Divide-and-Conquer using Recursion
Linear Search

- One of the most common computational problems is search.
- In its simplest form, you are given a list L of items and an item k. The problem is to determine if k belongs to L or not.
- If we don’t know anything about L, then the only way to solve this problem is by scanning the entire list in some systematic manner.
- This is called linear search and it takes time proportional to the size of the list.
- For many applications, this can be quite inefficient.
Binary Search

- If the list $L$ is known to be sorted (in ascending or descending order), then we can use a much more efficient algorithm called binary search.

- Binary search is so much more efficient than linear search that it provides a significant incentive to keep lists sorted.

- More on the efficiency of binary search later.
Binary Search Algorithm

- Suppose that L is sorted in ascending order.
- Compare k with the middle element of L
  - If \( k = L[\text{middle}] \), we are done
  - If \( k < L[\text{middle}] \), we need to search the first half of L
  - If \( k > L[\text{middle}] \), we need to search the second half of L

- Notice that after one comparison, the size of the problem shrinks to \( \frac{1}{2} \) of what it was earlier.
- This is the same type of behavior we saw in the power(a, n) algorithm.
Binary Search Algorithm: More Details

- Define a function
  - `binarySearch(L, k, left, right)`

- This function returns `True` if `k` is in the list slice `L[left:right+1]` and `False` otherwise.

- **Base case**: If `left > right` then the slice has size 0 the the function should return `False` without doing any work.
```python
def recursiveBinarySearch(L, k, left, right):
    # Base case
    if left > right:
        return False

    # Recursive cases
    mid = (left + right)/2
    if k == L[mid]:
        return True
    if k < L[mid]:
        return recursiveBinarySearch(L, k, left, mid-1)
    if k > L[mid]:
        return recursiveBinarySearch(L, k, mid+1, right)
```
Examples

- \( L = [3, 5, 13, 18, 18, 27, 34, 56, 57, 60, 89] \), \( k = 13 \):
  
  **Slices searched:**
  
  0 10
  
  0 4

  Found the element.

- \( L = [3, 5, 13, 18, 18, 27, 34, 56, 57, 60, 89] \), \( k = 65 \):
  
  **Slices searched:**
  
  0 10
  
  6 10
  
  9 10
  
  10 10
  
  10 9

  Did not find the element.
Worst Case Running Time

- Assume the **worst case**, i.e., we don’t find k.
- After each comparison of k with L[mid], the size of the problem shrinks to $\frac{1}{2}$ of what it was earlier.

<table>
<thead>
<tr>
<th>Size of input</th>
<th>Number of iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>0</td>
</tr>
<tr>
<td>N/2</td>
<td>1</td>
</tr>
<tr>
<td>N/2^2</td>
<td>2</td>
</tr>
<tr>
<td>N/2^3</td>
<td>3</td>
</tr>
<tr>
<td>N/2^4</td>
<td>4</td>
</tr>
</tbody>
</table>

- Thus, the number of iterations $t$ needed for the size of list to shrink to 1 is such that:

$$\frac{N}{2^t} = 1 \quad \Rightarrow \quad N = 2^t \quad \Rightarrow \quad t = \log_2 N$$
Linear Search versus Binary Search

- In this code we generate a list of 50,000 numbers, each chosen at random from the range 1 through 100,000.

- As we are building the list, we don’t pay attention to the fact that the list may contain duplicates.

- Then we count the number of distinct items there are in this list.

```
N = 50000
L = []
for i in range(N):
    L.append(random.randint(1, 2*N+1))

count = 0
for i in range(1, N+1):
    if i in L:
        count = count + 1

print count
```
I added some lines of code to time different parts of the program.

Time to generate list is: 0.112533092499
Time to count distinct items is: 40.1433010101
19692

The time to count is quite large because the operation $i \in L$ is performed by doing a linear search on $L$.

This means that the first for-loop takes $N$ time and the second for-loop takes $N^2$ time.
Using Binary Search

- In this code, I replaced `i` in `L` by `binarySearch(i, L)`. Also, notice the use of `L.sort()` before we used `binarySearch`.

```python
N = 500000
L = []
for i in range(N):
    L.append(random.randint(1, 2*N+1))
L.sort()

count = 0
for i in range(1, N+1):
    if binarySearch(i, L):
        count = count + 1

print count
```
The second for-loop now takes 0.398 seconds (compared to 40 seconds earlier).

Look at how fast the in-built sorting function is.

I increased N to 500,000 and got the following output:

Time to generate list is: 1.11994886398
Time to sort list is: 0.276556968689
Time to count distinct items is: 4.82709383965
196218