4 metody ze sobą.

Czy można wyłonić jednogłośnie zwycięzce?

Zadanie 34: Drzewo online (SPLAY) (eksperymentalne)

Struktury danych noszące miano online (lub samoorganizujących się) to struktury, które potrafią się dostosować do kolejnych kwerend użytkownika bez jego ingerencji (wywołania jakieś funkcji, która przereorganizuje strukturę). Czyli należy przygotować taki algorytm, który potrafi przeanalizować wcześniejsze zapytania i dostosować strukturę do następnych. Jeśli chcesz poczytać więcej o takich algorytmach to polecam tę książkę.
Struktury nieposortowane po kluczach, łatwo sortować po innych argumencie. Jednak drzewo binarne jest posortowane. Zatem trzebaby wykonać sortowanie dwuwymiarowe. Okazuje się, że większość metod wymyślonych dla list jest bezużyteczna.

Dlatego wymyślono drzewo SPLAY. Jest to samoroganizujące się drzewo, które po zakończonym wyszukiwaniu wykonuje rotacje, tak aby wyszukany wcześniej węzeł znalazł się bliżej korzenia.
Zadanie polega na zaimplementowaniu drzewa SPLAY oraz przeprowadzeniu eksperymentów, które porównają go ze zwykłym drzewem BST oraz z drzewem czerwono-czarnym (RBT).
Możesz wzorować się na tych implementacjach: BST, RBT oraz SPLAY.
Struktura powinna wyglądać podobnie do tej:

```
1  typedef struct Node
2  {
3      struct Node* left;
4      struct Node* right;
5      struct Node* up;
6
7      int data;
8
9      /* some other node variables like color, counter etc */
10 } Node;
11
12 typedef struct Tree
13 {
14     Node* root;
15     size_t entries;
16 } Tree;
```

Czy warto używać drzewa SPLAY? Jakie są jego wady? Jakie są zalety i wady innych drzew?
Dlaczego we wszystkich popularnych językach mapa implementowana jest za pomocą RBT?

Zadanie 35: Silnia (eksperymentalne)

Chcemy policzyć $n!$ dla dużych liczb. Powiedzmy, że maksymalny rozmiar $n!$ to 128 bitów.
Napisz kilka wersji tego programu:

1. Do reprezentacji liczby użyj uint64_t[2], a kod napisz w języku C.
2. Do reprezentacji liczby użyj __uint128_t (działa tylko w trybie gnu), a kod napisz w języku C.
3. Przy użyciu 128bitowych rejestrów i asemblerowych instrukcji SSE2 (możesz wzorować się na tym artykule.
4. Przy użyciu biblioteki gmp

Przeprowadź eksperymenty i pokaż który algorytm jest najszybszy.
Wybierz najlepszy algorytm. Tzn taki który jest szybki, prosty i jednocześnie szybki do napisania.

Zadanie 36: Bankomat (algorytmiczne)

Zaprojektuj algorytm wydający pieniądze z bankomatu wykorzystując przy tym minimalną liczbę banknotów. Pamiętaj, że liczba banknotów o różnych nominałach jest ograniczona. Gdy brakuje banknotów na wydanie pieniędzy należy wyświetlić odpowiedni komunikat.
Przykład:

```
1 int cash = 1380;
2 int nominals[] = {10, 20, 50, 100, 200, 500};
3 int available[] = {3, 2, 9, 10, 2, 1}
4 /* output should be: 500x1, 200x2, 100x4, 1x50, 1x20, 1x10 */
```

Czy Twój algorytm działa również na innych nominałach?
Sprawdź czy Twój algorytm zadziała na poniższym przykładzie:

```
1 int cash = 33;
2 int nominals[] = {1, 15, 25};
3 int available[] = {5, 5, 5}
4 /* output should be: 2x 15, 3x 1 */
```

Jeśli Twój algorytm działał dla pierwszego przykładu, ale nie zadziałał dla drugiego, tzn że byłeś zbyt zachłanny i algorytm działał tylko dla matroidów!

Zadanie 37: Pamięć podręczna komputera (eksperymentalne)

Pamięć cache jest to podręczna pamięć umieszczona na procesorze. Gdyby nie ona, nasze programy działały by około 20 razy wolniej. Dlatego algorytmy zarządzające tą pamięciom są tak ważne.

Nie trudno się domyślić, że użyte struktury danych jak i ich ułożenie w pamięci jest kluczowe w uzyskaniu odpowiedniej szybkości naszego kodu. Dlatego bardzo ważne jest aby podczas pisania kodu, starać się tak zarządzać pamięciom aby mieć jak najmniej doczytywania nowych fragmentów pamięci RAM do pamięci cache (tzw cache-friendly memory layout).

Istnieją programy, które dzięki HW monitorom potrafią mierzyć faktyczne statystki dotyczące pamięci cache jak np. perf. Niestety często pracujemy na wirtualnych maszynach. Wtedy najlepiej użyć programu cachegrind z pakietu valgrind.

Napisz kilka przykładów funkcji, które idealnie używają pamięci cache. Oraz kilka takich, które co chwila doczytują dane z pamięci RAM. Pokaż wynik wyżej wymienionych programów.

Zadanie 38: Automatyczna weryfikacja (inżynierskie)

Dzięki modelowi automatów nieskończonych takich jak automaty Buchiego możliwe jest sprawdzenie poprawności kodu. Nie są to testy, które sprawdzają tylko pojedyncze przypadki. Są to pełne dowody matematyczne postawionych hipotez!

Aby ułatwić pracę nad sprawdzaniem poprawności stworzono język promela i weryfikator spin interpretujący ten język. Niestety nadal nakład pracy związany z utrzymywaniem modeli jest zbyt duży, aby firmy decydowały się na użycie weryfikatora spin. Obecnie głównymi użytkownikami tego programu jak i języka promela są informatycy teoretyczni, matematycy, fizycy oraz wojsko.
Jeśli jesteś ciekaw jak sprawdzić kod w języku C bez przepisywania go na język promela, możesz przeczytać te artykuły: Verifying Multi-threaded C Programs with SPIN, Model Checking C Programs by Translating C to Promela.

Zadanie polega na zaimplementowaniu systemu opisanego poniżej i zweryfikowaniu postawionej tezy.
System:
Modelujemy proces zmieniania się koloru kameleonów na ich wyspie. Zbadano obszar gdzie żyło 13 żółtych, 16 zielonych oraz 15 czerwonych kameleonów. Naukowcy zauważyli, że jeśli 2 kameleony o różnych kolorach spotkają się to zmieniają kolor na ten trzeci. Przykładowo jeśli spotka się kameleon żółty z zielonym to obaj zmienią kolor na czerwony. Dodatkowo zauważono, że wszystkie pozostałe interakcje nie wpływały na zmianę kolorów. Więc specjalnie doprowadzano tylko do spotkań 2 kameleonów (niekoniecznie o różnych kolorach), zatem symulacja powinna przyjąć, że w spotkaniach zawsze uczestniczą 2 kameleony.

Naukowcy chcą sprawdzić czy:

1. Dojdzie do sytuacji, w której wszystkie kameleony będą miały ten sam kolor.
2. Dojdzie do sytuacji, w której 2 populacje kameleonów o różnych kolorach będą miały tyle samo przedstawicieli

Wykonaj model wyspy kameleonów i ich spotkań w języku promela. Odpowiedz na pytania naukowców weryfikując postawione hipotezy programem spin. Możesz użyć do tego techniki never claim.
Możesz wzorować się na przykładowej implementacji umieszczonej w dodatku 3.3.

Zadanie 39: Wzajemne wykluczanie (eksperymentalne)

Problem wzajemnego wykluczania to fundamentalny problem programowania współbieżnego. Gdyby go nie rozwiązano, nie mielibyśmy programów wielowątkowych!

Zazwyczaj ten problem rozwiązuje się sprzętowo. Jednak są algorytmy, które potrafią rozwiązań ten problem algorytmicznie. Do najprostszych należą algorytm Dekkera i algorytm Petersona.

Zaimplementuj te algorytmy i przeprowadź na nich stress testy pokazujące ich poprawność. Możesz (nie musisz!) także udowodnić te algorytmy w języku promela. Możesz to zrobić na podstawie przykładu z dodatku 3.4, który dowodzi poprawności algorytmu Dijkstry.

Co jest lepsze zwykły mutex czy zabawa w takie algorytmy? Porównaj czas wykonywania programu opartego na algorytmie Dekkera, Petersona oraz mutexów z biblioteki pthreads.

Zadanie 40: Gra w węża (algorytmiczne)

Pewnie większość pamięta prostą grę w węża. Nie wiele osób zna jednak jej wersję logiczną (tamta była tylko zręcznościowa).

Zasady gry

- Gracz widzi całą planszę od razu. Plansza jest to macierz kwadratowa, zawierająca pola liczbowe,
- W pierwszym ruchu gracz wybiera dowolne miejsce w pierwszym wierszu macierzy, gdzie rozpocznie rozgrywkę,
- Gracz może poruszać się tylko między polami(góra, dół, lewo, prawo), różniącymi się o 1,
- Gra polega na przejściu przez macierz najdłuższą drogą

Przykład:

```
#define N 5

const int board[N][N] =  { { 7, 5, 2, 3, 1 },
                           { 3, 4, 1, 4, 4 },
                           { 1, 5, 6, 7, 8 },
                           { 3, 4, 5, 8, 9 },
                           { 3, 2, 2, 7, 6 },
                         };

/*
    output should be:
    _, 5, _, _, _
    _, 4, _, _, _
    _, 5, 6, 7, _
    _, _, _, 8, _
    _, _, _, 7, 6
*/
```

Zaprojektuj algorytm, który zawsze wygrywa, czyli zawsze znajduje najdłuższą drogę.

3 Dodatek

3.1 Benchmark

```c
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

#define CLOCK_TYPE CLOCK_MONOTONIC

#define MEASURE_FUNCTION(func, label) \
    do { \
        struct timespec start; \
        struct timespec end; \
        double timeTaken; \
        \
        clock_gettime(CLOCK_TYPE, &start); \
        (void)func; \
        clock_gettime(CLOCK_TYPE, &end); \
        \
        timeTaken = (double)(end.tv_sec - start.tv_sec) * 1e9; \
        timeTaken = (double)(timeTaken + (double)(end.tv_nsec -
   start.tv_nsec)) * 1e-9; \
        \
        printf(label " time = %lf[s]\n", timeTaken); \
    } while (0)

extern int fib(int n);

int main(void)
{
    MEASURE_FUNCTION(fib(45), "Fibonacci(45)");

    return 0;
}
```

3.2 Makefile

Zakładamy że posiadamy następującą strukturę katalogów:

- **./** - katalog głowny projektu gdzie trzymamy plik Makefile, README oraz inne pliki konfiguracyjne.
- **./src** - katalog zawierający kod źródłowy wszystkich plików prócz pliku zawierającego funkcję main.
- **./app** - katalog zawierający kod źródłowy pliku z funkcją main.
- **./inc** - katalog zawierający pliki nagłówkowe.

Wtedy przykładowy plik Makefile może wyglądać następująco.

```
# Shell commands
RM          := rm -rf

# Compiler setting (default it will be gcc)
CC          ?= gcc

# Enable max optimization
CC_OPT      := -O3

# Maybe some flags are duplicated, but who cares
CC_WARNINGS := -Wall -Wextra -pedantic -Wcast-align \
               -Winit-self -Wmissing-include-dirs \
               -Wredundant-decls -Wshadow -Wstrict-overflow=5 \
               -Wundef -Wwrite-strings -Wpointer-arith \
               -Wmissing-declarations -Wuninitialized \
               -Wold-style-definition -Wstrict-prototypes \
               -Wmissing-prototypes -Wswitch-default \
               -Wbad-function-cast -Wnested-externs \
               -Wconversion -Wunreachable-code \

ifeq ($(CC),$(filter $CC), gcc cc))
CC_SYM      :=  -rdynamic
CC_STD      :=  -std=gnu99
else ifeq ($(CC),clang)
CC_SYM      :=  -Wl,--export-dynamic
CC_WARNINGS +=  -Wgnu -Weverything -Wno-newline-eof \
                -Wno-unused-command-line-argument \
                -Wno-reserved-id-macro -Wno-documentation \
                -Wno-documentation-unknown-command \
                -Wno-padded
CC_STD      :=  -std=c99
endif

CC_FLAGS    := $(CC_STD) $(CC_WARNINGS) $(CC_OPT) $(CC_SYM)

PROJECT_DIR := $(shell pwd)

# To enable verbose mode type make V=1
ifeq ("$(origin V)", "command line")
  VERBOSE = $(V)
endif

ifndef VERBOSE
  VERBOSE = 0
endif

ifeq ($(VERBOSE),1)
  Q =
else
  Q = @
endif

```

```
define print_info
  $(if $(Q), @echo "$(1)")
endef

define print_make
  $(if $(Q), @echo "[MAKE]     $(1)")
endef

define print_cc
  $(if $(Q), @echo "[CC]       $(1)")
endef

define print_bin
  $(if $(Q), @echo "[BIN]      $(1)")
endef


IDIR := $(PROJECT_DIR)/inc
SDIR := $(PROJECT_DIR)/src
ADIR := $(PROJECT_DIR)/app

SRCS := $(wildcard $(SDIR)/*.c) $(wildcard $(ADIR)/*.c)
OBJS := $(SRCS:%.c=%.o)
DEPS := $(wildcard $(IDIR)/*.h)

# Put here all needed libraries like math, pthread etc
LIBS := -lm

# Type here name of your output file
EXEC := $(PROJECT_DIR)/main.out

all: $(EXEC)

%.o: %.c
  $(call print_cc, $<)
  $(Q)$(CC) $(CC_FLAGS) -I$(IDIR) -c $< -o $@

$(EXEC): $(OBJS)
  $(call print_bin, $@)
  $(Q)$(CC) $(CC_FLAGS) -I$(IDIR) $(OBJS) $(LIBS) -o $@

clean:
  $(call print_info,Cleaning)
  $(Q)$(RM) $(OBJS)
  $(Q)$(RM) $(EXEC)
```

3.3 Przykład weryfikatora jednowątkowego w języku promela

Model implementuje proces, w którym losujemy zawsze 2 kule. Jeśli kule są tego samego koloru to dokładamy kulę czarną do puli, w przeciwnym razie dokładamy kule białą.
Model ma za zadanie sprawdzić czy gdy zaczynamy z 150 kulami czarnymi i 75 kulami białymi to zawsze skończymy z jedną kulą białą?
Prawdopodobieństwo wylosowania się kul jest jednostajne.

```
/*
    System:
    We have 150 black balls and 75 white balls

    Process:
    Get 2 balls
    if (balls have the same color)
        add black ball to urn
    else
        add white ball to urn

    CHECK:
        Always at the end we have 1 white ball

    Program:
        Check assert at the end of process

    Command:
        spin -run urns.pml
*/

active proctype urns()
{
    int blacks = 150;
    int whites = 75;
    do
    :: (blacks + whites > 1) ->
        if
        ::  (blacks >= 2) -> blacks--
        ::  (whites >= 2) -> whites = whites - 2; blacks++
        ::  (whites >= 1 && blacks >= 1) -> blacks--;
        fi

    :: else -> break;
    od

    assert(blacks == 0 && whites == 1);
}
```

3.4 Przykład weryfikatora wielowątkowego w języku promela

```
/*
    Algorithm: Dijkstra's Mutual exclusion
    CHECK:
       1) Only 1 process can execute crit section code
    Program:
       1) Never Claim is used. This will check assert in every step of program
    Command:
        spin -T -u10000 -g mutual_exclusion.pml
*/
#define N 8
int values[N]
chan msg[N] = [0] of {int}
int crit = 0
#define START_JOB() crit = crit + 1;
#define END_JOB() crit = crit - 1

active proctype thread_first()
{
    int val
    int id = _pid
    assert(id == 0)
    do
    ::  msg[id] ? val ; if
                        :: (val == values[id]) -> START_JOB();
    values[id] = (values[id] + 1) % (N + 1) ; END_JOB();
                        :: else -> skip
                        fi
    ::  msg[id + 1] ! values[id]
    od
}

active[N - 1] proctype thread()
{
    int val
    int id = _pid
    assert(id > 0 && id < N)
    do
    :: msg[id] ? val ;  if
                        :: (val != values[id]) -> START_JOB();
    values[id] = val ; END_JOB();
                        :: else -> skip
                        fi
    :: msg[(id + 1) % N] ! values[id]
    od
}
never
{
   do
   :: assert(crit <= 1)
   od
}
```

Powyższy model sprawdza algorytm wykluczania się opracowany przez Dijkstre. Algorytm ten zawiera dodatkowo piękną własność samostabilizacji, czyli jeśli kiedyś przypadkowo (np. przez bitflip) dojdzie do błędu w systemie. Czyli do sytuacji, w której więcej niż 1 wątek będzie korzystał z sekcji krytycznej to w skończonej liczbie kroków algorytm sam się naprawi i dalej będzie bronił sekcji krytycznej.
Jego schemat działania jest bardzo prosty. Ustawiamy wątki w pierścień według ich *id*. Zamieniamy wtedy ich *id* na liczby porządkowe, tak aby sąsiad z prawej miał większe *id* tylko o 1, a sąsiad z lewej mniejsze o 1. Wyróżniamy 2 rodzaje wątków, wątek pierwszy i pozostałe. Wątek pierwszy rusza się wtedy i tylko wtedy gdy wątek ostatni posiada taką samą wartość *val*. Zwiększa on wtedy swoją wartość *val* o 1. Pozostałe wątki ruszają się tylko gdy ich wartość *val* jest różna od wartości *val* poprzednika (lewego sąsiada). Wtedy po zakończeniu ruchu ustawiają *val* na wartość swojego sąsiada.