Week 10

Divide and Conquer

- Divide — if it is a base case, solve directly, otherwise break up the problem into several parts.
- Recur — recursivly solve each part [each sub-problem]
- Conque — Combine the solution of each part into the Overall

Binary Search

Constarin:

- Each element in the given Collection is sortable and comparatable
- Given collection is in sorted order

Implementation Sample

```
def binary_search(given_collection:list, x)->bool:
  if given_collection == None:
    return given_collection

  if len(given_collection) == 0:
    return given_collection

  middle = len(given_collection)//2
  if given_collection[middle] == x:
    return True;
  if given_collection[middle] > x:
    return binary_search(given_collection[:middle], x);
  else:
    return binary_search(given_collection[middle:], x);

if __name__ == '__main__':
  sample_list = [1,2,3,4,5,6,7,8,9,10,19,20,30,40,45,55]
  print(binary_search(sample_list, 20));
```

Time Complexity:

Divide step (find middle and compare to x) — $O(1)$

Recur step (solve left or right subproblem) — $T(n/2)$

Conquer step (return answer from recursion) — $O(1)$

According to the Master theorem, the total time complexity is $T(n) = O(logn)$.

Merge-Sort

- Divide — divide the array into two half
- Recur — recursively sort each half
- Conquer — two sorted halves to make a single sorted array

Implementation Sample

```
def merge(left:list, right:list)->list:
  sorted_combine = [];
  l,r = 0,0;
  sum_length = len(left) + len(right);
  while (l+r) < sum_length:
    index = (l+r);
    if r >= len(right) or (l < len(left) and left[l] <= right[r]):
      sorted_combine.append(left[l]);
      l+=1;
    else:
      sorted_combine.append(right[r]);
      r+=1;
  return sorted_combine;

def merge_sort(given_collection:list)->list:
  if len(given_collection) < 2:
    return given_collection;

  # divide
  mid = len(given_collection)//2;

  # recur
  sorted_left = merge_sort(given_collection[:mid]);
  sorted_right = merge_sort(given_collection[mid:]);

  # conquer
  return merge(sorted_left, sorted_right);
```

```
if __name__ == '__main__':

  sample_list2 = [2,5,1,0,9,6,8,0,1]
  print(merge_sort(sample_list2));
```

Time complexity:

Divide — $O(1)$

Recur — $2 * T(n/2)$

Merge — $O(n)$

Total time complexity: $O(nlogn)$

Quick-Sort

- Divide — Choose a random element from the list as the pivot partition the elements into 3 lists
  - (1) less than, (2) equal to, (3) greater than the pivot
- Recur — recursively sort the less than and greater than lists
- Conquer — Join the sorted 3 lists together

Implementation Sample:

```
import random

def quick_sort(given_collection:list)->list:
  if len(given_collection)<2:
    return given_collection;

  choice = random.choice(given_collection);
  greater_choice, lesser_choice, equal_choice = [], [], [];
  for element in given_collection:
    if element == choice:
      equal_choice.append(element);
    elif element < choice:
      lesser_choice.append(element);
    else:
      greater_choice.append(element);
```

```
  greater_choice = quick_sort(greater_choice);
  lesser_choice = quick_sort(lesser_choice);

  return lesser_choice + equal_choice + greater_choice;

if __name__ == '__main__':
  sample_list2 = [2,5,1,0,9,6,8,0,1]
  print(quick_sort(sample_list2));
```

Time Complexity:

Worst case — $O(n^2)$

Everage case — $E[T(n)] = O(nlogn)$