Section B3 Searching

Linear Search

```
def linear_search(slist, item):
    '''
    Search the slist for a matching item.

    :param slist: the data to be searched.
    :param item: an item to match against
    :return True when item is found in slist and the index of the location
    '''

    index = 0
    found = False

    # Match the value with each data element
    while not found and index < len(slist):
        if slist[index] == item:
            found = True
        else:
            index = index + 1

    return found, index
```

time complexity: O(n)

Binary Search

This provides worst case time complexity of O(log n).
We can narrate the process of a recursive binary search for a value 'x' as follows:

```
# compare x with the middle element.
#
# if x matches with the middle element, we return the mid index.
#
# else
#
#    if x is greater than the mid element, then x can only lie in right half
#    sublist after the mid element. So we recursively search in the right half.
#
#    else (x is smaller) recursively search in the left half.
```

Note that it's feasible to implement this algorithm iteratively or recursively. Here we provide a recursive sample implementation.

```python
def binary_search(slist, low, high, match):
    '''
    Search a list for a matching item.

    :param slist: input data to search.
    :param low: the index in slist of the lowest element in this search.
    :param high: the index in slist of the highest element in this search.
    :param match: the element to match against.
    :return True when the item is found and the index position in the list.
    '''

    # Check base case
    if high >= low:

        mid = (high + low) // 2

        # If element is present at the middle itself
        if slist[mid] == match:
            return True, mid

        # If element is smaller than mid, then it can only
        # be present in left subslist
        elif slist[mid] > match:
            return binary_search(slist, low, mid - 1, match)

        # Else the element can only be present in right sublist
        else:
            return binary_search(slist, mid + 1, high, match)

    else:
        # Element is not present in the list
        return False, -1
```

Interpolation Search

The worst case time complexity of this search algorithm is O(n). However, assuming a known and uniform distribution of the data along a linear scale used for interpolation, average performance is O(log log n).

```
def interpolation_search(slist, match):
    '''
    Using interpolation, search a list of integers for a target value 'match' and
    return whether 'match' was found and at what indematch of slist.
    '''
    low = 0
    high = (len(slist) - 1)

    while((slist[high] != slist[low]) and (match >= slist[low]) and (match <= slist[high])):

        # Interpolate a search index
        mid = low + int((match - slist[low]) * (high - low) / (slist[high] - slist[low]))

        # Compare the value at the interpolated index with search value
        if slist[mid] < match:
            low = mid + 1
        elif match < slist[mid]:
            high = mid - 1
        else:
            return True, mid

    if match == slist[low]:
        return True, low
    else:
        return False, -1
```

插值公式是插值查找的核心。它估算 match 可能在列表中的位置。此估算基于 slist 在 low 和 high 之间的线性分布。
如果循环结束后，match 等于 low 索引处的值，函数返回 True 和 low 值。否则，表示 match 不在 slist 中，函数返回 False 和-1。

Jump Search

Jump search is a hybrid search algorithm similar to binary search in that it works on a sorted array, and uses a similar *divide and conquer* approach to search through it. With each jump, we store the previous value we looked at and its index. When we find a set of values where slist[i] < element < slist[i + jump], we perform a linear search with slist[i] as the left-most element and slist[i + jump] as the right-most element in our search set:

```python
import math

def jump_search (slist, match):
    '''
    Perform a jump search of slist looking for match. Return True and the found index of
    match in slist when found, otherwise False.
    '''
    length = len(slist)
    jump = int(math.sqrt(length))
    left, right = 0, 0

    while left < length and slist[left] <= match:
        right = min(length - 1, left + jump)
        if slist[left] <= match and slist[right] >= match:
            break
        left += jump
    if left >= length or slist[left] > match:
        return False, -1
    right = min(length - 1, right)
    i = left
    while i <= right and slist[i] <= match:
        if slist[i] == match:
            return True, i
        i += 1
    return False, -1
```

时间复杂度:

最佳情况 (Best Case): 当我们的搜索元素在列表的开始位置时，时间复杂度是 $O(1)$。

最差情况 (Worst Case): 当元素分布均匀且我们的搜索元素接近列表的末尾或在最后一块中时，我们必须对最后一块进行线性搜索。在这种情况下，时间复杂度为 $O(\sqrt{n})$。其中 n 是列表的大小。

平均情况 (Average Case): 对于平均情况，时间复杂度也是 $O(\sqrt{n})$。

空间复杂度:

跳跃搜索是一个原地搜索算法，这意味着它不需要额外的存储来完成搜索。因此，它的空间复杂度是 $O(1)$。

需要注意的是，跳跃搜索适用于已排序的列表，并且通常比线性搜索更快，但比二分搜索慢。它的一个优点是相对于二分搜索，它对于反向链接的列表（例如双向链接的列表）更加友好，因为在这种列表中，前后移动成本相似。