Algorithms and Data Structures

Sorting 2

Marcin Sydow

Topics covered by this lecture:

- Stability of Sorting Algorithms
- Quick Sort
- Is it possible to sort faster than with $\Theta(n \cdot log(n))$ complexity?
- Countsort
- RadixSort

Stability

A sorting algorithm is stable if it preserves the original order of ties (elements of the same value)

Most sorting algorithms are easily adapted to be stable, but it is not always the case.

Stability is of high importance in practical applications. E.g. when the records of a database are sorted, usually the sorting key is one of many attributes of the relation. The equality of the value of this attribute does not mean equality of the whole records, of course, and is a common case in practice.

If sorting algorithm is stable it is possible to sort multi-attribute records **in iterations** - attribute by attribute (because the outcome of the previous iteration is not destroyed, due to stability)

Short recap of the last lecture

3 sorting algorithms were discussed up to now:

- selectionSort
- insertionSort
- mergeSort

Two first algorithms have **square** complexity but the third one is **faster** it has **linear-logarithmic** complexity.

In merge sort, the choice of the underlying **data structure** is important (linked list instead of array) to avoid unacceptably high **space complexity** of algorithm.

Quick Sort - idea

Quick sort is based on the "divide and conquer" approach.

The idea is as follows (recursive version):

1. For the sequence of length 1 nothing has to be done (stop the recursion)
2. longer sequence is reorganised so that some element M (called "pivot") of the sequence is put on "final" position so that there is no larger element "to the left" of M and no smaller element "to the right" of M.
3. subsequently steps 1 and 2 are applied to the "left" and "right" subsequences (recursively)

The idea of quick sort comes from C.A.R.Hoare.

Analysis

The algorithm described above can be efficient only when the procedure described in step 2 is efficient.

This procedure can be implemented so that it has **linear** time complexity and it works in place (constant space complexity) if we take comparison as the dominating operation and sequence length as the datasize.

Due to this, quick sort is efficient.

Note: the procedure is nothing different than **Partition** discussed on the third lecture.

Partition procedure - reminder

`partition(S, l, r)`
For a given sequence S (bound by two indexes l and r) the `partition` procedure **selects** some element M (called "pivot") and efficiently reorganises the sequence so that M is put on such a "final" position so that there is no larger element "to the left" of M and no smaller element "to the right" of M.
The partition procedure **returns** the final index of element M.

For the following assumptions:

- **Dominating operation:** comparing 2 elements
- **Data size**: the length of the array $n = (r - l + 1)$

The `partition` procedure can be implemented so that it's time complexity is $W(n) = A(n) = \Theta(n)$ and space complexity is $S(n) = O(1)$

Partition - possible implementation

input: a - array of integers; l,r - leftmost and rightmost indexes, respectively;
output: the final index of the "pivot" element M; the side effect: array is reorganised (no larger on left, no smaller on right)

```
partition(a, l, r){

  i = l + 1;
  j = r;
  m = a[l];
  temp;

  do{
    while((i < r) && (a[i] <= m)) i++;
    while((j > i) && (a[j] >= m)) j--;
    if(i < j) {temp = a[i]; a[i] = a[j]; a[j] = temp;}
  }while(i < j);
  // when (i==r):
  if(a[i] > m) {a[l] = a[i - 1]; a[i - 1] = m; return i - 1;}
  else {a[l] = a[i]; a[i] = m; return i;}
}
```

QuickSort - pseudo-code

Having defined partition it is now easy to write a recursive QuickSort algorithm described before:

input: a - array of integers; l,r - leftmost and rightmost indexes of the array

(the procedure does not return anything)

```
quicksort(a, l, r){

    if(l >= r) return;
    k = partition(a, l, r);
    quicksort(a, l, k - 1);
    quicksort(a, k + 1, r);
}
```

QuickSort - analysis

Let n denote the lenght of the array - data size.

Dominating operation: comparing 2 elements of the sequence

The above version of quick sort is recursive and its time complexity depends directly on the recursion depth.

Notice that on each level of the recursion the total number of comparisons (in partition) is of the rank $\Theta(n)$

QuickSort - analysis, cont.

The quick sort algorithm, after each `partition` call, calls itself recursively for each of 2 parts of “reorganised” sequence (assuming the length of subsequence is igher than 1)

First, for simplicity assume that the “pivot” element is put always in the middle of the array. In such a case the recursion tree is as in the merge sort algorithm (i.e. it is “**balanced**”). Thus, the recursion depth would be $\Theta(log(n))$.

In such a case, the time complexity of the algorithm would be: $T(n) = \Theta(n \cdot log(n))$

QuickSort - average complexity

It can be proved, that if we assume the uniform distribution of all the possible input permutations, the average time complexity is also **linear-logarithmic**:

$$A(n) = \Theta(n \cdot log(n))$$

Furthermore, it can be shown that the multiplicative constant is not high - about 1.44.

Both theoretical analyses and empirical experiments show that quick sort is one of the fastests sorting algorithms (that use comparisons). Thus the name - quick sort.

QuickSort - pessimistic complexity

The pessimistic case is when the recursion depth is maximum possible. What input data causes this?

QuickSort - pessimistic complexity

The pessimistic case is when the recursion depth is maximum possible. What input data causes this?

Input data which is **already sorted** (or invertedly sorted).

What is the recursion depth in such case?

QuickSort - pessimistic complexity

The pessimistic case is when the recursion depth is maximum possible. What input data causes this?

Input data which is **already sorted** (or invertedly sorted).

What is the recursion depth in such case? **linear** ($\Theta(n)$)

QuickSort - pessimistic complexity

The pessimistic case is when the recursion depth is maximum possible. What input data causes this?

Input data which is **already sorted** (or invertedly sorted).

What is the recursion depth in such case? **linear** ($\Theta(n)$)

Thus, the pessimistic complexity of the presented version of the QuickSort algorithm is, unfortunately **square**

$W(n) = \Theta(n^2)$

Properties of Quick Sort

The algorithm is fast in average case, however its pessimistic time complexity is a serious drawback.

To overcome this problem many "corrected" variants of quick sort were invented. Those variants have **linear-logarithmic pessimistic** time complexity (e.g. special, dedicated sub-procedures for sorting very short sequences are applied)

Ensuring **stability** is another issue in quicksort. Adapting `partition` procedure to be stable is less natural compared to the algorithms discussed before.

Space complexity Finally, notice that quicksort sorts **in place** but it does not yet mean: S(n)=O(1). Recursion implementation has its implicit memory cost, the algorithm has pessimistic $O(n)$ (linear!) pessimistic space complexity. It is possible to re-write one of the two recursive calls (the one that concerns the longer sequence) as iterative one, what results in $\Theta(log(n))$ pessimistic space complexity.

Is it possible to sort faster?

Among the algorithms discussed up to now, the best **average** time complexity order is **linear-logarithmic**[1] (merge sort, quick sort).

Is there comparison-based sorting algorithm which has better rank of time complexity?

[1] Assuming comparison as the dominating operation and sequence length as the data size

Is it possible to sort faster?

Among the algorithms discussed up to now, the best **average** time complexity order is **linear-logarithmic**[1] (merge sort, quick sort).

Is there comparison-based sorting algorithm which has better rank of time complexity?

It can be mathematically proven that the answer is **negative**: i.e. **linear-logarithmic** average time complexity is **the best possible for comparison-based sorting** algorithms!

[1] Assuming comparison as the dominating operation and sequence length as the data size

Linear-logarithmic bound - explanation

The problem of sorting n-element sequence by means of comparisons can be viewed as follows. The task is to discover the permutation of the "original" (sorted) sequence by asking binary "questions" (comparisons).

Thus any comparison-based sorting algorithm can be represented as a **binary decision tree**, where each node is a comparison and each leaf is the "discovered permutation". Notice that **the number of leaves is n!** (factorial)

Thus, the number of necessary comparisons (time complexity) is the length of path from root to a leaf (**height of tree**). It can be shown that for any binary tree with $n!$ leaves its average height is of rank $\Theta(log(n!)) = \Theta(n \cdot log(n))$ (n is lenght of sequence)

Beyond comparisons...

To conclude:

is it possible to sort faster than with linear-logarithmic time complexity?

Beyond comparisons...

To conclude:

is it possible to sort faster than with linear-logarithmic time complexity?

yes

Beyond comparisons...

To conclude:

is it possible to sort faster than with linear-logarithmic time complexity?

yes

how is it possible?

Beyond comparisons...

To conclude:

is it possible to sort faster than with linear-logarithmic time complexity?

yes
how is it possible?

It is possible to beat the limit if we **do not use comparisons**.

In practice, it means **achieving lower time complexity with higher space complexity**.

It is the most typical “deal” in algorithmics: “time vs space”.

CountSort algorithm

The idea of the algorithm is based on application of **direct addressing** to place the sorted elements on their final positions.

The necessary technical assumption here is that the input data fits in Random Access Memory (RAM). The algorithm **does not use comparisons**.

The algorithm has **lower time complexity** than quick sort, but the price is **very high space complexity** (2 helper arrays).

CountSort - code

input: a - array of **non-negative** integers; l - its length

```
countSort(a, l){

    max = maxValue(a, l);
    l1 = max + 1;
    counts[l1];
    result[l];
    for(i = 0; i < l1; i++) counts[i] = 0;

    for(i = 0; i < l; i++) counts[a[i]]++;
    for(i = 1; i < l1; i++) counts[i] += counts[i - 1];
    for(i = l - 1; i >= 0; i--)
       result[--counts[a[i]]] = a[i];
}
```

(in the last line, notice **pre**-decrementation to avoid shifing all the elements by 1 to the right)

CountSort - analysis

dominating operation: put value into array

data size (2 arguments): length of sequence n, maximum value in the sequence

The algorithm needs 2 sequential scans through the arrays (n-element one and m-element one). Its time complexity is linear (!).

$$A(n,m) = W(n,m) = 2n + m = \Theta(n,m)$$

Unfortunately, the space complexity is also **linear** (very high):

$$S(n,m) = n + m = \Theta(n,m)$$

RadixSort

The Radix Sort algorithm is a **scheme** of sorting rather than a proper sorting algorithm. It applies **another, inernal** sorting algorithm.

It is ideal for **lexicographic sort** of object sequences having fixed length (e.g. strings, multi-digit numbers, etc.)

Radix sort applies any **stable** sorting algorithm to all consecutive positions of the sorted objects starting from the **last position** to the first one.

If the universe of symbols (digits, alphabet, etc.) is fixed and small, the count sort algorithm is a very good choice for the internal algorithm.

Questions/Problems:

- Stability
- Partition
- QuickSort
- Lower bound for sorting by comparisons
- CountSort
- Comparative analysis of (strong and weak) properties of all sorting algorithms discussed
- RadixSort

Thank you for your attention!