Problemas de Diseño y Análisis de Algoritmos

Problema 3: Proyectos Evaluativos

Equipo: 3

Miembros:
- Laura Brito Guerrero
- Sheyla Cruz Castro
- Rodrigo García Gómez

Enunciado del Problema:

Se termina el semestre y algunos profesores ya han orientado proyectos opcionales para que los alumnos mejoren sus notas. Uno de los estudiantes del año, se decide a realizar algunos de estos proyectos y está organizando su horario particular. Cada proyecto tiene un día de entrega especificado y la cantidad de días que se necesitan para completarlo. El estudiante puede trabajar en cada proyecto siguiendo cualquier orden, pero en cada día solo puede trabajar en uno de ellos. El día de entregar un proyecto no se puede hacer otra cosa que la entrega en cuestión. Se sabe además que no hay dos proyectos que deban ser entregados el mismo día. La cuestión se complica, porque existen ciertos eventos familiares a los que el estudiante no puede faltar. Para cada uno de estos eventos se conoce el día en que se celebra y se asegura que nunca tendrán lugar dos de ellos el mismo día. En esos días especiales de evento, el estudiante no puede trabajar porque debe celebrar con su familia. Ninguna fecha de entrega coincide con los días de estos eventos. El objetivo del problema consiste en seleccionar el mayor conjunto de proyectos, que el estudiante pueda realizar completamente, sin perder ninguno de los eventos familiares.

Abstract

Se atacó inicialmente por una idea *fuerza bruta* exponencial, la cual funciona ya que se prueban todas las posibles combinaciones. Luego se tienen varias ideas de solución *greedys* y una dinámica, pero estas no son correctas. Finalmente se alcanzó una solución *greedy* satisfactoria en n^2 y se logró bajarle el costo a $n \log n$.

Idea Fuerza Bruta:

(brute)

Para resolver este problema primeramente se tiene un algoritmo $fuerza\ bruta$ que genera todas las posibles combinaciones, para así tomar el mayor subconjunto. Sea la entrada una lista de tuplas, donde en la posición i se encuentra el proyecto i-ésimo, y una lista con los días de celebraciones con las familias. Sin falta de formalismos se refiere a los proyectos en el array de tuplas x, donde el primer ítem representa el día de entrega y el segundo ítem la cantidad de días q se demora realizarlo, y el array de familia se denotará por f. Una vez dado x, se ordenará en orden logarítmico por el primer ítem.

El problema se razonará de la siguiente manera: Dado una lista de tuplas x (con las características antes mencionadas), y una lista de números f, hallar el mayor subconjunto de los valores del primer ítem de x tal que se puedan eliminar la cantidad de sus valores correspondientes en el segundo ítem tal que no intercepten entre ellos, es decir, que aun así, puedan quedar posiciones vacías delante de los valores seleccionados por el primer ítem(puede darse el caso en que todas se tomen, lo que sí es incorrecto que suceda es que se tomen más posiciones de las que se encuentren disponibles), teniendo en cuenta en cada momento que las posiciones que informa f siempre están ocupadas. Para trabajar se crea un array booleano del tamaño del primer ítem del último elemento de $x+1$. Como las posiciones que brinda f siempre están ocupadas, entonces se marcan como nulas ($true$). Luego generamos el conjunto potencia utilizando combinaciones sobre x, para luego cada vez que se arme un subconjunto analizar si es válido. La validez es que existan siempre posiciones no marcadas para poder satisfacer la cantidad de posiciones (se refiere al segundo elemento de las tuplas de x) del subconjunto dado. Este procedimiento se realiza analizando en el array que se creó auxiliar e ir marcando las posiciones que no han sido descubiertas, para que así no ocurra intersección entre ellos, ya que según la orden no puede pasar. Si se lograron tachar todas las requeridas por el subconjunto correspondiente, entonces es válida la selección. Se debe dejar claro que en este caso sí importa como marquemos las posiciones, ya que no se deben de tomar aleatoriamente. Si se analiza un poco es evidente que las mismas se analizan en orden creciente, es decir, el puntero señala la i-ésima posición, cuando ya se analizaron las $i-1$ anteriores, y a partir de ahí poder tachar las que no se han tomado. Ya que puede pasar que se tomen celdas luego de la posición límite, y eso no tiene sentido. Aclarando la cuestión anterior se tiene un algoritmo que resuelve todas las posibles combinaciones de los elementos de x que analiza si satisface la condición de unión de elementos entre ellos, ya que no puede existir una intersección dada por la lógica que desempeña el significado de los elementos de x.

Como esta plantilla de solución genera todas las subcadenas y va guardando la más óptima en cada momento, entonces su orden computacional es de $2^n * m * n$, con n: canti-

dad de elementos de x y m: mayor elemento de x con respecto al primer ítem. La idea es bajar el orden a este algoritmo ya que a pesar que nos brinda la respuesta, es muy ineficiente y para valores no tan pequeños puede demorarse considerablemente en ejecutarse.

Idea 2:

Una nueva visión de este problema es enfrentarlo con un algoritmo *greedy*. La idea es ordenar de manera creciente los costos de x, es decir ordenarlos por $item2$(cantidad de días que toma hacer el proyecto), de manera tal que la mejor forma de elegir los $item1$(fecha de entrega) de x sea los que menor costo tenga. Y así se garantizaría el óptimo en cada paso. Se tendrían como ayuda un arreglo *ocupado*, el cual recoge en la posición i la cantidad de unidades ocupadas por el $x[i].item1$ antes de agotar las unidades necesarias por $x[i].item2$, ya sea por las posiciones que ocupa f o porque durante el transcurso del problema se ocuparon posiciones que se encuentran antes que $x[i].item1$.

El objetivo es recorrer el arreglo ordenado por el criterio anteriormente explicado, y así verificamos si se puede colocar por las unidades que se encuentran disponibles, si es posible entonces con la ayuda de un arreglo booleano vamos marcando las posiciones que ya tomé, para así garantizar que no se intercepten entre sí. El criterio de prioridad sería: recorro en sentido inverso el arreglo comenzando por la posición anterior al $x[i].item1$ que estoy analizando y se van marcando aquellas posiciones que estén libres lo más cercano a el posible, las posiciones que coincidan con algún $x[k].item1$,o sea a alguna fecha de entrega, no se toman, ya que se desconocen si luego esas posiciones son necesarias marcarlas. En el caso que no se puedan marcar todas las necesarias, entonces se recurren a las posiciones de fechas de entregas de algun proyecto (algun x[k].item1) previas a ti, las cuales se necesitan tachar. Como a la vez que se verifique si es posible rellenar las posiciones de un $x[i]$ entonces siempre se puede tachar, ya que por el criterio de ordenación de los x se garantiza en cada paso el óptimo. En ese caso se tacharán las posiciones de x las cuales se encuentren antes que $x[i].item1$ (valor modular) pero las más alejadas posible del arreglo ordenado, para en un futuro garantizar la menor cantidad de anulaciones en posibles posiciones de x.

Este algoritmo funciona, pero no siempre da el más óptimo, es decir no responde al mayor subconjunto posible. El contraejemplo sería:
x $->$ (3,2), (5, 3), (7, 4), (8,1)
f $->$ 4, 6

La respuesta del algoritmo *greedy* planteado fuera 8, el subconjunto representado por la posición 8, mientras que existe una solución mejor, la cual es $8,3$, puesto que la posición 3 ocupa las dos posiciones libres antes que él y el proyecto 8 ocupa la posición 7, la cual no puede satisfacer sus posiciones, así que se asume como posición inválida desde un comienzo. Este tipo de casos, este algoritmo no lo recoge, puesto que siempre se asume que es mejor tomar posiciones libres y no despreciar una posición que es válida (es decir un $x[i].item1$) desde el principio, y esta lógica no es verdadera, de ahí el contraejemplo anterior.

Idea 3:

(binary-search)

Una tercera idea también $fuerza\ bruta$, pero mucho más eficiente que la $fuerza\ bruta$ inicialmente planteada, pues reduce considerablemente el número de combinaciones generadas, es la siguiente:

Para lograr el objetivo de optener la mayor cantidad de proyectos que es posible realizar se define la función :

$$f(x) = \begin{cases} 1 & si \quad posible \\ 0 & si \quad imposible \end{cases} \tag{1}$$

Donde *posible* significa que se pueden hacer x cantidad de proyectos e *imposible* que no se pueden hacer dicha cantidad de proyectos.

Se puede ver que esta función es un predicado que aplicado sobre el intervalo $[1,n]$, siendo n la cantidad total de proyectos para una entrada del problema, se obtiene una secuencia de resultados $[1,1,1,1,....1,0,0,0,....,0]$. Se aprecia además que la secuencia de resultados es bimodal. Analizando el anterior razonamiento se tiene que demostrar que $f(x) = 0 => f(x+1) = 0$ o sea una vez que no se puedan realizar x cantidad de proyectos no se pueden realizar más de dicha cantidad.

Se demostrará por Contrarrecíproco:

Tenemos $f(x+1) = 1 => f(x) = 1$, en efecto si podemos resolver $x+1$ proyectos, basta quitar uno de los proyectos asignado para alcanzar la cantidad x de proyectos deseada. Esto es posible pues no existe intersección entre la cantidad de días asignados a cada proyecto, luego quitar uno de estos con sus respectivos días no perjudica a los demás.

Luego por contrarrecíproco se cumple que $f(x) = 0 => f(x+1) = 0$.

Teniéndose este resultado se puede realizar una búsqueda binaria para obtener el mayor número de proyectos realizables, que sería equivalente a encontrar el primer *si*.

Aunque para encontrar el máximo número de proyectos realizables se hace en $\log n$ pasos, el costo de computar $f(x)$ es bastante elevado, dado que se debe probar en cada caso con los subconjuntos de tamaño k de los n proyectos y verificar en orden lineal que sea posible. Si se obtiene un subconjunto válido no es necesario revisar los demás subconjuntos de tamaño k, dado que solo es necesario que exista una forma posible, en caso que no sea posible, será necesario revisar todos los subconjuntos de tamaño k lo cual es ineficiente, en el peor de los casos sería 2^n.

Idea 4:

Un cuarto enfoque sería otra solución *greedy*, en la cual ordenamos los proyectos donde se prioricen los que menos tiempo requieren y más alejados se encuentran, pues estos son los que de realizarse más días libres dejan. Escoger los proyectos en este orden y realizarlos siempre que se puedan, o sea si se tienen los días necesarios disponibles hasta el día de la entrega, es siempre lo mejor, pues de decidir no hacer un proyecto que se encuentra antes siguiendo este criterio y hacer otro en su lugar que requiere más días y/o termina después , igualaría la cantidad de proyectos realizados(pues se sustituye uno por otro) y disminuiría la cantidad de días disponibles para el resto de los proyectos que tienen fecha de entrega después, y podrían hacer uso de estos, perjudicaría a la larga a los proyectos posteriores.

Para esto ordenamos los proyectos crecientemente por la razón entre la cantidad de días que requiere hacer el proyecto y la fecha de entrega. Luego recorremos ese array ordenado intentando en todo momento realizar el proyecto en cuestión. Para decidir si se puede hacer el proyecto o no, vemos si la cantidad de días disponibles me lo permite, para esto llevamos una variable *remain* que me permite conocer la cantidad total de días inutilizados, si los días que requiere el proyecto es menor que *remain* entonces existe la posibilidad, luego verificamos los días antes de la fecha de entrega, si también son suficientes se puede realizar el proyecto, para analizar los días disponibles previos a la fecha de entrega llevamos un array donde almacenamos dicha información, que actualizamos sobre la marcha, al igual que *remain*, para esto cada vez que se decide hacer un proyecto se disminuye *remain* en la cantidad de días que requiere este, y se actualiza la disponibil-

idad particular de cada proyecto, si este se entrega después del mismo siempre va a ser afectado, luego se le substraen dichos días, en cambio si se encuentra antes no siempre lo afecta, esto lo hace solo en caso de que la diferencia de días entre las fechas de entrega del proyecto que estoy realizando y este sea menor que los días que necesita este para realizarse, de esto pasar y no ser suficiente, se le quitan solo los días que le faltan al proyecto para poderse realizar. Esto se hace por cada proyecto escogido en ese orden mientras que *remain* sea mayor que 0.

Esta idea no funciona, para la mayoría de los casos es de imaginarse que deberían priorizarse los proyectos que menor cantidad de días requieran y antes cierren, pues ofrecen una mayor disponibilidad para el resto de los proyectos, pero sin embargo, aunque para una cantidad ínfima de casos, esta no es la mejor decisión a tomar.

Idea 5:

Otra idea para enfrentar este problema es teniendo una visión dinámica al respecto. Se tendrá un arreglo dp de dos estados i, j donde se guardará en $dp[i][j]$ cual es la cantidad máxima para realizar i proyectos en j días de trabajo. Como dato inicial se tiene el arreglo de tuplas que representa a los proyectos, es decir, la entrada va a ser invariante a las soluciones anteriores.

Inicialmente se llenará $dp[i][1] = 0$ para toda i > 1, ya que puede existir algún proyecto que necesite un día. Pero esta relación no va a ser verdadera para valores mayores que la unidad, ya que se necesita al menos un día por proyecto. De la misma manera se va llenando dp[1][i] para toda i > 1, ya que esta relación está en dependencia de los días que se le asigne a cada uno.

Luego de llenar estos datos, se puede comenzar a realizar la programación dinámica. Se realiza un doble ciclo para abarcar todas las posibles opciones. Pero no se van a ir insertando los proyectos en cualquier orden, sino que se ordenarán por la cantidad de días que necesita cada uno. El problema se encuentra en garantizar que los proyectos seleccionados no intercepten, ya que los días se pueden repartir en cualquier orden y no se tiene el control de que día habitable se toma. Esta cuestión nos obliga a probar las combinaciones posibles en i proyectos en j días, lo que nos transporta a un algoritmo exponencial, ya que se prueban los subconjuntos de tamaño i.

Esta situación es la que se quiere evitar, ya que se quiere mejorar el orden exponencial, por ello se desistió de la idea ya que no nos brinda una solución mejor que la que se tiene

hasta el momento. Luego también es percatable que organizar los proyectos por el costo en días que lleva no es eficiente, puesto que esta organización no siempre nos beneficia en realizar la menor cantidad de combinaciones.

También se encuentra la posibilidad de cambiar el patrón para organizar realizándose sobre la razón de costo en días y fecha de entrega, con el objetivo de ir colocándolos tal que queden situados lo más alejados posibles, pero igual no se tiene el control de las posiciones en días que tomen, por lo que igual hay que garantizar que no intersepten los días entre ellos, ya que esta idea no garantiza la intersección nula. Con toda esta lógica se llega a la conclusión que este algoritmo no cumple con la percepción de la programación dinámica, ya que $dp[i][k]$ no nos ayuda a precalcular los posteriores. Si los días a trabajar fuera en intervalos continuos, es decir, que los días que se tomen tuvieran que ser consecutivos, entonces esta idea fuera un poco más aceptable, pero este requisito no se cumple.

Idea 6:

(solution1)

El siguiente algoritmo se basa en, dados los días de entrega dispuestos de forma creciente, inicialmente acumular los días posibles de forma también creciente (a la entrega 1 se asociarán todos los días que tenga disponibles delante, que no sean festivos, hasta completar su costo de días, al día 2 se le asocian los días entre él y el día uno más los que hayan sobrado de los anteriores al primer día, o sea, los que no cupieron en este) este proceso se realiza de forma lineal por los n días, siendo 'n' la última fecha de entrega. Luego, se va por todas las entregas (siempre de forma creciente) analizando si es mejor dejar a esa entrega con sus días asociados, o por el contrario *trasladar* sus días hacia las fechas siguientes, esto pasará si la cantidad de días restantes para completar su costo en al menos dos de dichas fechas siguientes, es menor que la cantidad de días invertidos en el proyecto actual más 1 (este incremento en uno representa el día de la entrega del proyecto actual, ya que como se renunció a este proyecto, esta pasaría a ser un día posible para que los siguientes proyectos trabajen).

Al realizar el proceso de avance por los proyectos, se revisan los proyectos siguientes y se hace un proceso auxiliar de ordenación de los proyectos de fechas mayores, de forma creciente a partir de los días restantes que tienen para completarse (o sea, los días totales necesarios, menos los días actualmente invertidos en ellos) si al restar la cantidad de días invertidos en el proyecto actual menos los días restantes de las primeras posiciones resultantes del proceso auxiliar, se pueden restar al menos dos días antes de que volver negativa la cantidad de días que el actual puede invertir (o si esta nunca se vuelve negativa

a pesar de sustraerle dos o más procesos), entonces efectivamente es mejor invertir los días en los procesos siguientes que en este proceso actual. Además, todos los procesos por los que se pase, que hasta el momento no hayan logrado completar su costo, siempre enviarán hacia atrás sus días actualmente invertidos, pues no tiene sentido mantenerlos. Al pasar alguna de estas situaciones, los días invertidos en el proceso actual se intentarán mover hacia los siguientes, siempre que sea posible (o sea, que quepan) al aumentar los días invertidos en un proyecto, disminuyen los días pendientes, y al gastar días invertidos, incrementan los días pendientes. Al completar un proyecto, se incrementa además en uno los días invertidos, puesto que se cuenta el día de entrega del mismo, y se enciende una bandera que señala que se gasta ese día invertido para tener constancia de que ya esto se realizó (esta bandera también se enciende cuando se envían los días de un proceso a otro, en el proceso que envía, puesto que si ya se decidió no hacer ese proyecto, su día de entrega se puede usar también en los siguientes).

Al analizar confrontando las soluciones de casos (generados aleatoriamente mediante el *tester*) con las soluciones de la *fuerza bruta* el equipo se percató de que pueden ocurrir ocaciones en que al anallizar si vale la pena invertir días de un proyecto en otros de la forma antes explicada, puede darse el caso de que resulte en respuesta negativa, sin embargo, al enviar hacia adelante los días de los procesos incompletos, puede resultar en respuesta positiva después de dichas transformaciones. Este problema se solucionó realizando dos veces el ciclo principal, de forma que estos casos se vuelvan a analizar luego de las transformaciones.

Nuevamente se comparó las respuestas con las soluciones de la *fuerza bruta* y surgió un nuevo problema. Puede darse el caso en que el algoritmo decida que es necesario repartir los días de un proyecto, sin embargo, hay delante otro proyecto que puede también repartir sus días para completar los dos o más días que el anterior podía completar, pero tener más días disponibles y, por tanto, ser más óptimo para realizar esta acción. Para solucionar esto se realizó otro proceso auxiliar de mirar hacia adelante para comprobar la existencia de estos casos. Al recibir una respuesta afirmativa del *mirar hacia adelante*, se cancela la repartición de días del proceso actual y este mantiene sus días invertidos.

Al confrontar por tercera vez las soluciones, se llegó a un resultado de 10000 casos, solo 30 fallidos, y todos estos por una diferencia modular de 1, para desgracia de los miembros del equipo, nunca pudieron encontrar el arreglo necesario. Sin embargo, esto dio paso a nuevas vías de pensamiento.

Idea 7:

(greedy-solution)

Se halló una nueva posible solución al problema. El grupo piensa que es la acertada, pues se probó con 30000 casos generados aleatoriamente, confrontando los resultados con la *fuerza bruta*, y se obtuvo 0 diferencias de resultado), en este caso atacándolo con una idea *greedy*. El objetivo es arreglar los casos que no se estaban contemplando en la primera percepción de solución *greedy*, en este algoritmo es posible deshacerte de proyectos tal que se aumente la cantidad de posibles a resolver.

Primeramente se ordenarán los proyectos por fecha de entrega. Se tiene un arreglo de trabajo p de longitud 'mayor día de entrega de proyectos' el cual es boolean, aquí se recogen los días de proyectos que se van a marcar con los días que ocupa realizarlo respectivamente. Se avanza en el orden de las entregas, y en cada entrega se analiza cuál es la mejor opción: si poner el proyecto que se entrega en la fecha i y rellenar los días que ocupa, si quitar algunos de los puestos a realizar y colocar al que se está analizando, o simplemente no es conveniente agregarlo.
En el instante de analizar el proyecto j, se revisa si ocupa menos días que algún proyecto que se agregó, en este caso es mejor quitar el agregado y colocar este, para así aumentar la cantidad de días disponibles y facilitarle a los proyectos venideros más días que tomar. Se tendrán tres posibles acciones sobre el proyecto j:
- Primero se analiza si la cantidad de días disponibles hasta el momento es mayor o igual que las necesarios para su ejecución, de cumplirse entonces se agrega.
- Si no, se revisa si la cantidad de días disponibles más la cantidad de proyectos que no se realizaron hasta el día de entrega del proyecto j es mayor o igual que los necesarios para su ejecución, y se cae en el primer caso.
- En el caso que no se cumplan ninguna de las anteriores, entonces ese proyecto no es posible colocarlo y se comparan los días de trabajo que se lleva el proyecto j con el mayor costo en días de los proyectos agregados, si esta comparación es menor, significa que puedo cambiar este proyecto por el de mayor costo, no se agrega ningún proyecto nuevo pero aumenta la cantidad de días disponibles para el análisis de los proyectos venideros, y de esta manera aseguro que se puedan tomar la mayor cantidad de proyectos.

En *count* se guardará la mayor cantidad de proyectos que se pueden realizar. Es necesario demostrar que este algoritmo siempre devuelve una solución válida. Este requisito se cumple ya que siempre se colocan los proyectos que menor cantidad de días ocupen. No existirá el caso que se quede algún proyecto válido a realizar y no se agregue. Supongamos que ocurre, entonces significa que no existen días disponibles, el único caso para

que esto suceda es porque existe un proyecto agregado que impida que se inserten dos nuevos proyectos, y de esta manera poder aumentar el contador. Pero por la lógica del algoritmo siempre se insertarán y quedarán guardados aquellos proyectos que se lleven la menor cantidad de dias posibles, entonces los proyectos que no están es porque cambiando a alguno por algún proyecto agregado no mejorará esta cantidad, por lo que en este caso se llegaría a una contradicción ya que el proyecto válido que aumentara el contador de proyectos a realizar no es posible colocarlo.

DEMOSTRACIÓN

Sean los vectores de soluciones:

$$Opt = \{Y_1, Y_2, ..., Y_l\}$$
$$G = \{X_1, X_2, ..., X_r\}$$

Donde cada vector simboliza un proyecto escogido. Estas soluciones se encuentran ordenadas crecientemente por la fecha de entrega (tiempo de cierre).

Primero demostremos los siguientes lemas:
Lema 1: En cada iteración de G se cumple que un proyecto que no se haya puesto en iteraciones anteriores, nunca formará parte de la solución

Demostración:
Sea p1 el proyecto que el greedy retiró de la solución, la única forma en que pudiera tener una cantidad de espacios libres delante de él mayor o igual que su costo, sería que al analizar otro proyecto, en este caso p1, se hubiera retirado un proyecto p2, liberando una cantidad de espacios igual a la diferencia entre p2 y p1. Sea además y la cantidad de espacios que tiene vaciíos delante p0 durante la iteración actual(analizando p1). Es evidente que p0 $>$ p2 $>$ p1 $>$ y. Si p0 fuera menor que p2, entonces p0 estaría en la solución en lugar de p2, si p1 no fuera menor que p2, entonces no se hubiera quitado a p2 para poner a p1, y si y no fuera menor que p1, entonces p1 se hubiera colocado sin necesidad de quitar a nadie. Luego x = y + (p2 - p1 + 1), el +1 es el espacio que ocupa la entrega de p2. Entonces, siendo k = y + (-p1), k $<$ 0, entonces x = p2 + 1 + k, está claro que x $<=$ p2. Luego p0 $<$ x, lo que demuestra el lema pues no quedarán suficientes espacios libres delante de p0 aunque se quitara a cualquier otro proyecto.

Lema 2: En cada iteración i se cumple que la cantidad de días disponibles que va dejando el algoritmo greedy ($disp_G$) va a ser mayor o igual que los días disponibles que tenga una solución óptima ($disp_{Opt}$)

Demostración:

Supongamos que $disp_G < disp_{Opt}$
Significa que existe iteración i tal que hasta ese momento se encuentre

$$\sum_{k=1}^{l} Y_k \leq \sum_{k=1}^{r} X_k.$$

Quitemos de ambos lados de la relación aquellos que tengan igual duración (en días) en la misma iteración. Se tienen dos casos:

1: Si $X_k.d < Y_k.d$
Este caso no genera contradicción.

2: Si $X_k.d > Y_k.d$
Esto significa que ambos no están simultáneamente en G.

2.1: Si $X_k.f > Y_k.f$
Sería imposible porque $X_k.d > Y_k.d$.

2.2: Si $X_k.f < Y_k.f$
Se coloca X_k y luego se coloca Y_k, por máximo de ls duraciones X_m tal que $X_m.d > Y_k.d$ (existe al menos X_k)

En ambos casos disminuye $\sum X_k.d$.

Tras una cantidad finita de pasos nos queda:

$$\sum Y_k.d = \sum Z_k.d, Z_k = X_k \textit{ luego de ser modificado}$$

Como se conoce que esta relación no puede ser: $\sum Y_k.d < \sum X_k.d$

Entonces se demuestra que $\sum Y_k.d \geq \sum Z_k.d$

O lo que es lo mismo: $disp_G > disp_{Opt}$.

Se demostrará por inducción que en cada posición i de las soluciones G y Opt se cumple

que $X_i.fecha \leq Y_i.fecha$.

Inducción:

Paso inicial : $X_1.fecha \leq Y_1.fecha$
Supongamos que $X_1.fecha > Y_1.fecha$, si esto pasara entonces Y_1.duración $> X_1$.duración, de lo contrario $Y_1 \in G$ por Lema 1 y X_1 no puede estar simultáneamente con Y_1. O lo que es lo mismo: no existe $X_t = Y_1$ tal que $X_t \in G$. Por lo tanto se tiene que

$$X_1 \in G \text{ y } X_1 \text{ no} \in Opt, Y_1 \in Opt \text{ y } Y_1 \text{ no} \in G.$$

De la misma manera ocurre con X_2 y Y_2, y así sucesivamente.
Si se continúa con este análisis entonces no existe Z_t tal que $Z_t \in Opt$ y $Z_t \in G$. Esto genera una contradicción ya que existen muchos casos tal que hay proyectos que coinciden en ambas soluciones, por lo tanto X_1.fecha $\leq Y_1$.fecha.

Hipótesis: $\forall$ *posición i se cumple que* $X_i.fecha \leq Y_i.fecha$

Paso Inductivo: Si $X_i.fecha \leq Y_i.fecha$ *entonces* $X_{i+1}.fecha \leq Y_{i+1}.fecha$
Como Y_i y Y_{i+1} son compatibles entonces

$$X_i\text{.fecha} \leq Y_i\text{.fecha} \leq Y_{i+1}\text{.fecha}.$$

Además X_i.fecha $\leq X_{i+1}$.fecha .
Se tienen varios casos:

1: $X_{i+1}.fecha \leq Y_i.fecha \leq Y_{i+1}.fecha$
Aquí se cumple que X_{i+1}.fecha $\leq Y_{i+1}$.fecha.

2: $Y_i.fecha \leq X_{i+1}.fecha \leq Y_{i+1}.fecha$
Igualmente es un caso que demuestra X_{i+1}.fecha $\leq Y_{i+1}$.fecha.

3: $Y_i.fecha \leq Y_{i+1}.fecha \leq X_{i+1}.fecha$
En este caso es necesario demostrar que esto no puede pasar. Se dividen dos nuevos subcasos:
3.1 : $Y_i.fecha < Y_{i+1}.fecha = X_{i+1}.fecha$
Equivalente al caso 2.

3.2: $Y_i.fecha < Y_{i+1}.fecha < X_{i+1}.fecha$

El cual se ramifica en:
3.2.1: Si $Y_i.fecha = X_i.fecha$ ya que $X_i.fecha \leq Y_i.fecha$
X_i.fecha $< Y_{i+1}$.fecha $< X_{i+1}$.fecha.
Se cumple que Y_{i+1}.duración $> X_{i+1}$.duración. Se tiene que Y_{i+1} tuvo la capacidad de días disponibles y luego fue quitado para poner a X_{i+1} y Y_{i+1} tuvo el espacio para Opt.
En el caso que Y_{i+1} estuvo en alguna iteración en G, fue porque luego fue quitado. Si X_{i+1} quitó a Y_{i+1} de G fue porque el máximo de las duraciones de los proyectos marcados a realizar fue el de Y_{i+1}, entonces Y_{i+1} ocupa la mayor cantidad de días hasta el momento. En el punto i se cumple que

$$disp_{Opt} - Y_{i+1}.\text{duración} < disp_G - X_{i+1}.\text{duración},$$

esto se deriva de aplicar el Lema 2.
Por muy pequeño que sea X_{i+1} no tiene los días disponibles para ponerse por culpa de Y_{i+1}, esto mismo ocurre con Opt por Lema 2, si X_{i+1} no cabe en G también se cumple que

no existe Y_p , $p \in N \; \forall p \; i+2 \leq p \leq l$ tal que $Y_p \in Opt$ y tenga espacio disponible, Y_p.fecha $\geq X_{i+1}$.fecha y se cumpla además Y_p.duración $\geq X_{i+1}$.duración $+ c$, donde c son los días disponibles después de X_{i+1}.

No es necesario tomar los siguientes casos ya que pueden existir varios proyectos que ocupen Opt teniendo a Y_{i+1} pero estos son los que cumplen que

$$Y_p.\text{duración} \leq Y_{i+1}.\text{duración} + disp_{Opt},$$

y los mismos también se contemplan en G, por fusión de Lema 1 y Lema 2. Luego de este razonamiento nos queda que la cardinalidad de G pudiera ser mayor que Opt, lo cual es una contradicción. Por tanto X_{i+1}.fecha $\leq Y_{i+1}$.fecha.
En el caso que nunca estuvo Y_{i+1}, entonces queda claro que nunca se pudo poner en G, por lo tanto tampoco tendrá espacio para Opt (Lema 2).

3.2.2: $X_i.fecha < Y_i.fecha < Y_{i+1}.fecha < X_{i+1}.fecha$
Significa que Y_i, Y_{i+1} no pertenecen a G, entonces Y_i.duración $+ Y_{i+1}$.duración $> disp_G$. Por el Lema 1 se tiene que estos dos proyectos nunca podrán colocarse en G. Como en el algoritmo greedy siempre se colocan los proyectos de manera tal que se quede la mayor cantidad de días disponibles, se puede afirmar que si Y_i y Y_{i+1} no son posibles e colocarse en G tampoco es posible que puedan estar simultáneamente en Opt. por lo cual esta relación es una contradicción, y por tanto se cumple que X_{i+1}.fecha $\leq Y_{i+1}$.fecha por

Lema 2.

Por tanto se demostró por inducción que X_i.fecha $\leq Y_i$.fecha.
Como $r \leq l$, y r no puede ser menor que l ya que no se puede tener más óptimo que el procesado, X_r es el último procesado.
Por Lema 2 $disp_G > disp_{Opt}$ entonces la mayor cardinalidad de Opt es r, ya que en todo momento se trata de dejar la mayor cantidad de días disponibles posibles lo que resulta permitir agregar la mayor cantidad de proyectos posibles. Entonces $r = l$, por lo tanto G es una solución óptima.

Idea 7:

(greedy-solution-upgraded)

El algoritmo anterior tenía un costo de n^2, puesto que por cada iteración del ciclo principal (que visita los n proyectos), existe la posibilidad de que sea necesario revisar todos los proyectos de fecha de entrega menor en busca de uno de mayor costo, para introducir el nuevo en la respuesta y retirar el anterior. Luego de analizar esto, fue evidente que no era necesario revisarlos todos, es posible, mediante un heap, mantener en $\log n$ siempre al posible retirado. Con una simple modificación, la complejidad temporal del algoritmo disminuyó a $n * \log n$. Las anteriores ideas y demostraciones, claramente, se mantienen para el algoritmo nuevo.