Java Algorithms

Divide and Conquer

1. **Multiplication**
    a. Gauss's idea
        i. Multiplication is expensive.
        ii. adding/subtracting is cheap.
        iii. 2 complex number: **a+bi**, **c+di**.
        iv. For example: **(a+bi)(c+di) = ac-bd + (bc+ad)i**. Try to make it less than 4 multiplications.
        v. Compute **ac**, **bd**, **(a+b)(c+d)**, then get **(a+bi)(c+di)**.

2. **Multiply n-bit integer**
    a. Input: **n**-bit integers **x** and **y**. Assume n is a power of **2**.
    b. Goal: compute **z = xy**, running time in terms of **n**.
    c. D&C idea: break input into 2 halves, generally the first and second halves are both **n/2** bits.
    d. For example:
        i. Number **x** = 182 = (10110110)
        ii. Break **x** into halves will be x_L = (1011) = 11, x_R = (0110) = 6.
        iii. 182 = **11***2^4+**6**. Therefore, the formula derived is $x = x_L*2^{n/2}+x_R$.
    e. Using the formula, $xy = (x_L*2^{n/2}+x_R)(y_L*2^{n/2}+y_R) = 2^n x_Ly_L+ 2^{n/2}(x_Ly_R+x_Ry_L) + x_R y_R$.
    f. Design
        i. `EasyMultiply`(**x**, **y**) computes **z = xy**.
        ii. **A** = `EasyMultiply`(x_Ly_L), **B** = `EasyMultiply`($x_R y_R$), **C** = `EasyMultiply`(x_Ly_R), **D** = `EasyMultiply`(x_Ry_L).
        iii. Return $z = 2^n*A + 2^{n/2}(C+D) + B$.
        iv. Runtime: $O(n^2)$.
    g. Better approach
        i. Compute **z = xy**.
        ii. Using the formula, $xy = (x_L*2^{n/2}+x_R)(y_L*2^{n/2}+y_R) = 2^n x_Ly_L+ 2^{n/2}(x_Ly_R+x_Ry_L) + x_R y_R$.
        iii. $A = x_Ly_L$, $B = x_Ry_R$, $C = (x_L + x_R)(y_L + y_R)_L$.
        iv. Result $z = 2^n*A + 2^{n/2}(C-A-B) + B$.
        v. Runtime: $O(n^{\log2(3)})$.
    h. Code example:

```
int multiply(String x, String y) {
    // Add zeros
```

```java
        String extra = "";
        for (int i = 0; i < Math.abs(y.length() - x.length()); i++) {
            extra += "0";
        }
        if (x.length() < y.length()) {
            x = extra + x;
        } else if (binaryX.length() > y.length()) {
            x = extra + y;
        }
        int n = x.length();

        // Base cases
        if (n == 0) return 0;
        if (n == 1) return Integer.valueOf(x)*Integer.valueOf(y);

        int fH = n/2;  // First half
        int sH = (n-fH); // Second half

        String xL = x.substring(0, fH);
        String xR = x.substring(fH);

        // Find the first half and second half of second string
        String yL = y.substring(0, fH);
        String yR = y.substring(fH);

        // Recursively calculate the 3 products of inputs of size n/2
        int A = multiply(xL, yL);
        int B = multiply(xR, yR);
        int C = multiply(addBinary(xL, xR), addBinary(yL, yR));

        // Combine the three products to get the final result.
        return A*(1<<(sH*2)) + (C - A - B)*(1<<(sH)) + B;
    }

    String addBinary(String x, String y) {
        if (x.length() == 0 && y.length() == 0) {
            return "";
        } else if (x.length() == 0){
            return y;
        } else if (y.length() == 0) {
            return x;
        }
        int num1 = Integer.parseInt(x, 2);
        int num2 = Integer.parseInt(y, 2);
        return Integer.toBinaryString(num1 + num2);
    }
```

3. **Median problem**
    a. Give an unsorted list **A** = **[a_1, …, a_n]** of **n** numbers.
    b. Goal: find the median of **A**.
    c. Easy algorithm
        i. Sort A and then output the **k**th element.
        ii. Merge sort takes **O(log(n))** time.
4. **Quick sort**
    a. Choose a pivot **p**.
    b. Partition **A** into **A < p**, **A = p**, **A > p**.
    c. Recursively sort **A < p** and **A > p**.
    d. For example: **A** = [5, 2, 20, 17, 11, 13, 8, 9, 11], **p** = 11
        i. **A < p:** [5, 2, 8, 9], **A = p:** [11, 11], **A > p:** [20, 17, 12].
        ii. Get new array = [5, 2, 8, 9, 11, 11, 20, 17, 12]
        iii. Recursively sort <u>index < 4</u> and <u>index ≥ 6</u>.
    e. Pivot **p** is good if [count of **A<p**] ≤ **3n/4** and [count of **A>p**] ≤ **3n/4**.
    f. Random pivots is fine. Always choose first or last element as pivot may cause worst case of **O(n^2)** in nearly sorted array.

```java
void quickSort(int[] array, int low, int high) {
   if (low < high) {
       int pivot = partition(array, low, high);
       quickSort(array, low, pivot-1);
       quickSort(array, pivot+1, high);
   }
}

private int partition(int[] arr, int low, int high) {
   int right = arr[high];
   int i = low - 1;
   for (int j = low; j < high; j++) {
       // If current element is smaller than or equal to pivot
       if (arr[j] <= right) {
           i++;
           // swap arr[i] and arr[j]
           int temp = arr[i];
           arr[i] = arr[j];
           arr[j] = temp;
       }
   }

   // swap arr[i+1] and arr[high] (or pivot)
   int temp = arr[i+1];
   arr[i+1] = arr[high];
```

```
        arr[high] = temp;

        return i+1;
    }
```

5. **Quick select**

    a. This uses the concept of quick sort but it finds the result in the process of quick sort.
    b. Runtime: **O(n)**.
    c. Code example:

```
    int quickSelect(int[] array, int left, int right, int k) {
        if (k > 0 && k <= right - left + 1) {
            // Partition the array around last element and get
            // position of pivot element in sorted array
            int p = partition(array, left, right);

            // If position is same as k
            if (pivot - left == k - 1)
                return array[pivot];

            // If position is more, recur for left subarray
            if (pivot - left > k - 1)
                return quickSelect(array, left, pivot - 1, k);

            return quickSelect(array, p + 1, right, k - p + left - 1);
        }

        return Integer.MAX_VALUE;
    }
```

6. **Geometric series**
    a. A geometric series **a, ar, ar^2, ….**
    b. The formula to calculate the sum of the series is a(1 – rn)/(1 – r).

7. **Polynomial multiplication**
    a. Polynomials **A(x)**, **B(x)**.
    b. **A(x)** = $a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1}$, **B(x)** = $b_0 + b_1x + b_2x^2 + \ldots + b_{n-1}x^{n-1}$.
    c. **C(x)** = **A(x)*B(x)** = $c_0 + c_1x + c_2x^2 + \ldots + c_{n-1}x^{n-1}$ where $c_k = a_0b_k + a_1b_{k-1} + \ldots + a_kb_0$.
    d. Problem need to be solved: **c = a*b** where * stands for **convolution**.
    e. Naive method: **O(k)** time for C_k, total time **O(n^2)**.
    f. Target: **O(n logn)** time algorithm.
    g. Multiplying polynomials is easy in **values** representative.

8. **Convolution applications**
    a. Filtering, mean filtering, Gaussian filtering
    b. Gaussian blur

9. **FFT - Fast Fourier Transform**
    a. Given **a** = $(a_0, a_1, \ldots, a_{n-1})$ for poly **A(x) =** $a_0 + a_1x + \ldots + a_{n-1}x^{n-1}$.
    b. Define
        i. $\mathbf{a_{even}} = (a_0, a_2, a_4, \ldots, a_{n-2})$.
        ii. $\mathbf{a_{odd}} = (a_1, a_3, a_5, \ldots, a_{n-1})$.
    c. Therefore,
        i. $\mathbf{A_{even}(y)} = a_0 + a_2y + a_4y^2 \ldots, a_{n-2}y^{n-2/2}$.
        ii. $\mathbf{A_{odd}(y)} = a_1 + a_3y + a_5y^2 \ldots, a_{n-1}y^{n-2/2}$.
    d. Then,
        i. $\mathbf{A(x) = A_{even}(x^2) + xA_{odd}(x^2)}$.
        ii. Degrees go down to n/2 - 1.
    e. Set ± property, $\mathbf{x_1, \ldots, x_n}$ are opposite of $\mathbf{x_{n+1}, \ldots, x_{2n}}$.
    f. $\mathbf{A(x_i) = A_{even}(x_i^2) + x_iA_{odd}(x_i^2)}$.
    g. $\mathbf{A(x_{n+i}) = A(-x_i) = A_{even}(x_i^2) - x_iA_{odd}(x_i^2)}$.
    h. Runtime: **O(n logn)**.
    i. Nth roots of unity
        i. $\mathbf{w_N^k = e^{jk2\pi}}$ **, k = 1, 2, 3, …, N - 1**.
        ii. $\mathbf{[\ w_N^k\ ]^N = [\ e^{jk2\pi}\ ]^N = 1}$.
        iii. Reference: https://ccrma.stanford.edu/~jos/mdft/Nth_Roots_Unity.html
    j. Reference: https://www.geeksforgeeks.org/fast-fourier-transformation-poynomial-multiplication/

10. **FFT(a, w) Algorithm**
    a. Input: coefficients $\mathbf{a = (a_0, a_1, \ldots, a_{n-1})}$ for poly **A(x)** when n is power of **2**.
    b. Output: $\mathbf{A(w^0), A(w), A(w^2), \ldots, A(w^{n-1})}$. Use $\mathbf{w = wn = (1, 2\pi/n) = e^{2\pi i/n}}$.
    c. If **n** = **1**, return **A(1)**.
    d. Partition
        i. Let $\mathbf{a_{even}}$ **= (a0, a2, a4, …, an-2)**.
        ii. Let $\mathbf{a_{odd}}$ **= (a1, a3, …, an-1)**.
        iii. Runtime: **O(n)**.
    e. Call $\mathbf{FFT(a_{even}, w^2)}$, get $\mathbf{A_{even}(w^0)}, \mathbf{A_{even}(w^2)}, \ldots, \mathbf{A_{even}(w^{n-2})}$.
    f. Call $\mathbf{FFT(a_{odd}, w^2)}$, get $\mathbf{A_{odd}(w^0)}, \mathbf{A_{odd}(w^2)}, \ldots, \mathbf{A_{odd}(w^{n-2})}$.
    g. If **n = 1**, return $\mathbf{a_0}$.
    h. For **j = 0 → n/2 - 1**: $\mathbf{A(w^j)} = \mathbf{A_{even}(w^{2j})} + \mathbf{A_{odd}(w^{2j})}$.

i. Runtime: $T(n) = 2T(n/2) + O(n) =$ **$O(n \log n)$**.
j. See source code here.

11. Poly Multiplication using FFT

a. Input: coefficients **a** = $(a_0, a_1, \ldots, a_{n-1})$ and **b**= $(b_0, \ldots, b_{n-1})$.
b. Output: coefficients **c** = $(c_0, c_1, \ldots, c_{2n-2})$ for $C(x) = A(x)*B(x)$.
c. $(r_0, r_1, \ldots, r_{2n-1})$ = **$FFT(a, w_{2n})$**.
d. $(s_0, s_1, \ldots, s_{2n-1})$ = **$FFT(b, w_{2n})$**.
e. For $j = 0$ to $2n-1$: $t_j = r_j * s_j$.
f. Have $C(x)$ at $2n^{th}$ roots of unity.
g. Inverse FFt
    i. $A = M_n(w_n)a = FFT(a, w_n)$.
    ii. $M_n(w_n)^{-1}A = a$
    iii. $M_n(w_n)^{-1} = 1/n\ M_n(w_n^{-1})$.
    iv. $w_n * w_{n-1} = 1$.
    v. Thus, **n**a $= M_n(w_n)^{-1}A = FFT(A, w_n^{n-1})$, a = **1/n** $FFT(A, w_n^{n-1})$.
h. $(c_0, c_1, \ldots, c_{2n-1}) = 1/2n\ FFT(t, w_{2n}^{2n-1})$.
i. See source code here.