GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: CONSEJOS PARA SER PROGRAMADOR COMPETITIVO

1. Introducción

Si te esfuerzas por ser un programador competitivo, eso es, si desea ser seleccionado a través de selecciones provinciales/nacionales para el equipo nacional y participar y obtener una medalla en el IOI, o ser uno de los integrantes del equipo que represente a tu Universidad en el ICPC (Nacionales, Regionales y hasta Finales Mundiales), o para hacer bien en otros concursos de programación, entonces te daremos algunos consejos de programación. derivados de nuestras propias experiencias que pueden ser útiles en situaciones de concurso:

2. Conocimientos previos

2.1. International Olympiad in Informatics (IOI)

IOI se inició en 1989 (en Bulgaria) y ha existido anualmente desde entonces. La competencia del IOI consta (normalmente) de 2 horas de práctica, 4 sesiones y dos días de competencia, 5 horas por sección. IOI es un concurso individual. Cada concurso (normalmente) consta de 3 tareas, normalmente una tarea fácil (más), una mediana y una tarea difícil (más), que se dividen en subtareas con varios puntos.

2.2. International Collegiate Programming Contests (ICPC)

ICPC se estableció en 1970, se originó en los EE. UU., Se extendió por todo el mundo a partir de la 1990 Desde 2000 (excepto 2003 y 2007), los ganadores suelen ser rusos (especialmente de 2012 a 2019) y universidades asiáticas.

La competencia ICPC tiene una duración de 5 horas. Cada equipo está formado por tres estudiantes universitarios. Cada equipo sólo cuenta con un ordenador. Solo las presentaciones que son Aceptadas (totalmente correcta) dará +1 punto al equipo. El equipo recibe una penalización por cada envío no aceptado (generalmente +20 minutos a su tiempo total).

Los conjuntos de problemas del ICPC generalmente están diseñados de tal manera que todos los equipos resuelven al menos uno. problema (para evitar desmoralizar totalmente a los recién llegados a la competencia, esto es lo que nos esforzamos para ayudar a través de este libro), ningún equipo resuelve todos los problemas (para que el concurso sea interesante hasta que el final de la quinta hora), y todos los problemas son solucionables por al menos un equipo (así se minimiza el cantidad de problemas ímposibles'que requieren mucho más de 5 horas para pensar y codificar la solución correctamente, incluso para los equipos favoritos percibidos antes del concurso).

3. Desarrollo

Aqui va de algunos de nuestros consejos para convertirte en un programador competitivo:

1. **Escriba el código más rápido:**: Aunque este consejo puede no significar mucho ya que ICPC y (especialmente) IOI no son concursos de mecanografía, hemos visto equipos ICPC separados solo por unos pocos minutos y concursantes frustrados de IOI que se pierden de salvar

marcas importantes al no ser capaz de codificar correctamente una solución de fuerza bruta de última hora. Cuando puedes resolver el mismo número de problemas que sus adversarios, entonces será la habilidad de codificación (su capacidad para producir código conciso y robusto) y ... velocidad de escritura ... que determinan el ganador. Pruebe esta prueba de mecanografía en y siga las instrucciones allí sobre cómo mejorar su habilidad de mecanografía. Si su velocidad de escritura es mucho menor que 55-65 ppm , por favor ¡toma este consejo en serio! Además de poder escribir caracteres alfanuméricos de forma rápida y correcta, podrá también necesita familiarizar sus dedos con las posiciones de la programación de uso frecuente caracteres de idioma: paréntisis () o llaves {} o corchetes [] o operadores <>, el punto y coma; y dos puntos:, comillas simples para caracteres, comillas dobles para cadenas, el ampersand &, la barra vertical |, el signo de exclamación !, etc.

2. **Identifique rápidamente los tipos de problemas:** El objetivo principal no es solo asociar problemas con las técnicas requeridas para resolverlos. Lo importante es que usted sea capaz de clasificar los problemas en los cuatro tipos siguientes:

No	Categoría	Confianza y velocidad de resolución esperada
A1	He resuelto este tipo antes	Estoy seguro de que puedo volver a resolverlo de nuevo (y rápido)
A2	He resuelto este tipo antes	Estoy seguro de que puedo volver a resolverlo de nuevo (pero lento)
B1	He visto este tipo antes	Has estudiado acerca del tema y tienes nociones de como solucionarlo o sabes dentro del equipo quien este tipo de ejercicio es un A1 o A2
B2	He visto este tipo antes	No puedo resolverlo la falta de estudio provoca que tenga que hacerlo de un lado
C	No he visto este tipo antes.	No tengo ni la menor idea de como solucionarlo por tanto voy a tener que esperar que alguien lo explique.

Para ser competitivo, es decir, hacerlo bien en un concurso de programación, debe ser capaz de y con frecuencia clasificar los problemas como tipo A1 y minimice la cantidad de problemas que clasifican en tipo A2, B1, B2. Es decir, debe adquirir suficiente conocimiento del algoritmo y desarrolle sus habilidades de programación para que considere que muchos problemas clásicos son fáciles especialmente al comienzo del concurso.

Sin embargo, para ganar un concurso de programación, también deberá desarrollar habilidades de resolución para que usted (o su equipo) pueda derivar la solución requerida a un problema de tipo C dificiles/original en IOI o ICPC y hacerlo dentro de la duración del concurso, no después de que la(s) solución(es) sea(n) revelada(s) por el(los) autor(es) del problema/juez(es) del concurso. Algunos de las habilidades necesarias para resolver problemas son:

- Reducir el problema dado a otro problema (más fácil),
- Reducir un problema (NP-)difícil conocido en el problema dado,

- Identificar pistas sutiles o propiedades especiales en el problema,
- Atacar el problema desde un ángulo no obvio/hacer una pregunta diferente,
- Comprimir los datos de entrada,
- Reelaboración de fórmulas matemáticas,
- Listado de observaciones/patrones,
- Realización de análisis de casos de posibles subcasos del problema, etc.

3. **Hacer análisis de algoritmos:** Una vez que haya diseñado un algoritmo para resolver un problema particular en un concurso de programación, entonces debe hacer esta pregunta: dado el límite máximo de entrada (generalmente dado en un buen descripción del problema), ¿puede el algoritmo desarrollado actualmente, con su complejidad de tiempo/espacio, pasar el límite de tiempo/memoria dado para ese problema en particular?

   A veces, hay más de una manera de atacar un problema. Algunos enfoques pueden ser incorrectos, otros no lo suficientemente rápidos y otros exagerados. Una buena estrategia es hacer una lluvia de ideas. Para muchos algoritmos posibles y luego elegir la solución más simple que funcione (es decir, es rápido suficiente para pasar el límite de tiempo y memoria y aun así producir la respuesta correcta)

   Las computadoras modernas son bastante rápidas y pueden procesar hasta $\approx$ 100M (o 10^8 ; 1M = 1 000 000) operaciones en un segundo. Puede utilizar esta información para determinar si su el algoritmo se ejecutará en el tiempo.

   Los límites del problema son tan importantes como la complejidad temporal de su algoritmo para determinar si su solución es adecuada. Suponga que sólo puede idear un modelo relativamente simple de algoritmo de código que se ejecuta con una complejidad de tiempo horrenda de O(n^4). Esto puede parecer ser una solución no factible, pero si n $\approx$ 50, entonces realmente ha resuelto el problema.

   Tenga en cuenta, sin embargo, que el orden de complejidad no indica necesariamente el número real número de operaciones que requerirá su algoritmo. Si cada iteración implica un gran número de operaciones (muchos cálculos de punto flotante, o un número significativo de subbucles constantes), o si su implementación tiene una constante alta en su ejecución (bucles repetidos innecesarios, varias pasadas del conjunto de datos, o incluso sobrecarga de ejecución de entrada/salida (E/S), su código puede tardar más en ejecutarse de lo esperado. Sin embargo, esto no suele ser un gran problema ya que el los autores de problemas deberían haber diseñado los límites de tiempo para que unos pocos (más de uno). Las implementaciones razonables del algoritmo con la complejidad de tiempo objetivo prevista serán todas alcanzar el veredicto Aceptado (AC).

   Al analizar la complejidad de su algoritmo con el límite de entrada dado y el indicado límite de tiempo/memoria, puede decidir mejor si debe intentar implementar su algoritmo (que ocupará un tiempo precioso en los IOI y los ICPC), intenta mejorar su algoritmo primero, o cambie a otros problemas en el conjunto de problemas.

Muchos programadores novatos se saltarían esta fase e implementarían inmediatamente la primera (ingenua) algoritmo en el que pueden pensar solo para darse cuenta de que la estructura de datos elegida y/o el algoritmo es/no es lo suficientemente eficiente (o incorrecto). Nuestro consejo para los concursantes : abstenerse desde la codificación hasta que esté seguro de que su algoritmo es correcto y lo suficientemente rápido. Para ayudarlo a comprender el crecimiento de varias complejidades de tiempo comunes y, por lo tanto, ayudarlo a usted juzga qué tan rápido es suficiente. Por lo general, el algoritmo más simple tiene la menor complejidad temporal, pero si puede Ya pasa el límite de tiempo, ¡solo úsalo!

N	El peor algoritmo de AC
$N \leq [10 \dots 11]$	$O(n!), O(n^6)$
$N \leq [17 \dots 19]$	$O(2^n \times n^2)$
$N \leq [18 \dots 22]$	$O(2^n \times n)$
$N \leq [24 \dots 26]$	$O(2^n)$
$N \leq 100$	$O(n^4)$
$N \leq 450$	$O(n^3)$
$N \leq 1{,}5K$	$O(n^{2,5})$
$N \leq 2{,}5K$	$O(n^2 \log n)$
$N \leq 10K$	$O(n^2)$
$N \leq 200K$	$O(n^{1,5})$
$N \leq 4{,}5M$	$O(n \log n)$
$N \leq 10M$	$O(n \log \log n)$
$N \leq 100M$	$O(n), O(\log n), O(1)$

4. **Maestros en lenguajes de programación:** Hay varios lenguajes de programación compatibles con las competencias de programación , incluidos C/C++, Java y Python. ¿Qué lenguajes de programación se debe aspirar a dominar?

   Nuestra experiencia nos da esta respuesta: preferimos C++ (std=gnu++17) con su Biblioteca de plantillas estándar (STL), pero aún necesitamos dominar Java y algunos conocimientos de Python. Aunque es más lento, Java tiene potentes bibliotecas integradas y API como BigInteger/BigDecimal, GregorianCalendar, Regex, etc. Los programas Java son más fáciles de depurar con la capacidad de la máquina virtual para proporcionar un seguimiento de la pila cuando falla (a diferencia de volcados de núcleo o fallas de segmentación en C/C++). De manera similar, el código de Python puede ser sorprendentemente muy corto para algunas tareas adecuadas. Por otro lado, C/C++ tiene sus propios méritos en cuanto tiempo y uso de memoria.

5. **Domina el arte de probar el código:** Pensaste que habías resuelto un problema en particular. Había identificado su tipo de problema, diseñó el algoritmo para ello, verificó que el algoritmo (con las estructuras de datos que utiliza) se ejecutaría en el tiempo (y dentro de los límites de la memoria) considerando el tiempo (y el espacio) e implementó el algoritmo, pero su solución aún no es aceptada (AC).

   En cualquier caso, deberá poder diseñar pruebas buenas, completas y complicadas. casos.

La entrada-salida de muestra dada en la descripción del problema es por naturaleza trivial y sólo allí para ayudar a la comprensión del enunciado del problema. Por lo tanto, los casos de prueba de muestra generalmente son insuficientes para determinar la corrección de su código.

En lugar de desperdiciar enviós, es posible que desee diseñar casos de prueba complicados para probar su código en su propia máquina. Asegúrese de que su el código es capaz de resolverlos correctamente (de lo contrario, no tiene sentido enviar su solución ya que es probable que sea incorrecto, a menos que desee probar los límites de los datos de prueba).

Aquí hay algunas pautas para diseñar buenos casos de prueba a partir de nuestra experiencia:

- Para problemas con múltiples casos de prueba en una sola ejecución, debe incluir dos casos de prueba de muestra idénticos consecutivamente en la misma ejecución. Ambos deben generar las mismas respuestas correctas conocidas. Esto ayuda a determinar si ha olvidado para inicializar cualquier variable, si la primera instancia produce la respuesta correcta pero la el segundo no, es probable que no haya reiniciado sus variables.
- Sus casos de prueba deben incluir casos extermos complicados. Piense como el autor del problema y intente encontrar la peor entrada posible para su algoritmo identificando casos que están ocultos o implícitos en la descripción del problema. Estos casos suelen ser incluidos en los casos de prueba secretos del juez, pero no en la muestra de entrada y salida. Los casos de esquina generalmente ocurren en valores extremos como N = 0, N = 1, negativo valores, valores finales grandes (y/o intermedios) que no se ajustan a enteros con signo de 32 bits, grafo vacío/ en línea/ árbol / bipartito / cíclico / acíclico / completo / desconectado, etc.
- Sus casos de prueba deben incluir casos grandes. Aumente el tamaño de entrada gradualmente hasta los límites de entrada máximos establecidos en la descripción del problema. Use casos de prueba grandes con estructuras triviales que son fáciles de verificar con cálculo manual y grandes cantidades aleatorias casos de prueba para probar si su código termina a tiempo y aún produce resultados razonables (ya que la corrección sería difícil de verificar aquí). A veces su programa puede funciona para casos de prueba pequeños, pero produce una respuesta incorrecta, falla o excede el tiempo límite cuando el tamaño de entrada aumenta. Si eso sucede, verifique si hay desbordamientos, fuera de límite errores, o mejorar su algoritmo.
- Aunque esto es raro en los concursos de programación modernos, no asuma que la entrada siempre estará bien formateado si la descripción del problema no lo establece explícitamente (especialmente para un problema mal escrito). Intente insertar espacios en blanco adicionales (espacios, tabs) en la entrada y pruebe si su código aún puede obtener los valores correctamente.

6. **Práctica y más práctica:** Los programadores competitivos, como los atletas reales, deben entrenar regularmente y seguir programando. Creemos que el éxito viene como resultado de un esfuerzo continuo para superarte a ti mismo. La puesta en práctica de este consejo de

va ayudar mucho cuando apliques el consejo 2.

7. **Trabajo en equipo (para ICPC):** Este consejo no es algo fácil de enseñar, pero aquí hay algunas ideas que pueden servir. Vale la pena intentarlo para mejorar el rendimiento de tu equipo:

   - Practica codificar (o escribir pseudocódigo) en un papel en blanco. Esto es útil cuando su compañero de equipo está usando la computadora. Cuando sea su turno de usar la computadora, puede luego simplemente escriba el código lo más rápido posible.
   - La estrategia de .enviar e imprimir": si su código obtiene un veredicto AC, ignore la impresión. Si todavía no es AC, depure su código usando esa impresión (y deje que su compañero de equipo use la computadora por otro problema). Cuidado: La depuración sin la computadora no es una habilidad fácil de dominar.
   - Si su compañero de equipo está programando actualmente (y no tiene idea de otros problemas), entonces preparar datos de prueba (y con suerte el código de su compañero de equipo pasa todos aquellos). Con dos miembros del equipo acordando la corrección (potencial) de un código, la probabilidad de tener penalidades disminuye o desaparece.
   - Si sabe que su compañero de equipo es (significativamente) más fuerte en cierto tipo de problema que usted mismo y actualmente está leyendo un problema con ese tipo (especialmente en la etapa inicial del concurso), considere pasar el problema a su compañero de equipo en su lugar de insistir en resolverlo uno mismo.
   - Practique la codificación de un algoritmo bastante largo/complicado como un par o incluso como un triple (con una presión de límite de tiempo de codificación) para la situación del final del concurso o competencia donde su equipo tiene como objetivo obtener +1 AC más en los últimos minutos.
   - El factor X: hazte amigo de tus compañeros de equipo fuera de las sesiones de entrenamiento y los concursos.

4. Aplicaciones

La aplicación de estos consejos te pueden ayudar a ser un mejor programador competitivo y mejorar tus resultados en concursos y competencias.