Algorithms and Data Structures

Searching in Lists

Ulf Leser
Topics Today

• **Search**: Given a (sorted or unsorted) list \( A \) with \( |A| = n \) elements (integers). Check whether a given value \( c \) is contained in \( A \) or not
  - Search returns true or false
  - In the sorted case, we obviously can exploit transitivity
  - Fundamental problem with a zillion applications

• **Select**: Given an unsorted list \( A \) with \( |A| = n \) elements (integers). Return the \( i \)'th largest element of \( A \).
  - Returns an element of \( A \)
  - The sorted case is trivial – simply return \( A[i] \)
  - Interesting problem (especially for median) with many applications
  - [Interesting proof]
Content of this Lecture

- Searching in Unsorted Lists
- Searching in Sorted Lists
- Selecting in Unsorted Lists
Searching in an Unsorted List

- There is not much we can do, no magic is known
- Compare c to every element of A
- Worst case (c \not\in A): O(n)
- Average case (c \in A)
  - We perform i tests for all i with the same probability 1/n
  - This gives

\[
\frac{1}{N} \sum_{i=1}^{N} i = \frac{1}{N} \cdot \frac{N^2 + N}{2} = \frac{N + 1}{2} = O(N)
\]

```plaintext
1. A: unsorted_int_array;
2. c: integer;
3. for i := 1.. |A| do
4.   if A[i]=c then
5.     return true;
6.   end if;
7. end for;
8. return false;
```
Content of this Lecture

- Searching in Unsorted Lists
- **Searching in Sorted Lists**
  - Binary Search
  - Fibonacci Search
  - Interpolation Search
- Selecting in Unsorted Lists
Binary Search (binsearch)

- If A is sorted, we can be much faster
- Binsearch: Exploit transitivity

```plaintext
1. func bool binsearch(A: sorted_arr; c, l, r : int) {
2.   If l>r then
3.     return false;
4.   end if;
5.   m := l+(r-l) div 2;
6.   If c<A[m] then
7.     return binsearch(A, c, l, m-1);
8.   else if c>A[m] then
9.     return binsearch(A, c, m+1, r);
10.  else
11.    return true;
12.  end if;
13. }
```
Iterative Binsearch

- Binsearch uses only end-recursion
- Thus, transformation to an equivalent iterative program is easy
  - No call stacks
  - \(O(1)\) additional space

```plaintext
1. A: sorted_int_array;
2. c: integer;
3. l := 1;
4. r := |A|;
5. while l<r do
6.   m := l+(r-l) div 2;
7.   if c<A[m] then
8.     r := m-1;
9.   else if c>A[m] then
10.    l := m+1;
11.   else
12.     return true;
13. end while,
14. return false;
```
Complexity of Binsearch

- With every call to binsearch (or every while-loop), we reduce the size of sub-array by 50%
- In every call to binsearch, we only do constant work
- Thus, we call binsearch once with n, with n/2, with n/4, ... = \log(n+1) times
- Binsearch has worst-case complexity \(O(\log(n))\)
- Average case is only marginally better
  - Ottmann/Widmayer

Source: railspikes.com
Content of this Lecture

- Searching in Unsorted Lists
- Searching in Sorted Lists
  - Binary Search
  - Fibonacci Search
  - Interpolation Search
- Selecting in Unsorted Lists
Fibonacci Search

- If we want to be ultra-fast, we should try to use only **simple arithmetic operations**
  - Division is not simple
- We want a search algorithm that has complexity $O(\log(n))$ and does not use division
- We need to “imitate” the iterative halving of indexes
- Recall **Fibonacci numbers**
  - $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$
  - 1, 2, 3, 5, 8, 13, 21, 34, ...
  - Thus, $\text{fib}(n-2)$ is roughly 1/3, $\text{fib}(n-1)$ roughly 2/3 of $\text{fib}(n)$
- Dividing the array like this might suffice for $O(\log(n))$
Complexity

- Let’s assume we can always compute $x \sim 1+2*(r-l)/3$ using only integer additions and subtractions
- In the worst-case, we always have $c$ in the larger $(2/3)$ fraction of the array
  - We call once for $n$, once for $2n/3$, once for $4n/9$, ..., 1
- I.e., we look at arrays of size $\text{fib}(n-1), \text{fib}(n-2), \text{fib}(n-3), ...$
- Consider that
  \[
  \text{fib}(n) = \left[ \frac{1}{\sqrt{5}} \left( \frac{1+\sqrt{5}}{2} \right)^n \right] \sim c*1.62^n
  \]
- Thus, for $n \sim c*1.62^n$ (for some $n'$) we make $O(n')$ comparisons
- We thus need $1/c*\log_{1.62}(n)=O(\log(n))$ comparisons
Algorithm

- Not totally trivial
- Having only fib(n) doesn’t suffice to compute fib(n-1) and fib(n-2)
- But if we know fib(n), fib(n-1) and fib(n-2), we can compute all other fib’s
  - fib(n)=fib(n-1)+fib(n-2)
  - fib(n-1)=fib(n-2)+fib(n-3)
  - ...
- Always keep fib, fib1, and fib2
- **Offset**: Never move outside A

```plaintext
1. A: sorted_int_array;
2. c: integer;
3. fib2 := 1;
4. fib1 := 1;
5. fib := 2;
6. while fib<n do
7.   fib2 := fib1;
8.   fib1 := fib;
9.   fib := fib1+fib2;
10. end while;
11. i := 0;
12. offset := 0;
13. while fib>1 do
14.   i := min(offset+fib2, n)
15.   if c<A[i] then
16.     fib := fib2;
17.     fib1 := fib1-fib2;
18.     fib2 := fib-fib1;
19.   else if c>A[i] then
20.     fib := fib1;
21.     fib1 := fib2;
22.     fib2 := fib-fib1;
23.     offset := i;
24.   else
25.     return true;
26.   end if;
27. end while;
28. return false;
```
1. A: sorted_int_array;
2. c: integer;
3. fib2 := 1;
4. fib1 := 1;
5. fib := 2;
6. while fib<n do
7.   fib2 := fib1;
8.   fib1 := fib;
9.   fib := fib1+fib2;
10. end while;
11. i := 0;
12. offset := 0;
13. while fib>1 do
14.   i := min(offset+fib2, n)
15.   if c<A[i] then
16.     fib := fib2;
17.     fib1 := fib1-fib2;
18.     fib2 := fib-fib1;
19.   else if c>A[i] then
20.     fib := fib1;
21.     fib1 := fib2;
22.     fib2 := fib-fib1;
23.     offset := i;
24.   else
25.     return true;
26.   end if;
27. end while;
28. return false;
Outlook

• We can solve the search problem in $O(\log(n))$ using only comparisons
• Transform $A$ into a balanced binary search tree, i.e,
  – At every node, the depth of the two subtrees differ by at most 1
  – At every node $n$, all values in the left (right) subtree are smaller (larger) than $n$
• Search problem
  – Recursively compare $c$ to node labels and descend left/right
  – Tree has depth $O(\log(n))$
  – We need at most $\log(n)$ comparisons – and nothing else
• See Heap-based Priority Queues later
Content of this Lecture

- Searching in Unsorted Lists
- Searching in Sorted Lists
  - Binary Search
  - Fibonacci Search
  - Interpolation Search
- Selecting in Unsorted Lists
Interpolation Search

- Imagine you have a telephone book and search for „Zacharias“
- Will you open the book in the middle?
- As in sorting, we can exploit additional knowledge about our values, i.e., use more than just comparisons
- Interpolation Search: Estimate where c lies in A based on the distribution of values in A
  - Simple: Use max and min values in A and assume equal distribution
  - Complex: Approximation of real distribution (histograms, ...)
Simple Interpolation Search

- Assume **equal distribution** – values within A are equally distributed in \([ A[1], A[n] ]\)
- Best guess for the **rank of c**

\[
rank(c) = l + (r - l) \times \frac{c - A[l]}{A[r] - A[l]}
\]

- Idea: Use \(m = rank(c)\) and proceed as in binsearch
- Example: “Xylophon”
Analysis

• In average-case, Interpolation Search on equally distributed data requires only $O(\log(\log(N)))$ comparison
  – See [OM93]
• But: Worst-case is $O(N)$
  – If concrete distribution deviates heavily from expected distribution, e.g., A is very large and contains only names “Xanthippe”
• Further disadvantage: In each phase, we perform $\sim 4$ adds/subs and 2*mults/divs
  – Assume this takes 12 cycles (1 mult/div = 4 cycles)
  – Binsearch requires 2*adds/subs + 1*div $\sim 6$ cycles
  – Even for $N=2^{32} \sim 4E9$, this yields $12*\log(\log(4E9)) \sim 72$ ops versus $6*\log(4E9) \sim 180$ ops – not that much difference
Going Further

- For very large N, it might be worth to use additional knowledge on A
- Idea: If $|\Sigma|=k$, pre-compute the frequency $f(k)$ of values starting with a character smaller-or-equal than $k$ - for all $k$
  - Names: How many start with A, A or B, A or B or C, ...
  - Pre-computation: One scan, or use sampling
- Given $c$, use $f(c[1])$ as start point
  - More on this: Histograms in databases
Content of this Lecture

- Searching in Unsorted Lists
- Searching in Sorted Lists
- Selecting in Unsorted Lists
  - Naïve or clever
Quartiles

- The median is the middle value
  - Sort all values and take the one in the middle
- Generalization: x%-Quartiles
  - Sort all values and take the value at x% of the values
  - Typical: 25, 75, 90, -quartiles
    - How long do 90% of all students need?
  - Median = 50%-quartile
Selection Problem

• Definition
  The selection problem is to find the \( x\% \)-quartile of a set of \( A \) of \(|A|\) unsorted values

• We can sort \( A \) and then take the appropriate value directly
• Thus, \( O(n \times \log(n)) \) is easy to reach
• Can we solve the problem in linear time?
• It is easy to see that we have to look at least at each value once; thus, the problem is in \( \Omega(n) \)
Top-k Problem

- **Top-k**: Find the k largest values in A
- For small k, the naïve solution already is linear
  - repeat k times
  - go through A and find largest value v;
  - remove v from A;
  - return v
  - Requires $k \cdot |A| = O(|A|)$ comparisons
- Naïve solution is optimal for constant k
- But if $k = c \cdot |A|$, we need $c \cdot |A| \cdot |A| = O(|A|^2)$ comparisons
  - See: It is decisive whether k depends on input or not
  - We measure complexity in size of the input – but what is the input?
Selection Problem in Linear Time

- We sketch an algorithm which solves the problem for arbitrary x in linear time
  - Actually, we solve the equivalent problem of returning the k’th value in the sorted A (of course, without sorting A)
- Interesting from a theoretical point-of-view: It is possible
- Practically, the algorithm is of no importance because the constant factors might get enormously large
- It is instructive to see why (and where)
Algorithm

- Recall **QuickSort**: Chose pivot element \( p \), divide array wrt \( p \), recursively sort both partitions using the same trick.
- We can reuse the idea: Chose pivot element \( p \), divide array wrt \( p \), recursively select in the one partition that must contain the \( k \)'th element.

```plaintext
func integer divide(A array; l,r integer) {
    ...
    while true
    repeat
        i := i+1;
    until A[i]>=val;
    repeat
        j := j-1;
    until A[j]<=val or j<i;
    if i>j then
        break while;
    end if;
    swap( A[i], A[j]);
end while;
swap( A[i], A[r]);
return i;
}
```

```plaintext
func int quartile(A array; k, l, r int) {
    if r≤l then
        return A[l];
    end if;
    pos := divide( A, l, r);
    if (k ≤ pos-1) then
        return quartile(A, k, l, pos-1);
    else
        return quartile(A, k-pos+1, pos, r);
    end if;
}
```
Analysis

- Assume **arbitrarily badly chosen** pivot elements
  - Worst-case
- pos always r-1 (or l+1)
- Gives $O(n^2)$
- Need to chose the pivot element v **more carefully**
Chosing p

- Assume we can chose p such that we always continue with only q% of A
  - Any q, but extend of reduction depends on n
- Then, we would perform $T(n) = T(q*n) + c*n$ operations
  - $T(q*n)$ – recursive descent
  - $c*n$ – function “divide”
- $T(n) = T(q*n) + c*n = T(q^2*n) + q*c*n + c*n = T(q^2*n) + (q+1)*c*n = T(q^3*n) + (q^2 + q + 1)*c*n = ...$

$$T(n) \leq c \cdot n \cdot \sum_{i=0}^{\infty} q^i = c \cdot n \cdot \frac{1}{1-q} = O(n)$$
Discussion

• Our algorithm has **worst-case complexity** $O(n)$ when we manage to always reduce the array by a fraction of its size – no matter, how large the fraction
  – Beware: This is not an average-case. We require to always (not on average) cut some fraction of A

• Eh – magic?

• No – follows from the **way we estimate complexity** and what we consider as input

• Many operations now are “hidden” in the constant factors
  – $q=0.9$: $c\times10\times n$
  – $q=0.99$: $c\times100\times n$
  – $q=0.999$: $c\times1000\times n$
Median-of-Median

- How can we guarantee to always cut a fraction of $A$?
- **Median-of-median** algorithm
Median-of-Median

- How can we guarantee to always cut a fraction of A?
- **Median-of-median** algorithm
  - Partition A in stretches of length 5
Median-of-Median

- How can we guarantee to always cut a fraction of $A$?
- **Median-of-median** algorithm
  - Partition $A$ in stretches of length 5
  - Compute the median $v_i$ for each partition (with $i<\text{floor}(n/5)$)
Median-of-Median

• How can we guarantee to always cut a fraction of A?
• **Median-of-median** algorithm
  – Partition A in stretches of length 5
  – Compute the median $v_i$ for each partition (with $i<\text{floor}(n/5)$)
  – Use the **median** $v$ of all $v_i$ as pivot element
    • Note: We are not using the $v$'th element of A, but we generate the value for dividing A into two halves by analyzing A
Median-of-Median

• How can we guarantee to always cut a fraction of A?

• Median-of-median algorithm
  – Partition A in stretches of length 5
  – Compute the median $v_i$ for each partition (with $i<\text{floor}(n/5)$)
  – Use the median $v$ of all $v_i$ as pivot element
    • Note: We are not using the $v$'th element of A, but we generate the value for dividing A into two halves by analyzing A
  – This is possible in $O(n)$
    • Run through A in jumps of length 5
    • Find each median in constant time (“sorting” of lists of length 5 – 5 not dependent on $n$ – constant time)
    • Call algorithm recursively on all medians
    • Since we always reduce the range of values to look at by 80%, this requires $O(n)$ time (see previous slides)
Why Does this Help?

- We have $\sim n/5$ first-level-medians $v_i$
- $v$ (as median of medians) is smaller than halve of them and greater than the other halve (both $\sim n/10$ values)
- Each $v_i$ itself is smaller than (greater than) 2 values from $A$
- Since for the smaller (greater) medians this median itself is also smaller (greater) than $v$, $v$ is larger (smaller) than at least $3*n/10$ elements
• Finding median-of-median of a randomly permuted list of values 0..99
• For clarity, each 5-tuple is sorted (top-down) and all 5-tuples are sorted by median (left-right)
• Gray/white: Values with actually smaller/greater than median 47
• Blue: Range with certainly smaller / larger values