Lecture 25: Algorithms for Sorting and Searching

CS 1110
Introduction to Computing Using Python

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

• Labs 17 & 18 due Friday & Monday, respectively

• Next week’s discussion sections → office hours for A6 and Prelim 2

• Final Exam on May 21st 1:30-4pm. Your assigned exam session (in-person or online) will be given in CMS tomorrow. Submit a “regrade request” in CMS by May 12 if you have a legitimate reason for requesting a change
Algorithms for Search and Sort

• Well known algorithms
  ▪ focus on reviewing programming constructs (while loop) and analysis
  ▪ will not use 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 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

See search.py
Search Algorithms

• 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 elements to check}

\begin{align*}
  v & : 12 \ 35 \ 33 \ 15 \ 42 \\
  x & : 14
\end{align*}

\textbf{Linear search}

• Search for a target $x$ in a \textit{sorted} list $v$

Searching in a sorted list should require less work!

\begin{align*}
  v & : 12 \ 15 \ 33 \ 35 \ 42 \\
  x & : 14
\end{align*}

\textbf{Binary search}
How do you search for a word in a dictionary? (NOT linear search)

To find the word “tanto” in my Spanish dictionary…

while dictionary is longer than 1 page:
    Open to the middle page
    if first entry comes before “tanto”:
        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 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!

<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>
</tbody>
</table>
**Binary Search: target x = 70**

<table>
<thead>
<tr>
<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>v</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
- mid: 5
- j: 11

v[mid] is not x

v[mid] < x

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

$v = [12, 15, 33, 35, 42, 45, 51, 62, 73, 75, 86, 98]$

$i: 6$

$\text{mid}: 8$

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

$x < v[\text{mid}]$

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

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

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

So throw away the left half...

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

$v[\text{mid}] < x$
**Binary Search: target x = 70**

\[ v \begin{bmatrix} 12 & 15 & 33 & 35 & 42 & 45 & 51 & 62 & 73 & 75 & 86 & 98 \end{bmatrix} \]

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

\[
i: \quad 8 \\
\text{mid}: \quad 7 \\
j: \quad 7
\]

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

one push down

Push down 4 into the sorted segment

Compare adjacent components:
- swap 9 & 4
- swap 8 & 4
- swap 6 & 4

DONE! No more swaps.

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

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

\begin{align*}
0 & \quad 1 & \quad 2 & \quad 3 & \quad 4 & \quad 5 \\
\text{push_down}(b, 1) & \text{ Then sorted segment has length 2} \\
\text{push_down}(b, 2) & \text{ Then sorted segment has length 3} \\
\text{push_down}(b, 3) & \text{ Then sorted segment has length 4} \\
\text{push_down}(b, 4) & \text{ Then sorted segment has length 5} \\
\text{push_down}(b, 5) & \text{ Then entire list is sorted}
\end{align*}

For a list of length \( n \), call \texttt{push_down} \( n-1 \) times.

See \texttt{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)
```

Difficult to understand!!

```python
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
```

VS.
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)

```
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)
```

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

A. ~ 1 comparison
B. ~ n comparisons
C. ~ n^2 comparisons
D. ~ n^3 comparisons
E. I don’t know