Programming Tips & Tricks

Paul L. Fackler, North Carolina State University
paul_fackler@ncsu.edu

Search Algorithms

An operation that appears in many situations involves a sorted $n+1$-vector of values x and an input value X. The goal is to find k such that $x_k \le X < x_{k+1}$. This is the bin membership problem where a bin k is composed of the values $[x_k, x_{k+1})$. Here we are interpreting the values of x as bin edges and the $n+1$ edges define n bins.

For some purposes it is useful to make the alternative assumption that there are n values of x that are interpreted as bin centers and we seek k such that $\frac{x_k+x_{k-1}}{2} \le X < \frac{x_k+x_{k+1}}{2}$ but this is simply shifting the values of x and making appropriate endpoint adjustments. Thus define the edges to be $\tilde{x}_k = \frac{x_k+x_{k-1}}{2}$ for $k \in \{2, \dots, n\}$ and either set $\tilde{x}_1 = -\infty$ and $\tilde{x}_{n+1} = \infty$ or set them to boundaries beyond which bin membership is undefined.

An important consideration is how to define what happens if X is outside the interval $[x_1, x_{n+1})$. The answer to this question depends on the purpose of the exercise and on whether such behavior ever happens. We'll explore some alternatives in a latter article that focusses on the uses of search algorithms. For the time being we'll assign $X < x_1$ to bin 0 and $X \ge x_{n+1}$ to bin $n+1$ and assume that specific applications will either alter the basic algorithm or handle these values as needed. It is also assumed that there are no repeated values in x; the algorithms described here may need to be modified to handle repeated values correctly for a specified application.

Although there are many variations on search algorithms it is probably fair to say that there are 3 main alternatives, linear $O(n)$, binary $O(\log_2 n)$ and constant time $O(1)$. An $O(n)$ linear search simply increments k until as $X \ge x_{k+1}$. Here is a MATLAB implementation:

```
function k=findkLS(X,x)
if X <  x(1);   k = 0;        return; end
if X >= x(end); k = length(x); return; end
k=1;
while X >= x(k+1), k=k+1; end
```

I've put explicit code to handle the boundaries; this could be easily changed to handle other alternatives.

A binary search is generally faster, $O(\log_2 n)$, because it eliminates about half of the remaining possibilities at each step. It first determines whether X is in the top half of x or the bottom half. At each subsequent iteration it cuts the search interval in half (approximately). Here is a MATLAB implementation:

```
function k=findkBS(X,x)
if X <  x(1);   k = 0;        return; end
if X >= x(end); k = length(x); return; end
a = 1;
b = length(x);
k = floor((a+b)/2);
while k>a
 if X >= x(k), a = k;
 else         b = k;
 end
 k = floor((a+b)/2);
end
```

Before examining whether we can improve on this performance note that it we want to repeatedly do searches with the same x vector we can define a function that that performs the search**:**

```
searchx = @(X) findkBS(X,x);
```

To see how we can get a constant time search we first examine the special case in which the values of x are all evenly spaced, which implies that $x_k = x_0 + k\Delta$. It is straightforward, therefore, to define a function that map values of X into values of k:

$$k = I(X) = \text{floor}((X - x_0)/\Delta)$$

One problem with this arises when floating point operations are used because $(x_k - x_0)/\Delta$ might be evaluated as sight less than k in which case this function would return $k - 1$. If this is a problem we could use instead

$$k = I(X) = \text{floor}\left(\frac{X - x_0}{\Delta} - \epsilon\right)$$

where $\epsilon = \min(0, \min((X - x_0)/\Delta))$. It is possible that values of X slightly less than x_k will return k but this is generally less problematic than having X exactly equal to x_k returning $k - 1$; in any case it is easily tested and corrected.

A similar idea can also be applied to values of x that are unevenly spaced. Consider a function of the form

$$j = I(X) = \text{floor}(a + bX)$$

that guarantees that the value of j is unique when I is evaluated at the values x_k. To ensure this it must be true that $b(x_{k+1} - x_k) \geq 1$ for every $k = 1, \ldots, n$. This is ensured by setting $b = 1/\min_k(x_{k+1} - x_k)$. Setting $a = 1 - bx_1$ ensures that the lowest value returned by I is 1. The highest value returned is $c = \text{floor}(a + bx_{n+1})$. Let I be a c-element vector and set $I\big(\text{floor}(a + bx_k)\big) = k$. Other values of I are filled in with the previous value:

for j=1:c, if I(j)==0, I(j)=I(j-1); end; end

Notice that, for evenly spaced x we get the result that $c = n + 1$ which leads to same values for a and b obtained above. There is, therefore, no need to test for even spacing when using this algorithm.

With a, b and I we can now obtain the correct position for any X by setting

$$k = I\big(\text{floor}(a + bX)\big)$$

which is a constant time algorithm. This is not quite correct however. To see why define

$$\epsilon_k = (a + bx_k) - \text{floor}(a + bx_k)$$

Unlike the situation when the x values are evenly spaced, $a + bx_k$ need not be integer valued so value of X slightly less than x_k may be incorrectly assigned. Specifically, any

$$X \in \left[x_k - \frac{\epsilon_k - a}{b}, x_k\right)$$

will be incorrectly associated with index k. To correct this we need to put a check in to the effect that

if x(k) > X, k = k-1; end

One problem that arises with this method is that the value of c can become very large if there are x_k values that are very close to one another. This can be addressed by specifying a maximum value for c. In this case it is not enough to check the previous k but must go backwards until the appropriate k is found. This is accomplished by simply replacing **if x(k) > X** with **while x(k) > X.** Ideally only a few iterations are needed in most cases. In the code below I've set the maximum value of c to be $10(n + 1)$. Setting this too high can result in a degradation of performance, presumably from slowdowns due to memory access. In fact setting $c = n + 1$ does not result in a significant change in the speed of the computations.

The basic algorithm implemented in the following MATLAB code. I've made small adjustments to a and b to ensure that roundoff error does not affect the correct operation of the algorithm.

```
function If=getfindkfunc(x)
 n1=length(x);
 cmax = 10*n1;
 b=(1+2*eps)/min(diff(x));
 b=min((cmax-1)/(x(n1)-x(1)), b);
 a=(1+2*eps)-b*x(1);
 c=floor(a+b*x(end));
 I=zeros(c,1);
 j=floor(a+b*x);
 I(j)=(1:n1)';
 for j=2:c
  if I(j)==0, I(j)=I(j-1); end
 end
 If = @(X) findkCT(X,x,a,b,c,I);
```

This code is a setup function that creates a, b, c and I. It returns a function that can be used to perform the actual search using the following function:

```
function k=findkCT(X,x,a,b,c,I)
 j=min(c,floor(a+b*X));
 if X < x(1)
  k=0;
 else
  k=I(j);
  while x(k) > X, k=k-1; end
 end
```

This function does not depend on n and thus provides an $O(1)$ operation.

All of these functions can be altered to handle a vector of X values by looping over the X values. The Table 1 compares the time used by the algorithms with different values of n and looping over 1 million values of X. The linear search (LS) algorithm clearly exhibits $O(n)$ behavior with a doubling of n leading to (approximately) a doubling of the time taken. The right most column shows the ratio of the time used by the binomial search (BS) to $\log_2 n$, demonstrating that the ration is approximately constant. The constant time (CT) algorithm does exhibit constant time behavior.

A couple other points are evident in Table 1. First, even for small n the vectorized CT algorithm is the clear choice but the LS algorithm is also a good choice because each comparison takes very little time. For $n < 20$ the LS algorithm beats the BS algorithm. This is similar to the advantage of insertion sort, which exhibits $O(n^2)$ behavior, over various $O(n\ log_2 n)$ sorting algorithms, such as merge and heap sorts.

Table 1. Timing comparisons of different search algorithms to search 1,000,000 values

n	LS	BS	CT	BS/ $\log_2 n$
2	0.0471	0.0482	0.0763	20.7535
5	0.0651	0.0674	0.0663	34.4458
10	0.0991	0.2024	0.0652	16.4146
25	0.1768	0.2477	0.0811	18.7462
50	0.4334	0.2487	0.0651	22.6908
100	0.6175	0.2812	0.0645	23.6279
500	2.6574	0.3602	0.0682	24.8937
1000	5.5061	0.4814	0.0706	20.7002

Implementations and code for comparisons are available in **`findktests`**.