Searching

Russell Impagliazzo and Miles Jones (ack. Janine Tiefenbruck)

April 1, 2016
Last Time

- We saw a variety of sorting algorithms, including
  - Selection Sort (Min Sort)
  - Bubble Sort
  - Insertion Sort
Today's Agenda

• We will investigate our claim that is easier to find something in a sorted data set than an unsorted one.

• We will consider two different searching algorithms, and see how much faster you can find something when your data is sorted versus unsorted.
The search problem

- Given an array $A[1], A[2], A[3], \ldots, A[n]$ and a target value $x$,
  
  find an index $j$ where $x = A[j]$ 

or determine that no such index exists because $x$ is not in the array; 

Return $j=0$ if no index exists
Example

• Suppose our array is


• If \( x = 1 \), what \( j \) would the search problem find?
  - A) \( j=2 \)
  - B) \( j=4 \)
  - C) \( j=0 \)
  - D) no such \( j \) exists
Searching an unsorted array

- If you needed to find a DVD on this shelf, where the DVDs are in no particular order, how would you proceed?
Linear Search

- Search through the array one index at a time, asking "Is this my target value?" at each position.
- If you answer yes, return the position where you found it.
Linear Search Pseudocode

LinearSearch(A[1, ..., n] an integer array, x an integer)
  i ← 1
  While i ≤ n and x does not equal A[i]
    i ← i + 1
  If i ≤ n, Then location ← i
  Else location ← 0
  Return location
Proof of correctness

• Why is this algorithm correct?
• Easy case: if the algorithm returns some i in the range 1 through n, it must be because \( x = A[i] \)
• Remaining case: ?
Invariant for main loop

- ``While x is not A[i] and  i <= n do: i ++``
- Invariant: If after the loop, i=k, then  ????
Invariant for main loop

- ``While x is not A[i] and i <= n do: i ++``
- Invariant: If after the loop, i=k, then x is not among A[1],...A[k-1]
- Base case: k=1; x is not among the empty set
- Induction step. Assume i=k+1. Then in the previous iteration, i was k. Thus, x is not among the first k-1 elements of the array. In the current iteration, we compared x to A[k]. Since we incremented i, x was also not A[k].
If Linear Search returns 0

- Linear search only returns 0 if $i = n+1$
- Then the invariant says $x$ is not in the first $n$ positions of the array
- But this means $x$ is not in the array, so 0 is the correct answer
How fast is linear search?

- Suppose you have an unsorted array and you are searching for a target value \( x \) that you know is in the array somewhere.

- What position for \( x \) will cause linear search to take the longest amount of time?
  - A) \( x \) is the first element of the array
  - B) \( x \) is the last element of the array
  - C) \( x \) is somewhere in the middle of the array
How fast is linear search?

- Suppose you have an unsorted array and you are searching for a target value $x$ that you know is in the array somewhere.
- What position for $x$ will cause linear search to take the longest amount of time?
  - A) $x$ is the first element of the array
  - B) $x$ is the last element of the array
  - C) $x$ is somewhere in the middle of the array
How fast is linear search?

- Say our array $A$ has $n$ elements and we are searching for $x$.
- The time it takes to find $x$ (or determine it is not present) depends on the number of probes, that is the number of entries we have to retrieve and compare to $x$. 
How many probes?

- Worst-case scenario:

- Best-case scenario:

- On average:
How many probes?

- Worst-case scenario: The target is the last element in the array or not present at all.
  \[ n \text{ probes} \]

- Best-case scenario: The target is the first element in the array.
  \[ 1 \text{ probe} \]

- On average: We'd expect to have to search about half of the array until we find the target.
  \[ \text{about } n/2 \text{ probes} \]
Searching a sorted array

- How would you search through a pile of alphabetized papers to find the one with your name on it?
How does having a sorted array help us?

- Suppose we probe $A$ at position $m$. What do we learn?
  - If $x = A[m]$,
  - If $x < A[m]$,
  - If $x > A[m]$, 

How does having a sorted array help us?

• Suppose we probe $A$ at position $i$. What do we learn?
  
  - If $x = A[m]$, we are done.

  - If $x < A[m]$, then if $x$ is in the array, it is in position $p$ for $p < m$. $m - 1$ positions remain to be checked

  - If $x > A[m]$, then if $x$ is in the array, it is in position $p$ for $p > m$. $n - m$ positions remain to be checked
How do we choose where to probe?

- If our array has 100 elements and we probe at position 5, it is most likely that the target is in the second half, so we'll have to search 95 entries in the worst case.
- How do we make the worst case as favorable as possible?
  **Probe in the middle.**
- If we are searching the subarray $A[i], \ldots, A[j]$, probe at index

\[
\left\lfloor \frac{i + j}{2} \right\rfloor
\]
Binary Search: The Approach

• Probe in the middle of the array.

• Based on what you find there, determine which half to search in next.

• Continue until the target is found or you can be sure the target is not in the array.

• Return the location of the target, or 0 if it's not in the array.
Binary Search Pseudocode

```
BinarySearch(A[1, ..., n] a sorted integer array, x an integer)
    i ← 1
    j ← n
    While i ≤ j
        m ← (i + j)/2
        If x = A[m], Then Return m
        If x > A[m], Then i ← m + 1
        If x < A[m], Then j ← m - 1
    Return 0
```
Binary Search is Telling the Truth

• We must show two things:
  1) If binary search returns some nonzero position $p$, then $A[p] = x$.
  2) If binary search returns 0, then there is no position $p$ for which $A[p] = x$. 
1) If it returns nonzero $p$, then $A[p] = x$.

```
BinarySearch(A[1, ..., n] a sorted integer array, x an integer)
i ← 1
j ← n
While i ≤ j
    m ← (i + j)/2
    If x = A[m], Then Return m
    If x > A[m], Then i ← m + 1
    If x < A[m], Then j ← m - 1
Return 0
```
What is the contrapositive of (2)?

2) If binary search returns 0, then there is no position $p$ for which $A[p] = x$. 
What is the contrapositive?

2) If binary search returns 0, then there is no position \( p \) for which \( A[p] = x \).

**Contrapositive**: If there is a position \( p \) for which \( A[p] = x \), then binary search doesn't return 0.
Loop Invariant

- **Want to show:** If there is a position $p$ for which $A[p] = x$, then binary search doesn't return 0.

- **Loop invariant:** Suppose there is a position $p$ for which $A[p] = x$. Then:
  \[ i \leq p \leq j \]
Invariant: $i \leq p \leq j$

BinarySearch(A[1, ..., n] a sorted integer array, x an integer)
    i ← 1
    j ← n
    While i ≤ j
       m ← (i + j )/2
       If x = A[m], Then Return m
       If x > A[m], Then i ← m + 1
       If x < A[m], Then j ← m - 1
    Return 0
Binary Search Terminates

- With each probe, you reduce the size of the subarray you are searching to at most half of its previous size.

<table>
<thead>
<tr>
<th>Number of probes</th>
<th>Max size of subarray</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>n/2</td>
</tr>
<tr>
<td>2</td>
<td>n/4</td>
</tr>
<tr>
<td>3</td>
<td>n/8</td>
</tr>
<tr>
<td>4</td>
<td>n/16</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>w</td>
<td>?</td>
</tr>
</tbody>
</table>
Binary Search Terminates

- With each probe, you reduce the size of the subarray you are searching to at most half of its previous size.

<table>
<thead>
<tr>
<th>Number of probes</th>
<th>Max size of subarray</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$n/2$</td>
</tr>
<tr>
<td>2</td>
<td>$n/4$</td>
</tr>
<tr>
<td>3</td>
<td>$n/8$</td>
</tr>
<tr>
<td>4</td>
<td>$n/16$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$w$</td>
<td>$n/2^w$</td>
</tr>
</tbody>
</table>
Binary Search Terminates

- How many probes \( w \) do we need to make sure our search has terminated? That is, what value of \( w \) is sure to make the subarray size less than 1?

\[
\frac{n}{2^w} < 1 \\
n < 2^w \\
\log(n) < w
\]

- The smallest integer \( w \) that is sure to be greater than \( \log(n) \) is

\[
w = \lceil \log(n) \rceil + 1
\]
Comparison with Linear Search

- Therefore, we have shown that it takes at most

$$\lfloor \log(n) \rfloor + 1$$

probes for binary search to terminate. This is the worst case.

- By comparison, linear search takes as many as $n$ probes. Even in the average case, linear search takes about $n/2$ probes.
The cost of binary search

• While binary search is faster than linear search, it depends upon your array being sorted.

• What if you have an unsorted array you need to search? Should you sort it first so you can use the better search?
The cost of binary search

• There's a tradeoff between the cost of sorting an array and the benefit of being able to search faster.

• If you need to search the same array many times, it becomes more worthwhile to invest the initial time in sorting your array.
Next Time

- We'll explore more what we mean by the costs associated with algorithms.

- We'll consider various factors that go into deciding which algorithms are better than others, in particular, how they scale with input size.

- These are widely applicable ideas that are important for all types of algorithms, not just sorting and searching.