Algorithms for Search and Sort

• Moving beyond correctness!
• Our approach:
  ▪ review programming constructs (while loop) and analysis
  ▪ no built-in methods such as index, insert, sort, etc.
• Today we’ll discuss
  ▪ Linear search
  ▪ Binary search
  ▪ Insertion sort
• More on sorting next lecture
• More on the topic in next course, CS 2110!

Searching for an item in a collection

Is the collection organized? What is the organizing scheme?

Searching in a List

• Search for a target \( x \) in a list \( v \)
• Start at index 0, keep checking until you find it

\[
\begin{array}{llllll}
0 & 1 & \ldots & k & \ldots
\end{array}
\]

\[
\begin{array}{llllll}
v & 12 & 35 & 33 & 15 & 42
\end{array}
\]

\[
\begin{array}{llllll}
x & \text{33}
\end{array}
\]
Searching in a List (Q)

- Search for a target \( x \) in a list \( v \)
- Start at index 0, keep checking until you find it or until no more element to check

Suppose another list is twice as long as \( v \). The expected "effort" required to do a linear search is

<table>
<thead>
<tr>
<th>Option</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. Squared</td>
<td>( n^2 )</td>
</tr>
<tr>
<td>B. Doubled</td>
<td>( 2n )</td>
</tr>
<tr>
<td>C. The same</td>
<td>( n )</td>
</tr>
<tr>
<td>D. Halved</td>
<td>( \frac{n}{2} )</td>
</tr>
<tr>
<td>E. I don't know</td>
<td>( n )</td>
</tr>
</tbody>
</table>

Effort is linearly proportional to list size. Needs \( n \) comparisons for list of size \( n \) (at worst case).

Searching in a List (A)

- Search for a target \( x \) in a list \( v \)
- Start at index 0, keep checking until you find it or until no more element to check

Suppose another list is twice as long as \( v \). The expected "effort" required to do a linear search is

<table>
<thead>
<tr>
<th>Option</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. Squared</td>
<td>( n^2 )</td>
</tr>
<tr>
<td>B. Doubled</td>
<td>( 2n )</td>
</tr>
<tr>
<td>C. The same</td>
<td>( n )</td>
</tr>
<tr>
<td>D. Halved</td>
<td>( \frac{n}{2} )</td>
</tr>
<tr>
<td>E. I don't know</td>
<td>( n )</td>
</tr>
</tbody>
</table>

Effort is linearly proportional to list size. Needs \( n \) comparisons for list of size \( n \) (at worst case).

Search Algorithms

- Search for a target \( x \) in a list \( v \)
- Start at index 0, keep checking until you find it or until no more elements to check

To find the word “Tierartz” in my German dictionary...

- Open to the middle page
- If last word of 1st half comes before Tierartz: Rip* and throw away the 1st half
- Else: Rip* and throw away the 2nd half

* For dramatic effect only--don’t actually rip your dictionary! Just pretend that the part is gone.

Repeated halving of “search window”

- Original: 3000 pages
- After 1 halving: 1500 pages
- After 2 halvings: 750 pages
- After 3 halvings: 375 pages
- After 4 halvings: 188 pages
- After 5 halvings: 94 pages
- After 6 halvings: 47 pages
- After 12 halvings: 1 page

Binary Search

- Repeatedly halve the “search window”
- An item in a sorted list of length \( n \) can be located with just \( \log_2 n \) comparisons.
- “Savings” is significant!
**Binary Search: target \( x = 70 \)**

<table>
<thead>
<tr>
<th>( v )</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></td>
<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>

\( i \) 0, \( j \) 11, \( v[\text{mid}] \) is not \( x \)
\( v[\text{mid}] < x \)
So throw away the left half...

<table>
<thead>
<tr>
<th>( v )</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></td>
<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>

\( i \) 5, \( j \) 11
\( v[\text{mid}] < x \)
So throw away the right half...

<table>
<thead>
<tr>
<th>( v )</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></td>
<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>

\( i \) 6, \( j \) 11, \( v[\text{mid}] \) is not \( x \)
\( v[\text{mid}] < x \)
So throw away the right half...

<table>
<thead>
<tr>
<th>( v )</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></td>
<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>

\( i \) 7, \( j \) 7
\( v[\text{mid}] < x \)
So throw away the left half...

<table>
<thead>
<tr>
<th>( v )</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></td>
<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>

\( i \) 7, \( j \) 7
\( v[\text{mid}] < x \)
So throw away the left half...

<table>
<thead>
<tr>
<th>( v )</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></td>
<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>

\( i \) 8, \( j \) 7
\( i \) is greater than \( j \)
\( \rightarrow \) Not a valid search window

**Binary search is efficient, but we need to sort the vector in the first place so that we can use binary search**

- Many sorting algorithms out there...
- We look at insertion sort now
- Next lecture we’ll look at merge sort and do some analysis
The Insertion Process

- Given a sorted list \( x \), insert a number \( y \) such that the result is sorted.
- Sorted: arranged in ascending (small to big) order.

We’ll call this process a “push down,” as in push a value down until it is in its sorted position.
Sort list \( b \) using Insertion Sort (1)

Need to start with a sorted segment. How do you find one?

\[ b \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \]

See insertion_sort()
Helper functions make clear the algorithm

```python
def swap(b, h, k):
    :
def push_down(b, k):
    while k > 0 and b[k-1] > b[k]:
        swap(b, k-1, k)
    k= k-1
def insertion_sort(b):
    for i in range(1,len(b)):
        push_down(b, i)
```

Algorithm Complexity

- Count the number of comparisons needed
- In the worst case, need $i$ comparisons to push down an element in a sorted segment with $i$ elements.

How much work is a push down?

- This push down takes 2 comparisons

<table>
<thead>
<tr>
<th>push down a “big” value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3 6 9 8</td>
</tr>
</tbody>
</table>

| 2 3 6 8 9 |

<table>
<thead>
<tr>
<th>push down a “small” value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3 6 9 1</td>
</tr>
</tbody>
</table>

| 2 3 6 1 9 |

| 2 3 6 9 |
| 1 6 9 |
| 1 6 9 |
| 1 6 9 |

Algorithm Complexity (Q)

Count (approximately) the number of comparisons needed to sort a list of length $n$

```python
def swap(b, h, k):
    :
def push_down(b, k):
    while k > 0 and b[k-1] > b[k]:
        swap(b, k-1, k)
    k= k-1
def insertion_sort(b):
    for i in range(1,len(b)):
        push_down(b, i)
```

Algorithm Complexity (A)

- Count the number of comparisons needed
- In the worst case, need $i$ comparisons to push down an element in a sorted segment with $i$ elements.
- For a list of length $n$
  - 1st push down: 1 comparison
  - 2nd push down: 2 comparisons (worst case)
  - $\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$, say, $n^2$ for big $n$

Complexity of algorithms discussed

- Linear search: on the order of $n$
- Binary search: on the order of $\log_2 n$
  - Binary search is faster but requires sorted data
- Insertion sort: on the order of $n^2$

For fun, check out this visualization: [https://www.youtube.com/watch?v=xcep5GrC8c](https://www.youtube.com/watch?v=xcep5GrC8c)