1 Introduction to Divide and Conquer

Divide and conquer alogirthms can noramlly be broken into three parts:

1. **Divide:** If it is a base case, solve directly, otherwise break up the problem into several parts.
2. **Recur/Delegate:** Recursively solve each part [each sub-problem].
3. **Conquer:** Combine the solutions of each part into the overall solution.

2 Searching Sorted Array

Given a sorted sequence S of n numbers $a_0, ..., a_{n-1}$ stored in an array A[0, 1, . . . , n - 1], determine if a given number x is in S. The Naiive appraoch is to check every element and see if it is equal to x.

2.1 Binary Search in sorted A[0 to n-1]

1. If the array is empty, then return "No"
2. Compare x to the middle element, namely A[floor(n/2)]
3. If this middle element is x, then return "Yes"
4. If A[floor(n/2)]>x, then recursively search A[0 to floor(n/2)-1]
5. if A[floor(n/2)]<x, then recursively search A[floor(n/2)+1 to n-1]

```
1. def BinarySearch(A, left, right, x):          # A is sorted and left <= right
2.     if left = right then
3.         return "Unsuccessful"
4.     mid = ⌊(left + right) ÷ 2⌋
5.     if A[mid] < x then
6.         return BinarySearch(A, mid + 1, right, x)
7.     else if A[mid] > x then
8.         return BinarySearch(A, left, mid - 1, x)
9.     else
10.        return mid
```

2.2 Binary Search correctness

Invariant: If x is in A before the divide step, then x is in A after the divide step.

1. if A[mid] < x, then x is in A[mid + 1 to right]
2. if A[mid] > x, then x is in A[left to mid - 1]

Every divide step leads to a smaller array. Thus, if x is in A, we will eventually inspect its position due to the invariant and return yes. Thus, if x is not in A, then eventually we will reach an empty array and return "No".

2.3 Recurrence formula

An easy way to analyze the time complexity of a divide-andconquer algorithm is to define and solve a recurrence. Let T(n) be thr running tme of the alogirthm, then:

1. Divide step cost in terms of n.
2. Recur step(s) cost in terms of T(smaller values).
3. Conquer step cost in terms of n.

Using this, we can set up a recurrence for T(n) and then solve it:

$$T(n) = \begin{cases} \text{“Recur” + “Divide and Conquer”} & \text{for } n > 1 \\ \text{“Base case” (typically O(1))} & \text{for } n = 1 \end{cases}$$

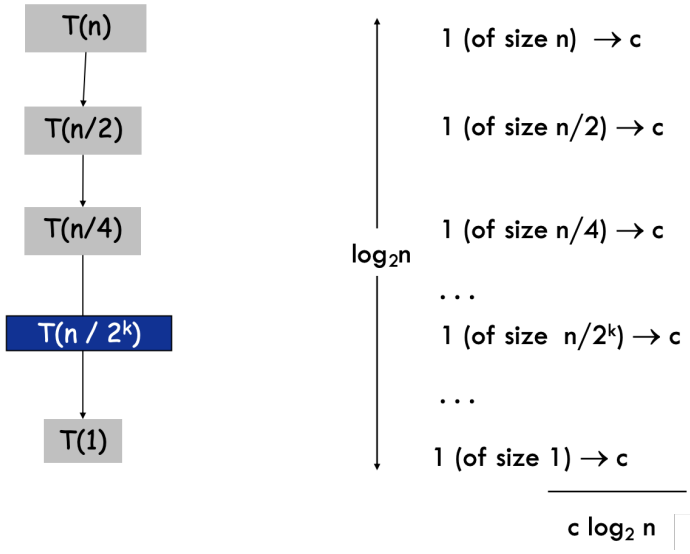
3 Unrolling

Divide step (find middle and compare to x) takes O(1)
Recur step (solve left or right subproblem) takes T(n/2)
Conquer step (return answer from recursion) takes O(1)

Now we can set up the recurrence for T(n):

$$T(n) = \begin{cases} T(n/2) + O(1) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to T(n) = O(log n), since we can only halve the input O(log n) times before reaching a base case



Divide step (find middle and compare to x) takes O(n)

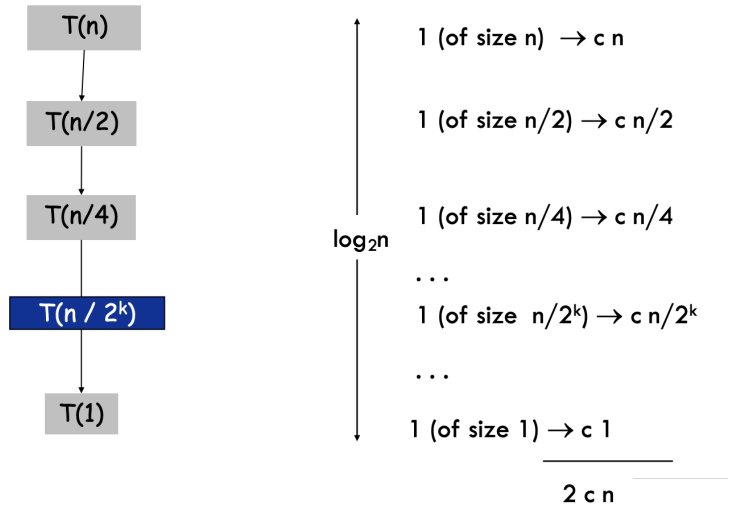
Recur step (solve left or right subproblem) takes T(n/2)

Conquer step (return answer from recursion) takes O(1)

Now we can set up the recurrence for T(n):

$$T(n) = \begin{cases} T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to T(n) = O(n), since to access the next index we end up with n/2 + n/4 + n/8 + …



4 Merge-Sort

Divide the array into two halves. Recurrsively sort each half. Conquer two sorted halves to make a single sorted array.

```
1. def merge_sort(s):
2.     if |S| < 2 then                                          # base case
3.         return s
4.     else
5.         mid ← ⌊|S| ÷ 2⌋                                      # divide
6.         left ← S[:mid]
7.         right ← S[mid:]
8.         sorted_left ← merge_sort(left)                       # recursion
9.         sorted_right ← merge_sort(right)
10.         return merge(sorted_left, sorted_right)             # conquer
```

```
1. def merge(L, R):
2.         result ← array of length (|L| + |R|)
3.         l, r ← 0, 0
4.         while l + r < |result| do
5.             index ← l + r
6.             if r >= |R| or (l < |L| and L[l] < R[r]) then
7.                 result[index] ← L[l]
8.                 l ← l + 1
9.             else
10.                result[index] ← R[r]
11.                r ← r + 1
12.         return result
```

4.1 Merge Correctness

Induction hypothesis: After the i-th iteration, our result contains the i smallest elements in sorted order.

Base case: After 0 iterations, our result is empty, so it contains the 0 smallest elements in sorted order.

Induction: Assume IH after iteration k. RTP to prove it after iteration k+1. Since both halves are sorted and we add the smallest element not already in result, result now contains the k+1 smallest elements. Sorted order follows from the fact that both halves are sorted, thus adding the smallest element implies sorted order of result.

4.2 Merge-Sort Correctnes

Induction hypothesis: Merge-Sort correctly sorts an array of size i

Base case: If our array has size 0 or 1, it's already sorted.

Induction: Assume IH for all arrays up to size k. RTP it for array of size k+1. Splitting the array in half gives us two array of size at most k, so by IH those are sorted correctly. We proved that given two sorted arrays, Merge returns a correctly sorted array containing the elements of both arrays. Hence, by running Merge on the two sorted halves, we sort the original array.

4.3 Merge sort complexity analysis

Divide step (find middle and split) takes O(n)

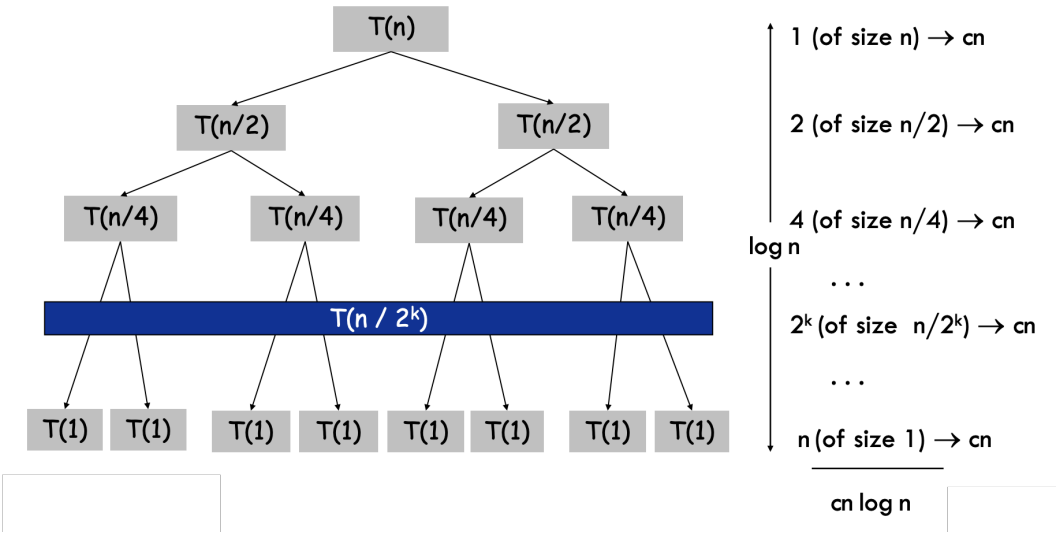
Recur step (solve left and right subproblem) takes 2 T(n/2)

Conquer step (merge subarrays) takes O(n)

Now we can set up the recurrence for T(n):

$$T(n) = \begin{cases} 2\,T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to T(n) = O(n log n)



5 Quick sort

1. **Divide:** Choose a random element from the list as the pivot. Partition the elements into 3 lists: (i) less than, (ii) equal to and (iii) greater than the pivot.
2. **Recursively sort:** Recursively sort the less than and greater than lists.
3. **Conquer:** Concatenate the 3 sorted lists.

5.1 Quick sort complexity analysis

Divide step (pick pivot and split) takes O(n)

Recur step (solve left and right subproblem) takes $T(n_L) + T(n_R)$

Conquer step (merge subarrays) takes O(n)

Now we can set up the recurrence for T(n):

$$E[T(n)] = \begin{cases} E[T(n_L) + T(n_R)] + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $E[T(n)] = O(n \log n)$ expected time

(details available on the textbook but not examinable)

6 Other notes on divide and conquer

Important: Simply using Merge-Sort in your algorithm doesn't make your algorithm a divide and conquer algorithm. For example, A greedy algorithm could use merge sort for its sorting step.

6.1 Comparison sorting lower bound

Fact: Comparison-based sorting algorithms take $\Omega(n \log n)$ time.

Proof: The decision tree associated with a comparison-based sorting algorithm is binary and has $n!$ external nodes. Thus, the height is $\log n!$, which is $\Omega(n \log n)$.

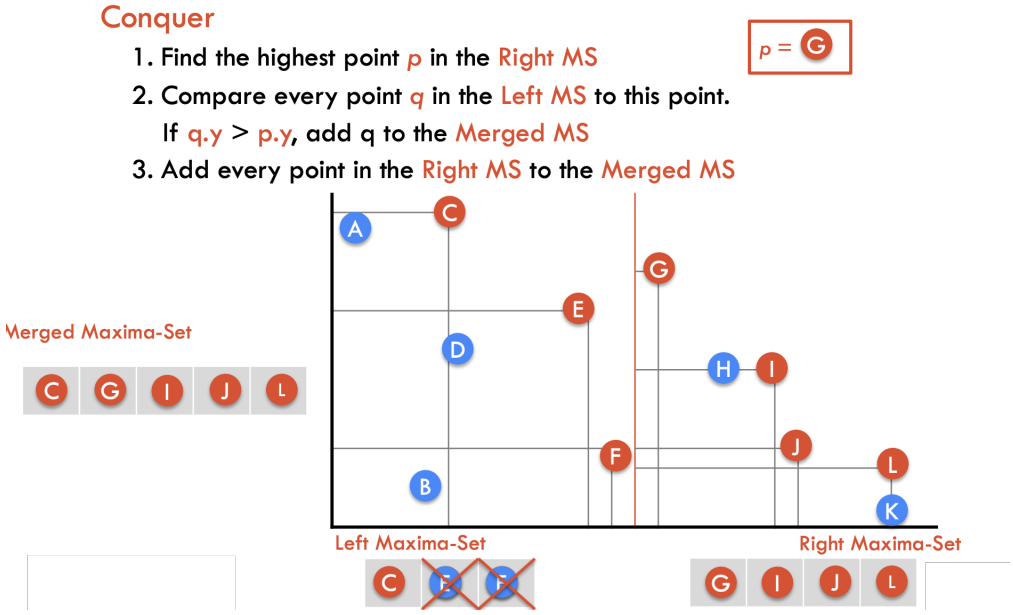
6.2 Some recurrence formulas with solutions

Recurrence	Solution
$T(n) = 2\,T(n/2) + O(n)$	$T(n) = O(n \log n)$
$T(n) = 2\,T(n/2) + O(\log n)$	$T(n) = O(n)$
$T(n) = 2\,T(n/2) + O(1)$	$T(n) = O(n)$
$T(n) = T(n/2) + O(n)$	$T(n) = O(n)$
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log n)$
$T(n) = T(n\text{-}1) + O(n)$	$T(n) = O(n^2)$
$T(n) = T(n\text{-}1) + O(1)$	$T(n) = O(n)$

7 Maxima-Set (Pareto frontier)

Definition: A point is maximum in a set if all other points in the set have either a smaller x- or smaller y-coordinate.
Problem: Given a set S of n distinct points in the plane (2D), find the set of all maximum points.

1. **Preprocessing:** Sort the points by increasing x coordinate and store them in an array. Note: we only do this once. Break ties in x by sorting by increasing y coordinate.
2. **Divide:** Split the sorted array into two halves.
3. **Recur:** Recursively find the maximum points in each half.
4. **Conquer:** Merge the two sets of maximum points.



7.1 Maxima-Set: Analysis

Preprocessing Sort the points by increasing x coordinate and store them in an array. Note: we only do this once. Break ties in x by sorting by increasing y coordinate. — $O(n \log n)$

Divide sorted array into two halves. — $O(n)$

Recur recursively find the MS of each half. — $2T(n/2)$

Conquer compute the MS of the union of Left and Right MS — $O(n)$

1. Find the highest point p in the Right MS
2. Compare every point q in the Left MS to this point.
   If q.y > p.y, add q to the Merged MS
3. Add every point in the Right MS to the Merged MS

$$T(n) = 2T(n/2) + O(n) = O(n \log n)$$

Overall Running Time: pre-processing + $T(n) = O(n \log n)$

7.2 Maxima-Set: Correctness

Preprocessing Sort the points by increasing x coordinate and store them in an array. Note: we only do this once. Break ties in x by sorting by increasing y coordinate.

Divide sorted array into two halves.

Recur recursively find the MS of each half.

Conquer compute MS of union of Left/Right MS

1. Find the highest point p in the Right MS
2. Compare every point q in the Left MS to this point.
   If q.y > p.y, add q to the Merged MS
3. Add every point in the Right MS to the Merged MS

Observations:

1. Every point in MS of the whole is in Left MS or Right MS
2. Every point in Right MS is in MS of the whole
3. Every point in Left MS is either in MS of the whole or is dominated by p

8 Integer multiplication

```
1. def multiply(x, y):
2.     if x = 0 or y = 0 then return 0                    # base case
3.     if x = 1 then return y
4.     if y = 1 then return x
5.     let x_1 and x_0 be such that x = x_1 2^{n/2} + x_0
6.     let y_1 and y_0 be such that y = y_1 2^{n/2} + y_0
7.     return multiply(x_1, y_1) 2^n + (multiply(x_1, y_0) + multiply(x_0, y_1)) 2^{n/2}
        + multiply(x_0, y_0)
```

Recall $x\,y = x_1\,y_1\,2^n + (x_1\,y_0 + x_0\,y_1)\,2^{n/2} + x_0\,y_0$

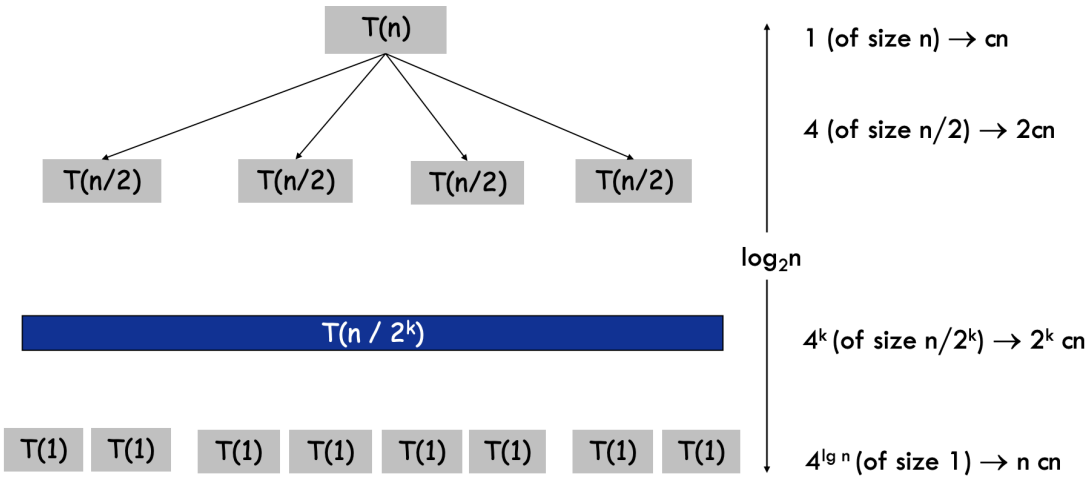
Divide step (produce halves) takes O(n)

Recur step (solve subproblems) takes 4 T(n/2)

Conquer step (add up results) takes O(n)

$$T(n) = \begin{cases} 4\,T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $T(n) = O(n^2)$. No better than naïve!!!



8.1 Attempt 2

Let $x = x_1\ 2^{n/2} + x_0$ and $y = y_1\ 2^{n/2} + y_0$

$$x\ y = x_1\ y_1\ 2^n + (x_1\ y_0 + x_0\ y_1)\ 2^{n/2} + x_0\ y_0$$
$$(x_1 + x_0)\ (y_1 + y_0) = x_1\ y_1 + x_1\ y_0 + x_0\ y_1 + x_0\ y_0$$

We can compute the product of two n-digit numbers by making 3 recursive calls on n/2-digit numbers and then combining the solutions to the subproblems.

```
1. def multiply(x, y):
2.     if x = 0 or y = 0 then return 0                          # base case
3.     if x = 1 then return y
4.     if y = 1 then return x
5.     first_term  ← multiply(x_1, y_1)
6.     last_term   ← multiply(x_0, y_0)
7.     other_term  ← multiply(x_1 + x_0,y_1 + y_0)
8.     return first_term 2^n + (other_term - first_term - last_term) 2^(n/2) + last_term
```

Divide step (produce halves) takes O(n)

Recur step (solve subproblems) takes 3 T(n/2)

Conquer step (add up results) takes O(n)

$$T(n) = \begin{cases} 3\ T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $T(n) = O(n^{\log_2 3})$, where $\log_2 3 \approx 1.6$
Better than naïve!!!

9 Math concepts to remember

Base exchange rule:

$$\log_a x = \frac{\log_b x}{\log_b a}$$

Product rule:

$$\log_a(xy) = \log_a x + \log_a y$$

Power rule:

$$\log_a x^b = b \log_a x$$

Let r be +real and k a +integer

$$1 + r + r^2 + \ldots + r^k = \frac{r^{k+1} - 1}{r - 1}$$

Consequently, if $r > 1$

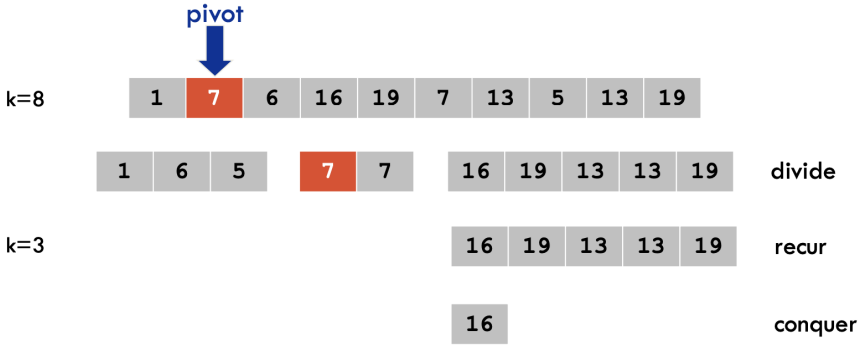
$$1 + r + r^2 + \ldots + r^k < \frac{r^{k+1}}{r - 1}$$

and if $r < 1$

$$1 + r + r^2 + \ldots + r^k < \frac{1}{1 - r}$$

10 Quick selection

1. Divide Choose a random element from the list as the pivot
   Partition the elements into 3 lists:
   (i) less than, (ii) equal to and (iii) greater than the pivot
2. Recur Recursively select right element from correct list
3. Conquer Return solution to recursive problem



Divide step (pick pivot and split) takes O(n)

Recur step (solve left and right subproblem) takes T(n')

Conquer step (return solution) takes O(1)

Now we can set up the recurrence for T(n):

$$E[T(n)] = \begin{cases} E[T(n')] + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $E[T(n)] = O(n)$

(details available on the textbook but not examinable)