18. Searching a List

Topics:
- Linear Search
- Binary Search
- Measuring Execution Time
- The Divide and Conquer Framework

---

Search

Examples:
- Is this song in that playlist?
- Is this number in that phone book?
- Is this name in that phone book?
- Is this fingerprint in that archive of fingerprints?
- Is this photo in that yearbook?

---

More on Using Phone Books

The Manhattan phone book has 1,000,000+ entries.

How is it possible to locate a name by examining just a tiny, tiny fraction of those entries?

There must be a great search algorithm behind the scenes.

---

LinSearch: The Spec

```python
def LinSearch(x,a):
    """ Returns an int k with the property that a[k]==x is True. If no such k exists, then k=-1. """
    PreC: a is a nonempty list of ints and x is an int.
    """
    for k in range(len(a)):
        if x == a[k]:
            return k
    return -1
```

Could also apply the same idea for searching a list of strings.

---

Linear Search

Walk down the list looking for a match.
def LinSearch(x, a):
    for k in range(len(a)):
        if x == a[k]:
            return k
    return -1

Walk down the list looking for a match.
Linear Search: No Match Case

```
0 1 2 3 4 5 6 7 8 9 10 11
a-> 86 73 43 35 23 45 42 62 15 25 51 35
```

```
def LinSearch(x,a):
    for k in range(len(a)):
        if x == a[k]:
            return k
        return -1
```

Walk down the list looking for a match.

Linear Search: No Match Case

```
0 1 2 3 4 5 6 7 8 9 10 11
a-> 86 73 43 35 23 45 42 62 15 25 51 35
```

```
def LinSearch(x,a):
    for k in range(len(a)):
        if x == a[k]:
            return k
        return -1
```

Return -1 if no match.

Linear Search: While Implementation

```
def LinSearchW(x,a):
    k=0
    while k<len(a) and a[k]!=x:
        k+=1
    return k
```

Binary Search

Now we assume that the list to be searched is sorted from little to big.

```
a = [10,20,40,60,90]
a = ['brown','dog','fox','lazy','quick','the']
```

Back to Using Phone Books

The Ithaca phone book has 10,000+ entries.

The Manhattan phone book has 1,000,000+ entries. But it does not take 100 x longer to look something up. Why?

Key Idea: Repeated Halving

To find Derek Jeter’s number...

```
B = phone book
while (B is longer than 1 page):
    1. P = middle page of B
    2. Let Q be the first name on P
    3. if ‘Jeter” comes before Q:
        Rip away the 2nd half of B
    else:
        Rip away the 1st half of B.
Scan remaining page P line-by-line for ‘Jeter’
```
What Happens to Phone Book Length?

Original: 3000 pages
After 1 rip: 1500 pages
After 2 rips: 750 pages
After 3 rips: 375 pages
After 4 rips: 188 pages
After 5 rips: 94 pages
After 12 rips: 1 page

Binary Search

The idea of repeatedly halving the size of the "search space" is the main idea behind the method of binary search.

An item in a sorted array of length $n$ can be located with approximately $\log_2 n$ comparisons.

$\log_2 8 = 3 \quad \log_2 64 = 7 \quad \log_2 2^k = k$  

What is $\log_2(n)$?

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\text{ceil}(\log_2(n))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
</tr>
<tr>
<td>10000</td>
<td>14</td>
</tr>
<tr>
<td>100000</td>
<td>17</td>
</tr>
<tr>
<td>1000000</td>
<td>20</td>
</tr>
</tbody>
</table>

BinSearch: The Spec

```python
def BinSearch(x,a):
    """ Returns an int $k$ with the property that $a[k]==x$ is True.
    If no such $k$ exists, then $k=={-1}$.
    PreC: $a$ is a nonempty list of ints that is sorted from smallest to largest. $x$ is an int.  """
```

Example: Does this List have an Element With Value Equal to 70?

```text
0 1 2 3 4 5 6 7 8 9 10 11
a=[12 15 33 35 42 45 51 62 73 75 86 98]
L: 0 a[Mid] <= x ????
Mid: 5
R: 11 x: 70 Mid = (L+R)/2
```
The Midpoint Computations

L | R | (L+R)/2
---|---|---
0 | 11 | 5
2 | 6 | 4
1 | 100 | 50

Let's Look For x in a

Let's Look For x in a

Let's Look For x in a

Let's Look For x in a
Let's Look For \( x \) in a

\[
\begin{array}{cccccccccc}
  & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\text{a-} & 12 & 15 & 33 & 35 & 42 & 45 & 51 & 62 & 73 & 75 & 86 & 98 \\
\text{L:} & 12 & a[\text{Mid}] \leq x \\
\text{Mid:} & 6 & \text{No} & \text{So throw away the "right half"} \\
\text{R:} & 11 & x: 70 \\
\end{array}
\]

Let's Look For \( x = 70 \)

\[
\begin{array}{cccccccccc}
  & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\text{a-} & 12 & 15 & 33 & 35 & 42 & 45 & 51 & 62 & 73 & 75 & 86 & 98 \\
\text{L:} & 5 & a[\text{Mid}] \leq x \\
\text{Mid:} & 8 & \text{Revise R and Mid} \\
\text{R:} & 11 & x: 70 \\
\end{array}
\]

Let's Look For \( x = 70 \)

\[
\begin{array}{cccccccccc}
  & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\text{a-} & 12 & 15 & 33 & 35 & 42 & 45 & 51 & 62 & 73 & 75 & 86 & 98 \\
\text{L:} & 5 & a[\text{Mid}] \leq x \\
\text{Mid:} & 6 & \text{Revise R and Mid} \\
\text{R:} & 8 & x: 70 \\
\end{array}
\]

Let's Look For \( x \) in a

\[
\begin{array}{cccccccccc}
  & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\text{a-} & 12 & 15 & 33 & 35 & 42 & 45 & 51 & 62 & 73 & 75 & 86 & 98 \\
\text{L:} & 5 & a[\text{Mid}] \leq x \\
\text{Mid:} & 6 & \text{Yes} & \text{Throw away the Left half} \\
\text{R:} & 8 & x: 70 \\
\end{array}
\]

Let's Look For \( x \) in a

\[
\begin{array}{cccccccccc}
  & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\text{a-} & 12 & 15 & 33 & 35 & 42 & 45 & 51 & 62 & 73 & 75 & 86 & 98 \\
\text{L:} & 6 & a[\text{Mid}] \leq x \\
\text{Mid:} & 7 & \text{Yes} \\
\text{R:} & 8 & x: 70 \\
\end{array}
\]
Let's Look For $x$ in $a$

$0$  $1$  $2$  $3$  $4$  $5$  $6$  $7$  $8$  $9$  $10$  $11$

$a$-$x$  $12$  $15$  $33$  $35$  $42$  $45$  $51$  $62$  $73$  $75$  $86$  $98$

$L$: $6$  $a[Mid] \leq x$

$Mid$: $7$

$R$: $8$  $x$: $70$

Throw away the left half

What We Just Did

$L = 0$

$R = \text{len}(a) - 1$

while $R - L > 1$:

# $a[L] \leq x \leq a[R]$

Mid = $(L+R)/2$

if $x \leq a[\text{mid}]$:

R = Mid

else:

L = Mid

Note that $a[L] \leq x \leq a[R]$ remains True throughout the loop

What is the situation when the loop terminates?

$L = 0$

$R = \text{len}(a) - 1$

while $R - L > 1$:

# $a[L] \leq x \leq a[R]$

Mid = $(L+R)/2$

if $x \leq a[\text{mid}]$:

R = Mid

else:

L = Mid

What We Just Did

$L = 0$

$R = \text{len}(a) - 1$

while $R - L > 1$:

# $a[L] \leq x \leq a[R]$

Mid = $(L+R)/2$

if $x \leq a[\text{mid}]$:

R = Mid

else:

L = Mid

$R-L \leq 1$ implies $R = L+1$

After the Loop Ends

$a[L]$  $a[L+1]$

This is True: $a[L] \leq x \leq a[L+1]$
After the Loop Ends

\[ a[L] \quad a[L+1] \]

if \( x == a[L] \):
    return \( L \)
elif \( x == a[L+1] \):
    return \( L+1 \)
else:
    return -1

Measuring Execution Time

We now have two ways to search a list:

- LinSearch\((x, a)\)
- BinSearch\((x, a)\)

Intuition: BinSearch much faster.

Can we quantify this with a "stop watch"?

The \texttt{timeit} Module

This module can be used to time how long it takes to execute a chunk of code.

Typical chunk = some function of interest.

This is called benchmarking.

Benchmarking

Let's benchmark LinSearch\((x, a)\) and BinSearch\((x, a)\).

Compare how long it takes when \( \text{len}(a) \) equals 1000, 10000, 100000, and 1000000.

Our intuition tells us that as \( \text{len}(a) \) increases, BinSearch will be dramatically faster.

\begin{center}
\begin{tabular}{c c c c}
\hline
n & tBin & tLin & tLinW \\
\hline
1000 & 0.0007 & 0.0064 & 0.0119 \\
10000 & 0.0009 & 0.0668 & 0.1203 \\
100000 & 0.0011 & 0.8296 & 1.2082 \\
1000000 & 0.0015 & 17.7388 & 13.9341 \\
\hline
\end{tabular}
\end{center}

\( tBin = \text{time for BinSearch} \)
\( tLin = \text{time for LinSearch (for loop version)} \)
\( tLinW = \text{time for LinSearch (while-loop version)} \)

BinSearch vs LinSearch

\begin{center}
\begin{tabular}{c c c c}
\hline
n & tLin/tBin \\
\hline
1000 & 9 \\
10000 & 74 \\
100000 & 754 \\
1000000 & 7095 \\
\hline
\end{tabular}
\end{center}

Reporting ratios is more illuminating since we do not really care about the time units in this informal comparison.
Using the `timeit` Module

We show how this module was used to get the results on the previous slides.

Our LinSearch vs BinSearch example is very typical: is one function faster than another?

A Benchmarking Framework

```python
from timeit import *
S = """
Set-up code
"""
B = """
Code to Benchmark
"""
p = 10; m = 100
t = min(Timer(B, setup=S).repeat(p, m))
```

Yes, these are doc strings.

The Set-Up and Bench Codes

```python
from random import randint as randi
from ShowSearch import BinSearch
n = 10000
s = [randi(0, 10*n) for i in range(n)]
s.sort()
x = s[n//2]
k = BinSearch(x, s)
```

The set-up code is run once. It is not timed. It just sets up the code to be timed.

A Benchmarking Framework

```python
from timeit import *
S = """
Set-up code
"""
B = """
Code to Benchmark
"""
p = 10; m = 100
t = min(Timer(B, setup=S).repeat(p, m))
```

An "experiment" consists of running the blue code m times.
The stopwatch will time how long it takes to do one experiment.

A Benchmarking Framework

```python
from timeit import *
S = """
Set-up code
"""
B = """
Code to Benchmark
"""
p = 10; m = 100
t = min(Timer(B, setup=S).repeat(p, m))
```

Larger values necessary if the blue code executes very quickly.

In general, it is best to take the minimum as the most reliable. The benchmark time is assigned to t.

This helps control for other stuff that may be running on your computer.
Why Benchmarking is Important

Confirms/refutes what our intuition might say about efficiency.

Makes us sensitive to the various issues that affect efficiency.

Steers us away from simplistic comparisons of different methods that can be used on the same problem.