Aufgabe zum Sortieren

Erik Imgrund, TINF19B1

29. Mai 2020

1 Grundlegendes

1.1 Aufgabenstellung

Die verwendete Zeit von Sortierverfahren wird bei Listen der Länge 1000, 10000 und 100000 gemessen. Die Listen haben jeweils zufällige Daten, sortierte Daten und invers sortierte Daten.

1.2 Verwendete Verfahren

Radixsort, Mergesort, Selectionsort, Shakersort, Quicksort, Bubblesort, Insertionsort, Shellsort und Heapsort wurden implementiert. Bei Quicksort wurde das Pivot-Element als Median aus erstem, mittleren und letztem Element bestimmt.

2 Ergebnisse

Die Messungen mit zufällig sortierten Zahlen wurden 10 Mal wiederholt.

Name	Sortiert			Invers			Zufällig		
	1e3	1e4	1e5	1e3	1e4	1e5	1e3	1e4	1e5
Bubble	0,4	44,5	3561	0,7	61,2	5871	0,7	135	16204
Merge	0,4	2,4	17,6	0,2	1,8	16,6	0,2	2,6	22,3
Heap	0,1	0,6	7,4	0,1	0,7	8,8	0,1	1,1	11,8
Shaker	0,0	0,0	0,1	1,2	124	11661	0,8	129	12480
Radix	0,1	1,2	6,2	0,1	1,4	5,9	0,4	5,5	65,9
Selection	0,1	14,8	1513	0,2	15,2	1560	0,2	15,4	1541
Insertion	0,0	0,0	0,3	0,7	69,9	6783	0,5	38,4	3343
Shell	0,0	0,1	1,3	0,0	0,3	2,3	0,1	0,8	11,1
Quick	0,0	0,1	1,5	0,0	0,1	1,8	0,1	0,6	6,8

Tabelle 2.1: Benötigte Zeit[ms] beim Sortieren

3 Interpretation der Ergebnisse

Quicksort ist grundsätzlich meist am schnellsten. Mergesort könnte noch als Natural Mergesort verbessert werden. Die Radixsort-Implementierung ist ebenfalls noch sehr naiv und das ist bei größeren Listenlängen bemerkbar. Bei sehr kleinen Listen ist es beinahe egal, welcher Algorithmus verwendet wird, denn alle werden in unter einer Millisekunde fertig. Bei großen Listen ist es sehr wichtig, effiziente Sortieralgorithmen zu verwenden, die besser als $O(n^2)$ skalieren.

4 Programm- bzw. Quellcode

4.1 Quellcode

Der komplette Quellcode inklusive Graph-Generierung wurde in Rust geschrieben. Ausgeführt der Code über das Tool „cargo“. Die Konfiguration der Messung erfolgt über den Quellcode.

Da gefordert wurde, dass der komplette Quellcode im Dokument enthalten ist, folgt eine Sektion mit demselben. Alternativ kann das Projekt auch angenehm auf GitHub unter [1] eingesehen werden. Die Dokumentation und Erklärung des Codes ist über „Doc Comments“ realisiert, kann also im Code durch Kommentare gelesen werden.

4.2 Vollständiger Quellcode

Listing 4.1: main.rs

```
extern crate rand;

use rand::{thread_rng, Rng};
use std::time::Instant;

fn main() {
    // list lengths
    let sizes: [usize; 3] = [1000, 10_000, 100_000];
    let random_samples = 10;
    let mut sorts: Vec<(Box<dyn Fn(&mut Vec<usize>)>, String)> = vec![
        (Box::new(bubble_sort), "bubble".to_owned()),
        (Box::new(merge_sort), "merge".to_owned()),
        (Box::new(heap_sort), "heap".to_owned()),
```

[1] https://github.com/imkgerC/uni-theo2-sort

```
        (Box::new(shaker_sort), "shaker".to_owned()),
        (Box::new(radix_sort), "radix".to_owned()),
        (Box::new(selection_sort), "selection".to_owned()),
        (Box::new(insertion_sort), "insertion".to_owned()),
        (Box::new(shell_sort), "shell".to_owned()),
        (Box::new(quick_sort), "quick".to_owned()),
    ];
    for size in &sizes {
        // sorted
        let mut vals = (0..(*size)).collect();
        for (sort, name) in &mut sorts {
            println!("sorted␣{}␣{}␣{}", name, size, measure(&vals, sort)
↪ );
        }
        // inversely sorted
        vals.reverse();
        for (sort, name) in &mut sorts {
            println!("inverse␣{}␣{}␣{}", name, size, measure(&vals, sort
↪ ));
        }
        // random
        for (sort, name) in &mut sorts {
            let mut sum = 0;
            for _ in 0..random_samples {
                let vals = (0..(*size)).map(|_| thread_rng().gen()).
↪ collect::<Vec<_>>();
                sum += measure(&vals, sort)
            }
            println!("random␣{}␣{}␣{}", name, size, sum / random_samples
↪ );
        }
    }
}
```

```
fn bubble_sort(arr: &mut Vec<usize>) {
    for j in 0..(arr.len() - 1) {
        for i in 1..(arr.len() - j) {
            if arr[i - 1] > arr[i] {
                arr.swap(i - 1, i);
            }
        }
    }
}

fn merge_sort(arr: &mut Vec<usize>) {
    mergesort(arr, 0, arr.len() - 1)
}

pub fn mergesort(arr: &mut Vec<usize>, b: usize, e: usize) {
    if b < e {
        let m = (b + e) / 2;
        mergesort(arr, b, m);
        mergesort(arr, m + 1, e);
        merge(arr, b, m, e);
    }
}
fn merge(arr: &mut Vec<usize>, b: usize, m: usize, e: usize) {
    let mut left = arr[b..m + 1].to_vec();
    let mut right = arr[m + 1..e + 1].to_vec();
    left.reverse();
    right.reverse();
    for k in b..e + 1 {
        if left.is_empty() {
            arr[k] = right.pop().unwrap();
            continue;
        }
        if right.is_empty() {
            arr[k] = left.pop().unwrap();
```

```
            continue;
        }
        if right.last() < left.last() {
            arr[k] = right.pop().unwrap();
        } else {
            arr[k] = left.pop().unwrap();
        }
    }
}

fn radix_sort(arr: &mut Vec<usize>) {
    let mut max = 0;
    for i in 0..arr.len() {
        if arr[i] > max {
            max = arr[i];
        }
    }
    let bits = 64 - max.leading_zeros();
    for i in 0..bits {
        let mut ones: Vec<usize> = Vec::with_capacity(arr.len());
        let mut zeros: Vec<usize> = Vec::with_capacity(arr.len());

        for j in 0..arr.len() {
            if (arr[j] & (1 << i)) > 0 {
                ones.push(arr[j]);
            } else {
                zeros.push(arr[j]);
            }
        }

        let mut j = 0;
        for x in zeros {
            arr[j] = x;
            j += 1;
```

```
        }
        for x in ones {
            arr[j] = x;
            j += 1;
        }
        assert!(j == arr.len());
    }
}

fn counting_sort(arr: &mut Vec<usize>) {
    let mut max = 0;
    for i in 0..arr.len() {
        if arr[i] > max {
            max = arr[i];
        }
    }
    let mut buckets = (0..max).map(|_| 0).collect::<Vec<_>>();
    for i in 0..arr.len() {
        buckets[arr[i] - 1] += 1;
    }
    let mut i = 0;
    for j in 0..arr.len() {
        while buckets[i] == 0 {
            i += 1;
        }
        buckets[i] -= 1;
        arr[j] = i + 1;
    }
}

fn heap_sort(arr: &mut Vec<usize>) {
    heapsort(arr, arr.len() as isize)
}
```

```rust
fn heapsort(arr: &mut Vec<usize>, n: isize) {
    // Build heap (rearrange array)
    let mut i = n / 2 - 1;
    while i >= 0 {
        heapify(arr, n as usize, i as usize);
        i -= 1;
    }

    // One by one extract an element from heap
    i = n - 1;
    while i > 0 {
        // Move current root to end
        arr.swap(0, i as usize);

        // call max heapify on the reduced heap
        heapify(arr, i as usize, 0);
        i -= 1;
    }
}

fn heapify(arr: &mut Vec<usize>, n: usize, i: usize) {
    let mut largest = i; // Initialize largest as root
    let l = 2 * i + 1; // left = 2*i + 1
    let r = 2 * i + 2; // right = 2*i + 2

    // If left child is larger than root
    if l < n && arr[l] > arr[largest] {
        largest = l;
    }

    // If right child is larger than largest so far
    if r < n && arr[r] > arr[largest] {
        largest = r;
    }
```

```
    // If largest is not root
    if largest != i {
        arr.swap(i, largest);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

fn selection_sort(arr: &mut Vec<usize>) {
    for i in 0..(arr.len() - 1) {
        let mut x = arr[i];
        let mut k = i;
        for j in (i + 1)..arr.len() {
            if arr[j] < x {
                x = arr[j];
                k = j;
            }
        }
        arr[k] = arr[i];
        arr[i] = x;
    }
}

fn shell_sort(arr: &mut Vec<usize>) {
    let column_count = [
        2147483647, 1131376761, 410151271, 157840433, 58548857,
    ↪ 21521774, 8810089, 3501671,
        1355339, 543749, 213331, 84801, 27901, 11969, 4711, 1968, 815,
    ↪ 271, 111, 41, 13, 4, 1,
    ];
    for k in 0..column_count.len() {
        let h = column_count[k];
```

```
        for i in h..arr.len() {
            let t = arr[i];
            let mut j = i;
            while j >= h && arr[j - h] > t {
                arr[j] = arr[j - h];
                j = j - h;
            }
            arr[j] = t;
        }
    }
}

fn quick_sort(arr: &mut Vec<usize>) {
    let len = arr.len() as isize;
    quicksort(arr, 0, len - 1)
}

fn quicksort(arr: &mut Vec<usize>, lo: isize, hi: isize) {
    if lo < hi {
        let p = partition(arr, lo, hi);
        quicksort(arr, lo, p - 1);
        quicksort(arr, p + 1, hi);
    }
}

fn partition(arr: &mut Vec<usize>, lo: isize, hi: isize) -> isize {
    let pivot_place;
    let mid = (lo + hi)/2;
    if arr[lo as usize] < arr[hi as usize] {
        if arr[mid as usize] < arr[lo as usize] {
            pivot_place = lo as usize;
        } else {
            if arr[mid as usize] < arr[hi as usize] {
                pivot_place = mid as usize;
```

```
            } else {
                pivot_place = hi as usize;
            }
        }
    } else {
        if arr[mid as usize] < arr[hi as usize] {
            pivot_place = hi as usize;
        } else {
            if arr[mid as usize] < arr[lo as usize] {
                pivot_place = mid as usize;
            } else {
                pivot_place = lo as usize;
            }
        }
    }
    let pivot = arr[pivot_place];
    let mut i = lo;
    for j in lo..hi {
        if arr[j as usize] < pivot {
            arr.swap(i as usize, j as usize);
            i += 1;
        }
    }
    arr.swap(i as usize, pivot_place);
    i
}

fn shaker_sort(arr: &mut Vec<usize>) {
    let mut swapped = true;
    while swapped {
        swapped = false;
        for i in 0..(arr.len() - 2) {
            if arr[i] > arr[i + 1] {
                swapped = true;
```

```
                arr.swap(i, i + 1);
            }
        }
        if !swapped {
            return;
        }
        swapped = false;
        for j in 0..(arr.len() - 2) {
            let i = (arr.len() - 2) - j;
            if arr[i] > arr[i + 1] {
                swapped = true;
                arr.swap(i, i + 1);
            }
        }
    }
}

fn insertion_sort(arr: &mut Vec<usize>) {
    for i in 1..arr.len() {
        let mut j = (i - 1) as isize;
        let x = arr[i];
        while j >= 0 && arr[j as usize] > x {
            j -= 1;
        }
        let mut k = i;
        while k >= (j + 2) as usize {
            arr[k] = arr[k - 1];
            k -= 1;
        }
        arr[(j + 1) as usize] = x;
    }
}
```

```
fn measure(arr: &Vec<usize>, sort: &mut Box<dyn Fn(&mut Vec<usize>)>) ->
    ↪ u128 {
    let mut arr = arr.clone();
    let now = Instant::now();
    sort.as_ref()(&mut arr);
    now.elapsed().as_micros()
}
```