COMP3121/9101
Algorithm Design

Problem Set 2 – Divide and Conquer

[**K**] – key questions [**H**] – harder questions [**E**] – extended questions [**X**] – beyond the scope of this course

Contents

§ SECTION ONE: RECURRENCES

[K] Exercise 1. Determine the asymptotic growth rate of the solutions to the following recurrences. You may use the Master Theorem if it is applicable.

(a) $T(n) = 2T(n/2) + n(2 + \sin n)$.

(b) $T(n) = 2T(n/2) + \sqrt{n} + \log n$.

(c) $T(n) = 8T(n/2) + n^{\log_2 n}$.

(d) $T(n) = T(n-1) + n$.

Solution. (a) Here, we realise that $a = 2$, $b = 2$, and $f(n) = n(2 + \sin n)$. But $\sin n \leq 1$ for all n and so, $f(n) = \Theta(n)$. The critical exponent is

$$c^* = \log_b a = \log_2 2 = 1.$$

Thus, the second case of the Master Theorem applies and we get

$$T(n) = \Theta(n \log n).$$

(b) Again, we repeat the same process. We realise that $a = 2$, $b = 2$, and $f(n) = \sqrt{n} + \log n$. So, the critical exponent is $c^* = 1$. Since $\log n$ *eventually* grows slower than $\sqrt{n}$, we have that

$$f(n) = \Theta(\sqrt{n}). = \Theta\left(n^{1/2}\right).$$

This implies that

$$f(n) = O(n^{0.9}) = O(n^{c^* - 0.1}),$$

so the first case of the Master Theorem applies and we obtain $T(n) = \Theta(n)$.

(c) Here, $a = 8$, $b = 2$, and $f(n) = n^{\log_2 n}$. So the critical exponent is

$$c^* = \log_b a = \log_2 8 = 3.$$

On the other hand, for large enough n, we have that $\log_2 n \geq 4$. So

$$f(n) = n^{\log_2 n} = \Omega(n^4).$$

Consequently,

$$f(n) = \Omega(n^{c^*+1}).$$

To be able to use the third case of the Master Theorem, we have to show that for some $0 < c < 1$, the following holds for sufficiently large n:

$$af\left(\frac{n}{b}\right) < cf(n).$$

In our case, this translates to

$$8\left(\frac{n}{2}\right)^{\log_2 \frac{n}{2}} < cn^{\log_2 n}.$$

Now, we have

$$\begin{aligned} 8\left(\frac{n}{2}\right)^{\log_2 \frac{n}{2}} &= 8\left(\frac{n}{2}\right)^{\log_2 n - \log_2 2} \\ &= 8\left(\frac{n}{2}\right)^{\log_2 n - 1} \\ &< 8\left(\frac{n}{2}\right)^{\log_2 n} \\ &= \frac{8n^{\log_2 n}}{2^{\log_2 n}} \\ &= \frac{8}{n} n^{\log_2 n}. \end{aligned}$$

If $n > 16$, then

$$8\left(\frac{n}{2}\right)^{\log_2 \frac{n}{2}} < \frac{1}{2} n^{\log_2 n},$$

so the required inequality is satisfied with $c = \frac{1}{2}$ for all $n > 16$. Therefore, by case 3 of the Master Theorem, the solution is

$$T(n) = \Theta(f(n)) = \Theta\left(n^{\log_2 n}\right).$$

(d) Note that the Master Theorem does not apply. However, we can alter the proof of the Master Theorem to obtain the solution to the recurrence. For every k, we have $T(k) = T(k-1) + k$. So unrolling the recurrence, we obtain

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= [T(n-2) + (n-1)] + n \\ &= T(n-2) + (n-1) + n \\ &= [T(n-3) + (n-2)] + (n-1) + n \\ &= \cdots \\ &= T(1) + (n-(n-2)) + (n-(n-3)) + \cdots + (n-1) + n \\ &= T(1) + (2 + 3 + \cdots + n) \\ &= T(1) + \frac{n(n+1)}{2} - 1 \\ &= \Theta(n^2). \end{aligned}$$

□

[K] Exercise 2. Explain why we cannot apply the Master Theorem to the following recurrences.

(a) $T(n) = 2^n T(n/2) + n^n$.

(b) $T(n) = T(n/2) - n^2 \log n$.

(c) $T(n) = \frac{1}{3} T(n/3) + n$.

(d) $T(n) = 3T(3n) + n$.

Solution. (a) The Master Theorem requires the value of a to be constant. Here, the value of $a = 2^n$ depends on the input size which is non-constant.

(b) The Master Theorem requires $f(n)$ to be non-negative. We can see that, for $n \in \mathbb{N}$, $f(n) \leq 0$. So, the Master Theorem cannot be applied.

(c) The Master Theorem requires $a \geq 1$. Here, $a = 1/3$ and so, the conditions of the Master Theorem are not met.

(d) The Master Theorem requires $b > 1$. However, we see that we can write $T(n)$ as

$$T(n) = 3T\left(\frac{n}{1/3}\right) + n;$$

in this case, we see that $b = 1/3 < 1$. So the conditions of the Master Theorem are not met.

□

[H] Exercise 3. Consider the following naive Fibonacci algorithm.

Algorithm 1 $F(n)$: The naive Fibonacci algorithm

Require: $n \geq 1$
  if $n = 1$ or $n = 2$ **then**
    return 1
  else
    return $F(n-1) + F(n-2)$
  end if

When analysing its time complexity, this yields us with the recurrence $T(n) = T(n-1) + T(n-2)$. Show that this yields us with a running time of $\Theta(\varphi^n)$, where $\varphi = \frac{1+\sqrt{5}}{2}$ which is the golden ratio. How do you propose that we improve upon this running time?

Hint: This is a standard recurrence relation. Guess $T(n) = a^n$ and solve for a.

Solution. We solve $T(n) = T(n-1) + T(n-2)$. Using the standard trick of solving recurrence relations, we guess $T(n) = a^n$ for some a. Then this produces the recurrence:

$$a^n = a^{n-1} + a^{n-2}.$$

Dividing both sides by a^{n-2}, we obtain the quadratic equation:

$$a^2 - a - 1 = 0.$$

Then the quadratic equation yields

$$a = \frac{1 \pm \sqrt{5}}{2}.$$

In other words, we obtain

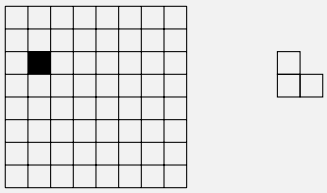
$$T(n) = A \cdot \left(\frac{1+\sqrt{5}}{2}\right)^n + B \cdot \left(\frac{1-\sqrt{5}}{2}\right)^n = \Theta\left(\varphi^n\right),$$

where $\varphi = \frac{1+\sqrt{5}}{2}$.

The inefficiency comes from the fact that we are making unnecessary and repeated calls to problems that we have already seen before. For example, to compute $F(n)$, we compute $F(n-1)$ and $F(n-2)$. To compute $F(n-1)$, we compute $F(n-2)$ again. Because these values never change, a way to improve the overall running time is to *cache* the results that we have computed previously and only make $O(1)$ calls for results that we have already computed. This reduces our complexity dramatically from exponential to $O(n)$. This is an example of *dynamic programming*, something we will see later down the track. □

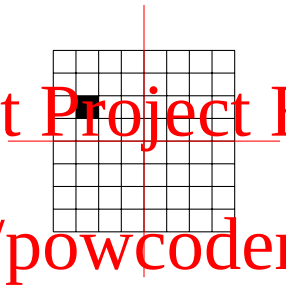
§ SECTION TWO: DIVIDE AND CONQUER

[K] Exercise 4. You are given a $2^n \times 2^n$ board with one of its cells missing (i.e., the board has a hole). The position of the missing cell can be arbitrary. You are also given a supply of "trominoes", each of which can cover three cells as below.

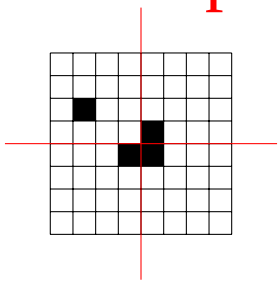


Design an algorithm that covers the entire board (except for the hole) with these "trominoes". Note that the trominoes can be rotated and your solution should not try to brute force all possible placement of those "trominoes".

Solution. We proceed by a divide and conquer recursion; thus, we split the board into 4 equal parts:



We can now apply our recursive procedure to the top left board which has a missing square. To be able to apply recursion to the remaining 3 squares we place a domino at the centre as shown below.



We treat the covered squares in each of the three pieces as a missing square and can proceed by recursion applied on all 4 squares whose sides are half the size of the size of the original square. □

[K] Exercise 5. Given positive integers M and n, compute M^n using only $O(\log n)$ many multiplications.

Solution. Note that

$$M^n = \begin{cases} \left(M^{\frac{n}{2}}\right)^2 & \text{if } n \text{ is even,} \\ \left(M^{\frac{n-1}{2}}\right)^2 \times M & \text{if } n \text{ is odd.} \end{cases}$$

Hence, we can proceed by divide and conquer. If n is even, we recursively compute $M^{\frac{n}{2}}$ and then square it. If n is odd, we recursively compute $M^{\frac{n-1}{2}}$, square it and then multiply by another M. Since n is (approximately) halved in each recursive call, there are at most $O(\log n)$ recursive calls, and since we perform only one or two multiplications in each recursive call, the algorithm performs $O(\log n)$ many multiplications, as required.

Alternative solution: Any positive integer n can be uniquely written in base 2, and therefore can be uniquely written as the sum of distinct powers of 2. Thus, M^n can be written the product of powers of M where the index is itself a power of 2, i.e. $M, M^2, M^4, M^8, \ldots$. We can obtain these powers of M in $O(\log n)$ time by repeated squaring, and then multiply together the appropriate powers to get M^n. The appropriate powers to multiply are the powers M^{2^i} such that the ith least significant bit of the binary representation of n is 1. For example, to obtain M^{11}, the binary representation of 11 is $(1011)_2$, and hence we should multiply together M, M^2, and M^8. □

[H] Exercise 6. You and a friend find yourselves on a TV game show! The game works as follows. There is a **hidden** $N \times N$ table A. Each cell $A[i, j]$ of the table contains a single integer and no two cells contain the same value. At any point in time, you may ask the value of a single cell to be revealed. To win the big prize, you need to find the N cells each containing the **maximum** value in its row. Formally, you need to produce an array $M[1..N]$ so that $A[r, M[r]]$ contains the maximum value in row r of A, i.e., such that $A[r, M[r]]$ is the largest integer among $A[r, 1], A[r, 2], \ldots, A[r, N]$. In addition, to win, you should ask at most $2N\lceil \log N \rceil$ many questions. For example, if the hidden grid looks like this:

	Column 1	Column 2	Column 3	Column 4
Row 1	10	5	8	**3**
Row 2	1	**9**	7	6
Row 3	-3	**4**	-1	0
Row 4	-10	-9	-8	**2**

then the correct output would be $M = [1, 2, 2, 4]$.

Your friend has just had a go, and sadly failed to win the prize because they asked N^2 many questions which is too many. However, they whisper to you a hint: they found out that M is **non-decreasing**. Formally, they tell you that $M[1] \leq M[2] \leq \cdots \leq M[N]$ (this is the case in the example above).

Design an algorithm which asks at most $\mathbf{O(N \log N)}$ many questions that produces the array M correctly, even in the very worst case.

Solution. We first find $M[N/2]$. We can do this in N questions by simply examining each element in row $N/2$, and finding the largest one.

Suppose $M[N/2] = x$. Then, we know that $M[i] \leq x$ for all $i < N/2$ and that $M[j] \geq x$ for all $j > N/2$. Thus, we can recurse to solve the same problem on the sub-grids

$$A\left[1..\left(\frac{N}{2} - 1\right)\right][1..x]$$

and

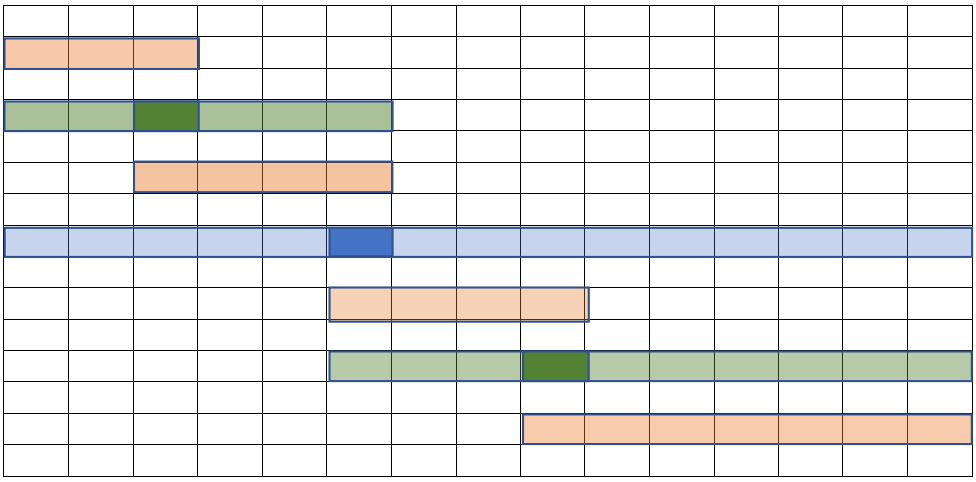
$$A\left[\left(\frac{N}{2} + 1\right)..N\right][x..N].$$

It remains to show that this approach uses at most $2N\lceil \log N \rceil$ questions.

Claim. At each stage of recursion, at most two cells of any column are investigated.

Solution. We observe the following.

- In the first call of the recursion, only one cell of each column has been investigated. These cells are marked in blue, with the maximum in dark blue.



- In the second call of the recursion, we investigate two cells of one column, and one cell of each of the other columns. These cells are marked in green, with the maxima in dark green.
- In the third call of the recursion, we investigate two cells in each of three columns, and one cell of each of the other columns. These cells are marked in orange.

Since the investigated ranges overlap only at endpoints, the claim holds true. □

So in each recursion call, at most $2N$ cells were investigated. The search space decreases by at least half after each call, so there are at most $\lceil \log_2 N \rceil$ many recursion calls. Therefore, the total number of cells investigated is at most $2N \lceil \log_2 N \rceil$.

Additional exercise: using the above reasoning, try to find a sharper bound for the total number of cells investigated. □

[H] Exercise 7. Define the Fibonacci numbers as

$$F_0 = 0, F_1 = 1, \text{ and } F_n = F_{n-1} + F_{n-2} \text{ for all } n \geq 2.$$

Thus, the Fibonacci sequence is as follows:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots.$$

(a) Show, by induction or otherwise, that

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

for all integers $n \geq 1$.

(b) Hence or otherwise, give an algorithm that finds F_n in $O(\log n)$ time.

Solution. (a) We proceed by induction. If $n = 1$, then we have

$$\begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix},$$

as required.

Let $n \geq 2$. Now, suppose that

$$\begin{pmatrix} F_{(n-1)+1} & F_{n-1} \\ F_{n-1} & F_{(n-1)-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1}.$$

Then

$$\begin{aligned} \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} &= \begin{pmatrix} F_n + F_{n-1} & F_{n-1} + F_{n-2} \\ F_{n-1} + F_{n-2} & F_{n-2} + F_{n-3} \end{pmatrix} \\ &= \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} && \text{(inductive hypothesis)} \\ &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n}. \end{aligned}$$

This finishes the induction proof.

(b) Let $G = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. We can compute $G^2 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^2$ in $O(1)$ since each element in the matrix is just a sum and product of small numbers. Likewise, we can also compute powers of G that are powers of two; in other words, we compute the matrices: $G, G^2, G^4, \ldots$.

To determine F_n, we need to find which combination of G^{2^ℓ} matrices we want to take. Thus, we can write n in its binary representation; that is, write

$$n = a_0 2^0 + a_1 2^1 + a_2 2^2 + \ldots.$$

Each a_i term tells us precisely which matrices of the form G^{2^i} we want to multiply to obtain a final matrix H. The top right (or bottom left) of H then tells us precisely what F_n. To argue the time complexity, we observe that there are maximally $\lfloor \log_2 n \rfloor$ to consider when writing down the binary representation of n. Therefore, we are performing $\log n$ many constant time operations, which give us $O(\log n)$ time complexity.

□

[H] Exercise 8. Let A be an array of n integers, not necessarily distinct or positive. Design a $\Theta(n \log n)$ algorithm that returns the maximum sum found in any contiguous subarray of A.

Note: there is an $O(n)$ solution that solves the problem; however, the intuition behind that approach is much more involved and will be taught in due time.

Solution. Note that brute force takes $\Theta(n^2)$.

The key insight to this problem is the following; once we split the original array in half, the maximum subarray must occur in one of the following cases:

- the maximum subarray sum occurs *entirely* in the left half of the original array.
- the maximum subarray sum occurs *entirely* in the right half of the original array.
- the maximum subarray sum overlaps across both halves.

We can compute the first two cases simply by making recursive calls on the left and right halves of the array. The third case can be computed in linear time. We now fill in the details of the algorithm.

Split the array into the left half and right half at the midpoint m.

- We can compute the maximum subarray sum in the left half using recursive calls on the left half of the array $A[1..m]$.
- Similarly, we can compute the maximum subarray sum in the right half using recursive calls on the right half of the array $A[(m+1)..n]$.
- To compute the maximum sum in the case where the subarray overlaps across both halves, we think of such a subarray as containing a portion from the left half and a portion from the right half. Now, we
  - find the maximum sum of any subarray of the form $A[l..m]$ by scanning left from m, and
  - similarly maximise the sum $A[(m+1)..r]$ by scanning right from $m+1$.

  Adding these two results gives the maximum sum of a subarray of the form $A[l..r]$ where $l \leq m < r$ as required.
- Our algorithm then returns the maximum of these three cases.
  - One caveat is that, if the maximum sum is negative, then our algorithm should return 0 to indicate that choosing an *empty* subarray is best.

To compute the time complexity of the algorithm, note that we're splitting our problem from size N to two $N/2$ problems and doing an $O(n)$ overhead to combine the solutions together. This gives us the recurrence relation

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

This falls under Case 2 of the Master Theorem, so $T(n) = \Theta(n \log n)$ is the total running time of the algorithm. □

[H] Exercise 9. Suppose you are given an array A containing $2n$ numbers. The only operation that you can perform is make a query if element $A[i]$ is equal to element $A[j]$, $1 \leq i, j \leq 2n$. Your task is to determine if there is a number which appears in A at least n times using an algorithm which runs in linear time.

Hint: The reasoning resembles a little bit like the celebrity problem from Tutorial 1; compare them in pairs. How many potential candidates can exist that appears in A at least n times?

Solution. Create two arrays B and C, initially both empty. Repeat the following procedure n times:

- Pick any two elements of A and compare them.
  - If the numbers are different, discard both of them.
  - If instead the two numbers are equal, append one of them to B and the other to C.

Claim. If a value x appears in at least half the entries of A, then the same value also appears in at least half the entries of B.

Proof. Suppose such a value x exists. In a pair of distinct numbers, at most one of them can be x, so after discarding both, x still makes up at least half the remaining array elements. Repeating this, x must account for at least half the values appearing in B and C together. But B and C are identical, so at least the entries of B are equal to x.

We can now apply the same algorithm to B, and continue reducing the instance size until we reach an array of size two. The elements of this array are the only candidates for the original property, i.e., the only values that could have appeared n times in the original array.

There is a special case; B and C could be empty after the first pass. In this case, the only candidates for the property are the values which appear in the last pair to be considered. For each of these, perform a linear scan to find their frequency in the original array, and report the answer accordingly.

Finally we analyse the time complexity. Clearly, each step of the procedure repeated n times above takes constant time, so we can reduce the problem from array A (of length $2n$) to array B (of length $\leq n$) in $\Theta(n)$ time. Letting the

instance size be *half* the length of the array (to more neatly fit the Master Theorem), we can express the worst case using the recurrence

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n).$$

Now $a\,f(n/b) = f(n/2)$, which is certainly less than say $0.9f(n)$ for large n as f is linear. By case 3 of the Master Theorem, the time complexity is $T(n) = \Theta(f(n)) = \Theta(n)$, as required. □

§ SECTION THREE: KARATSUBA'S TRICK

[K] Exercise 10. Recall that the product of two complex numbers is given by

$$\begin{aligned}(a+ib)(c+id) &= ac + iad + ibc + i^2bd \\ &= (ac - bd) + i(ad + bc)\end{aligned}$$

(a) Multiply two complex numbers $(a+ib)$ and $(c+id)$ (where a, b, c, d are all real numbers) using only 3 real number multiplications.

(b) Find $(a+ib)^2$ using only two multiplications of real numbers.

(c) Find the product $(a+ib)^2(c+id)^2$ using only five real number multiplications.

Solution. (a) This is again the Karatsuba trick:

$$\begin{aligned}(a+ib)(c+id) &= ac - bd + (bc + ad)i \\ &= ac - bd + [(a+b)(c+d) - ac - bd]\, i,\end{aligned}$$

so we need only 3 multiplications: ac and bd and $(a+b)(c+d)$.

(b) We again expand and refactorise:

$$\begin{aligned}(a+ib)^2 &= a^2 - b^2 + 2abi \\ &= (a+b)(a-b) + (a+a)bi.\end{aligned}$$

(c) Observe that

$$(a+ib)^2(c+id)^2 = [(a+ib)(c+id)]^2.$$

Therefore, we can now use (a) to multiply $(a+ib)(c+id)$ with only three multiplications and then use (b) to square the result with two additional multiplications.

□

[K] Exercise 11. Let $P(x) = a_0 + a_1x$ and $Q(x) = b_0 + b_1x$, and define $R(x) = P(x) \cdot Q(x)$. Find the coefficients of $R(x)$ using only three products of pairs of expressions each involving the coefficients a_i and/or b_j.

Addition and subtraction of the coefficients do not count towards this total of three products, nor do scalar multiples (i.e. products involving a coefficient and a constant).

Solution. We use (essentially) the Karatsuba trick:

$$\begin{aligned}R(x) &= (a_0 + a_1x)(b_0 + b_1x) \\ &= a_0b_0 + (a_0b_1 + b_0a_1)x + a_1b_1x^2 \\ &= a_0b_0 + [(a_0 + a_1)(b_0 + b_1) - a_0b_0 - a_1b_1]\, x + a_1b_1x^2.\end{aligned}$$

Note that the last expression involves only three multiplications: a_0b_0, a_1b_1 and $(a_0 + a_1)(b_0 + b_1)$. □

[H] Exercise 12. Let $P(x) = a_0 + a_1x + a_2x^2 + a_3x^3$ and $Q(x) = b_0 + b_1x + b_2x^2 + b_3x^3$, and define $R(x) = P(x) \cdot Q(x)$. You are tasked to find the coefficients of $R(x)$ using only twelve products of pairs of expressions each involving the coefficients a_i and/or b_j.

Solution. We again use the Karatsuba trick:

$$\begin{aligned}
R(x) &= (a_0 + a_1x + a_2x^2 + a_3x^3)(b_0 + b_1x + b_2x^2 + b_3x^3) \\
&= (a_0 + a_1x)(b_0 + b_1x) + (a_0 + a_1x)(b_2 + b_3x)x^2 \\
&+ (a_2 + a_3x)(b_0 + b_1x)x^2 + (a_2 + a_3x)(b_2 + b_3x)x^4 \\
&= a_0b_0 + [(a_0 + a_1)(b_0 + b_1) - a_0b_0 - a_1b_1]\,x + a_1b_1x^2 \\
&+ \left\{a_0b_2 + [(a_0 + a_1)(b_2 + b_3) - a_0b_2 - a_1b_3]\,x + a_1b_3x^2\right\}x^2 \\
&+ \left\{a_2b_0 + [(a_2 + a_3)(b_0 + b_1) - a_2b_0 - a_3b_1]\,x + a_3b_1x^2\right\}x^2 \\
&+ \left\{a_2b_2 + [(a_2 + a_3)(b_2 + b_3) - a_2b_2 - a_3b_3]\,x + a_3b_3x^2\right\}x^4.
\end{aligned}$$

Note that each of the last four lines involves only three multiplications. Expanding and regrouping will yield the coefficients of $R(x)$. □

§ SECTION FOUR: CONVOLUTIONS

[K] Exercise 13. Find the sequence x satisfying $x * \langle 1, 1, -1\rangle = \langle 1, 0, -1, 2, -1\rangle$. Is it true that $\langle 1, 1, -1\rangle * x = \langle 1, 0, -1, 2, -1\rangle$? Explain why or why not.

Solution. We note that the sequence $\langle 1, 1, -1\rangle$ corresponds to the polynomial $Q(y) = 1 + y - y^2$ and the sequence $\langle 1, 0, -1, 2, -1\rangle$ corresponds to the polynomial $R(y) = 1 - y^2 + 2y^3 - y^4$. We, therefore, want to find a polynomial $P(y)$ such that $P(y) \cdot Q(y) = R(y)$. Since R is a degree 4 polynomial with Q being a degree 2 polynomial, then P must be a degree 2 polynomial, which means that x contains three elements in its sequence. Thus, let $x = \langle a_0, a_1, a_2\rangle$ which corresponds to the polynomial $P(y) = a_0 + a_1 y + a_2 y^2$. Therefore, we have that

$$\begin{aligned} 1 - y^2 + 2y^3 - y^4 &= (a_0 + a_1 y + a_2 y^2)(1 + y - y^2) \\ &= a_0 + (a_0 + a_1)y + (a_2 + a_1 - a_0)y^2 + (a_2 - a_1)y^3 - a_2 y^4. \end{aligned}$$

Equating coefficients, we deduce that $a_0 = 1$, $a_1 = -1$, and $a_2 = 1$. Thus, $x = \langle 1, -1, 1\rangle$. It turns out that the convolution operator is commutative because polynomial multiplication is commutative! Therefore,

$$\langle 1, 1, -1\rangle * x = x * \langle 1, 1, -1\rangle = \langle 1, 0, -1, 2, -1\rangle.$$

□

[K] Exercise 14.

(a) Compute the convolution $\langle 1, \underbrace{0, 0, \ldots, 0}_{k}, 1\rangle * \langle 1, \underbrace{0, 0, \ldots, 0}_{k}, 1\rangle$.

(b) Compute the DFT of the sequence $\langle 1, \underbrace{0, 0, \ldots, 0}_{k}, 1\rangle$.

Solution. (a) The sequence $\langle 1, \underbrace{0, 0, \ldots, 0}_{k}, 1\rangle$ corresponds to the polynomial $P(x) = 1 + x^{k+1}$. Therefore, the convolution $\langle 1, \underbrace{0, 0, \ldots, 0}_{k}, 1\rangle * \langle 1, \underbrace{0, 0, \ldots, 0}_{k}, 1\rangle$ corresponds to the polynomial multiplication:

$$(1 + x^{k+1})^2 = 1 + 2x^{k+1} + x^{2(k+1)}.$$

This corresponds to the sequence $\langle 1, \underbrace{0, \ldots, 0}_{k}, 2, \underbrace{0, \ldots, 0}_{k}, 1\rangle$.

(b) To compute the DFT, we start by converting the sequence into its polynomial representation; that is,

$$\langle 1, \underbrace{0, \ldots, 0}_{k}, 1\rangle \to P(x) = 1 + x^{k+1}.$$

The DFT is the sequence (with the same size) that is constructed by computing the powers of the $(k+2)$-th roots of unity. In other words, we have that

$$\begin{aligned} DFT(\langle 1, \underbrace{0, \ldots, 0}_{k}, 1\rangle) &= \langle P(\omega_{k+2}^0), P(\omega_{k+2}^1), \ldots, P(\omega_{k+2}^{k+1})\rangle \\ &= \langle 1 + \left(\omega_{k+2}^0\right)^{k+1}, 1 + \left(\omega_{k+2}^1\right)^{k+1}, \ldots, 1 + \left(\omega_{k+2}^{k+1}\right)^{k+1}\rangle \\ &= \langle 2, 1 + \omega_{k+2}^{k+1}, \ldots, 1 + \omega_{k+2}^{(k+1)^2}\rangle. \end{aligned}$$

□

[K] Exercise 15. Compute directly (i.e. without FFT) and maximally simplify the DFT of the following sequences:

(a) $\langle 1\underbrace{0,0,\ldots,0}_{n-1}\rangle$.

(b) $\langle \underbrace{0,0,\ldots,0}_{n-1},1\rangle$.

(c) $\langle \underbrace{1,\ldots,1}_{n}\rangle$.

Solution. (a) The sequence $\langle 1,\underbrace{0,\ldots,0}_{n-1}\rangle$ corresponds to the polynomial $P(x)=1$. Therefore, $P\left(\omega_n^k\right)=1$ for every k. Therefore,

$$DFT(\langle 1,\underbrace{0,\ldots,0}_{n-1}\rangle)=\langle\underbrace{1,1,\ldots,1}_{n}\rangle.$$

(b) The sequence $\langle\underbrace{0,\ldots,0}_{n-1},1\rangle$ corresponds to the polynomial $P(x)=x^{n-1}$. We also observe that

$$\omega_n^{n-k}=\omega_n^n\cdot\omega_n^{-k}=\omega_n^{-k},$$

because $\omega_n^n=1$. Therefore,

$$\begin{aligned}DFT(\langle\underbrace{0,\ldots,0}_{n-1},1\rangle)&=\langle P(\omega_n^0),P(\omega_n^1),\ldots,P(\omega_n^{n-1})\rangle\\&=\langle\omega_n^0,\omega_n^{n-1},\ldots,\omega_n^{(n-1)(n-1)}\rangle\\&=\langle 1,\omega_n^{-1},\omega_n^{-2},\ldots,\omega_n^{-(n-1)}\rangle\end{aligned}$$

(c) The sequence $\langle\underbrace{1,\ldots,1}_{n}\rangle$ corresponds to the polynomial

$$P(x)=1+x+\cdots+x^{n-1}=\frac{1-x^n}{1-x}.$$

Now we observe that, for each $k\neq 0$, we have that

$$\left(\omega_n^k\right)^n=\left(\omega_n^n\right)^k=1^k=1.$$

Therefore, $P\left(\omega_n^k\right)=0$ if $k\neq 0$. If $x=1=\omega_n^0$, then we have that

$$P\left(\omega_n^0\right)=1+\omega_n^0+\cdots+\omega_n^0=n.$$

Therefore,

$$DFT(\langle\underbrace{1,\ldots,1}_{n}\rangle)=\langle n,\underbrace{0,\ldots,0}_{n-1}\rangle.$$

□

[K] Exercise 16. You are given a sequence

$$A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$$

of length n and sequence

$$B = \langle 1, \underbrace{0, \ldots, 0}_{k}, 1 \rangle$$

of length $k+2$, where $1 \leq k \leq n/4$.

(a) Compute the DFT of sequence B.

(b) Compute the convolution sequence $C = A * B$ in terms of the elements in sequence A.

(c) Show that, in this particular case, such a convolution can be computed in $O(n)$ time.

Solution. (a) The sequence B corresponds to the polynomial $P_B(x) = 1 + x^{k+1}$. The DFT is

$$DFT(\langle 1, \underbrace{0, \ldots, 0}_{k}, 1 \rangle) = \langle 2, 1 + \omega_{k+2}^{k+1}, \ldots, 1 + \omega_{k+2}^{(k+1)^2} \rangle.$$

(b) The polynomial corresponding to the sequence A is

$$P_A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}.$$

Therefore,

$$\begin{aligned} P_C(x) &= P_A(x) \cdot P_B(x) \\ &= \left(a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}\right)\left(1 + x^{k+1}\right) \\ &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_0 x^{k+1} + a_1 x^{k+2} + \cdots + a_{n-1} x^{k+n}. \end{aligned}$$

We now use the fact that $k \leq n/4$ to transform this expression into

$$\begin{aligned} P_C(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_k x^k + (a_{k+1} + a_0)\, x^k + (a_{k+2} + a_1)\, x^{k+2} + \\ \cdots + (a_{n-1} + a_{n-k-1})\, x^{n-1} + a_{n-k-1} x^n + \cdots + a_{n-1} x^{n+k}. \end{aligned}$$

Therefore, $C = A * B$ is given by

$$C = \langle a_0, a_1, a_2, \ldots, a_k, (a_{k+1} + a_0), (a_{k+2} + a_1), \ldots, (a_{n-1} + a_{n-k-1}), a_{n-k-1}, \ldots, a_{n-2}, a_{n-1} \rangle.$$

(c) For any given sequence A, it is easy to construct the respective sequence C by simply looking at the coefficients of the polynomial $P_A(x)$. Since there are n many terms, each of which take $O(1)$ time to look up, the entire convolution can be computed in $O(n)$ time.

□

[K] Exercise 17. You have a set of N coins in a bag, each having a value between 1 and M, where $M \geq N$. Some coins may have the same value. You pick two coins (**without replacement**) and record the sum of their values. Determine what possible sums can be achieved in $O(M \log M)$ time.

Hint: If the coins have values $v_1, \ldots, v_N$, how might you use the polynomial $x^{v_1} + \ldots x^{v_N}$?

Solution. Form the polynomial $x^{v_1} + \cdots + x^{v_N}$ and use convolution to obtain its square. We make the following observation.

Observation. Let $Q(x) = (x^{v_1} + \cdots + x_{v_N})^2$. If the coefficient of x^k in $Q(x)$ is 1, then the only way for this to happen is if you obtained $x^{k/2}$ twice. Note that, if $v_i \neq v_j$, then the coefficient of $x^{v_i}x^{v_j} = x^{v_i+x_j}$ is 2 because we obtain the sum once with $x^{v_i}x^{v_j}$ and once with $x^{v_j}x^{v_i}$.

Since we are picking coins *without replacement*, we therefore discard all sums where the coefficient is 1 and the remaining powers are precisely all of the sums that can be obtained using two distinct coins. To argue the time complexity, observe that we perform a convolution on the polynomial that has degree $\leq M$. Using the FFT, this operates in $O(M \log M)$ time to compute the convolution and obtain the square. To determine which sums are possible, we perform a linear scan on $Q(x)$ which runs in $O(M)$; therefore, the overall time complexity is indeed $O(M \log M)$. □

[K] Exercise 18. Consider the polynomial

$$P(x) = (x - \omega_{64}^0)(x - \omega_{64}^1) \dots (x - \omega_{64}^{63}).$$

(a) Compute $P(0)$.

(b) What is the degree of $P(x)$? What is the coefficient of the highest degree term of x present in $P(x)$?

(c) What are the values of $P(x)$ at the roots of unity of order 64?

(d) Can you represent $P(x)$ in the coefficient form without any computation?

Solution. (a) We have that

$$\begin{aligned} P(0) &= (0 - \omega_{64}^0)(0 - \omega_{64}^1) \dots (0 - \omega_{64}^{63}) \\ &= (-1)^{64} \omega_{64}^{0+1+\cdots+63} \\ &= \omega_{64}^{32 \cdot 63} \\ &= (\omega_{64}^{32})^{63} \\ &= (-1)^{63} \\ &= -1. \end{aligned}$$

(b) The degree of $P(x)$ is 64. The coefficient of the highest degree of x is 1 (there is only one term in the product of degree 64, which is all the x's multiplied together).

(c) The values of $P(x)$ at all roots of unity ω_{64}^k, $0 \leq k < 64$ are all 0.

(d) Since $P(x)$ is precisely the polynomial whose roots are the 64 roots of unity of order 64 and whose leading coefficient (in front of the largest degree) is 1, we must have $P(x) = x^{64} - 1$, by definition. Note that this clearly also provides an easier way to solve part (a)!

□

[H] Exercise 19. Suppose you have K polynomials $P_1, \dots, P_K$ each of degree at least one, and that

$$\deg(P_1) + \cdots + \deg(P_K) = S.$$

(a) Show that you can find the product of these K polynomials in $O(KS \log S)$ time.

(b) Show that you can find the product of these K polynomials in $O(S \log S \log K)$ time.

Solution. (a) We can obtain the partial multiplications $\Pi(i) = P_1(x)P_2(x)\cdots P_i(x)$ for each $1 \le i \le K$. Initially, $\Pi(1) = P_1(x)$ and for all $i < K$, we have that $\Pi(i+1) = \Pi(i) \cdot P_{i+1}(x)$. At each stage, the degree of $\Pi(i)$ and $P_{i+1}(x)$ are both less than S; therefore, each of the multiplications (if performed using fast evaluation of the FFT) is bounded by the same constant multiple of $S \log S$. Since we perform K of these multiplications, we can obtain the product of K polynomials in $O(KS \log S)$.

(b) The easiest way is, just as in the *celebrity problem* from Problem Set 1, to organize polynomials and their intermediate products into a complete binary tree, a trick which is useful in many situations which involve recursively operating on pairs of objects. To remind you of the construction of a complete binary tree, we first compute $m = \lfloor \log_2 K \rfloor$ and construct a perfect binary tree with $2^m \le K$ leaves (i.e., a tree in which each node except the leaves has precisely two children and all the leaves are at the same depth). If $2^m < K$ add two children to each of the leftmost $K - 2^m$ leaves of such a perfect binary tree. In this way you obtain $2(K - 2^m) + (2^m - (K - 2^m)) = 2K - 2^{m+1} + 2^m - K + 2^m = K$ leaves exactly, but each leaf now has its pair, and the depth of each leaf is either $\lfloor \log_2 n \rfloor$ or $\lfloor \log_2 n \rfloor + 1$, see the picture below.

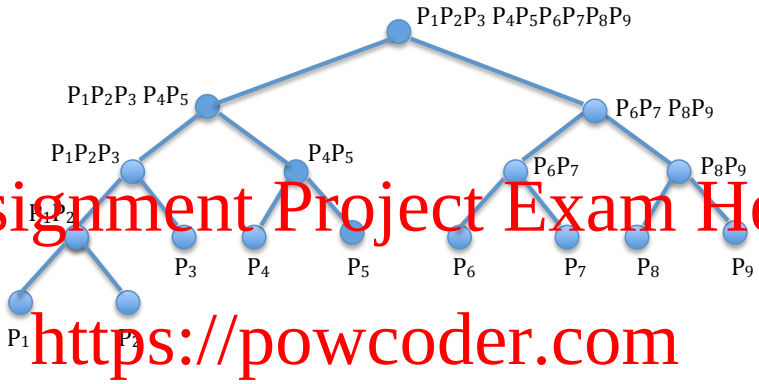


Figure 1: Here $K = 9$; thus, $m = \lfloor \log_2 K \rfloor = 3$ and we add two children to $K - 2^m = 9 - 2^3 = 1$ leaf of a perfect binary tree with 8 leaves. Thus obtained tree has 9 leaves.

Each leaf is now assigned one of the polynomials and the inner nodes of the tree represent partial products of polynomials corresponding to the two children. Note that, with the possible exception of the deepest level $\lfloor \log_2 n \rfloor + 1$ of the tree (which in the example on the picture contains only two polynomials), the sum of the degrees of polynomials on each level is equal to the sum of the degrees of all K polynomials $P_i(x)$, i.e. is equal to S. Let d_1 and d_2 be the degrees of two polynomials corresponding to the children of a node at some level k; then the product polynomial is of degree $d_1 + d_2$ and thus it can be evaluated (using the FFT to compute the convolution of the sequences of the coefficients) in time $O((d_1 + d_2)\log(d_1 + d_2))$ which is also $O((d_1 + d_2)\log S)$, because clearly $\log(d_1 + d_2) \le \log S$. Adding up such bounds for all products on level k we get the bound $O(S \log S)$, because the degrees of all polynomials at each of the levels add up precisely to S. Since we have $\lfloor \log K \rfloor + 1$ levels we get that the total amount of work is $O(\log K \cdot S \cdot \log S)$, as required.

□