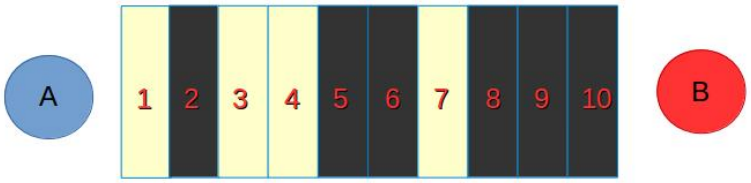
Problema a resolver:

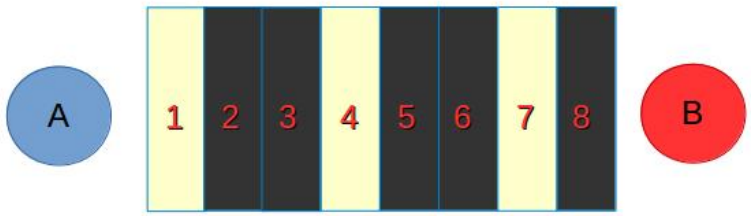
El problema esta dado por la siguiente situación: nos encontramos en un extremo de un puente (afuera de él) y queremos llegar al otro extremo, bajo ciertas circunstancias y, si es posible hacer esto, hacerlo de la "mejor manera" (mas adelante se detallará qué significa esto y por qué podría no ser posible atravesar el puente). El puente está hecho con una cantidad n de tablones seguidos y, algunos de ellos pueden estar rotos. Cuando esto suceda, vamos a querer saltar estos tablones rotos al atravesar el puente, pisando siempre tablones sanos cuando estemos avanzando.
Tenemos una cantidad fija tablones seguidos que podemos saltar, la denominaremos c, así podemos ver que, si el puente tiene, en algún momento, una cantidad $k > c$ de tablones rotos **seguidos**, entonces claramente no tendremos manera de atravesarlo ya que, intuitivamente, podemos pensar que, en el mejor de los casos (donde mejor significa estar lo más alejados posibles del comienzo del puente) podríamos estar en el tablon sano anterior (anterior inmediato) al primero de esos k tablones rotos y, aún así no podríamos atravesar el puente ya que no podemos saltar más de c tablones seguidos, por lo tanto en cualquier otro caso (donde nos encontráramos en un tablón anterior al anterior inmediato del primero de los k rotos por ejemplo), estaríamos en una situación similar porque eventualmente llegaríamos al tablón sano que es el anterior inmediato al primero de los k y quedaríamos estancados de la misma manera.
En el caso en el que no se de la situación descripta anteriormente (es decir, en el caso en el cual sí podamos atravesar el puente), vamos a querer dar la menor cantidad de saltos (la "mejor manera").

Presentamos algunos ejemplos graficos junto a sus soluciones y las secuencias que lo representan. Los circulos con el A y el B determinan el punto de partida y el punto de llegada, ambos fuera del puente. Los tablones rotos están pintados de negro.



En este ejemplo, $n = 10$ y el puente se escribe como 10 c 0 1 0 0 1 1 0 1 1 1.
Si $c = 3$, entonces la solución está dada por caer en los tablones 4 7 11. Notar que en realidad no existe un trablón numerado con el 11, pero cuando exista una solución, para indicar que llegamos al punto de llegada, diremos que saltamos a un tablón mayor estricto que la cantidad de tablones del puente (o

sea, que estamos efectivamente fuera del puente). Si tuvieramos el mismo puente pero con $n = 2$, claramente no existiría una solución ya que, si bien podríamos llegar al tablón 7 sin problemas, una vez ahí no tendríamos manera de saltar el 8, 9 y 10 que están rotos.



Este otro puente se representa como 8 c 0 1 1 0 1 1 0 1.
Si $c = 2$ entonces la solución es 1 4 7 9.
Si $c = 1$ entonces no habría solución ya que hay 2 tablones rotos seguidos (esto ocurre dos veces en este caso particular pero con que ocurra una ya no hay solución posible).
Si $c = 3$ la solución es 4 7 9.

Finalmente introducimos este ultimo ejemplo del puente 6 c 0 1 1 1 1 1.
Siguiendo la misma lógica que veníamos teniendo, podemos ver que este puente no tiene solución para $c < 5$. Si $c = 5$ entonces la solución es 1 7 y sí $c > 5$ la solución es 7.

Resolución:

La idea basicamente es ir recorriendo el puente (tablón por tablón) e ir almacenanando cual es el último tablón sano (lo llamaremos *ultimosano*) desde el comienzo del puente hasta el tablón de la iteración actual i (notar que *ultimosano* podría ser i), los tablones que conformarían la solución en el caso de que exista y la cantidad de tablones por los que pasamos desde el último tablón que agregué a la solución (sin contarlo) hasta el tablón de la iteración actual (contándolo), llamaremos a esta última variable *saltados*.
Eventualmente podemos llegar a una iteración donde $saltados = c + 1$ (c es dato y representa la cantidad de tablones como máximo que puedo saltar) y, si no llegamos a esta iteración quiere decir que terminamos de recorrer todos los tablones del puente antes de llegar a pasar por $c + 1$ tablones desde el último sano.
Analicemos el primer caso donde efectivamente llegamos a esa iteración donde vale $saltados = c + 1$: si llegamos a este punto significa que estamos parados en el tablón mas lejano al cual puedo llegar saltando desde el último tablón que agregué a la solución (porque si puedo saltar c tablones seguidos desde donde estoy, entonces caigo en el tablón $c + 1$). Entonces nuevamente pueden pasar dos cosas: 1) $i - ultimosano > c$ (que la cantidad de tablones entre el tablón de la iteración actual i hasta *ultimosano* sea mas grande que c) 2) $i - ultimosano <= c$. Si estamos en 1) voy a tener una cantidad mayor estricta que c de tablones seguidos donde ninguno de ellos es un tablón sano (la última vez que asigné un valor a *ultimosano* fue o bien antes de empezar a iterar, o sea cuando estoy fuera del puente, o en un tablón que dista a más de c tablones del actual), esto quiere decir que están todos rotos y, como no puedo saltar una cantidad mayor que c de tablones seguidos, concluimos que no existe una solución al problema. Ahora bien si estamos en 2), entonces quiere decir que existe un tablón sano (y está almacenado en *ultimosano* entre el último que agregué a la solución (si no agregué ninguno, entonces este tablón representa el punto de partida fuera del puente) y el tablón representado por la iteración actual i y este tablón NO es el último que agregué a la solución (de ser así no estaríamos en este caso). Entonces lo que hacemos es agregar *ultimosano* a la solución y actualizar el valor de *saltados* para que represente la cantidad de tablones que salté desde el último que agregué a la solución (que es el que agregamos recién, *ultimosano*) hasta llegar al tablón de la iteración actual i. Una vez hecho esto, repetimos el proceso descripto con la siguiente iteración.
Dado que los tablones que deben ser solución se agregan a la misma cuando nos encontramos en una iteración i en la cual vale $saltados = c + 1$, podría suceder que terminemos de iterar y no hayamos pasado por esta iteración i. Esto quiere decir que a partir del último tablón que agregué a la solución, hay una cantidad menor estricta que $c + 1$ tablones y por lo tanto, puedo saltarlos todos, alcanzando así el punto de llegada fuera del puente.
Cuando terminamos de iterar todos los tablones del puente, agregamos como el último "tablón" de la solución, un número mayor estricto que n para indicar

que estamos fuera del puente.

A continuación, presentamos el pseudocódigo que hace lo que describimos arriba:

```
1: procedure ResolverPuente(puente, c)
2:     ultimosano ← −1
3:     saltados ← 0
4:     output ← ∅
5:     n ← cantidadTablones(puente)
6:     for i ← 0, n − 1 do
7:         saltados + +
8:         if puente[i] ← 0 then
9:             ultimosano ← i
10:        end if
11:        if saltados = c + 1 then
12:            if i − ultimosano > c ∨ ultimosano == −1 then
13:                no hay solución, termino
14:            end if
15:            agregar(output, i + 1)
16:            saltados = i − ultimosalto
17:        end if
18:    end for
19:    agregar(output, size(puente) + 1)
20: end procedure
```

Demostración de correctitud:

Para realizar la demostración de correctitud del algoritmo, vamos a separarla en dos casos: cuando el problema no tiene solución (caso I) y cuando sí tiene (caso II).

Caso I) Queremos probar que: 1)si el algoritmo informa que el problema no tiene solución, entonces efectivamente el problema no tiene solución y 2)no puede pasar que el problema no tenga solución y el algoritmo no lo informe, devolviendo así, una solución incorrecta.
1) Dado que nuestro algoritmo informa que el problema no tiene solución cuando vale A: $(saltados = c + 1) \wedge (i - ultimosano > c \ \vee \ ultimosano == -1)$ y $A \Rightarrow (i - ultimosano > c \ \vee \ ultimosano == -1) = B$, como en cada iteración sabemos el índice del último tablón sano porque vamos pisando la variable *ultimosano* (líneas 8 a 10 del pseudocódigo), B nos está diciendo que entre el tablón de la iteración actual y el último tablón sano que miré, hay una cantidad **estrictamente mayor a** c de tablones, por lo tanto nunca voy a poder saltarlos, ya que si estoy parado en el tablón *ultimosano*, vemos que hay mas

de c tablones para saltar y no puedo y, la otra opción es que esté parado en un tablón anterior a *ultimosano*, y eventualmente llegaría, como muy lejos, al tablón *ultimosano* sin caerme. Nunca voy a poder estar en un tablón posterior a *ultimosano*, por lo tanto concluimos que el problema no tiene solución.
2)Para que el algoritmo no tenga solución, tiene que pasar que haya una cantidad $k > c$ de tablones rotos seguidos ya que si esto no pasa, entonces siempre puedo avanzar y caer en un tablón sano más adelante del que estoy parado y, eventualmente voy a poder saltar al punto de llegada fuera del puente. Llamemos j al primer tablón de esos k tablones rotos seguidos. $j-1$ entonces representa el último tablón sano antes del primero de esos k tablones rotos seguidos, o también puede valer -1 en el caso de que esos k tablones seguidos sean los primeros k tablones del puente.
Debido a que estamos recorriendo tablón por tablón, sabemos que en algún momento vamos a caer en el tablón $j + c$ (si no llegamos a este tablón es porque nos dimos cuenta antes que no había solución porque, antes de este tablón sano $j - 1$, existían $k' > c$ tablones rotos seguidos) el cual existe ya que a partir de j hay $k > c$ tablones seguidos y, en esa iteración va a valer A ya que se habrá incrementado *saltados* en uno desde j hasta la iteración actual i, y dado que $i = j + c$ y $ultimosano = j - 1$, $\Rightarrow i - ultimosano \equiv j + c - (j - 1) = c + 1 > c$ y como vale A entramos en el if de la línea 12, indicando así que no hay solución.

Caso II) Como ya demostramos que el algoritmo funciona correctamente cuando el problema no tiene solución y ahora queremos ver que también funcione correctamente cuando sí la tiene, vamos a asumir de ahora en más que *existe* una solución al problema.
Vimos que una solución al problema no es más que un conjunto (de cardinal mínimo) de tablones sanos del puente, en el cual si sacamos un tablón y el tablón próximo a él (el que le sigue en índice), la cantidad de tablones entre ellos es menor o igual a c y, además, este conjunto contiene al punto de llegada fuera del puente, indentificado como un tablón más de índice mayor a la cantidad de tablones del puente. Entonces, sería similar si pensáramos a la solución como ese mismo conjunto descripto anteriormente, menos el tablón que representa el punto de llegada y, que ademas cumple que la cantidad de tablones desde el tablón con mayor índice (sin contarlo) hasta el último tablón del puente (contándolo), es menor o igual a c. Y ese será nuestro concepto de solución para hacer la siguiente demostración (pasar de ese conjunto a la salida que pide el problema que tenga el algoritmo es trivial, ya que basta solo con agregarle un índice mayor a n al conjunto).
Vamos a querer probar que, al finalizar el for, *output* va a ser un conjunto que cumpla con lo que dijimos recién (*).

Decimos que un conjunto S es subsolución de una solución S', cuando $S \subseteq S'$.
Para poder probar (*), vamos a demostrar que I)*output* comienza siendo subsolución de una solución S y II)al comenzar y finalizar cada iteración, *output*

sigue siendo subsolución de alguna solución (no necesariamente siempre la misma)

I)Vale trivialmente, ya que asumimos que el problema tiene solución y, como *output* comienza siendo el cojunto vacío y sabemos que vale $(\emptyset \subseteq C)(\forall\ C)$, entonces particularmente vale para $C = S$

II)Si al finalizar la iteración i, *output* no cambió con respecto a su valor antes de empezar dicha iteración, entonces claramente sigue valiendo que *output* es subsolución de una solución S, ya que *output* empieza siéndolo.
Ahora bien, si *output* sí cambió de valor, significa que se agregó el tablón *ultimosano* (no puede cambiar de otra manera) y pueden ocurrir dos cosas: que $ultimosano \in S$ (siendo S la solución que tenía a *output* como subsolución antes de que cambie su valor) y en tal caso *output* seguiría siendo subsolución de S porque antes lo era y ahora agregamos un elemento que pertenece a S, o que $ultimosano \notin S$. Analicemos este último caso: llamemos t al último tablón que agregué a *output* (antes de que cambie su valor en esta iteración) y r al tablón próximo a t de S. Dado que $r \neq ultimosano$, entonces necesariamente $r < ultimosano$, ya que *ultimosano* es el tablón sano más lejano a t, entonces podemos tomar $S' = (S \setminus r) \cup \{ultimosano\}$ y S' efectivamente es una solución ya que desde t puedo saltar a *ultimosano* (*ultimosano* se encuentra, a lo sumo a c tablones mas adelante de t) y, desde *ultimosano* puedo saltar a los mismos tablones que podría saltar desde r y más también. Concluimos entonces que, como $output \subseteq S'$ y S' es solución, *output* sigue siendo subsolución de una solución.

Cuando finalice la última iteración del ciclo, el último tablón que habré agregado (llamémoslo m) estará a lo sumo a c tablones del final del puente ya que si no, hubieramos llegado al tablón $m + c + 1$ e informaríamos que no hay solución ya que la distancia a *ultimosano* es mayor que c y esto no puede pasar ya que asumimos que exixte una solución), y probamos que *output* es subsolución de una solución al iniciar y finalizar cada iteración, por lo tanto también lo es al terminar el ciclo. Como no necesitamos agregar otro tablón más porque ya podemos saltar desde el último tablón agregado al punto de llegada, entonces efectivamente *ultimosano* es una solución.

Complejidad temporal:

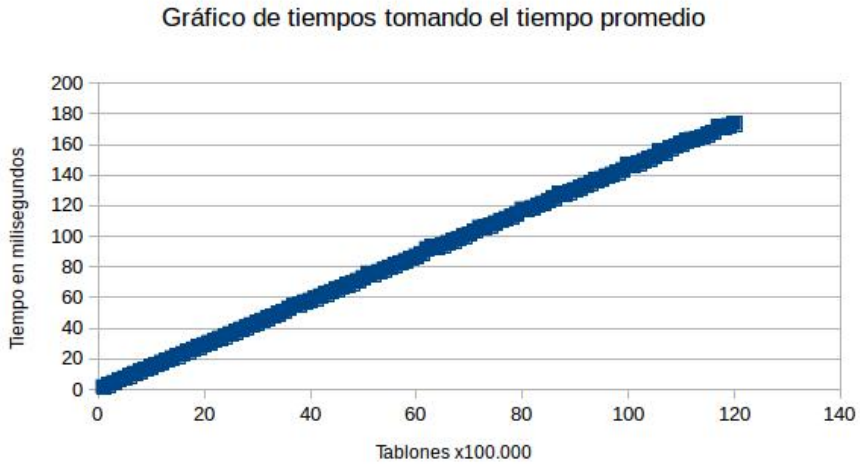
La complejidad del algoritmo es claramente lineal a la cantidad de tablones, ya que todas las operaciones que se hacen dentro del ciclo son de orden constante (en el código fuente almacenamos la solución en el tipo std::list, el cual tiene orden constante en la inserción según http://en.cppreference.com/w/cpp/container/list y para almacenar el puente utilizamos std::vector el cual tiene orden constante en el acceso a una posición según http://en.cppreference.com/w/cpp/container/vector).

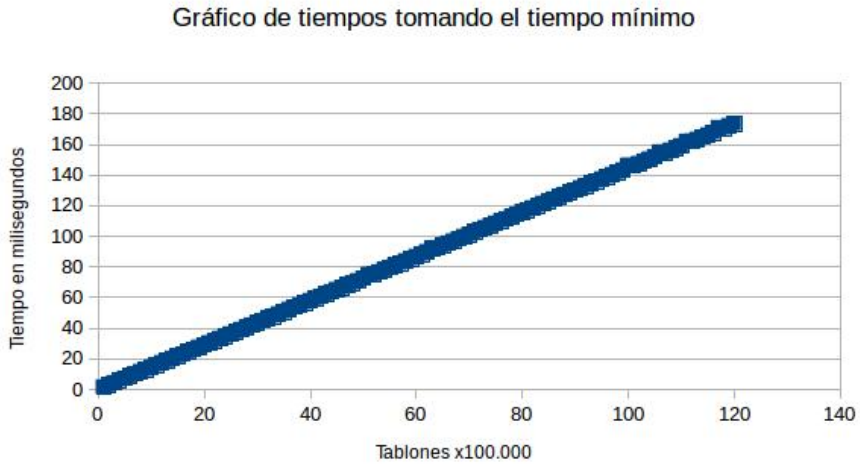
Todas las demás operaciones realizadas son if's y asignaciones/sumas de enteros, que también tienen órdenes de complejidad constantes.
El peor caso del algoritmo entonces, es cuando tiene solución, ya que si pasa eso entonces debe recorrer necesariamente todos los tablones (no sale antes del ciclo) y es $\theta(n)$.
El caso donde el algoritmo tarda menos de $n.m$ (m una constante) operaciones es claramente cuando el problema no tiene solución. Es decir, dado que el algoritmo sale del ciclo y termina cuando se da cuenta que el problema no tiene solución, podríamos pensar en un mejor caso cuando existen $k > c$ tablones rotos seguidos. En particular, si esos k tablones rotos seguidos, son los primeros k tablones del puente, la complejidad del algoritmo es $\theta(c+1) = \theta(c)$ para $c < n$ (si esto ultimo no fuera así, entonces si habría solución).

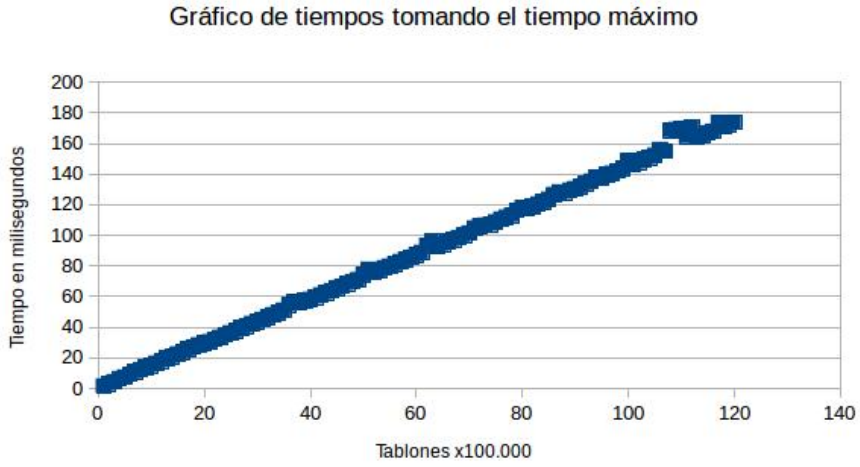
Mediciones de tiempo:

Para medir la performance del algoritmo se han generado 120 instancias de manera aleatoria. La instancia i tiene $n = i * 100.000$ tablones (o sea van de un 100.000 a 12.000.000 de tablones) y el salto máximo c, también elegido de manera aleatoria, está en el rango [0, 999.999].
Cada tablón tiene igual probabilidad de estar roto que de estar sano, pero todas las instancias generadas tienen solución (para poder medir siempre el peor caso).
Cada una de las instancias fue resuelta (o no) por el algoritmo 10 veces y se registraron los tiempos de cada ejecución, tomando el mínimo, máximo y promedio de tiempo de cada una de las 10 ejecuciones de cada instancia.

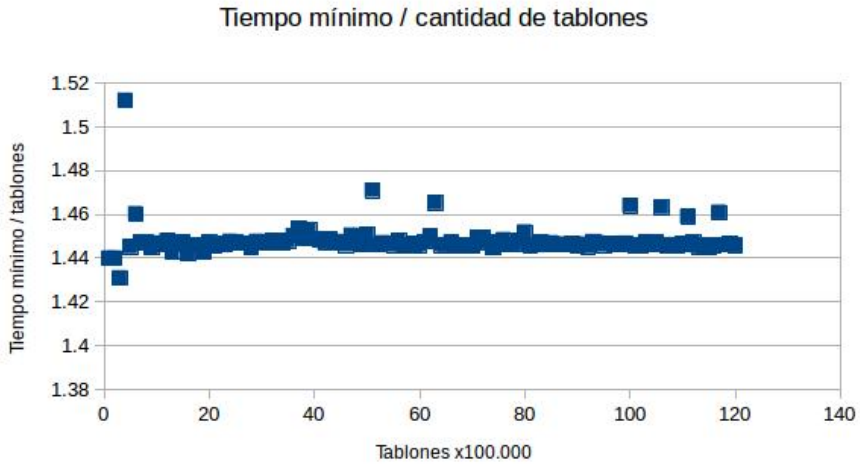
Presentamos los gráficos de tiempo:







Tomando el tiempo mínimo como el tiempo "más cercano" a lo que tarda el algoritmo independientemente de los demás procesos que están corriendo paralelamente y consumiendo tiempo, si dividimos ese tiempo por la cantidad de tablones de cada instancia, nos queda el siguiente gráfico:



en el cual podemos observar que, claramente la función tiende a una constante. Dicha constante es la que "despreciamos" a la hora de decir que en el peor caso es lineal a la cantidad de tablones.
Como los tiempos aproximan a una función lineal y, si dividimos por el n al tiempo mínimo nos queda una función que aproxima a una constante, podemos concluir que hemos demostrado empíricamente que el algoritmo tiene un peor caso $\theta(n)$.