Hipótesis 1.3 *Abstracción Computacional (Computational Abstraction)*
Al menos en el contexto del diseño de algoritmos, la abstracción (o poder de abstracción) sería un recurso computacional adicional e independiente del espacio y del tiempo.

De hecho, a modo de ilustración, podríamos incluso considerar el poder de abstracción como una tercera dimensión de los recursos computacionales (véase figura 2.4). Siendo un recurso añadido al tiempo y al espacio que nos da la oportunidad de buscar nuevos enfoques, donde el espacio y el tiempo sean polinómicos, aunque el poder de abstracción utilizado sea exponencial.

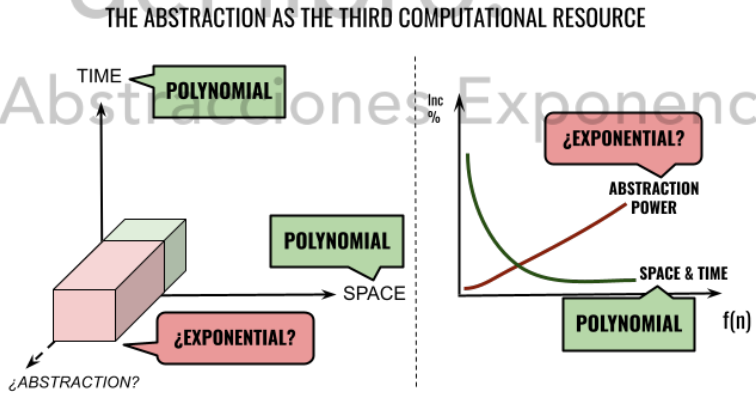

Figura 2.4: La abstracción como el tercer recurso computacional

Esto implica que la abstracción podría tener un estudio independiente de complejidad. Sin embargo, por el momento, y siguiendo nuestra línea informal de análisis de la complejidad, entiéndase este estudio como algo informal e implícito en el proceso de diseño de algoritmos, durante el cual nosotros, como diseñadores, podremos asociar ideas abstractas entre estas y argumentar su poder de abstracción, interesándonos especialmente en diferenciar entre poder de abstracción acotado (polinómico o contante) y no acotado (exponencial). Hablaremos de Poder de Abstracción Exponencial (EAP) en la medida en que una "pieza de información", mediante una lógica, pueda ser capaz de dar, a esa información, un número de ideas abstractas (significados) que crezcan en orden exponencial.

Ahora bien, si dicho análisis fuera más riguroso, debería estar siempre bajo el paraguas del diseño del algoritmo en cuestión, en la medida en que solo

podremos estudiar el poder de las abstracciones de una estructura de datos concreta si conocemos la lógica con la cual pretendemos codificarla y decodificarla pues, como hemos introducido, sin conocer la lógica, una información podría ser un simple conjunto de bits sin significado alguno. En cambio, conociendo una lógica concreta, este mismo conjunto de bits podría, por ejemplo, representar un conjunto con todos los Tours solución de una instancia I para nuestros casos de estudio HC o TSP. Con esto entiéndase que lo que pretendemos es explorar la posibilidad de que una pieza de información (estructura de datos) acotada especialmente podría tener para nosotros un valor exponencial, incluso a pesar de que, físicamente, el ordenador únicamente trabaje con una información acotada.

Cabe adelante, a modo de ejemplo, que, en nuestro caso de estudio, lo que nos interesará será poder disponer de una información acotada espacialmente (espacio polinómico) que pueda representar de forma abstracta un número exponencial de Tours solución.

2.4 La abstracción como puerta hacia $P = NP$

Todo esto dicho anteriormente nos llevó a preguntarnos:

Pregunta Clave de la Investigación (PCI) 1.1 *¿Podría ser la abstracción la puerta que nos permita resolver la conjetura P vs. NP a favor de que NP sea igual a P?*

Al tiempo que seguía luchando con los prototipos, cuando mi mente comenzaba a explorar la forma de sostener teóricamente ese tipo de estructuras basadas en el poder de abstracción exponencial que estaba intentando diseñar, una noche, llegó a mí una inspiradora conferencia de Barbara Liskov, titulada "How Data Abstraction changed Computing forever" (Liskov 2019 - TEDxMIT). En esta, Barbara Liskov narra en primera persona cómo el fluir histórico de las ideas de ilustres pioneros (como Edsger Wybe Dijkstra, Charles Antony Richard Hoare, David Lorge Parnas, Niklaus Wirth, entre otros) les dieron pie, a ella y a Stephen Zilles, para concebir una metodología de desarrollo y programación fundamentada en la abstracción de datos. Esta metodología fue finalmente publicada en 1974, en el ya histórico *paper* "Programming with Abstract Data Types" (Liskov and Zilles 1974). Abordaremos esta metodología más adelante (pág. 79) con el fin de dar soporte metodológico al tipo de estructuras de datos que propondremos. Sin embargo, por el mo-

mento, nos interesa resaltar el contexto histórico, pues esta metodología nace de la necesidad de dar respuesta y salida a la encrucijada histórica a la que se enfrentaba nuestra ciencia, conocida como la "Crisis del Software", oficialmente declarada por E. W. Dijkstra en 1968. Esta crisis, como bien narró Barbara en esta misma conferencia, se enmarcaba en un momento histórico en el que todo estaba comenzando. Por lo tanto, se desconocía cómo diseñar, organizar y estructurar el software. Esto, unido a la ausencia de metodologías de desarrollo claras y al uso de lenguajes de programación de bajo nivel, daba lugar a que el pan de cada día en nuestra industria fueran noticias sobre proyectos fracasados, sin mencionar que los pocos que lograban finalizar lo hacían por encima de los presupuestos. Estos sobrecostos y fracasos hacían mella en nuestra industria, mientras que, desde el mundo académico, muchos de los pioneros intentaban buscar la forma adecuada para estructurar y organizar los programas, así como el propio desarrollo software (Wikipedia 2021c).

El trazado histórico que ella hace a fin de argumentar cómo la abstracción de datos contribuyó de forma crucial a nuestra ciencia y llegó a cambiar para siempre el software fue, quizás, el empujón que me faltaba para estudiar de primera mano la contribución de la abstracción a nuestra ciencia. De este modo, comencé a profundizar más en serio, abordando durante varios meses una extensa revisión histórica de los orígenes de los lenguajes de programación (FORTRAN, LISP, COBOL, etc.), así como de las principales metodologías de programación (SP, OOP, etc.), a fin de buscar los orígenes de la Abstracción Software, y dilucidar, de primera mano, cuál ha sido la contribución real de la abstracción a nuestra ciencia. Este estudio histórico lo comencé partiendo de un compendio de los principales mecanismos de abstracción software que previamente había intentado hacer. Este trabajo paralelo de investigación de carácter histórico no fue acabado por completo y, en la actualidad, permanece *aparcado* debido, principalmente, a mi continua sensación personal de que, al sumergirme en el vasto mundo de la abstracción software, me estaba alejando de mi objetivo principal: enfrentarme al reto P vs. NP desde un enfoque práctico.

Sin embargo, cabe decir que ese buceo inacabado en las profundidades de la abstracción software no hizo más que reforzar, a cada paso, esa idea casi intuitiva del software que solemos tener hoy día, en que la abstracción lo rodea todo, en menor o mayor medida. De hecho, podemos comenzar a percibir esa abstracción en los propios lenguajes de programación de alto nivel, los cuales nos alejan del *lenguaje máquina*, así como también en los procedimientos y funciones que podemos definir fácilmente con estos lenguajes, dado que estas divisiones conceptuales nos permiten hablar en términos de una tarea o proceso

concreto, lo cual nos permite temporalmente dejar al margen el resto de las tareas del programa. Es decir, nos permite abstraernos pues, por simple que parezca esa división lógica de los procesos, ello nos permite ganar enfoque y perspectiva, lo que hace que podamos centrarnos en la implementación de responsabilidades específicas y lograr, de esta forma, crear mentalmente una división lógica —he aquí la abstracción— entre nuestro procedimiento o función y el resto del programa. Ello, a su vez, nos permite tomar una visión más general de todo nuestro programa, que se conformaría como un conjunto de funciones que individualmente tiene un único propósito concreto, tal como hoy en día las buenas prácticas defienden: destacar que, agazapada a la sombra de este tipo de procesos mentales, está la abstracción. Esta nos conduce a construir esas sutiles divisiones mentales que, a su paso, sutilmente, lo cambian todo.

Esto sucede también con los mecanismos de control del flujo de los programas donde, por ejemplo, nuestro ya viejo amigo, IF-ELSE, nos permite expresar sutilmente esa necesidad de distinguir entre dos situaciones, casos independientes que requerirán un flujo de trabajo diferente. Estas estructuras, inicialmente llamadas *estructuras de selección*, se unieron a las *estructuras iterativas* (por ejemplo, While, For, etc.), las cuales nos ofrecen una forma de expresar la necesidad de que un proceso de cómputo tenga que repetirse un número determinado de veces, o en función de cierta condición. Son amigos cotidianos para los programadores de hoy en día, que controlan el flujo de ejecución de nuestros programas, pero que no siempre estuvieron.

Hablamos pues, cómo no, de la evidente apuesta ganadora de la Programación Estructura (SP, por sus siglas en inglés) por establecer unas reglas mínimas que nos permitan poder seguir un flujo de ejecución ordenado coherentemente (O. J. Dahl, Dijkstra, and Hoare 1972, Böhm and Jacopini 1966), en claro contraste con los vaivenes y descontroles propios de los saltos incondicionales. Hablamos de los famosos GOTO, que no permitían a los programadores seguir el devenir de los procesos y acciones del programa, los cuales, en vez de ayudar a focalizarse, dispersaban la mente. Esto dificultaba (y mucho) el trabajo de equipos de programadores pioneros que, además, contaban con limitaciones tecnológicas importantes y, por ende, no siempre estaban abiertos a que lenguajes de alto nivel redujeran los pocos recursos de los que disponían sus ordenadores de la época. A fin de cuentas, incluso hoy en día, el uso de los lenguajes de alto nivel tampoco garantiza el éxito de un proyecto software aunque, siendo honestos, es mucho más fácil mirar la historia con los ojos del hoy cuando los lenguajes de programación de alto nivel traen consigo extensas librerías estándar, que nos ofrecen a los programadores es-