Binary Search

Binary search is a method of searching a sorted array. Assume that the array is named `list` and that it has N elements. The elements are `list[0]`, `list[1]`, `list[2]`, ..., `list[N-1]`. The array is being searched to see if it contains a particular value. That value is called the ***search key***. For example, if the array contains names of animals, and we are searching for the word "gorilla", then "gorilla" is the search key.

The idea is to compute the middle position, middle = N/2, and compare `list[middle]` to the search key. If it is bigger, then the key must be in the lower half, so we repeat this procedure in the lower half. For example, if `list[middle]` contains "leopard", then if "gorilla" is in the list at all, it must be somewhere between `list[0]` and `list[middle-1]`. So we repeat the same algorithm on the array that starts at 0 and ends at middle-1.

Summarizing, if `list` is the array being searched, N is the size of the array, and `key` is the item being sought, binary search is of the form

```
start of search region = 0;
end of search region   = N-1;
while key is not found in list and the search region is not empty
    middle = (start of search region + end of search region)/2
    if key is smaller than list[ middle ]
        search the lower half
    else if key is larger than list[ middle]
        search the upper half
    else
        key is found at position middle
```

This can be refined as follows:

```
int first = 0;
int last  = N-1;
bool found = false;

while ( !found and (last - first >= 0 ) ) {
    int middle = (first + last)/2;
    if (key < list[ middle ] )
        last = middle - 1;
    else if key > list[ middle]
        first = middle + 1;
    else
        found = true;
}
if ( found )
    return middle;
else
    return -1;
```

This can be packaged into a function and tidied up a bit to make it a little more efficient, but the idea is still the same. We can eliminate the Boolean variable `found` since it is redundant. We can return from within the loop if we find our key.

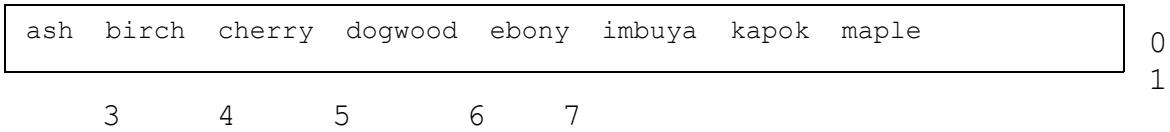
```
int bsearch(const element list[], int size, const element & key)
// precondition: size == # elements in list AND list is sorted
// postcondition: returns index of key in list, -1 if key not found
{
    int low = 0;            // leftmost possible entry
    int high = size-1;      // rightmost possible entry
    int mid;                // middle of current range
    while (low <= high) {
        mid = (low + high)/2;
        if ( key < list[mid] )        // key in lower half
            high = mid - 1;
        else if ( key > list[mid])    // key in upper half
            low = mid + 1;
        else                          // key == list[mid]
            return mid;

    }
    return -1;                      // not in list
}
```

We can analyze the performance of this algorithm. Suppose `list` has N elements and N is a power of 2, i.e., $N = 2^k$. Each time that the while loop is executed, the size of the list being searched is cut in half. Either `high` is reduced to `mid-1` or `low` is increased to `mid+1`. Therefore the size of the sub-array being searched goes roughly from 2^k to 2^{k-1} to 2^{k-2}, and so on, until it becomes size 4, then 2, then 1, and then 0 in the worst possible situation. (Actually, it is slightly less than this each time, because we are removing the middle element each time.) So in the worst case, it loops $\log_2 N$ times. In other words, the total number of comparisons in which the key is compared to an element of the list is at most $2 * \log_2 N$ (twice in each loop iteration). This is MUCH smaller than the time used by linear search.

Example

If we search for "kapok" in



We first compare "kapok" to "dogwood" because 3 = (0 + 7)/2. Since "kapok" is larger, we set first = 4 and compare "kapok" to "imbuya" because (4+7)/2 = 11/2 = 5. Since it is still larger, we set first = 6 and we compare "kapok" to "kapok" because (6+7)/2 = 13/2 = 6, and that is where we find the key.