Quicksort

Algorytm, implementacja, wydajność

Piotr Sobieraj

Wydział Matematyki
i Informatyki
Uniwersytetu Łódzkiego

seminarium licencjackie
prowadzący dr Dariusz Doliwa
październik 2024

Zawartość

UNIWERSYTET ŁÓDZKI

Algorytm

Opis algorytmu

- dziel i zwyciężaj

Algorytm

Opis algorytmu

- dziel i zwyciężaj
- dzieli tablice na dwie podtablice i sortuje je niezależnie

Algorytm

Opis algorytmu

- dziel i zwyciężaj
- dzieli tablice na dwie podtablice i sortuje je niezależnie
- dwa rekurencyjne wywołania mają miejsce po operacji na całej tablicy

Algorytm

Opis algorytmu

- dziel i zwyciężaj
- dzieli tablice na dwie podtablice i sortuje je niezależnie
- dwa rekurencyjne wywołania mają miejsce po operacji na całej tablicy
- miejsce podziału zależy od zawartości tablicy

UNIWERSYTET ŁÓDZKI

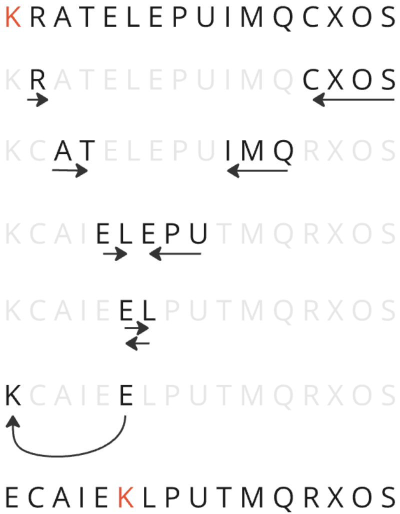
Sposób podziału tablicy

- Element a[j] znajduje się na ostatecznym miejscu w tablicy (dla pewnego j)

Sposób podziału tablicy

- Element a[j] znajduje się na ostatecznym miejscu w tablicy (dla pewnego j)
- Żaden element w przedziale od a[lo] do a[j-1] nie jest większy niż a[j]
- Żaden element w przedziale od a[j+1] do a[hi] nie jest mniejszy niż a[j]

Przebieg działania algorytmu



Wybór elementu osiowego

- pierwszy element

Wybór elementu osiowego

- pierwszy element
- ostatni

Wybór elementu osiowego

- pierwszy element
- ostatni
- losowy

Wybór elementu osiowego

- pierwszy element
- ostatni
- losowy
- mediana z trzech

Wybór elementu osiowego

- pierwszy element
- ostatni
- losowy
- mediana z trzech
- mediana z całej tablicy
- ...

Implementacja

`https://github.com/piotr-sobieraj/quicksort`

UNIWERSYTET
ŁÓDZKI

Dowód poprawności działania

Dowód indukcyjny

Dla $n = 1$ tablica jest już posortowana.

Załóżmy, że quicksort sortuje tablicę o długości n, $n = 1, 2, \ldots$. Weźmy tablicę o długości $n + 1$ i wybierzmy z niej element osiowy t.

Partycjonujemy tablicę na dwie części L i P w taki sposób, że $L < t < P$.

Każdą z tablic L i P z założenia indukcyjnego możemy posortować quicksortem.

Element osiowy znajduje się pomiędzy nimi, zatem cała tablica (L, t, P) posiadająca $n + 1$ elementów jest posortowana.

UNIWERSYTET ŁÓDZKI

Kod źródłowy - partycjonowanie

```
template <typename T>
int partition(std::vector<T>& v, int lo) {
    int hi = v.size() - 1;
    T pivot = v[lo];  // Wybieramy element osiowy (pierwszy element)
    int i = lo, j = hi + 1;  // Lewy i prawy wskaźnik

    while (true) {
        // Znajdź element większy od pivota z lewej strony
        while (less(v[++i], pivot)) if (i == hi) break;

        // Znajdź element mniejszy od pivota z prawej strony
        while (less(pivot, v[--j])) if (j == lo) break;

        // Jeśli wskaźniki się spotkają, przerwij
        if (i >= j) break;

        // Zamień elementy, które są nie na swoim miejscu
        std::swap(v[i], v[j]);
    }
    // Zamień element osiowy na właściwe miejsce
    std::swap(v[lo], v[j]);

    return j;  // Zwróć indeks pivota
}
```

UNIWERSYTET ŁÓDZKI

Kod źródłowy - sortowanie

```
template <typename T>
void sort(std::vector<T>& v, int lo, int hi)
{
    if (hi <= lo) return;
    int j = partition(v, lo); // Podział
    sort(v, lo, j - 1); // Sortowanie lewej strony a[lo .. j-1].
    sort(v, j + 1, hi); // Sortowanie prawej strony a[j+1 .. hi].
}


template <typename T>
void sort(std::vector<T>& v)
{
    sort(v, 0, v.size() - 1);
}
```

Kod źródłowy - funkcje wprowadzające porządek

```
bool less(char a, char b){
    return a < b;
}

bool less(std::string a, std::string b){
    return a.length() < b.length();
}
```

Kod źródłowy - funkcja *main*

```
#include <iostream>
#include <vector>
#include <string>

int main()
{
    std::vector<char> v_char = {'G', 'B', 'D', 'L', 'W', 'I', 'G', 'S', 'Q',
    ↪  'L', 'X', 'C'};
    print(v_char);
    sort(v_char);
    print(v_char);

    std::cout<<std::endl;

    std::vector<std::string> v_str = {"SEMINARIUM", "JEDEN", "GRUSZKA",
    ↪  "POLSKA", "INFORMATYKA", "UL", "C++"};
    print(v_str);
    sort(v_str);
    print(v_str);

    return 0;
}
```

Kod źródłowy - wynik działania programu

```
K    B    D    L    W    I    G    S    Q    L    X    C
B    C    D    G    I    K    L    L    Q    S    W    X

SEMINARIUM   JEDEN   GRUSZKA   POLSKA   INFORMATYKA   UL   C++
UL   C++   JEDEN   POLSKA   GRUSZKA   SEMINARIUM   INFORMATYKA
```

Cechy związane z wydajnością

- w podstawowej wersji nie jest stabilny

Cechy związane z wydajnością

- w podstawowej wersji nie jest stabilny
- złożoność pamięciowa dla niezoptymalizowanej wersji
  - $O(n)$ - najgorszy przypadek
  - $O(\log(n))$ - najlepszy

UNIWERSYTET ŁÓDZKI

Cechy związane z wydajnością - najlepszy przypadek

Porównania

Na każdym poziomie podziału tablica jest dzielona dokładnie na dwie połowy. Wtedy drzewo wywołań rekurencyjnych jest zbalansowane i jego wysokość wynosi $\log(n)$.
Ponadto na każdym poziomie drzewa dokonujemy *n - 1* porównań.
Stąd w najlepszym wypadku mamy

$$O(n-1) \cdot O(\log(n)) = O(n\log(n))$$

porównań.

UNIWERSYTET ŁÓDZKI

Cechy związane z wydajnością - najlepszy przypadek

Przestawienia

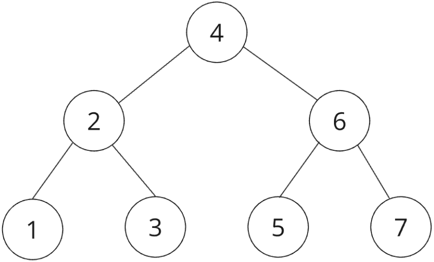
Podobnie jak dla porównań, mamy $O(n) \cdot O(\log(n)) = O(n\log(n))$ przestawień.
Ostatecznie złożoność obliczeniowa algorytmu quicksort w najlepszym wypadku wynosi

$$2 \cdot O(n) \cdot O(\log(n)) = O(n\log(n)).$$

UNIWERSYTET ŁÓDZKI

Cechy związane z wydajnością - najlepszy przypadek

Obserwacja: Wygenerowanie najlepszego przypadku sprowadza się do umieszczenia danego ciągu w idealnie zbalansowanym drzewie BST. Na przykład, dla (1, 2, 3, 4, 5, 6, 7) będzie to



czyli ciągiem jest (4, 2, 6, 1, 3, 5, 7).

UNIWERSYTET ŁÓDZKI

Cechy związane z wydajnością - najgorszy przypadek

Porównania

Na każdym poziomie podziału tablica jest dzielona na części (∅, t, P). Wtedy wysokość drzewa wywołań rekurencyjnych wynosi n. Ponadto na każdym poziomie drzewa dokonujemy *n - 1* porównań. Stąd ilość porównań w najgorszym wypadku wynosi

$$O(n-1) \cdot O(n) = O(n^2).$$

UNIWERSYTET ŁÓDZKI

Cechy związane z wydajnością - najgorszy przypadek

Przestawienia

Na pierwszym poziomie wykonujemy $n-1$ przestawień, na drugim $n-2$ itd. W rezultacie mamy

$$(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{n}=O(n^2).$$

Ostatecznie (porównania i przestawienia) złożoność obliczeniowa w najgorszym przypadku wynosi

$$2\cdot O(n^2)=O(n^2).$$

UNIWERSYTET ŁÓDZKI

Cechy związane z wydajnością - najgorszy przypadek

Obserwacja: Najgorszym przypadkiem jest tablica już posortowana rosnąco (gdy sortujemy rosnąco).

Podsumowanie - algorytm

- dziel i zwyciężaj
- średnia złożoność czasowa: $O(n \log n)$
- w najgorszym przypadku $O(n^2)$.
- działa w miejscu - nie wymaga dodatkowej pamięci, poza stosami rekurencyjnymi
- nie jest stabilny.
- optymalny dla losowych danych i średnich rozmiarów tablic
- w najgorszym przypadku działa nieoptymalnie dla już posortowanych danych

UNIWERSYTET ŁÓDZKI

Podsumowanie - złożoność obliczeniowa

- może być wybierany jako pierwszy, ostatni, losowy lub mediana
- najlepsze wyniki daje wybór mediany lub losowy wybór elementu
- wybór skrajnych wartości w posortowanej tablicy prowadzi do najgorszego przypadku

Podsumowanie - złożoność pamięciowa

- w najlepszym przypadku: $O(\log n)$ (optymalne zużycie pamięci
- w najgorszym przypadku: $O(n)$

Literatura

Robert Sedgewick, Kevin Wayne. *Algorytmy*. Wyd. IV. Helion, 2017. ISBN: 978-83-283-3711-4.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Wprowadzenie do algorytmów*. Wydawnictwo Naukowe PWN SA, Warszawa, 2024. ISBN: 978-83-01-229600-3.

UNIWERSYTET ŁÓDZKI

Autor

Piotr Sobieraj ul0249089@edu.uni.lodz.pl

`https://github.com/piotr-sobieraj/quicksort`