Syllabus para la Olimpiada Chilena de Informática

1. Versión y estatus de este documento

Este documento es la versión oficial del Syllabus para la **OCI 2023**. El Syllabus es un documento oficial relacionado a la OCI. Cada año el Comité Científico de la OCI publicará una versión actualizada del Syllabus. Todas las secciones con modificaciones desde la versión anterior están marcados en color rojo.

2. Autores e información de contacto

Una primera versión de este documento fue publicado para la OCI 2013 a cargo de Cristian Ruz, Pontificia Universidad Católica de Chile (`cruz@ing.puc.cl`) y Jorge Pérez, Universidad de Chile (`jperez@dcc.uchile.cl`). La versión actual del documento ha sido modificada a partir de la original por Nicolás Lehmann, actual director del Comité Científico.

Cualquier persona es bienvenida a realizar consultas y sugerencias sobre el Syllabus a la dirección de correo del mantenedor actual (`nico.lehmannm@gmail.com`).

3. Introducción

Este documento contiene información acerca de los conceptos mínimos (recomendados) para la fase regional y nacional de la Olimpiada Chilena de Informática (OCI). Los objetivos principales de este documento son dos:

- Servir de guía para las distintas delegaciones que se encuentren entrenando a sus alumnos para la OCI.
- Servir como guía al Comité Científico para determinar qué problemas son adecuados para la OCI.

El Syllabus pretende cumplir estos objetivos dando una categorización de tópicos en ciencias de la computación. De forma específica, el Syllabus divide los tópicos en tres categorías.

♡ **Básico**
Los tópicos en esta categoría son considerados requisito mínimo. Se espera que los participantes los conozcan. Estos tópicos podrán aparecer en la descripción de un problema sin una explicación adicional. El comité científico asegurará que en cada problema al menos una subtarea podrá ser resuelta usando solo tópicos dentro de esta categoría (referirse a apéndice A).

⊖ **Avanzado**
Es recomendable para los participantes conocer estos tópicos pues les permitirá obtener el máximo puntaje en cada problema. Algunos tópicos en esta categoría no serán evaluados de forma explícita, sino más bien son una recomendación para ayudar a abordar los problemas (ej. Sec. 5.1 - Análisis básico de algoritmos).

⊗ **Solo fase nacional**
Algunos tópicos avanzados serán marcados como *solo fase nacional*. El Comité Científico garantizará que para todos los problemas en la fase regional será posible obtener puntaje máximo sin conocimiento sobre los tópicos en esta categoría.

Es intención del Comité Científico aumentar gradualmente la complejidad de los tópicos cubiertos en este documento. Por esta razón, cada año el Comité Científico puede decidir incluir tópicos avanzados no cubiertos previamente o catalogar como básicos ciertos tópicos anteriormente considerados como avanzados. No obstante, entendiendo la diversidad de los participantes, es también compromiso del Comité Científico preparar en cada competencia un conjunto de tareas adecuado para todos los niveles. Este compromiso se verá reflejado en la variedad de subtareas propuestas para cada problema (referirse a apéndice A).

4. Conceptos fundamentales de programación

4.1. Componentes fundamentales de lenguajes de programación

♡ Sintaxis y semántica de lenguajes de programación en al menos uno de los lenguajes aceptados en la OCI (Java y C++): Creación de programas, inclusión de librerías, compilación.

♡ Variables: declaración de variables, tipos de variables, asignación.

♡ Instrucciones de I/O: instrucciones de input/output desde la entrada/salida estándar, término de input, término de línea en output. Para detalles técnicos específicos referirse a apéndice C.

♡ Comparaciones: igualdad, distinto, menor (o igual), mayor (o igual).

- ♡ Expresiones Booleanas: uso de conectores lógicos (conjunción, disyunción, negación), uso de paréntesis, evaluación de una expresión.
- ♡ Estructuras de control de flujo `if`/`else`: control de flujo simple, que requieran uso de expresiones Booleanas compuestas (conjunción, disyunción, negación), `if`/`else` anidados.
- ♡ Estructuras de control de iteración `for`/`while`: ciclos `for`/`while` simples, con condiciones compuestas (conjunción, disyunción negación) para su terminación, ciclos anidados.
- ♡ Funciones: Sintaxis para la declaración de funciones, paso de parámetros, valor de retorno, llamado a funciones, uso de funciones en expresiones, llamado a funciones de librerías incluidas.

4.2. Tipos de datos básicos y operaciones

- ♡ Tipos Booleanos: uso de tipos Booleanos en expresiones lógicas (conjunción, disyunción, negación), en control de flujo y estructuras iterativas.
- ♡ Enteros: manejo de números enteros, lectura/escritura de enteros desde/hacia entrada/salida estándar.
- ♡ Expresiones aritméticas sobre enteros: uso de operadores entre enteros incluido resto/módulo (`%`), división entre enteros, exponenciación y otras funciones de librerías matemáticas básicas, expresiones bien formadas en uso de paréntesis.
- ♡ Strings: lectura y escritura de strings, manipulación básica de strings.

4.3. Estructuras de datos simples

- ♡ Arreglos: arreglos de tipos de datos básicos, creación de un arreglo de tamaño fijo, creación de un arreglo de tamaño especificado por una variable, acceso a las posiciones del arreglo.
- ♡ Recorrer arreglos: recorrer arreglos para calcular alguna operación entre sus elementos, para obtener máximos, mínimos, promedios, contar la aparición de valores, etc., comparar arreglos, determinar elementos en común, subsecuencias, etc.
- ♡ Matrices (arreglos de dos dimensiones): matrices de tipos básicos, creación de matriz de tamaño variable, acceso a las posiciones de la matriz.
- ♡ Recorrer matrices: recorrer matrices para calcular alguna operación entre sus elementos, para obtener máximos, mínimos, promedios, contar la aparición de valores, etc., comparar matrices, determinar la existencia de submatrices, etc.
- ♡ Arreglos de tamaño dinámico: creación y manipulación de arreglos de tamaño dinámico provistos por la librería del lenguaje de programación escogido. Acceso a una

posición arbitraria en el arreglo. Aumentar o disminuir tamaño de un arreglo añadiendo o eliminando elementos del final de este. Para información técnica detallada referirse a apéndice D.1.

⊖ Definición de nuevos tipos de datos simples: definición de estructuras para encapsular un conjunto de datos simples, por ejemplo, estructura para representar pares de números. Definición de operaciones (funciones) sobre nuevos tipos de datos.

⊖ Arreglos de tipos distintos de los básicos, por ejemplo, arreglo de pares de números.

4.4. Recursión

♡ Concepto de recursión.

♡ Funciones matemáticas recursivas.

♡ Implementación de funciones recursivas básicas.

♡ Implementación de funciones mutuamente recursivas.

5. Algoritmos y estructuras de datos

5.1. Análisis básico de algoritmos

⊖ Análisis de cotas superiores asintóticas (de manera informal).

⊖ Notación de Landau (big $\mathcal{O}$ notation).

⊖ Clases de complejidad estándar (constante, logarítmica, lineal, $\mathcal{O}(n \log n)$, cuadrática, cúbica, exponencial).

5.2. Ordenación y búsqueda

♡ Búsqueda lineal en secuencias de datos (no necesariamente ordenados).

♡ Búsqueda lineal en secuencias de datos ordenados.

⊖ Algoritmos de ordenación: algún algoritmo simple de ordenación de valores enteros como InsertSort, SelectSort, BubbleSort; de menor a mayor, de mayor a menor y con criterio alternativo.

⊖ Búsqueda binaria para encontrar elementos en una secuencia ordenada y en su forma general para encontrar puntos en funciones monótonas de dominio discreto.

5.3. Estructuras de datos y algoritmos provistos por la librería estándar

⊖ Funciones de ordenación: uso de las funciones de la librería estándar para ordenar arreglos de tipos básicos. Uso de las funciones de ordenación para arreglos de tipos distintos de básicos y con criterio de comparación personalizado. Para información técnica detallada referirse a apéndice D.2.

⊖ Uso de estructuras que implementen conjuntos de elementos, con conciencia de su eficiencia por sobre una implementación basada en arreglos y búsqueda lineal. No se espera conocimiento sobre la implementación subyacente de estas estructuras (tablas de hash, árboles auto-balanceantes). Para información técnica detallada referirse a apéndice D.3.

⊖ Uso de estructuras que implementen tablas de asociación (clave,valor), con conciencia de su eficiencia por sobre una implementación basada en arreglos y búsqueda lineal. No se espera conocimiento sobre la implementación subyacente de estas estructuras (tablas de hash, árboles auto-balanceantes). Para información técnica detallada referirse a apéndice D.4.

⊖ Uso de estructuras que implementen colas de prioridad, con conciencia de su eficiencia por sobre una implementación basada en arreglos y búsqueda lineal. No se espera conocimiento sobre la implementación subyacente de estas estructuras (árboles auto-balanceantes). Para información técnica detallada referirse a apéndice D.5.

5.4. Algoritmos numéricos

♡ Dígitos de un número entero: descomponer un número entero en sus dígitos constituyentes.

♡ Operaciones usando módulo: aritmética modular, reportar los últimos dígitos de un número grande.

♡ Números primos: determinación de primalidad de un número por búsqueda exhaustiva.

⊖ Números primos: determinación de primalidad de un número por búsqueda exhaustiva hasta la raíz cuadrada.

5.5. Grafos

⊗ Noción de grafos como matriz de adyacencia y como listas de adyacencia: grafos simples (a lo más una arista entre cada par de nodos) dirigidos y no dirigidos, con y sin ciclos.

⊗ Noción de grafos con peso: grafos con peso en nodos, peso en aristas.

⊗ Noción de grafo implícito en estructura de adyacencia (ej. grafo generado en un tablero donde los vecinos son los casilleros de arriba, abajo, izquierda y derecha).

⊗ Algoritmos de recorrido en grafos: BFS y DFS.

⊗ Algoritmos de distancia mínima entre nodos en grafos con peso (Dijkstra) y sin peso (BFS).

5.6. Estrategias algorítmicas

⊗ Programación dinámica

6. Matemáticas

6.1. Aritmética y geometría

♡ Enteros y sus operaciones (incluida exponenciación)

♡ Propiedades básicas de enteros (signo, paridad, divisibilidad)

♡ Aritmética modular simple: suma, resta y multiplicación.

♡ Teorema de Pitágoras.

A. Descripción de un problema de la OCI

Esta sección describe la estructura típica de un problema en la OCI, la división en distintas subtareas, la forma de asignar puntaje, cómo se corrigen durante la competencia y el feedback que reciben los participantes.

A.1. Estructura de los problemas

En general, todo problema de la OCI comienza con una descripción (entregando contexto y motivando típicamente desde una situación cercana a la realidad). Después de analizar el problema, el estudiante podrá entender cómo resolverlo usando técnicas de programación. El siguiente ejemplo muestra un enunciado resumido del problema "Palabras" de la OCI 2013.

Ejemplo A.1 *Para crear palabras, una tribu extraterrestre usa solo un conjunto reducido de letras y todas las combinaciones de letras forman palabras válidas. Se requiere saber cuántas palabras de un largo fijo se pueden formar para un conjunto de letras dado. Por ejemplo, si las letras que se usan son '$'y '&', entonces las palabras distintas de largo 3 que se pueden formar son $$$, $$&, $&$, $&&, &$$, &$&, &&$ y &&&, que en total son 8. No se pide el número total, sino que basta con conocer los últimos 4 dígitos de ese valor.*

Luego de la descripción del problema se describen la forma en que se entregará la entrada/input junto con las restricciones de los valores, y la forma en que debe ser entregada la salida/output por parte de la solución. Por ejemplo, en el caso del problema "Palabras" descrito anteriormente, la entrada era un par de números enteros N y L, donde N corresponde a la cantidad de letras usadas y L al largo de las palabras. La salida debía ser un entero correspondiente a los 4 últimos dígitos de la cantidad de palabras de largo L que se pueden crear con N letras. En el caso de la entrada, la restricción era que N podía tomar valores entre 1 y 10, y que L podía tomar valores entre 1 y $10^9 - 1$.

Posteriormente, el enunciado del problema describe las distintas subtareas que se someterán a evaluación y el puntaje por completar cada una. Cada subtarea tendrá restricciones adicionales al problema que pueden hacer que su solución sea más simple. En general las distintas subtareas van aumentando su nivel de dificultad y una solución a la última subtarea (la más difícil) debiera abarcar todas las subtareas anteriores. Esta es una característica común, pero no una regla estricta. Es posible que la solución a una subtarea no resuelva las subtareas anteriores. El siguiente ejemplo muestra las subtareas del problema "Palabras" de la OCI 2013.

Ejemplo A.2 *Se probarán varios casos con los siguientes puntajes y restricciones:*

	Puntaje	**Restricción**
Subtarea 1	*30 puntos*	$1 \leq L < 4$
Subtarea 2	*30 puntos*	$4 \leq L < 12$
Subtarea 3	*20 puntos*	$12 \leq L < 10^8$
Subtarea 4	*20 puntos*	$10^8 \leq L < 10^9$

El puntaje total de un problema, sumando los puntajes de cada subtarea que lo componen, será siempre de 100 puntos, tal como en el ejemplo anterior. En dicho ejemplo se puede notar cómo el problema va aumentando la dificultad para cada una de las subtareas.

A continuación se hace un análisis del problema "Palabras" y la solución más simple para cada una de las posibles tareas.

Lo primero que se puede observar en el problema "Palabras" es que la cantidad de palabras distintas de largo L que se pueden hacer con N letras es simplemente N^L. La segunda observación es que ese número podría ser muy grande dependiendo de los valores de N y L y que por lo tanto podría ser muy lento de calcular y su valor podría no ser almacenable en un tipo de dato básico. Con esas observaciones se pueden utilizar las siguientes estrategias:

Subtarea 1 : Dado que N es siempre menor o igual a 10 y L es menor que 4, sabemos que N^L tendrá menos de 4 dígitos. Luego, en este caso bastaba con entregar como salida el valor N^L. Más aún, el cálculo de N^L se podría hacer de forma muy simple: dado que los valores posibles de L son $1, 2, 3$, se podría hacer una implementación en donde la salida fuera, o bien N, o $N \times N$, o $N \times N \times N$, dependiendo del valor de L, lo que puede ser implementado incluso usando solo `if`/`else` y operaciones básicas entre enteros.

Subtarea 2 : En este caso el valor N^L podría tener más de 4 dígitos, por lo que sería incorrecto simplemente entregar N^L como salida. Además, los valores de L podrían ser muchos (entre 4 y 11), y en consecuencia, era necesario hacer una iteración para calcular N^L. Dado que el valor máximo para N^L era 10^{11}, este número podría ser almacenado en una variable y posteriormente obtener los últimos 4 dígitos del resultado. Obtener los dígitos se podía hacer dividiendo y restando, o usando el operador `%`.

Subtarea 3 : En este caso el valor de N^L podría ser gigante (hasta 10^{10^8}), por lo que sería imposible calcular primero el valor, ya que no es almacenable en ningún tipo de dato. La observación que se podía hacer es que como solo se necesitaban los últimos 4 dígitos, entonces podríamos usar aritmética modular de manera tal de multiplicar N consigo mismo L veces usando una iteración, pero tomando el módulo (`%`) 10000 luego de cada multiplicación. De esta forma, todos los números intermedios tendrían a lo mas 8 dígitos, y cuando se terminara el cálculo se tendría exactamente el valor pedido.

Subtarea 4 : Esta es la tarea más difícil. En este caso el valor de L puede ser tan grande $(10^9 - 1)$ que incluso hacer una iteración desde 1 hasta L podría necesitar demasiado tiempo (mayor a 1 segundo, que era el tiempo máximo de ejecución).
Una solución a este problema que obtenía todo el puntaje era notar que, o bien L o $L - 1$ se podía dividir por 2 para hacer el cómputo más rápido, de la siguiente forma: (1) si L es par, entonces se calcula $N^{L/2}$ y luego se eleva al cuadrado, (2) si L es impar, entonces se calcula $N^{(L-1)/2}$ y luego se eleva al cuadrado y se multiplica por N. En ambos casos se debía tomar módulo 10000 después de cada cálculo

intermedio. Esta solución llevaba a la mitad el tiempo necesario para calcular N^L. Otra solución general, que alcanzaba la mejor eficiencia, era usar *exponenciación rápida*que es esencialmente la idea anterior pero dividiendo L de manera recursiva. Esencialmente se calcula una función recursiva $\text{expRap}(N, L)$ de manera tal que

$$\text{expRap}(N, L) = \begin{cases} \text{expRap}(N, L/2) \times \text{expRap}(N, L/2) & \text{si } L \text{ es par} \\ N \times \text{expRap}(N, (L-1)) & \text{si } L \text{ es impar} \end{cases}$$

con caso base $\text{expRap}(N, 1) = N$ y de manera que en cada llamada intermedia se tome el módulo 10000.

Este es un problema claramente difícil para obtener el puntaje completo. Note además que cualquier solución para una de las tareas resolvía satisfactoriamente todas las tareas anteriores.

A.2. Evaluación y feedback

Cuando un competidor envía una posible solución para ser evaluada durante la competencia, el sistema de evaluación ejecutará su programa con todos los casos de prueba disponibles y una vez obtenidos los resultados se le notificará al competidor del puntaje obtenido. Los casos de prueba están divididos en subtareas y para obtener el puntaje en una subtarea se debe tener correctos todos los casos de prueba de la subtarea. El sistema de evaluación entregará al participante el detalle de todos los casos que fueron resueltos correctamente y aquellos en los que hubo un error, pero el contenido de los casos de prueba siempre permanecerá oculto. El puntaje final de un participante corresponde a la suma de los puntajes máximos que haya sacado en cada problema.

B. Lenguajes de programación acceptados en la competencia

Los lenguajes permitidos en la versión 2023 de la OCI son C++ y Java. En su versión 2022, la OCI aceptó de forma piloto el envío de soluciones en Python. Dada la diferencia en los tiempos de ejecución entre Python, C++ y Java, y la experiencia recopilada durante la versión 2022, el Comité Científico ha decidido no aceptar el envío de soluciones en Python para la versión 2023 de la OCI.

C. Operaciones de entrada y salida (IO)

Cada programa enviado como solución en la OCI es evaluado ejecutándolo con distintos casos de prueba. Un caso de prueba corresponde a una instancia del problema y está compuesto por una entrada y una salida esperada. La solución será ejecutada una vez por cada caso de prueba, entregándole a esta la entrada a través de la entrada estándar. El

programa deberá imprimir el resultado del problema por la salida estándar. La salida del programa es posteriormente comparada con la salida esperada para el caso de prueba. El comportamiento por defecto es hacer una comparación *exacta* entre la salida del programa y la salida esperada para el caso de prueba (similar al resultado entregado por el comando `diff` en linux); por lo tanto, es importante que el programa respete el formato de salida esperado de forma estricta, considerando mayúsculas y minúsculas, espacios y saltos de línea. La entrada de cada caso de prueba también seguirá el formato estricto especificado en el enunciado del problema y por lo tanto las soluciones no deberán preocuparse de verificarlo.

Puede haber problemas donde existe más de una solución posible. En este caso se instruirá explícitamente en el enunciado que cualquier solución es válida y en vez de compararse la respuesta de forma exacta con la salida esperada, esta se evaluará mediante un *verificador*. Un verificador es un programa que lee la respuesta producida por una solución y verifica si es correcta.

La entrada y salida deben ser siempre por la salida estándar. A continuación se detallan las formas recomendadas de hacerlo en Java y C++.

C++. Existen dos formas de leer y escribir por la entrada y salida estándar en C++. La primera es usando las funciones `printf` y `scanf` del encabezado `cstdio`. La siguiente forma es a través de *streams* usando lo objetos `cin` y `cout` provistos por el encabezado `iostream`. Ambas formas están recomendadas y su uso solo depende de preferencias personales.

Java. La forma recomendada en Java para escribir a la salida estándar es utilizando las funciones `System.out.println` y `System.out.print`. Puede ser de utilidad conocer también la función `System.out.format`.

Por otro lado, la lectura de la entrada estándar es más sutil en Java. En Java puede resultar tentador utilizar la clase `Scanner`; sin embargo, esta forma de lectura puede resultar muy ineficiente pues no utiliza un *buffer*. Debido a esta ineficiencia, el Comité Científico no garantiza que las soluciones que utilicen `Scanner` obtengan el mayor puntaje en problemas que tengan restricciones ajustadas de tiempo. No obstante, su uso queda a criterio del competidor. En problemas donde el tamaño de la entrada es pequeño, utilizar `Scanner` puede no resultar significativo en el tiempo total de ejecución.

La forma recomendada de leer en Java es utilizando el método `readLine` de la clase `BufferedReader`. Para complementar su uso es recomendable utilizar la clase `StringTokenizer` y funciones como `Integer.parseInt`.

D. Algoritmos y estructuras de la librería estándar

El Syllabus de la OCI categoriza dentro de tópicos avanzados el uso de algunos algoritmos y estructuras de datos provistos en la librería estándar de los lenguajes. A continuación se detallan, para cada lenguaje, las funciones y estructuras que cumplen con la

expectativa en el Syllabus.

D.1. Arreglos de tamaño dinámico

C++. La clase `vector` implementa arreglos de tamaño dinámico en C++.

Java. En Java los arreglos de tamaño dinámico son implementados en la clase `ArrayList`.

D.2. Funciones de ordenación

C++. Entre otras funciones, el encabezado `algorithms` provee la función `sort`. Esta función puede ser utilizada tanto con la clase `vector` como con arreglos de tamaño fijo.

Java. Existen dos funciones básicas que implementan ordenación en Java. La primera es `Arrays.sort` que puede ser utilizada con arreglos de tamaño fijo. La segunda es `Collections.sort` que puede ser utilizada con objetos de la clase `ArrayList`.

D.3. Conjuntos

C++. Las clases `set` y `unordered_set` implementan conjuntos.

Java. Las clases `TreeSet` y `HashSet` implementan conjuntos.

D.4. Tablas de asociación clave valor

C++. Las clases `map` y `unordered_map` implementan tablas de asociación (clave,valor) en C++.

Java. Las clases `TreeMap` y `HashMap` implementan tablas de asociación (clave,valor) en Java.

D.5. Colas de prioridad

C++. La clase `priority_queue` implementa colas de prioridad.

Java. La clase `PriorityQueue` implementa colas de prioridad.