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

What is the right amount to jump?

Algorithm:

- Check every k'th element ($L[k]$, $L[2k]$, ...).
- If X is greater, then go on.
- If X is less, then use linear search on the k elements.

This is called Jump Search.

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

```
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
    if (K == array[i]) return i; // Found it
    if (K > array[i]) l = i;     // In right half
  }
  return UNSUCCESSFUL; // Search value not in array
}
```

Worst Case for Binary Search

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

Since $n/2 \geq \lfloor n/2 \rfloor$, and since $f(n)$ is assumed to be non-decreasing (why?), we can use

$$f(n) = f(n/2) + 1.$$

Alternatively, assume n is a power of 2.

Expand the recurrence:

$$\begin{array}{rcl} f(n) & = & f(n/2) + 1 \\ & = & \{f(n/4) + 1\} + 1 \\ & = & \{\{f(n/8) + 1\} + 1\} + 1 \end{array}$$

Collapse to

$$f(n) = f(n/2^i) + i = \log n + 1$$

Now, prove it with induction.

$$\begin{array}{rcl} f(n/2) + 1 & = & (\log(n/2) + 1) + 1 \\ & = & (\log n - 1 + 1) + 1 \\ & = & \log n + 1 = f(n). \end{array}$$

Lower Bound

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 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

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^{i-1} to hit in i probes.
- $i \leq k$.

What is the equation?

Average Cost (cont.)

$$\frac{1 \times 1 + 2 \times 2 + 3 \times 4 + ... + \log n 2^{\log n - 1}}{n}$$

$$= \frac{1}{n} \sum_{i=1}^{\log n} i2^{i-1}$$

$$\begin{aligned}
\sum_{i=1}^{k} i2^{i-1} &= \sum_{i=0}^{k-1} (i+1)2^i \\
&= \sum_{i=0}^{k-1} i2^i + \sum_{i=0}^{k-1} 2^i \\
&= 2\sum_{i=0}^{k-1} i2^{i-1} + 2^k - 1 \\
&= 2\sum_{i=1}^{k} i2^{i-1} - k2^k + 2^k - 1
\end{aligned}$$

Now what? Subtract from the original!

$$\sum_{i=1}^{k} i2^{i-1} = k2^k - 2^k + 1 = (k-1)2^k + 1.$$

Result

$$\begin{aligned}
\frac{1}{n}\sum_{i=1}^{\log n} i2^{i-1} &= \frac{(\log n - 1)2^{\log n} + 1}{n} \\
&= \frac{n(\log n - 1) + 1}{n} \\
&\approx \log n - 1
\end{aligned}$$

So the average cost is only about one or two comparisons less than the worst cost.

If we want to relax the assumption that $n = 2^k$, we get:

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ \frac{\lceil \frac{n}{2} \rceil - 1}{n} f(\lceil \frac{n}{2} \rceil - 1) + \frac{1}{n}0 + & \\ \frac{\lfloor \frac{n}{2} \rfloor}{n} f(\lfloor \frac{n}{2} \rfloor) + 1 & n > 1 \end{cases}$$

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.