A recursive binary search algorithm for partially sorted data

James A. Riechel

January 20, 2009

**Abstract**

A recursive binary search algorithm is presented which is appropriate for searching partially sorted data. In the case of completely random input, the algorithm requires on the average of $O(an+b\lg n+c)$ operations to find a specified key in a list of length *n*. In the case of partially sorted data, on average $O(an+b\lg n+c)$ operations are also required to find a specified key in a list of length *n*. In the case of completely sorted data, on average only $O(a\lg n+b)$ operations are necessary to find a specified key in a list of length *n*, assuming the specified search key can be found in our list, the same average case running time as ordinary binary search on completely sorted data.

## 1 Algorithm

To recursively search for a specified key, *key*, in a list of length *n*, $data[0],...,data[n-1]$, a call to $rbspsd(data,key,0,n-1)$ is made. The author considers the recursive binary search algorithm for partially sorted data, $rbspsd()$, to be classified, but the C or C++ code for $rbspsd()$ can be found in Figure 1.

```
int rbspsd(int* data, int key, int n1, int n2) {

   if (n2 < n1) return –1;

   int m = ((int) ((n2 – n1 + 1) / 2)) + n1;

   if (key == data[m]) return m;

   int result;

   if (key < data[m]) {
      if ((result = rbspsd(data, key, n1, m – 1)) != -1) return result;
      if ((result = rbspsd(data, key, m + 1, n2)) != -1) return result;
   }
   else { // key > data[m]
      if ((result = rbspsd(data, key, m + 1, n2)) != -1) return result;
      if ((result = rbspsd(data, key, n1, m – 1)) != -1) return result;
   }

   return -1;
}
```

Figure 1: Recursive binary search for partially sorted data, *rbspsd()*.

## 2 Time complexity

The time complexity of the $rbspsd()$ algorithm depends on the data distribution of the input: $data[0],...,...data[n-1]$, for a list of length *n*, and it depends on whether or not the specified search key can be found in our list. We consider the cases of completely random input before and after partial sorting, and the case of completely sorted input.

### 2.1 Completely random input

Assume the input to the algorithm is completely random data, $data[0],...,data[n-1]$, for a list of length *n*, and assume we are searching for a specified key, *key*. Now define

$$m = \left\lfloor \frac{n_2 - n_1 + 1}{2} \right\rfloor + n_1 = n_2 + 1 - \left\lfloor \frac{n_2 - n_1 + 1}{2} \right\rfloor$$

Since the input is completely random:

$$P(key = data[m]) = \frac{1}{n_2 - n_1 + 1}$$

$$P(key < data[m]) = \frac{\left\lfloor \frac{n_2 - n_1 + 1}{2} \right\rfloor}{n_2 - n_1 + 1}$$

$$P(key > data[m]) = \frac{\left\lfloor \frac{n_2 - n_1 + 1}{2} \right\rfloor - 1}{n_2 - n_1 + 1}$$

Now define $n = n_2 - n_1 + 1$. Then the running time of the algorithm, $R(n)$, is given by:

$$R(n < 1) = c$$

$$R(n \geq 1) = P(key = data[m])c_0 + P(key < data[m])\left[ c_1 R\left( \left\lfloor \frac{n}{2} \right\rfloor - 1 \right) + c_2 R\left( \left\lfloor \frac{n}{2} \right\rfloor \right) \right] +$$
$$P(key > data[m])\left[ c_3 R\left( \left\lfloor \frac{n}{2} \right\rfloor \right) \right] + c_4 R\left( \left\lfloor \frac{n}{2} \right\rfloor - 1 \right)$$

Simplifying:

$$R(n < 1) = k$$

$$R(n \geq 1) = k_0 + k_1 R\left(\left\lfloor \frac{n}{2} \right\rfloor - 1\right) + k_2 R\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$$

$R(n)$ is a recursive formula which, in the worst case, is linear. In other words, in the worst case, $O(R(n)) = O(n)$ for completely random data, or when $key \notin data$, since we must examine every element in the data to compare it to the specified search key, *key*. On average, the time complexity for completely random data is given by $O(R(n)) = O(an + b\lg n + c)$

### 2.2 Completely sorted input

The beauty of recursive binary search for partially sorted data is that it has the same time complexity as regular binary search if the input to the algorithm is completely sorted data. If $R(n)$ is the running time of the recursive binary search algorithm for partially sorted data, and if the input to the algorithm is completely sorted data, and if we assume $key \in data$, then the average case running time of the algorithm is given by $O(R(n)) = O(a\lg n + b)$.

However, if $key \notin data$, then *O(R(n)) = O(n).*

### 2.3 Partially sorted data

We partially sort the completely random input using the partial sorting algorithm given in my last article, “A linear time algorithm for partial sorting” (2009).

As in section 2.1, the running time of the algorithm, $R(n)$, is given by:

$$R(n < 1) = k$$

$$R(n \geq 1) = k_0 + k_1 R\left(\left\lfloor \frac{n}{2} \right\rfloor - 1\right) + k_2 R\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$$

As before, in the worst case, $O(R(n)) = O(n)$, and in the average case, $O(R(n)) = O(an + b\lg n + c)$.

## 3 Summary / Conclusion

Figure 2 summarizes the time complexity of the recursive binary search algorithm for partially sorted data. If $key \notin data$, then regardless of whether the input is completely random, partially sorted, or completely sorted, the algorithm is linear, or requires on the order of $O(n)$ operations to complete, for a list of length *n*. The advantage of using the $rbspsd()$ algorithm is seen when $key \in data$, and the input is either partially sorted, or completely sorted. When $key \in data$, and the input is completely sorted, there is no

difference between $rbspsd()$ and regular binary search on completely sorted data. When $key \in data$, and the input is partially sorted, the $rbspsd()$ algorithm performs well in comparison to completely random input, and the benefit of partially sorting the completely random input before searching is clear and evident.

| | Input | Average case | Worst case |
|---|---|---|---|
| $key \in data$ | Completely random | $O(an + b\lg n + c)$ $a > b$ or $a \gg b$ | $O(n)$ |
| | Partially sorted | $O(an + b\lg n + c)$ $b > a$ *or* $b \gg a$ | $O(n)$ |
| | Completely sorted | $O(a\lg n + b)$ | $O(a\lg n + b)$ |
| $key \notin data$ | Completely random | $O(n)$ | $O(n)$ |
| | Partially sorted | $O(n)$ | $O(n)$ |
| | Completely sorted | $O(n)$ | $O(n)$ |

Figure 2: Time complexity of $rbspsd()$