PENSAMIENTO COMPUTACIONAL: UNA IDEA A LA QUE LE LLEGÓ EL MOMENTO

En marzo de 2006, Jeanette Wing escribió un pequeño artículo que apareció en Communications of the ACM [Wing2006]. En el artículo, Wing propuso una idea atrevida: que además de aprender a leer, escribir y a realizar operaciones aritméticas básicas, todos deberíamos aprender pensamiento computacional. Dijo que el pensamiento computacional, que implica "resolver problemas, diseñar sistemas, y comprender el comportamiento humano a través de los conceptos fundamentales de la Ciencia de la Computación", es "una destreza fundamental para todos, no sólo para los científicos de la Computación".

CRISTIANBRAVO-LILLO

Doctor en Ingeniería y Políticas Públicas de Carnegie Mellon University; Ingeniero Civil en Computación, Universidad de Chile. Le interesa profundamente la mejora de la educación a través de la tecnología. Actualmente investiga en aprendizaje a través de juegos educativos en RSEduca.

cbravo@kind.cl | http://kind.cl/cbravo

Las ideas en el artículo no eran precisamente nuevas: de hecho, Alan Perlis ya había dicho en 1962 que todos deberían aprender a programar como parte de una educación liberal, y que esto llevaría a los estudiantes a repensar su comprensión de temas como cálculo y economía en función de conceptos computacionales [Guzdial2008]. Sin embargo, el artículo de Wing expuso de manera tan clara e influyente el tema, que comenzó una pequeña revolución. Hoy, vemos a Mark Zuckerberg, Bill Gates, Barack Obama, Michael Bloomberg y una veintena de otras personalidades, invitándonos a aprender a programar a través de iniciativas como "La hora de programación" (The Hour of Code), una ingeniosa herramienta web para que cualquiera, pero especialmente niños y niñas de seis años de edad en adelante, aprendan los primeros rudimentos de programación.

Al parecer, aprender a programar es una idea cuyo momento de surgir simplemente llegó.

La sutileza en la propuesta de Wing es que ella no propone que aprendamos a programar, sino a pensar de manera computacional. No se requiere un computador para aprender a pensar computacionalmente, aunque dicho proceso se enriquezca y acelere cuando uno escribe programas y los ejecuta en un computador. Por ejemplo, no se necesita un computador para aprender a ordenar una serie de documentos por su fecha, o alfabéticamente. Sin embargo, la mejor manera de entender cómo ordenar cualquier conjunto de objetos es programando un algoritmo de ordenamiento, y probando ordenar conjuntos arbitrarios de objetos. Esto no es otro sino el "aprender haciendo" de los constructivistas.

La distinción anterior (que todos deberíamos aprender pensamiento computacional, no programación) es importante. En 2012, Michael Bloomberg, quien entonces era alcalde de Nueva York, publicó en Twitter que su resolución de año nuevo era aprender a programar (**Imagen 1**). Algunos críticos del movimiento han argumentado, no sin cierta cuota de razón, que si personas como Bloomberg realmente necesitan aprender a programar para realizar su trabajo, entonces "algo anda terriblemente mal con la política en Nueva York" [Blog2012].

Por supuesto, no se trata de eso. Detengámonos un momento para pensar qué queremos decir con "pensamiento computacional". Una definición que personalmente encuentro muy acertada es la siguiente: "Pensamiento computacional es el proceso de reconocer aspectos de computación en el mundo que nos rodea, y aplicar herramientas y técnicas de ciencias de la computación para comprender y razonar acerca de sistemas y procesos naturales y artificiales". [RoyalSociety2012].

Dicho de otra forma: el pensamiento computacional es a los computadores como el pensamiento matemático es a las calculadoras. Antes de poder usar una calculadora, tienes que aprender las operaciones básicas (suma, multiplicación, etc.). Luego, para responder una pregunta cuya respuesta requiere de cálculos matemáticos (por ejemplo, "¿cuál es la velocidad final de un auto que acelera a 3 m/s^2 por 10 segundos?"), necesitas formular un plan en tu cabeza de qué es lo que quieres calcular ("velocidad final es velocidad inicial más el producto de la aceleración por el tiempo"). Luego, tienes que transformar ese plan en una serie de operaciones realizables por la calculadora ("3 por 10 igual…"), y ejecutar ese plan ("aprieto la tecla 3, luego la tecla *, ..."). Uno no necesita saber cómo funciona internamente una calculadora para usarla, pero sí necesita saber aritmética básica. También uno necesita saber cómo expresar una pregunta propia en el "lenguaje de la calculadora".

Si "pensamiento computacional" son todos aquellos modelos mentales que necesitamos para entender cómo resolver problemas a través de los computadores, entonces sí, todos necesitamos aprender urgentemente pensamiento computacional, incluso el alcalde de Nueva York. Es importante entender, tal como dice Ken Robinson, que el sistema escolar tradicional está completamente obsoleto, que los niños y niñas no están aprendiendo gracias a las escuelas, sino a pesar de ellas, y que al decidirnos por un currículum específico estamos diciéndoles "esto es lo que necesitas saber para desempeñarte exitosamente los próximos cincuenta años de tu vida", cuando al mismo tiempo no →

IMAGEN 1.
RESOLUCIÓN DE AÑO NUEVO DE MICHAEL BLOOMBERG EN TWITTER.

tenemos idea de lo que va a ocurrir dentro de los próximos cinco años [RobinsonTED2006]. El mundo ha cambiado radicalmente en formas que han sido imposibles de prever antes de que se produzcan los cambios; dado que no tenemos certeza sobre qué problemas van a enfrentar nuestros hijos dentro de cincuenta años, una de las pocas cosas efectivas que podemos hacer es enseñarles cómo expresar sus problemas en el "lenguaje del computador", de manera que puedan resolver ellos mismos los problemas que nosotros no somos capaces de prever.

¿Cuáles son esas nociones computacionales básicas que todos deberíamos estar aprendiendo? Luego de una revisión de la creciente literatura sobre el tema, Grover y Pea [GroverEtAl2013] sugieren los siguientes conceptos:

1. Abstracciones y generalizaciones (incluyendo modelos y simulaciones).
2. Sistemas de símbolos y su representación abstracta.
3. Noción algorítmica de "control de flujo".
4. Descomposición estructurada de problemas (modularización).
5. Pensamiento iterativo, recursivo, y paralelo.
6. Lógica condicional.
7. Restricciones de eficiencia y desempeño (performance).
8. Debugging y detección sistemática de errores.

Para cada una de las ideas anteriores podemos encontrar ejemplos en áreas que no están (en apariencia) directamente relacionadas con la Computación. Por ejemplo, la abstracción y generalización de características biológicas de las especies de plantas y animales nos permite clasificarlos en taxones; muchos sistemas de transporte urbano pueden ser comprendidos a través de simulaciones donde varios agentes actúan en paralelo, dando lugar a problemas típicos de la programación paralela como la concurrencia; en epidemiología, los modelos que representan la propagación de una enfermedad son modelos recursivos; etc. Existe además cierto consenso en que el pensamiento computacional es suficientemente distinto de otras disciplinas similares como la Matemática, Ingeniería, Ciencia, etc., como para enseñarlo de manera diferenciada [GroverEtAl2013].

Según cifras de code.org, a la fecha de este artículo más de 98 millones de personas alrededor del mundo han completado la "Hora de Programación", escribiendo casi 5.000 millones de líneas de código[1]; esto es, alrededor de 100 veces el tamaño de Windows Server 2003 [Blog2005], y 25 veces el tamaño de la distribución Linux Fedora 9 [McPhersonEtAl2008]. Este aparente éxito contrasta con el diagnóstico que se puede observar en dos reportes elaborados recientemente, uno en Estados Unidos en 2010 [WilsonEtAl2010] y otro en Inglaterra en 2012 [RoyalSociety2012]. Desde realidades muy distintas, ambos reportes llegan a resultados similares:

1. Los currículums nacionales en tecnología son extensos y ambiguos a la vez; pueden ser reducidos a habilidades muy básicas que pueden ser entregadas por profesores no especializados, o por profesionales sin formación en pedagogía, lo que refuerza la baja percepción de utilidad del currículum y la idea de que puede ser abordado por cualquier profesional, sin formación en pedagogía.
2. La Computación no es considerada por las autoridades escolares como un área de conocimiento fundamental que los estudiantes deban manejar al salir de la educación secundaria.
3. Los profesores no tienen el nivel de formación necesario para enseñar conceptos de Computación, y no poseen la confianza necesaria para ello; en parte ello se debe a que no existen cursos de formación o de actualización en el área.
4. Falta un lenguaje común mínimo entre la industria, la academia y el gobierno para referirse a las habilidades y conocimientos relacionados con la Computación; en particular, la sigla ICT (Information and Communication Technologies, equivalente a nuestra sigla TIC) tiene una connotación negativa fuerte.
5. El problema del número reducido de estudiantes que entran a la educación superior a carreras relacionadas con Computación se agudiza cuando nos enfocamos en mujeres, hispanos y afroamericanos.

El énfasis puesto en el aprendizaje de la programación en Estados Unidos e Inglaterra ha dejado en evidencia la muy baja cantidad de mujeres que se dedican a la Computación. Las niñas no se interesan por aprender Computación o Programación, y en términos generales, la encuentran "aburrida" y "de hombres". En Estados Unidos hoy sólo un 0.3% de las niñas que salen del high school escogen Ciencia de la Computación como "major" en la universidad. Iniciativas como Girls Who Code (girlswhocode.com) y Black Girls Code (blackgirlscode.com) pretenden paliar esta tremenda carencia.

A pesar de que en Chile[2] no tenemos estudios acabados sobre nuestra situación, es probable que en los próximos años enfrentemos los mismos problemas descritos arriba (¡lo que nos entrega la oportunidad única de tratar de evitarlos!) Por ejemplo, una rápida revisión del currículum en tecnología para la educación básica en Chile evidencia que:

1. Es un área transversal, donde se confunde el uso de software a nivel de usuario con la aplicación de tecnología en áreas demasiado diversas.
2. Posee muy pocas horas comparado con el resto del currículum, lo que es un indicador de la escasa importancia que se le atribuye en comparación con matemática o lenguaje.
3. Está orientado a la comprensión "de la relación del ser humano con el mundo artificial", donde se busca que los estudiantes "observen en su entorno los objetos y la tecnología que los rodea".

Aunque la definición anterior es suficientemente ambigua como para incluir la Ciencia de la Computación, una revisión rápida muestra que no existe espacio para las nociones computacionales básicas que constituirían la base del pensamiento computacional.

Uno de los pilares fundamentales descritos en los estudios en ambos países es la formación y perfeccionamiento de los profesores. ¿Existen herramientas que faciliten el trabajo a los profesores y mentores, a la hora de enseñar las operaciones básicas

1 Por supuesto, con mucha probabilidad esos cinco mil millones de líneas de código contienen un enorme nivel de redundancia.

2 http://www.curriculumenlineamineduc.cl/605/w3-propertyvalue-52053.html

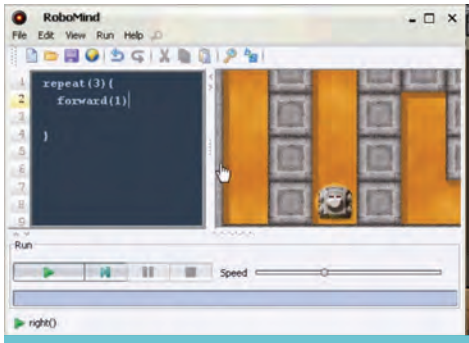

IMAGEN 2.
ENTORNO DE PROGRAMACIÓN ROBOMIND.

IMAGEN 3.
ENTORNO DE PROGRAMACIÓN CODE COMBAT.

de pensamiento computacional? Sí, literalmente decenas de ellas, aunque la mayoría está pensada para aprender a programar, no para aprender pensamiento computacional propiamente tal. Algunas de mis herramientas favoritas son:

1. **Scratch (*http://scratch.mit.edu/*):** proyecto del MIT Media Lab, que permite aprender a programar visualmente, a través de bloques que se arrastran y apilan uno sobre otro con el mouse.
2. **Snap (*http://snap.berkeley.edu/*):** una reimplementación de Scratch hecha por investigadores de Berkeley, que permite crear tus propios bloques, característica que Scratch no tiene.
3. **Google Blocky (*https://blockly-games.appspot.com/*):** una serie de juegos cortos diseñados por programadores de Google, en los que usan bloques muy similares a los de Scratch.
4. **Robomind (*http://www.robomind.net/es/*):** juego en el cual uno controla un robot que se mueve horizontal y verticalmente en una grilla con comandos muy simples, tecleados en una consola (**Imagen 2**). El conjunto es sencillo, aunque conceptualmente muy poderoso para enseñar a niños de menos de ocho años.
5. **Code Combat (*http://codecombat.com/*):** juego de aventuras (RPG) donde en cada etapa tenemos que cumplir una misión, dándole instrucciones al personaje principal en nuestro lenguaje preferido (Python y Javascript entre otros) (**Imagen 3**).

Finalmente, uno de los grandes problemas que los profesores enfrentan es cómo organizar una serie de lecciones alrededor de los conceptos básicos de pensamiento computacional. Incluso con las herramientas adecuadas, preparar una lección implica invertir una cantidad no despreciable de horas entendiendo cómo funcionan varias herramientas, escogiendo la más adecuada, y diseñando actividades para ser completadas por el grupo curso, además de una lección alternativa en caso de que la primera falle. El proyecto Bootstrap[3] ha generado módulos curriculares para profesores de matemáticas y computación, orientados a estudiantes de entre 12 y 16 años, quienes diseñan videojuegos muy simples, entendiendo en el camino cómo aplicar álgebra básica al movimiento de los personajes de juego.

En síntesis, existe una amplia variedad de herramientas disponibles para enseñar pensamiento computacional y programación (en ese orden). Existen también recursos para apoyar y facilitar la labor de nuestros profesores. Los estudios internacionales nos permiten entender lo importante de enseñar Computación en la educación primaria y secundaria, y de incluir la Computación en la formación de futuros y actuales profesores. Sabemos que nuestras hijas e hijos tendrán que resolver problemas que hoy no somos capaces de prever. Es nuestro deber y responsabilidad entregarles las herramientas adecuadas para que puedan hacerlo. ■

3 http://www.bootstrapworld.org

REFERENCIAS

[Blog2005] How Many Lines of Code in Windows? Blog personal de Larry O'Brien, publicado el 06/diciembre/2005. Disponible en http://www.knowing.net/index.php/2005/12/06/how-many-lines-of-code-in-windows/. Consultado el 08/junio/2015.

[Blog2012] Please don't learn to code. Blog personal de Jeff Atwood, publicado el 15/mayo/2012. Disponible en http://blog.codinghorror.com/please-dont-learn-to-code/. Consultado el 30/enero/2015.

[GroverEtAl2013] Computational Thinking in K–12: A Review of the State of the Field. S. Grover and R. Pea, 2013. Educational Researcher, 42(1), 38-43.

[Guzdial2008] Paving the way for computational thinking. Mark Guzdial, Communications of the ACM, agosto 2008, Vol. 51, No. 8, 25-27.

[McPhersonEtAl2008] Whitepaper: Estimating the Total Development Cost of a Linux Distribution. Amanda McPherson, Brian Proffitt, y Ron Hale-Evans. Octubre 2008, Linux Foundation. Disponible en http://www.linuxfoundation.org/sites/main/files/publications/estimatinglinux.html. Consultado el 30/enero/2015.

[RobinsonTED2006] How schools kill creativity. Ken Robinson. TED talk, febrero 2006. Disponible en http://www.ted.com/talks/ken_robinson_says_schools_kill_creativity. Consultado el 30/enero/2015.

[RoyalSociety2012] Shut down or restart? The way forward for computing in UK schools. The Royal Society, enero 2012. Disponible en https://royalsociety.org/education/policy/computing-in-schools/report/. Consultado el 30/enero/2015.

[WilsonEtAl2010] Running on Empty: The Failure to Teach K-12 Computer Science in the Digital Age. Cameron Wilson, Leigh Ann Sudol, Chris Stephenson, y Mark Stehlik. 2010, ACM y CSTA (Computer Science Teachers Association). Disponible en http://www.acm.org/Runningonempty/. Consultado en 30/enero/2015.

[Wing2006] Computational Thinking. Jeanette Wing, Communications of the ACM, marzo 2006, Vol. 49, No. 3, 33-35.