SPRAWOZDANIE - LISTA 2

Zuzanna Pawlik, 282230

28.11.2024

FRAGMENTY KODÓW

QUICKSORT

Fragment kodu QUICKSORT wraz z funkcją pomocniczą PARTITION:

```
int PARTITION(int A[], int p, int k){
    int x = A[k];
    int i = p - 1;
    for(int j = p; j < k; j++){
        if(A[j] <= x){
            i++;
            swap(A[i],A[j]);
        }
    }
    swap(A[k],A[i+1]);
    return(i+1);
}


void QUICK_SORT(int A[],int p, int k){
    if(p<k){
        int s = PARTITION(A,p,k);
        QUICK_SORT(A,p,s-1);
        QUICK_SORT(A,s+1,k);
    }
}
```

Fragment kody zmodyfikowanego QUICKSORTA, który wykorzystuje 2 uporządkowane pivoty do podziału tablicy, tym samym dzieląc ją na trzy części (zamiast dwóch jak we wcześniejszej wersji). Pomocnicza funkcja PARTITION (zmodyfikowana) zwraca tym razem dwa punkty podziału tablicy:

```
void PARTITION_ZMOD(int A[], int p, int k, int& s1, int& s2){
    if(A[p]>A[k]){
        swap(A[p],A[k]);
```

```
    }

    int x1 = A[p];
    int x2 = A[k];
    int i1 = p + 1;
    int i2 = k - 1;

    for(int j = p+1; j <= i2; j++){
        if(A[j] < x1){
            swap(A[i1],A[j]);
            i1++;
        }else if(A[j] > x2){
            swap(A[j],A[i2]);
            i2--;
            j--;
        }
    }
    swap(A[p],A[i1-1]);
    swap(A[k],A[i2+1]);
    s1 = i1-1;
    s2 = i2+1;
}

void QUICK_SORT_ZMOD(int A[], int p, int k){
    if(p<k){
        int s1,s2;
        PARTITION_ZMOD(A,p,k,s1,s2);
        QUICK_SORT_ZMOD(A,p,s1-1);
        QUICK_SORT_ZMOD(A,s1+1,s2-1);
        QUICK_SORT_ZMOD(A,s2+1,k);
    }
}
```

RADIXSORT

Fragment kodu RADIXSORT z wykorzystaniem COUNTINGSORT. Dodatkowy parametr d w tym algorytmie oznacza system liczbowy, np 2 dla binarnego, czy 10 dla dziesiętnego. Ponieważ funkcja ta sortuje po kolejnych cyfrach (zaczynając od jedności) w tej wersji sortuje ona wyłącznie liczby dodatnie.

```
void COUNTING_SORT(int A[], int B[], int n, int k, int d){
    int C[d] = {0};

    for(int j = 0; j < n; j++){
        C[(A[j]/k)%d]++;
    }
```

```
    for(int i = 1; i < d; i++){
        C[i] += C[i-1];
    }

    for(int j = n-1; j >= 0; j--){
        B[C[(A[j]/k) % d] -1] = A[j];
        C[(A[j]/k) % d]--;
    }

    for (int i = 0; i < n; i++) {
        A[i] = B[i];
    }

}

void RADIX_SORT(int A[], int n, int d){
    int max = MAX(A,n);
    int B[n];
    for(int k = 1; max/k > 0; k *= d){
        COUNTING_SORT(A,B,n,k,d);
    }
}
```

Poniższa modyfikacja algorytmu RADIXSORT pozwala na sortowanie zarówno liczb ujemnych, jak i dodatnich. Wykorzystuje ona nadal COUNTINGSORT jak wyżej, ale w tej wersji najpierw odpowiednio modyfikujemy dane, aby uzyskać dobre wyniki. Funkcja ta wykorzystuje pomocnicze MIN, które zwraca najmniejszy element tablicy. Jeżeli najmniejszy element jest nieujemny to algorytm wykonuje się jak wyżej. W przeciwnym wypadku od każdego elementu tablicy odejmujemy wartość minimalną (odejmujemy bo $min < 0$), dzięki czemu wszystkie wartości są nieujemne. Następnie tablica jest sortowana jak wyżej. Po posortowaniu do każdego elementu tablicy dodajemy min, dzięki czemu przywracamy oryginalne wartości. Operacje te jednak wydłużają czas działania algorytmu (wykonują się dodatkowe przypisania).

```
void RADIX_SORT_ZMOD(int A[], int n, int d){
    int min = MIN(A,n);
    if(min>=0){
        int max = MAX(A,n);
        int B[n];
        for(int k = 1; max/k > 0; k*=d){
            COUNTING_SORT(A,B,n,k,d);
        }
    }else{
        for(int i = 0; i < n; i++){
            A[i] = A[i] - min;
```

```
        }
        int max = MAX(A,n);
        int B[n];
        for(int k = 1; max/k > 0; k*=d){
            COUNTING_SORT(A,B,n,k,d);
        }
        for(int i = 0; i < n; i++){
            A[i] = A[i] + min;
        }

    }

}
```

INSERTIONSORT NA LISTACH

Poniżej zdefiniowany jest algorytm INSERTIONSORT działający na listach wraz z funkcjami pomocniczymi LISTINSERT i LISTDELETE:

```
void LIST_INSERT(Node*& head, Node* x){
    x->next = head;
    x->prev = nullptr;
    if(head != nullptr){
        head->prev = x;
    }
    head = x;
}

void LIST_DELETE(Node*& head, Node* x){
    if(x->prev != nullptr){
        x->prev->next = x->next;
    }else{
        head = x->next;
    }
    if(x->next != nullptr){
        x->next->prev = x->prev;
    }
    delete x;
}
void INSERTION_SORT_LISTY(Node*& head){//I_S na listach
    if(!head || !head->next) return;
    Node* a = head->next;
    while(a != nullptr){
        Node* next = a->next;
        int key = a->key;
```

```
        LIST_DELETE(head,a);//usuwamy element ze starego miejsca

        Node* posortowana = head;//szukamy miejsca w posortowanej czesci
        while(posortowana != nullptr && posortowana->key < key){//gdzie wstawic
            posortowana = posortowana->next;
        }

        Node* Nowy = new Node(key);

        if(posortowana == nullptr){//wstawianie na koniec
            Node* tail = head;
            while (tail->next != nullptr) {
                tail = tail->next;
            }
            tail->next = Nowy;
            Nowy->prev = tail;
        }else if(posortowana == head){//wstawianie na poczatek
            LIST_INSERT(head, Nowy);
        }else{//w srodku
            Nowy->next = posortowana;
            Nowy->prev = posortowana->prev;
            if(posortowana->prev != nullptr){
                posortowana->prev->next = Nowy;
            }
            posortowana->prev = Nowy;
        }
        a = next;
    }
}
```

BUCKETSORT

Fragment algorytmu BUCKETSORT wykorzystujący powyższy INSERTION-SORT działający na listach (ta wersja działa tylko na liczbach z przedziału $(0, 1]$):

```
void BUCKET_SORT(double A[], int n){
    List* B = new List[n];
    for(int i = 0; i < n; i++){
        B[i].head = nullptr;
    }

    for(int i = 0; i < n; i++){
        int indeksB = min(static_cast<int>(n*A[i]),n-1);
        LIST_INSERT(B[indeksB].head, new Node(A[i]));
    }
```

```
    for(int j = 0; j < n; j++){
        INSERTION_SORT_LISTY(B[j].head);
    }
    //przypisanie z B[i] do tablicy A
    int k = 0;//indeksy z tablicy A
    for(int i = 0; i < n; i++){
        Node* a = B[i].head;
        while(a != nullptr){
            A[k] = a->key;
            a = a->next;
            k++;
        }
    }
}
```

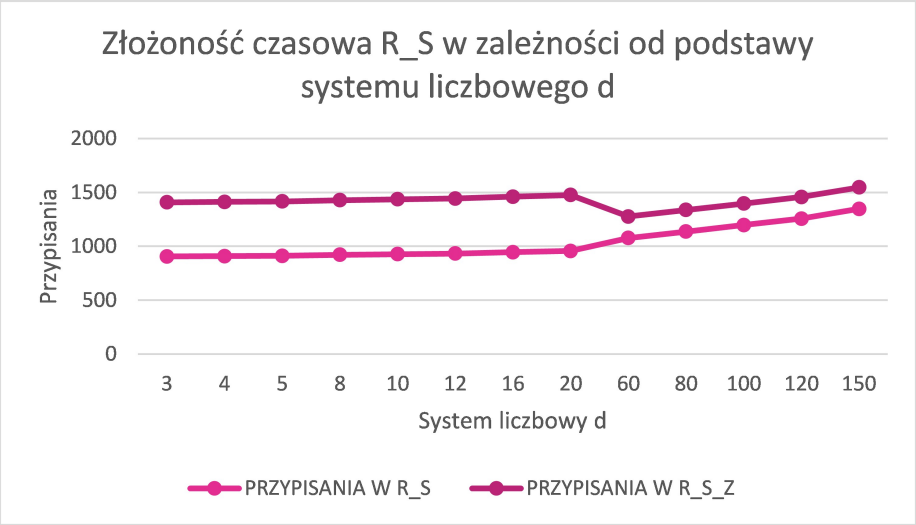
Modyfikacja BUCKETSORT pozwala na sortowanie dowolnych liczb, a nie tylko z przedziału $(0, 1]$. Najpierw wszystkie wartości przesuwamy na dodatnie poprzez odjęcie najmniejszego elementy, następnie dzielimy przez (przesunięty) element największy. Dzięki temu wszystkie wartości możemy sortować jak w powyższym algorytmie. Po posortowaniu zmiany są odwracane. Podobnie jak w przypadku modyfikacji RADIXSORT takie operacje mogą wydłużyć czas działania algorytmu.

```
void BUCKET_SORT_ZMOD(double A[], int n){
    double max = MAX(A,n);
    double min = MIN(A,n);
    if(max == min){//wszystkie wartosci takie same - unikanie dzielenia przez 0
        return;
    }else if(0<min && max<=1){//czy wszystkie wartości są z (0,1]
        BUCKET_SORT(A,n);
    }else{
        for(int i = 0; i < n; i++){
            //najpierw chcemy miec same dodatnie
            //potem dzieląc przez najwiekszą dostaniemy wartości z (0,1]
            A[i] = (A[i]-min)/(max-min);
        }

        BUCKET_SORT(A,n);

        for(int i = 0; i < n; i++){//odwracamy zmiany
            A[i] = (A[i]*(max-min))+min;
        }

    }
}
```

TABELE I WYKRESY

PORÓWNANIE RADIXSORT Z MODYFIKACJĄ

d	PRZYPISANIA W R_S	PRZYPISANIA W R_S_Z
2	903	1404
3	906	1408
4	909	1412
5	912	1416
8	921	1428
10	927	1436
12	933	1444
16	945	1460
20	957	1476
60	1077	1277
80	1137	1337
100	1197	1397
120	1257	1457
150	1347	1547

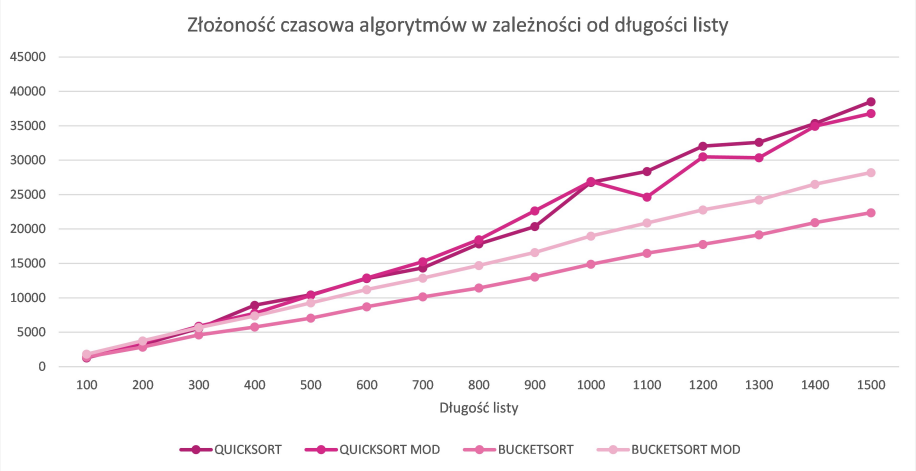


WNIOSKI

Jak widać na wykresie RADIXSORT działa dość stabilnie względem podstawy systemu liczbowego. Modyfikacja wykonuje się dłużej ze względu na występowanie w listach, na których była testowana, liczb ujemnych. Dodatkowe przypisania pozwalają dostosować dane do sortowania.

PORÓWNANIE QS I BS W ZALEŻNOŚCI OD DŁUGOŚCI LISTY

len	QS przyp	QS por	QUICKSORT	QSZ przyp	QSZ por	QUICKSORT MOD	BS przyp	BS por	BUCKETSORT	BSZ przyp	BSZ por	BUCKETSORT MOD
100	710	574	1284	744	691	1435	1076	321	1397	1294	527	1821
200	1787	1448	3235	1706	1822	3528	2192	657	2849	2656	1093	3749
300	2923	2650	5573	2762	3116	5878	3518	1092	4610	4027	1661	5688
400	4765	4164	8929	3686	4073	7759	4434	1336	5770	5249	2138	7387
500	5805	4628	10433	4928	5433	10361	5431	1623	7054	6584	2680	9264
600	7159	5647	12806	6228	6652	12880	6664	2048	8712	7941	3254	11195
700	7743	6608	14351	7190	8068	15258	7766	2379	10145	9142	3733	12875
800	9824	8012	17836	8484	9969	18453	8785	2647	11432	10427	4273	14700
900	10999	9365	20364	10474	12171	22645	9989	3049	13038	11765	4825	16590
1000	14306	12477	26783	12472	14428	26900	11384	3504	14888	13441	5543	18984
1100	15478	12894	28372	10560	14093	24653	12617	3864	16481	14813	6073	20886
1200	17638	14401	32039	12848	17647	30495	13586	4194	17780	16136	6655	22791
1300	18227	14379	32606	13812	16551	30363	14664	4490	19154	17190	7050	24240
1400	18796	16549	35345	15450	19490	34940	16009	4933	20942	18817	7695	26512
1500	21221	17277	38498	17260	19529	36789	17123	5251	22374	20014	8199	28213



WNIOSKI

Z wykresu wynika, że BUCKETSORT działa zarówno szybciej jak i stabilniej niż QUICKSORT. Modyfikacja QUICKSORT nieznacznie skraca czas działania algorytmu, zwłaszcza dla dłuższych list. Wykonuje ona mniej przypisań, ale kosztem ilości porównań. Modyfikacja BUCKETSORT nadal jest szybsza od QUICKSORT, a pozwala na sortowanie dowolnych wartości, w przeciwieństwie do pierwszej wersji, która sortuje wyłącznie wartści dodatnie.