Sprawozdanie z listy 1 (AiSD)

Szymon Wojtasik, nr albumu: 287306

Streszczenie

W tym sprawozdaniu omówimy 4 algortymy sortujące. Każdy z nich jest inny, a szczególnie czasy ich wykonywania często się bardzo różnią. Z tych 4 algorytmów wyjmiemy sobie najciekawsze fragmenty kodu.

1 QuickSort

Algorytm bazuje na pivotach, czyli elementach odpowiadających za porównanie czy dane elementy sa mniejsze(równe) czy większe od pivotów. Najciekawszy dla mnie fragment kodu to ten poniższy:

```
for (int p = m; p <= n;) {
        if (A[p] < y) {
            swap(A[p], A[m]);
            m++;
            p++;
        }
        else if (A[p] > k) {
            swap(A[p], A[n]);
            n--;
            continue;
        }
        else {
            p++;
        }
    }
    swap(A[poczatek_2], A[m - 1]);
    swap(A[koniec_2], A[n + 1]);
    return {m - 1, n + 1};
}
```

W 2 linijce porównujemy element z mniejszym pivotem(na pierwszym miejscu) i jeżeli element jest mniejszy od pivota to zamieniamy go z A[m], gdzie m jest skorelowane z tym ile jest elementów mniejszych od pivota mniejszego. Zwiększami p, ale póki co może być to nieoczywiste. Na razie zastanówmy się nad dalszym działaniem algorytmu.
Jeżeli pierwszy warunek nie jest prawdą to sprawdzamy czy element jest większy od większego pivota. W przypadku prawdy zmniejszami indeks n, gdzie n jest skorelowanym z tym ile elementów jest mniejszych od większego pivota.
W innym(jedynym) przypadku element jest pośrodku pivotów.

Pierwsza zagwostka:
Czemu zwiększamy tylko p na pierwszym i ostatnim warunku(gdzie p odzwierciedla ile elementów jest mniejszych od większego pivota)? Odpowiedź jest nieoczywista. Rozpatrzmy jakie mamy możliwości.

$\Longrightarrow$ Element jest mniejszy od większego pivota i większy od mniejszego. W takim przypadku zwiększamy p.
$\Longrightarrow$ Element jest mniejszy od mniejszego pivota. Teraz również zwiększamy p, ponieważ jest to oczywiście prawdą, że ten element jest większy od większego pivota
$\Longrightarrow$ Jeżeli element jest większy od większego pivota to nie zwiększamy p, co myślę że jest logiczne.
Czemu w takim razie **continue**? Odpowiedź jest taka, że przecież po zamianie elementu, może się okazać że dalej będzie on większy od większego pivota, a przecież nie o to nam chodzi.

2 RadixSort

Ideą radix sort jest sortowanie poprzez sprawdzanie kolejnych cyfr liczby(jedności, dziesiątki, setki itd.).

```
1 for (int j = 1; j < d; j++) {
2     C[j] += C[j-1];
3 }
4 for (int i = n - 1; i >= 0; i--) {
5     B[C[f(A[i])]-1] = A[i];
6     C[f(A[i])]--;
7 }
```

Po pętli wartość C[j] pokazuje na cyfrę jednościw tablicy B, którą ma zająć element z cyfrą jedności równą j.
Iterujemy od tyłu, aby zachować stabilność. Pętla przechodzi przez tablicę A od końca. Wstawiamy element A[i] do tablicy B na cyfrę o jeden mniejszą(jak były jedności to teraz są dziesiątki) niż aktualna wartość w C, czyli B[C[f(A[i])]-1].
Algorytm jest stabilny, ponieważ po wstawieniu licznik C[f(A[i])] jest zmniejszany o 1. Dzieję się tak, ponieważ zmniejszenie licznika daje to, że następny element w A, który będzie miał tę samą cyfrę trafi na cyfrę o jeden mniejszą w B. Daje to dużo, ponieważ w ten sposób zachowujemy pierwotną kolejność elementów o tych samych cyfrach(na przykład jedności). Bez tego RadixSort nie działałby tak dobrze

3 List Insertion Sort

Sortowanie przez wstawianie na liście to zupełnie co innego niż na tablicy. Tutaj nie przesuwamy elementów, tylko musimy operować na wskaźnikach. Nie mamy też swobodnego dostępu do elementów, więc nie możemy skakać po indeksach. Poniżej fragment, który sprawił mi najwięcej trudności, czyli pętla główna algorytmu:

```
1  while (unsorted_start != nullptr) {
2      Node* kolejny = unsorted_start->next;
3
4      // Wstawienie elementu na poczatek
5      if (unsorted_start->val <= sorted_start->val) {
6          unsorted_start->next = sorted_start;
7          sorted_start = unsorted_start;
8      }
9      // Wstawienie w srodek
10     else {
11         Node* mniejszy = sorted_start;
12         while (mniejszy->next != nullptr &&
13                !(unsorted_start->val <= mniejszy->next->val)) {
14             mniejszy = mniejszy->next;
15         }
16         if (mniejszy->next == nullptr) {
17             mniejszy->next = unsorted_start;
18             unsorted_start->next = nullptr;
19         }
20         else {
21             unsorted_start->next = mniejszy->next;
22             mniejszy->next = unsorted_start;
23         }
24     }
25     unsorted_start = kolejny;
26 }
```

Kluczowe w 2 linijce jest zapamiętanie wskaźnika na kolejny element. Gdybyśmy tego nie zrobili przed przepięciem elementu unsorted start, urwałby nam się dostęp do reszty nieposortowanej listy.

Dalej sprawdzamy warunek w linii 5. Jeżeli element jest mniejszy od głowy listy posortowanej, to po prostu wpinamy go na początek. To jest ten prostszy przypadek, bo zmieniamy tylko start listy.

Schody zaczynają się w bloku else. Musimy szukać miejsca wstawienia pętlą while. Co ciekawe, wskaźnik mniejszy zatrzymuje się na elemencie przed miejscem wstawienia, żebyśmy mogli łatwo przepiąć wskaźniki next. W pętli sprawdzamy, czy wartość następnika jest wciąż mniejsza od wstawianego elementu. Jeśli pętla dojdzie do końca (linia 16), to dopinamy element na sam ogon listy posortowanej. W przeciwnym wypadku (linia 20) wpinamy go pomiędzy dwa węzły.

4 Bucket Sort

W tym algorytmie najciekawsze jest to, że moje kubełki to tak naprawdę tablica wskaźników na listy. Zamiast sortować wszystko naraz, wrzucamy liczby do odpowiednich worków. Poniżej funkcja główna algorytmu:

```
1  void BUCKET_SORT(float tab[], int n) {
2
3      List* buckets[n];
4      for (int i = 0; i < n; i++) {
5          buckets[i] = new List();
6      }
7
8      for (int i = 0; i < n; i++) {
9          int do_ktorego_bucketa = ...; // wyliczanie indeksu
10         Node* wagonik_dla_bucketa = new Node(tab[i]);
11         INSERT(buckets[do_ktorego_bucketa], wagonik_dla_bucketa);
12     }
13
14
15 }
```

W pierwszej pętli po prostu tworzymy puste listy dla każdego kubełka. Ciekawiej dzieje się w drugiej pętli (od linii 8). Bierzemy element z tablicy i tworzymy dla niego Node, który nazywam tutaj wagonikiem.

Najważniejsza jest tutaj funkcja INSERT w linijce 11. Użyłem tutaj funkcji z poprzedniego zadania, czyli Insertion Sort dla listy. Dzięki temu, wkładając element do kubełka, od razu dbamy o to, żeby w tym kubełku był porządek.

Na koniec algorytmu wystarczy tylko przejść po tablicy buckets i posklejać te listy w jedną długą, co jest już bardzo szybkie, bo elementy wewnątrz kubełków są już posortowane.

5 Licznik operacji w QuickSort

Zastanawiałem się, co bardziej obciąża program – czy porównywanie liczb, czy ich zamiana miejscami. Żeby to sprawdzić, zmodyfikowałem QuickSorta dodając liczniki, które przekazuję przez referencję. Poniżej funkcja zamiany z licznikiem:

```
void swap_with_count(int &a, int &b, long long &przypisania) {
    int temp = a;
    a = b;
    b = temp;
    przypisania += 3;
}
```

Wyniki są dosyć ciekawe. Standardowa funkcja swap wygląda niewinnie, ale pod spodem generuje aż 3 przypisania, bo musimy użyć zmiennej tymczasowej temp.

Zmodyfikowałem też funkcję partycjonującą:

```
// Fragment Partition:
if (A[p] < y) {
    swap_with_count(A[p], A[m], przypisania);
    m++;
    // ...
}
```

Dzięki temu widzę, że przy wersji z dwoma pivotami wykonujemy więcej porównań (bo trzeba sprawdzić dwa pivoty), ale za to drzewo rekurencji jest płytsze. Liczba przypisań zależy mocno od stopnia nieuporządkowania tablicy wejściowej.

6 Wnioski końcowe

Podobnie jak przy liście 3, tutaj też nasuwają mi się pewne przemyślenia odnośnie zaimplementowanych algorytmów:

1. **Wskaźniki to nie tablice** – Insertion Sort na liście uświadomił mi, jak cenny jest swobodny dostęp do indeksów. Na liście musimy wędrować wskaźnikiem, co komplikuje kod, mimo że idea algorytmu jest ta sama. Musimy też bardzo uważać, żeby nie zgubić wskaźnika na resztę listy przy przepinaniu.

2. **Rozkład danych** – Bucket Sort działa świetnie, ale tylko pod warunkiem, że dane wpadają do kubełków w miarę równo. Gdyby wszystkie liczby wpadły do jednego, to i tak skończylibyśmy ze zwykłym Insertion Sortem i złożonością kwadratową. Dlatego funkcja dobierająca kubełek jest kluczowa.

3. **Ukryte koszty** – Liczenie operacji w QuickSorcie pokazało mi, że sama złożoność obliczeniowa to nie wszystko. Czasami mniejsza liczba zamian jest ważniejsza niż mniejsza liczba porównań, bo dostęp do pamięci bywa kosztowny. Funkcja swap to tak naprawdę trzy operacje, o czym łatwo zapomnieć używając gotowych bibliotek.

4. **Dual Pivot** – Wersja QuickSorta z dwoma pivotami wydaje się bardziej skomplikowana w implementacji (trzy przedziały zamiast dwóch), ale pozwala szybciej dzielić problem na mniejsze kawałki. Widać to przy bardzo dużych danych, gdzie zyskujemy na głębokości rekurencji.