Algorithms and data structures

- How do you find efficient algorithms?
  - Hard to invent new ones
  - Easier to reduce problems to known solutions
    - Understand inherent complexity of problem
    - Think about how to break problem into sub-problems
    - Relate sub-problems to other problems for which there already exist efficient algorithms

Search algorithms

- Search algorithm – method for finding an item or group of items with specific properties within a collection of items.
- Collection called the search space
- Saw examples – finding square root as a search problem
  - Exhaustive enumeration
  - Bisection search
  - Newton-Raphson

Linear search and indirection

- Simple search method

```
def search(L, e):
    for i in range(len(L)):
        if L[i] == e:
            return True
    return False
```

- Complexity?
  - If element not in list, `O(len(L))` tests
  - So at best linear in length of L

Linear search and indirection

- Why “at best linear”?
  - Assumes each test in loop can be done in constant time
  - But does Python retrieve the i^{th} element of a list in constant time?

Indirection

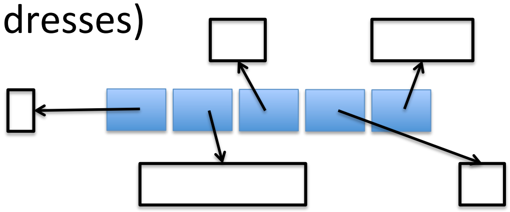
- Simple case: list of ints
  - Each element is of same size (e.g., four units of memory – or four eight bit bytes)
  - Then address in memory of i^{th} element is `start + 4 * i` where `start` is address of start of list
  - So can get to that point in memory in constant time

Indirection

- If length field is 4 units of memory, and each pointer occupies 4 units of memory
- Then **address** of i^{th} element is stored at `start + 4 + 4 * i`
- This address can be found in constant time, and value stored at address also found in constant time
- So search is linear
- **Indirection –** accessing something by first accessing something else that contains a reference to thing sought

Indirection

- But what if list is of objects of arbitrary size?
- Use indirection
- Represent a list as a combination of a length (number of objects), and a sequence of fixed size pointers to objects (or memory addresses)



Binary search

- Can we do better than `O(len(L))` for search?
- If know nothing about values of elements in list, then no.
- Worst case, would have to look at every element

What if list is ordered?

- Suppose elements are sorted in ascending order

```
def search(L, e):
    for i in range(len(L)):
        if L[i] == e:
            return True
        if L[i] > e:
            return False
    return False
```

- Improves average complexity, but worst case still need to look at every element

Binary search

```
def search(L, e):
    def bSearch(L, e, low, high):
        if high == low:
            return L[low] == e
        mid = low + int((high - low)/2)
        if L[mid] == e:
            return True
        if L[mid] > e:
            return bSearch(L, e, low, mid - 1)
        else:
            return bSearch(L, e, mid + 1, high)

    if len(L) == 0:
        return False
    else:
        return bSearch(L, e, 0, len(L) - 1)
```

Use binary search

1. Pick an index, `i`, that divides list in half
2. Ask if `L[i] == e`
3. If not, ask if `L[i]` larger or smaller than `e`
4. Depending on answer, search left or right half of `L` for `e`

A new version of a divide-and-conquer algorithm

- Break into smaller version of problem (smaller list), plus some simple operations
- Answer to smaller version is answer to original problem

Analyzing binary search

- Does the recursion halt?
  - Decrementing function
    1. Maps values to which formal parameters are bound to non-negative integer
    2. When value is <= 0, recursion terminates
    3. For each recursive call, value of function is strictly less than value on entry to instance of function
  - Here function is high – low
    - At least 0 first time called (1)
    - When exactly 0, no recursive call, returns (2)
    - Otherwise, halt or recursively call with value halved (3)
- So terminates

Analyzing binary search

- What is complexity?
  - How many recursive calls? (work within each call is constant)
  - How many times can we divide `high - low` in half before reaches 0?
  - $\log_2$`(high - low)`
  - Thus search complexity is `O(log(len(L)))`

Amortizing costs

- But suppose we want to search a list k times?
- Then is sort(L) + k*log(len(L)) < k*len(L)?
  - Depends on k, but one expects that if sort can be done efficiently, then it is better to sort first
  - Amortizing cost of sorting over multiple searches may make this worthwhile
  - How efficiently can we sort?

Sorting algorithms

- So what about cost of sorting?
- Assume complexity of sorting a list is O(sort(L))
- Then if we sort and search we want to know if sort(L) + log (len(L)) < len(L)
  - I.e. should we sort and search using binary, just use linear search
- Can't sort in less than linear time!

Selection sort

```
def selSort(L):
    for i in range(len(L) - 1):
        minIndx = i
        minVal= L[i]
        j = i + 1
        while j < len(L):
            if minVal > L[j]:
                minIndx = j
                minVal= L[j]
            j += 1
        temp = L[i]
        L[i] = L[minIndx]
        L[minIndx] = temp
```

Analyzing selection sort

- Loop invariant
  - Given prefix of list L[0:i] and suffix L[i+1:len(L)-1], then prefix is sorted and no element in prefix is larger than smallest element in suffix
  1. Base case: prefix empty, suffix whole list – invariant true
  2. Induction step: move minimum element from suffix to end of prefix. Since invariant true before move, prefix sorted after append
  3. When exit, prefix is entire list, suffix empty, so sorted

Merge sort

- Use a divide-and-conquer approach:
  1. If list is of length 0 or 1, already sorted
  2. If list has more than one element, split into two lists, and sort each
  3. Merge results
     1. To merge, just look at first element of each, move smaller to end of the result
     2. When one list empty, just copy rest of other list

Analyzing selection sort

- Complexity of inner loop is O(len(L))
- Complexity of outer loop also O(len(L))
- So overall complexity is O(len(L)^2) or quadratic
- Expensive

```
def selSort(L):
    for i in range(len(L) - 1):
        minIndx = i
        minVal= L[i]
        j = i + 1
        while j < len(L):
            if minVal > L[j]:
                minIndx = j
                minVal= L[j]
            j += 1
        temp = L[i]
        L[i] = L[minIndx]
        L[minIndx] = temp
```

Example of merging

Left in list 1	Left in list 2	Compare	Result
[1,5,12,18,19,20]	[2,3,4,17]	1, 2	[]
[5,12,18,19,20]	[2,3,4,17]	5, 2	[1]
[5,12,18,19,20]	[3,4,17]	5, 3	[1,2]
[5,12,18,19,20]	[4,17]	5, 4	[1,2,3]
[5,12,18,19,20]	[17]	5, 17	[1,2,3,4]
[12,18,19,20]	[17]	12, 17	[1,2,3,4,5]
[18,19,20]	[17]	18, 17	[1,2,3,4,5,12]
[18,19,20]	[]	18, --	[1,2,3,4,5,12,17]
[]	[]		[1,2,3,4,5,12,17,18,19,20]

Complexity of merge

- Comparison and copying are constant
- Number of comparisons – O(len(L))
- Number of copyings – O(len(L1) + len(L2))
- So merging is linear in length of the lists

Putting it together

```
import operator

def mergeSort(L, compare = operator.lt):
    if len(L) < 2:
        return L[:]
    else:
        middle = int(len(L)/2)
        left = mergeSort(L[:middle], compare)
        right = mergeSort(L[middle:], compare)
        return merge(left, right, compare)
```

```
def merge(left, right, compare):
    result = []
    i,j = 0, 0
    while i < len(left) and j < len(right):
        if compare(left[i], right[j]):
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    while (i < len(left)):
        result.append(left[i])
        i += 1
    while (j < len(right)):
        result.append(right[j])
        j += 1
    return result
```

Complexity of merge sort

- Merge is O(len(L))
- Mergesort is O(len(L)) * number of calls to merge
  - O(len(L)) * number of calls to mergesort
  - O(len(L) * log(len(L)))
- Log linear – O(n log n), where n is len(L)
- Does come with cost in space, as makes new copy of list

Improving efficiency

- Combining binary search with merge sort very efficient
  - If we search list k times, then efficiency is n*log(n) + k*log(n)
- Can we do better?
- Dictionaries use concept of hashing
  - Lookup can be done in time almost independent of size of dictionary

Hashing

- Convert key to an int
- Use int to index into a list (constant time)
- Conversion done using a **hash function**
  - Map large space of inputs to smaller space of outputs
  - Thus a many-to-one mapping
  - When two inputs go to same output – a **collision**
  - A good hash function has a uniform distribution – minimizes probability of a collision

Complexity

- If no collisions, then O(1)
- If everything hashed to same bucket, then O(n)
- But in general, can trade off space to make hash table large, and with good function get close to uniform distribution, and reduce complexity to close to O(1)