Notaciones Asintóticas

O(f(n)) (Notación "Big-O"):

- Indica una **cota superior** asintótica.
- Por ejemplo, $T(n) \in O(f(n))$ significa que para n suficientemente grande, T(n) crece como máximo tan rápido como f(n), hasta una constante:

  $T(n) \leq c \cdot f(n)$ para algún $c>0$ y $n \geq n0$.

o(f(n)) (Notación "Little-o"):

- Indica que una función crece **estrictamente más lento** que f(n).
- Por ejemplo, $T(n) \in o(f(n))$ significa que para n suficientemente grande, T(n) es asintóticamente insignificante en comparación con f(n):

  $\lim_{n \to \infty} T(n)/f(n)=0$.

Ω(f(n)):

- Indica una **cota inferior** asintótica.
- Por ejemplo, $T(n) \in \Omega(f(n))$ significa que para n suficientemente grande, T(n) crece al menos tan rápido como f(n):

  $T(n) \geq c \cdot f(n)$ para algún $c>0$ y $n \geq n0$.

Ejemplo: Cuando se dice que $O(n)=o(n \log n)$, significa que el tiempo O(n) crece estrictamente más lento que $n \log n$.

Para los tiempos de cada algoritmo revisar sección de resumen de cada capítulo del apunte.

Cotas inferiores

Adversario

- El **argumento del adversario** es una técnica utilizada para demostrar que cualquier algoritmo que resuelva el problema necesita realizar al menos un cierto número de comparaciones en el peor caso.

Teoría de la información

- Está relacionado con el análisis de **cuánta información mínima** es necesaria para determinar la solución correcta a un problema.
- Si un problema tiene N posibles soluciones, necesitamos al menos $\log_2(N)$ bits de información para distinguir entre ellas.
- implica que cualquier algoritmo para resolver el problema debe realizar un número de operaciones suficiente para recuperar esa cantidad de información.
- Aproximación de Stirling $\log_2 n! \approx n \log_2 n$
- Permite establecer una cota inf. de $\Omega(n \log n)$ para alg. basados en comparaciones.

Reducciones

Se tiene un problema conocido A, y el del enunciado B, para las reducciones se reduce el problema A conocido al problema B planteado. Para ello se utiliza un problema conocido conveniente que tenga la misma cota buscada por demostrar para el problema B.

Ejemplo: Reducir ordenamiento (**conocido**) a emparejamiento (**desconocido**)

Pasos de una reducción:

1. **Definir los problemas**:
    - P1: Problema **conocido**, cuya cota inferior ya sabemos (en este caso, ordenamiento, que requiere Ω(n logn)).
    - P2: Problema **desconocido** cuyo tiempo mínimo queremos demostrar (en este caso, emparejamiento).
2. **Transformar una instancia de P1 en una instancia de P2**:
    - Diseña una transformación que convierte cualquier entrada de P1 en una entrada de P2. Debe ser eficiente (en este caso, O(n)).
3. **Resolver P2 con un algoritmo hipotético**:
    - Supongamos que tenemos un algoritmo eficiente para P2 que toma tiempo menor a la cota que queremos demostrar (digamos, o(n logn)).
4. **Convertir la solución de P2 en una solución de P1**:
    - Diseña una transformación que convierta la solución de P20 en la solución de P1 de manera eficiente (nuevamente, en O(n)).
5. **Concluir que resolver P2 en tiempo menor a Ω(n logn) implicaría resolver P1 en tiempo menor a Ω(n logn)**, lo cual es imposible por la cota inferior conocida de P1.

Memoria

Memoria Principal (RAM)

- Rápida y volátil, lo que significa que los datos almacenados en ella se pierden cuando el sistema se apaga.
- Se utiliza para almacenar datos e instrucciones que el procesador necesita de manera inmediata o frecuente.
- Tiene capacidad limitada, y el espacio disponible suele ser mucho menor que el tamaño total de los datos procesados.

Memoria Secundaria (dispositivos de almacenamiento: discos duros, ssh)

- Dispositivos de almacenamiento más grandes y lentos
- Conserva los datos incluso cuando el sistema se apaga.
- Almacena grandes volúmenes de datos, como archivos del sistema operativo, programas y datos de usuario.
- Se recurre a la memoria secundaria cuando los datos son demasiado grandes para caber en la memoria principal (RAM).

Árboles B

Variante específica de los árboles balanceados, diseñados especialmente para operar de manera eficiente en **memoria externa.**

- Los árboles B almacenan múltiples claves en un solo nodo. Esto permite que el árbol sea más "ancho" y menos "profundo".
- Cada nodo ocupa una página de disco. Al tener múltiples claves por nodo, se optimiza el número de accesos al disco.
- Inserción y búsqueda$O(\log_m n)$ m: bloques que caben en mem. secundaria; n: nro. claves.
- Todos los nodos tienen un número de claves entre ⌈m/2⌉−1 y m−1 excepto la raíz (que puede tener menos claves), donde m es el grado del árbol.
- Es un árbol balanceado: todas las hojas están al mismo nivel.
- Cada nodo tiene entre ⌈m/2⌉ y m hijos, excepto las hojas.

Ordenamiento

El ordenamiento en memoria externa es necesario cuando los datos no caben en la memoria principal. Se utiliza un enfoque por etapas para minimizar las operaciones de entrada/salida (I/O).

Sort(n) representa el costo de ordenar n elementos almacenados en disco, donde

- M: Capacidad de la memoria principal en enteros.
- B: Tamaño de un bloque de disco en enteros.
- n: Número de elementos que queremos ordenar

El algoritmo de ordenamiento más común utilizado en este modelo es **Merge Sort Externo (secundaria)**, optimizado para minimizar las operaciones de entrada/salida.

El costo total de Merge Sort Externo (**Mem. Secundaria**) es $O\left(\frac{n}{B} \cdot \log_{M/B}\left(\frac{n}{M}\right)\right)$

En **Memoria Principal** el costo total de Merge Sort es $O(n \log n)$.

Colas de prioridad

Las **colas de prioridad** en memoria externa se implementan con estructuras diseñadas para minimizar accesos a disco.

Cola de prioridad limitada

En una cola de prioridad limitada, los valores de prioridad están restringidos a un rango discreto y conocido de antemano.

1. Se puede usar un arreglo de p listas, donde cada índice i representa un nivel de prioridad.
2. Cada lista almacena los elementos con esa prioridad en el orden en que fueron insertados.
3. Costos: Insertar O(1), Búsqueda O(p) para p niveles.

Cola de prioridad general

Los valores de prioridad no están restringidos a un rango fijo. Las prioridades pueden ser valores arbitrarios.

Heaps O(log n) HeapSort (n log n)

Heap binario:

- Es una estructura completa donde el nodo raíz tiene la mayor (o menor) prioridad y cada nodo hijo tiene menor (o mayor) prioridad que su padre.
- Inserción O(log n), se agrega el elemento al final del heap y luego se ajusta la estructura con una operación de *heapify* hacia arriba.
- Búsqueda O(n), se elimina la raíz (máximo o mínimo) y se ajusta la estructura con una operación de *heapify* hacia abajo.

Colas implementadas con árboles balanceados (como árboles AVL o árboles rojo-negro):

- Permiten almacenar elementos con prioridades arbitrarias y mantener el orden eficiente.

Un **árbol balanceado** es una estructura de datos en forma de árbol en la que se asegura que la diferencia entre las alturas de los subárboles izquierdo y derecho de cualquier nodo sea pequeña (generalmente, menor o igual a 1). Esto permite que las operaciones como búsqueda, inserción y eliminación se realicen en un tiempo eficiente, típicamente $O(log\ n)$.

Árbol AVL:

- Garantiza que la diferencia de altura entre los subárboles de cualquier nodo no sea mayor a 1.
- Después de una inserción o eliminación, puede realizar **rotaciones** para mantener el balance.

Árbol B y B+:

- Mantiene múltiples claves por nodo para minimizar accesos al disco

Hashing

El **hashing** es una técnica para acceder a datos en **O(1)** (esperado). En memoria externa, se utilizan variantes que optimizan el uso de bloques de disco.

4.1. Hashing Lineal

- Utiliza un esquema de división modular para almacenar claves.
- Si un bucket se llena, se crea un bucket adicional y las claves se redistribuyen parcialmente.
- Inserción: O(1), pero puede ser O(n) en casos extremos. Búsqueda: O(1).

4.2. Hashing Extendible

- Expande dinámicamente el espacio de direcciones sin rehashing completo.
- Usa un directorio que apunta a los buckets, el cual puede duplicarse si es necesario.
- Inserción y búsqueda: O(1) esperado, con O(logn) en el peor caso

R-Trees

Agrupa elementos en regiones mínimas delimitadas por rectángulos. Los nodos contienen un número fijo de entradas, cada una representando un rectángulo delimitador (bounding box) y un puntero a un nodo hijo o datos.

Búsqueda:

- $O\left(n^{\beta}\right)\ con\ \beta\ la\ prob.de\ intersección$
- I**nserción**: Inserta elementos en el nodo más apropiado, basándose en el incremento mínimo del área. $O(\log_B N)$

B representa el **tamaño de un bloque en memoria secundaria (disco)**

Razón del diseño típico (k=B):

- Cada nodo cabe en un bloque de memoria secundaria.Al hacer que k=B, un nodo puede almacenar hasta B−1 claves, maximizando el uso de cada acceso al disco.
- Esto minimiza la cantidad de accesos al disco, ya que cada vez que se lee un nodo del B-tree, se aprovecha al máximo la información contenida en el bloque correspondiente.

Por ejemplo, si el tamaño de un bloque es 4 KB y cada clave ocupa 16 bytes, entonces B=4,096/16=256. Cada nodo puede almacenar hasta 255 claves y tener 256 hijos.

La altura del B-tree es O(logB N) con N: Número de claves almacenadas y B: Número máximo de hijos por nodo (tamaño del bloque por lo general).

Tiempo de búsqueda, inserción y eliminación: O($log_B N$ +B)

Amortizado

La fórmula general del costo amortizado es:

$$Costo\ amortizado = \frac{Costo\ total}{Núm.\ de\ operaciones} = \frac{T(n)}{n}$$

Método de contabilidad de costos

- El método de contabilidad asigna un **costo amortizado** a cada operación individual, que puede ser mayor o menor que el costo real de esa operación. El costo amortizado se establece de tal manera que cubra no solo el costo actual de la operación, sino también "ahorre" parte del costo para operaciones futuras que puedan ser más caras.

Método de función potencial

- El método de función potencial utiliza una función matemática para medir el "estado" del sistema en cualquier momento de la secuencia de operaciones. Este enfoque asigna costos amortizados basados en cómo cambia el estado del sistema antes y después de cada operación.
- Función potencial Φ mide la cantidad de "trabajo pendiente"

El costo amortizado de una operación i se define como:

$$Costo\ Amortizado = Costo\ real + \Delta\phi$$

Donde $\Delta\phi = \phi_{después\ de\ la\ operación} - \phi_{antes\ de\ la\ operación}$

Colas Binomiales

Estructura de datos basada en un conjunto de árboles binomiales, que permite manejar una cola de prioridad con operaciones eficientes. Los árboles binomiales tienen propiedades que permiten combinar rápidamente dos colas.

Inserción: el peor caso es O(logn), porque puede implicar O(logn) uniones de árboles. Amortizado O(1)

Eliminar mínimo y unir dos colas O(log n)

Colas de Fibonacci

Utilizan una colección de árboles más flexible y menos estricta que las colas binomiales, retrasando las uniones de árboles hasta que sean absolutamente necesarias.

Insertar: Se agrega un nuevo nodo como un árbol independiente O(1)

Eliminar mínimo: O(log n) Decrease-key: Costo amortizado O(1)

Union-Find

El **Union-Find** es una estructura de datos para manejar conjuntos disjuntos, que soporta las operaciones:

Find: Determinar el representante de un conjunto.

Union: Combinar dos conjuntos en uno.

Siempre se une el árbol mas pequeño al mas grande. Durante find, cada nodo visitado se conecta directamente a la raíz, reduciendo la profundidad del árbol.

Costo amortizado las operaciones cuestan $O(\alpha(n))$ que es la inversa de la fn. de Ackermann.

Splay Trees

Estructura de árbol binario de búsqueda autoajustable. Después de cada operación árbol se reestructura para mover el elemento accedido recientemente hacia la raíz.

Operación Splay: Peor caso O(n), costo amortizado O(logn)

Zig

Se realiza cuando el nodo x accedido es el **hijo directo de la raíz**. Solo implica una rotación simple.

Si x es el hijo izquierdo de la raíz. Se realiza una **rotación a la derecha** sobre la raíz, y viceversa.

Zig-Zig

Ocurre cuando x y su padre (p) están en el mismo lado respecto al abuelo (g).

Zig-zig izquierda: x es el hijo izquierdo de p, y p es el hijo izquierdo de g

Zig-zig derecha: x es el hijo derecho de p, y p es el hijo derecho de g.

Se realizan dos rotaciones del mismo tipo (ambas derecha o ambas izquierda)

Zig-Zag

Ocurre cuando x y su padre (p) están en lados opuestos respecto al abuelo (g).

Zig-zag izquierda-derecha: x es el hijo derecho de p, y p es el hijo izquierdo de g.(Rotación derecha el hijo, izq el padre)

Zig-zag derecha-izquierda: x es el hijo izquierdo de p, y p es el hijo derecho de g.(Rotación izq el hijo, derecha el padre)

Universos discretos y finitos

Counting Sort: Algoritmo de ordenamiento que no compara elementos, sino que cuenta cuántas veces aparece cada valor en el conjunto.

Bucket Sort

Divide los elementos en buckets (intervalos), ordena los elementos dentro de cada bucket, y luego concatena los buckets en orden.

Radix Sort

Ordena los elementos procesando sus dígitos, comenzando desde el dígito menos significativo hasta el más significativo. Usa Counting Sort como subrutina para ordenar por cada dígito.

van Emde Boas Tree

Estructura de datos para manejar conjuntos con un rango discreto [0,u−1], donde u es el tamaño del universo. divide el universo en "clusters" (subconjuntos) de claves. Soporta operaciones como búsqueda, inserción, eliminación, y predecesor/sucesor en O(loglogu).

Tries

Estructura de árbol para almacenar cadenas de texto, donde cada nodo representa un prefijo común.

Árboles Patricia

Variante compacta de los tries que elimina nodos con un único hijo (los nodos que no aportan información). Optimiza el espacio en comparación con los tries. Costo en estructura $O(log_b\, n)$. Las claves se almacenan en nodos solo cuando comparten un prefijo común.

Árbol de sufijo

Un árbol de sufijo es una estructura que almacena todos los sufijos de una cadena en forma de un trie compacto.

Arreglo de sufijo

Es un arreglo que almacena los índices de todos los sufijos de una cadena en orden lexicográfico.

El **modelo de comparaciones** es un paradigma para analizar algoritmos en el cual:

1. **Operación principal**: El algoritmo sólo puede comparar dos elementos a la vez para decidir cómo proceder. Estas comparaciones son las operaciones fundamentales para resolver un problema.
2. **Restricción**: No se permite ningún otro tipo de operación sobre los datos, como acceso directo a índices, aritmética avanzada u otras propiedades especiales de los datos.

Algoritmos Probabilísticos y Aleatorizados

Monte Carlo

Un algoritmo Monte Carlo es un algoritmo aleatorizado que puede producir un resultado incorrecto, pero el tiempo de ejecución está garantizado.

1. Exactitud no garantizada:

- Puede devolver un resultado incorrecto, pero con una probabilidad de error conocida y controlable.
- Es posible reducir la probabilidad de error al ejecutar el algoritmo varias veces.

2. Tiempo de ejecución fijo

One-Sided Monte Carlo

Uno de los resultados es determinístico y correcto. El error solo ocurre para el otro resultado

Ejemplo: Si el algoritmo dice que n no es primo (siempre correcto) & Si el algoritmo dice que n es primo (puede fallar).

Two-Sided Monte Carlo

Ambos resultados ("sí" y "no") tienen una probabilidad de error asociada (controlable.

Las Vegas

Un algoritmo **Las Vegas** es un algoritmo aleatorizado que **siempre produce una solución correcta**, pero el tiempo de ejecución puede variar dependiendo de las decisiones aleatorias que tome.

1. **Exactitud garantizada**:

- El resultado es siempre correcto, sin importar las decisiones aleatorias.
- Si no puede garantizar la solución correcta, el algoritmo sigue ejecutándose hasta encontrarla.

2. **Tiempo de ejecución variable**:

- El tiempo de ejecución depende de las decisiones aleatorias, pero **siempre termina** (con probabilidad 1).
- El análisis del tiempo suele hacerse en términos del **tiempo de ejecución esperado**.

Numero esperado de pasadas

- El número esperado de pasadas se calcula utilizando la **esperanza matemática** de una distribución geométrica, ya que estamos modelando el número de intentos hasta el primer éxito.
- La probabilidad de éxito en una sola pasada es p=1/2.

- La esperanza de una variable geométrica, que cuenta el número de intentos hasta el primer éxito. E[número de pasadas]=1/p.
  Distribución geométrica
  - ➔ Número de intentos hasta obtener el primer éxito
  - ➔ La **esperanza (valor esperado)** de una distribución geométrica con probabilidad de éxito p es: $E(X) = \frac{1}{p}$

Probabilidad de error

La probabilidad de error se refiere a la probabilidad de que el algoritmo **no identifique correctamente al mayoritario**. Esto sucede si después de un número k de pasadas, no se ha seleccionado el elemento mayoritario. P(error después de k pasadas)

Esperanza de una variable aleatoria

La fórmula general para calcular la esperanza de una variable aleatoria es:

$E[X] = \sum P(x_i) \cdot x_i$

Donde:

- **E[X]** es la **esperanza** o valor esperado de la variable aleatoria **X**.
- **$P(x_i)$** es la **probabilidad** de que **X** tome el valor **x_i**.
- **x_i** es el valor de la variable aleatoria en la **i-ésima** posible salida.

Factor de aproximación

$$Factor\ de\ aproximación = \frac{Valor\ esperado\ de\ la\ solución\ aleatoria}{Valor\ óptimo\ de\ la\ solución\ (nro\ máx\ de\ clausulas\ satisfechas)}$$

Ejemplo: Un factor de aproximación de 2, indica que el número de cajas utilizadas por el algoritmo nunca excede el doble del número óptimo de cajas.

El valor esperado de la solución aleatoria corresponde a lo obtenido por el algoritmo.

$$SOL \geq \frac{OPT}{2}\ es\ una\ 2 - aproximación$$

K-Competitivo

Un algoritmo es **k-competitivo** si su rendimiento en un problema **online** (donde las decisiones deben tomarse sin conocimiento completo del futuro) está dentro de un factor k del rendimiento del algoritmo **óptimo** (que tiene conocimiento completo del futuro).

$C_A(I) \leq K \cdot C_{OPT}(I)$+c

- $C_A(I)$: Es el costo del algoritmo online A para una instancia I del problema.
- $C_{OPT}(I)$: Es el costo del algoritmo óptimo offline para la misma instancia I.
- $k \geq 1$: Es el **factor de competitividad**, que mide cuán cerca está el algoritmo online del óptimo.
- c: Es una constante que no depende del tamaño de la entrada I.

Árboles Aleatorizados

Los árboles aleatorizados son estructuras de datos que utilizan aleatorización para mantener propiedades de balance, mejorando el rendimiento esperado de las operaciones.

Skip Lists

Permite operaciones de búsqueda, inserción y eliminación en O(log⁡n)O(\log n)O(logn) promedio.

Consiste en **varios niveles de listas enlazadas**, donde cada nivel salta elementos para acelerar la búsqueda.

Treaps

Los treaps combinan propiedades de árboles binarios de búsqueda y heaps.

Cada nodo tiene una **clave** (para el orden del árbol binario) y una **prioridad aleatoria** (para la propiedad de heap). Búsqueda, inserción y eliminación en O(log n)

Hashing Universal

Un esquema de hashing es **universal** si minimiza las colisiones en cualquier conjunto de claves, independientemente de cómo estén distribuidas. Se selecciona una familia de funciones hash H de manera que para cualquier par de claves x e y. Inserción y búsqueda O(1)

Hashing Perfecto

Un esquema de hashing es **perfecto** si no hay colisiones entre las claves. Utiliza un enfoque en dos niveles: En el primer nivel, una función hash divide las claves en grupos. En el segundo nivel, cada grupo utiliza su propia tabla hash, cuidadosamente diseñada para evitar colisiones. Construcción O(n), Búsqueda O(1).

Ex - 2021

P1 Se desea una estructura de datos para almacenar un conjunto de strings en memoria secundaria. Estos strings pueden ser bastante largos, posiblemente ocupando varias paginas de disco.

1. Diseñe una variante directa del B-tree para almacenar estos strings: describa la estructura de datos y como se busca (no necesita considerar los algoritmos de insercion y borrado). Analice el costo de busqueda en el modelo de memoria secundaria e indique que problema tiene su estructura comparada con un B-tree de numeros.

Sol: Cada nodo del B-Tree tiene múltiples claves (strings), en cada nodo se almacena un puntero a la página de disco donde está almacenado el string completo. Para la búsqueda se navega a través de los nodos del B-Tree, comparando las claves con el string buscado. Una vez obtenido el puntero se accede a la página en disco y al string completo para comparar.

Costo: Búsqueda depende de altura del B-Tree: altura h=$O(\log_b n)$;b=val. ramificación, además en cada nodo que se recorre se realiza una I/O para cargar la pág. en memoria.

Un B-Tree de números puede almacenar los valores directamente en los nodos, lo que hace las comparaciones más rápidas en RAM. Sin embargo, para string largos no se puede almacenar el contenido en los nodos debido al tamaño de los datos, y por tanto, tiene introduce un mayor costo de acceso a mem. secundaria.

2. Considere ahora una variante donde en cada nodo del B-tree se guarda un arbol Patricia de las claves en vez de las claves directamente. Nuevamente describa la estructura y el algoritmo de busqueda, y analice el costo de busqueda. Como se compara el costo con el de un B-tree de numeros? Es el resultado optimo en el modelo de comparaciones?

Sol: El **B-tree de números** tiene una estructura más simple, ya que los nodos contienen directamente las claves numéricas, lo que permite comparaciones directas en memoria. El B-tree con árboles Patricia introduce una **complicación adicional** en términos de búsqueda dentro de los nodos, pero la búsqueda es más eficiente en el sentido de que reduce las comparaciones. El modelo propuesto no necesariamente es más óptimo.

P2 Se tiene una secuencia P[1,n] de números reales positivos y un rango real positivo [x,y],0<x≤y. Se desea encontrar todas las posiciones j tal que, para algun i≤j, $x \leq \sum_{k=i}^{j} P[k] \leq y$.
1.Demuestre que el problema no se puede resolver con menos de n accesos a P.

Sol: Por estrategia del adversario, podemos definir que este responderá a la suma de P[k] con números reales y positivos manteniendo que la sumatoria es mayor igual a x, pero que se requiera acceder al último valor para saber si es menor que y, necesitando hacer n accesos.

2. Disene un algoritmo de tiempo O(n) que resuelva el problema. Demuestre su correctitud y use alguna tecnica de analisis amortizado para analizarlo.

Sol: Realizamos un calculo de las sumas acumuladas para S[k] con k=1 hasta n. Luego cuando el algoritmo desee saber si se cumple la condición, basta que accede a las posiciones i, j del arreglo con los resultados. Cada intervalo es recorrido una sola vez, sin necesidad de recalcular la suma gracias al uso de los índices. El algoritmo realiza $O(n)$ operaciones de manera amortizada ya que cada elemento se añade y elimina de la suma de manera constante.

P3 Dado un grafo no dirigido G = (V,E), considere el problema de dividir V = S U S', con $S' = V - S$, de manera de maximizar la cantidad de aristas que cruzan entre S a S'.

1. Considere el algoritmo que toma cada vertice y tira una moneda para decidir si lo pone en S o S'. Pruebe que, en el caso esperado, se obtiene una 2-aproximacion. Hint: considere que, si de v_i salen e_i aristas hacia $v_1, \dots, v_{i-1}$, al poner v_i en S habra k_i aristas que cruzan a S' y al ponerlo en S' habra $e_i - k_i$ aristas que cruzan a S.

Sol: Para cada vertice v_i se lanza una moneda para decidir si va a S o S'. Es decir, prob. = 50%, y por tanto el nro. esperado de aristas cruzadas es la mitad del nro. total de aristas que podría cruzar. Como el nro. esperado de aristas cruzadas es ½ del nro. De aristas en la partición optima, es una 2-aproximación.

2. Considere el algoritmo determinstico que toma cada vertice v_i, para i = 1,...,|V| , y lo pone en S o en S' segun en cual de los dos se maximice la cantidad de aristas entre v_i y $v_1, \dots, v_{i-1}$ que cruzan entre S y S'. Pruebe que este algoritmo es una 2-aproximacion.

Sol: Para cada v_i lo colocamos en S o S' según el lugar donde más aristas se cruzan. Esto quiere decir que, para cada vertice, se maximiza el número de aristas cruzadas, asegurando que la solución es la menos la mitad de la óptima. Es decir, un 2-aproximación.