Lower Bound Analysis

\[ \log n! \leq \log n^2 = n \log n. \]
\[ \log n! \geq \log \left( \frac{n}{2} \right)^{\frac{n}{2}} \geq \frac{1}{2} (n \log n - n). \]

- So, \(\log n! = \Theta(n \log n)\).
- Using the decision tree model, what is the average depth of a node?
- This is also \(\Theta(\log n!)\).

A Search Model (1)

**Problem:**
Given:
- A list \(L\) of \(n\) elements
- A search key \(X\)

Solve: Identify one element in \(L\) which has key value \(X\), if any exist.

**Model:**
- The key values for elements in \(L\) are unique.
- One comparison determines \(<\), \(\leq\), \(\geq\).
- Comparison is our only way to find ordering information.
- Every comparison costs the same.

A Search Model (2)

**Goal:** Solve the problem using the minimum number of comparisons.

- Cost model: Number of comparisons.
- (Implication) Access to every item in \(L\) costs the same (array).

Is this a reasonable model and goal?

Linear Search

**General algorithm strategy:** Reduce the problem.
- Compare \(X\) to the first element.
- If not done, then solve the problem for \(n-1\) elements.

\[
\text{Position} = \begin{cases} 
1 & n = 1 \\
\text{position}(L, \text{lower}+1, \text{upper}, X); & n > 1 
\end{cases}
\]

What equation represents the worst case cost?
**Lower Bound on Problem**

**Theorem:** Lower bound (in the worst case) for the problem is \( n \) comparisons.

**Proof:** By contradiction.
- Assume an algorithm \( A \) exists that requires only \( n - 1 \) (or less) comparisons of \( X \) with elements of \( L \).
- Since there are \( n \) elements of \( L \), \( A \) must have avoided comparing \( X \) with \( L[i] \) for some value \( i \).
- We can feed the algorithm an input with \( X \) in position \( i \).
- Such an input is legal in our model, so the algorithm is incorrect.

Is this proof correct?

---

**Fixing the Proof (1)**

Error #1: An algorithm need not consistently skip position \( i \).

Fix:
- On any given run of the algorithm, *some* element \( i \) gets skipped.
- It is possible that \( X \) is in position \( i \) at that time.

---

**Fixing the Proof (2)**

Error #2: Must allow comparisons between elements of \( L \).

Fix:
- Include the ability to “preprocess” \( L \).
- View \( L \) as initially consisting of \( n \) “pieces.”
- A comparison can join two pieces (without involving \( X \)).
- The total of these comparisons is \( k \).
- We must have at least \( n - k \) pieces.
- A comparison of \( X \) against a piece can reject the whole piece.
- This requires \( n - k \) comparisons.
- The total is still at least \( n \) comparisons.

---

**Average Cost**

How many comparisons does linear search do on average?

We must know the probability of occurrence for each possible input.

(Must \( X \) be in \( L \)?)

Ignore everything except the position of \( X \) in \( L \). Why?

What are the \( n + 1 \) events?

\[
P(X \notin L) = 1 - \sum_{i=1}^{n} P(X = L[i]).
\]
Average Cost Equation

Let $k_i = i$ be the number of comparisons when $X = L[i]$. Let $k_0 = n$ be the number of comparisons when $X \notin L$.

Let $p_i$ be the probability that $X = L[i]$. Let $p_0$ be the probability that $X \notin L[i]$ for any $i$.

$$f(n) = k_0p_0 + \sum_{i=1}^{n} k_ip_i$$

$$= np_0 + \sum_{i=1}^{n} ip_i$$

What happens to the equation if we assume all $p_i$'s are equal (except $p_0$)?

Computation

$$f(n) = p_0n + \sum_{i=1}^{n} ip_i$$

$$= p_0n + p\sum_{i=1}^{n} i$$

$$= p_0n + p\frac{n(n+1)}{2}$$

$$= p_0n + p\frac{1 - p_0}{n} \frac{n(n+1)}{2}$$

$$= n + p_0(n-1)$$

Depending on the value of $p_0$, $\frac{n+1}{2} \leq f(n) \leq n$.

Problems with Average Cost

- Average cost is usually harder to determine than worst cost.
- We really need also to know the variance around the average.
- Our computation is only as good as our knowledge (guess) on distribution.

Sorted List

Change the model: Assume that the elements are in ascending order.

Is linear search still optimal? Why not?

Optimization: Use linear search, but test if the element is greater than $X$. Why?

Observation: If we look at $L[5]$ and find that $X$ is bigger, then we rule out $L[1]$ to $L[4]$ as well.

More is Better: If we look at $L[n]$ and find that $X$ is bigger, then we know in one test that $X$ is not in $L$. Great!

- What is wrong here?
Jump Search

Algorithm:
- From the beginning of the array, start making jumps of size \( k \), checking \( L[k] \) then \( L[2k] \), and so on.
- So long as \( X \) is greater, keep jumping by \( k \).
- If \( X \) is less, then use linear search on the last sublist of \( k \) elements.

This is called Jump Search.

What is the right amount to jump?

Analysis of Jump Search

- If \( mk \leq n < (m + 1)k \), then the total cost is at most \( m + k - 1 \) 3-way comparisons.
  \[
  f(n, k) = m + k - 1 = \left\lfloor \frac{n}{k} \right\rfloor + k - 1.
  \]
- What should \( k \) be?
  \[
  \min_{1 \leq k \leq n} \left\{ \left\lfloor \frac{n}{k} \right\rfloor + k - 1 \right\}
  \]
- Take the derivative and solve for \( f'(x) = 0 \) to find the minimum.
- This is a minimum when \( k = \sqrt{n} \).
- What is the worst case cost?
  - Roughly \( 2\sqrt{n} \).

Lessons

We want to balance the work done while selecting a sublist with the work done while searching a sublist. In general, make subproblems of equal effort.

This is an example of divide and conquer

What if we extend this to three levels?
- We'd jump to get a sublist, then jump to get a sub-sublist, then do sequential search.
- While it might make sense to do a two-level algorithm (like jump search), it almost never makes sense to do a three-level algorithm.
- Instead, we resort to recursion

Binary Search

```c
int binary(int K, int* array, int left, int right) {
    // Return position of element (if any) with value K
    int l = left-1;
    int r = right+1;  // l and r beyond array bounds
    while (l+1 != r) {  // Stop when l and r meet
        int i = (l+r)/2;  // Middle of remaining subarray
        if (K < array[i]) r = i;  // In left half
        else if (K == array[i]) return i;  // Found it
        else l = i;  // In right half
    }
    return UNSUCCESSFUL;  // Search value not in array
}
```

\[
 f(n) = \begin{cases} 
 1 & n = 1 \\
 f(\lfloor n/2 \rfloor) + 1 & n > 1 
\end{cases}
\]

m is number of big steps, \( k \) is size of big step.
Lower Bound (for Problem Worst Case)

How does $n$ compare to $\sqrt{n}$ compare to $\log n$?

Can we do better?

Model an algorithm for the problem using a decision tree.
- Consider only comparisons with $X$.
- Branch depending on the result of comparing $X$ with $L[i]$.
- There must be at least $n$ leaf nodes in the tree. (Why?)
- Some path must be at least $\log n$ deep. (Why?)

Thus, binary search has optimal worst cost under this model.

Average Cost of Binary Search (1)

An estimate given these assumptions:
- $X$ is in $L$.
- $X$ is equally likely to be in any position.
- $n = 2^k$ for some non-negative integer $k$.

Cost?
- One chance to hit in one probe.
- Two chances to hit in two probes.
- $2^{k-1}$ to hit in $i$ probes.
- $i \leq k$.

Average cost is $\log n - 1$.

Average Cost Lower Bound

- Use decision trees again.
- **Total Path Length**: Sum of the level for each node.
- The cost of an outcome is the level of the corresponding node plus 1.
- The average cost of the algorithm is the average cost of the outcomes (total path length / $n$).
- What is the tree with the least average depth?
- This is equivalent to the tree that corresponds to binary search.
- Thus, binary search is optimal.

Interpolation Search

(Also known as Dictionary Search) Search $L$ at a position that is appropriate to the value of $X$.

$$p = \frac{X - L[1]}{L[n] - L[1]}$$

Repeat as necessary to recalculate $p$ for future searches.
**Quadratic Binary Search**

This is easier to analyze:
- Compute $p$ and examine $L[\lfloor pn \rfloor]$.
- If $X < L[\lfloor pn \rfloor]$ then sequentially probe $L[\lfloor pn - i \sqrt{n} \rfloor]$, $i = 1, 2, 3, ...$
  until we reach a value less than or equal to $X$.
- Similar for $X > L[\lfloor pn \rfloor]$.
- We are now within $\sqrt{n}$ positions of $X$.
- ASSUME (for now) that this takes a constant number of comparisons.
- Now we have a sublist of size $\sqrt{n}$.
- Repeat the process recursively.
- What is the cost?

This assumes uniformly distributed data.

**Useful fact (Čebyshev’s Inequality):**

The probability that we need probe $p$ times is $P_i = \frac{p(1-p) \sqrt{n}}{(i-2)^2}$.

We require at least two probes to set the bounds, so cost is:

$$2 + \sum_{i=3}^{\sqrt{n}} P_i$$

Useful fact (Čebyshev’s Inequality):

The probability that we need probe $i$ times is $P_i = \frac{p(1-p) \sqrt{n}}{(i-2)^2}$.

Since $p(1-p) \leq \frac{1}{4}$.

This assumes uniformly distributed data.
QBS Probe Count (4)

Final result:

\[ 2 + \sum_{i=3}^{\sqrt{n}} \frac{1}{4(i-2)^2} \approx 2.4112 \]

Is this better than binary search?

What happened to our proof that binary search is optimal?

Comparison (1)

Let's compare \( \log \log n \) to \( \log n \).

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \log n )</th>
<th>( \log \log n )</th>
<th>Diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>256</td>
<td>8</td>
<td>3</td>
<td>2.7</td>
</tr>
<tr>
<td>64K</td>
<td>16</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2^{32}</td>
<td>32</td>
<td>5</td>
<td>6.4</td>
</tr>
</tbody>
</table>

Now look at the actual comparisons used.

- Binary search \( \approx \log n - 1 \)
- Interpolation search \( \approx 2.4 \log \log n \)

Comparison (2)

Not done yet! This is only a count of comparisons!

- Which is more expensive: calculating the midpoint or calculating the interpolation point?

Which algorithm is dependent on good behavior by the input?