Binary search (C)

From LiteratePrograms

Other implementations: C | C++ | Java

Binary search (http://en.wikipedia.org/wiki/Binary_search_algorithm) is an algorithm used to quickly locate a particular value in a sorted random-access container, such as a sorted array. It operates by repeatedly narrowing the range of possible locations by examining the element in the middle of this range.

C has built-in support in its library for a generic form of binary search based on function pointers (bsearch() (http://www.cplusplus.com/ref/cstdlib/bsearch.html)). Here, we describe a straightforward implementation on integer arrays. We begin with a simple recursive implementation. The search receives four arguments:

- The array
- Lower and upper bound indexes indicating the inclusive range of elements to search
- The value to search for

We then compare the desired value to the value in the center of the range and use this to narrow the range passed to the next recursive call. If the range ever becomes empty (upper < lower) then the element is not present and we return −1 to indicate this.

<<main algorithm (recursive)>>=
```
int binary_search(int a[], int low, int high, int target) {
    if (high < low)
        return -1;
    int middle = (low + high)/2;
    if (target < a[middle])
        return binary_search(a, low, middle-1, target);
    else if (target > a[middle])
        return binary_search(a, middle+1, high, target);
    else if (target == a[middle])
        return middle;
}
```

While effective and clear, this implementation has a few practical problems:

- The recursive calls are all *tail calls*, and so can be rewritten as a loop to save stack space and call overhead. Many C compilers will not perform this rewriting automatically.
- The computation `(low + high)/2` may return incorrect results for very large arrays due to overflow; see Joshua Bloch's article (http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html) .
- The last comparison (==) is not needed, since it is implied by the failure of the first two and the trichotomy law (http://mathworld.wolfram.com/TrichotomyLaw.html) .

Taking these into account, we rewrite it as:

```
<<main algorithm>>=
int binary_search(int a[], int low, int high, int target) {
    while (low <= high) {
        int middle = low + (high - low)/2;
        if (target < a[middle])
            high = middle - 1;
        else if (target > a[middle])
            low = middle + 1;
        else
            return middle;
    }
    return -1;
}
```

To test it, we generate a list of random numbers, sort it, and then search for some values known to be in the list. We allow the number of items in the list to be specified for performance testing. We deliberately exclude one value from the list to test the "not found" case:

```
<<binary_search.c>>=
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

main algorithm

/* For qsort */
int less_int(const void* left, const void* right) {
    return *((const int*)left) - *((const int*)right);
}

int main(int argc, char* argv[]) {
    clock_t start;
    int num_elements = atoi(argv[1]);
    int* a = (int*)malloc(num_elements*sizeof(int));
    int i;
    for(i=0; i<num_elements; i++) {
        do {
            a[i] = rand() % num_elements;
        } while(a[i] == num_elements/7);
    }
    qsort(a, num_elements, sizeof(int), less_int);

    start = clock();
    for(i=0; i<50000; i++) {
        int present_val = a[rand() % num_elements];
        int found_at = binary_search(a, 0, num_elements - 1, present_val);
        if (found_at == -1 || a[found_at] != present_val)
            puts("ERROR");
    }
    printf("Average search time: %f sec\n",
           ((double)(clock() - start))/(CLOCKS_PER_SEC)/50000);

    if (binary_search(a, 0, num_elements - 1, num_elements/7)   != -1 ||
        binary_search(a, 0, num_elements - 1, -1)               != -1 ||
        binary_search(a, 0, num_elements - 1, num_elements + 1) != -1)
    {
        puts("ERROR");
    }
    return 0;
}
```

The cast to `int*` enables this code to compile as C++. It's necessary to add timing code to this example for performance testing because the binary search itself is much faster than the array generation. Because binary search is very fast, we must repeat the test many times to get useful measurements with the resolution of the typical `clock()` timer.

Performance

We measured the performance of binary search using the above test code on lists of a number of sizes. The code was compiled using gcc with the "-O3" flag and run on a Pentium 2.8 GHz with 1GB of RAM under Gentoo Linux. The results show that binary search is extremely fast, even for the largest lists that could be allocated (notice that the time is measured in microseconds, µs).

List size	1,000	10,000	100,000	1M	10M	100M
Binary search (microseconds)	0	0	0	1	1	2

Download code (http://en.literateprograms.org/index.php?title=Special:Downloadcode/Binary_search_(C)&oldid=16429)