Topic of Next Lessons

- **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
  - If $A$ is sorted, we 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 – 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

- No magic
- Compare c to every element of A
- Worst case (c \notin A): O(n)
- Average case (c \in A)
  - If c is at position i, we require i tests
  - All positions are equally likely: probability \frac{1}{n}
  - This gives

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

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>A: unsorted_int_array;</td>
</tr>
<tr>
<td>2.</td>
<td>c: int;</td>
</tr>
<tr>
<td>3.</td>
<td>for i := 1..</td>
</tr>
<tr>
<td>4.</td>
<td>if A[i]=c then</td>
</tr>
<tr>
<td>5.</td>
<td>return true;</td>
</tr>
<tr>
<td>6.</td>
<td>end if;</td>
</tr>
<tr>
<td>7.</td>
<td>end for;</td>
</tr>
<tr>
<td>8.</td>
<td>return false;</td>
</tr>
</tbody>
</table>
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_array; c, l, r : int) {
2.   If l>r then
3.     return false;
4.   end if;
5.   m := (l+r) 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. }
```

Source: railspikes.com
Iterative Binsearch

- Binsearch uses only end-recursion
- Equivalent **iterative program**
  - No call stack
  - We don’t need old values for l,r
  - O(1) additional space

```plaintext
1. A: sorted_int_array;
2. c: int;
3. l := 1;
4. r := |A|;
5. while l≤r do
6.   m := (l+r) 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

• In every call to binsearch (or every while-loop), we only do constant work
• With every call, we reduce the size of sub-array by 50%
  – We call binsearch once with \( n \), with \( n/2 \), with \( n/4 \), ...
• Binsearch has worst-case complexity \( O(\log(n)) \)
• Average case only marginally better
  – Chances to “hit” target in the middle of an interval are low in most cases
  – See 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
Searching without Divisions

• If we want to be ultra-fast, we should use only simple arithmetic operations

• **Fibonacci search:** $O(\log(n))$ without division/multiplication
  – Note: Bin-search usually uses bit shift (div 2) – very fast
  – Fibonacci search also has slightly better access locality (cache)
  – Also interesting: $O(\log(n))$ without the “always 50%” trick

• Recall **Fibonacci numbers**
  – $\text{fib}(1)=\text{fib}(2)=1; \text{fib}(i)=\text{fib}(i-1)+\text{fib}(i-2)$
  – 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
  – Thus, fib(i-2) is roughly 1/3, fib(i-1) roughly 2/3 of fib(i)
Fibonacci Search: Idea

- Let $\text{fib}(i)$ be the smallest fib-number $>|A|$
- If $A[\text{fib}(i-2)] = c$: stop
- Otherwise, continue searching in $[1 \ldots \text{fib}(i-2)]$ or $[\text{fib}(i-2)+1 \ldots n]$
- Beware out-of-range part $A[n+1\ldots\text{fib}(i)]$
- No divisions
Algorithm (assume |A| = \text{fib}(n) - 1)

- 3-6: Search at A[\text{fib}(i-2)]
  - With fib1, fib2 we can compute all other fib’s
  - \text{fib}(i) = \text{fib}(i-1) + \text{fib}(i-2)
  - \text{fib}(i-1) = \text{fib}(i-2) + \text{fib}(i-3)
  - ...
- 7-24: Break A at descending Fibonacci numbers
- After each comparison, update fib1 and fib2

```
1. A: sorted_int_array;
2. c: int;
3. compute i;
4. fib1 := \text{fib}(i-3);
5. fib2 := \text{fib}(i-2);
6. m := fib2;
7. repeat
   8. if c > A[m] then
      9.   if fib1 = 0 then return false
          10.   else
               11.     m := m + fib1;
               12.     tmp := fib1;
               13.     fib1 := fib2 - fib1;
               14.     fib2 := tmp;
               15.   end if;
    16. else if c < A[m]
        17.   if fib2 = 1 then return false
            18.   else
                19.     m := m - fib1;
                20.     fib2 := fib2 - fib1;
                21.     fib1 := fib1 - fib2;
                22.   end if;
        23.   else return true;
        24. until true;
```
Example

Search 3 in \{1,2,3\}

<table>
<thead>
<tr>
<th>fib2</th>
<th>fib1</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

true

Search 6 in \{1,2,3,4\}

<table>
<thead>
<tr>
<th>fib2</th>
<th>fib1</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

false

Search 100 in \{1...10000\}

<table>
<thead>
<tr>
<th>fib2</th>
<th>fib1</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>4181</td>
<td>2584</td>
<td>4181</td>
</tr>
<tr>
<td>1597</td>
<td>987</td>
<td>1597</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

1. A: sorted_int_array;
2. c: int;
3. compute i;
4. fib1 := fib(i-3);
5. fib2 := fib(i-2);
6. m := fib2;
7. repeat
8. if c > A[m] then
9. if fib1 = 0 then return false
10. else
11. m := m + fib1;
12. tmp := fib1;
13. fib1 := fib2 - fib1;
14. fib2 := tmp;
15. end if;
16. else if c < A[m]
17. if fib2 = 1 then return false
18. else
19. m := m - fib1;
20. fib2 := fib2 - fib1;
21. fib1 := fib1 - fib2;
22. end if;
23. else return true;
24. until true;
Complexity

- Worst-case: C is always in the larger (fib1) fraction of A
  - We roughly call once for n, once for 2n/3, once for 4n/9, …
- Formula of Moivre-Binet:
  - \( \text{fib}(i) = \text{round}\left(\frac{\phi^i}{\sqrt{5}}\right) \approx \frac{\phi^i}{\sqrt{5}} \approx c \times 1.62^i \)
  - Where \( \phi := \text{golden ratio} \approx 1.62 \)
- We find fib such that \( \text{fib}(i-1) \leq n \leq \text{fib}(i) \sim c \times 1.62^i \)
- In worst-case, we make \( \sim i \) comparisons
  - We break the array i times
- Since \( i = \log_{1.62}(n/c) \), we are in \( O(\log(n)) \)
Outlook: Searching without Math (later in this course)

- Will turn out that we actually can solve the search problem in $O(\log(n))$ using only comparisons (no additions etc.)
- Transform A into a balanced binary search tree
  - 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
  - Recursively compare $c$ to node labels and descend left/right
  - Balanced bin-tree has depth $O(\log(n))$
  - We need at most $\log(n)$ comparisons – and nothing else
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?
• We can exploit additional knowledge about our values
• 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, ...)

Marius Kloft: Alg&DS, Summer Semester 2016
Simple Interpolation Search

- Assume equal distribution – values within A are equally distributed in range [ A[1], A[n] ]
- Best guess for the rank of c
  \[ rank(c) = l + (r - l) \cdot \frac{c - A[l]}{A[r] - A[l]} \]
- Idea: Use m=rank(c) and proceed recursively
- Example: “Xylophon”
Analysis

• On average, Interpolation Search on equally distributed data requires $O(\log(\log(n)))$ comparison (proof: see [OW])

• But: **Worst-case is $O(n)$**
  – If concrete distribution deviates heavily from expected distribution
  – E.g., A contains only names > “Xanthippe”

• Further disadvantage: In each phase, we perform ~4 adds/subs and 2*mults/divs
  – Assume this takes 12 cycles (1 mult/div = 4 cycles)
  – Binsearch requires 2*adds/subs + 1*div ~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
Content of this Lecture

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

- The **median** of a list $A$ is its middle value
  - Sort all values and take the one in the middle
- **Generalization**: $q$-quantiles
  - Sort all values and partition the list into subsequent bins of size $q\% |A|$
  - 25%, 50%, 75% are called quartiles
  - Median = 2-quantile
Selection Problem

- Definition
  *The selection problem is to find the x%-quantile of a set A of unsorted values*

- We can sort A and then access the quantile directly
- Thus, O(n*\log(n)) is easy
- Can we solve this 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 **constant k**, a naïve solution is linear (and optimal)
  – repeat k times
  – go through A and find largest value v;
  – remove v from A;
  – return v
  – Requires $k \times |A| = O(|A|)$ comparisons

• But if $k = x \times |A|$, we are in $O(x \times |A| \times |A|) = O(|A|^2)$
  – We measure complexity in size of the input
  – It is decisive whether k is part of the input or not
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 \( A \) (without sorting \( A \))
- Interesting from a theoretical point-of-view
- Practically, the algorithm is of no importance because the linear factor gets 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 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
1. func int quantile(A array; 2. k, l, r int) { 3. if r≤l then 4. return A[l]; 5. end if; 6. pos := divide( A, 1, r); 7. if (k ≤ pos-1) then 8. return quantile(A, k, l, pos-1); 9. else 10. return quantile(A, k-pos+1, pos, r); 11. end if; 12. }

```
Analysis

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

```plaintext
1. func int quantile(A array; k, l, r int) {
2.    if r ≤ l then
3.        return A[l];
4.    end if;
5.    pos := divide( A, l, r);
6.    if (k ≤ pos-l) then
7.        return quantile(A, k, l, pos-1);
8.    else
9.        return quantile(A, k-pos+1, pos, r);
10.   end if;
11. }
```
Choosing $p$

- Assume we can chose $p$ such that we always continue with at most $q = y\%$ of $A$
  - "$y\%" means: Extend of reduction depends on $n$
- We perform at most $T(n) = T(q*n) + c*n$ comparisons
  - $T(q*n)$ – recursive descent
  - $c*n$ – function “divide”
- $T(n) = T(q*n)+c*n = T(q^{2*n})+q*c*n+c*n = T(q^{2n})+(q+1)*c*n = T(q^{3n})+(q^{2}+q+1)*c*n = ...$

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

- Our algorithm has **worst-case complexity** $\mathcal{O}(n)$ when we manage to always reduce the array by a fraction of its size – no matter, how large the fraction
- This is not an average-case. We must always (not on average) cut some fraction of $A$
- Eh – magic?
- No – follows from the way we defined complexity and what we consider as input
- Many ops are  “hidden” in the linear factor
  - $q=0.9: c*10*n$
  - $q=0.99: c*100*n$
  - $q=0.999: c*1000*n$
Median-of-Median (Assume $|A| = 5^l$)

- 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
  - Use the (approximated) median $v$ of all $v_i$ as pivot element
Complexity

- Run through A in jumps of length 5
- Find each median in constant time
  - Runtime of sorting a list of length 5 does not depend on n
- Call algorithm recursively on all medians
- Since we always reduce the range of values to look at by 80%, this requires $O(n)$ time

![Diagram](image-url)
Why Does this Help?

- We have n/5 first-level medians $v_i$
- $v$ (as median of medians) is smaller than halve of them and greater than the other half (both are n/10 values)
- Each $v_i$ itself is smaller than (and greater than) 2 values
- Thus $v$ is smaller than (and greater than) at least $3\times n/10$ elements
Illustration (source: Wikipedia)

- Median-of-median of a randomly permuted list 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