4. Problem Podziału Zbioru PARTITION

Metody Programowania

Marcin Klimek

4 kwietnia 2023

Politechnika Krakowska
Wydział Inżynierii
Elektrycznej i Komputerowej

Spis treści

1 Wstęp

W sprawozdaniu przedstawione zostaną dwa warianty dynamicznego algorytmu dla problemu decyzyjnego PARTITION[1]. Pierwszy przechowuje w macierzy wartości boolowskie o wielkości jednego bajta, co pozwala na szybkie obliczenia przy umiarkowanym zużyciu pamięci dla zbiorów o mniejszej ilości elementów oraz mniejszej sumie. Drugi przechowuje wartości boolowskie za pomocą pakowania bitów (dzięki czemu w jednym bajcie możliwe jest przechowanie 8 wartości boolowskich), co pozwala na działanie algorytmu dla większych liczb dzięki dużym oszczędnościom w pamięci, jednak kosztem wydajności.

Algorytmy oparte są na technice programowania dynamicznego, która polega na rozwiązaniu problemu poprzez rozwiązanie podproblemów. W przypadku problemu PARTITION, rozwiązanie problemu polega na sprawdzeniu istnienia podziału dla podzbiorów o rosnącej długości (zaczynając od pustego a kończąc na pełnym zbiorze), oraz dla każdej sumy mniejszej niż połowa sumy. Na podstawie wcześniej wykonanych obliczeń algorytm jest w stanie znajdywać kolejne rozwiązania, dzięki czemu jest on dynamiczny.

2 Implementacja

Działanie obu implementacji algorytmów zostanie pokazane na przykładzie języka `C`, jednak algorytm jest uniwersalny i może być zaimplementowany w dowolnym języku programowania. W celach zwięzłości przedstawione zostaną jedynie fragmenty kodu odpowiedzialne za działanie algorytmu, implementacja niektórych funkcji pomocniczych została pominięta. Całość kodu można zobaczyć na `https://github.com/klmkyo/partition`.

2.1 Algorytm bez pakowania bitowego

```c
void print_matrix(bool** matrix, int y_len, int x_len);
bool** init_matrix(int y_len, int x_len);
void free_matrix(bool** matrix, int y_len);
void save_matrix(bool** matrix, int y_len, int x_len, char* filename);
int partition(int* arr, int arr_len);

int main(int argc, char** argv);
```

Program implementujący algorytm bez pakowania bitowego składa się z 6 funkcji. Pierwsza z nich, `print_matrix`, służy do wyświetlania macierzy na ekranie. Druga, `init_matrix`, służy do alokacji pamięci dla macierzy o podanych wymiarach, oraz wypełnienia jej wartościami `false`. Trzecia, `free_matrix`, służy do zwolnienia pamięci zajmowanej przez macierz. Czwarta, `save_matrix`, służy do zapisania macierzy do pliku. Piąta, `partition`, jest funkcją odpowiedzialną za działanie algorytmu. Ostatnia, `main`, jest funkcją odpowiedzialną za pobranie danych wejściowych, wywołanie funkcji `partition` oraz wypisanie wyniku na ekranie. W ramach tego sprawozdania zostaną omówione funkcje `partition` oraz `main`.

2.1.1 Funkcja `main`

```c
int main(int argc, char *argv[]) {
    // sprawdzanie czy poadno argumenty do programu
    if (argc < 2) {
        printf("Usage: %s [--save filename] n1 n2 n3 ...\n", argv[0]);
        return 1;
```

```
}

// sprawdzanie czy podano opcję zapisu do pliku
char *filename = "";
if (strcmp(argv[1], "--save") == 0) {
    if (argc < 4) {
        printf("Usage: %s [--save filename] n1 n2 n3 ...\n", argv[0]);
        return 1;
    }
    // ustawienie statycznych zmiennych
    SAVE_FLAG = true;
    SAVE_FILENAME = argv[2];
    // przesunięcie argumentów dla dalszego odczytu
    argv += 2;
    argc -= 2;
}

// konwertowanie argumentów z char* na int
// tablica VLA
int arr[argc - 1];
char *endptr;
for (int i = 1; i < argc; i++) {
    errno = 0;
    int val = strtoi(argv[i], &endptr, 10);
    if (errno != 0 || *endptr != '\0') {
        printf("Error: argument %d is not an integer (errno=%d)\n", i, errno);
        return errno;
    }
    arr[i-1] = val;
}

int arr_len = sizeof(arr) / sizeof(arr[0]);

int set[arr_len];
int set1_len = 0;

int result = partition(arr, arr_len, set, &set1_len);

printf("\n");
if (result == 1) {
    printf("The set can be partitioned\n");

    printf("Set 1: {");
    for (int i = 0; i < set1_len - 1; i++) {
        printf("%d, ", set[i]);
    }
    printf("%d}\n", set[set1_len - 1]);

    printf("Set 2: {");
    for (int i = set1_len; i < arr_len - 1; i++) {
        printf("%d, ", set[i]);
```

```
        }
        printf("%d}\n", set[arr_len - 1]);
    } else if (result == 0) {
        printf("The set cannot be partitioned\n");
    } else if (result == -1) {
        printf("The set cannot be partitioned (sum is odd)\n");
    }

    return 0;
}
```

Zadaniem funkcji `main` jest pobranie danych wejściowych, wywołanie funkcji `partition` oraz wypisanie wyniku na ekranie. W pierwszej kolejności sprawdzane jest, czy podano argumenty niezbędne do działania programu. Następnie sprawdzane jest, czy użytkownik chce zapisać wynik działania programu do pliku. Jeśli tak, to nazwa pliku zostanie zapisana do statycznej zmiennej `SAVE_FILENAME`, a flaga `SAVE_FLAG` zostanie ustawiona na `true`. Te wartości zostaną potem odczytane przez funkcję `partition`. Jeśli znaleziono tę flagę, to wartości zostają przesunięte o 2 pozycje w tablicy `argv` oraz zmniejszone o 2 wartości w zmiennej `argc`, aby dalsza część programu mogła odczytać argumenty poprawnie. Następnie korzystając z funkcji `strtoi` oraz zmiennej `errno` z biblioteki `errno.h` konwertowane są argumenty z typu `char*` na `int`. W przypadku wystąpienia błędu (co jest sprawdzane poprzez odczytywanie wartości `errno`), program wypisuje komunikat o błędzie i kończy działanie. W innym przypadku, jeśli wszystko przebiegło pomyślnie, kalkulowana jest ilość podanych argumentów, po czym przygotowane są zmienne pomocnicze dla algorytmu. W tablicy `set` znajdywać się będą dwa zbiory: jeden zaczynający się od zera, a drugi od końca. Miejsce ich granicy oznaczone będzie w pozycji wyznaczonej przez `set1_len`, która mówi, jak długi jest pierwszy zbiór. Następnie wywoływana jest funkcja `partition`. Jej wyniki są następnie interpretowane i wypisywane dla użytkownika na ekranie.

2.1.2 Funkcja `partition`

Funkcja `partition` przyjmuje jako argumenty zbiór elementów `arr`, dla których chcemy sprawdzić, czy istnieje podział, `arr_len`, czyli ilość elementów w zbiorze, oraz zmienne, do których będą zapisywane dane podzielonych zbiorów. Zwracane są wartości 0, 1 lub -1, które oznaczają odpowiednio, że zbiór nie jest podzielny, że zbiór jest podzielny, lub że zbiór nie jest podzielny, ponieważ suma jego elementów jest nieparzysta.

```
int partition(int* arr, int arr_len, int* set, int* set1_len) {

    // obliczanie sumy elementów zbioru
    int sum = 0;
    for (int i = 0; i < arr_len; i++) {
        sum += arr[i];
    }

    // jeśli suma jest nieparzysta, zbiór nie może być podzielony
    if (sum % 2 != 0) {
        return -1;
    }

    // obliczanie wymiarów macierzy oraz jej inicjalizacja
    int y_len = (sum / 2) + 1;
    int x_len = arr_len + 1;
```

```
    bool** matrix = init_matrix(y_len, x_len);

    // wypełnienie pierwszego wiersza macierzy wartościami true
    for (int i = 0; i < x_len; i++) {
        matrix[0][i] = true;
    }

    // algorytm
    for (int y = 1; y < y_len; y++) {
        for (int x = 1; x < x_len; x++) {
            // zainicjalizowanie wartości na podstawie wartości
            // z poprzedniego wiersza
            matrix[y][x] = matrix[y][x - 1];

            // decydowanie możliwości podziału zbioru na podstawie
            // wcześniejszych obliczeń
            int new_element = arr[x - 1];
            if (!matrix[y][x] && new_element <= y) {
                matrix[y][x] = matrix[y - new_element][x - 1];
            }
        }
    }

    // wypisanie macierzy do wyjścia
    print_matrix(matrix, y_len, x_len);

    // sprawdzanie czy zbiór jest podzielny na podstawie
    // wartości w ostatnim wierszu
    int is_partitionable = matrix[y_len - 1][x_len - 1];

    // ustawienie wartości początkowych dla zmiennych pomocniczych
    *set1_len = 0;
    int set2_len = 0;

    // jeśli można podzielić, oraz jeśli użytkownik chce zobaczyć
    // podział, to zbiory zostają wypisane
    if (is_partitionable && set != NULL && set1_len != NULL) {
        int x = arr_len;
        int y = sum / 2;

        while (x > 0 && y >= 0) {
            x--;
            // jeśli aktualny element jest
            // częścią sumy, to należy do set2
            if (matrix[y][x]) {
                set[(arr_len - 1) - set2_len++] = arr[x];
            }

            // jeśli aktualny element nie jest częścią sumy,
            // to należy do set1
            else {
```

```
                y -= arr[x];
                set[(*set1_len)++] = arr[x];
            }
        }
    }

    // zapisanie macierzy do pliku, jeśli została podana flaga
    if (SAVE_FLAG) {
        save_matrix(matrix, y_len, x_len, SAVE_FILENAME);
    }

    // zwolnienie pamięci zajmowanej przez macierz
    free_matrix(matrix, y_len);

    return is_partitionable;
}
```

Na początku funkcji kalkulowana jest suma elementów w zbiorze. Jest ona potrzebna do utworzenia macierzy o prawidłowej wielkości. Następnie sprawdzane jest, czy suma jest parzysta. Jeśli nie, zbiór nie może być podzielony, więc funkcja zwraca wartość -1, która następnie jest interpretowana przez funkcję `main` jako błąd.

```
// obliczanie sumy elementów zbioru
int sum = 0;
for (int i = 0; i < arr_len; i++) {
    sum += arr[i];
}

// jeśli suma jest nieparzysta, zbiór nie może być podzielony
if (sum % 2 != 0) {
    return -1;
}
```

Następnie na podstawie sumy elementów w zbiorze oraz ilości elementów kalkulowane są wymiary macierzy. Wymiar `y` jest równy połowie sumy elementów w zbiorze, ponieważ suma jednej połowy zbioru musi być równa sumie drugiej połowy. Do `y` dodawane jest 1, aby brać pod uwagę sumę 0, która występuje dla pustego zbioru. Wymiar `x` reprezentuje fragment zbioru w zakresie $[0, x]$, np. dla zbioru $\{1, 3, 2\}$ będą to fragmenty $\{\}$, $\{1\}$, $\{1, 3\}$, $\{1, 3, 2\}$. Następnie na podstawie tych wymiarów tworzona jest macierz, która początkowo jest wypełniona wartościami `false`.

```
// obliczanie wymiarów macierzy oraz jej inicializacja
int y_len = (sum / 2) + 1;
int x_len = arr_len + 1;

bool** matrix = init_matrix(y_len, x_len);
```

Ponieważ suma pustego zbioru jest równa 0, to pierwszy wiersz macierzy jest wypełniany wartościami `true`, ponieważ z każdego wzoru można wybrać pusty zbiór.

```
for (int i = 0; i < x_len; i++) {
    matrix[0][i] = true;
}
```

Właściwy algorytm iteruje przez wszystkie wartości w macierzy, idąc od lewego górnego rogu do

prawego dolnego. Na samym początku wartość komórki jest inicjalizowana przez wartość po jej lewo, ponieważ zbiór liczb, którego podzbiór może się sumować do liczby `y`, to podzbiór zbioru z dodatkowym elementem również będzie w stanie się do tej liczby sumować.

Następnie, jeśli w aktualnie rozpatrywanym elemencie jest wartość fałszywa, sprawdzane jest, czy element, który ma zostać dodany do podzbioru, może pomóc w uzyskaniu sumy `y`. W tym celu sprawdzane jest, czy nowy element jest mniejszy lub równy szukanej sumie `y`. Jeśli nie, to dalsze sprawdzanie nie ma sensu, ponieważ dodanie elementu większego od docelowej sumy nie może pomóc w uzyskaniu tej sumy. Jeśli jednak element jest mniejszy lub równy, to sprawdzane jest, czy wartość komórki dla podzbioru bez

```
// algorytm
for (int y = 1; y < y_len; y++) {
    for (int x = 1; x < x_len; x++) {
        // zainicjalizowanie wartości na podstawie wartości
        // z poprzedniego wiersza
        matrix[y][x] = matrix[y][x - 1];

        // decydowanie możliwości podziału zbioru na podstawie
        // wcześniejszych obliczeń
        int new_element = arr[x - 1];
        if (!matrix[y][x] && new_element <= y) {
            matrix[y][x] = matrix[y - new_element][x - 1];
        }
    }
}
```

Następnie w algorytmie występuje logika zajmująca się zdobywaniem podzbiorów, na które został podzielony zbiór. Korzystając z macierzy wygenerowanej przez algorytm, możemy odczytać, która liczba będzie należeć dla danego podzbioru. W tym celu czytamy macierz, zaczynając od dolnego prawego rogu (od elementu decydującego podzielność zbioru), po czym po kolei czytana jest każda kolumna, oraz gdy natrafi się na wartość fałszywą, wiersz jest zmniejszany o wartość elementu w zbiorze, jaki reprezentuje. W celu oszczędności pamięci zbiór ten zapisywany jest do jednej tablicy, w której zbiór pierwszy jest zapisywany od końca tablicy, a zbiór drugi od początku (ich granica jest zapisana w zmiennej `set1_len`).

```
// sprawdzanie czy zbiór jest podzielny na podstawie
// wartości w ostatnim wierszu
int is_partitionable = matrix[y_len - 1][x_len - 1];

// ustawienie wartości początkowych dla zmiennych pomocniczych
*set1_len = 0;
int set2_len = 0;

// jeśli można podzielić, oraz jeśli użytkownik chce zobaczyć
// podział, to zbiory zostają wypisane
if (is_partitionable && set != NULL && set1_len != NULL) {
    int x = arr_len;
    int y = sum / 2;

    while (x > 0 && y >= 0) {
        x--;
        // jeśli aktualny element jest
```

```
        // częścią sumy, to należy do set2
        if (matrix[y][x]) {
            set[(arr_len - 1) - set2_len++] = arr[x];
        }

        // jeśli aktualny element nie jest częścią sumy,
        // to należy do set1
        else {
            y -= arr[x];
            set[(*set1_len)++] = arr[x];
        }
    }
}
```

Na samym końcu macierz jest wypisywana, oraz jeśli została podana flaga `--save`, macierz jest zapisywana do pliku. Na koniec, pamięć zajmowaną przez macierz jest zwalniana.

```
// wypisanie macierzy do wyjścia
print_matrix(matrix, y_len, x_len);

// zapisanie macierzy do pliku, jeśli została podana flaga
if (SAVE_FLAG) {
    save_matrix(matrix, y_len, x_len, SAVE_FILENAME);
}

// zwolnienie pamięci zajmowanej przez macierz
free_matrix(matrix, y_len);

return is_partitionable;
```

2.2 Algorytm z wykorzystaniem pakowania bitowego

W algorytmie z pakowaniem bitowym funkcja `main` pozostaje bez zmian. Reszta funkcji uległa zmianie, by działać z nowym sposobem przechowywania danych. Algorytm ten posługuje się logiką boolowską, by oszczędzić miejsce w pamięci poprzez przechowywanie ośmiu wartości logicznych w jednym bajcie. Ze względu na jednak na działanie współczesnych procesorów, które są w stanie adresować jedynie wartości o wielkości jednego bajta, odczytywanie tych wartości jest czasochłonne. W tym celu konieczne jest wykorzystanie operacji bitwise, które zauważalnie spowalniają działanie algorytmu, ale dzięki oczszczędnościom w pamięci możliwe jest działanie na o wiele większych zbiorach danych.

```
void print_matrix(unsigned char** matrix, int y_len, int x_len);
unsigned char** init_matrix(int y_len, int x_len);
void free_matrix(unsigned char** matrix, int y_len);
void save_matrix(bool** matrix, int y_len, int x_len, char* filename);
inline bool get_bit_from_byte(unsigned char byte, int i);
inline void set_bit_in_byte(unsigned char* byte_ptr, int i);
inline bool get_matrix_bit(unsigned char** matrix, int y, int x);
inline void set_matrix_bit(unsigned char** matrix, int y, int x);
int partition(int* arr, int arr_len);

int main(int argc, char** argv);
```

Funkcje pełnią identyczne zadanie jak w wersji bez pakowania bitowego, jednak różnią się w implemen-

tacji ze względu na operowanie na innej strukturze danych. Zamiast korzystania ze struktury `bool` wielkości 8 bitów, która przechowuje jedynie jedną wartość boolowską, wersja z pakowaniem korzysta z `unsigned char`, która również jest wielkości 8 bitów, jednak algorytm każdy z bitów interpretuje jako osobną wartość boolowską. Dlatego też dodano funkcje, które pomagają w odczytywaniu i zapisywaniu wartości w strukturze `unsigned char`.

2.2.1 Funkcje pomocnicze boolowskie

```
inline bool get_bit_from_byte(unsigned char byte, int i) {
    return (byte >> i) & 1;
}
inline void set_bit_in_byte(unsigned char* byte_ptr, int i) {
    *byte_ptr |= 1 << i;
}

inline bool get_matrix_bit(unsigned char** matrix, int y, int x) {
    return get_bit_from_byte(matrix[y][x / 8], x % 8);
}
inline void set_matrix_bit(unsigned char** matrix, int y, int x) {
    set_bit_in_byte(&matrix[y][x / 8], x % 8);
}
```

Funkcje `get_bit_from_byte` i `set_bit_in_byte` pozwalają na odczytywanie i zapisywanie wartości logicznych w strukturze `unsigned char`. Funkcje `get_matrix_bit` i `set_matrix_bit` znajdują odpowiedni bajt, w którym znajduje się żądana wartość logiczna, następnie wywołują odpowiednie funkcje pomocnicze.

Funkcje `get_bit_from_byte` i `set_bit_in_byte` posługują się przesunięciem bitowym, by dokonać operacji na żądanym bicie. Funkcja `set_bit_in_byte` tworzy bajt, w którym tylko bit, który chcemy ustawić, jest równy 1, a następnie nadpisuje wartość w strukturze `unsigned char` przez wykonanie operacji `OR` na obu wartościach. Funkcja `get_bit_from_byte` tworzy kopię bajta z macierzy, przesuwa interesującą wartość na koniec bajta, po czym wykonuje operację `AND` na wartościach `1` i bajcie z macierzy, dzięki czemu wiadomo czy interesujący nas bit jest równy 1, czy 0.

Funkcje `get_matrix_bit` i `set_matrix_bit` znajdują odpowiedniego bajta, w którym znajduje się interesująca nas wartość logiczna. Robią to poprzez dzielenie indeksu `x` przez `8`, co daje poszukiwany bajt, oraz wykorzystując resztę z dzielenia przez `8`, która jest równa pozycji bitu w wybranym bajcie.

2.2.2 Funkcja `partition`

```
int partition(int* arr, int arr_len) {
    unsigned int sum = 0;
    for (int i = 0; i < arr_len; i++) {
        sum += arr[i];
    }

    if (sum % 2 != 0) {
        return -1;
    }

    int y_len = (sum / 2) + 1;
    int x_len = arr_len + 1;
```

```
    unsigned char** matrix = init_matrix(y_len, x_len);

    for (int x = 0; x < x_len; x++) {
        set_matrix_bit(matrix, 0, x);
    }

    for (int y = 1; y < y_len; y++) {
        for (int x = 1; x < x_len; x++) {
            bool bit = get_matrix_bit(matrix, y, x - 1);
            if (bit) {
                set_matrix_bit(matrix, y, x);
            }

            int new_element = arr[x - 1];

            if (!bit && new_element <= y) {
                bit = get_matrix_bit(matrix, y - new_element, x - 1);
                if (bit) {
                    set_matrix_bit(matrix, y, x);
                }
            }
        }
    }

    int is_partitionable = get_matrix_bit(matrix, y_len - 1, x_len - 1);

    *set1_len = 0;
    int set2_len = 0;

    if (is_partitionable && set != NULL && set1_len != NULL) {
        int i = arr_len;
        int curr_sum = sum / 2;

        while (x > 0 && y >= 0) {
            x--;
            if (get_matrix_bit(matrix, y, x)) {
                set[(arr_len - 1) - set2_len++] = arr[x];
            }

            else {
                y -= arr[x];
                set[(*set1_len)++] = arr[x];
            }
        }
    }

    print_matrix(matrix, y_len, x_len);

    if (SAVE_FLAG) {
        save_matrix(matrix, y_len, x_len, SAVE_FILENAME);
    }
```

```
    free_matrix(matrix, y_len);

    return is_partitionable;
}
```

Funkcja `parition` odgrywa identyczną rolę jak jej odpowiednik w wersji bez pakowania bitowego. Jedyną różnicą jest to, że w tej wersji korzysta z funkcji pomocniczych `get_matrix_bit` i `set_matrix_bit`, aby operować na macierzy utworzonej z `unsigned char`.

3 Przykład działania

Obie wersje algorytmów dają identyczne wyniki. Poniżej przedstawiono przykład działania algorytmu dla zbioru liczb $\{1, 3, 2, 4\}$.

```
$ ./partition_bitpack.out 1 3 2 4

1 1 1 1 1
0 1 1 1 1
0 0 0 1 1
0 0 1 1 1
0 0 1 1 1
0 0 0 1 1

The set can be partitioned
Set 1: {2, 3}
Set 2: {1, 4}
```

Przykład działania algorytmu dla zbioru liczb $\{4, 5, 1, 2, 3, 3\}$.

```
$ ./partition.out 4 5 1 2 3 3
1 1 1 1 1 1 1
0 0 0 1 1 1 1
0 0 0 0 1 1 1
0 0 0 0 1 1 1
0 1 1 1 1 1 1
0 0 1 1 1 1 1
0 0 0 1 1 1 1
0 0 0 0 1 1 1
0 0 0 0 1 1 1
0 0 1 1 1 1 1

The set can be partitioned
Set 1: {5, 4}
Set 2: {1, 2, 3, 3}
```

Przykład działania algorytmu dla zbioru liczb $\{3, 1, 3, 3\}$. W tym przypadku algorytm zwraca wynik `0`, ponieważ nie istnieje podzbiór elementów o sumie równiej połowie sumy elementów w zbiorze.

```
$ ./partition.out 3 1 3 3
1 1 1 1 1
0 0 1 1 1
0 0 0 0 0
0 1 1 1 1
```

```
0 0 1 1 1
0 0 0 0 0

The set cannot be partitioned
```

Przykład działania algorytmu dla zbioru liczb $\{1, 1, 1, 2\}$. W tym przypadku algorytm zwraca wynik `-1`, ponieważ suma elementów w zbiorze jest nieparzysta.

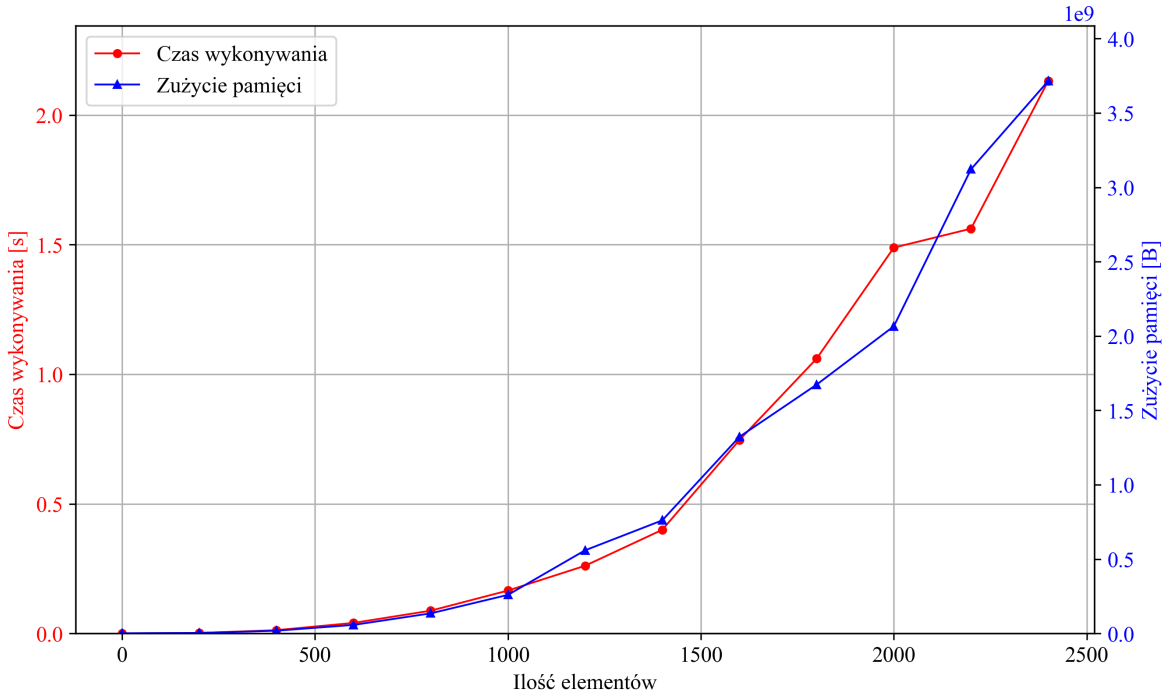
```
$ ./partition.out 1 1 1 2

The set cannot be partitioned (sum is odd)
```

4 Wydajność

Testy wydajności zostały przeprowadzone zarówno dla wersji z pakowaniem bitowym, jak i jego brakiem. Pakowanie bitowe zgodnie z oczekiwaniami powinno pozwolić na działanie na liczbach większych rozmiarów (wydajniejsze wykorzystanie pamięci) kosztem wydajności (w celu odczytywania wartości muszą być wykonywane operacje bitwise, które znacząco spowalniają algorytm). Program został skompilowany przy użyciu kompilatora `clang` z flagą optymalizacyjną `O3`, na komputerze z procesorem Apple M2 oraz na systemie macOS 13.3. Pomiary czasu wykonania zostały wykonane przy użyciu funkcji `clock()` z biblioteki `time.h`, natomiast pomiary szczytowego zużycia pamięci zostały dokonane poprzez odczytywanie wartości `peak memory footprint` z komendy `time -l` (flaga `-l` dostępna jest jedynie na systemach opartych na BSD).

Testy wersji bez pakowania bitowego zostały przeprowadzone na zakresie wartości od 200 do 2000 (z krokiem 200) ze względu na wysokie zużycie pamięci. Elementy w zbiorach są w zakresie od (1, n), więc suma elementów równa jest $\frac{n(n+1)}{2}$ (dlatego też jedynie ilości elementów będące wielokrotnością 4 są dozwolone; inaczej suma elementów nie będzie parzysta). Na potrzeby testów zostały wyłączone również instrukcje `printf`, które znacząco wpłynęłyby na wyniki.

Ilość elementów	Suma elementów	Czas średni	Czas min.	Czas max.	Zużycie pamięci
200	20100	1.6 ms ± 0.08	1.51 ms	1.7 ms	3.06 MiB
400	80200	12.52 ms ± 0.03	12.46 ms	12.55 ms	17.41 MiB
600	180300	40.56 ms ± 0.29	40.24 ms	41.01 ms	54.66 MiB
800	320400	87.75 ms ± 0.81	86.49 ms	88.69 ms	128.75 MiB
1000	500500	165.24 ms ± 0.98	164.31 ms	167.1 ms	247.19 MiB
1200	720600	260.96 ms ± 1	259.95 ms	262.62 ms	533.11 MiB
1400	980700	399.86 ms ± 0.47	399.11 ms	400.48 ms	725.33 MiB
1600	1280800	746.25 ms ± 0.76	745.34 ms	747.73 ms	1.23 GiB
1800	1620900	1.06 s ± 0	1.06 s	1.06 s	1.56 GiB
2000	2001000	1.49 s ± 0.01	1.47 s	1.5 s	1.92 GiB
2200	2421100	1.56 s ± 0.01	1.55 s	1.58 s	2.91 GiB
2400	2881200	2.14 s ± 0.04	2.06 s	2.17 s	3.46 GiB

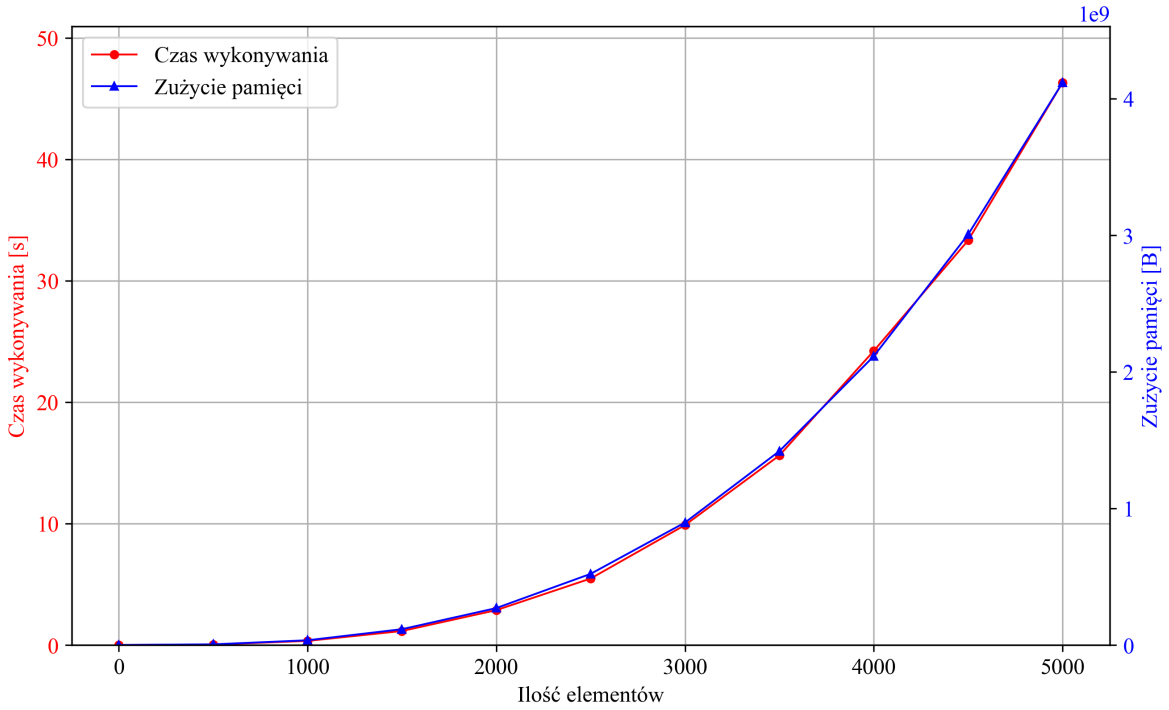


Rysunek 1: Wykres sprawności algorytmu bez pakowania bitowego

Jak można zauważyć, algorytm wykonuje się w bardzo szybkim czasie, jednak zużycie pamięci szybko staje się czynnikiem ograniczającym, przez co algorytm może mieć problemy z działaniem na większych zbiorach.

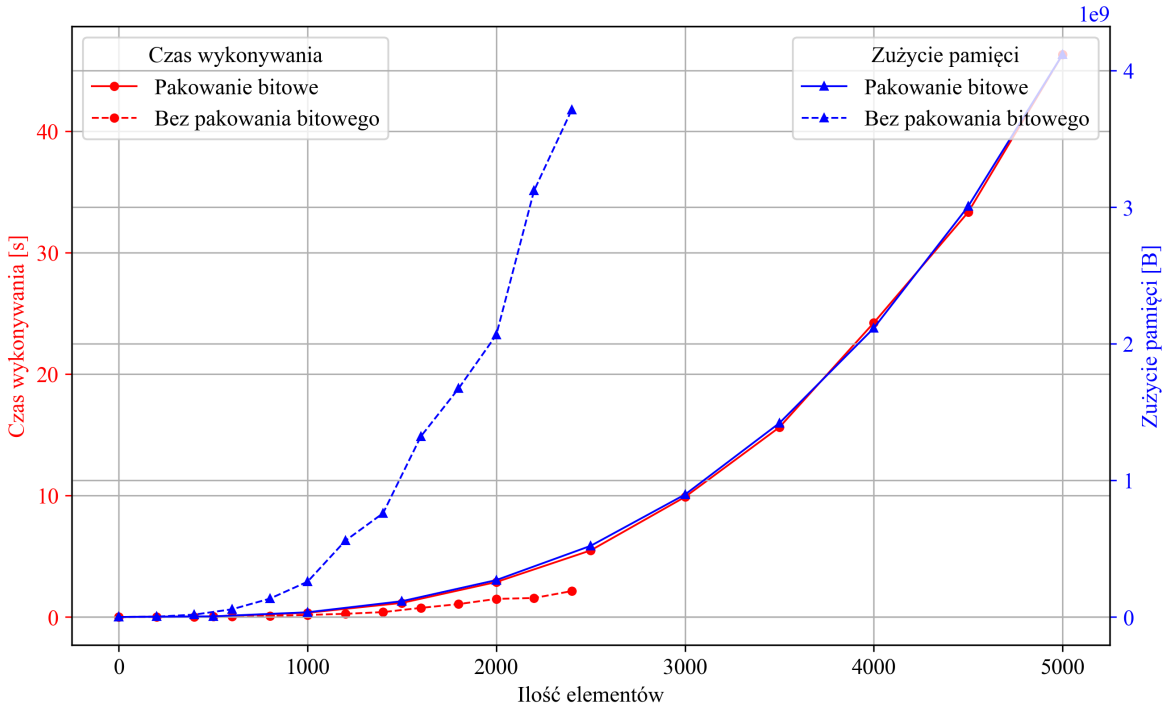
W przypadkach, gdy wykorzystanie algorytmu staje się niewykonywalne z powodu braku pamięci, warto skorzystać z wersji z pakowaniem bitowym, która poświęca trochę wydajności na rzecz bardziej efektywnego wykorzystania pamięci. Dzięki temu algorytm może działać na zbiorach o znacznie większych rozmiarach. Poniższe testy na wersji algorytmu z pakowaniem bitowym zostały przeprowadzone na zbiorach o rozmiarach od 500 do 5000 (z krokiem 500).

Ilość elementów	Suma elementów	Czas średni	Czas min.	Czas max.	Zużycie pamięci
500	125250	43.76 ms ± 0.53	43.32 ms	44.97 ms	5.28 MiB
1000	500500	343.61 ms ± 0.61	342.55 ms	344.66 ms	33.72 MiB
1500	1125750	1.16 s ± 0	1.16 s	1.16 s	111.04 MiB
2000	2001000	2.88 s ± 0.01	2.86 s	2.89 s	258.41 MiB
2500	3126250	5.48 s ± 0.01	5.47 s	5.49 s	497.52 MiB
3000	4501500	9.88 s ± 0.02	9.85 s	9.92 s	855.75 MiB
3500	6126750	15.62 s ± 0.04	15.56 s	15.68 s	1.32 GiB
4000	8002000	24.22 s ± 0.06	24.15 s	24.3 s	1.97 GiB
4500	10127250	33.32 s ± 0.02	33.29 s	33.37 s	2.8 GiB
5000	12502500	46.35 s ± 0.02	46.3 s	46.39 s	3.84 GiB



Rysunek 2: Wykres sprawności algorytmu z pakowaniem bitowym

Jak można zauważyć, czas wykonywania wersji z pakowaniem bitowym jest średnio kilkukrotnie razy większy w zakresie przeprowadzonych pomiarów, natomiast zużycie pamięci jest zgodnie z oczekiwaniami blisko 8 razy mniejsze, co pozwala na wykonywanie algorytmu na zbiorach o znacznie większych rozmiarach.



Rysunek 3: Porównanie sprawności algorytmów

5 Wnioski

Algorytm z pakowaniem bitowym jest znacznie bardziej wydajny pod względem zużycia pamięci, co pozwala na wykonywanie go na zbiorach o znacznie większych rozmiarach. Dla mniejszych zbiorów warto rozważyć wykorzystanie wersji bez pakowania bitowego, która jest zauważalnie szybsza, jednak jej złożoność pamięciowa jest znacznie większa.

Bibliografia

[1] S. Mertens, “The easiest hard problem: Number partitioning.” 2003. Available: https://arxiv.org/abs/cond-mat/0310317