Laboratorium 01 WDWK
Badanie prędkości funkcji quick sort

Grzegorz Wszoła

Marzec 2025

Spis treści

1 Wprowadzenie **2**

2 Kod programu **2**
2.1 Generacja danych 2
2.2 Obliczanie tików i sortowanie 2

3 Analiza wyników badania czasu **4**
3.1 Losowa tablica i losowy pivot 4
3.2 Tablica posrtowana i pivot ostatnim elementem 4
3.3 Tablica posortowana odwrotnie i pivot ostatnim elementem 6

4 Wnioski **7**

1 Wprowadzenie

Quick Sort to jeden z najpopularniejszych algorytmów sortowania z powodu stosunkowo małej złożoności i dużej optymalności $O(n \log n)$, jednak "worst case" $O(n^2)$, gdy tablica jest posortowana i wybierzemy nieodpowiedni pivot W swoim programie użyłem losowo wybieranego pivotu oraz losowych tablic co wpływa na wynik czasu sortowania, jednak w większości przypadków czas mieści się w $O(n \log n)$.

2 Kod programu

2.1 Generacja danych

Program wczytuje rozmiar tablicy od użytkownika z użciem funckji scanf z biblioteki libc

```
push $rozmiar
push $sc_string
call scanf
add $8, %esp
mov rozmiar, %eax
cmp overflow_max, %eax
jg overflow
```

Później program używa funkcji "generacja_danych" zdefiniowanej w pliku .c, która tworzy dynamicznie tabele oraz generuje losowe dane do posrtowania, zwraca wskaźnik na tabele.

```
int* generacja_danych(int rozmiar) {
int* tab = (int*)malloc(rozmiar * sizeof(int));
static int seeded = 0;
if (!seeded) {
    srand(time(NULL));
    seeded = 1;
}
for (int i = 0; i < rozmiar; i++) {
    tab[i] = rand() % rozmiar;
}
return tab;
}
```

Wywołanie funkcji w asembler

```
push rozmiar
call generacja_danych
add $4, %esp
mov %eax, tab_ptr
```

Następnie w asemblerze obliczany jest prawy pivot potrzebny później do funkcji quick sort

```
mov rozmiar, %ebx
sub $1, %ebx
mov %ebx, rozmiar
```

2.2 Obliczanie tików i sortowanie

Na początku zapisywany jest ilość tików procesora jako time_start za pomocą rozkazu rdtsc

```
rdtsc
mov %eax, time_start
```

Następnie sortujemy tablice i zapisujemy ilość tików jako time_end

```
push rozmiar
push $0
push tab_ptr
call quick_sort
add $12, %esp

rdtsc
mov %eax, time_end
```

Obliczamy ilosc tików procesora, który zajął quicksort i zamiana go na milisekundy, gdzie clock_per_sec to szybkość procesora w moim przyapdku 3.3GHz

```
mov time_end, %eax
sub time_start, %eax
```

Użyty algorytm quick sort z losowym pivotem

```
void quick_sort(int* tab, int left, int right) {
    if (left < right) {
        random_pivot(tab, left, right);
        int m = partition(tab, left, right);
        quick_sort(tab, left, m - 1);
        quick_sort(tab, m + 1, right);
    }
}

int partition(int* tab, int left, int right) {
    int pivot = tab[right];
    int i = left - 1;

    for (int j = left; j < right; j++) {
        if (tab[j] < pivot) {
            i++;
            swap(&tab[i], &tab[j]);
        }
    }
    swap(&tab[i + 1], &tab[right]);
    return i + 1;
}

void swap(int* left, int* right) {
    int temp = *left;
    *left = *right;
    *right = temp;
}

void random_pivot(int* tab, int left, int right) {
    int rand_idx = left + rand() % (right - left + 1);
    swap(&tab[rand_idx], &tab[right]);
}
```

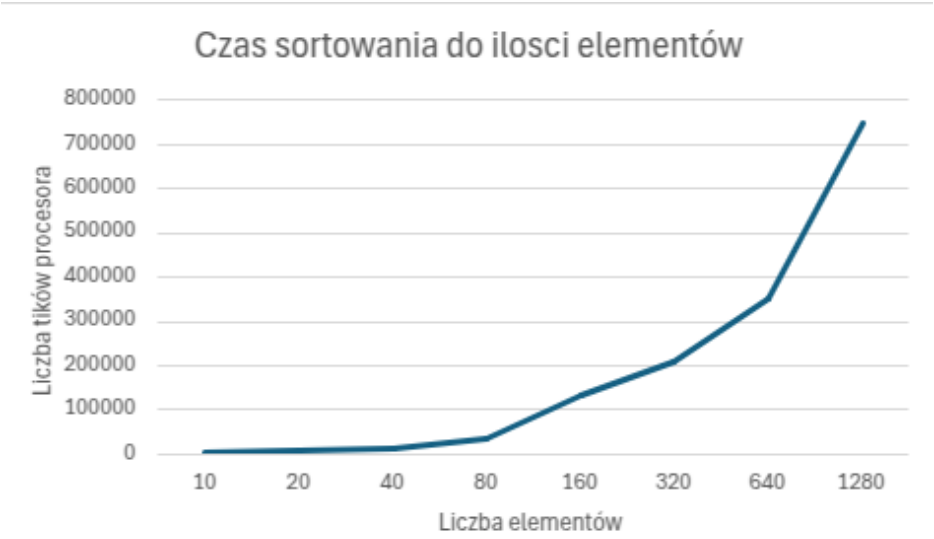
3 Analiza wyników badania czasu

3.1 Losowa tablica i losowy pivot

Zmierzony został czas dla kilku wielkości tablic

Tabela 1: Czas sortowania w stosunku do ilości elementów

ilość elementów tablicy	czas sortowania w tikach procesora
10	3084
20	7037
40	14585
80	36817
160	131124
320	208414
640	351494
1280	747494



Rysunek 1: Wykres reprezentujący czas sortowania w stosunku do ilości sortowanych elementów

Jak widać funkcja czasu jest zbliżona do funkcji $f(n) = n \log n$

3.2 Tablica posrtowana i pivot ostatnim elementem

Możemy teraz zmodyfikować lekko funkcje generacja_danych aby generowała posortowaną tablice. To razem z wyborem pivota jako ostatni element powinno stworzyć "worst case" funkcji quick sort i doprowadzić w do złożoności $O(n^2)$
Tworzymy nową metode która generuje dane i sortuje je

Sortuje funkcje rosnąco

```
int cmpfunc(const void* a, const void* b) {
    return (*(int*)a - *(int*)b);
}

int* generacja_danych_sortowane(int rozmiar) {
    int* tab = (int*)malloc(rozmiar * sizeof(int));
    if (tab == NULL) {
        return NULL;
    }

    static int seeded = 0;
    if (!seeded) {
        srand(time(NULL));
        seeded = 1;
    }

    for (int i = 0; i < rozmiar; i++) {
        tab[i] = rand() % rozmiar;
    }

    qsort(tab, rozmiar, sizeof(int), cmpfunc);
    return tab;
}
```

Musimy zmienić jeszcze kod quick sorta, aby nie losował pivota

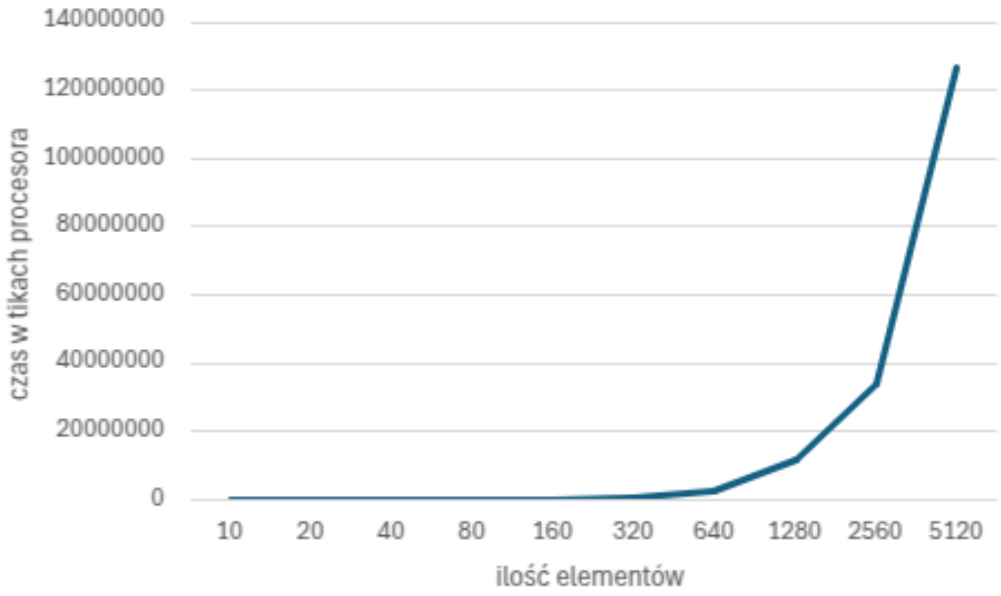
```
void quick_sort(int* tab, int left, int right) {
    if (left < right) {
        //random_pivot(tab, left, right)
        int m = partition(tab, left, right);
        quick_sort(tab, left, m - 1);
        quick_sort(tab, m + 1, right);
    }
}
```

Analiza danych

Tabela 2: Tablica czasu dla tablicy posortowanej i prawego pivota

ilość elementów w tabeli	ilość czasu w tikach procesora
10	3570
20	6951
40	17146
80	54611
160	256071
320	800479
640	2706695
1280	11574791
2560	33763807
5120	126458174

Jak widać na wykresie ilość czasu potrzebna do obliczenia większej ilośći danych rośnie dużo szybciej niż w pierwszym przypadku



Rysunek 2: Graf prezentujący ilość tików procesora do ilości danych

Jak widać funkcja zbliżona jest do funkcji $f(n) = n^2$

3.3 Tablica posortowana odwrotnie i pivot ostatnim elementem

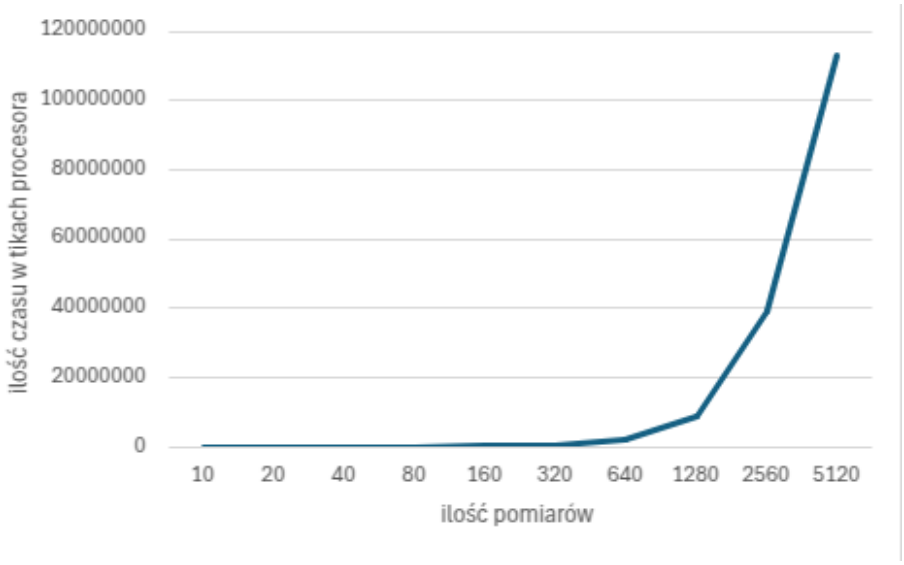
Jedyne co trzeba zmienić w przypadku to funkcje określającą kierunek sortowania tablicy z libc

```
int cmpfunc_mal(const void* a, const void* b) {
    return (*(int*)b - *(int*)a);
}
```

Analiza danych

Tabela 3: Tablica czasu dla tablicy odwrotnie posortowanej i prawego pivota

ilość elementów w tabeli	ilość czasu w tikach procesora
10	2255
20	6811
40	18632
80	121802
160	336911
320	585093
640	2104105
1280	8784587
2560	39100139
5120	113359842



Rysunek 3: Wykres zależności czasu w tikach procesora do ilości danych

W tym przypadku podobnie jak w poprzednim widać, że funkcja czasu rośnie dużo szybciej niż na 1 wykresie i jest zbliżona do funkcji $f(n) = n^2$

4 Wnioski

1. Funkcja sortująca quick sort jest algorytmem o złożoności obliczeniowej $O(n \log n)$, jeżeli wybierzemy odpowiednie parametry funkcji jak np. losowy pivot, lub lepiej pivot obliczony jako mediana trzech
2. Źle dobrane parametry funkcji oraz ewentualny "edge case" może sprawić, że funkcja będzie miała złożoność $O(n^2)$ wybierając pivot jako ostatni lub pierwszy element i gdy tabela jest posortowana. Sprawia to, że połówki tabeli sa nierówno podzielona doporwadzając do niezoptamilozowanej funkcji.