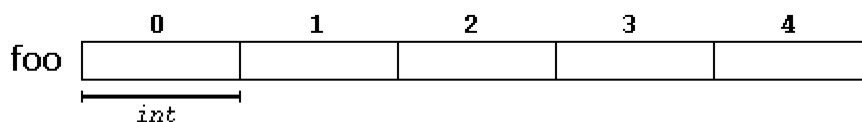
OIFem 2021: Tercera semana

Arrays y vectores
Inicializando arrays y std::vector
Accediendo a los valores de un array o vector
Arrays multidimensionales
La técnica de la ventana deslizante
Utilizando ventana deslizante:
Programación Dinámica
Bottom-Up: completar la tabla explicitamente.
Amnesia parcial: recordemos sólo lo necesario
El problema de la suma de subconjuntos

Arrays y vectores

Un array representa una secuencia de valores del mismo tipo, almacenados uno al lado del otro en memoria. Estos elementos pueden ser accedidos individualmente utilizando su *índice*.

Por ejemplo, para declarar cinco enteros, no es necesario hacerlo uno por uno, sino que utilizando un array podemos declararlos todos a la vez, y los cinco compartirán el mismo identificador (el nombre del array, en este caso "foo"):



cada posición representa, en este caso, un int. Los elementos están numerados de 0 a 4. Este número es el *índice* del elemento. Se puede pensar en estos índices como la numeración en las plantas de un edificio, donde la planta a nivel de suelo es el índice 0, la primera planta tiene índice 1, etc).

Los arrays se declaran así:

```
tipo nombre [número-de-elementos];
```

Por ejemplo, en el caso del array "foo" visto anteriormente:

```
int foo[5];
```

OJO: el número de elementos, es decir, el tamaño del array, ha de ser una *expresión constante*: dicho de otro modo, su valor ha de ser conocido antes de ejecutar el programa. Esta limitación nos lleva a introducir std::vector, similar a los arrays pero capaces de crearse, crecer y menguar dinámicamente.

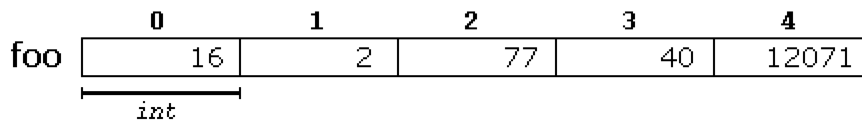
¿Cuándo utilizar arrays y cuándo std::vector? Si el tamaño es conocido de antemano, sin necesidad de ejecutar el programa, es en principio preferible utilizar arrays dado que son ligeramente más rápidos, ya que no requieren reservar memoria de manera dinámica, una operación costosa. Sin embargo, no siempre somos capaces de conocer el tamaño de la secuencia de antemano. En ese caso, necesitaríamos utilizar std::vector. También será necesario std::vector si el tamaño de la secuencia es variable.

Inicializando arrays y std::vector

Por defecto, los valores de un array *no* se inicializan automáticamente. Es decir, contienen "basura". Para darles un valor explícitamente podemos utilizar la siguiente sintaxis:

```
int foo[5] = { 16, 2, 77, 40, 12071 };
```

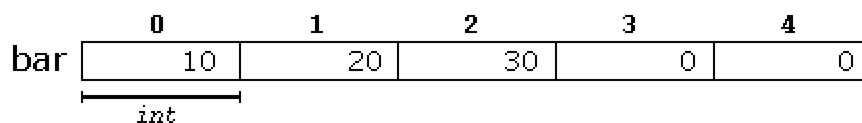
Esto resulta en:



El número de valores entre los {} no puede ser mayor que el tamaño declarado del array. Sin embargo, *sí* puede ser menor, en cuyo caso al resto de elementos se les da el valor por defecto del tipo (que en el caso de tipos como int, float, etc. es cero). Por ejemplo:

```
int bar[5] = { 10, 20, 30 };
```

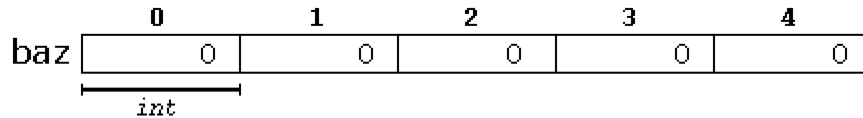
Resulta en:



Incluso podemos no poner ningún valor:

```
int baz[5] = { };
```

Lo que inicializará el array con ceros:



También podemos dejar que el compilador calcule el tamaño del array dada una lista de inicialización:

```
int foo[] = { 16, 2, 77, 40, 12071 };
```

std::vector nos permite especificar qué valor utilizar para su inicialización. Sin embargo, al ser una estructura dinámica, no suele ser necesario conocer los valores de inicialización. Si conocemos el tamaño (incluso si es de manera aproximada) del std::vector, podemos inicializarlo con ceros de la siguiente manera:

```
std::vector<int> v( N );
```

O también:

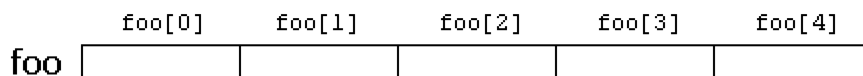
```
std::vector<float> v;
v.reserve(N);
```

Accediendo a los valores de un array o vector

La sintaxis es :

```
nombre[índice]
```

En el ejemplo anterior "foo", cada una de sus posiciones se accedería de la siguiente manera:



Para almacenar el valor 75 en la tercera posición (recordemos que los índices comienzan en 0):

```
foo[2] = 75;
```

Y para leer esa tercera posición, asignando su valor a otra variable "x":

```
x = foo[2];
```

Es muy importante no intentar acceder a una posición más allá del final de un array o vector. En el caso de los arrays, esto puede resultar en la terminación repentina del proceso (en el mejor de los casos), o en un fallo silencioso, o en cualquier otra cosa. Formalmente, acceder una posición más allá del final de un array tiene "comportamiento no definido". El problema es análogo en instancias de std::vector, aunque existe una manera de acceder a los elementos, utilizando el método .at(índice), que realiza comprobaciones para asegurarse de que "índice" es una posición válida. Sin embargo, "at()" no suele utilizarse, ya que es ligeramente más costoso.

Arrays multidimensionales

Los arrays o vectores multidimensionales pueden describirse como "arrays de arrays". Por ejemplo, un array puede visualizarse como una tabla de dos dimensiones, cuyos elementos son todos del mismo tipo:

jimmy	0	1	2	3	4
0					
1					
2					

"jimmy" representa un array bidimensional de 3 por 5 elementos. Se declararía mediante:

```
int jimmy[3][5];
```

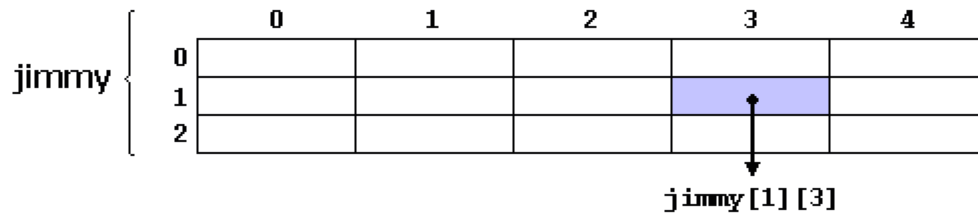
O, en el caso de utilizar std::vector,

```
std::vector< std::vector<int>(5) > jimmy(3);
```

ya que realmente estamos creando un vector que contendrá vectores. En este caso, el vector más externo, correspondiente a las filas, contiene 3 posiciones, cada una de las cuales tendrá dentro otro vector de 5 posiciones, las columnas.

Para acceder al cuarto elemento de la segunda fila, haremos:

```
jimmy[1][3]
```



Aunque en los problemas que veamos no será común, podríamos crear arrays de más de dos dimensiones, pero cuidado, ya que la cantidad de memoria necesaria se incrementa exponencialmente con el tamaño de cada nueva dimensión. Por ejemplo:

```
char century [100][365][24][60][60];
```

declara un array con elementos de tipo carácter, ¡con espacio para el número de segundos en un siglo! Esto se traduce en más de tres mil millones de caracteres, con lo que consumiría unos 3 gigabytes de memoria.

La técnica de la ventana deslizante

Esta técnica demuestra cómo reducir el coste computacional (número de operaciones) de algunos bucles.

Tomemos el siguiente problema como ejemplo:
Dado un array de enteros de tamaño "n", calcúlese la mayor suma de "k" elementos consecutivos.

```
Input  : arr[] = {100, 200, 300, 400}
         k = 2
Output : 700

Input  : arr[] = {1, 4, 2, 10, 23, 3, 1, 0, 20}
         k = 4
Output : 39
La mayor suma se obtiene del subarray {4, 2, 10, 23} de tamaño 4.

Input  : arr[] = {2, 3}
         k = 3
Output : Invalido
No existe un subarray con tres elementos.
```

Utilizando un enfoque de fuerza bruta, empezaríamos con el primer índice y sumaríamos hasta el k-ésimo elemento. Haríamos esto para cada uno de los "n-k" posibles índices iniciales.

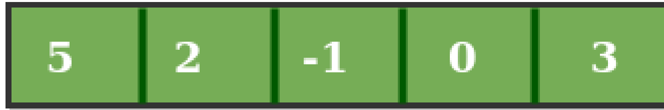
Posible implementación *a fuerza bruta*:

```
// Solución en tiempo O(n*k) para encontrar la máxima suma de un subarray
// contiguo de longitud k.
#include <bits/stdc++.h>
using namespace std;

// Devuelve la suma del máximo subarray de longitud k.
int maxSum(const vector<int>& v, int k) {
    int max_sum = INT_MIN;

    // Para cada bloque de long. k empezando en i...
    for (int i = 0; i < v.size() - k + 1; i++) {
        int current_sum = 0;
        // ... calcular la suma
        for (int j = 0; j < k; j++) {
            current_sum = current_sum + v[i + j];
        }
        // Potencialmente actualizar el resultado.
        max_sum = max(current_sum, max_sum);
    }
    return max_sum;
}

int main()
{
    const vector<int> v = { 1, 4, 2, 10, 2, 3, 1, 0, 20 };
    int k = 4;
    cout << maxSum(v, k);
    return 0;
}
```

Utilizando ventana deslizante:

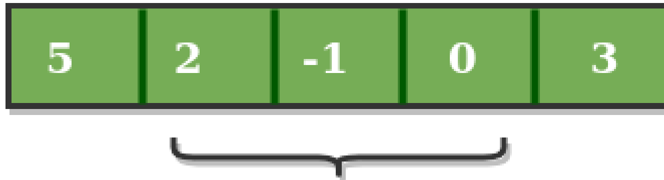
1. Calculamos la suma de los primeros "k" elementos, guardando el resultado, por ejemplo en "suma".
2. Avanzaremos "como un gusano", moviendo el extremo derecho de la "ventana" una posición hacia adelante, a la vez que también adelantamos el extremo izquierdo de la ventana para que ésta siga teniendo longitud "k".

Gráficamente:



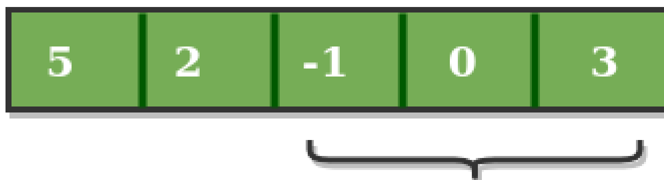
La suma inicial resulta en 6, que guardamos como "suma = 6".

Avanzamos:



La suma del intervalo cubierto por la ventana es 1, pero *nótese que para actualizar la suma sólo es necesario realizar dos operaciones, independientemente del valor de "k":* **partiendo de la suma inicial, actualizamos la suma de la ventana sumando el valor que "entra por la derecha" (0, en este caso), y restando el que "sale por la izquierda" (5, en este caso)**. Ya que partíamos de 6, 6 + 0 - 5 = 1.

Repitiendo en proceso una vez más:



Ya que el valor actual de "suma" es 1, actualizamos mediante 1 - 2 + 3 = 2.

```
// Solución en tiempo O(n) para encontrar la máxima suma de un subarray
// contiguo de longitud k.
#include <iostream>
using namespace std;

int maxSum(const vector<int>& v, int k) {
    // Asegurarnos de que es posible.
    if (v.size() < k) {
        cout << "Invalido";
        return -1;
    }
```

```
    // Cálculo de la suma de la ventana inicial.
    int max_sum = 0;
    for (int i = 0; i < k; i++) {
        max_sum += v[i];
    }

    // Cálculo de las sumas de ventanas sucesivas, avanzando "a lo gusano".
    int window_sum = max_sum;
    for (int i = k; i < v.size(); i++) {
        window_sum += v[i] - v[i - k];
        max_sum = max(max_sum, window_sum);
    }
    return max_sum;
}

int main() {
    const vector<int> v = { 1, 4, 2, 10, 2, 3, 1, 0, 20 };
    int k = 4;
    cout << maxSum(arr, k);
    return 0;
}
```

Programación Dinámica

Nota: el nombre de esta técnica es casi completamente arbitrario. "Programación dinámica" fue un nombre elegido por Richard Bellman, su precursor, como manera de ocultar, en 1953, el carácter matemático de su técnica a sus jefes militares, que eran abiertamente hostiles al concepto de "investigación matemática". El término "programación" no se refiere a la escritura de código, sino al significado original de planear o planificar, típicamente a base de completar una tabla. La fuerza aérea de EE.UU. financió a Bellman y a otros para que desarrollasen métodos para construir horarios de entrenamiento y logística, que llamaban "programas" (al igual que en una ópera, o concierto de música clásica, por ejemplo). El término "dinámico" es mayormente una palabra de "marketing", para que sonase bien dada la mentalidad de la época posterior a la segunda guerra mundial, de futurismo y de se-puede-hacer.

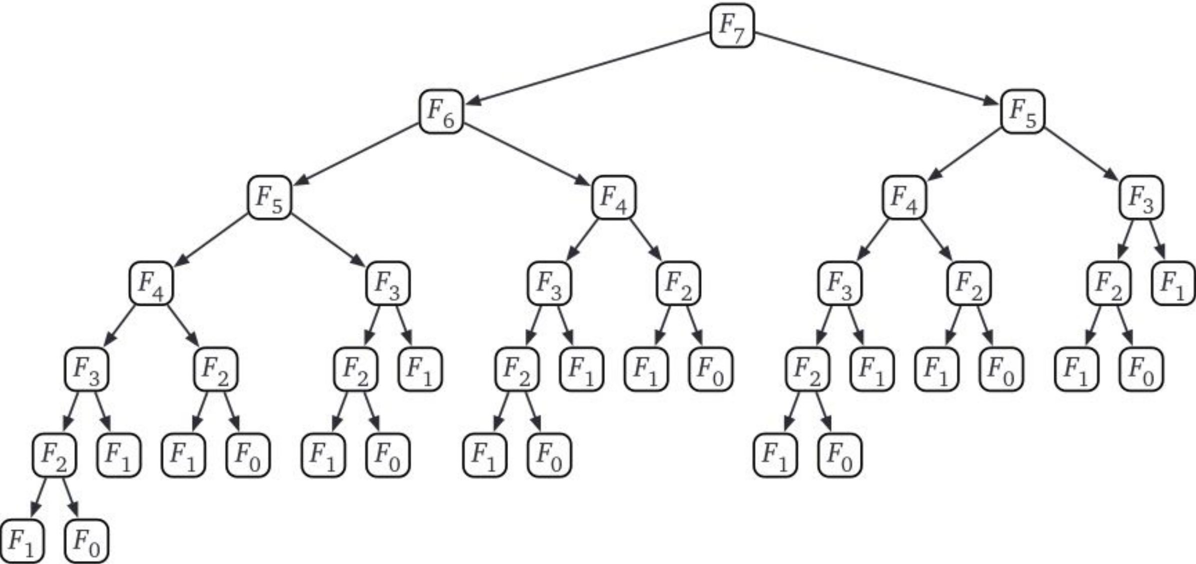
Volvamos a considerar la implementación recursiva de Fibonacci de la semana pasada:

```
RecFibo(n):
  if n = 0
      return 0
  else if n = 1
      return 1
  else
      return RecFibo(n − 1) + RecFibo(n − 2)
```

Desafortunadamente, esta implementación *naive* es terriblemente lenta. Excepto por las llamadas recursivas, el algoritmo requiere únicamente un número constante de pasos: una comparación y quizás una suma. Sea T(n) el número de llamadas recursivas a `RecFibo`; esta función satisface la recurrencia:

$$T(0) = 1, \quad T(1) = 1, \quad T(n) = T(n-1) + T(n-2) + 1,$$

La razón más clara de por qué esta implementación es lenta se debe al cálculo una y otra vez de los mismos (sub)problemas. Por ejemplo, para el cálculo del séptimo número de Fibonacci tendríamos el siguiente árbol de llamadas.



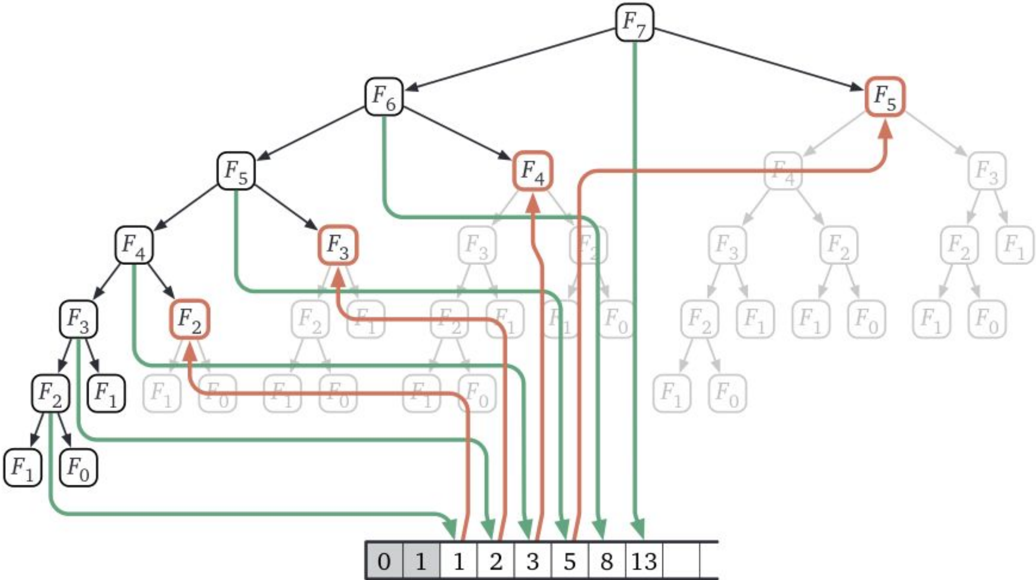
Una manera de no recalcular subproblemas es recordarlos:

```
MEMFIBO(n):
  if n = 0
      return 0
  else if n = 1
      return 1
  else
      if F[n] is undefined
          F[n] ← MEMFIBO(n−1) + MEMFIBO(n−2)
      return F[n]
```

Esta técnica se denomina *"memoization"* (sí, no es lo mismo que "memorization"), derivada de la idea de ir recordando resultados anteriores.

Antes de cada llamada, comprobamos si ya conocemos el resultado para el subproblema en cuestión, en cuyo caso lo devolvemos directamente. Si no, lo computamos —por primera vez— y lo guardamos para futura referencia.

Utilizando memoization el árbol de llamadas ahora sería:



Las flechas verdes hacia abajo indican escritura en el array utilizado como memoria de resultados anteriores. Las flechas rojas hacia arriba representan la lectura de un valor previamente "*memorizado"*.

Bottom-Up: completar la tabla explicitamente.

Una vez sabemos cómo el vector de resultados se va completando, podemos reemplazar la versión con memoization por un simple bucle for que lo va completando de manera explícita, en el orden adecuado, en vez de depender de un algoritmo recursivo más complejo:

```
ITERFIBO(n):
    F[0] ← 0
    F[1] ← 1
    for i ← 2 to n
        F[i] ← F[i−1] + F[i−2]
    return F[n]
```

Ahora es muy sencillo analizar el coste computacional: se usan O(n) sumas y O(n) escrituras a memoria.

Amnesia parcial: recordemos sólo lo necesario

En muchos algoritmos de programación dinámica, no es siquiera necesario recordar todos los resultados intermedios. Por ejemplo, podemos reducir la cantidad de memoria utilizada significativamente (de O(n) a O(1)) de la siguiente manera:

```
ITERFIBO2(n):
    prev ← 1
    curr ← 0
    for i ← 1 to n
        next ← curr + prev
        prev ← curr
        curr ← next
    return curr
```

Ahora sólo necesitamos recordar dos valores, "prev" y "curr" para poder calcular "next".

El problema de la suma de subconjuntos

El problema de la suma del subconjuntos pregunta si, dado un vector "X" de tamaño "n", existe algún subconjunto del mismo tal que su suma sea "t", un entero dado.

En este problema vamos a utilizar índices del 1 al n, en vez de los usuales 0..n-1. En algunos casos (por ejemplo en la estructura de datos llamada "montículo", que ya veremos) esto simplifica la notación.

Definamos la siguiente función booleana:

```
SS(i, t) = TRUE si y sólo si existe un subconjunto de X[i..n] que suma t.
```

Partiendo de esta definición, podemos plantearnos el problema de manera recursiva. Si consideramos el "sufijo" de X formado por los elementos a partir de la posición "i" (inclusive), tenemos dos opciones: considerar el elemento i-ésimo para la suma, o no considerarlo.

- Si no lo consideramos, simplemente acortamos el sufijo en una posición incrementando "i", pero como no lo hemos utilizado para la suma, "t" no cambia: SS(i + 1, t).
- Por otro lado, si incluimos el valor de X[i] en la suma hacia "t", igualmente incrementamos "i", progresando en el vector, pero también descontamos el X[i] añadido a "t" de lo que buscaremos en el subproblema para X[i+1..n]: SS(i+1, t - X[i]).
    - Nótese que si "t < X[i]", la operación t - X[i] dará un número negativo, y por tanto no existiría solución válida. Ya que X OR FALSE == X, podemos simplificar este caso a simplemente no considerar utilizar X[i].

Devolveremos TRUE si cualquiera de estos dos subproblemas devuelve TRUE, por lo que los combinamos con un OR (también representado en lógica como ∨).

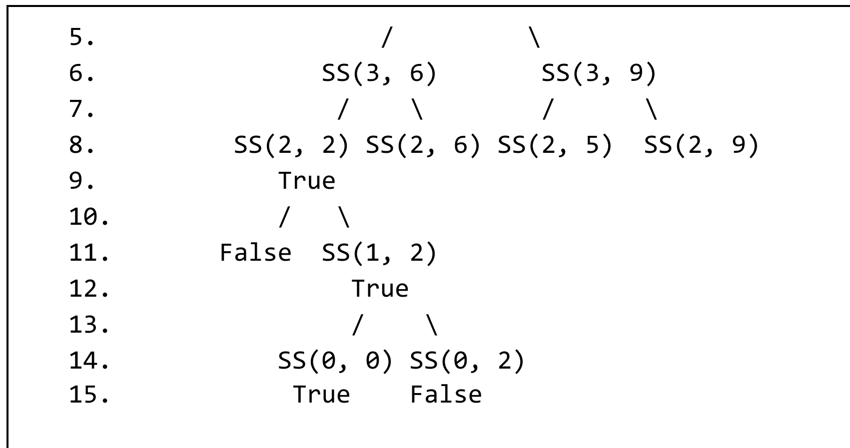
Casos base:

- Sabemos que si se nos pide sumar 0, podemos retornar TRUE, ya que siempre podemos "sumar hasta cero", incluso si el vector de entrada X está vacío.
- Si "t" es mayor que cero pero ya hemos procesado todo el vector X (i > n), no nos es posible satisfacer esas "t" unidades a sumar que nos falta, por lo tanto devolveremos FALSE.

$$SS(i,t) = \begin{cases} \text{TRUE} & \text{if } t = 0 \\ \text{FALSE} & \text{if } i > n \\ SS(i+1,t) & \text{if } t < X[i] \\ SS(i+1,t) \lor SS(i+1,t-X[i]) & \text{otherwise} \end{cases}$$

Por ejemplo:

```
1. X[]={3, 4, 5, 2}, t = 9
2.
3.                     SS(4, 9)
4.                      True
```



```
5.                    /         \
6.                SS(3, 6)       SS(3, 9)
7.                 /    \        /      \
8.          SS(2, 2) SS(2, 6) SS(2, 5)  SS(2, 9)
9.             True
10.            /   \
11.        False  SS(1, 2)
12.                 True
13.                 /    \
14.            SS(0, 0) SS(0, 2)
15.             True    False
```

Aún no hemos utilizado programación dinámica. Al igual que en el caso de Fibonacci, esta formulación recursiva resultaría en una repetición de cálculos. Para evitarlo, recurriremos a la construcción de una tabla con los valores que el problema va necesitando a medida que crece:

- Estructura de datos: Una tabla (array/vector bidimensional) con n+1 filas y T+1 columnas: S[1..n+1, 0..T], donde S[i, t] representa el valor de la función S(i, t).
- Orden de evaluación: cada posición S[i, t] depende de como mucho otras dos posiciones, ambas del tipo S[i+1, ...]. Por tanto, podemos ir completando la tabla S por filas, empezando por las filas de mayor índice. Para las columnas podemos proceder en cualquier orden, ya que la dependencia es únicamente en la final anterior, cuyas columnas estarían ya todas completadas.
- Coste computacional y de espacio: es inmediato ver que la tabla S utiliza O(nT) espacio. Se puede computar S[i, t] a partir de S[i+1, t] y S[i+1,t−X[i]] en tiempo constante por lo que el algoritmo se ejecuta en tiempo O(nT).

```
FastSubsetSum(X[1..n], T):
    S[n+1, 0] ← True
    for t ← 1 to T
        S[n+1, t] ← False

    for i ← n downto 1
        S[i, 0] = True
        for t ← 1 to X[i] − 1
            S[i, t] ← S[i+1, t]          ⟨⟨Avoid the case t < 0⟩⟩
        for t ← X[i] to T
            S[i, t] ← S[i+1, t] ∨ S[i+1, t − X[i]]

    return S[1, T]
```

Ejemplo para `X[] = {3, 4, 5, 2}, t = 6`

i \ t	0	1	2	3	4	5	6
X[1] = 3	T	F	T	T	T	T	T
X[2] = 4	T	F	T	F	T	T	T
X[3] = 5	T	F	T	F (1)	F	T (2)	F
X[4] = 2	T	F	T	F	F	F	F
X[5]	T	F	F	F	F	F	F

(1) Es FALSE ya que SS(3, 3) = SS(4, 3), que es FALSE. En este caso, t = 3 < X[i] = 5.
(2) Es TRUE ya que SS(3, 5) = SS(4, 5) OR SS(4, 5 - 5 = 0) = FALSE OR TRUE = TRUE.

Por último, el resultado del problema inicial, SS(1, 6), se puede leer de la tabla, S[1][6], siendo TRUE.