17. Searching and Sorting

Topics:

- Linear Search
- Binary Search
- Measuring Execution Time
- The Divide and Conquer Framework
- Merge Sort
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.
Linear Search
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.
    """
Linear Search

```
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

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
Linear Search

Walk down the list looking for a match

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

```bash
a-> [86, 73, 43, 35, 23, 45, 42, 62, 15, 25, 51, 35]

k-> 1

x-> 23
```
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

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
Linear Search

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>a-&gt;</td>
<td></td>
<td>86</td>
<td>73</td>
<td>43</td>
<td>35</td>
<td>23</td>
<td>45</td>
<td>42</td>
<td>62</td>
<td>15</td>
<td>25</td>
<td>51</td>
<td>35</td>
</tr>
</tbody>
</table>

```python
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

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

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

k→ 4

x→ 23

All done
Linear Search: No Match Case

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
```
Linear Search: No Match Case

a→ 86 73 43 35 23 45 42 62 15 25 51 35

x→ 7

```
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

```python
def LinSearchW(x,a):
    k=0
    while k<len(a) and a[k]!=x:
        k+=1
    if k==len(a):
        return -1
    else:
        return k
```
Now we assume that the list to be searched is sorted from little to big.

\[
a = [10, 20, 40, 60, 90]
\]

\[
a = ['brown', 'dog', 'fox', 'lazy', 'quick', 'the']
\]
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 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’
<table>
<thead>
<tr>
<th>Rip</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>3000 pages</td>
</tr>
<tr>
<td>After 1 rip</td>
<td>1500 pages</td>
</tr>
<tr>
<td>After 2 rips</td>
<td>750 pages</td>
</tr>
<tr>
<td>After 3 rips</td>
<td>375 pages</td>
</tr>
<tr>
<td>After 4 rips</td>
<td>188 pages</td>
</tr>
<tr>
<td>After 5 rips</td>
<td>94 pages</td>
</tr>
<tr>
<td>After 12 rips</td>
<td>1 page</td>
</tr>
</tbody>
</table>
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>
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?

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>15</td>
<td>33</td>
<td>35</td>
<td>42</td>
<td>45</td>
<td>51</td>
<td>62</td>
<td>73</td>
<td>75</td>
<td>86</td>
<td>98</td>
</tr>
</tbody>
</table>
Let’s Look For \( x \) in a

\[
\begin{align*}
L: & \quad 0 \\
\text{Mid:} & \quad 5 \\
R: & \quad 11 \\
x: & \quad 70
\end{align*}
\]

\[
a[\text{Mid}] \leq x ????
\]

\[
\text{Mid} = (L+R)/2
\]
## The Midpoint Computations

<table>
<thead>
<tr>
<th>L</th>
<th>R</th>
<th>((L+R)/2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>50</td>
</tr>
</tbody>
</table>
Let's Look For $x$ in $a$

L: 0
Mid: 5
R: 11

$x$: 70

$a[Mid] \leq x$ ????
Let's Look For \( x \) in \( a \)

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\end{array}
\]

\[a\rightarrow [\begin{array}{cccccccccccccccc}
12 & 15 & 33 & 35 & 42 & 45 & 51 & 62 & 73 & 75 & 86 & 98 \\
\end{array}]\]

L: \(0\)  
Mid: \(5\)  
R: \(11\)  

\(x: 70\)  

\(a[\text{Mid}] \leq x\)  
Yes!  
So throw away The “left half”
Let's Look For x in \( a \)

\[ a \rightarrow \begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
12 & 15 & 33 & 35 & 42 & 45 & 51 & 62 & 73 & 75 & 86 & 98
\end{array} \]

**L:** 0  
**Mid:** 5  
**R:** 11  
**x:** 70

\[ a[\text{Mid}] \leq x \]

Yes!  
So throw away The “left half”
Let's Look For x in a

Let's look for 70 in the array a. We start with L = 0, Mid = 5, and R = 11.

1. a[Mid] = 45 ≤ x = 70, so we revise L and Mid.
2. Mid = 5.

Revised array:

```
0 1 2 3 4 5 6 7 8 9 10 11
```
```
12 15 33 35 42 45 51 62 73 75 86 98
```

L: 0
Mid: 5
R: 11

x: 70
Let's Look For x in a

L: 5
Mid: 8
R: 11
x: 70

a[Mid] <= x  ???
Let's Look For x in a

L: 5  
Mid: 8  
R: 11  
x: 70

a[Mid] <= x  
No  
So throw away the “right half”
Let's Look For $x = 70$

```
a-> 12 15 33 35 42 45 51 62 73 75 86 98

L: 5
Mid: 8
R: 11
x: 70
```

Revise $R$ and $Mid$
Let's Look For $x = 70$

Revise $R$ and $Mid$

$a[Mid] \leq x$
Let's Look For x in a

L: 5
Mid: 6
R: 8
x: 70

a[Mid] <= x ????
Let's Look For $x$ in $a$

$a \rightarrow 12 \ 15 \ 33 \ 35 \ 42 \ 45 \ 51 \ 62 \ 73 \ 75 \ 86 \ 98$

$L$: 5  
Mid: 6  
$R$: 8  
$x$: 70

$a[\text{Mid}] \leq x$  
Yes

Throw away the Left half
Let's Look For x in a

Let's look for 70 in the array a.

- **L:** 6
- **Mid:** 7
- **R:** 8

a[7] = 70

a[Mid] = 73

No, 73 is greater than 70.

a[Mid] = 62

Yes, 62 is less than 70.

a[Mid] = 51

Yes, 51 is less than 70.

a[Mid] = 45

No, 45 is greater than 70.

a[Mid] = 35

Yes, 35 is less than 70.

a[Mid] = 15

No, 15 is greater than 70.

a[Mid] = 12

Yes, 12 is less than 70.

The search is complete.

x: 70
Let's Look For x in a

L: 6
Mid: 7
R: 8

x: 70

a[Mid] <= x

Throw away the left half
Let's Look For x in a

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>62</td>
<td>73</td>
<td>75</td>
<td>86</td>
<td>98</td>
</tr>
</tbody>
</table>

L: 6

Mid: 7

R: 8

x: 70

Done! At this point we just compare x with a[L] and a[L+1].
What We Just Did

L = 0
R = len(a) - 1
while R - L > 1:
    # a[L] <= x <= a[R]
    Mid = (L + R) / 2
    if x <= a[mid]:
        R = Mid
    else:
        L = Mid

Note that a[L] <= x <= a[R] remains True throughout the loop.
What We Just Did

\[
L = 0 \\
R = \text{len}(a) - 1 \\
\text{while} \ R - L > 1: \\
\quad \# \ a[L] \leq x \leq a[R] \\
\quad \text{Mid} = (L + R) / 2 \\
\quad \text{if} \ x \leq a[\text{mid}]: \\
\quad \quad R = \text{Mid} \\
\quad \text{else}: \\
\quad \quad L = \text{Mid}
\]

What is the situation when the loop terminates?
What We Just Did

L = 0
R = len(a) - 1
while R-L > 1:
    # a[L] <= x <= a[R]
    Mid = (L+R)/2
    if x <= a[mid]:
        R = Mid
    else:
        L = Mid

R-L<=1  implies R = L+1
After the Loop Ends

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

```python
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 `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.
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.
## BinSearch vs LinSearch

<table>
<thead>
<tr>
<th></th>
<th>tBin</th>
<th>tLin</th>
<th>tLinW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.0007</td>
<td>0.0064</td>
<td>0.0119</td>
</tr>
<tr>
<td>10000</td>
<td>0.0009</td>
<td>0.0668</td>
<td>0.1203</td>
</tr>
<tr>
<td>100000</td>
<td>0.0011</td>
<td>0.8296</td>
<td>1.2082</td>
</tr>
<tr>
<td>1000000</td>
<td>0.0015</td>
<td>17.7388</td>
<td>13.9341</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>tBin</th>
<th>tLin</th>
<th>tLinW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.0007</td>
<td>0.0064</td>
<td>0.0119</td>
</tr>
<tr>
<td>10000</td>
<td>0.0009</td>
<td>0.0668</td>
<td>0.1203</td>
</tr>
<tr>
<td>100000</td>
<td>0.0011</td>
<td>0.8296</td>
<td>1.2082</td>
</tr>
<tr>
<td>1000000</td>
<td>0.0015</td>
<td>17.7388</td>
<td>13.9341</td>
</tr>
</tbody>
</table>

**tBin** = time for BinSearch  
**tLin** = time for LinSearch (for loop version)  
**tLinW** = time for LinSearch (while-loop version)
BinSearch vs LinSearch

<table>
<thead>
<tr>
<th>n</th>
<th>tLin/tBin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>9</td>
</tr>
<tr>
<td>10000</td>
<td>74</td>
</tr>
<tr>
<td>100000</td>
<td>754</td>
</tr>
<tr>
<td>1000000</td>
<td>7095</td>
</tr>
</tbody>
</table>

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 use 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.
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.

Larger values necessary if the blue code executes very quickly.
from timeit import *

S = ""

Set-up code

""
B = ""

Code to Benchmark

""
p = 10; m = 100
t = min(Timer(B, setup=S).repeat(p, m))

Timer returns a length-p list. Each element is the stopwatch time for 1 experiment.

This helps control for other stuff that may be running on your computer.
from timeit import *
S = ""

B = ""

p = 10; m = 100

t = min(Timer(B, setup=S).repeat(p, m))

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.
Binary Search is an example of a “divide and conquer” approach to problem solving.

A method for sorting a list that features this strategy is MergeSort.
**Motivation**

You are asked to sort a list but you have two “helpers”: H1 and H2.

**Idea:**

1. Split the list in half and have each helper sort one of the halves.

2. Then merge the two sorted lists into a single larger list.

This idea can be repeated if H1 has two helpers and H2 has two helpers.
Subdivide the Sorting Task

HEMGGBKQAQFLPLPDRCRCJN
Subdivide Again
And Again
And One Last Time
Now Merge
And Merge Again
And Again
And One Last Time

A B C D E F G H J K L M N P Q R

A B E G H K M Q

C D F J L N P R
Done!
Let's write a function to do this making use of

```python
def Merge(x, y):
    """ Returns a float list that is the merge of sorted lists x and y.

    PreC: x and y are lists of floats that are sorted from small to big. """
```
8 Merges Producing length-2 lists
Handcoding the n = 16 case

\[
\begin{align*}
A_0 &= \text{Merge}(a[0], a[1]) \\
A_1 &= \text{Merge}(a[2], a[3]) \\
A_2 &= \text{Merge}(a[4], a[5]) \\
A_3 &= \text{Merge}(a[6], a[7]) \\
A_4 &= \text{Merge}(a[8], a[9]) \\
A_5 &= \text{Merge}(a[10], a[11]) \\
A_6 &= \text{Merge}(a[12], a[13]) \\
A_7 &= \text{Merge}(a[14], a[15])
\end{align*}
\]
4 Merges Producing Length-4 lists
Handcoding the $n = 16$ case

\[
\begin{align*}
B_0 &= \text{Merge}(A_0, A_1) \\
B_1 &= \text{Merge}(A_2, A_3) \\
B_2 &= \text{Merge}(A_4, A_5) \\
B_3 &= \text{Merge}(A_6, A_7)
\end{align*}
\]
2 Merges Producing Length-8 Lists
Handcoding the n =16 case

C0 = Merge(B0, B1)
C1 = Merge(B2, B3)
1 Merge Producing a Length-16 List

\[
\begin{array}{cccccccccccccccc}
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
A & B & E & G & H & K & M & Q & \bullet & C & D & F & J & L & N & P & R \\
\end{array}
\]
All Done!

D0 = Merge(C0, C1)

For general n, it can be handled using recursion.
def MergeSort(a):
    n = length(a)
    if n==1:
        return a
    else:
        m = n/2
        u0 = list(a[:m])
        u1 = list(a[m:])
        y0 = MergeSort(u0)
        y1 = MergeSort(u1)
        return Merge(y0, y1)