CS218 Design and analysis of algorithms **Jan-Apr 2025**

Exercise sheet 2

Lecture 8, 9 Fibonacci, Integer Multiplication

1. Let us try to apply the divide and conquer approach on the integer multiplication problem. Suppose we want to multiply two n-bit integers a and b. Write them as

$$a = a_1 2^{n/2} + a_0$$

$$b = b_1 2^{n/2} + b_0$$

   The product of the two integers can be written as

$$ab = a_1 b_1 2^n + (a_1 b_0 + a_0 b_1) 2^{n/2} + a_0 b_0.$$

   Can you compute these three terms $a_1 b_1$, $a_1 b_0 + a_0 b_1$, $a_0 b_0$, using only **three multiplications of** $n/2$ **bit integers** and a few additions/subtractions? If yes, then we will get an $O(n^{1.58})$ time algorithm.

2. Can you find square of an n-bit integer a, using square subroutine on **five** $n/3$ bit integers and a few additions/subtractions? You need to compute the following five terms using the square operation only 5 times.

$$P^2, PQ, 2PR + Q^2, QR, R^2.$$

3. Can you find multiplication of two n-bit integers, using the multiplication subroutine on **six** pairs of $n/3$ bit integers and a few additions/subtractions? You need to compute the following five terms using the multiplication operation only six times.

$$a_0 b_0, a_1 b_0 + a_0 b_1, a_0 b_2 + a_1 b_1 + a_2 b_0, a_1 b_2 + a_2 b_1, a_2 b_2.$$

   Now, do this with only five multiplications.

4. Solve the recurrences

   - $T(n) = 6T(n/3) + O(n)$.
   - $T(n) = 5T(n/3) + O(n)$.
   - $T(n) = 8T(n/4) + O(n)$.
   - $T(n) = 7T(n/4) + O(n)$.

   Arrange the items in increasing order of complexity.

5. Can you find square of an n-bit integer a, using square subroutine on **2k-1** integers with n/k bits and a some additions/subtractions? What's the running time you get? What if you take k as something like $n/2$? Does that give you a really fast algorithm?

6. Show that $n2^{\sqrt{logn}}$ is asymptotically smaller than $n^{1.01}$. That is, show that there exists a number N, such that for all $n > N$,

$$n2^{\sqrt{logn}} < n^{1.01}.$$

   Do you think it works if we replace 1.01 with any constant greater than 1.

7. Write a program to compare different multiplication algorithms for multiplying 1024 bit integers. You can try the school method, Karatsuba, Toom-cook, or any combination of these.

8. **Matrix Multiplication.** Let us say we have 8 numbers $a_1, a_2, a_3, a_4, b_1, b_2, b_3, b_4$ and we consider these **seven** expressions.

$$p_1 = (a_1 + a_4)(b_1 + b_4), \quad p_2 = (a_3 + a_4)b_1, \quad p_3 = a_1(b_2 - b_4), \quad p_4 = a_4(b_3 - b_1)$$
$$p_5 = (a_1 + a_2)b_4, \quad p_6 = (a_3 - a_1)(b_1 + b_2), \quad p_7 = (a_2 - a_4)(b_3 + b_4)$$

(a) Compute the following four sums. This will be helpful later.

$$p_1 + p_4 - p_5 + p_7, \quad p_3 + p_5, \quad p_2 + p_4, \quad p_1 - p_2 + p_3 + p_6$$

Now, we want to apply divide and conquer technique to matrix multiplication. Let A and B be two $n \times n$ matrices, and we want to compute their product $C = A \times B$. The naive algorithm for this will take $O(n^3)$ arithmetic operations. We want to significantly improve this using divide and conquer.

A natural way to split any matrix can be this:

$$A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix},$$

where each A_i is an $n/2 \times n/2$ matrix.

(b) Can you express the product matrix C, in terms of A_1, A_2, A_3, A_4 and B_1, B_2, B_3, B_4.

(c) Design an algorithm for matrix multiplication using divide and conquer which takes $O(\mathbf{7}^{\log_2 n}) = O(n^{\log_2 7}) = O(n^{2.81})$ time.

Polynomial mulitplication, convolution, Fast Fourier transform

9. Can you apply Karatsuba's trick on polynomial multiplication and get a runtime bound better than $O(d^2)$ for degree d polynomials.

10. Given probability distributions of two (discrete) random variables, we want to compute the probability distribution for the sum of the two random variables. Do you see how convolution can be used to compute this distribution.

11. **Implementation of FFT.** It is possible to do an iterative implementation of FFT, which is "in-place". That is, we can just work on the input array of length d and need only constant size extra memory. Can you think about such an implementation?

12. Let $P(x) = a_0 + a_1 x + \cdots + a_{d-1}x^{d-1}$ be a degree $d-1$ polynomial. Let the dth roots of unity be $\omega^0, \omega^1, \ldots, \omega^{d-1}$. Let the evaluations of $P(x)$ on the dth roots of unity be $e_0, e_2, \ldots, e_{d-1}$. That is, for each $0 \leq i \leq d-1$
$$e_i = P(\omega^i) = a_0 + a_1\omega^i + \cdots + a_{d-1}\omega^{(d-1)i}.$$
Define a new polynomial $Q(y)$ as $e_0 + e_1 x + \cdots + e_{d-1}x^{d-1}$. Prove that for each $0 \leq i \leq d-1$
$$Q(\omega^{-i}) = e_0 + e_1\omega^{-i} + \cdots + e_{d-1}\omega^{-(d-1)i} = da_i.$$
Hint: Useful facts to prove, $\omega^{-i} = \omega^{d-i}$, For any $1 \leq i \leq d-1$, $\sum_{j=0}^{d-1} \omega^{ij} = 0$.

13. Suppose we want to evaluate $P(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3$ at the fourth roots of unity $1, -1, i, -i$. Consider how FFT algorithm will compute these. We will represent the algorithm as a circuit, with the a gate shown in figure 1. The gate labeled with α takes two numbers a and b (in that order) as inputs and outputs two numbers $a + \alpha b$ and $a - \alpha b$ (in that order).

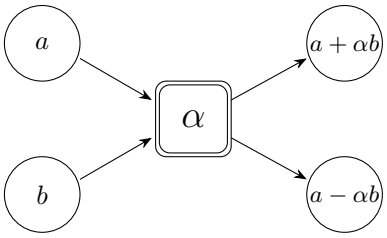

Figure 1: The operation that forms the building block of the FFT algorithm, represented as a gate labeled with a root of unity α

The FFT circuit shown in Figure 2 shows the computation for evaluations of a degree 3 polynomial at 4th roots of unity. Fill in the empty circles with the appropriate input values and intermediate values.

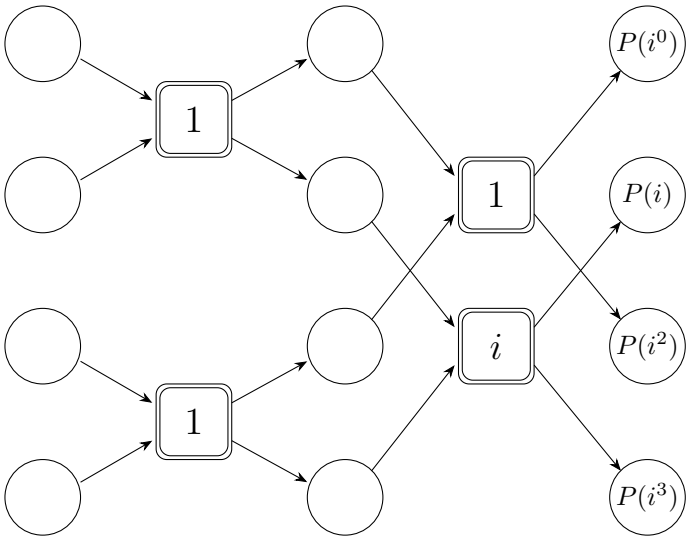

Figure 2: The FFT algorithm for a degree 3 polynomial $P(x)$ evaluated at 4th roots of unity

The FFT circuit shown in Figure 3 shows the computation for evaluations of a degree 7 polynomial at 8th roots of unity. Fill in the empty circles with the appropriate input values and intermediate values.

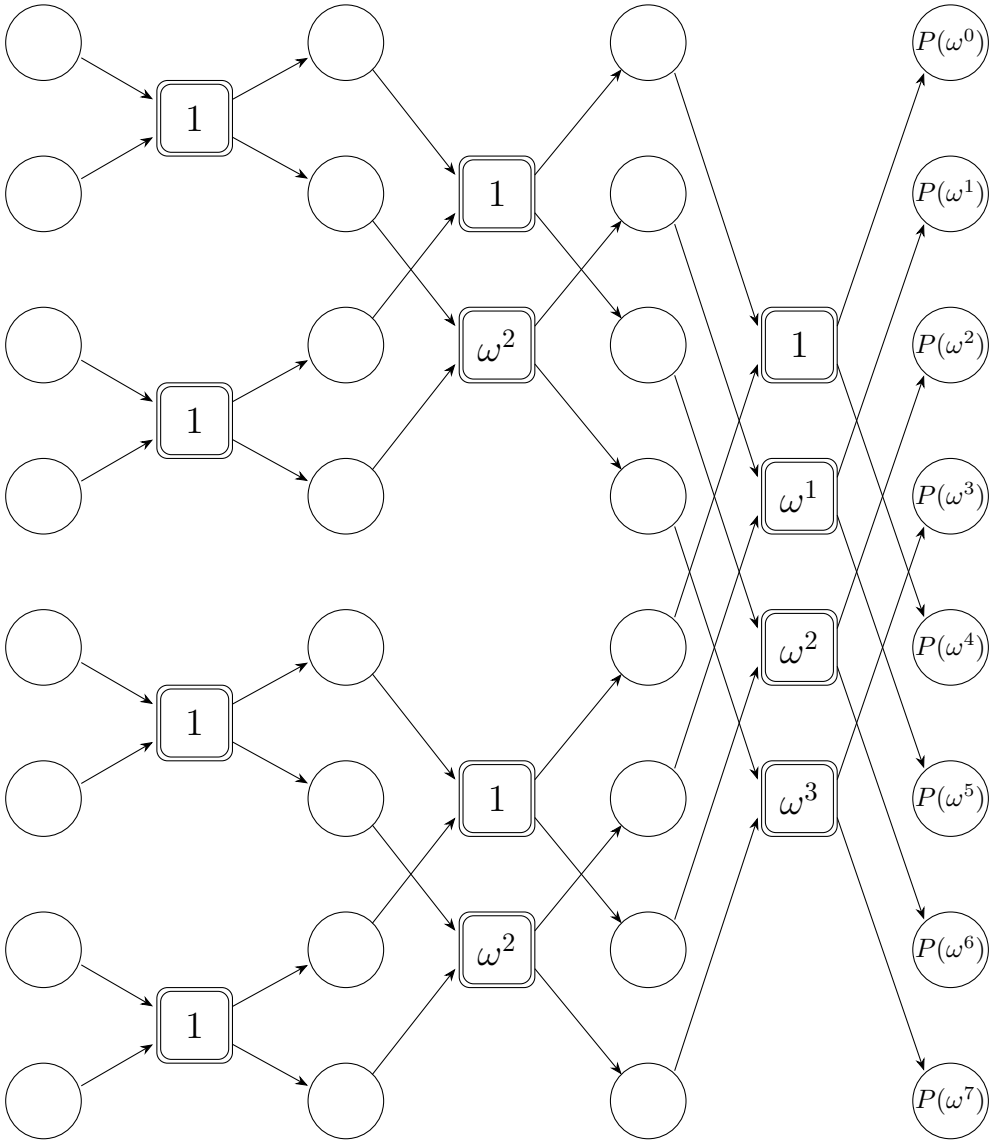

Figure 3: The FFT algorithm for a degree 7 polynomial $P(x)$ evaluated at 8th roots of unity. Here ω is the primitive 8th root of unity.

4. You've been working with some physicists who need to study, as part of their experimental design, the interactions among large numbers of very small charged particles. Basically, their setup works as follows. They have an inert lattice structure, and they use this for placing charged particles at regular spacing along a straight line. Thus we can model their structure as consisting of the points $\{1, 2, 3, \ldots, n\}$ on the real line; and at each of these points j, they have a particle with charge q_j. (Each charge can be either positive or negative.)

They want to study the total force on each particle, by measuring it and then comparing it to a computational prediction. This computational part is where they need your help. The total net force on particle j, by Coulomb's Law, is equal to

$$F_j = \sum_{i<j} \frac{Cq_iq_j}{(j-i)^2} - \sum_{i>j} \frac{Cq_iq_j}{(j-i)^2}$$

They've written the following simple program to compute F_j for all j:

```
For j = 1, 2, ..., n
  Initialize F_j to 0
  For i = 1, 2, ..., n
    If i < j then
      Add C q_i q_j / (j - i)^2 to F_j
    Else if i > j then
      Add -C q_i q_j / (j - i)^2 to F_j
    Endif
  Endfor
  Output F_j
Endfor
```

It's not hard to analyze the running time of this program: each invocation of the inner loop, over i, takes $O(n)$ time, and this inner loop is invoked $O(n)$ times total, so the overall running time is $O(n^2)$.

The trouble is, for the large values of n they're working with, the program takes several minutes to run. On the other hand, their experimental setup is optimized so that they can throw down n particles, perform the measurements, and be ready to handle n more particles within a few seconds. So they'd really like it if there were a way to compute all the forces F_j much more quickly, so as to keep up with the rate of the experiment.

Help them out by designing an algorithm that computes all the forces F_j in $O(n \log n)$ time.

Figure 4: Kleinberg Tardos: Divide and Conquer Chapter 5, Exercise 4.