Searching

Many of the data structures we’re considering are containers — they hold collections of some other data.

Often the data we’re holding is in the form of records with many fields (e.g., student records containing name, student number, address, grades etc.).

In order to find a particular record, we often are given the value for one field — called the key field (e.g., student number to find a student’s record).

As a minimum we need to be able to compare keys to see if they are the same (override operator== on the key type if necessary).

Sequential Search

Assume records stored in a sequential structure (e.g., List).

Algorithm

\[
i = \text{begin} \\
\text{while } i \neq \text{end AND } i \rightarrow \text{key} \neq \text{key we're looking for do} \\
i++ \\
\text{end while} \\
\text{if } i \neq \text{end then} \\
\text{result} = \ast i \\
\text{else} \\
\text{Doesn't exist} \\
\text{end if}
\]

Analysis

How long does sequential search take to execute if the key we’re looking for isn’t in the list?

On a particular machine and compiler, \( a \times \text{list.length} + b \), where \( a \) and \( b \) are positive constants.

We say that this algorithm is linear or \( O(N) \) (“order N”), where \( N \) is the length of the list.

Execution Time

The actual execution time depends on

- The details of the computer
- The compiler and language (and options)
- The “size” of the input
- The value of the input

For the sequential search algorithm above,

- \( a \) and \( b \) relate to the first two bullets,
- \( \text{list.length} \) is a measure of the input size, and
- input value determines if the key is in the list, and if so where.

For large list, the \( b \) term is insignificant, so the time is proportional to the length of the list.
Average Case

What about if the item is in the list?

- If it’s the first item, only one comparison is required.
- If it’s the last item, \( \text{list} \).length comparisons are required.
- If it’s the middle item, \( \frac{\text{list} \).length}{2} \) comparisons are required.

What is the average number of comparisons?

Assume that each position is equally likely, and let \( N = \text{list}.\text{length} \).

\[
T_{\text{avg}} = f(1+2+3+...+N) = \frac{N(N+1)}{2N} = \frac{1}{2}(N + 1)
\]

Binary Search

Algorithm: Find \( k \) in \( L \)

Pre: \( L \) is sorted in non-decreasing order

Post: \( b \) is the index of \( k \) if it is in \( L \).

\( b = 0, e = \text{L}.\text{length} - 1 \)

while \( b < e \) do

  // Invariant: \( L[b] \leq k \leq L[e] \)
  // Variant: \( e - b \)
  \( m = \lfloor \frac{b+e}{2} \rfloor \)
  if \( L[m] < k \) then // look in second half of \( L \)
    \( b = m + 1 \)
  else // look in first half of \( L \)
    \( e = m \)
  end if
end while

result = e
How many comparisons are required by this second version?

Each time through the loop:

- two comparisons are made
- the length of the list is cut in half.

Let \( C(N) \) be the number of comparisons to search a list of length \( N \).

If the element is not found:

\[
C(N) = 2 + C(\lceil \frac{N}{2} \rceil)
\]
\[
= 2 + 2 + C(\lceil \frac{N}{4} \rceil)
\]
\[
= 2 + 2 + 2 + C(\lceil \frac{N}{8} \rceil)
\]
\[
= 2 \lg(N + 1)
\]
\[
\approx 2 \lg N
\]

If the element is found, average case: \( C(N) \approx 2 \lg N - 3 \) (see text).

For large \( N \), the \( \lg N \) term dominates, and the multiplier is significant.

---

**Analysis**

To compare different data structures for solving the same problem we usually consider:

**Time complexity** — the number of computational steps required to solve the problem.

**Space complexity** — the amount of memory required to solve the problem.

- Consider the rate of increase in time/space as the problem size increases (e.g., number of elements in the list).
- Analysis is independent of specific details of the computer etc.

---

**Big-Oh Notation**

Describe the rate of growth of a function:

"\( f(n) \) is in \( O(g(n)) \)" means \( f \) grows slower, or equal to \( g \):

\[
\exists C, \exists N, \forall n, n \geq N \rightarrow f(n) \leq C g(n)
\]

or another way: \( \lim_{n \to \infty} \frac{f(n)}{g(n)} \) is finite.

Technically \( O(g(n)) \) is a set of functions — all those that grow slower or equal to \( g \).

Suppose an algorithm takes \( c \times N^2 + d \times N + e \)

For large \( N \), the time is essentially proportional to \( N^2 \) — we say that the algorithm is quadratic or \( O(N^2) \) ("order \( N \) squared").

Given an \( O(N) \) algorithm and an \( O(N^2) \) algorithm there exists a size of input such that the \( O(N) \) algorithm is the faster for all equal or greater input sizes.

When comparing algorithm performance on large inputs we can ignore

- constants
- all but the dominant terms
- base of logarithms

This means we don’t need to consider the details of the computer (which are subject to change and hard to know).
Examples

\[ f(N) \]  
\[ 2N^2 + N + 1 \]  
\[ O(N^2) \]  
\[ 2N^2 + N + 1 \]  
\[ O(N^3) \]  
\[ kN \log N \]  
\[ O(N \log N) \]  
\[ kN \log N \]  
\[ O(N^2) \]  
\[ k_2N^2 + k_1N + k_0 \]  
\[ O(N^3) \]  

binary search (either version)  
\[ O(\log N) \]

If algorithm A is order \( f(N) \) and algorithm B is not, then (on sufficiently large inputs), A is quicker.

\[ O(1) \subset O(\log N) \subset O(N) \subset O(N \log N) \subset O(N^2) \subset O(2^N) \]

Constant, logarithmic, linear, superlinear, polynomial, exponential

**List Analysis**

\( n \) is number of elements in the list.

**Array**

- Push, pop, retrieve are \( O(1) \) (constant) time (except in overflow case).
- Overflow may be \( O(n) \) time.
- Insert, delete (in the middle) are \( O(n) \) time.
- Space is \( O(n) \), but potentially wasteful.

**Linked**

- Push, pop (front or back) are \( O(1) \) (constant) time.
- Insert, retrieve, delete (in the middle) are \( O(n) \) time.
- Space is \( O(n) \).
- Iteration is \( O(1) \) time.

**Comparison of complexity**

Assume each operation takes 1 µs.

<table>
<thead>
<tr>
<th>( N )</th>
<th>10</th>
<th>50</th>
<th>100</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N \log N )</td>
<td>33 µs</td>
<td>282 µs</td>
<td>664 µs</td>
<td>10 ms</td>
</tr>
<tr>
<td>( N^2 )</td>
<td>100 µs</td>
<td>2.5 ms</td>
<td>10 ms</td>
<td>1 s</td>
</tr>
<tr>
<td>( N^3 )</td>
<td>1 ms</td>
<td>125 ms</td>
<td>1 s</td>
<td>1000 s</td>
</tr>
<tr>
<td>( 1.1N^2 )</td>
<td>2.6 µs</td>
<td>117 µs</td>
<td>13 ms</td>
<td>8 × 10^23 y</td>
</tr>
<tr>
<td>( 2N^2 )</td>
<td>1 ms</td>
<td>3.5 × 10^27 y</td>
<td>4 × 10^42 y</td>
<td>3 × 10^113 y</td>
</tr>
<tr>
<td>( N! )</td>
<td>3 s</td>
<td>10 × 10^6 y</td>
<td>3 × 10^170 y</td>
<td>1.3 × 10^12580 y</td>
</tr>
</tbody>
</table>

**Recursion Analysis**

Use the call tree to determine time and space complexity (assuming that all other parts of the algorithm are constant):

**Time** — depends on the number of vertices in the call tree.

- factorial call tree is line of length \( n \) — time complexity of the algorithm is \( O(n) \).
- hanoi call tree is a complete binary tree (i.e., every non-leaf vertex has two children) with all leaves at the same level — time complexity is \( O(2^n) \), where \( n \) is the number of disks.

**Space** — depends on the depth of the call tree.

- Both trees have depth \( n \), so the space complexity is \( O(n) \).