CS218 Design and analysis of algorithms **Jan-Apr 2023**

Quiz 1

Total Marks: 20 *Time: 85 minutes*

Instructions.

- Please write your answers concisely.
- Describe the steps in your algorithms using English text. Use some kind of code only when necessary.
- Explain the reasoning behind the steps in your algorithm, whenever it's not obvious.

Question 1 [10 marks]. Suppose you are given a list of linear inequalities in two variables (x, y) of the following form.

$$\begin{aligned} y &\leq a_1x + b_1 \\ y &\leq a_2x + b_2 \\ &\vdots \\ y &\leq a_nx + b_n \end{aligned}$$

Here each a_i is a number (positive/negative/zero) and each b_i is a **positive** number. We are interested in the region defined by these inequalities, together with

$$y \geq 0.$$

That is, we are interested in the set of points which simultaneously satisfy all these inequalities. This set of points forms a (convex) polygon. We want to find the number of vertices in the polygon.

See Figure 1 (last page), for an example. It shows the region (shaded area) defined by the following six inequalities together with $y \geq 0$. The polygon we get is a pentagon, that is, a polygon with 5 vertices. This is because two of the inequalities become redundant.

$$\begin{aligned} &y \leq 2x + 10 \\ &y \leq x + 6 \\ &y \leq x/2 + 5 \\ &y \leq -x/2 + 2 \\ &y \leq -2x + 6 \\ &y \leq 4 \end{aligned}$$

Design an algorithm that takes as input an array of pairs $[(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)]$ and outputs the number of vertices in the polygon defined by inequalities

$$y \leq a_ix + b_i \text{ for } 1 \leq i \leq n \text{ and } y \geq 0.$$

For full marks your running time should be $O(n \log n)$. You will get only 4 marks for $O(n^2)$ running time. Assume basic arithmetic operations can be done in constant time.

Hint: Feel free to use or ignore the hint. There can be two approaches to design an $O(n \log n)$ algorithm. One approach is divide and conquer. Note that just computing the number of vertices for a subproblem might not be good enough. You might need to compute more information about the polygon. The other approach can be to sort the input in a particular order and then make one pass over the input to compute the required information. In this approach, you might need a special data structure.

Question 2. Consider the following approach for finding the square of an n bit integer a. We divide the integer into four parts each of $n/4$ bits (assume n is divisible by 4) and write

$$a = a_3\ 2^{3n/4} + a_2\ 2^{2n/4} + a_1\ 2^{n/4} + a_0.$$

Squaring both the sides we get,

$$a^2 = \mathbf{a_3^2}\ 2^{6n/4} + \mathbf{2a_3a_2}\ 2^{5n/4} + (\mathbf{a_2^2} + \mathbf{2a_3a_1})\ 2^{4n/4} + \mathbf{2(a_2a_1} + \mathbf{a_3a_0)}\ 2^{3n/4} + (\mathbf{a_1^2} + \mathbf{2a_2a_0})\ 2^{2n/4} + \mathbf{2a_1a_0}\ 2^{n/4} + \mathbf{a_0^2}.$$

Suppose we have a subroutine to find square of an integer with $n/4$ bits (or a few more, say up to $n/4 + 6$ bits). Assume that addition of two n bit integers can be done in $O(n)$ time. Assume that an n bit integer can multiplied/divided by a constant (e.g., 15, 2/3) in $O(n)$ time.

(i) [7 marks]. Show that you only need to use the above subroutine 7 times together with some $O(n)$ time computations to find the above 7 terms $(a_3^2, 2a_3a_2, (a_2^2 + 2a_3a_1), (a_2a_1 + a_3a_0), (a_1^2 + 2a_2a_0), 2a_1a_0, a_0^2)$.

Hint: Feel free to use or ignore this hint. Suppose we have a degree 2 polynomial $c_0 + c_1x + c_2x^2$. Suppose we evaluate it over three points $x = 0, 1, 2$. The three evaluations e_0, e_1, e_2 can be given by

$$\begin{pmatrix} e_0 \\ e_1 \\ e_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix}$$

Then coefficients can be computed from evaluations as

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \end{pmatrix}^{-1} \begin{pmatrix} e_0 \\ e_1 \\ e_2 \end{pmatrix}$$

(ii) [3 marks]. Using the above approach, write a recurrence relation for $T(n)$, time to find square of an n bit integer. What is the time complexity you get on solving the recurrence? Is it better or worse than $O(n^{\log_2 3})$?

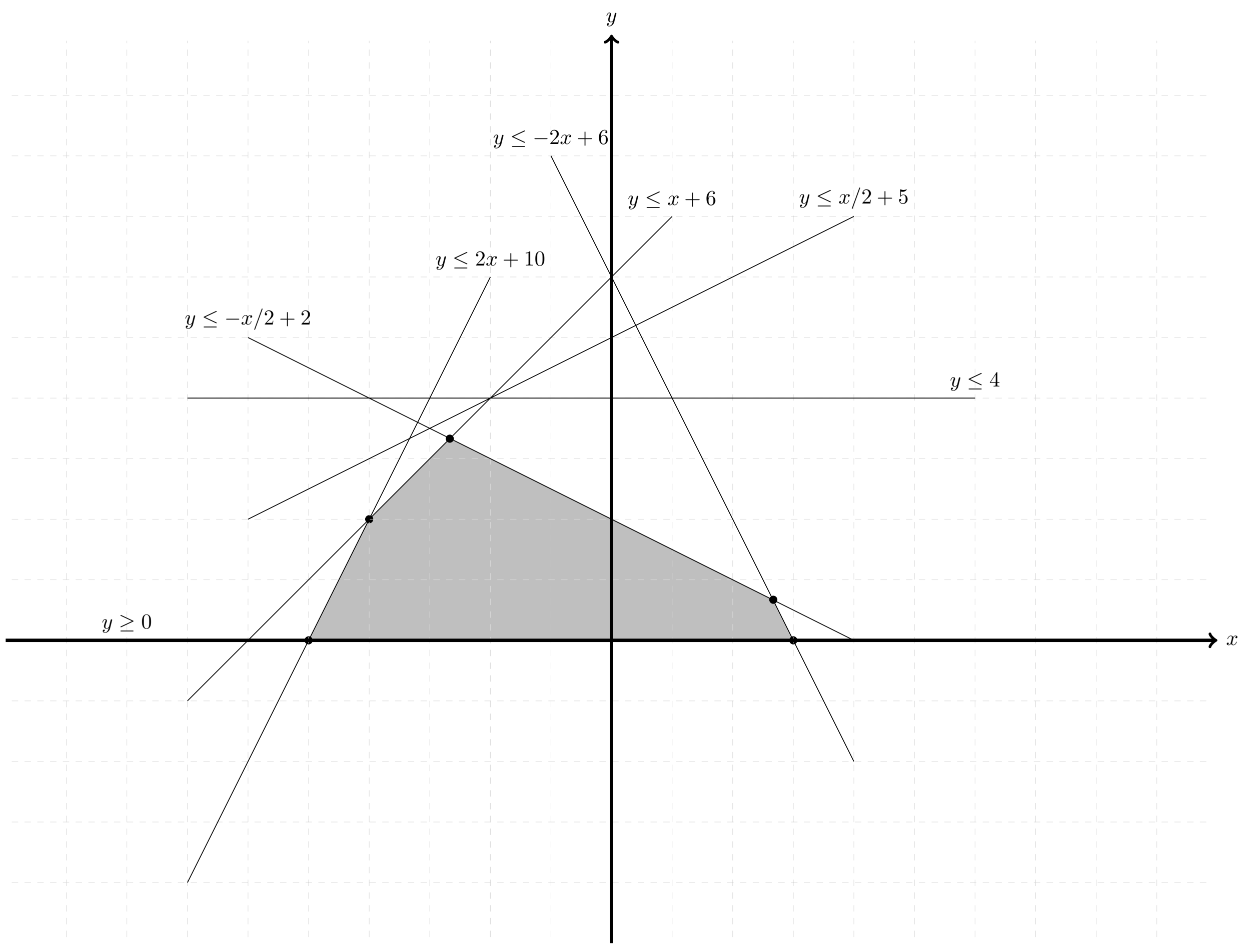

Figure 1: A polygon defined by 7 given inequalities, but with only 5 vertices.

Question 1 solution.

Here are some useful observations. The number of vertices in the polygon is same as the number of sides. So, we need to find the number of sides in the polygon. As seen in the example, some inequalities become redundant and don't make a side of the polygon. So, basically we need to find the number of inequalities which are not redundant.

Here is another observation. When we look at the consecutive sides of the polygon form left to right, their slopes (a_i) are decreasing. 2 marks for this observation. When we look at consecutive vertices, their x coordinates are in increasing order. This will be useful in both the approaches for $O(n \log n)$ time. First let us look at few simple approaches.

Time $O(n^3)$: when is a line l_i redundant? When it is "between" two lines l_j and l_k and is strictly above the intersection of l_j and l_k. By "between", we mean the slope of l is between the two other slopes, i.e., $a_j \leq a_i \leq a_k$. For example, in Figure 1, the line $y \leq x/2+5$ is redundant, because it is above the intersection point of $y \leq x+6$ and $y \leq -x/2+2$. One can easily check whether intersection of two lines is above or below a third line. Overall time is $O(n^3)$.

Time $O(n^2)$: We will process the lines one by one, in an arbitrary order. We will maintain the polygon formed by the current set of lines (to ensure that we have a polygon in the beginning, we can start by taking two lines one with positive and one with negative slope). We will maintain the vertices of the current polygon in increasing order of x coordinates. Note that number of vertices is same as the number of sides. When a new line comes, if it is above all the current vertices, then it is redundant and we can throw it. Otherwise the new line will intersect the current polygon at two points. We find the two line segments where the new line intersects and compute the points of intersections. Any vertices above the new line must be thrown away. All the update per new line can be done in total $O(n)$ time. Total 4 marks for $O(n^2)$ time.

Approach 1: Sorting in decreasing order of slopes. We need to find the line segments that form the boundary of the polygon.

Sorting: We first sort the lines w.r.t to their slope in decreasing order.

Processing lines: We insert the lines one by one in the decreasing order of slopes. For the lines already seen, we maintain the boundary of the current polygon (formed together with $y \geq 0$) in the following format.

Solution format: a list of line segments forming the boundary of the polygon, in increasing order of x-values (the last piece could be a line instead of a line segment).

When we insert a new line, it will intersect at most one of the existing line segments (or line). This is because the slope of the new line is smaller than all the existing line segments (see Figure 2).

To find the intersecting line segment, we can do a binary search as follows: start with the middle line segment in the list. If both the endpoints of this line segment are below the new line then go to the right half. If both the endpoints are above the new line then go to the left half. If one endpoint is above and one is below then we have found the intersecting segment.

Note that all other segments which come after the intersecting segment will disappear. The new line makes them redundant. Hence, we delete all the line segments in the list which appear after the intersecting segment. We compute the intersecting point of the segment and the new line and update the endpoint of the segment accordingly.

Each insertion will take $\mathcal{O}(\log n)$ time, because of binary search. Note that the deletion step does not take much time because, the deletion is from the end of the list. Hence, the overall time complexity is $\mathcal{O}(n \log n)$.

Approach 2: Divide and conquer We divide the given set of lines into two sets of roughly equal size, arbitrarily. Assume that we have computed the polygons for both the sets recursively. Now, we need to 'merge' the two polygons to compute the new polygon. It will be important how exactly we are representing the computed polygons.

Solution format: The list of lines which form the boundary of the polygon, in decreasing order of slopes.

The merge step: We have two arrays containing lines in decreasing order of slopes. We want to decide which of these line will be part of the boundary of the new polygon. We will go over the two arrays and decide which line should be put into the output array and which should be discarded (when it's redundant).

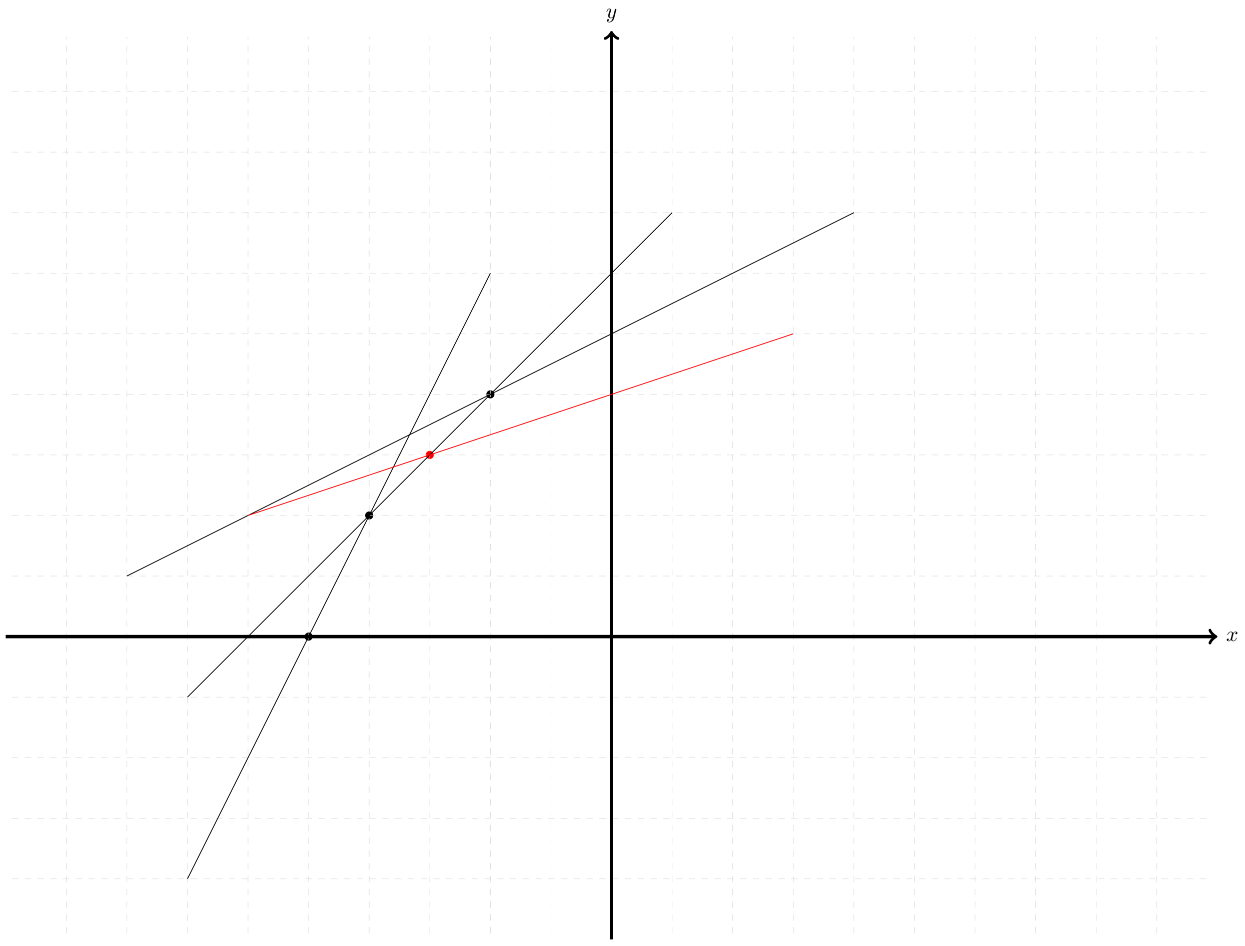

Figure 2: The new incoming line (in red) intersects one of the existing line segments.

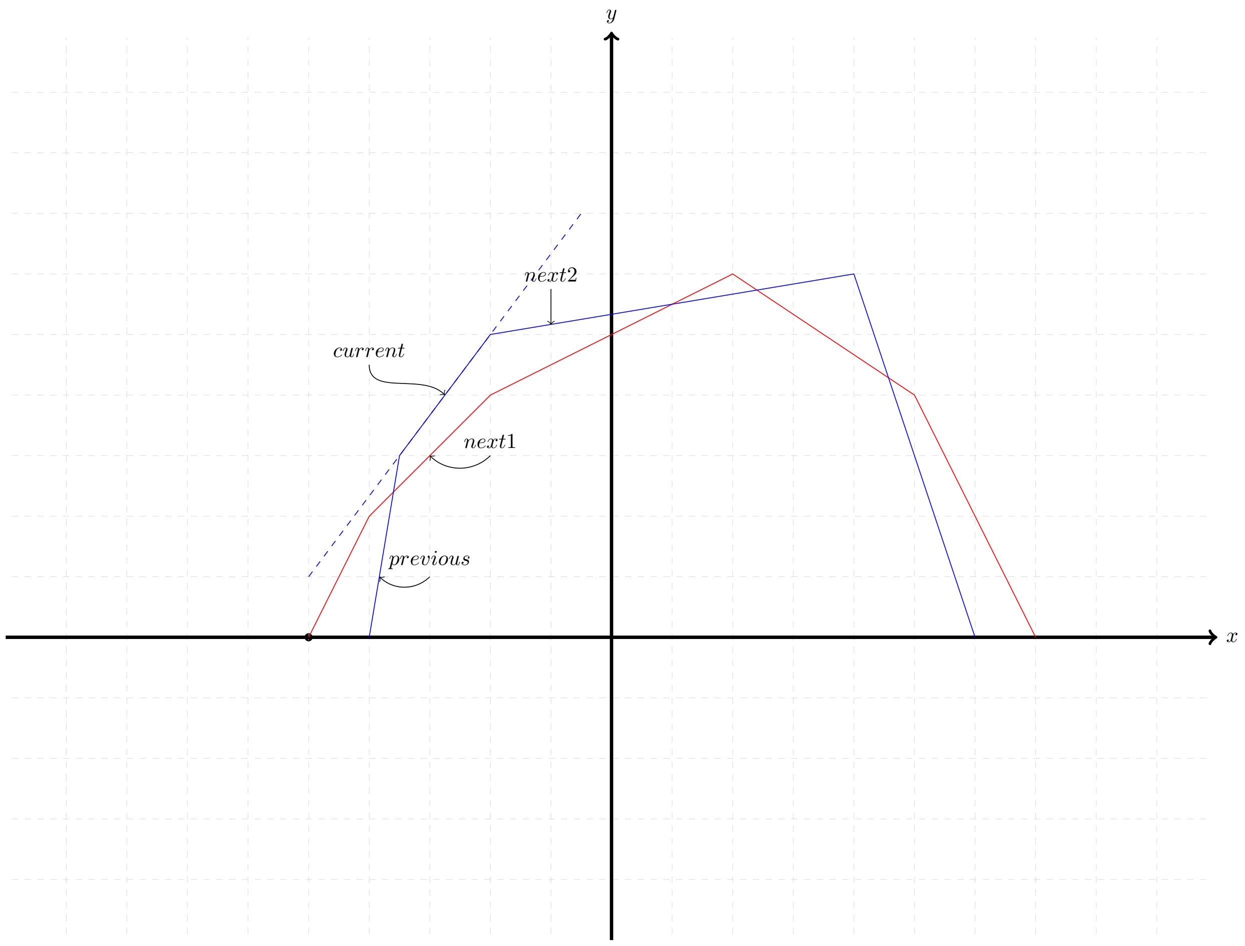

Figure 3: The current line is redundant because it is above the intersection of previous and next1.

Let's keep a pointer for each array, which are initially at the first entries of the arrays. Consider an intermediate stage, where the pointers in two arrays are pointing to lines ℓ_i and ℓ_j, respectively. Let the corresponding inequalities be $y \leq a_i x + b_i$ and $y \leq a_j x + b_j$.

Compare the slopes of ℓ_i and ℓ_j (i.e., a_i and a_j). Let us call the one with higher slope as the *current* line and the other one as the $next_1$ line. Let us call the line appearing after the current line in its array as $next_2$ line. Let us refer to the last line which was put in the output array as *previous* line.

- If the intersection of the *previous* line and the $next_1$ line is below the *current* line (i.e., the intersection point satisfies $y \leq a_i x + b_i$ for the current line), then the current line is redundant, and hence should be discarded (see Figure 4).
- If the intersection of the *previous* line and the $next_2$ line is below the *current* line, then the current line is redundant, and hence should be discarded (see Figure **??**).
- Else, the *current* line should be pushed in the output array.

Move the pointer ahead from the *current* line in its array. And repeat.

Corner case: When there is nothing in the output array, then the *previous* line can be taken as $x \geq \alpha_0$, where $(\alpha_0, 0)$ is leftmost vertex of the polygon. Similarly, when the pointer is at the last line in an array, $next2$ line can be taken as $x \leq \beta_0$, where $(\beta_0, 0)$ is the rightmost vertex of the polygon.

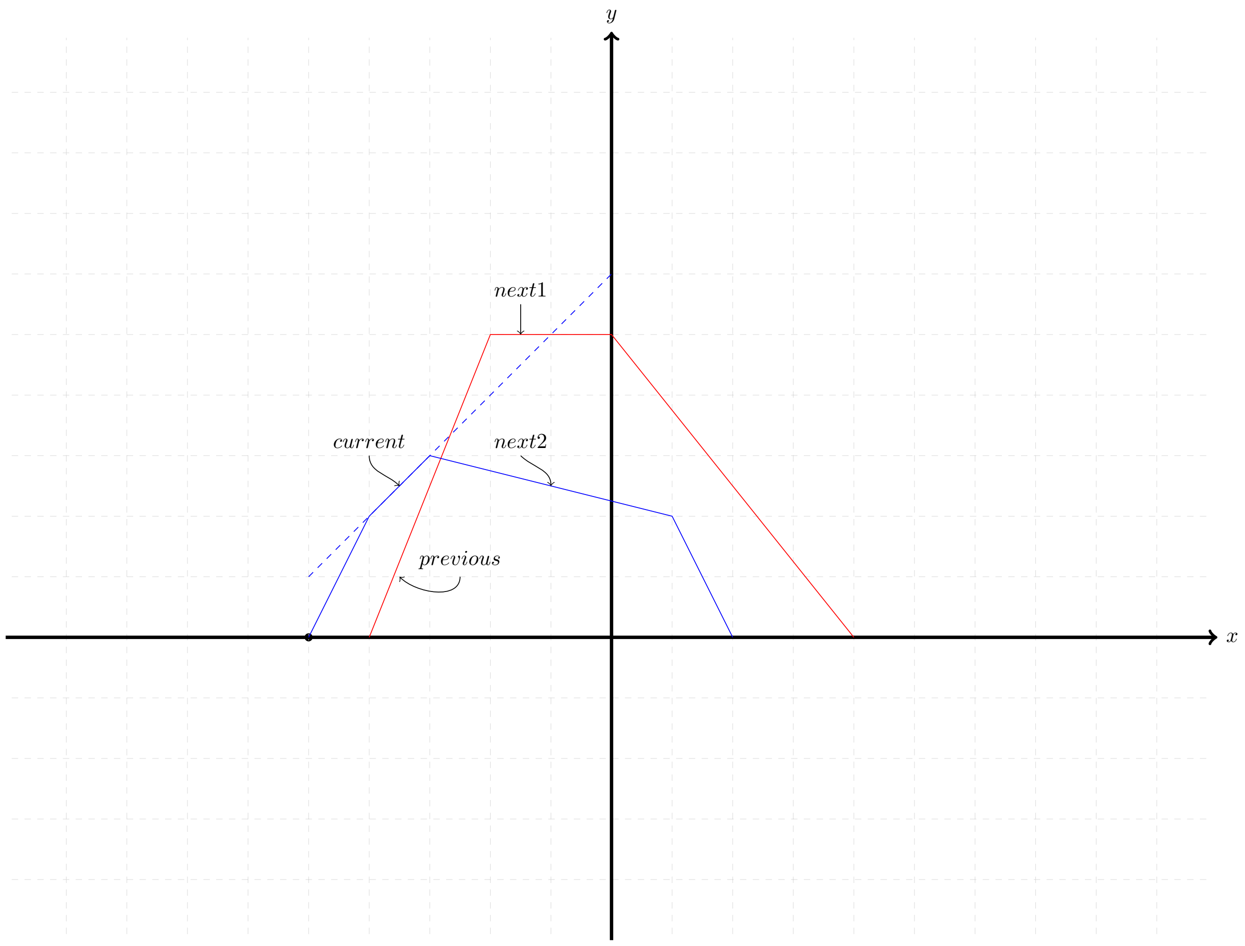

Figure 4: The current line is redundant because it is above the intersection of previous and next2.

Question 2 solution.

Part (a) Let us divide into four parts where each of a0, a1, a2, a3 is an n/4 bit integer. Squaring both the sides we get -

$a^2 = \mathbf{a_3^2}\ 2^{6n/4} + \mathbf{2a_3a_2}\ 2^{5n/4} + (\mathbf{a_2^2 + 2a_3a_1})\ 2^{4n/4} + \mathbf{2(a_2a_1 + a_3a_0)}\ 2^{3n/4} + (\mathbf{a_1^2 + 2a_2a_0})\ 2^{2n/4} + \mathbf{2a_1a_0}\ 2^{n/4} + \mathbf{a_0^2}.$

Now, we have to compute these 7 terms except the power of 2. For this, we need to see computation of these 7 terms as as a squaring of a polynomial. More precisely let us define a polynomial

$$A(x) = a_3\ x^3 + a_2\ x^2 + a_1\ x + a_0.$$

Its square will be given by

$$(A(x))^2\ = a_3^2\ x^6 + 2a_3a_2\ x^5 + (a_2^2 + 2a_3a_1)\ x^4 + 2(a_2a_1 + a_3a_0)\ x^3 + (a_1^2 + 2a_2a_0)\ x^2 + 2a_1a_0\ x + a_0^2.$$

Thus, finding the square of $A(x)$ is equivalent to compute the desired 7 terms. We have a three step plan for computing the square of $A(x)$:

1. Compute the evaluations of the polynomial $A(x)$ at 7 different values of x.
2. Square these 7 numbers to get 7 evaluations of $(A(x))^2$.
3. Since $(A(x))^2$ is a degree 6 polynomial, we can obtain its coefficients from its 7 evaluations

Step 1: The 7 values for x can be chosen arbitrarily. We choose x = -3, -2, -1, 0, 1, 2, 3. To compute $A(x)$ for any of these values of x, we first need to multiply a_3, a_2, a_1, a_0 with some constants and then add them up. From the assumption in the question, constants can be multiplied in $O(n)$ time. After that there are 3 additions, which can also be done in $O(n)$. Thus, step 1 can be finished in $O(n)$ time.

Step 2: We need to find squares of $A(3), A(2), A(1), A(0), A(1), A(2), A(3)$. From step 1, it can be seen that these numbers have at most n/4 + 6 bits. Thus, each of these numbers can be written as $(\alpha\ 2^6 + \beta)$, where α is n/4 bits and β is bounded by 2^6 . For squaring $(\alpha\ 2^6 + \beta)$, we need to compute α^2 plus a few additional $O(n)$ time operations. In summary, step 2 involves computing squares of 7 n/4 bit numbers and some other operations doable in $O(n)$ time.

Step 3: Given $A(-3)^2, A(-2)^2, A(-1)^2, A(0)^2, A(1)^2, A(2)^2, A(3)^2$, we want to compute coefficients of $(A(x))^2$. Lets us define $B(x) = (A(x))^2$. In other words, we have 7 evaluations for the degree 6 polynomial $B(x)$ and we want to compute its coefficients. Suppose coefficients of $B(x)$ are given by

$$B(x)\ = b_6\ x^6 + b_5\ x^5 + b_4\ x^4 + b_3\ x^3 + b_2\ x^2 + b_1\ x + b_0.$$

The relation between coefficients and evaluations of $B(x)$ are given as below

$$\begin{pmatrix} B(-3) \\ B(-2) \\ B(-1) \\ B(0) \\ B(1) \\ B(2) \\ B(3) \end{pmatrix} = \begin{pmatrix} 3^6 & -3^5 & 3^4 & -3^3 & 3^2 & -3 & 1 \\ 2^6 & -2^5 & 2^4 & -2^3 & 2^2 & -2 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2 & 1 \\ 3^6 & 3^5 & 3^4 & 3^3 & 3^2 & 3 & 1 \end{pmatrix} \begin{pmatrix} b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{pmatrix}$$

Let us call the 7 7 matrix above as M. Then coefficients of $B(x)$ can be computed as

$$M^{-1} \begin{pmatrix} B(-3) \\ B(-2) \\ B(-1) \\ B(0) \\ B(1) \\ B(2) \\ B(3) \end{pmatrix}$$

Note that all the entries in M^{-1} are constants. The evaluations of $B(x)$ have n/2 + 12 bits at most. Thus, the above matrix vector product can be done in $O(n)$ time.

Part (b) To find a^2, we did squares of 7 n/4 bit integers together with some $O(n)$ operations. Thus, for the time complexity we can write following recurrence relation -

$$T(n) \; = 7\, T(n/4) + O(n).$$

Solving it, we get

$$T(n) \; = O(n^{log_4 7}) = O(n^{(log_2 7)/2}) = O(n^{1.4037}).$$

We can see that the complexity of the above recurrence relation is less than $O(n^{log_2 3})$. This is because $\log_2 3 = \log_4 9 < \log_4 7$.