Revista *mexicana* de
bachillerato a distancia

Artículo
por invitación

¿Debe introducirse el aprendizaje de la programación de computadoras en el bachillerato?

Autor: Alejandro Pisanty

Must learning computer programming in baccalaureate be introduced?

Resumen

Discuto los argumentos a favor de introducir el aprendizaje de la programación de computadoras en el bachillerato, así como algunos en contra; además, propongo una vía de acción de carácter general.

Palabras clave: aprendizaje de la programación; pensamiento algorítmico; aprendizaje de la técnica de programación; lenguajes de programación

Abstract

In this paper I discuss arguments in favor of introducing computer-programming learning in baccalaureate, as well as some against it. Additionally, I propose a general course of action.

Keywords: Programming learning; Algorithmic thinking; Programming technique learning; Programming languages

Introducción

En el presente trabajo se usa de manera generalizada el término "programación" o "programación de computadoras". La palabra y la frase representan de manera condensada un campo muy amplio que contiene a las computadoras visiblemente identificadas como tales y a un conjunto muy amplio de dispositivos que son gobernados, total o parcialmente, por programas que toman decisiones con base en la captura de señales del exterior y con resultados a veces inmateriales y, en ocasiones, materiales. Todos tienen en común el contar con un procesador, una memoria, al menos para los programas y la operación, y un sistema de entrada y de salida. El procesador es un microcircuito electrónico digital. Estos objetos pueden ser tan diversos como teléfonos, automóviles, anteojos con capacidades de realidad virtual o aumentada y, progresivamente, todos los que se integren a la *Internet of Things*.[1]

Queda abierto el concepto de "programación", también en cuanto a múltiples paradigmas computacionales, tanto de *hardware* (básicamente se asume una arquitectura de Von Neumann, aunque la posición expresada es agnóstica al respecto) como de modelos de lenguajes de programación (hay una carga conceptual orientada a los procedimentales, pero también aquí la posición es independiente de estos paradigmas).

Igualmente se dejan de lado características de los lenguajes como que sean compilados o interpretados; se parte de una visión de lenguajes de alto nivel pero no se excluye a los "de máquina" o cercanos a éstos, como los que se utilizan en la programación de algunos juguetes y robots.

Toda referencia a marcas registradas y productos comerciales es ajena a su posible promoción o ataque. Todas las marcas y nombres comerciales citados son propiedad de sus titulares, en los países donde proceda.

Los procesos de aprendizaje de la programación implican, por lo menos, tres aprendizajes que se pueden considerar independientes: algoritmos, técnica de programación y lenguajes de programación. Cada uno de los anteriores tiene asociado un número apreciable de otras disciplinas.

El aprendizaje de los algoritmos se presenta de distintas maneras en diferentes niveles educativos. Sin duda, en educación superior en áreas científicas y de ingeniería es una disciplina matemática exigente y abstracta. En niveles educativos anteriores se puede reducir, en una mínima expresión útil, a la identificación y formulación de procedimientos, que deben ser completos por sí mismos, para que un autómata pueda ejecutarlos y lograr el resultado planeado.

Se puede inducir el pensamiento algorítmico con tareas sencillas, no computacionales, como las recetas de cocina, los procedimientos que se requieren para cruzar una calle, y muchos más. También se pueden analizar en el uso de programas de computadora de frecuente aplicación, como procesadores de texto o tabuladores electrónicos, a través de la propuesta de la automatización de secuencias de pasos utilizados repetitivamente, y mediante el uso de la facilidad de grabación por pasos de estos programas.

El aprendizaje de la técnica de programación requiere adquirir conceptos que aparecen con frecuencia en los algoritmos. Entre éstos se incluyen procesos repetitivos (iteraciones), ramificación de procesos como resultado de decisiones, inicio y terminación de procesos, ingreso y comunicación de datos y resultados, anidamiento de procedimientos y uso de programas que se encargan de procedimientos de manera autocontenida, etcétera.

[1] Tesis para obtener el título de Licenciado en psicología de José Manuel Sánchez Sordo, UNAM 2014.

El aprendizaje de lenguajes de programación tiene mucho en común con la adquisición de lenguas (distintas de la materna) en general. Sin embargo, debe observarse que la sintaxis de los lenguajes de programación es muy restringida (si bien, mucho más inflexible que la del lenguaje natural informal) y el número de instrucciones de los lenguajes de programación (el "vocabulario") también es un conjunto en extremo pequeño. Estos dos aspectos de los lenguajes de programación pueden ser adquiridos en plazos relativamente cortos; el dominio de un lenguaje de programación, por supuesto, no se alcanza sin avanzar también en las dos disciplinas ya descritas.

Debe tomarse en cuenta que existen lenguajes de programación orientados al fácil aprendizaje. Algunos de ellos cuentan con conjuntos de instrucciones extremadamente reducidos y parecidos al lenguaje natural (el paradigmático entre ellos es BASIC) o, incluso, la opción de construir los programas mediante una interfaz gráfica, sin escritura de instrucciones en forma de texto (Scratch, Mindstorms).

Programación, desarrollo de *software* y "coding" son tres términos estrechamente relacionados que en ocasiones se utilizan como sinónimos. Para los fines de este artículo podemos diferenciarlos señalando que la programación puede entenderse en un sentido estrecho como lo ya descrito o en uno amplio como el conjunto de actividades que llevan a la producción y ejecución de programas de computadora; el desarrollo de software sería sinónimo de esto último, especialmente si se consideran las disciplinas organizacionales y la planeación que conllevan. El término "coding", en inglés, ha entrado en boga en los últimos años y es esencialmente sinónimo de "programación", en el sentido lato.

En algunos países la tendencia a incluir la programación como un objeto de estudio en los programas educativos ha crecido recientemente, llegando incluso a ser considerada obligatoria, aun en la educación elemental (Pretz, 2014).

Beneficios

Aprender programación de computadoras conlleva beneficios educativos importantes (eClassroom News, 2014):

a. *Pensamiento computacional*. En años recientes se ha distinguido con claridad la importancia del "pensamiento computacional" como un componente indispensable de la educación actual. Se le considera como un complemento a las artes y a las letras, a la ciencia y tecnología, y a las matemáticas, pues está integrado al tejido de la mentalidad contemporánea. No se trata sólo del conocimiento técnico de algunos usos de las computadoras, ni del aprecio por el impacto de la tecnología, sino de una modalidad de pensamiento y de una actitud. Los argumentos in extenso han sido presentados por Zapata-Ros (2014).

b. *Autonomía*. Los estudiantes de bachillerato deben ser capaces de producir, no nada más consumir, tecnología y productos y servicios de base tecnológica. Este mandato va mucho más allá de la escuela, es el del país. La edad crítica para abrir esta ruta al desarrollo es la de la educación media superior. No todos los estudiantes serán ingenieros en computación, aunque cualquiera que sea su profesión, oficio u ocupación, debe poder formular una solución computacional a un problema y originar que aquella sea creada. Las oportunidades de generar ingresos, ganar valía en las organizaciones y transformar la realidad, se multiplican.

c. Aprecio por el automatismo de los objetos controlados por programas y otros aspectos fundamentales considerados por Vinton Cerf (McFarland, 2014).

d. *Comprensión de algunas implicaciones importantes del automatismo*. Todos los objetos a nuestro alrededor que han sido programados se comportan –en principio– de acuerdo con los programas mismos. Las desviaciones de este comportamiento sólo pueden deberse a efectos aleatorios y a fallas. Cuando los objetos se comportan de maneras no deseadas debemos investigar si esa conducta es producto del propio programa.

e. Los programas que rigen a los objetos de nuestro entorno son materializaciones de algoritmos. Unos y otros son productos humanos. Hasta ahora –y descartando para fines del argumento un debatible desarrollo de una inteligencia artificial suprahumana– las máquinas por sí solas no tienen agencia, ésta proviene siempre de humanos (y el azar). Si se desea cambiar el comportamiento de las máquinas (una cafetera programable, la búsqueda en Internet y la asociación de anuncios comerciales con aquélla, el sistema municipal de catastro y cobro de impuesto predial) se debe incidir en humanos y sus organizaciones para lograr el cambio.

f. Abstracción. Todo algoritmo –por ello, todo programa– se basa en abstracciones. Hacemos abstracción de muchos elementos concretos de la máquina (el tamaño del depósito de la cafetera automática, el color del edificio en el que se coloca un cajero automático bancario, los circuitos del procesador de la computadora) y diseñamos algoritmos para operar sobre el número mínimo indispensable de abstracciones para los fines del producto. Lograr el número mínimo de abstracciones puede requerir alcanzar el máximo nivel de abstracción. El algoritmo para cruzar las calles ignora detalles del sujeto que cruza la calle tan importantes como su edad y género; el programa de cobro de impuesto predial se construye dejando para otros programas la interfaz detallada a través de la cual interactuará con el ciudadano.

g. Una actividad de aprendizaje sobre programación con objetivos ambiciosos debe hacer que los alumnos razonen acerca del pensar (Corballis, 2007), acerca de cómo pensamos, acerca del pensamiento mismo. ¿Pensamos como computadoras? ¿Las computadoras piensan como nosotros?, ¿es concebible una aproximación o simulación entre ambas formas de pensar? ¿Son los seres humanos entes autónomos o autómatas que ejecutan programas? De ser autómatas, ¿dónde y por quién es escrito el programa? Las conclusiones a que se llegue en estos debates son de importancia primordial en las tensiones entre ciencia, tecnología y sociedad, en la discusión acerca del libre albedrío y sus implicaciones, así como en pugnas metafísicas que se pueden originar y anclar en el propio individuo. También son de gran importancia para la toma de decisiones en sociedad.

h. Un buen aprendizaje de la programación de computadoras acompañará el estudio de los algoritmos con el de las estructuras de datos. Iluminará el concepto de "base de datos" para darle un significado más preciso que el que se ha vuelto cotidiano (igualmente pasará con términos como "sistema" y otros fundamentales). El buen desempeño de un algoritmo, y en muchos casos su propia definición, depende de la organización que se dé a los datos sobre los que operará. Una estructura de datos adecuada al programa, no sólo al algoritmo, debe partir también de un conocimiento de los objetivos y los casos de uso del programa. Un algoritmo fácil de proponer –inocente, intuitivo– puede ser extremadamente ineficiente.

i. El concepto de escalabilidad entra de la mano del de estructuras de datos. Pocas veces se aprecia en nuestra sociedad la importancia de la escalabilidad; asumimos con frecuencia que lo que se diseña, observa u opera dentro del alcance de nuestros sentidos puede operar para miles de personas (o casas, o automóviles) y para millones o cientos de millones de operaciones. El programador descubre ipso facto lo contrario, y pasa a apreciar el denso trabajo contenido en los sistemas escalables, así como la importancia de diseños como el de Internet, que de hecho parte de una apreciable simplicidad para alcanzar escalamientos más que astronómicos.

j. La adquisición de un lenguaje de programación es por sí misma un punto atractivo de la educación en programación. Si bien resulta motivo de frustración, enriquece las estructuras mentales, provee cierta humildad ante el imperio de una gramática precisa e inflexible, y permite entender mejor la complejidad y el costo de los sistemas informáticos que nos rodean (visibles o no).

k. Entre las disciplinas que se deben adquirir para poder producir con éxito programas de computadora que funcionen de acuerdo a lo deseado está la conocida como *debugging* o "depuración de código". Con programas relativamente pequeños (de pocas instrucciones) el usuario tiene visibilidad sobre el programa completo; puede probarlo, observar fallas de sintaxis o de ejecución, volver sobre el programa escrito y realizar modificaciones que permitan mejorarlo, hasta alcanzar los resultados deseados. Este proceso no es fácilmente escalable: quien aprende programación está obligado a seguir más estrictamente la disciplina sintáctica, a diseñar pruebas rigurosas, a entender los resultados de distintos cambios en el programa y, con ellos, la "ingeniería inversa" (¿qué programa o parte de él pudo producir este resultado observado?) y, frecuentemente, a colaborar, por ello, comunicarse con otros programadores. No sobra mencionar que la comprobación de que una operación produce los resultados correctos (esperados de acuerdo con el programa) es una disciplina de utilidad general, la educación elemental parece haber abandonado el hábito de verificar sumas y multiplicaciones y quizás ello contribuya a que las pruebas de *software* sean vistas como una tarea superpuesta en lugar de una parte integral del proceso de desarrollo.

l. Quien ha llevado a cabo tareas de programación y depuración de programas de computadora puede apreciar mejor el valor de la documentación, lo que es extensible a actividades tan variadas como la gestión del hogar, oficina o taller; la realización de actividades manuales, los recorridos urbanos y muchas otras.

m. Optimización. Los programas de computadora pueden producir resultados correctos sin dejar de requerir mejoras. Las primeras mejoras que requiere todo programa, una vez garantizado que sea correcto, se enfocan en reducir el uso de recursos computacionales. En esta tarea se busca reducir el uso de espacio en la memoria y el disco o dispositivo de almacenamiento permanente, el número de operaciones (en el procesador, de acceso a la memoria, de entrada al almacenamiento permanente, de comunicaciones, tanto total como por unidad de tiempo). Lograr eficiencia en un programa de computadora necesita de una combinación de ciencia y arte: comienza en la elección de un algoritmo adecuado para el tipo y volumen de datos que se van a procesar y para la arquitectura de las computadoras en las que el programa va a ser ejecutado, a su vez, descansa sobre la

escritura de programas parsimoniosos en el uso de recursos. No es inusual que se requieran más instrucciones para lograr una mayor eficiencia; por ejemplo, en algunos cálculos numéricos, los accesos a localidades de memoria deben ser programados individualmente en lugar de, lo que es más elegante, calcular índices mediante fórmulas en ciclos recursivos.

n. *Recursividad*. La recursividad es un concepto que adquirió particular relieve cultural a partir de la publicación y difusión masiva del libro *Escher, Gödel, Bach – an Eternal Golden Braid*, de Douglas Hofstadter, en la década de 1980. En programación de computadoras el concepto de recursividad es fundamental y se ilustra de maneras particularmente sencillas, que permiten a su vez apreciarlo en estructuras mucho mayores y no nada más computacionales.

o. *Parsimonia y economía de medios*. Como se ha mencionado, la optimización de los programas de computadora requiere buscar la forma de realizar las tareas regidas por los programas con la mayor economía de medios. Vuelvo a abordar este concepto en un contexto específico: al hablar de programación en este artículo, estoy manteniendo una referencia muy general a "computadoras", como se explicó en la introducción. El estudiante de bachillerato que aborda en una clase formal la programación suele hacerlo con programas breves y que operan sobre conjuntos limitados de datos, en computadoras personales cuyas capacidades de almacenamiento y proceso exceden en órdenes de magnitud las necesidades de esos programas. Sin embargo:

    - No siempre ha sido así. Las primeras décadas de la computación se basaron en el uso de computadoras con recursos limitados y obligaban a los programadores a extremos de eficiencia para poder realizar las tareas de cálculo deseadas. Solamente mucho después del advenimiento de la computadora personal se empezó a programar en contextos de exceso de recursos. En éstos, algunos problemas se resuelven mejor con programas sub óptimos y recursos abundantes, dado que el costo de la optimización puede ser elevado, en comparación con el de los recursos computacionales. Incluso en estas situaciones conviene que los estudiantes estén conscientes del ethos del programador, que busca parsimonia y optimización.

    - No todos los dispositivos son igualmente permisivos. Después de algunos años de crecimiento constante en la capacidad de las computadoras y dispositivos programables relacionados, la programación y uso de programas se ha extendido a dispositivos de recursos limitados como teléfonos móviles, cámaras, reproductores de música, relojes, anteojos con capacidad de realidad virtual y realidad aumentada, etcétera, a nivel de consumidor, y a dispositivos muy complejos contenidos dentro de sistemas de uso general como los automóviles. Una estimación reciente adjudica más de cien millones de líneas de programa a los automóviles de uso corriente en la actualidad. La programación para estos dispositivos exige parsimonia y optimización. Esto puede parecer lejano a las necesidades de los estudiantes, a no ser que se reconozca el explosivo crecimiento del mundo de las apps, programas de computadora para uso en dispositivos móviles con objetivos de utilidad inmediata, a cuyo desarrollo comercial exitoso aspiran muchos jóvenes. Las apps requieren parsimonia.

p. *Pruebas*. Es por completo excepcional que un programa funcione correctamente en la forma en que es escrito por primera vez. Es casi una constante universal que la primera escritura contenga errores de interpretación del algoritmo, errores en la representación del algoritmo en el programa, erratas sintácticas en el lenguaje, y errores de programación. El ciclo de pruebas y correcciones es una fuerte prueba a la disciplina, la paciencia y la tenacidad de los programadores. Sólo mediante pruebas es posible verificar que el software corresponda al diseño. Mientras más complejo se vuelve el *software*, más lo son las pruebas requeridas. Esto es tanto más cierto cuando el *software* está formado por módulos independientes y desacoplados, de tal forma que las situaciones que puede enfrentar son impredecibles. Diseñar, planear, ejecutar e interpretar pruebas es un gran desafío. Existen métodos formales para generar pruebas en algunos sistemas pero, en general, en proyectos de gran escala se debe asumir que no es posible probar todos los casos. Los estudiantes se pueden beneficiar de saber que existen diversos niveles de prueba en software: unitarias, de integración, de rendimiento y de estrés. La misma descripción se puede aplicar a otros frentes de la experiencia humana, háblese de literatura, legislación, ingeniería civil o procedimientos médicos.

q. Los programas de computadora tienen, como es bien sabido, características funcionales y no funcionales. Las primeras se refieren directamente a la ejecución de las tareas a las que está destinado el *software*: calcular correctamente impuestos, representar en pantalla la forma de un objeto. Las segundas incluyen características como seguridad, robustez, eficiencia, resiliencia, usabilidad y acceso universal. Aun cuando un estudiante realice pocas (o ninguna) tareas de programación, será toda su vida un usuario de computadoras y, por ello, de programas. Conviene que una persona educada a nivel de bachillerato sepa identificar las características citadas, evaluarlas y exigirlas, y especificarlas en las situaciones posibles en la vida profesional en que sea responsable o corresponsable de adquisiciones para empresas o el gobierno.

r. *Fuente abierta, economía de compartir y "ética* hacker*"*. Las frases hechas y los estereotipos acerca de la figura del *hacker* (de nocivo a creativo pero siempre marginal) y de las construcciones colectivas de software en comunidades no comerciales orientadas ante todo a compartir el camino para la solución de problemas han sido fuerza generatriz fértil para mitos y mistificaciones. Unas cuantas horas escribiendo programas de computadora pueden tener un efecto saludable para que los estudiantes de bachillerato aprecien el esfuerzo que implica hacerlo y valoren las opciones y decisiones de las comunidades, tanto de quienes comercializan *software* y servicios basados en éste en un paradigma "propietario", basado en la propiedad exclusiva sobre los programas, hasta ideologías como la de Stallman, que impulsa una visión extrema de propiedad colectiva del *software* como un bien común.

s. *Formalidad*. En los intersticios y en los márgenes del sistema educativo formal, una población difícil de cuantificar aprende programación informalmente. Como consecuencia puede llegar a ser competente en la solución de problemas y la elaboración de código útil y eficaz. Sin embargo, la falta de bases formales (algoritmos, matemáticas discretas, estructuras de datos, pruebas, escalabilidad) la lleva a producir código de baja cali-

dad. Puede iniciar una vida laboral lucrativa pero corre el riesgo que los déficits le permitan alcanzar un nivel máximo muy bajo y conlleven la frustración correspondiente. Un mínimo de bases formales puede hacer diferencia directamente y encaminarle a adquirir mejores competencias en ciclos de estudio posteriores.

t. *Conceptos fundamentales de tecnología*. La programación provee una oportunidad única para el descubrimiento y aprendizaje de conceptos como modularidad, estandarización y arquitecturas de capas, que son de uso extenso en la tecnología no computacional. Definir a un objeto por sus *input*, *output* y función haciendo abstracción de su funcionamiento interior se aplica desde las instalaciones eléctricas domésticas y los equipos de las cocinas, hasta los más complejos desarrollos tecnológicos (aviones, por ejemplo). No es menor la importancia de dichos conceptos para comprender los sistemas sociales.

u. *La tendencia creciente al uso de Big Data*. Datos abiertos y gobierno abierto exigen ciudadanos capaces de analizar la información contenida en grandes bases de datos, mediante la combinación de habilidades matemáticas, estadísticas y computacionales. Todo egresado universitario debe ser capaz de participar en la explotación de esta clase de información.

Argumentos en contra

a. Se argumenta en contra del aprendizaje de la programación que ésta no es una habilidad que toda la población necesite, que no se cuenta con maestros, equipos, aulas y sistemas para darle soporte, y que el aprendizaje de la programación requiere tiempo y energía de los estudiantes que compite con otros usos –desde la adquisición de la lengua materna y las matemáticas básicas hasta la educación física, la música y la vida cívica.

b. Los argumentos opuestos a la inclusión de la programación en las escuelas varían desde posiciones ludditas y conservadoras hasta las específicas por nivel educativo y objetivo planteado en los programas.

c. Los programas de las asignaturas de computación, su práctica y la competencia de sus docentes en la mayoría de las instituciones educativas tienen un retraso de, por lo menos, dos décadas.

Síntesis y propuesta

a. El aprendizaje de la programación debe ser considerado en algún punto de la formación preuniversitaria, de acuerdo con las condiciones de los distintos sistemas educativos.

b. Las instituciones educativas deben tomar decisiones acerca de la forma, nivel de profundidad y diversidad de opciones que pueden ofrecer para el aprendizaje de la programación. La multiplicidad de oportunidades actuales –aprendizaje formal de algoritmos y lenguajes, uso de "macros" en software de uso general, robótica educativa, cultura *maker*, etcétera– provee una riqueza nunca vista de opciones.

c. El alcance de los proyectos no debe quedar limitado por el número de los profesores especializados en asignaturas informáticas, debe aumentar mediante el reclutamiento de profesores en otras asignaturas que puedan contribuir, incluso, con capacidades limitadas (mientras estén al día).

d. Las instituciones deben buscar las alianzas que puedan producir resultados, aunque sea

por plazos limitados. En este tema, la visión milenaria y de permanencia rayana en la inmanencia que impera en las grandes instituciones educativas es un severo impedimento para alcanzar resultados. Los *maker spaces*, los *hacker spaces*, *hackathones*, iniciativas de datos y gobierno abiertos, etcétera, proveen oportunidades para que los estudiantes desplieguen y adquieran habilidades, así como para captar instructores y evaluadores para las instituciones.

e. Las modificaciones curriculares y el reconocimiento de actividades semi o extracurriculares deben dejar parámetros muy generales y criterios de evaluación susceptibles de una evolución rápida en adaptación al medio.

f. Debe imperar una visión formativa, integral, reflexiva, sobre una mecanicista. Las reorganizaciones del trabajo docente deberán basarse en la capacidad de cada sistema educativo de adaptarse a nuevos requerimientos y aprendizajes.

Referencias

K. Pretz. (2014). Computer Science Classes for Kids Becoming Mandatory. *The institute. The IEEE news source*. Recuperado de http://theinstitute.ieee.org/career-and-education/preuniversity-education/computer-science-classes-for-kids-becoming-mandatory.

Zapata-Ros, M. (2014, 26 de noviembre). Por qué el pensamiento computacional. [Mensaje de blog]. Recuperado de http://computational-think.blogspot.com.es/2014/11/por-que-el-pensamiento-computacional-v.html. revisado 28/VI/2015.

McFarland, M. (2014). 5 insights from Vinton Cerf on bitcoin, network neutrality and more. The Washington Post. Recuperado de http://www.washingtonpost.com/blogs/innovations/wp/2014/10/10/5-insights-from-vinton-cerf-on-bitcoin-net-neutrality-and-more/

Corballis, M.C. (2007). *Pensamiento recursivo. Mente y cerebro*. (27). 78-87.

Harrell, M. (2015, 17 de marzo). Add coding to your elementary curriculum-right now. *Edutopia*. Recuperado de http://www.edutopia.org/blog/add-coding-elementary-curriculum-now-matt-harrell

Educational Technology and Mobile Learning. (2015, 14 de marzo). Two new resources on coding for teachers. Recuperado de http://www.educators-technology.com/2015/03/two-new-resources-on-coding-for-teachers.html

School of Data. (2015). *School of Data*. [Sitio web]. http://schoolofdata.org/courses/

Garritz Online Media. (2014). Hábitos multidispositivo en México 2014. [Documento PDF]. Recuperado de http://garritz.com/media/Habitos-Multidispositivo-de-Mexico-2014.pdf

Farnós, J.D. (2010, 14 de octubre). 150 herramientas gratuitas para crear materiales didácticos on line. *Juandon. Innovación y conocimiento*. Recuperado de https://juandomingofarnos.wordpress.com/2010/10/14/150-herramientas-gratuitas-para-crear-materiales-didacticos-on-line/

de Melo, G., Machado, A., Miranda, A., Viera, M. (2013). *Profundizando en los efectos del Plan Ceibal*. [Documento PDF]. Recuperado de http://www.ccee.edu.uy/jacad/2013/file/MESAS/Economia%20de%20la%20educacion_plan%20ceibal/Profundizando%20en%20los%20efectos%20del%20Plan%20Ceibal.pdf

Mejía, F. (2014, 19 de septiembre). Laptops, children and Darth Vader. *IDB Improving lives*. [Mensaje en blog]. Recuperado de http://ht.ly/HcAZf

Cobo, C. (2014). Currículo nacional en ciencias de la computación (National Curriculum for Computing): el ejemplo de Inglaterra. *Notas para Educación*. (19). Recuperado de http://www.ceppe.cl/images/stories/recursos/notas/notas_para_la_educacion_dic2014.pdf

Duarte, E. (2015, enero). 6 razones por las que la programación no es para todo el mundo. *Blog Corporativo CAPACITY*. [Mensaje de blog]. Recuperado de http://blog.capacityacademy.com/2015/01/23/6-razones-porque-la-programacion-es-para-todo-el-mundo/

Adell, J. (2010). *El diseño de actividades didácticas con TIC*. Recuperado de http://www.slideshare.net/epdrntr/jordi-adell-el-diseo-de-actividades-didcticas-con-tic-jedi2010-bilbao

López, J. C. (2015). SAMR, Modelo para integrar las TIC en procesos educativos. *Eduteka*. Recuperado de http://www.eduteka.org/samr.php

PHET. (2015). PHET Interactive Simulations Research and Development. Recuperado de http://phet.colorado.edu/es/research

Berry, M. W. y Browne, M. (2005). Software, Environments and Tools. *Understanding Search Engines: Mathematical Modeling and Text Retrieval*. Recuperado de http://epubs.siam.org/doi/abs/10.1137/1.9780898718164.ch1

UNESCO. (2008). *Estándares de competencia en TIC para docentes*. [Documento PDF]. Recuperado de http://www.eduteka.org/pdfdir/UNESCOEstandaresDocentes.pdf

Granger, C. (2015, 15 de febrero). *Why coding is not the new literacy*. Recuperado de http://qz.com/341447/why-coding-is-not-the-new-literacy/

Learn Python the Hard Way. [Sitio web]. Recuperado de http://learnpythonthehardway.org/book/ex0.html

RELATE. (s/f.). MAPS Pedagogy - Modeling Applied to Problem Solving. Recuperado de http://relate.mit.edu/current-projects/maps-pedagogy/

AUTOR

Alejandro Pisanty
Departamento de Física y Química Teórica
Facultad de Química, UNAM
apisan@unam.mx

AGRADECIMIENTOS

Agradezco los apoyos para la realización de este trabajo, del Departamento de Física y Química Teórica de la Facultad de Química de la Universidad Nacional Autónoma de México (UNAM), a Guadalupe Vadillo, José Luis Chiquete y, con especial profundidad y crítica experta, Erik Huesca, quienes revisaron versiones preliminares del texto final e hicieron comentarios valiosos que espero haber incorporado. Asumo la responsabilidad por no saber absorberlos.