❖ Divide and Conquer

An **iterative** or **incremental** algorithm like insertion sort has the advantage of being relatively simple to analyze and implement. However, it can be hard to find an algorithm that achieves optimal running times in that model. A lot of algorithms that perform really well are **recursive** in structure. In this section we introduce the **divide and conquer** paradigm for algorithm design.

In divide and conquer, we implement the following strategy:

1. If the problem is **small enough**, we solve the problem directly.
2. If the problem is **large**, we
   - **Divide** the problem into smaller subproblems
   - **Conquer** the subproblems by solving them recursively
   - **Combine** the subproblem solutions to form a solution for the original problem.

3.1 Mergesort

To formalize some of the ideas and notation we will be using, let's start with a divide and conquer algorithm for solving the sorting problem.

- **Divide**: If the array has two or more elements, divide it into two halves.
- **Conquer**: Sort each of the two halves by **recursively** calling mergesort on each of the halves.
- **Combine**: Combine the two sorted halves into a sorted array of all the elements.

```
def merge(arr1, arr2):
    n = len(arr1) + len(arr2)
    [[Create new array A of length n]]
    p1, p2 = 0, 0  # Pointers to the starting indices of arrays
    while p1 < len(arr1) and p2 < len(arr2):
        [[Choose the smaller of arr1[p1] and arr[p1]]]
        [[Append the chosen element to A]]
        [[Add 1 to the pointer from the chosen array]]
    return A
```

Question 28. Suppose we call `merge` on two arrays whose combined length is n. Write down the runtime of merge using Big-O notation.

```
def merge_sort(arr):
    n = len(arr)
    [[Split arr into two halves arr1 and arr2]]
    arr1 = merge_sort(arr1)
    arr2 = merge_sort(arr2)
    return merge(arr1, arr2)
```

Question 29. Let $T(n)$ be the total running time of the `merge_sort` as written above. On the right of each line, write down the time complexity of each line. What is the total runtime of `merge_sort`?

Question 30. Let's assume that n is some power of 2. Solve the recurrence from above for this case.

In general, divide and conquer algorithms can have their running times described by recurrences in the following form:

$$T(n) = D(n) + aT(n/b) + C(n). \tag{18}$$

3.2 The Master Theorem

Coming up with the recurrence is the first part of analyzing recursive algorithms. In the case of mergesort on an array whose size is a power of two, we were able to directly solve for the recurrence. This is not always easy to do, and in this section we will meet **the master theorem** which will give us a guide on how to analyze such algorithms.

3.2.1 Picturing the work in recursive algorithms

Question 31. Draw the recursion tree for the recurrence representing the runtime $T(n)$ of a call to mergesort, where the nodes are labeled by the cost of the nonrecursive part of the function.

Question 32. Draw the recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$.

3.2.2 The Master Theorem

We are now equipped with some intuition to think about the Master theorem.

Theorem 3.1 (Master theorem*)**.** Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a driving function that is defined and nonnegative on all sufficiently large reals. Define the recurrence $T(n)$ on $n \in \mathbb{N}$ by

$$T(n) = aT(n/b) + f(n). \tag{19}$$

Then the asymptotic behavior of $T(n)$ can be characterized as follows:

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b(a)-\epsilon})$,
   then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b(a)} \log^k n)$,
   then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b(a)+\epsilon})$, and if $f(n)$ satisfies the **regularity condition** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n,
   then $T(n) = \Theta(f(n))$.

In each case, we are interested in comparing two functions.

- The **watershed function**: $n^{\log_b a}$, and
- the **driving function**: $f(n)$.

Question 33. Use the master theorem to show that the running time for mergesort is $\Theta(n \log n)$.

Question 34. Solve the recurrence $T(n) = 16T(n/4) + 7\sqrt{n}$.

Question 35. Solve the recurrence $T(n) = T(7n/8) + 16$.

Question 36. Solve the recurrence $T(n) = 3T(n/4) + n \log n$.

Question 37. Solve the recurrence $T(n) = 4T(n/4) + 11n \log^2 n$.

3.3 Integer multiplication

Problem (INTMUL)**.**

Input: Two integers r, s with n digits.

Output: The product $z = rs$.

Recall the integer multiplication algorithm you learned in grade school.

Can we do better?

We can divide the inputs into two smaller integers with $n/2$ digits:

$$r = r_1 * 10^{n/2} + r_2$$
$$s = s_1 * 10^{n/2} + s_2$$

Question 38. Write down the product z with using the variables r_1, r_2, s_1, s_2. Let $T(n)$ be the time it takes to multiply two n digit integers. Write down a recurrence describing a recursive algorithm to solve the product you wrote down.

Question 39. Solve the recurrence you derived from the previous problem.

We can use the following change of variables:

- $u = r_1 * s_1$
- $v = r_2 * s_2$
- $w = (r_1 + r_2) * (s_1 + s_2)$

Question 40. Write down the product $z = r * s$ using the variables u, v, w.

Question 41. Write down a recurrence for this version of the product where again, $T(n)$ is the time it takes to multiply two n digit integers. Solve the recurrence.

3.4 Strassen's Algorithm

Suppose we want to multiply two square matrices each with size n by n. How fast can we achieve this? Let's start by looking at the matrix multiplication algorithm we learn in school.

$$\begin{bmatrix} 7 & 3 & 2 & 8 \\ 6 & 3 & 3 & 5 \\ 4 & 2 & 9 & 8 \\ 1 & 1 & 8 & 4 \end{bmatrix} \begin{bmatrix} 2 & 5 & 3 & 6 \\ 1 & 3 & 4 & 2 \\ 5 & 7 & 3 & 1 \\ 2 & 1 & 4 & 7 \end{bmatrix} =$$

Question 42. What is the total number of multiplications we have to do in this case? What is a lower bound on the time complexity?

$$\left(\begin{array}{c|c} A_1 & A_2 \\ \hline A_3 & A_4 \end{array}\right) \left(\begin{array}{c|c} B_1 & B_2 \\ \hline B_3 & B_4 \end{array}\right) \tag{20}$$

We will omit the details here, but Strassen's breakthrough algorithm showed that there was a way to recursively find the product of the full matrix. Instead of taking the product of the n by n matrices, the solution can be written as a combination of 7 different products of the submatrices A_i, B_i.

Question 43. Write down and solve the recurrence for Strassen's algorithm.

3.5 Quicksort

Quicksort is another divide and conquer sorting algorithm that is often preferred in practice due to its small constants hiding inside the $\Theta(n \log n)$ **expected** running time. Note that this is the expected running time, and in the **worst case** this algorithm can take $\Theta(n^2)$.

Here is the highlevel description for how quicksort works.

- **Divide**: Partition the input array into two subarrays based on a **pivot element**, such that the left subarray is filled with elements smaller than the pivot, and the right subarray is filled with elements larger than the pivot.
- **Conquer**: Call quicksort recursively on each of the halves.
- **Combine**: Combine the sorted subarrays and we are done!

```
def quick_sort(arr):
    n = len(arr)
    if n > 1:
        # Partition the array around a pivot, whose index is q
        q = partition(arr)
        arrL = quick_sort(arr[0:q])
        arrR = quick_sort(arr[q+1:n])
        return arrL + [arr[q]] + arrR
    else:
        return arr
```

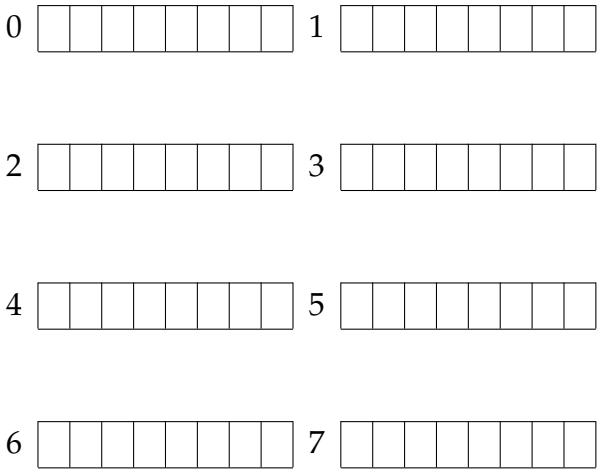
The "heavy lifting" of the algorithm is happening inside the partition function. A sample python implementation is given below, but essentially what we are doing is using the last element of the array as a **pivot**, and then deciding if each element is smaller or larger than this pivot element.

```
def partition(arr):
    pivot = arr[-1]
    leftend = 0
    for i in range(len(arr) - 1):
        if arr[i] < pivot:
            arr[leftend], arr[i] = swap(arr[leftend], arr[i])
            leftend += 1
    arr[leftend], arr[-1] = swap(arr[leftend], arr[-1])
    return leftend, arr
```

Question 44. Write down the state of the input array

$$[5, 2, 3, 7, 8, 1, 6, 4] \tag{21}$$

at the start of each iteration of the for loop in the partition function. Label the partition variable using p and the location of the leftend using l.



Question 45. What is the worst case partitioning that could occur in a run of quicksort? Write down the recurrence for this case and solve it to find the running time.

Question 46. Suppose that we have an input instance which partitions the array into two halves at each recursive call. Write down the recurrence for this case and solve it to find the running time.

An algorithm whose worst case runtime depends on the structure of the input is susceptible to adversarial attacks. That is, someone who knows the inner workings of the algorithm could force it to have a worst case running time every time. To avoid this, we often use randomness in our algorithms. Let's change our partition algorithm so that instead of picking the rightmost element, it chooses a random element and places it on the far right before running as usual.

To analyze the randomized version of quicksort, we will introduce some extra notation. For an input array A with n elements, let $z_1, z_2, \ldots, z_n$ be the n elements of A in sorted order.

Question 47. Let $A = [3, 2, 7, 4, 8]$. Write down z_1 through z_5.

We will also use the shorthand $Z_{i,j}$ to represent the subset of elements $\{z_i, z_{i+1}, \ldots, z_j\}$.

Question 48. Let $A = [6, 2, 3, 9, 10, 5]$. What is $Z_{2,5}$?

Let's now analyze the running time. To do this, we start by finding the probability that two elements will get compared in our algorithm.

Question 49. Let's consider the two endpoint elements, z_1 and z_n. What needs to happen in the partition function for the two elements to be compared? Alternatively, when do z_1 and z_n never get compared?

Question 50. Now consider two elements z_i and z_j such that $i < j$. What is the probability that these two elements get compared?

Question 51. Show that the expected running time of quicksort is $O(n \log n)$. In your analysis, use the indicator random variable

$$X_{ij} = \begin{cases} 1 \text{ if } z_i \text{ is compared with } z_j \\ 0 \text{ otherwise.} \end{cases} \tag{22}$$

You may also need the following fact:

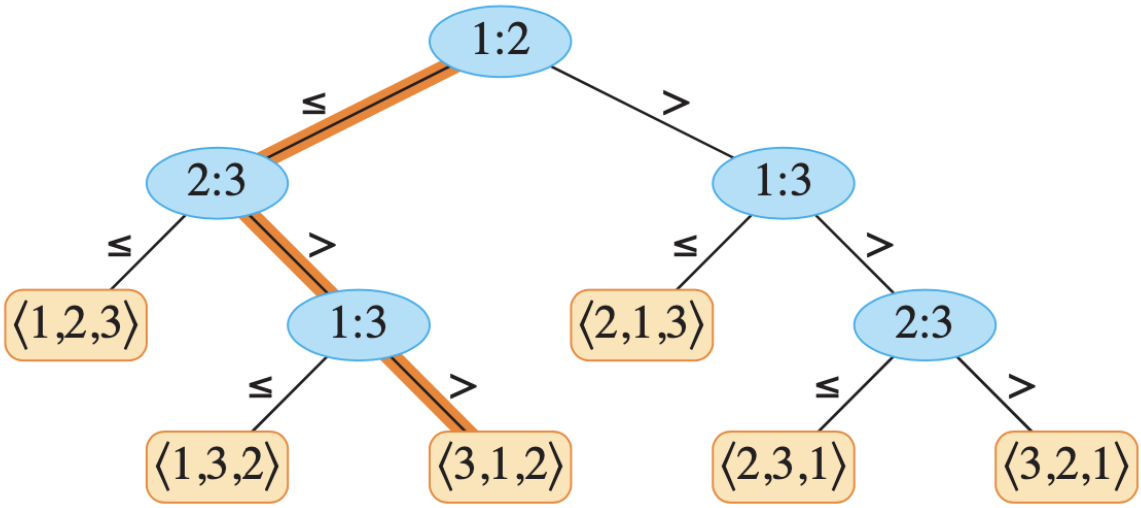
$$\sum_{k=1}^{n} \frac{1}{k} = O(\log n). \tag{23}$$

❖ Lower bounds for sorting

All the algorithms we have seen so far fall under the family of sorting algorithms called **comparison based sorting**. In these algorithms, we try to find clever ways to minimize the number of ways we compare a pair of elements. We were able to go from $\Theta(n^2)$ to $\Theta(n \log n)$, but can we do any better?

To see if we could, we will construct what is called a **decision tree**. This tree will represent some sequence of comparisons that our algorithm will do in order to distinguish between different permutations. For our sorting algorithm to be correct, we must be able to handle every permutation of inputs that appear.

Each node in our binary tree will represent some comparison between a pair of elements, and the leaves will represent all the possible permutations that are possible for our input array. The following figure is an example from the CLRS textbook.



A tree with N leaves will have a minimum height of $\log_2 N$, because $\log_2$ can be thought of as the number of times N can be halved until we get to 1.

Question 52. How many leaves are in a decision tree for comparison based sorting? What does this say about the minimum height of the tree?