Lecture 22:
Algorithms for Sorting and Searching

CS 1110
Introduction to Computing Using Python

[E. Andersen, A. Bracy, D. Gries, L. Lee, S. Marschner, C. Van Loan, W. White]
Announcements

• Remember:
  ▪ When you call a class method, call it via the object
    ▪ (We're seeing a lot of ppl calling it via the class name) the test cases won't catch this, but this is a style/concept issue

```python
c1 = Circle(1,2,3)
c1.draw()
```

NOT

```python
Circle.draw(c1)
```
• 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 \textit{until} you find it
Searching in a List

• 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

Linear search

See search.py
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

- A. Squared
- B. Doubled
- C. The same
- D. Halved
- E. I don’t know

### Linear search

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>...</th>
<th>k</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v$</td>
<td>12</td>
<td>35</td>
<td>33</td>
<td>15</td>
<td>42</td>
</tr>
<tr>
<td>$x$</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Searching in a List (A)

• Search for a target $x$ in a list $v$
• Start at index 0, keep checking \textit{until} you find it or \textit{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

A. Squared
B. Doubled \hspace{1cm} \text{CORRECT}
C. The same
D. Halved
E. I don’t know

Effort is \textit{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*

<table>
<thead>
<tr>
<th>$v$</th>
<th>12</th>
<th>35</th>
<th>33</th>
<th>15</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Linear search

- Search for a target $x$ in a *sorted* list $v$

<table>
<thead>
<tr>
<th>$v$</th>
<th>12</th>
<th>15</th>
<th>33</th>
<th>35</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Binary search

Searching in a sorted list should require less work!
How do you search for a word in a dictionary? (NOT linear search)

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

while dictionary is longer than 1 page:
  open to the middle page
  if last word of 1\textsuperscript{st} half comes before Tierartz:
    Rip\* and throw away the 1\textsuperscript{st} half
  else:
    Rip\* and throw away the 2\textsuperscript{nd} half

* For dramatic effect only--don’t actually rip your dictionary! Just pretend that the part is gone.
## Repeated halving of “search window”

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>3000 pages</td>
<td></td>
</tr>
<tr>
<td>After 1 halving</td>
<td>1500 pages</td>
<td></td>
</tr>
<tr>
<td>After 2 halvings</td>
<td>750 pages</td>
<td></td>
</tr>
<tr>
<td>After 3 halvings</td>
<td>375 pages</td>
<td></td>
</tr>
<tr>
<td>After 4 halvings</td>
<td>188 pages</td>
<td></td>
</tr>
<tr>
<td>After 5 halvings</td>
<td>94 pages</td>
<td></td>
</tr>
<tr>
<td>After 12 halvings</td>
<td>1 page</td>
<td></td>
</tr>
</tbody>
</table>
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!

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\log_2(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
</tr>
<tr>
<td>10000</td>
<td>13</td>
</tr>
<tr>
<td>100000</td>
<td>17</td>
</tr>
</tbody>
</table>
Binary Search: target \( x = 70 \)

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

\( i \) 0 \hspace{2cm} \( v[mid] \) is not \( x \)

\( j \) 11 \hspace{2cm} \( v[mid] < x \)

So throw away the left half...
**Binary Search: target** \( x = 70 \)

- \( v \) is not \( x \)
- \( x < v[\text{mid}] \)
- So throw away the right half...

**Diagram:**
- \( i \) starts at 6
- \( \text{mid} \) is 8
- \( j \) is 11
- The value of \( v[\text{mid}] \) is compared to \( x \)
- If \( x < v[\text{mid}] \), then throw away the right half of the array.
Binary Search: target x = 70

i
mid
j

v[\text{mid}] \text{ is not } x
v[\text{mid}] < x

So throw away the left half...
Binary Search: target $x = 70$

- $v[7]$ is not $x$
- $v[7] < x$
- So throw away the left half...
Binary Search: target \( x = 70 \)

<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>

\( i \) \( \quad \) \( 8 \) \( \quad \) \( \)

\( \text{mid} \) \( \quad \) \( 7 \) \( \quad \) \( \)

\( j \) \( \quad \) \( 7 \) \( \quad \) \( \)

DONE because i is greater than j

→ 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
Push Down

one push down

Push down 8 (b[4]) into the sorted segment b[0..3]

Just swap 8 & 9

The notation b[h..k] means elements at indices h through k of list b, i.e., including k
Push Down

2 3 6 9 8

2 3 6 8 9

sorted

2 3 6 8 9 4

Push down 4 into the sorted segment
Push Down

Compare adjacent components: swap 9 & 4
Push Down

Compare adjacent components:
swap 8 & 4
Push Down

Compare adjacent components: swap 6 & 4
Push Down

one push down

one push down

See `push_down()` in `insertion_sort.py`
Sort list \( b \) using Insertion Sort (1)

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

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

See \texttt{insertion_sort()}

Sort list $b$ using Insertion Sort (2)

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

Length 1 segment is sorted

push_down(b, 1)

See insertion_sort()
Sort list \( b \) using Insertion Sort (3)

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

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

Length 1 segment is sorted

\[
push\_down(b, 1) \quad \text{Then sorted segment has length 2}
\]

\[
push\_down(b, 2)
\]

See \texttt{insertion\_sort()}
Sort list \( b \) using Insertion Sort (4)

Need to start with a \textit{sorted} segment. How do you find one?

\begin{align*}
  \text{Length 1 segment is sorted} \\
  \text{push\_down}(b, 1) \quad \text{Then sorted segment has length 2} \\
  \text{push\_down}(b, 2) \quad \text{Then sorted segment has length 3} \\
  \text{push\_down}(b, 3)
\end{align*}

See \texttt{insertion\_sort()}
Sort list \( b \) using Insertion Sort (rest)

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

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

Length 1 segment is sorted

\[
\text{push\_down}(b, 1) \quad \text{Then sorted segment has length 2}
\]

\[
\text{push\_down}(b, 2) \quad \text{Then sorted segment has length 3}
\]

\[
\text{push\_down}(b, 3) \quad \text{Then sorted segment has length 4}
\]

\[
\text{push\_down}(b, 4) \quad \text{Then sorted segment has length 5}
\]

\[
\text{push\_down}(b, 5) \quad \text{Then entire list is sorted}
\]

For a list of length \( n \), call \text{push\_down} \( n-1 \) times.

See \text{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)

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

Difficult to understand!!
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?

push down a “big” value

push down a “small” value

This push down takes 2 comparisons

This push down takes 4 comparisons.
Worst case scenario: \( n \) comparisons needed to push down into a length \( n \) sorted segment.
Algorithm Complexity (Q)

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

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)

A.  \(~ 1\) comparison
B.  \(~ n\) comparisons
C.  \(~ n^2\) comparisons
D.  \(~ n^3\) comparisons
E.  I don’t know
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 \( n \)
  ▪ 1\(^{st} \) push down: 1 comparison
  ▪ 2\(^{nd} \) push down: 2 comparisons (worst case)
  : 
  ▪ 1+2+...+ (n-1) = n*(n-1)/2 , say, \( n^2 \) for big \( n \)

• For fun, check out this visualization: https://www.youtube.com/watch?v=xxcpvCGrCBe
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 $n^2$