Searching

Sequential / Linear Search

- Search from beginning to the end
- Compare every element in list to the required item
- Match --> found; no match --> go to next element
- Advantages:
  - Elements can be in any order
- Challenges:
  - Require large amount of memory
  - May cause stack overflow and/or memory problems
- Code:

```
array = [ ... ]
target = input("enter item to be searched")

for i in range( len(array) ):
    if array[i] is target:
        return i
```

Binary Search

- Search in order e.g. ascending, descending
- Steps:
  1. Arrange list of elements in order
  2. Compare key with mid-index value of list
  3. Determine which side the key is on the list
  4. E.g. key < mid-index element:
     - Change high-index to mid-1
     - Compare key with new mid
  5. E.g. key > mid-index element:
     - Change low-index to mid+1
     - Compare key with new mid
  6. Repeat until found or not found, low-index > high-index
- Advantages:
  1. Faster search by eliminating half the elements at once
  2. Uses less memory
- Challenges:

1. Elements must be arranged in specific order: ascending or descending

- Iterative code:

```
target = input("enter item to be searched")
array = [...]   # must be sorted

def binary_search(array, target):
    low, high = 0, len(array) - 1

    while low <= high:
        mid = (low + high) // 2
        pivot = array[mid]

        # target found
        if pivot == target:

            return mid

        # target in lower subarray
        elif target < mid_value:
            low = mid + 1

        # target in higher subarray
        else:
            high = mid - 1

    # pointers crossed -- target is not found
    if low > high:
        return -1
```

- Recursive code:

```
target = input("enter item to be searched")
array = [...]   # must be sorted
n = len(array) - 1

# array is already sorted
if len(array) <= 1:
    return array
```

```
else:
    rec_binary_search(target, array, 0, n)

def rec_binary_search(target, array, low, high):

    # target not found
    if low < high:
        return None

    mid = (low + high) // 2
    pivot = array[mid]


    # target found
    if pivot == target:
        return mid

    # target in lower subarray
    elif pivot < target:
        rec_binary_search(target, array, low, mid - 1)

    # target in higher subarray
    else:
        rec_binary_search(target, array, mid + 1, high)
```

Hash Table Search

- **Hash Function:**
  - Location of an item is determined directly as a function of the time itself rather than by a sequence of trial-and-error comparisons
  - Only one location is required to be examined
  - Time required to locate the item is constant and doesn't depend on the number of items stored
- **Collision Strategies:**
  - Collision: Many values may have the same hash value and is tried to be stored at the same location
  - **Linear Probing:**
    - Linear search of the table begins at the location where collision occurs and continues until an empty slot is found in which the item can be stored

    - Determining is an element is in the hash table:
        - Apply hash function to compute the position of the value
        - Three cases to consider:
            1. If location is empty, the value is not in the table
            2. If location contains the specified value, search is immediately successful
            3. If location contains a value other than the specified value, begin a “circular” linear search until the item is found or reach empty location or the starting location, indicating item isn’t in the table
- **Chaining:**
    - All elements that are stored at the same location are chained together
    - Advantages:
        - Fast searching
    - Challenges:
        - Collisions occur, causing some elements to occupy locations reserved for other hash values
        - Hash table may not have enough space to store all elements