Divide and Conquer Algorithms

- A general paradigm for algorithm design; inspired by emperors and colonizers.

1. Divide the problem into smaller problems.
2. Conquer by solving these problems.
3. Combine these results together.

Binary Search

- Search for x in sorted array A.
- If x is equal to the middle element of A, search is complete
- If x is less than the middle element of A, search on the left half of A
- Else, search on the right half of A

Time Complexity

- Let $T(n)$ denote the worst-case time to binary search in an array of length n.
- Recurrence is $T(n) = T(n/2) + O(1)$.
- $T(n) = O(logn)$

In [2]:
```python
def binarySearch(target: int, arr: list, left: int, right: int) -> int:
    if left > right:
        return -1

    middle = (left + right) // 2
    if target == arr[middle]:
        return middle
    elif target < arr[middle]:
        return binarySearch(target, arr, left, middle - 1)
    else: #target > arr[middle]
        return binarySearch(target, arr, middle + 1, right)

print(binarySearch(-1, list(range(10)), 0, 9))
print(binarySearch(10, list(range(10)), 0, 9))
print(binarySearch(5, list(range(10)), 0, 9))
```

Merge Sort

- Sort an unsorted array of numbers A
- If array is one element, return A
- Otherwise, recursively call mergesort on the left and right halves of A
- Then, merge the sorted result of the left and right haves of A

Time Complexity

- Let $T(n)$ denote the worst-case time to merge sortan array of length n.

- Recurrence is $T(n) = 2T(n/2) + O(n)$.
- $T(n) = O(nlogn)$

In [ ]:
```
# CODE
```

Multiplying Numbers

- We want to multiply two n-bit numbers. Cost is number of elementary bit steps.
- Grade school method has $O(n^2)$ cost: n^2 multiplies, $n^2/2$ additions, plus some carries.

Karatsuba's Algorithm

- Let X and Y be two n-bit numbers. Write $X = ab, Y = cd$ where ab and cd are concatenated to form an n-bit number.
- a, b, c, d are $n/2$ bit numbers. (Assume $n = 2^k$.)
$$XY = (a2^{n/2} + b)(c2^{n/2} + d) = ac2^n + (ad + bc)2^{n/2} + bd$$
- Note that $(a - b)(c - d) = (ac + bd) - (ad + bc)$.
- Solve 3 subproblems: $ac, bd, (a - b)(c - d)$.
- We can get all the terms needed for XY by addition and subtraction!

Time Complexity

- The recurrence for this algorithm is $T(n) = 3T(n/2) + O(n) = O(n^{log_2(3)})$.
- The complexity is $O(n^{log_2(3)}) = O(n^{1.59})$.

In [ ]:
```
# CODE
```

Recurence Solving

- Expand terms until a general formula is reached.
- Substitute for base case and solve.
- Can also use tree view with number of levels and work per level.
- Can solve by induction.

Master Method

- Recurrence in the form
$$T(n) = O(n^{log_b(a)}) + \sum_{i=0}^{log_b(n-1)} a^i f(\frac{n}{b^i})$$
- Let $f(n) = O(n^p log^k(n))$ where $p, k \geq 0$
- Condition: $a \geq 1, b > 1$ must be constant
- Case 1: $p < log_b a$ => $n^{log_b(a)}$ grows faster than $f(n)$. Thus, $T(n) = O(n^{log_b(a)})$.
- Case 2: $p = log_b a$ => both terms have same growth rates, thus $O(n^{log_b(a)} log^{k+1}(n))$
- Case 3: $p > log_b a$ => $n^{log_b(a)}$ grows slower than $f(n)$. Thus, $T(n) = O(f(n))$

Matrix Multiplication

- Multiply two $n \times n$ matrices: $C = A \times B$.

Traditional Algorithm

- Standard Method: $C[i][j] = \sum_{k=1}^{n} A[i][k] \times B[k][j]$
- For every element in C, it takes $O(n)$ computations.
- There are n^2 elements in C so it takes $O(n^3)$.

Strassen's Algorithm

- Let A, B be two $n \times n$ matrices.
- Divide matrices A, B, C into four $n/2 \times n/2$ submatrices.

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}; B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}; C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

- We can rewrite the product matrices as the following:

$$\begin{aligned} c_{11} &= a_{11} * b_{11} + a_{12} * b_{21} \\ c_{12} &= a_{11} * b_{12} + a_{12} * b_{22} \\ c_{21} &= a_{21} * b_{11} + a_{22} * b_{21} \\ c_{22} &= a_{21} * b_{12} + a_{22} * b_{22} \end{aligned}$$

- However, the recurrence for this relation listed below solves to $O(n^3)$:

$$T(n) = 8T(n/2) + O(n^2)$$

- Can reduce to seven multiplications using the following matrices:

$$\begin{aligned} P_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\ P_2 &= (a_{21} + a_{22})(b_{11}) \\ P_3 &= (a_{11})(b_{12} - b_{22}) \\ P_4 &= (a_{22})(b_{21} - b_{11}) \\ P_5 &= (a_{11} + a_{12})(b_{22}) \\ P_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\ P_7 &= (a_{12} - a_{22})(b_{21} + b_{22}) \end{aligned}$$

- We can rewrite the product matrices as the following:

$$\begin{aligned} c_{11} &= P_1 + P_4 - P_5 + P_7 \\ c_{12} &= P_3 + P_5 \\ c_{21} &= P_2 + P_4 \\ c_{22} &= P_1 + P_3 - P_2 + P_6 \end{aligned}$$

- The recurrence for this relation listed below solves to $O(n^{\log_2(7)}) = O(n^{2.81})$:

$$T(n) = 7T(n/2) + O(n^2)$$

Quicksort

- Simple, fast, and does not require extra space

Algorithm

- Partition among a pivot, splitting into elements smaller than the pivot, denoted L, and elements greater than the pivot, denoted R
- Sort L and R recursively
- Combine by appending R to L

Time Complexity

- $T(n)$ denotes the randomized runtime of Quicksort
- Each element randomly likely to be chosen as a pivot so there is $1/n$ probability that i is the pivot.
- Recurrence denoted by the following relation:

$$T(n) = 1/n * \sum_{i=1}^{n}(T(i-1) + T(n-1)) + n + 1$$

$$T(n) = 2/n * \sum_{i=1}^{n} T(i-1) + n + 1$$

$$T(n) = 2/n * \sum_{i=0}^{n-1} T(i) + n + 1$$

$$(1) : n * T(n) = 2 * \sum_{i=0}^{n-1} T(i) + n^2 + n$$

$$(2) : (n-1) * T(n-1) = 2 * \sum_{i=0}^{n-2} T(i) + (n-1)^2 + (n-1)$$

- Subtract (2) from (1) to arrive at the following:

$$n * T(n) = (n+1) * T(n-1) + 2n$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1}$$

$$\frac{T(n)}{n+1} = \frac{T(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

$$\frac{T(n)}{n+1} = \frac{T(2)}{3} + \sum_{i=3}^{n} \frac{2}{i}$$

$$\frac{T(n)}{n+1} = O(1) + 2\ln(n)$$

- Thus, $T(n) \leq 2(n+1)\ln(n)$, which is linearithmic.

In [ ]:
```
# CODE
```

Extrema Finding

- We can find the maximum and minimum in linear time with n comparisons.
- We can divide and conquer to find both the min and max in 3n/2 comparisons.

Min Algorithm

- Initialize current minimum to be the first element.
- Iterate through the rest of the elements; if any element is less than the current minimum, set it as the new current minimum.

Min Max Algorithm

- If the list A contains a single element, $min = max = A[0]$.
- Divide into two equal sublists A_1, A_2 and recursively find both the min and the max of both sublists. Then, return the more extreme of the two results for each min and max.

Time Complexity

- 2 calls on half the list + 2 comparisons has a recurrence of the following:

$$T(n) = 2T(n/2) + 2$$

Using the recurrence expansion method, we get...

$$T(n) = 2 * (2 * T(n/2^2) + 2) + 2 = 2^2 * T(n/2^2) + 2^2 + 2$$
$$T(n) = 2^2 * (2 * T(n/2^3) + 2) + 2^2 + 2 = 2^3 * T(n/2^3) + 2^3 + 2^2 + 2$$

...

$$T(n) = 2^i * T(n/2^i) + 2^i + ... + 2 = 2^i * T(n/2^i) + 2(2^{i-1} + ... + 2 + 1)$$
$$T(n) = 2^i * T(n/2^i) + 2(2^i - 1) = 2^i * T(n/2^i) + 2 * 2^i - 2$$

Use $T(2) = 1$. Then $n/2^i = 2$ when $i = \log_2 n/2$

Substitute i to get the recursion $T(n) = n/2 + 2 * n/2 - 2 = 3n/2 - 2$

In [3]:
```
def findMin(l: list) -> float:
    minimum = l[0]
    for element in l[1:]:
        minimum = element if element < minimum else minimum
    return minimum
print(findMin(list(range(10, 0, -1))))
```

```
1
```

In [6]:
```python
def minMax(l: list) -> tuple:
    if len(l) == 1:
        return (l[0], l[0])
    elif len(l) == 2:
        return (l[0], l[1]) if l[0] < l[1] else (l[1], l[0])
    else:
        half = len(l) // 2
        min1, max1 = minMax(l[:half])
        min2, max2 = minMax(l[half:])
        minimum = min1 if min1 < min2 else min2
        maximum = max1 if max1 > max2 else max2
        return (minimum, maximum)
print(minMax(list(range(20))))
```

```
(0, 19)
```

Linear Time Selection

- Find the item of rank k in the list (indexed 1 as smallest and n as largest).

Algorithm

- Divide items into $n/5$ groups of 5 each.
- Find the median of each group using sorting.
- Recursively find median of $n/5$ group medians.
- Partition using median-of-median, x, as a pivot.
- Let low side have s items and high side have $n - s$ items. If $k \leq s$, call this algorithm on the low side. Else, call this algorithm on the high side for rank $k - s$.

Correctness Proof

- The base case is trivial.
- If we call the low side, when $k \leq x$, we consider all items not in the quadrant greater than x. We use the inductive hypothesis to assume this recursion returns the correct result.
- Without loss of generality, we can apply this to the high side as well.

Time Complexity

- Recursively finding the group median is a recursive call of $T(n/5)$.
- Recrusively calling the low or high side is a recursive call of $T(7n/10)$ as there are $1/2 * n/5$ groups contributing at least 3 items to the opposite side.
- All other work can be done in linear time.
- The recurrence relation is the following:
$$T(n) \leq T(n/5) + T(7n/10) + O(n)$$
- We can inductively verify $T(n) \leq cn$ for some constant c:
$$T(n) \leq c(n/5) + c(7n/10) + O(n)$$
$$T(n) \leq (9/10)cn + O(n) \leq cn$$
$$T(n) \leq O(n) \leq cn/10$$
- Choose c so that $cn/10$ beats $O(n)$ for all n. Thus, $T(n) \leq cn$, meaning it runs in linear time.

In [ ]:
```
# CODE
```

Convex Hulls

- Smallest convex shape that contains a set of points

Algorithm

- Sort points by x-coordinates.
- Partition points into equal halves A (left) and B (right).
- Recursively compute the convex hull of A and B.
- Merge the convex hulls of A and B to arrive at the overall convex hull: start at the rightmost point a of A and leftmost point b of B; while a, b is not the lower tangent of the convex hulls of A and B: move A clockwise around points of A until it is a tangent of A, move b counter clockwise until it is a tangent of B. Then, repeat the process for the upper tangent in the reverse direction. Remove edges that were travelled in the rotation.

Correctness Proof

- Tangent of both objects does not cutoff any point
- Tangent of both objects also does not add any additional unnecessary space
- We explicitly check for tangent of both sides and remove unnecessary edges

Time Complexity

- Initial sorting takes $O(n \log(n))$.
- Recurrence = $T(n) = 2T(n/2) + O(n)$ with $O(n)$ for tangent merging.
- Recurrence solves to $O(n \log(n))$.

In [ ]:
```
# CODE
```