1. (a) For all $k \geqslant 3$, $T(k,1) = 1$ because the first k-sided polygonal number is 1.
   For all $k \geqslant 3$ and all $n \geqslant 2$, $T(k,n) = T(k,n-1) + (k-2)(n-1) + 1$ because the n-th k-sided polygonal number is obtained from the $(n-1)$-st k-sided polygonal number by adding $k-2$ rows of n dots each, but $k-3$ of the new dots belong to two new rows, and $(k-2)n - (k-3) = (k-2)(n-1) + 1$.

   (b) **Scratch work**: For all $k \geqslant 3$,

$$\begin{aligned} T(k,n) &= T(k,n-1) + (k-2)(n-1) + 1 \\ &= T(k,n-2) + (k-2)(n-2) + 1 + (k-2)(n-1) + 1 \\ &= T(k,n-2) + (k-2)(n-2+n-1) + 2 \\ &= \dots \\ &= T(k,n-i) + (k-2)(n-i+\dots+n-1) + i \\ &= T(k,1) + (k-2)(1+\dots+n-1) + n - 1 \\ &= 1 + (k-2)n(n-1)/2 + n - 1 \\ &= n + n(n-1)(k-2)/2 \end{aligned}$$

   Proof: We prove that $\forall k \geqslant 3, \forall n \geqslant 1, T(k,n) = n + n(n-1)(k-2)/2$.

   For universal generalization: Assume $k \geqslant 3$.
   Base Case: $T(k,1) = 1 = 1 + 0 = 1 + 1(1-1)(k-2)/2$.
   Ind. Hyp.: Assume $n > 1$ and $T(k,n-1) = (n-1) + (n-1)(n-2)(k-2)/2$.
   Ind. Step:

$$\begin{aligned} T(k,n) &= T(k,n-1) + (k-2)(n-1) + 1 && \text{(by recurrence, since } n > 1) \\ &= (n-1) + (n-1)(n-2)(k-2)/2 + (k-2)(n-1) + 1 && \text{(by I.H.)} \\ &= (n-1) + 1 + (n-1)(k-2)((n-2)/2 + 1) \\ &= n + (n-1)(k-2)(n/2) \\ &= n + n(n-1)(k-2)/2. \end{aligned}$$

   Conclusion: $\forall k \geqslant 3, \forall n \geqslant 1, T(k,n) = n + n(n-1)(k-2)/2$.

2. We start by writing down a recurrence relation to express the running time of each algorithm, then find a closed-form solution.

   (a) $T_A(1) = \Theta(1)$
   $T_A(n) = 5T_A(n/2) + \Theta(n)$ for $n > 1$
   The Master Theorem applies, with $a = 5$, $b = 2$, $d = 1$ so $a = 5 > 2 = b^d$ and $T_A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 5}) = \Theta(n^{2.32\dots})$.

   (b) $T_B(1) = \Theta(1)$
   $T_B(n) = 2T_B(n-1) + \Theta(1)$ for $n > 1$
   The Master Theorem does not apply but repeated substitution yields

$$\begin{aligned} T_B(n) &= 2T_B(n-1) + 1 \\ &= 2(2T_B(n-2) + 1) + 1 = 4T_B(n-2) + 3 \\ &= 4(2T_B(n-3) + 1) + 3 = 8T_B(n-3) + 7 \\ &= \dots \\ &= 2^i T_B(n-i) + 2^i - 1 \\ &= 2^{n-1} T_B(1) + 2^{n-1} - 1 = 2 \cdot 2^{n-1} - 1 = 2^n - 1 \end{aligned}$$

   so $T_B(n) \in \Theta(2^n)$.

(c) $T_C(1) = \Theta(1)$

$T_C(n) = 9T_C(n/3) + \Theta(n^2)$ for $n > 1$

The Master Theorem applies, with $a = 9$, $b = 3$, $d = 2$ so $a = 9 = 3^2 = b^d$ and $T_A(n) \in \Theta(n^d \log n) = \Theta(n^2 \log n)$.

Clearly, algorithm B is much worse than either A or C. Between algorithms A and C, algorithm C is better because $n^{\log_2 5} = n^2 \cdot n^{0.32\ldots} \geqslant n^2 \cdot \log n$ for all n large enough.

3. First, note that a single merge operation takes time proportional to the length of the resulting array.

(a) The first merge takes time proportional to $2n$, the second merge takes time proportional to $3n$, ..., the $(k-1)$-st merge takes time proportional to kn.

So the total time required is proportional to $2n + 3n + \cdots + kn = n(k-1)(k+2)/2 \in \Theta(nk^2)$.

(b) Let A_i denote array number i, for $1 \leqslant i \leqslant k$. The algorithm will merge the first half of the arrays together, recursively, then merge the second half of the arrays together, and then do one final merge of the two results. More formally:

```
RecMerge(b, e):
    # Precondition:   1 ⩽ b ⩽ e ⩽ k
    # Postcondition:  return the merge of A_b, ..., A_e
    if b = e:  return A_b
    m ← ⌊(b + e)/2⌋
    A_1 ← RecMerge(b, m)
    A_2 ← RecMerge(m + 1, e)
    return A_1 merged with A_2
```

The running time of the algorithm is described by the following recurrence, where $m = e + 1 - b$ is the number of arrays being merged.

$$T(n, m) = \begin{cases} \Theta(1) & \text{if } m = 1, \\ T(n, \lceil m/2 \rceil) + T(n, \lfloor m/2 \rfloor) + \Theta(nm) & \text{if } m > 1. \end{cases}$$

The Master Theorem applies to $T(n, m)$ for parameter m, with $a = 2$, $b = 2$, and $d = 1$, so $a = 2 = b^d$ and $T(n, m) \in \Theta(nm^d \log m) = \Theta(nm \log m)$.

Hence, the divide-and-conquer algorithm's running time is $\Theta(nk \log k)$, which is faster than the first algorithm.

4. (a) Let n be the total number of bits to represent both x and y.

If x and y are represented using only 1 bit in total, then ($x = 0$ and $y = 1$) or ($x = 1$ and $y = 0$) so the algorithm performs only a constant number of operations.

If x and y are represented using $n > 1$ bits in total, then in the worst-case, the algorithm makes a recursive call. This involves computing $y - x$ (or $x - y$) in time $\Theta(n)$ (in the worst-case), and the input size for the recursive call has size at most $n - 1$ (n_2 bits for $y - x$, so $n_2 - 1$ bits for $(y - x)/2$).

This gives the following recurrence relation for the worst-case running time:

$$T(1) = \Theta(1),$$
$$T(n) = T(n-1) + \Theta(n) \qquad \text{for all } n > 1.$$

The Master Theorem does not apply, but repeated substitution yields:

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &= \dots \\ &= T(1) + 2 + \cdots + n \\ &= n(n+1)/2 \end{aligned}$$

We can prove formally that this is correct (using 1 in place of $\Theta(1)$ and n in place of $\Theta(n)$ in the recurrence):

Base Case: $T(1) = 1 = 1(1+1)/2$.

Ind. Hyp.: Assume $n \geqslant 1$ and $T(n) = n(n+1)/2$.

Ind. Step: $T(n) = T(n-1) + n$ (from the recurrence)
$= (n-1)n/2 + n$ (by the I.H.)
$= ((n-1)/2 + 1)n$
$= n(n+1)/2$

Conclusion: $\forall n \geqslant 1, T(n) = n(n+1)/2$.

(b) By induction on $n \geqslant 1$, we prove that for all x, y represented using a total of n bits, the call REC-GCD(x, y) terminates and returns $\gcd(x, y)$, the greatest common divisor of x and y.

Base Case: If x and y are represented using a total of 1 bit, then either $x = 0$ and $y = 1$, or $x = 1$ and $y = 0$. In either case, the algorithm returns $1 = \gcd(x, y)$ on the first line.

Ind. Hyp.: Assume $n > 1$ and for all x, y represented using fewer than n bits, the call REC-GCD(x, y) terminates and returns $\gcd(x, y)$.

Ind. Step: Let x, y be inputs represented using exactly n bits, and consider the call REC-GCD(x, y). Then either $x = 0$ or $y = 0$ or $x = 1$ or $y = 1$ or ($x > 1$ and $y > 1$). And in the last case, either x is even or x is odd, and independently, either y is even or y is odd.

- If $x = 0$, then $\gcd(x, y) = \gcd(0, y) = y$ and the algorithm returns $x + y = 0 + y = y$ on the first line.
- Similarly, if $y = 0$, then the algorithm returns $x = \gcd(x, y)$ on the first line.
- If $x = 1$, then $\gcd(x, y) = \gcd(1, y) = 1$ and the algorithm returns 1 on the second line.
- Similarly, if $y = 1$, then the algorithm returns $1 = \gcd(x, y)$ on the second line.
- If $x > 1$ and $y > 1$ and x is even and y is even, then the algorithm makes a recursive call on $x/2$ and $y/2$, whose total bit-length is $n - 2$. By the I.H., the recursive call terminates and returns $\gcd(x/2, y/2)$. This means the call REC-GCD(x, y) also terminates and returns $2\gcd(x/2, y/2) = \gcd(2x/2, 2y/2) = \gcd(x, y)$.
- If $x > 1$ and $y > 1$ and x is even and y is odd, then the algorithm makes a recursive call on $x/2$ and y, whose total bit-length is $n - 1$. By the I.H., the recursive call terminates and returns $\gcd(x/2, y)$. This means the call REC-GCD(x, y) also terminates and returns $\gcd(x/2, y) = \gcd(x, y)$, since 2 is not a factor of y.
- Similarly, if $x > 1$, $y > 1$, x is odd, y is even, then the algorithm terminates and returns $\gcd(x, y/2) = \gcd(x, y)$.
- If $x > 1$ and $y > 1$ and x is odd and y is odd, then either $x < y$ or $x > y$ or $x = y$.
  - If $x < y$, then the algorithm makes a recursive call on $(y - x)/2$ and x, whose total length is at most $n - 1$ (since $y - x \leqslant y$ so $(y - x)/2 \leqslant y/2$). By the I.H., the recursive call terminates and returns $\gcd((y - x)/2, x)$.
    Now, by definition of gcd and since $x < y$, there exist positive integers $k < \ell$ such that $x = k \cdot \gcd(x, y)$ and $y = \ell \cdot \gcd(x, y)$ and k and ℓ have no common divisor. Since x

and y are both odd, both k and ℓ are also odd, and $y - x = (\ell - k) \cdot \gcd(x, y)$. Since there are no divisors common to k and ℓ, there are no divisors common to k and $\ell - k$ so $\gcd((y-x)/2, y) = \gcd(x, y) \cdot \gcd((\ell-k)/2, k) = \gcd(x, y)$. This means the call REC-GCD(x, y) also terminates and returns $\gcd((y - x)/2, x) = \gcd(x, y)$.

- Similarly, if $x > y$, the algorithm terminates and returns $\gcd((x - y)/2, y) = \gcd(x, y)$.
- Finally, if $x = y$, then the algorithm makes a recursive call on $(x - y)/2 = 0$ and x, which returns x. This means the call REC-GCD(x, y) terminates and returns $x = \gcd(x, y)$.

In every case, the call REC-GCD(x, y) terminates and returns $\gcd(x, y)$.

Conclusion: By induction on n, for all $n \geqslant 1$ and all x, y represented using a total of n bits, the call REC-GCD(x, y) terminates and returns the greatest common divisor of x and y.