Algorithms and Data Structures

Searching

Marcin Sydow

Topics covered by this lecture:

- **"Divide and Conquer"** Rule
- Searching
- **Binary Search** Algorithm
- Positional Statistics
- The second smallest element (The "Tournament" algorithm - idea)
- Hoare's Algorithm (idea)

"Divide and Conquer"

One of the most important methods for **algorithm design**. Divide the problem into a couple of smaller sub-problems such that it is easy to reconstruct the solution from the sub-solutions

It is quite often implemented as a **recursion** - a programming technique in which a function calls itself (for a sub-problem)

"Divide and Conquer" has origins in politics, and its name (originaly in Latin: "Divide et Impera") is traditionally assigned to Philip II, the king of Macedonia (382-336 BC) in the context of his rules over Greeks (source: Wikipedia)

The Searching Problem

`search(S, len, key)`

Input: S - a sequence of integers (indexed from 0 to len-1); len - length of the sequence; key - integer number (to be found)

Output: index - a natural number less than len, meaning (any) index in the sequence S under which the key is present (i.e. S[index]==key) or -1 if the key is not present in the sequence

E.g. for S = (3,5,8,2,1,8,4,2,9), and the above specification, search should behave as follows:

- search(S, 9, 2) stops and returns: 3
- search(S, 9, 7) stops and returns: -1

A natural candidate for **dominating operation** in standard algorithms solving the searching problem is **comparison operation** and the **data size** is usually the length of the sequence (len)

One can apply the **sequential search** algorithm to solve this problem (it was discussed on the previous lectures). It has **linear time complexity** $W(len) = \Theta(len)$. The **multiplicative factor is 1** (W(len) = len). and it cannot be improved, because of the specificity of the problem.

More Effective Searching?

What **additional property** of the input sequence would help to search more efficiently?

More Effective Searching?

What **additional property** of the input sequence would help to search more efficiently?

Sorting the input sequence

Searching in Sorted Sequence

It is a **different** problem (because of a different specification - the input condition is different)

Input: S - a sequence of **non-decreasingly sorted**[1] integers (indexed from 0 to len-1); len - a natural number - length of the sequence; key - integer number

Output: index - a natural number less than len meaning (any) index in the sequence S such that S[index]==key or -1 if key is not present in S.

With such an **additional** assumption on the input data it is possible to solve the problem **more efficiently** (faster) than previously

[1] if the sequence is non-increasingly sorted it also helps (in similar way) - but this would be even another problem specification

"Skipping" Algorithm

If input sequence is sorted it is possible to check each k-th cell (skipping k-1 elements on each jump) and, in case of finding the first number which is higher than the key, check only the last k-1 elements (or if the sequence ended return -1)

Notice that such an algorithm is asymptotically (i.e. for $len \to \infty$) **k times faster** (on average) than "normal" sequential search[2] (for unordered input sequence): $W(len) = \frac{1}{k} \cdot \Theta(len)$, but is is still **linear** - thus the **rank** of complexity was not improved with this version of the algorithm

Is it possible to improve the complexity rank of searching for sorted input sequence?

[2] it is possible to prove that the optimal choice for k, in terms of pessimistic complexity, is $k = \sqrt{len}$

`search(S, len, key)`
(input sequence is sorted)

The Binary Search Algorithm (the "Divide and Conquer" approach)

1. while the length of sequence is positive:
2. check the middle element of the current sequence
3. if it is equal to key - return the result
4. if it is higher than key - restrict searching to the "left" sub-sequence (from the current position)
5. if it is less than key - restrict searching to the "right" sub-sequence (from the current position)
6. back to the point 1
7. there is no key in the sequence (if you are here)

Binary Search Algorithm

```
search(S, len, key){

  l = 0
  r = len - 1

  while(l <= r){
      m = (l + r)/2
      if(S[m] == key) return m
      else
        if(S[m] > key) r = m - 1
        else l = m + 1
  }

  return -1
}
```

Notice that the operation of **random access** (direct access) to the m-th element S[m] of the sequence demands that the sequence is kept in RAM (to make the operation efficient)

Analysis of the Binary Search Algorithm

Data size: length of the sequence - len
Dominating operation: comparison - `(S[m] == key)`
(assume the sequence is kept in RAM)

With each iteration the sequence becomes 2 times shorter. The algorithm stops when the length of sequence becomes 1

$W(len) = \Theta(log_2(len))$

$A(len) = \Theta(log_2(len))$

$S(len) = O(1)$

(Notice: the assumption about data in RAM is important)

Observation: if data is sorted but it does not fit into RAM, this analysis is **inadequate** (because comparison `S[m] == key` cannot be considered as an "atomic" operation)

Positional Statistics

The k-th positional statistic in a sequence of elements (which can be ordered) is the k-th smallest (largest) element in the sequence

E.g. the task of finding minimum is nothing different than searching the 1-st positional statistic in this sequence

In the case of sorted sequence this task is trivial - k-th statistic is just the k-th element in the sequence.

Searching the 2nd Smallest Element in Sequence

`second(S, len)`
(no assumption that the sequence is sorted)

Input: S - sequence of elements that can be ordered (e.g. integers, symbols of alphabet, etc.); len - the length of sequence

Output: the second smallest element s of the sequence S

Data size, for algorithm solving this problem, is usually the length of sequence and the dominating operation (usually) comparison.

A simple solution: find minimum, exclude it from the sequence, and repeat this (needs $2 \cdot len$ comparisons).

Searching the 2nd Smallest Element in Sequence

`second(S, len)`
(no assumption that the sequence is sorted)

Input: S - sequence of elements that can be ordered (e.g. integers, symbols of alphabet, etc.); len - the length of sequence

Output: the second smallest element s of the sequence S

Data size, for algorithm solving this problem, is usually the length of sequence and the dominating operation (usually) comparison.

A simple solution: find minimum, exclude it from the sequence, and repeat this (needs $2 \cdot len$ comparisons).

Can it be done **more** effectively?

The “Tournament” Algorithm - idea

Let's apply the “divide and conquer” method:
Imagine a “tournament” proceeding in turns.

In each turn the sequence is divided into pairs. Both elements in each pair “compete” with each other - the winner is the smaller one. Only the winners from the current turn survive to the next turn.

We stop when the sequence consist of only 1 element - this is the smallest element in the original sequence.

But where in the “tournament history” is the **second smallest**?

The “Tournament” Algorithm - idea

Let's apply the “divide and conquer” method:
Imagine a “tournament” proceeding in turns.

In each turn the sequence is divided into pairs. Both elements in each pair “compete” with each other - the winner is the smaller one. Only the winners from the current turn survive to the next turn.

We stop when the sequence consist of only 1 element - this is the smallest element in the original sequence.

But where in the “tournament history” is the **second smallest**?

Among the direct competitors of the winner - the second smallest could lose only with the smallest.

Analysis of the "Tournament" Algorith

The tournament could be naturally represented as a tree, where the lowest level (the leaves) is the original sequence and the root is the winner. The number of levels is $O(log_2(len))$

Data size: length of the original sequence - len
Dominating operation: comparison between 2 elements

All the comparisons in the tournament need exactly **len-1** comparison operations (why?)

In the final phase (seeking for the second smallest) the whole path from the leaf (the first competitor of the subsequent winner) to the root is scanned for the minimum - accounting for another $O(log_2(len))$ comparisons (explained later)

Thus, this algorithm has **lower complexity** than "repeated minimum", but **the same rank** (linear)

K-th positional statistics - the Hoare's Algorithm (idea)

`kthSmallest(S, len, k)`
(no assumption of order of the input sequence)

Input: S - sequence of elements that can be ordered (e.g. integers, alphabet symbols, etc.); len - the length of the sequence; k - positive natural number ("which positional statistic are we searching for?")

Output: the element s in the sequence S, being the k-th smallest element

As previously, it is possible to repeat k times the minimum search (excluding the minimum each time) - it needs to linearly scan the sequence k times. However, the task can be solved much more efficiently with application of the "divide and conquer" method.

The Partition Procedure

`partition(S, l, r)`
input: S - sequence of elements that can be ordered; l - the left (starting) index of the subsequence to be processed; r - the right (ending) index
output: i - the final position of the "median" element M (see below in the description)

For a given sequence S the `partition` procedure effectively **selects** some element M and **re-organises** the sequence S such that all the elements on the left of M are not greater than M and all the elements on the right of M are not less than M. The procedure finally **returns** the index *i* of the element M.

Analysis of Partition

Notice the following property: if the index returned by partition equals k the element under this index (M) is the k-th positional statistic in S (due to the partition formulation) (assume indexing from 1).

For the following assumptions:

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

The **partition** procedure can be designed such that its complexity $W(n) = n + O(1)$ and $S(n) = O(1)$

The Hoare's Algorithm - idea

The `partition` procedure and the "divide and conquer" method can be applied to design a very efficient algorithm for searching the k-th positional statistics, as follows:

- call `partition` on the sequence
- if the returned index is equal to k - return $S[k]$
- else, depending on the value returned by the partition, continue (from point 1) on the "left" or "right" sub-sequence from M (take into account the number of "abandoned" elements to the left)

The time complexity of the above algorithm is linear (**independently on k**) and, on average, it is much faster than the "repeated minimum" algorithm. More detailed discussion of this algorithm will be given later (with **quickSort**).

The above algorithm (together with the partition procedure) was invented by J.R.Hoare.

Questions/Problems:

- The problem of searching (sequential algorithm)
- Searching in a sorted sequence (skipping k elements)
- **Binary Search** Algorithm (analysis + code)
- Positional Statistics
- The "Tournament" Algorithm
- The Partition Procedure (only the specification)
- The Hoare's Algorithm (only the idea)

Thank you for your attention