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 = begin
while i != end AND i−>key != key we're looking for **do**
  i++
end while
if i != end **then**
  result = *i
else
  Doesn't exist
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, $a \times \texttt{list.length} + b$, where a and b are positive constants.

We say that this algorithm is *linear* or $O(\texttt{N})$ ("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,
- `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 comparision is required.
- If it's the last item, `list.length` comparisons are required.
- If it's the middle item, $\frac{\texttt{list.length}}{2}$ comparisons are required.

What is the average number of comparisons?

Assume that each position is equally likely, and let $\mathtt{N} = \texttt{list.length}$.

$$\begin{aligned} T_{avg} &= f(\frac{1+2+3+\ldots+\mathrm{N}}{\mathrm{N}}) \\ &= \frac{\mathrm{N}(\mathrm{N}+1)}{2\mathrm{N}} \\ &= \frac{1}{2}(\mathrm{N}+1) \end{aligned}$$

Binary Search

Algorithm: Find $\mathtt{k}$ in $\mathtt{L}$

Pre: $\mathtt{L}$ is sorted in non-decreasing order
Post: $\mathtt{b}$ is the index of $\mathtt{k}$ if it is in $\mathtt{L}$.

$\mathtt{b} = 0, \mathtt{e} = \mathtt{L.length} - 1$
while $\mathtt{b} < \mathtt{e}$ **do**
  // ***Invariant:*** $\mathtt{L}[\mathtt{b}] \leq \mathtt{k} \leq \mathtt{L}[\mathtt{e}]$
  // ***Variant:*** $\mathtt{e} - \mathtt{b}$
  $\mathtt{m} = \lfloor \frac{\mathtt{b}+\mathtt{e}}{2} \rfloor$
  if $\mathtt{L}[\mathtt{m}] < \mathtt{k}$ **then** // *look in second half of L*
    $\mathtt{b} = \mathtt{m} + 1$
  else // *look in first half of L*
    $\mathtt{e} = \mathtt{m}$
  end if
end while
$\mathtt{result} = \mathtt{e}$

How many comparisons are needed to search using the binary search?

Each time through the loop:

- one comparison is 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.

$$\begin{aligned} C(N) &= 1 + C(\lceil \frac{N}{2} \rceil) \\ &= 1 + 1 + C(\lceil \frac{N}{4} \rceil) \\ &= 1 + 1 + 1 + C(\lceil \frac{N}{8} \rceil) \\ &= 1 + \lg N \end{aligned}$$

Binary Search 2

Algorithm: Find $\mathtt{k}$ in $\mathtt{L}$

$\mathtt{b} = 0, \mathtt{e} = \mathtt{L.length} - 1, \mathtt{found} = \mathbf{false}$
while $\mathbf{not}\ \mathtt{found}\ \mathbf{and}\ \mathtt{b} < \mathtt{e}$ **do**
  // ***Invariant:*** $\mathtt{L}[\mathtt{b}] \leq \mathtt{k} \leq \mathtt{L}[\mathtt{e}]$
  // ***Variant:*** $\mathtt{e} - \mathtt{b}$
  $\mathtt{m} = \lfloor \frac{\mathtt{b}+\mathtt{e}}{2} \rfloor$
  if $\mathtt{L}[\mathtt{m}] == \mathtt{k}$ **then** // *found it*
    $\mathtt{found} = \mathbf{true}$
    $\mathtt{result} = \mathtt{m}$
  else if $\mathtt{L}[\mathtt{m}] < \mathtt{k}$ **then** // *look in second half of L*
    $\mathtt{b} = \mathtt{m} + 1$
  else // *look in first half of L*
    $\mathtt{e} = \mathtt{m} - 1$
  end if
end while

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:
$$\begin{aligned} C(N) &= 2 + C(\lceil \tfrac{N}{2} \rceil) \\ &= 2 + 2 + C(\lceil \tfrac{N}{4} \rceil) \\ &= 2 + 2 + 2 + C(\lceil \tfrac{N}{8} \rceil) \\ &= 2 \lg(N+1) \\ &\approx 2 \lg N \end{aligned}$$

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: $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ is finite.

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

Suppose an algorithm takes $c \times \mathtt{N}^2 + d \times \mathtt{N} + e$

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

Given an $O(\mathtt{N})$ algorithm and an $O(\mathtt{N}^2)$ algorithm there exists a size of input such that the $O(\mathtt{N})$ 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

$f(N)$	Is in
$2N^2 + N + 1$	$O(N^2)$
$2N^2 + N + 1$	$O(N^3)$
$kN \lg N$	$O(N \lg N)$
$kN \lg N$	$O(N^2)$
$k_2N^2 + k_1N + k_0$	$O(N^2)$
sequential search	$O(N)$
binary search (either version)	$O(\lg N)$

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

$$O(1) \subset O(\lg N) \subset O(N) \subset O(N \lg N) \subset O(N^2) \subset O(N^3) \subset O(2^N)$$

Constant, logarithmic, linear, superlinear, polynomial, exponential

Comparison of compexity

Assume each operation takes 1 μs.

N	10	50	100	1000
$N \lg N$	33 μs	282 μs	664 μs	10 ms
N^2	100 μs	2.5 ms	10 ms	1 s
N^3	1 ms	125 ms	1 s	1000 s
N^{100}	3×10^{86} y	2.5×10^{182} y	3×10^{212} y	3×10^{313} y
1.1^N	2.6 μs	117 μs	13 ms	8×10^{53} y
2^N	1 ms	3.5×10^{27} y	4×10^{42} y	3×10^{313} y
$N!$	3 s	10×10^{76} y	3×10^{170} y	1.3×10^{2580} y

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.

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 $O(n)$.