Capítulo 4

Algoritmos no convencionales

ALGORITMOS PARALELOS

Conceptos básicos

- Modelo de memoria compartida
  - Consiste de un número de procesadores
  - Cada procesador tiene su propia memoria local
  - Pueden intercambiar datos entre si a través de una unidad de memoria compartida
  - Cada procesador se identifica de manera única a través de un índice (*id* del procesador)
- Modelo síncrono
  - Todos los procesadores operan en forma sincronizada bajo el control de un reloj común
  - *Parallel random-access machine* (PRAM)

Conceptos básicos

- Variantes del modelo PRAM
  - *Exclusive read exclusive write* (EREW): no permite el acceso simultáneo a una misma posición de la memoria
  - *Concurrent read exclusive write* (CREW): permite acceso simultáneo sólo para instrucciones de lectura
  - *Concurrent read concurrent write* (CRCW): permite acceso simultáneo para lectura y escritura

Conceptos básicos

- Tiempo de ejecución de un algoritmo: $T(n,p)$
  - n: tamaño del problema
  - p: número de procesadores
- *Speedup* del algoritmo:

$$S(n,p) = \frac{T(n,1)}{T(n,p)}$$

- $T(n,1)$: mejor algoritmo secuencial conocido

Conceptos básicos

- Eficiencia de un algoritmo paralelo:

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T(n,1)}{p \cdot T(n,p)}$$

  - Corresponde a la tasa de uso de los procesadores
  - 0 < E(n,p) <= 1
  - Algoritmo paralelo alcanza *speedup* óptimo cuando $p = \frac{T(n,1)}{T(n,p)}$

Conceptos básicos

- Se busca diseñar algoritmos que funcionen para la mayor cantidad de valores de p posible
- Principio de folding: en general se cumple

$$T(n,p) = X \Rightarrow T(n,p/k) \approx kX,\ k > 1$$

  - Se puede usar un factor k de procesadores menos a un costo en tiempo factor k
    - No aplica para todas las situaciones!
      - Pueden haber dependencias
      - Puede tomar tiempo decidir qué procesador emular

Conceptos básicos

- Lema de Brent
  - Si existe un algoritmo EREW con $T(n,p)=O(t)$, tal que $W(n)=s$ (trabajo total), entonces existe un algoritmo EREW con $T(n,s/t)=O(t)$
  - Demostración (1):
    - Sea a_i el # de instrucciones ejecutadas en el paso i del algoritmo por todos los procesadores (1 <= i <= t)
      - Por definición de trabajo total:

$$\sum_{i=1}^{t} a_i = s$$

Conceptos básicos

- Lema de Brent
  - Si existe un algoritmo EREW con $T(n,p)=O(t)$, tal que $W(n)=s$ (trabajo total), entonces existe un algoritmo EREW con $T(n,s/t)=O(t)$
  - Demostración (2):
    - Si $a_i <= s/t$, hay procesadores suficientes para realizar este paso sin cambios
    - Sino, reemplazar el paso i por ceil(a_i / (s/t)) pasos, donde los s/t procesadores simulan los pasos realizados por los p procesadores originales

Conceptos básicos

- Lema de Brent
  - Si existe un algoritmo EREW con $T(n,p)=O(t)$, tal que $W(n)=s$ (trabajo total), entonces existe un algoritmo EREW con $T(n,s/t)=O(t)$
  - Demostración (3):
    - El número total de pasos ahora es:

$$\sum_{i=1}^{t}\lceil\frac{a_i}{s/t}\rceil \leq \sum_{i=1}^{t}\left(\frac{a_i t}{s}+1\right) = t+\frac{t}{s}\sum_{i=1}^{t}a_i = 2t$$

Algoritmo para encontrar el máximo

- Problema: Encontrar el máximo entre *n* números en un arreglo
  - Algoritmo EREW basado en torneo (p=n proc.):
    - T(n,n) = T(n/2,n/2) + O(1) = O(log n)
    - S(n,n) = n/log n
    - E(n,n) = S(n,n)/p = 1/log n
    - W(n) = n/2 + n/4 + n/8 + … <= n
  - Usando Lema de Brent, n/log n procesadores:

$$T\left(n, \frac{n}{\log n}\right) = O(\log n), \quad E\left(n, \frac{n}{\log n}\right) = O(1)$$

Algoritmo para encontrar el máximo

- Problema: Encontrar el máximo entre *n* números en un arreglo
  - Algoritmo con p = n/log n procesadores:
    - Dividir la entrada en n / log n grupos de log n valores
    - Asignar un procesador a cada grupo
    - Cada procesador encuentra el máximo de su grupo en tiempo log n (búsqueda secuencial)
    - Los n / log n ganadores se procesan utilizando el algoritmo del torneo
      - T(n, n/log n) = O(log n)
      - S(n,n) = n/log n
      - E(n, n/log n) = S(n,n)/p = O(1)

Algoritmo para encontrar el máximo

- Otro algoritmo: CRCW
- Número de procesadores:

$$p = \frac{n(n-1)}{2}$$

- Se asigna el procesador $P_{i,j}$ a cada par [*i*,*j*] de elementos
- Se asigna la variable compartida v_i para cada elemento x_i, inicializada en 1

Algoritmo para encontrar el máximo

- Algoritmo de dos pasos
  - Paso 1: Cada procesador compara su par de elementos y escribe un 0 en la variable compartida asociada al menor de ellos
    - Como sólo un elemento es el mayor de todos, sólo un v_i queda con el valor 1
  - Paso 2: Los procesadores asociados al ganador pueden determinar que él es el ganador y lo anuncian

Algoritmo para encontrar el máximo

- Este algoritmo requiere de 2 pasos, independientemente de *n*:

$$T\left(n, \frac{n^2}{2}\right) = O(1)$$

- Eficiencia es muy baja:

$$E(n,p) = \frac{T(n,1)}{p \cdot T(n,p)} = \frac{n}{\frac{n^2}{2} \cdot O(1)} = O\left(\frac{1}{n}\right)$$

Algoritmo para encontrar el máximo

- ¿Cómo mejorar la eficiencia de este algoritmo?
  - Idea: dividir la entrada en grupos tal que se pueden asignar suficientes procesadores para encontrar el máximo de cada grupo usando el algoritmo de 2 pasos
  - Si el grupo tiene tamaño k, se necesitan $k(k-1)/2$ procesadores para encontrar el máximo en tiempo constante

Algoritmo para encontrar el máximo

- Se tienen *n* procesadores (*n* potencia de 2)
  - Paso 1: tamaño de cada grupo es 2, toma $O(1)$
  - Paso 2: quedan $n/2$ elementos y *n* procesadores
    - Si el tamaño de cada grupo es 4, resultan $n/8$ grupos con 8 procesadores cada uno
    - Esto es suficiente: $4(4-1)/2 = 6 < 8$
  - Paso 3: quedan $n/8$ elementos y *n* procesadores, ¿tamaño máximo de grupo?
    - Si el tamaño de cada grupo es *g*, resultan $n/8g$ grupos con $8g$ procesadores cada uno
    - Se necesita que $g(g-1)/2 <= 8g$, lo cual da $g=16$

Algoritmo para encontrar el máximo

- Resumiendo:
  - Se necesitan $k^2/2$ procesadores para hallar máximo entre k números en tiempo O(1)
  - n*(1/2) grupos de k=2 números, $n/2$ máximos, usa ($n/2$)* ($k^2/2$)=($n/2$)*2=n procesadores
  - ($n/2$)*(1/4) grupos de k=4 números, $n/8$ máximos, usa ($n/8$)* ($k^2/2$)=($n/8$)*8=n procesadores
  - ($n/8$)*(1/16) grupos de k=16 números, $n/128$ máximos, usa ($n/128$)* ($k^2/2$)=($n/128$)*128=n procesadores

Algoritmo para encontrar el máximo

- En el *i*-ésimo paso, aumentar tamaño del grupo al cuadrado:

$$\frac{n}{2^{2^{i-1}}} \text{ grupos de } 2^{2^{i-1}} \text{ elementos}$$

$$\frac{n}{2^{2^{i-1}}} \text{ maximos}$$

$$\frac{n}{2^{2^{i-1}}} \cdot 2^{2^{i-1}} = n \text{ procesadores}$$

- En total, el algoritmo requiere $O(\log \log n)$ pasos

Algoritmo para encontrar el máximo

- Complejidad del algoritmo:

$$T(n,n) = O(\log \log n)$$

- Eficiencia:

$$E(n,p) = \frac{T(n,1)}{p \cdot T(n,p)} = \frac{n-1}{n \cdot \log \log n} = O\left(\frac{1}{\log \log n}\right)$$

Algoritmo para encontrar el máximo

- Esta técnica se conoce como *divide and crush*
  - Se divide la entrada en grupos de tamaño suficientemente pequeño
  - Cada grupo se puede “aplastar” con muchos procesadores

Prefijo en paralelo

- Sea $\bullet$ (producto) una operación binaria asociativa:

$$x \bullet (y \bullet z) = (x \bullet y) \bullet z$$

- Ejemplos: +, *, max
- Problema: Dada una secuencia de números x_1, …, x_n, calcular el producto

$$x_1 \bullet x_2 \bullet \cdots \bullet x_k, \ \forall k, \ 1 \leq k \leq n$$

Prefijo en paralelo

- Se denotará

$$PR(i,j) = x_i \bullet x_{i+1} \bullet \cdots \bullet x_j$$

- Objetivo: Encontrar *PR*(1,*k*) para todo *k* entre 1 y *n*
- Versión secuencial de este problema es trivial: se calculan los prefijos en orden

$$T(n,1) = O(n)$$

Prefijo en paralelo

- Algoritmo CREW
- Método: *divide and conquer*
  - Supondremos que *n* es potencia de 2
  - Hipótesis de inducción:
    - Sabemos cómo calcular el prefijo en paralelo para *n*/2 elementos
  - Caso base: un elemento (trivial)

Prefijo en paralelo

- Algoritmo
  - Dividir entrada por la mitad, resolver recursivamente
    - Se obtienen valores para *PR*(1,*k*) y *PR*(*n*/2 + 1,*n*/2 + *k*)
  - Los valores de la primera mitad se utilizan directamente
  - Los valores *PR*(1,*m*) faltantes (entre n/2+1 y n) se obtienen calculando (por asociatividad)

$$PR\left(1,\frac{n}{2}\right) \bullet PR\left(\frac{n}{2}+1,m\right)$$

Prefijo en paralelo

- Complejidad:
  - *n* procesadores
  - Se asigna la mitad a cada subproblema
  - Combinar las soluciones toma n/2 pasos, que se pueden realizar en paralelo
    - Se debe accesar $x_{n/2}$ concurrentemente,pero se escribe en posiciones distintas (CREW)

$$T(n,n) = O(\log n), \;\; E(n,n) = O\left(\frac{1}{\log n}\right)$$

Prefijo en paralelo

- Trabajo total:

$$W(n) = 2W\left(\frac{n}{2}\right) + \frac{n}{2} = O(n \log n)$$

- No podemos utilizar el lema de Brent para mejorar la eficiencia
- Problema de este algoritmo: segundo llamado recursivo

Prefijo en paralelo

- ¿Cómo mejorar la eficiencia?
  - Truco: usar la misma hipótesis de inducción, pero dividir la entrada en una forma distinta
  - Suponemos nuevamente que n es potencia de 2 y disponemos de n procesadores
- Sea E el grupo de los x_i con i par
  - Si se encuentran los prefijos para E, los prefijos restantes se calculan como $PR(1,2i) \bullet x_{2i+1}$

Prefijo en paralelo

- Algoritmo EREW
  - Calcular (en paralelo) $x_{2i-1} \bullet x_{2i}$, guardando el resultado en x_{2i}
  - Resolver el problema para E (tamaño $n/2$) por inducción
    - Resultado es correcto porque ya incluye producto con x_{2i-1}
  - Resto de los prefijos se calcula en paralelo con un paso extra

Prefijo en paralelo

- Complejidad

$$T(n,n) = T\left(\frac{n}{2}\right) + O(1) = O(\log n)$$

- Eficiencia

$$E(n,n) = \frac{T(n,1)}{p \cdot T(n,p)} = O\left(\frac{1}{\log n}\right)$$

- Igual que para el algoritmo CREW, pero…

Prefijo en paralelo

- Trabajo total:

$$W(n) = W\left(\frac{n}{2}\right) + n - 1, W(2) = 1 \Rightarrow W(n) = O(n)$$

- Ahora si podemos utilizar el lema de Brent: mantener complejidad temporal pero utilizando sólo *n*/log *n* procesadores

$$T\left(n, \frac{n}{\log n}\right) = O(\log n), \;\; E\left(n, \frac{n}{\log n}\right) = O(1)$$

Rango en listas enlazadas

- El *rango* de un elemento en una lista enlazada se define como la distancia del elemento al final de la lista
  - Primer elemento tiene rango *n*
  - Segundo elemento tiene rango *n*-1
  - Etc.
- Problema: Dada una lista enlazada con *n* elementos almacenados en un arreglo *A*, calcular el rango para cada elemento de la lista

Rango en listas enlazadas

- Solución secuencial:
  - Recorrer la lista
  - Anotar rango de cada elemento visitado
- Método a usar: *doubling*
- Se ocuparan *n* procesadores
- Al comienzo, cada procesador conoce sólo al vecino derecho de su elemento correspondiente

Rango en listas enlazadas

- Algoritmo CREW:
  - En el primer paso, cada procesador encuentra al vecino de su vecino
  - Después del primer paso, cada procesador conoce al elemento a distancia 2 de su elemento
  - En general:
    - En el *i*-ésimo paso, cada procesador conoce el elemento a distancia *k* de su elemento
    - Al final de este paso, cada procesador conoce al elemento a distancia 2*k* de su elemento

Rango en listas enlazadas

- El proceso continúa hasta llegar al final de la lista
- Sea $N[i]$ el elemento más a la derecha de la lista que conoce el procesador P_i
  - Inicialmente, $N[i]$ es el vecino derecho de P_i, excepto por el último de la lista que es *null*
- En cada paso, P_i actualiza $N[i]$ por $N[N[i]]$, hasta que se alcanza el final de la lista

Rango en listas enlazadas

- Sea $R[i]$ el rango de i
- Inicialmente, $R[i]=0$ excepto para el último, que se inicializa en 1 (se detecta por su referencia *null*)
- Cuando un procesador encuentra un vecino con rango distinto de 0, puede calcular su propio rango

Rango en listas enlazadas

- Inicialmente, sólo 1 elemento conoce su rango
- Después del primer paso (“doblada”), el elemento de rango 2 encuentra su rango
- Después del segundo paso, los elementos de rango 3 y 4 encuentran su rango
- En general: Si P_i encuentra que $N[i]$ apunta a un elemento con rango mayor que 0 después de d “dobladas”, su rango es $2^{d-1}+R[N[i]]$

Rango en listas enlazadas

- Complejidad:
  - Proceso de *doubling* asegura que cada procesador alcanzará el final de la lista a lo más en log(*n*) pasos

$$T(n,p) = O(\log n)$$

- Eficiencia:

$$E(n,p) = O\left(\frac{1}{\log n}\right)$$

Rango en listas enlazadas

- Trabajo total:

$$W(n) = O(n \log n)$$

- Algoritmo requiere modificaciones mayores para mejorar su eficiencia
- Se puede convertir fácilmente en un algoritmo EREW

Odd-even sort

- n procesadores: P_1, …, P_n
- Entrada de tamaño n: x_1, …, x_n
- Cada procesador mantiene un valor
- Procesadores están conectados linealmente
  - P_i está conectado con P_{i+1}, i=1,…,n-1
- Problema: ordena la entrada

Odd-even sort

- La única comparación e intercambio posible es entre procesadores vecinos
  - En el peor caso, el algoritmo tomará *n*-1 pasos
- Idea del algoritmo:
  - Cada procesador compara su número con el de su vecino e intercambia los valores si están en el orden incorrecto
  - Luego lo hace con su segundo vecino
  - Proceso continúa hasta que la entrada esté ordenada

Odd-even sort

- Pasos están divididos en pasos impares y pasos pares (*odd-even transposition sort*)
  - Paso impar: los procesadores impares comparan su número con su vecino derecho
  - Paso par: los procesadores pares comparan su número con su vecino derecho
- De esta forma, los procesadores están sincronizados y cada comparación involucra los procesadores correctos

Odd-even sort

- Si un procesador no tiene el vecino correspondiente (e.g., primer procesador en el paso par), no hace nada
- Término anticipado es difícil de detectar
  - El algoritmo realiza un número de pasos equivalente al peor caso para garantizar su ejecución correcta $(\lceil n/2 \rceil)$
- Se puede probar por inducción que este algoritmo es correcto

Odd-even sort

- Complejidad y trabajo total:

$$T(n,n) = O(n), \;\; W(n) = O(n^2)$$

- Eficiencia:

$$E(n,n) = \frac{n \log n}{n \cdot n} = O\left(\frac{\log n}{n}\right)$$

Odd-even mergesort

- En un algoritmo paralelo, los pasos deben intentar hacerse lo más independientes posible
- Considere *mergesort*:
  - Las dos llamadas recursivas son independientes
    - Pueden realizarse en paralelo
  - La mezcla se realiza en forma secuencial
- Si se pudiera paralelizar la mezcla, se podría paralelizar *mergesort*

Odd-even mergesort

- Algoritmo de mezcla basado en *divide and conquer*
- n potencia de 2 (por simplicidad)
- Sea $a_1, \ldots, a_n$ y $b_1, \ldots, b_n$ dos secuencias ordenadas a mezclar
- Sea $x_1,\ldots,x_{2n}$ la secuencia final mezclada
- Idea: mezclar partes disjuntas de las secuencias en paralelo, para que la mezcla final sea fácil de calcular

Odd-even mergesort

- Procedimiento de mezcla:
  - Se dividen ambas secuencias en dos partes
    - Elementos pares
    - Elementos impares
  - Cada parte se mezcla con su correspondiente de la otra secuencia
  - Se realiza una mezcla final

Odd-even mergesort

- Sea $o_1, \ldots, o_n$ la mezcla de los elementos impares, y $e_1, \ldots, e_n$ la mezcla de los elementos pares
- Claramente $x_1 = o_1$ y $x_{2n} = e_n$
- ¿Resto de la mezcla?
  - Teorema: Para todo i=1,…,n-1 se tiene que

$$x_{2i} = \min\{o_{i+1}, e_i\}, \; x_{2i+1} = \max\{o_{i+1}, e_i\}$$

Odd-even mergesort

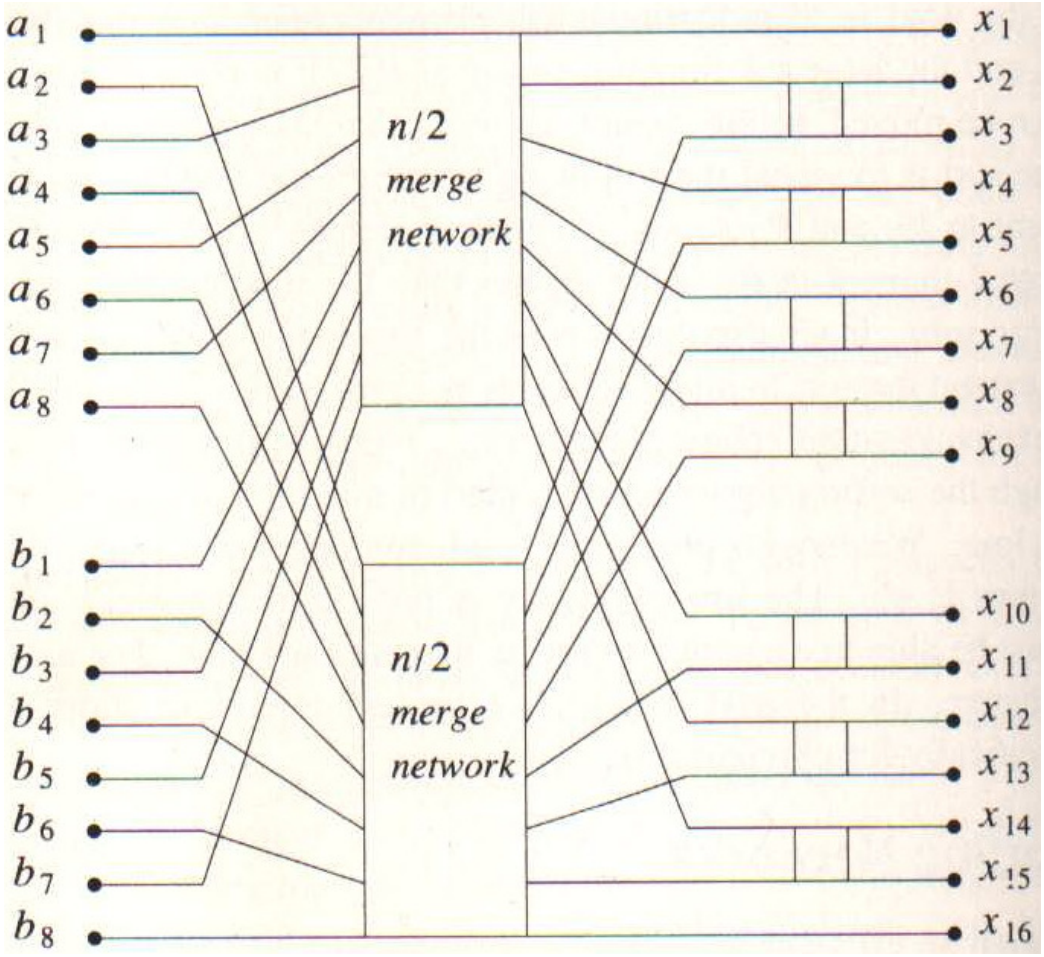
$$
\begin{array}{rcl}
x_1 & = & o_1 \\
x_{2n} & = & e_n \\
 & & \\
e_i & \geq & i \text{ elementos pares} \\
 & \geq & 2i \text{ elementos} \\
 & \geq & x_{2i} \\
 & & \\
o_{i+1} & \geq & i+1 \text{ elementos impares} \\
 & \geq & 2i \text{ elementos} \\
 & \geq & x_{2i}
\end{array}
$$

Odd-even mergesort

$$
\begin{gathered}
o_1 \to x_1 \\
o_2,\ e_1 \to x_2,\ x_3 \\
o_3,\ e_2 \to x_4,\ x_5 \\
o_4,\ e_3 \to x_6,\ x_7 \\
\dots \\
o_{i+1},\ e_i \to x_{2i},\ x_{2i+1} \\
\dots \\
o_n,\ e_{n-1} \to x_{2n-2},\ x_{2n-1} \\
e_n \to x_{2n}
\end{gathered}
$$

$$
\Rightarrow x_{2i} = \min\{e_i, o_{i+1}\},\ x_{2i+1} = \max\{e_i, o_{i+1}\}
$$

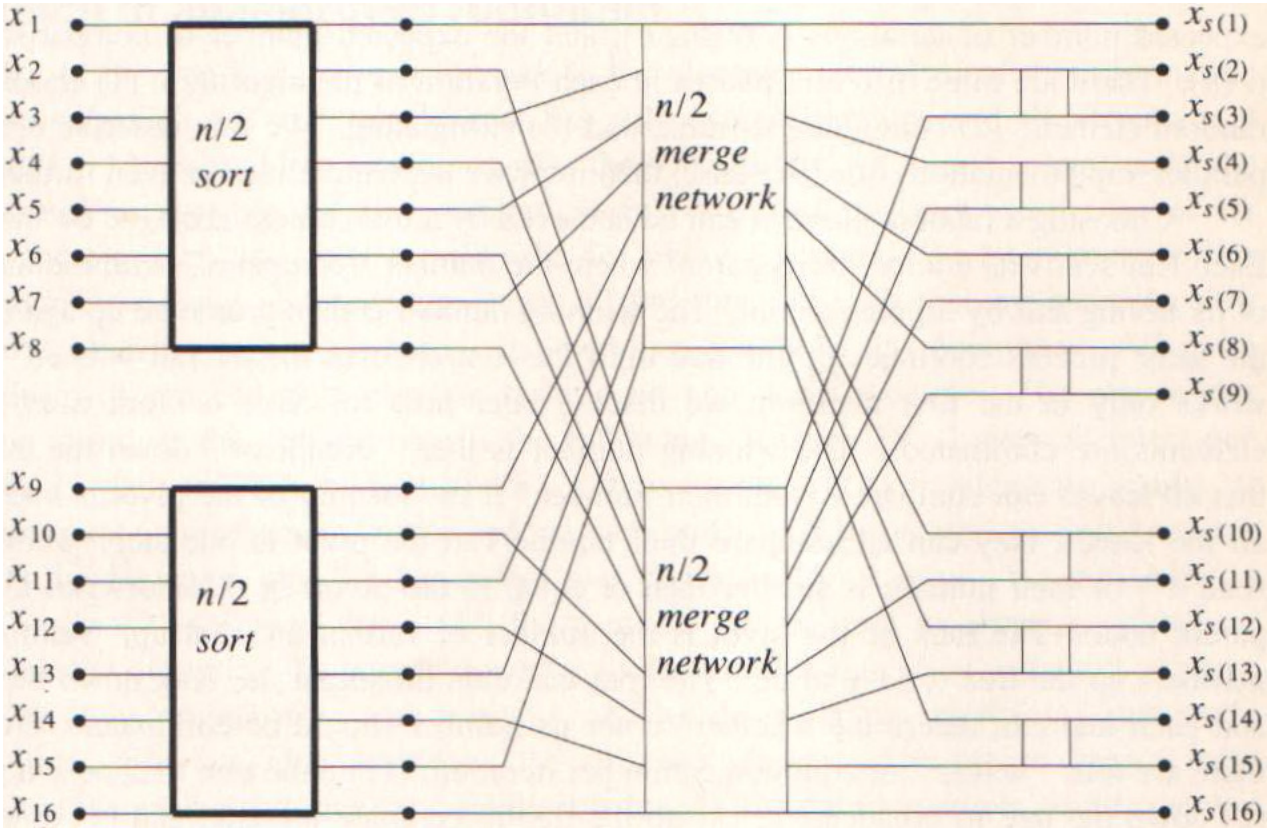
Odd-even mergesort



Note que mezcla final se puede obtener en un solo paso en paralelo. El resto se resuelve por inducción.

Odd-even mergesort

- Algoritmo paralelo:



Odd-even mergesort

- Complejidad:
  - Mezcla:
    - Tiempo: $T(n) = 1 + T(n/2) = O(\log n)$
    - # procs.: $P(n) = n + 2P(n/2) = O(n \log n)$
    - Eficiencia: $E(n) = \dfrac{n}{n \log n \cdot \log n} = O\left(\dfrac{1}{\log^2 n}\right)$

Odd-even mergesort

- Complejidad:
  - Ordenar:
    - Tiempo: $T(n) = T(n/2) + \log n = O(\log^2 n)$
    - # procs.:

$$P(n) = 2P(n/2) + n \log n = O(n \log^2 n)$$

    - Eficiencia:

$$E(n) = \frac{n \log n}{n \log^2 n \cdot \log^2 n} = O\left(\frac{1}{\log^3 n}\right)$$

Simple mergesort

- Método eficiente de ordenamiento en paralelo
- Lo estudiaremos en 4 partes
  - 1) Búsqueda en paralelo
  - 2) Rango de secuencias cortas
  - 3) Algoritmo de mezcla rápida
  - 4) Simple mergesort
- Existe algoritmo de mezcla en paralelo que toma tiempo $O(\log n)$ para n elementos

Simple mergesort

- 1) Búsqueda en paralelo
  - Entrada: x_1, …, x_n (ordenados y distintos)
  - Dado *y* encontrar índice *i* tal que

$$x_i \leq y < x_{i+1},\ x_0 = -\infty,\ x_{n+1} = \infty$$

  - Solución secuencial óptima: búsqueda binaria (*O*(log *n*))

Simple mergesort

- Extensión de la búsqueda binaria al caso paralelo:
  - Dividir X en p+1 segmentos de tamaños (casi) iguales
  - Comparar y con p elementos de X (definidos por los segmentos)
    - Salida de esta comparación:
      - Elemento x_i tal que $x_i=y$, o
      - Segmento que contiene a y
  - Repetir hasta encontrar y, o tamaño de segmento $< p$ elementos

Simple mergesort

- Complejidad:

$$T(n,p) = O\left(\frac{\log(n+1)}{\log(p+1)}\right)$$

Simple mergesort

- 2) Rango de secuencias cortas en una secuencia ordenada
  - Rango de un elemento *x* en *X*: número de elementos de *X* que son menores o iguales a *x*
    - Si *X* está ordenado, el rango permite determinar el predecesor de un elemento *x* arbitrario
  - Sea *X* una secuencia ordenada con *n* elementos e *Y* una secuencia arbitraria de tamaño $m=O(n^s)$, $0 < s < 1$
  - Algoritmo de búsqueda paralela se puede usar para calcular el rango de cada *y* en *X*

Simple mergesort

- Número de procesadores:

$$p = \lfloor n/m \rfloor$$

- Cada elemento de *Y* puede conocer su rango en *X* en tiempo

$$O\left(\frac{\log(n+1)}{\log(p+1)}\right) = O(1)$$

Simple mergesort

- 3) Algoritmo de mezcla rápida
  - Mezclar dos secuencias ordenadas *A* y *B* de largo *n* y *m* respectivamente
  - Supuesto: todos los elementos son distintos
  - Primero calcular el rango de *B* en *A* (*rank*(*B*:*A*))
  - Ejemplo:

$$A = \{-5, 0, 3, 4, 17, 18, 24, 28\},\ B = \{1, 2, 15, 21\}$$

$$rank(B : A) = \{2, 2, 4, 6\}$$

Simple mergesort

- Algoritmo para calcular *rank*(*B*:*A*)
  - Estrategia de particionamiento:
    - Calcular rango de un conjunto de $\sqrt{m}$ elementos de *B*, que particionen *B* en bloques de (casi) el mismo tamaño
    - Los rangos calculados inducen una partición de *A* tal que sus bloques deben encajar entre 2 de los objetos elegidos de *B*
  - El problema se reduce a calcular el rango de los elementos para cada bloque de *B* (excluyendo los ya calculados) en su bloque correspondiente de *A*

Simple mergesort

- Lema: *rank*(*B*:*A*) toma tiempo *O*(log log *m*)
- Sketch de la demostración:
  - $\sqrt{m}$ llamadas a búsqueda paralela con tiempo

$$O\left(\frac{\log(n+1)}{\log(\sqrt{n}+1)}\right) = O(1),\ p = \sqrt{n}$$

  - Llamados recursivos toman tiempo

$$T(n,m) \leq \max_i \left\{T(n_i, \sqrt{m})\right\} + O(1) = O(\log\log m)$$

Simple mergesort

- Mezcla se realiza calculando *rank*(*A*:*B*) y *rank*(*B*:*A*) en tiempo *O*(log log *n*)
  - Particionar *B* en segmentos de tamaño log *n* y determinar sus segmentos correspondientes en *A* (tiempo *O*(1) si *rank*(*B*:*A*) y *rank*(*A*:*B*) conocidos)
  - Si $|A_i|$>log *n* se particiona en segmentos de tamaño log *n* (tiempo *O*(log log *n*))
  - Se mezclan los segmentos (tiempo *O*(log log *n*))
- Complejidad total de la mezcla para conjuntos *A* y *B* de tamaño *n*: $O(\log \log n)$

Simple mergesort

- 4) Simple mergesort
  - Se parte con *n* conjuntos de 1 elemento
  - Se mezclan de a pares los conjuntos, utilizando el algoritmo de mezcla $O(\log \log n)$
  - Se itera hasta tener un conjunto ordenado de tamaño *n*

Simple mergesort

- Complejidad:
  - log(*n*) pasos
  - En cada iteración se procesan *n* elementos, lo cual toma tiempo O(log log *n*)
  - Tiempo total: O(log *n* log log *n*)
- Existe otro algoritmo más eficiente (óptimo)
  - Pipelined mergesort: O(log *n*)

ALGORITMOS ALEATORIZADOS Y PROBABILÍSTICOS

Motivación

- Algoritmos determinísticos vs aleatorizados
  - Un algoritmo determinístico siempre realiza las mismas operaciones frente a la misma entrada
    - Un algoritmo aleatorizado NO
  - Un algoritmo determinístico nunca da una respuesta errada
    - Un algoritmo aleatorizado PODRIA equivocarse
  - Un algoritmo determinístico siempre termina en un tiempo acotado (peor caso)
    - Un algoritmo aleatorizado PODRIA no terminar nunca

Algoritmos aleatorizados

- Algoritmo aleatorizado
  - Toma decisiones aleatorias
  - Esto sirve para independizarse de los supuestos sobre la entrada
  - Se analiza su costo esperado (vale para cualquier entrada)

Algoritmos aleatorizados

- Ejemplo: búsqueda en arreglo desordenado

```
Randomized-Search(x, A) // A[1,n]
1. r = random{1,2}
2. if r == 1
3.     Exact-Search(x, A, 1 -> n)
4. else // r == 2
5.     Exact-Search(x, A, n -> 1)
```

- Costo esperado de búsqueda de un elemento en la i-ésima posición

$$E(T(n)) = T(n) = \frac{1}{2} \cdot i + \frac{1}{2} \cdot (n - i) = \frac{n}{2}$$

Algoritmos Montecarlo

- Tienen una cierta probabilidad de equivocarse

$$Pr(error) < \varepsilon$$

- Ejemplo: Escoger un buen alumno
  - Curso con n alumnos
  - Se desea escoger un “buen alumno” (alumno con promedio de notas por sobre la media)
    - Algoritmo determinístico debe comparar n/2 + 1 alumnos (si compara menos, adversario sólo le presenta alumnos bajo la media)

Algoritmos Montecarlo

- Ejemplo: Escoger un buen alumno
  - Algoritmo Montecarlo:
    - Escoger un alumno aleatoriamente: Pr(error) = ½
      - Tiempo: O(1)
    - Escoger k alumnos aleatoriamente y quedarse con el mejor: Pr(error) = $1/2^k$
      - Tiempo: O(k)

Algoritmos Montecarlo

- Tipos de algoritmos Montecarlo:
  - One-sided error:

$$\begin{aligned} Pr(accept|negative) &< \varepsilon \\ Pr(refuse|negative) &> 1-\varepsilon \\ Pr(accept|positive) &= 1 \\ Pr(refuse|positive) &= 0 \end{aligned}$$

  - Two-sided error:

$$\begin{aligned} Pr(accept|negative) &< \varepsilon \\ Pr(refuse|negative) &> 1-\varepsilon \\ Pr(accept|positive) &> 1-\varepsilon \\ Pr(refuse|positive) &< \varepsilon \end{aligned}$$

Algoritmos Montecarlo

- Ejemplo: verificar multiplicación de matrices
  - Sean A, B, C matrices de n x n
  - Se desea chequear si A*B = C
  - La verificación toma tiempo $O(n^3)$ usando el algoritmo estándar (determinístico)
    - Multiplicar A*B (tiempo $O(n^3)$)
    - Verificar que resultado es igual a C (tiempo $O(n^2)$)

Algoritmos Montecarlo

- Algoritmo Montecarlo para la verificación:

```
1. for i = 1 to k
2.   Generar vector aleatorio de N x 1 con entradas en {-1, 1}
3.   if A*(Bx) != Cx
4.     return False // A*B != C
5. return True // A*B = C con alta probabilidad
```

  - Complejidad: $\Theta(n^2)$. ¿Probabilidad de error?
    - Sea D = A*B. Si D[i,j] != C[i,j] para algún i,j:
      - Dx = Cx implica Dx' != Cx'
        - x' es el vector donde se cambia x_j por $-x_j$
      - Por lo tanto, en cada iteración se tiene que Pr(error) <= ½
      - Se realizan k iteraciones, implica Pr(error) <= $1/2^k$

Algoritmos Las Vegas

- Siempre retornan la respuesta correcta, pero su tiempo de ejecución no está necesariamente garantizado
- Ejemplo: Randomized Quicksort
- Ejemplo: Escoger alumno con promedio >= 6
  - Escoger alumno en forma aleatoria
  - Mientras alumno escogido tenga promedio < 6
    - Escoger alumno en forma aleatoria
  - Retornar alumno escogido

Algoritmos Las Vegas

- Ejemplo: algoritmo para colorear conjuntos
  - Sean k conjuntos C_1, ..., C_k con r elementos cada uno (no disjuntos), con k <= 2^{r-2}
  - Se pide colorear los elementos de color rojo o azul, tal que ningún C_i sea homogéneo
  - Algoritmo:

```
1. while True
2.   Colorear los elementos aleatoriamente
3.   if ningun Ci es homogeneo
4.     break
```

Algoritmos Las Vegas

- Ejemplo: algoritmo para colorear conjuntos
  - Complejidad temporal:
    - ¿Cuántas veces es necesario repetir el ciclo para obtener una respuesta?

Pr(Ci homogeneo) = Pr(todos los elementos de Ci rojos) + Pr(todos los elementos de Ci azules) = $1/2^r + 1/2^r = 1/2^{r-1}$

  - => Pr(algún Ci homogeneo) = $k/2^{r-1} <= 1/2$ (ya que $k<=2^{r-2}$)
    - Esto implica que en promedio el ciclo se ejecuta 2 veces => O(k*r)

Test de Miller-Rabin

- Problema: dado n, determinar si es primo
- Test de Miller-Rabin
  - Sea n>2 primo, n-1 es par y se puede escribir

$$n = 2^s \cdot d, \;\; s, d \text{ enteros positivos}, \;\; d \text{ impar}$$

  - Se puede demostrar que para 1 <= a <= n-1

$$a^d \equiv 1 \mod n, \text{ o que } a^{2^r \cdot d} \equiv -1 \mod n \text{ para algun } 0 \leq r \leq s-1$$

  - El test de Miller-Rabin verifica lo contrario
    - n no es primo si se puede encontrar *a* (el testigo) tal que

$$a^d \not\equiv 1 \mod n, \text{ y } a^{2^r \cdot d} \not\equiv -1 \mod n \text{ para todo } 0 \leq r \leq s-1$$

Test de Miller-Rabin

- Ejemplo (de Wikipedia)
  - ¿n = 221 primo? n–1 = 220 = $2^2 \cdot 55$, s = 2 y d = 55
    - Escoger testigo a < n, por ejemplo a = 174, se calcula:
      - $a^{2^0} \cdot d \bmod n = 17455 \bmod 221 = 47 \neq 1, n - 1$
      - $a^{2^1} \cdot d \bmod n = 174110 \bmod 221 = 220 = n - 1$
    - Dado que 220 ≡ –1 mod n, 221 es primo o 174 es un "strong liar" para 221. Usando otro a = 137:
      - $a^{2^0} \cdot d \bmod n = 13755 \bmod 221 = 188 \neq 1, n - 1$
      - $a^{2^1} \cdot d \bmod n = 137110 \bmod 221 = 205 \neq n - 1$
    - 137 es un testigo que 221 es compuesto (no primo), por lo que 174 fue en efecto un "strong liar"

Test de Miller-Rabin

- Seudocódigo (de Wikipedia)

```
Miller-Rabin(n, k)
1. write n − 1 as 2^s·d, d odd, by factoring powers of 2 from n − 1
2. WitnessLoop: repeat k times:
3.    pick a random integer a in the range [2, n − 2]
4.    x ← a^d mod n
5.    if x = 1 or x = n − 1 then do next WitnessLoop
6.    repeat s − 1 times:
7.       x ← x^2 mod n
8.       if x = n − 1 then do next WitnessLoop
9.    return composite
10.return probably prime
```

Test de Miller-Rabin

- Complejidad temporal: $O(k \log^3 n)$
  - Ciclo de línea 2 toma tiempo $O(\log^3 n)$
- Se puede demostrar que un número compuesto pasa el test para a lo más un cuarto de los testigos (valores *a*) posibles
  - Se realizan k tests independientes
  - Probabilidad de error <= $1/4^k$

Hashing universal y hashing perfecto

- Hashing
  - Universo U de valores
  - Conjunto $K \subset U$ de llaves, $|K| = n$
  - Función de hash $h: U \rightarrow \{0,\ldots,m-1\}$ (mapeo)
  - Tabla $T = [0,m-1]$ ($m = \Theta(|K|)$)
  - Operaciones de inserción y búsqueda (promedio)
    - $\Theta(1+\alpha)$ usando encadenamiento ($\alpha = n/m$ "factor de carga")
    - $O(1/(1-\alpha))$ usando direccionamiento abierto

Hashing universal y hashing perfecto

- Problema con encadenamiento y función de hash fija
  - Adversario escoge n valores tal que la función de hash siempre retorna el mismo casillero
  - Tiempo promedio de búsqueda $\Theta(n)$
  - Cualquier función de hash fija es vulnerable a este problema

Hashing universal y hashing perfecto

- Solución
  - Escoger función de hash aleatoriamente, de forma que sea independiente de las llaves
  - Esto permitirá tener con alta probabilidad un buen tiempo promedio, no importando cómo el adversario escoja las llaves
  - Esto se denomina hashing universal

Hashing universal y hashing perfecto

- Idea principal detrás de hashing universal
  - Escoger la función de hash aleatoriamente desde una clase de funciones adecuada
  - Aleatorización implica
    - Garantizar que ninguna entrada en particular terminará siempre en el peor caso
    - El algoritmo podría funcionar en forma distinta en cada ejecución, incluso para la misma entrada
    - Esto permite garantizar que el caso esperado será bueno para cualquier entrada
    - La probabilidad de caer en el peor caso es baja

Hashing universal y hashing perfecto

- Idea principal detrás de hashing universal
  - Sea H una colección finita de funciones de hash h: U → {0,…,m-1}
  - Esta colección se dice universal si para cualquier par de llaves distintas k,l ϵ U, el número de funciones de hash h para las cuales h(k) = h(l) es a lo más |H|/m
    - Es decir, la probabilidad de una colisión es menor o igual a 1/m si la función se elige en forma aleatoria de H

Hashing universal y hashing perfecto

- Teorema
  - Sea h ϵ H una función de hash escogida en forma aleatoria, que se usa para mapear n llaves en una tabla T de tamaño m (usando encadenamiento)
    - Si la llave k no está en la tabla, el largo esperado de la lista $n_{h(k)}$ en donde k es mapeada es a lo más α
    - Si la llave k está en la tabla, el largo esperado de la lista $n_{h(k)}$ en donde k es mapeada es a lo más $1+\alpha$

Hashing universal y hashing perfecto

- Teorema: Demostración
  - Notar que la esperanza se calcula sobre la elección de h, y no depende de ningún supuesto sobre la distribución de las llaves
  - Sea la variable indicadora X_{kl} (1 si la condición es cierta, 0 sino)

$$X_{kl} = I\{h(k) = h(l)\}$$

Hashing universal y hashing perfecto

- Teorema: Demostración
  - Por definición, cualquier par de llaves colisiona con probabilidad a lo más de 1/m

$$Pr\{h(k) = h(l)\} \leq \frac{1}{m} \Rightarrow E[X_{kl}] \leq \frac{1}{m}$$

Hashing universal y hashing perfecto

- Teorema: Demostración
  - Se define para cada llave k una variable aleatoria Y_k, que es el número de llaves distintas de k que se mapean al mismo casillero que k

$$Y_k = \sum_{l \in T, l \neq k} X_{kl}$$

  - Entonces:

$$E[Y_k] = E\left[\sum_{l \in T, l \neq k} X_{kl}\right] = \sum_{l \in T, l \neq k} E[X_{kl}] \leq \sum_{l \in T, l \neq k} \frac{1}{m}$$

Hashing universal y hashing perfecto

- Teorema: Demostración
  - Si k ∉ T, entonces $n_{h(k)} = Y_k$ y {l:l ∈ T y l ≠ k} = n

$$E\left[n_{h(k)}\right] = E\left[Y_k\right] \leq \frac{n}{m} = \alpha$$

  - Si k ∈ T, entonces k aparece en la lista T[h(k)] y Y_k no incluye k, $n_{h(k)} = Y_k + 1$ y {l:l ∈ T y l ≠ k} = n-1

$$E\left[n_{h(k)}\right] = E\left[Y_k\right] + 1 \leq \frac{(n-1)}{m} + 1 = 1 + \alpha - \frac{1}{m} < 1 + \alpha$$

Hashing universal y hashing perfecto

- Teorema implica que ahora es imposible para el adversario escoger una secuencia de operaciones que fuerce el peor caso
  - Esto se logra por la aleatorización de la elección de la función de hash h
  - Con esto, se puede garantizar que cada secuencia de operaciones se puede procesar con un buen tiempo esperado de ejecución

Hashing universal y hashing perfecto

- Corolario
  - Usando hashing universal y encadenamiento en una tabla de tamaño m, toma tiempo esperado $\Theta(n)$ procesar una secuencia de n operaciones de inserción, búsqueda y borrado que contengan O(m) operaciones de inserción
  - Demostración:
    - Se tiene que n = O(m), y por lo tanto α = O(1). Insertar y borrar toman tiempo constante, y por el Teorema la búsqueda toma tiempo esperado O(1). Por linealidad de la esperanza, QED

Hashing universal y hashing perfecto

- Clase de funciones de hash universales
  - Se escoge un primo p>m suficientemente grande
    - Toda llave k está en el rango [0,p-1]
  - Sean
    - $Z_p = \{0,1,\ldots,p-1\}$
    - $Z^*_p = \{1,\ldots,p-1\}$
  - Se define $h_{a,b}$ para a en Z^*_p, b en Z_p, como

$$h_{a,b} = ((ak + b) \mod p) \mod m$$

Hashing universal y hashing perfecto

- La clase de funciones de hash se denota

$$H_{p,m} = \{h_{a,b} : a \in Z_p^*,\ b \in Z_p\}$$

  - Cada función de hash mapea Z_p a Z_m
  - Esta clase de funciones tiene la propiedad que el tamaño m de salida es arbitrario (no necesariamente primo)
  - Hay p(p-1) funciones de hash distintas en $H_{p,m}$
- Se puede demostrar que la clase $H_{p,m}$ es universal

Hashing universal y hashing perfecto

- Hashing perfecto
  - Si el conjunto de llaves es estático, se puede obtener una muy buena cota del peor caso
  - Esta técnica se llama hashing perfecto
    - El número de accesos de memoria para realizar una búsqueda es en el peor caso O(1)
  - Idea para implementar hashing perfecto
    - Usar un esquema de dos niveles con hashing universal
      - Primer nivel: igual que hashing encadenado
      - Segundo nivel: tabla de hash secundaria en casillero j, con función de hash h_j

Hashing universal y hashing perfecto

- Es necesario garantizar que no hay colisiones en el segundo nivel
  - Esto requiere que el tamaño m_j de la tabla S_j sea el cuadrado del número n_j de llaves que caen en el casillero j
  - Mostraremos que el espacio utilizado sigue siendo O(n)

Hashing universal y hashing perfecto

- Usaremos la clase de funciones de hash universal estudiadas
  - En el primer nivel se escoge h de $H_{p,m}$
  - En el segundo nivel se escoge h_j de H_{p,m_j}
- Ahora se necesita mostrar que
  - No hay colisiones en las tablas secundarias
  - La cantidad esperada de espacio a utilizar (tabla del primer nivel y tablas del segundo nivel) es O(n)

Hashing universal y hashing perfecto

- Teorema:
  - Si se almacenan n llaves en una tabla de tamaño $m = n^2$ usando una función de hash universal, la probabilidad de colisión es menor que ½
- Demostración: Hay $\binom{n}{2}$ pares que pueden colisionar con probabilidad 1/m
  - Sea X variable aleatoria que cuenta las colisiones

$$E[X] = \binom{n}{2} \cdot \frac{1}{m} = \binom{n}{2} \cdot \frac{1}{n^2} = \frac{n^2 - n}{2} \cdot \frac{1}{n^2} < \frac{1}{2}$$

  - Desigualdad de Markov:

$$Pr\{X \geq t\} \leq E[X]/t, \; t = 1 \text{ en este caso}$$

Hashing universal y hashing perfecto

- El Teorema indica que es más probable que h no produzca colisiones
  - Dado que el conjunto de llaves es estático, hay que probar funciones h hasta encontrar una que no produzca colisiones
  - En promedio, basta con probar pocas (dos) veces
- Bastaría entonces con usar una sola tabla de tamaño $m = n^2$, pero esto es excesivo
  - Por eso se usan dos niveles

Hashing universal y hashing perfecto

- Espacio utilizado con dos niveles
  - Primer nivel: m = n, O(n)
  - Segundo nivel: Si $n_j = 1$ basta con una celda, sino se necesitan $m_j = n_j^2$ celdas
- Teorema
  - Si se almacenan n llaves en una tabla con m = n celdas usando función de hash h universal:

$$E\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n,\ n_j \text{ es el numero de llaves mapeadas a } j$$

Hashing universal y hashing perfecto

- Teorema: demostración
  - Para todo entero a > 0 se cumple que $a^2 = a + 2\binom{a}{2}$
  - Se tiene:

$$
\begin{aligned}
E\left[\sum_{j=0}^{m-1} n_j^2\right] &= E\left[\sum_{j=0}^{m-1}\left(n_j + 2\binom{n_j}{2}\right)\right] \\
&= E\left[\sum_{j=0}^{m-1} n_j\right] + 2E\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right] \\
&= E[n] + 2E\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right] \\
&= n + 2E\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right]
\end{aligned}
$$

Hashing universal y hashing perfecto

- Teorema: demostración
  - El término $\sum_{j=0}^{m-1} \binom{n_j}{2}$ es el número total de colisiones
  - Por las propiedades de hashing universal, el valor esperado de esta suma es

$$\binom{n}{2}\frac{1}{m} = \frac{n(n-1)}{2m} = \frac{n-1}{2} \text{ dado que } n = m$$

  - Finalmente,

$$E\left[\sum_{j=0}^{m-1} n_j^2\right] \leq n + 2\frac{n-1}{2} = 2n - 1 < 2n$$

Hashing universal y hashing perfecto

- Corolario: la cantidad esperada de espacio necesario para el segundo nivel es < 2n
- Corolario: la probabilidad que el espacio total utilizado por las tablas del segundo nivel exceda 4n es menor que ½
  - Demostración: Usando desigualdad de Markov

$$Pr\left\{\sum_{j=0}^{m-1} m_j \geq 4n\right\} \leq \frac{E\left[\sum_{j=0}^{m-1} m_j\right]}{4n} < \frac{2n}{4n} = \frac{1}{2}$$

Hashing universal y hashing perfecto

- En resumen: basta con probar con unas pocas funciones de hash de la familia de funciones universales para que con alta probabilidad se encuentre alguna que utilice espacio razonable

ALGORITMOS APROXIMADOS

Motivación

- Muchos problemas prácticos importantes son NP-completos
  - Encontrar la solución óptima es “intratable”
- Si se debe enfrentar problema NP-completo:
  - Si la entrada es pequeña, encontrar la solución óptima puede ser factible
  - Aislar casos especiales que puedan resolverse en tiempo polinomial
  - Encontrar soluciones “casi óptimas”

Motivación

- Las soluciones “casi óptimas” pueden ser suficientemente buenas en la práctica
- Un algoritmo que retorna una respuesta “casi óptima” se denomina algoritmo aproximado
- Supondremos que queremos resolver problemas de optimización (maximización o minimización) en que cada solución tiene un costo positivo

Definiciones

- Un algoritmo tiene una razón de aproximación de ρ(n) si, para cualquier entrada de tamaño n, el costo C de la solución producida por el algoritmo se encuentra dentro de un factor ρ(n) del costo C* de la solución óptima

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$$

Definiciones

- Un algoritmo que alcanza una razón de aproximación de $\rho(n)$ se denomina $\rho(n)$-aproximado
- Esta definición sirve para problemas de minimización o maximización
  - Max: $0 < C \leq C^*$, razón C^*/C
  - Min: $0 < C^* \leq C$, razón C/C^*
  - Notar que $\rho(n)$ nunca es menor que 1
    - Un algoritmo 1-aproximado entrega soluciones óptimas

Definiciones

- Un esquema de aproximación para un problema de optimización es un algoritmo aproximado que recibe como parámetro $\varepsilon > 0$ y es $(1+\varepsilon)$-aproximado
  - Un esquema de aproximación de tiempo polinomial garantiza que para todo $\varepsilon > 0$ fijo, el esquema toma tiempo polinomial en el tamaño n de la entrada (e.g., $O(n^{2/\varepsilon})$)
  - El esquema es completamente polinomial si además es polinomial en $1/\varepsilon$ (e.g., $O((1/\varepsilon)^2 n^3)$)

Vertex-cover

- Sea G=(V,E) un grafo no dirigido
- Un vertex-cover (recubrimiento de vértices) es un subconjunto V' $\subseteq$ V tal que si (u,v) es una arista en G, entonces u ϵ V' o v ϵ V' (o ambos)
  - El tamaño de un recubrimiento es su número de nodos

Vertex-cover

- El problema de vertex-cover consiste en encontrar un recubrimiento de tamaño mínimo dado un grafo no dirigido
  - Vertex-cover óptimo
- El problema de decisión “grafo G tiene un vertex-cover de tamaño k” es NP-completo
  - No se conoce solución polinomial para resolver el problema (y es poco probable que exista alguna)

Vertex-cover

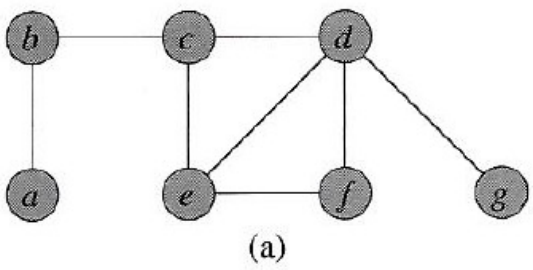
- Algoritmo aproximado para vertex-cover

```
Approx-Vertex-Cover(G)
1. C <- ϕ
2. E' <- E[G]
3. while E'≠ ϕ
4.     Sea (u,v) una arista arbitraria de E'
5.     C <- C U {u,v}
6.     Sacar de E' toda arista incidente en u o v
```

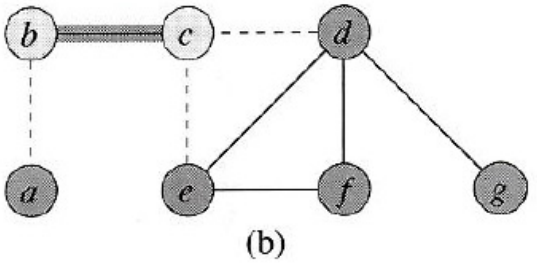
  - Notar que algoritmo toma tiempo polinomial O(V+E)

Vertex-cover

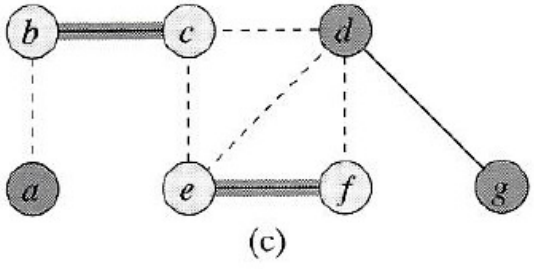
- Ejemplo:



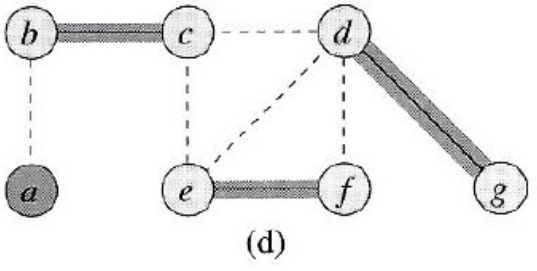
(a)



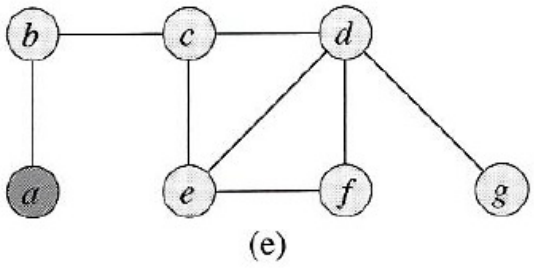
(b)



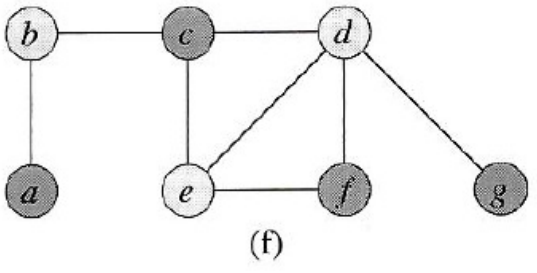
(c)



(d)



(e)



(f)

Vertex-cover

- Teorema: Approx-Vertex-Cover es una 2-aproximación de vertex-cover
- Demostración:
  - El conjunto de vértices retornado es un vertex-cover, dado que el algoritmo itera hasta que toda arista ha sido cubierta por algún vértice
  - Sea A el conjunto de aristas escogidas en línea 4
  - Para cubrir las aristas en A, cualquier vertex-cover (en particular el óptimo C*) debe incluir al menos uno de los vértices en cada arista de A

Vertex-cover

- Teorema: Approx-Vertex-Cover es una 2-aproximación de vertex-cover
- Demostración:
  - No hay dos aristas en A que compartan un vértice, ya que todas las aristas que comparten un vértice escogido en la línea 4 son descartadas en la línea 6
  - Por lo tanto, no hay dos aristas en A cubiertas por el mismo vértice de C*, lo que implica $|C^*| \geq |A|$

Vertex-cover

- Teorema: Approx-Vertex-Cover es una 2-aproximación de vertex-cover
- Demostración:
  - Cada ejecución de la línea 4 escoge una arista para la cual no hay ningún vértice incidente que ya esté en C, lo que implica |C| = 2|A|
  - Combinando ambas ecuaciones:

$$|C| = 2|A| \leq 2|C^*|$$

Vertex-cover

- Note que para la demostración no se requiere saber el tamaño óptimo de vertex-cover
  - Se utiliza una cota inferior al vertex-cover óptimo
  - Algoritmo aproximado es a lo más el doble de tamaño que la cota inferior
  - Por lo tanto, el algoritmo es una 2-aproximación del resultado óptimo

Vendedor viajero

- El problema del vendedor viajero:
  - Sea un grafo completo G = (V, E) con n vértices
  - Un vendedor desea hacer un tour (ciclo hamiltoniano) en el grafo
    - Visitar cada vértice exactamente una vez
    - Terminar el tour en el vértice de inicio
    - Costo entero c(i,j) asociado a viajar del nodo i al nodo j
    - Se desea encontrar tour de costo total mínimo
  - Problema de decisión “existe un tour de a lo más costo k” es NP-completo

Vendedor viajero

- Sea c(A) el costo de las aristas en A ⊆ E

$$c(A) = \sum_{(u,v)\in A} c(u,v)$$

- Primero se supondrá que el costo c cumple con la desigualdad triangular

$$c(u,w) \leq c(u,v) + c(v,w)$$

Vendedor viajero

- Algoritmo aproximado:
  - Se calculará un árbol cobertor mínimo (minimum spanning tree o MST) cuyo peso es una cota inferior al costo del tour óptimo
  - Se utilizará el árbol cobertor mínimo para crear un tour de costo menor a dos veces el costo del árbol cobertor mínimo (si es que se cumple la desigualdad triangular)
  - Con esto, se obtendrá un algoritmo que es una 2-aproximación del problema

Vendedor viajero

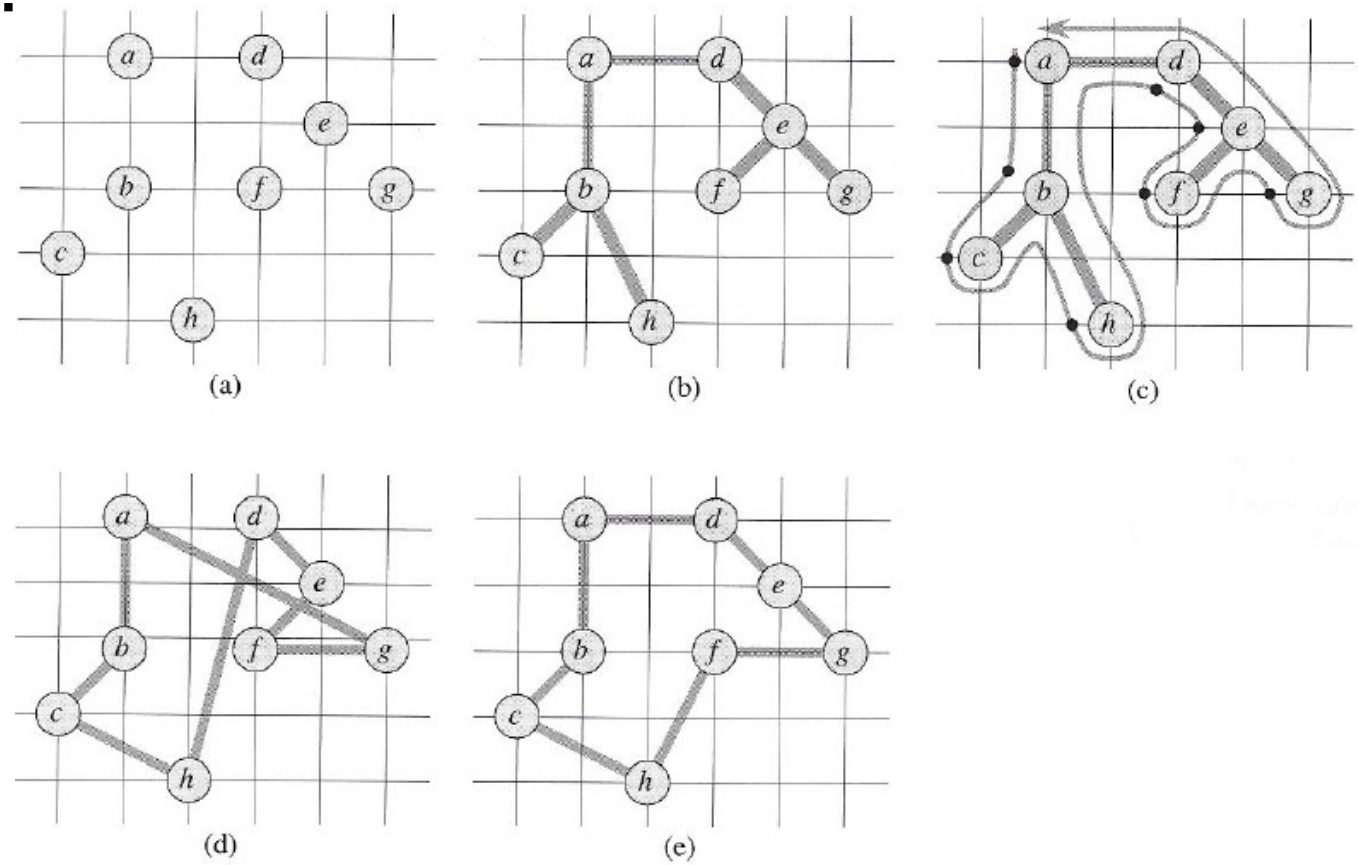
- Algoritmo aproximado:

```
Approx-TSP-Tour(G,c) // Traveling-salesman problem
1. Escoger un vértice r en V[G] como la raíz
2. Calcular un MST T de G con raíz r usando Prim
3. Sea L la lista de vértices visitadas en un
   recorrido en preorden de T
4. return ciclo hamiltoniano H que visita los
   vértices en el orden de L
```

  - Complejidad temporal es $\Theta(V^2)$

Vendedor viajero

- Ejemplo:



Vendedor viajero

- Teorema: Approx-TSP-Tour es una 2-aproximación del problema del vendedor viajero
- Demostración:
  - Sea H* un tour óptimo
  - Dado que se obtiene un árbol cobertor borrando alguna arista de H*, el MST es una cota inferior al costo del tour óptimo

$$c(T) \leq c(H^*)$$

Vendedor viajero

- Teorema: Approx-TSP-Tour es una 2-aproximación del problema del vendedor viajero
- Demostración:
  - Un recorrido completo W de T indica los vértices cuando son visitados por primera vez o cuando se vuelve a ellos luego de visitar su subárbol
  - Dado que W pasa por cada arista de T exactamente dos veces, se tiene que $c(W) = 2c(T)$

Vendedor viajero

- Teorema: Approx-TSP-Tour es una 2-aproximación del problema del vendedor viajero
- Demostración:
  - Lo anterior implica que $c(W) \leq 2c(H^*)$
  - Sin embargo, en general W no es un tour (visita vértices más de una vez)
  - Por la desigualdad triangular, se puede borrar una visita a un vértice de W sin aumentar el costo

Vendedor viajero

- Teorema: Approx-TSP-Tour es una 2-aproximación del problema del vendedor viajero
- Demostración:
  - Repitiendo el proceso, se eliminan de W todas las visitas a un vértice excepto la primera
  - Este orden es el mismo que se obtiene al recorrer T en preorden
  - Sea H el ciclo correspondiente a este recorrido en preorden

Vendedor viajero

- Teorema: Approx-TSP-Tour es una 2-aproximación del problema del vendedor viajero
- Demostración:
  - H es un ciclo hamiltoniano, dado que cada vértice se visita exactamente una vez, y H es el tour obtenido por Approx-TSP-Tour
  - Dado que H se obtiene de borrar vértices de W

$$c(H) \leq c(W) \leq 2c(H^*)$$

Vendedor viajero

- Teorema: si c no cumple con desigualdad triangular, no existe algoritmo polinomial que sea $\rho(n)$ – aproximado para el problema del vendedor viajero, salvo que P = NP
- Demostración por contradicción:
  - Se supondrá que existe algoritmo $\rho(n)$ – aproximado polinomial, y con éste se construirá un algoritmo polinomial para el problema de ciclo hamiltaniano en G (responde “si” si grafo G tiene un ciclo hamiltoniano), que es un problema NP-completo

Vendedor viajero

- Teorema: si c no cumple con desigualdad triangular, no existe algoritmo polinomial que sea $\rho(n)$ – aproximado para el problema del vendedor viajero, salvo que P = NP
- Demostración por contradicción:
  - Sea A algoritmo polinomial $\rho(n)$ – aproximado
    - Sin perder generalidad, $\rho(n)$ entero (se puede redondear)
  - Sea G = (V,E) una instancia del problema del ciclo hamiltoniano (|V| = n el número de vértices)
  - Se transforma G en una instancia del vendedor viajero

Vendedor viajero

- Teorema: si c no cumple con desigualdad triangular, no existe algoritmo polinomial que sea $\rho(n)$ – aproximado para el problema del vendedor viajero, salvo que P = NP
- Demostración por contradicción:
  - Sea G' = (V, E') el grafo completo en V
    - Costos de los vértices:
      - $c(u,v) = 1$ si $(u,v) \epsilon V$
      - $c(u,v) = \rho n + 1$ en caso contrario
    - Esto se puede calcular en tiempo polinomial en |V| y |E|

Vendedor viajero

- Teorema: si c no cumple con desigualdad triangular, no existe algoritmo polinomial que sea $\rho(n)$ – aproximado para el problema del vendedor viajero, salvo que P = NP
- Demostración por contradicción:
  - Problema del vendedor viajero en (G',c)
    - Si grafo original G tiene un ciclo hamiltoniano, éste será el tour óptimo con costo n = |V|
    - Si G no contiene un ciclo hamiltoniano, cualquier tour en G' debe usar alguna arista en E', por lo tanto el costo de un tour es de al menos $(\rho n + 1) + (n - 1) = \rho n + n > \rho n$

Vendedor viajero

- Teorema: si c no cumple con desigualdad triangular, no existe algoritmo polinomial que sea $\rho(n)$ – aproximado para el problema del vendedor viajero, salvo que P = NP
- Demostración por contradicción:
  - Si se utiliza algoritmo $\rho(n)$-aproximado A
    - Si G contiene ciclo hamiltoniano, A debe retornarlo (cualquier otra solución tiene costo > ρn)
    - Si no, A retorna necesariamente tour de costo > ρn
    - Se puede usar A para resolver ciclo hamiltoniano en tiempo polinomial. QED

Subset-sum

- Conjunto S ={x_1, …, x_n} de enteros positivos
- Valor t entero positivo
- Problema del subset-sum (decisión):
  - "Existe un subconjunto de S cuya suma es exactamente t"
  - Este problema es NP-completo
- Problema de optimización asociado:
  - Encontrar subconjunto de S cuya suma sea lo mayor posible pero no mayor que t

Subset-sum

- Algoritmo exacto (exponencial)
  - Idea: para todos los subconjuntos S' de S calcular su suma, quedarse con aquel cuya suma sea la más cercana a t (tiempo $O(2^n)$)
  - Estrategia de implementación
    - Procedimiento iterativo, en iteración i se calcula la suma de todos los subconjuntos de $\{x_1, \ldots, x_i\}$, usando como partida la suma de los subconjuntos $\{x_1, \ldots, x_{i-1}\}$
    - Si algún subconjunto particular excede el valor t, se descarta

Subset-sum

- Algoritmo exacto (exponencial)
  - Implementación
    - L_i: lista de las sumas de todos los subconjuntos $\{x_1, \ldots, x_i\}$ que no exceden t
      - Respuesta final es máximo valor de L_n
    - L + x: lista resultante de añadir x a todos los elementos de L
    - Merge-Lists(L, L'): retorna lista ordenada resultante de mezclar L con L', eliminando duplicados (tiempo O(|L|+|L'|))

Subset-sum

- Algoritmo exacto (exponencial)
  - Implementación

```
Exact-Subset-Sum(S,t)
1. n <- |S|
2. L_0 <- [0] // lista con un único elemento de valor 0
3. for i <- 1 to n
4.     L_i <- Merge_Lists(L_{i-1},L_{i-1}+x_i)
5.     eliminar de L_i los elementos mayores que t
6. return elemento mayor en L_n
```

Subset-sum

- Ejemplo: S ={104, 102, 201, 101}, t = 308

L_0 = [0]
L_1 = merge(L_0,L_0+104) = [0, 104]
L_2 = merge(L_1,L_1+102) = [0, 102, 104, 206]
L_3 = merge(L_2,L_2+201) = [0, 102, 104, 201, 206, 303, 305, ~~407~~]
L_4 = merge(L_3,L_3+101) = [0, 101, 102, 104, 201, 203, 205, 206, 302, 303, 305, 307, ~~404~~, ~~406~~]

Solución: 307

Subset-sum

- Esquema de aproximación completamente polinomial (FPTAS)
  - Se basa en "podar" cada lista L_i después de ser creada
    - Si dos valores en L son cercanos, basta con mantener uno de ellos para encontrar una solución aproximada
    - Parámetro de poda δ, $0 < \delta < 1$
    - Podar lista L con δ, resultado es lista L'
      - Por cada elemento y removido de L, hay un elemento z en L' que aproxima y, cumpliéndose que $\frac{y}{1+\delta} \leq z \leq y$

Subset-sum

- Esquema de aproximación completamente polinomial (FPTAS)
  - Ejemplo de poda, usando $\delta = 0.1$
    - L = [10, 11, 12, 15, 20, 21, 22, 23, 24, 29]
    - L' = [10, 12, 15, 20, 23, 29]
    - Valores borrados y su representante:
      - 11 -> 10
      - 21 y 22 -> 20
      - 24 –> 23
    - Representante es un valor ligeramente menor
    - Con esto se disminuye el tamaño de la lista

Subset-sum

- Esquema de aproximación completamente polinomial (FPTAS)
  - Algoritmo de poda (trimming), tiempo $\Theta(m)$

```
Trim(L,δ)
1. m <- |L| // L = [y1, y2, …, ym]
2. L' <- [y1]
3. last <- y1
4. for i <- 2 to m
5.     if yi > last*(1+δ)
6.         agregar yi al final de L'
7.         last <- yi
8. return L'
```

Subset-sum

- Esquema de aproximación completamente polinomial (FPTAS)
  - Algoritmo aproximado ($0 < \varepsilon < 1$)

```
Approx-Subset-Sum(S,t,ε)
1. n <- |S|
2. L_0 <- [0] // lista con un único elemento de valor 0
3. for i <- 1 to n
4.     L_i <- Merge_Lists(L_{i-1},L_{i-1}+x_i)
5.     L_i <- Trim(L_i,ε/2n) // i.e., δ=ε/2n
6.     eliminar de L_i los elementos mayores que t
7. sea z* el elemento mayor en L_n
8. return z*
```

Subset-sum

- Ejemplo: S ={104, 102, 201, 101}, t = 308, $\varepsilon = 0.40$ (es decir, $\delta = 0.05$)

L_0 = [0]
L_1 = trim(merge(L_0,L_0+104)) = [0, 104]
L_2 = trim(merge(L_1,L_1+102)) = [0, 102, 206]
L_3 = trim(merge(L_2,L_2+201)) = [0, 102, 201, 303, ~~407~~]
L_4 = trim(merge(L_3,L_3+101)) = [0, 101, 201, 302, ~~404~~]

Solución aproximada: 302 (exacta era 307), cumple con estar a $1+\varepsilon$ = 40% del óptimo

Subset-sum

- Teorema: algoritmo aproximado es FPTAS
- Demostración:
  - Los elementos que se mantienen en la lista luego de la poda representan la suma de algún subconjunto de S, por lo tanto z* es la suma de algún subconjunto de S
  - Sea y* la solución óptima al problema
  - Por la línea 6 del código, se sabe que $z^* \leq y^*$

Subset-sum

- Teorema: algoritmo aproximado es FPTAS
- Demostración:
  - Se necesita demostrar que $\frac{y^*}{z^*} \leq 1+\varepsilon$ y que el algoritmos es polinomial en el tamaño de la entrada (que es n log t) y en 1+ε
  - Sea P_i la i-ésima lista original (algoritmo exacto) y L_i la lista que genera el algoritmo aproximado
    - Usando inducción sobre i, se puede mostrar que para cada elemento y en P_i que a lo más vale t, existe un z en L_i tal que $\frac{y}{(1+\varepsilon/2n)^i} \leq z \leq y$

Subset-sum

- Teorema: algoritmo aproximado es FPTAS
- Demostración:
  - La desigualdad anterior se cumple en particular para y*, es decir

$$\frac{y^*}{(1+\varepsilon/2n)^n} \leq z \leq y^*$$

  - Por lo tanto,

$$\frac{y^*}{z} \leq \left(1+\frac{\varepsilon}{2n}\right)^n$$

Subset-sum

- Teorema: algoritmo aproximado es FPTAS
- Demostración:
  - La desigualdad anterior se cumple en particular para z* (dado que z* ∈ L_n y es el mayor), es decir

$$\frac{y^*}{z^*} \leq \left(1 + \frac{\varepsilon}{2n}\right)^n$$

  - Hay que mostrar que y*/z* ≤ (1+ε). Para esto, se ocupará que

$$\lim_{n\to\infty} \left(1 + \frac{\varepsilon}{2n}\right)^n = e^{\varepsilon/2} \text{ y que } 1 + x \leq e^x \leq 1 + x + x^2$$

Subset-sum

- Teorema: algoritmo aproximado es FPTAS
- Demostración:
  - Dado que $\frac{d}{dn}\left(1+\frac{\varepsilon}{2n}\right)^n > 0$, la función crece con n al acercarse a su límite, por lo tanto

$$\left(1+\frac{\varepsilon}{2n}\right)^n \leq e^{\varepsilon/2} \leq 1+\varepsilon/2+(\varepsilon/2)^2 \leq 1+\varepsilon \text{ dado que } 0<\varepsilon<1$$

  - Con esto demostramos que Approx-Subset-Sum es (1+ε)-aproximado, falta mostrar que es completamente polinomial

Subset-sum

- Teorema: algoritmo aproximado es FPTAS
- Demostración:
  - Para esto, se derivará una cota en largo de L_i
    - Elementos consecutivos z, z' de L_i deben cumplir la relación $z'/z = 1+\varepsilon/2n$ (sino, z' se habría eliminado)
    - Esto es, deben diferir en un factor de al menos $1+\varepsilon/2n$

$$L_i = [0, 1, (1+\delta), (1+\delta)^2, \ldots]$$

    - Entonces, cada lista contiene el valor 0, posiblemente el valor 1 y hasta $\log_{1+\varepsilon/2n} t$ valores adicionales

Subset-sum

- Teorema: algoritmo aproximado es FPTAS
- Demostración:
  - Para esto, se derivará una cota en largo de L_i
    - Por lo tanto, el largo de L_i es:

$$|L_i| \leq 2 + \log_{1+\varepsilon/2n} t = 2 + \frac{\ln t}{\ln(1+\varepsilon/2n)}$$

    - Recordando que x/(1+x) ≤ ln (1+x) ≤ x cuando x > -1

$$|L_i| \leq 2 + \frac{\ln t}{\ln(1+\varepsilon/2n)} \leq 2 + \frac{2n(1+\varepsilon/2n)\ln t}{\varepsilon} \leq 2 + \frac{4n \ln t}{\varepsilon} \text{ dado que } 0 < \varepsilon < 1$$

Subset-sum

- Teorema: algoritmo aproximado es FPTAS
- Demostración:
  - La cota para $|L_i|$ es polinomial en el tamaño de la entrada (ln t = # bits para representar t) y en $1/\varepsilon$
  - Dado que el tiempo de ejecución $O((1/\varepsilon)\, n^2 \ln t)$ de Approx-Subset-Sum es polinomial en los tamaños de las listas L_i y en $1/\varepsilon$, el algoritmo es FPTAS