CC4102/CC40A/CC53A - Diseño y Análisis de Algoritmos

Jérémy Barbay

12 marzo 2011

Contents

1 Lecture Notes CC4102

1.1 Presentacion y Evaluacion

1.1.1 Presentaciones

Quien somos?

- franco-ingles-castellano

Quien son?

- Quien
  - tomo el curso CC40A en los ultimos semestres?
  - toma CC53A
  - Quien
    - piensa seguir en la universidad despues de magister?

El curso

- Tematicas
- Conceptos basicos y complejidad (3 semanas = 6 charlas)
- Algoritmos y Estructras de Datos para Memoria Secundaria (3 semanas = 6 charlas)
- Tecnicas avanzadas de diseno y analisis de algoritmos (4 semanas = 8 charlas)
- Algoritmos no convencionales (5 semanas = 10 charlas)
- Modo
  - **Clases expositivas** del profesor de catedra
    - buscando la participacion de los alumnos en pequenos problemas que se van proponiendo durante la exposicion.
  - **Clases auxiliares** dedicadas a explicar ejemplos mas extensos, resolver ejercicios propuestos, y preparacion pre y post controles.
  - **Exposicion** de las mejores tareas de los alumnos, como casos de estudio de implementacion y experimentacion.

- Evaluacion
  - Antiguamente: 3 tareas
    2/9 Examen (todas unidades)
    4/9 Controles
    2/9 Control 1 (unidades 1, 2 y parte de 3)
    2/9 Control 2 (unidades 3,4)
    1/3 Tareas
    1/9 Tarea 1
    1/9 Tarea 2
    1/9 Tarea 3
      - Nuevo: 6 tareas
    2/9 Examen (todas unidades)
    4/9 Controles
    2/9 Control 1 (unidades 1, 2 y parte de 3)
    2/9 Control 2 (unidades 3,4)
    3/9 Tareas
    1/18 Tarea 1
    1/18 Tarea 2
    1/18 Tarea 3
    1/18 Tarea 4
    1/18 Tarea 5
    1/18 Tarea 6
      - Nota Final
  - controles y examen se promedian a partes iguales (el examen reemplaza el peor control si la nota del examen es mayor)
  - tareas se promedian a partes iguales
  - nota final es 2/3 nota de controles y 1/3 de la nota de tarea.

1.1.2 Programacion

- Cuanto memoria hay en un computador? Como se maneja?
- Cual es la diferencia entre el disco duro y la memoria?
- Cuanto procesadores hay en un computador? Como se programan?
- Cual algoritmo elegir a implementar para un problema dicho?
- Cual es la diferencia entre programacion imperativa y funcional?

1.1.3 Conceptos basicos (test)

Notaciones

- $O(), o(), \Omega(), \omega(), \Theta(), \theta()$

Definiciones

- Complejidad en el peor caso
- Complejidad en promedio
- Otros modelos computacionales?

Complejidad Computacional

- Cual Algoritmos conocen? Cual son sus complejidades?
  - para buscar en un arreglo (ordenado? no ordenado?)
  - para ordenar un arreglo (en el modelo de comparaciones o no?)
- Cuales cotas inferiores conocen para...
  - buscar?
  - ordenar?
- Que problemas dificiles conocen?
  - elegir sus cursos
  - asignar salas y horarios a los cursos
  - asignar enfermeros a hospitales

1.1.4 Busqueda y Codificacion de Enteros (BONUS)

- A VER EN CASA O TUTORIAL

1.2 1. Conceptos basicos y complejidad (3 semanas = 6 charlas)

1.2.1 Resultados de Aprendizajes de la Unidad

- Comprender el concepto de complejidad de un problema como cota inferior
- Conocer tecnicas elementales para demostrar cotas inferiores
- Conocer algunos casos de estudio relevantes
- Adquirir nociones basicas de experimentacion en algoritmos.

Principales casos de estudio

- Cota inferior para minimo y maximo de un arreglo
- Caso promedio del quicksort
- Cota inferior para busqueda en un arreglo con distintas probabilidades de acceso

1.2.2 Repaso del proceso de diseño y analisis de un algoritmo.

- Problema de la Torre de Hanoi
  - Definicion
  - Cota superior
  - Cota inferior
  - A VER EN CASA
  - Variantes de la Torre de Hanoi
    - Con pesos distintos?
    - Disk Pile Problem (n discos pero h<n tamanos)
      - Cota inferior en funcion de n, en funcion de n,h
      - Cota superior en funcion de n, en funcion de n,h

* A VER EN CASA
* Minimo (respectivamente maximo) de un arreglo
    - Cota superior
    - Cota inferior
    - Minimo y Maximo de un arreglo
    - Cota superior
    - Cota inferior

1.2.3 Minimo Maximo

Cota inferior

- Sean las variables siguentes:
  - O los o elementos todavia no comparados;
  - G los g elementos que "ganaron" todas sus comparaciones hasta ahora;
  - P los p elementos que "perdieron" todas sus comparaciones hasta ahora;
  - E las e valores eliminadas (que perdieron al menos una comparacion, y ganaron al menos una comparacion);
- (o, g, p, e) describe el estado de cualquier algoritmo:
  - siempre $o + g + p + e = n$;
  - al inicio, $g = p = e = 0$ y $o = n$;
  - al final, $o = 0$, $g = p = 1$, y $e = n - 2$.

- Despues una comparacion $a?b$ en cualquier algoritmo del modelo de comparacion, (o, g, p, e) cambia en funcion del resultado de la comparacion de la manera siguente:

	$a \in O$	$a \in G$	$a \in P$	$a \in E$
$b \in O$	$o-2, g+1, p+1, e$	$o-1, p, e+1$ $o-1, g, p+1, e$	$o-1, g, p, e+1$ $o-1, g+1, p, e$	$o-1, g+1, p, e$ $o-1, g, p+1, e$
$b \in G$		$o, g-1, p, e+1$	o, g, p, e $o, g-1, p-1, e+2$	o, g, p, e $o, g-1, p, e+1$
$b \in P$			$o, g, p-1, e+1$	o, g, p, e $o, g, p-1, e+1$
$b \in E$				o, g, p, e

- En algunas configuraciones, el cambio del vector estado depende del resultado de la comparacion: un adversario puede maximizar la complejidad del algoritmo eligando el resultado de cada comparacion. El arreglo siguente contiene en graso las opciones que maximizan la complejidad del algoritmo:

	$a \in O$	$a \in G$	$a \in P$	$a \in E$
$b \in O$	$o-2, g+1, p+1, e$	$o-1, p, e+1$ $\mathbf{o-1, g, p+1, e}$	$o-1, g, p, e+1$ $\mathbf{o-1, g+1, p, e}$	$o-1, g+1, p, e$ $o-1, g, p+1, e$
$b \in G$		$o, g-1, p, e+1$	$\mathbf{o, g, p, e}$ $o, g-1, p-1, e+2$	$\mathbf{o, g, p, e}$ $o, g-1, p, e+1$
$b \in P$			$o, g, p-1, e+1$	$\mathbf{o, g, p, e}$ $o, g, p-1, e+1$
$b \in E$				o, g, p, e

- Con estas opciones, hay
  - $\lceil n/2 \rceil$ transiciones de O a $G \cup P$, y
  - $n-2$ transiciones de $G \cup P$ a E.
- Eso resulta en una complejidad en el peor caso de $\lceil 3n/2 \rceil - 2 \in 3n/2 + O(1)$ comparaciones.

Cota superior

- Calcular el minimo con el algoritmo previo, y el maximo con un algoritmo simetrico, da una complejidad de $2n-2$ comparaciones, que es demasiado.

- El algoritmo siguente calcula el max y el min en $\frac{3n}{2} - 2$ comparaciones:
  - Dividir A en $\lfloor n/2 \rfloor$ pares (y eventualemente un elemento mas, x).
  - Comparar los dos elementos de cada par.
  - Ponga los elementos superiores en el grupo S, y los elementos inferiores en el grupo I.
  - Calcula el minima m del grupo I con el algorimo de la pregunta previa, que performa $\lfloor n/2 \rfloor - 1$ comparaciones
  - Calcula el maxima M del grupo I con un algoritmo simetrico, con la misma complejidad.
  - Si n es par,
    - m y M son respectivamente el minimo y el maximo de A.
  - Sino, si $x < m$,
    - x y M son respectivamente el minimo y el maximo de A.
  - Sino, si $x > M$,
    - m y x son respectivamente el minimo y el maximo de A.
  - Sino
    - m y M son respectivamente el minimo y el maximo de A.
    - La complejidad total del algoritmo es
      - $n/2 + 2(n/2 - 1) = 3n/2 - 2 \in 3n/2 + O(1)$ si n es par
      - $(n-1)/2 + 2(n-1)/2 + 2 = 3n/2 + 1/2 \in 3n/2 + O(1)$ si
  - en la clase $3n/2 + O(1)$ en ambos casos.

1.2.4 Tecnicas para demostrar cotas inferiores:

adversario, teoria de la informacion, reduccion

1. Busqueda Ordenada (en el modelo de comparaciones)
   (a) Cota superior: $2 \lg n$ vs $1 + \lg n$
   (b) Cota inferior en el peor caso: Strategia de Adversario cota inferior en el peor caso de $1 + \lg n$
   (c) Cota inferior en el caso promedio uniforme
      - Teoria de la Informacion
      - = Arbol de Decision
      - cota inferior de $\lg(2n + 1)$, i.e. de $1 + \lg(n + 1/2)$
   (d) La complejidad del problema
      - en el peor caso es $\Theta(\lg n)$
      - en el caso promedio es $\Theta(\lg n)$
   (e) Pregunta: en este problema las cotas inferiores en el peor caso y en el mejor caso son del mismo orden. Siempre es verdad?
2. Busqueda desordenada
   (a) Complejidad en el peor caso es $\Theta(n)$
   (b) Complejidad en el caso promedio?
      - cota superior
        - Move To Front
        - ?BONUS? Transpose
      - cota inferior

- algoritmo offline, lemma del ave

- A VER EN CASA O TUTORIAL: Huffman?

3. Ordenamiento (en el modelo de comparaciones)

   - cota superior $O(n \lg n)$
   - cota inferior en el peor caso
     - cual tecnica?
       * lema del ave?
       * Strategia de Adversario?
       * Arbol Binario de Decision
     - Resultado:
       * $\Omega(n \lg n)$
   - cota inferior en el caso promedio
     - $\Omega(n \lg n)$

4. BONUS: complejidad en promedio y aleatorizada

   - La relacion entre
     - complejidad en promedio de un algoritmo deterministico
     - complejidad en el peor caso de un algoritmo aleatorizado (promedio sobre su aleatoria)

1.2.5 Metodologia de experimentacion

Al inicio, Ciencias de la computacion fue solamente experimentacion.

Turing y el codigo Enigma

- el importante estaba de solucionar la instancia del dia romper la llave del dia, basado en los messages de meteo, para decryptar los messages mas importantes
  - no mucho focus en el problema, aunque Turing si escribio la definicion de la Maquina universal.

Experimentacion basica

- correga hasta que fonciona (o parece foncionar)
  - correga hasta que entrega resultados correctos (o que parecen correctos)
  - mejora hasta que fonciona en tiempo razonable (en las instancias que tenemos)

Problemas:

- Demasiado "Ah Hoc"
  - falta de rigor, de reproducibilidad
  - desde el inicio, no "test bed" estandard, cada uno tiene sus tests.
  - mas tarde, no estandard de maquinas

Respuestas: Knuth et al.

- complejidad asymptotica: independancia de la maquina
- complejidad en el peor caso y promedio: independancia del "test bed"
- todavia es necesario de completar las estudias teoricas con programacion y experimentacion: el modelo teorico es solamente una simplificacion.

Theoreticos desarollaron un lado "mathematico" de ciencias de la computacion, con resultados importantes tal que

- NP-hardness
- "Polynomial Hierarchy" (http://en.wikipedia.org/wiki/Polynomial_hierarchy)

Theoria y Practica se completen, pero hay conflictos en ambos lados:

- demasiado theorias sin implementaciones (resultado del ambiante social tambien).
- todavia hay estudios experimentales “no reproducibles”

Sobre la "buena" manera de experimentar (”A Theoretician’s Guide to the Experimental Analysis of Algorithms”, David S. Johnson, 2001)

Fija una hipothesis antes de programar.

- aunque el objetivo sea de programar un software completo, solamente es necesario de implementar de manera eficiente la partes relevantes. El resto se puede implementar de manera “brutal”. (E.g. “Intersection Problem”)

"Incremental Programming"

- busca en la red “Agile Programming”, “Software Engineering”.
  - una experimentacion es tambien un proyecto de software, y las tecnicas de ingeniera de software se aplican tambien.
  - Construe un simulador en etapas, donde a cada etapa fonctiona el simulador entero.

"Modular Programming"

- Experimentacion es Investigacion, nunca se sabe por seguro que se va a medir despues.
  - Hay que programar de manera modular por salgar tiempo en el futuro.

Sobre la "buena" manera de presentar sus resultados experimentales. (”Presenting Data from Experiments in Algorithmics”, Peter Sanders, 2002)

El proceso:

- Experimentacion tiene un ciclo:
  - "Experimental Design" (inclue la eleccion de la hypothesis)
  - "Description of Measurement"
  - "Interpretation"
  - vuelve al paso 1.
  - La presentacion sigue la misma estructura, pero solamente

Eliges que quieres comunicar.

- el mismo dato se presenta diferamente en funcion de la emfasis del reporte.
  - pero, siempre la descripcion debe permitir la **reproducibilidad** de la experimentacion.

Tables vs 2d vs 3d plot

- tables
  - son faciles, y buenas para menos de 20 valores
  - son sobre-usadas
  - Grafes 3d
    - mas modernos, impresionantes, pero
    - en impresion no son tan informativos
    - tiene un futuro con interactive media donde el usuario puede cambiar el punto de vista, leer las valores precisas, activar o no las surfacas.
    - Grafes 2d
      - en general preferables, pero de manera inteligente!

- cosas a considerar:
- log scale en x y/o y
- rango de valores en x y/o y.
- regla de "banking to 45 deg":
- "The weighted average of the slants of the line segments in the figure should be about 45"
- se puede aproximar on un grafo en "paysage" siguiendo el ratio de oro.
- factor out la informacion ya conocida
- Maximiza el Data-Ink ratio.

- For a good book on introductory statistics with computer science examples, I recommend
- Cohen: Empirical Methods for Artificial Intelligence
- Thomas Bartz-Beielstein et al, on Empirical Methods for the

Analysis of OPtimization Algorithms

- Catherine McGeoch, A Guide to Experimental Algorithmics, (January 2011)

Sobre la "buena" manera de describir una investigacion en general:
http://www.amazon.com/Making-Sense-Students-Engineering-Technical/dp/019542591X
Making sense; a student's guide to research and writing; engineering and the technical sciences, 2d ed. Northey, Margot and Judi Jewinski. Oxford U. Press 2007 252 pages $32.50 Paperback

Otras referencias:

- Research Design: Qualitative, Quantitative, and Mixed Methods Approaches John W. Creswell http://www.amazon.com/Research-Design-Qualitative-Quantitative-Approaches/

1.2.6 Resumen de la Section

Controles y Tareas

- TAREA 1: Busqueda Binaria vs others
- TAREA 2: static cache-oblivious tree
- TAREA 3: Interpolation vs others
- CONTROL 1
- TAREA 4: Hashing
- TAREA 5: Paginamiento
- TAREA 6: Vertex Cover
- CONTROL 2
- EXAMEN

Las tareas se hacen en dos semanas.

Recurrencias y Introduccion a la programacion dinamica

- $X_n = X_{n-1} + a_n$
- Torre de Hanoi
- Fibonacci
- Subsecuencia de suma maximal
- Subsecuencia commun mas larga
  - solucion ingenua
  - Solucion en tiempo polynomial (pero espacio $O(n^2)$)
  - Solucion en espacio lineal (y tiempo $O(n^2)$)
  - (BONUS) Solucion de Hirshberg en tiempo $O(nm)$ y espacio $\min(n, m)$

Resumen de la Section

1. Conceptos Basicos
    - $O(), o(), \Omega(), \omega(), \Theta(), \theta()$
    - Complejidad en el peor caso, en promedio
    - Modelos computacionales:
        - modelo de comparaciones
        - modelo de memoria externa
2. Tecnicas de Cotas Inferiores
    - lema del ave (reduccion)
    - strategia de adversario
    - teoria de la informacion (arbol de decision binario)
    - Analisis fine
3. Metodologia de experimentacion
    - Porque?
    - Como hacer la experimentacion
    - Como analizar y presentar los resultados
4. Casos de Estudios
    - Torre de Hanoi
    - "Disk Pile problem"
    - Busqueda y Codificacion de Enteros (busqueda doblada)
    - Busqueda binaria en $\Theta(1 + \lg n)$ (mejor que $2 \lg n$)
    - Algoritmo en $2n/3 + O(1)$ comparaciones para min max

1.3 2. Algoritmos y Estructuras de Datos para Memoria Secundaria (3 semanas = 6 charlas)

1.3.1 Modelo de computacion en memoria secundaria. Accesos secuenciales y aleatorios

1. Arquitectura de un computador: la memoria

   (a) Muchos niveles de memoria

   - Procesador
   - registros
   - Cache L1
   - Cache L2
   - Memory
   - Cache
   - Disco Duro magnetico / Memory cell
   - Akamai cache
   - Discos Duros en la red
   - CD y DVDs tambien son "memoria"

   (b) Diferencias

   - velocidad
   - precio de construccion
   - relacion fisica entre volumen y velocidad
   - volatil o no
   - accesso arbitrario en tiempo constante o no.
   - latencia vs debito

   (c) Modelos formales

- RAM
- Jerarquia con dos niveles, paginas de tamano B
- Jerarquia con k niveles, de paginas de tamanos $B_1, ..., B_k$
- "Cache oblivious"
- Otros... mas practicas, mas dificil a analizar.

1.3.2 Diccionarios en Memoria Externa

1. B-arbol

   (a) $(2, 3)$ Arbol: un arbol de busqueda donde

   - cada nodo tiene 1 o 2 llaves, que corresponde a 2 o 3 hijos, con la excepcion de la raiz;
   - todas las hojas son al mismo nivel (arbol completamente balanceado)
   - Propiedades:
     - altura de un $(2, 3)$ arbol?
     - tiempo de busqueda?
     - insercion en un $(2, 3)$ arbol?
     - delecion en un $(2, 3)$ arbol?

   (b) $(d, 2d)$ Arbol un arbol donde

   - cada nodo tiene de d a $2d$ hijos, con la excepcion de la raiz (significa $d - 1$ a $2d - 1$ llaves).
   - todas las hojas son al mismo nivel (arbol completamente balanceado)
   - Propiedades:
     - altura de un $(d, 2d)$ arbol?

- tiempo de busqueda?
- insercion en un $(d, 2d)$ arbol?
- delecion en un $(d, 2d)$ arbol?

(c) B-Arbol, y variantes

- B-Arbol
  - http://www.youtube.com/watch?v=coRJrcIYbF4
  - http://en.wikipedia.org/wiki/B-tree
- B^* arbol
  - otros nodos que la raiz son llenos al menos hasta 2/3 (en vez de 1/2)
  - http://en.wikipedia.org/wiki/B*-tree
- B^+ arbol
  - the leaf nodes of the tree are chained together in the form of a linked list.

2. Van Emde Boas arbol (vEB) http://en.wikipedia.org/wiki/Van_Emde_Boas_tree

(a) Historia:

- Originalemente (1977) un estructura de datos normal, que suporta todas las operaciones en $O(\lg \lg n)$, inventada por el equipo de Peter van Emde Boas.
- No considerado utiles en practica para "pequenos" arboles.
- Applicacion a "Cache-Oblivious" algoritmos y estructuras de datos
  - optimiza el cache sin conocer el tamano B de sus paginas

- => optimiza todos los niveles sin conocer $B_1, ..., B_k$

- otras applicaciones despues en calculo parallelo (?)

(b) Definicion

- Cada nodo contiene un arbol van EmdeBoas sobre $\sqrt{n}$ elementos
- $\lg \lg n$ niveles de arboles
- operadores:
  - Findnext
  - Insert
  - Delete

(c) Analisis

- Busqueda en "tiempo" $O(\lg n / \lg B)$ a cualquier nivel i, donde el tiempo es la cuantidad de accessos al cache del nivel considerado
- Insercion
- Delecion

3. Finger Search Tree: la busqueda doblada de los arboles de busqueda

(a) Estructura de datos

(b) algorimo de busqueda

(c) analisis: busqueda en $O(\lg p)$

1.3.3 Colas de prioridad en memoria secundaria. Cotas inferiores.

1. Colas de Prioridades tradicional:
    - que se necesita?
        - Operadores
            * *insert(key,item)*
            * *findMind()*
            * *extractMin()*
            * Operadores opcionales
                · *heapify*
                · *increaseKey, decreaseKey*
                · *find*
                · *delete*
                · *successor/predecessor*
                · *merge*
                · ...
    - diccionarios: demasiado espacio para que se pide
        - menos operadores que diccionarios
            * => mas flexibilidad en la representacion
            * => mejor tiempo y/o espacio
    - **binary heap**: una estructura a dentro de muchas otras:
        - sequence-heaps
        - binomial queues

  - Fibonacci heaps
  - leftist heaps
  - min-max heaps
  - pairing heaps
  - skew heaps
  - *van Emde Boas queues*

- **van Emde Boas queues** http://www.itl.nist.gov/div897/sqg/dads/HTML/vanemdeboas.h

  - Definicion:

    "An efficient implementation of priority queues where insert, delete, get minimum, get maximum, etc. take O(log log N) time, where N is the total possible number of keys. Depending on the circumstance, the implementation is null (if the queue is empty), an integer (if the queue has one integer), a bit vector of size N (if N is small), or a special data structure: an array of priority queues, called the bottom queues, and one more priority queue of array indexes of the bottom queues."

    * rendimiento en memoria secundaria de "binary heap": muy malo?

2. Colas de Prioridades en Memoria Secundaria: diseno

   - Reference:

     - http://www.dcc.uchile.cl/ gnavarro/algoritmos/tesisRapa.pdf

       * paginas 9 hasta 16

     - Otras referencias en http://www.leekillough.com/heaps/

   - El equivalente de B-Arbol

- Muchas alternativas en practica

  (a) Buffer trees

  (b) M/B-ary heaps

  (c) array heaps

  (d) R-Heaps

  (e) Array Heaps

  (f) sequence heaps

  (g) Mas en "An experimental study of priority queues in external memory" http://portal.acm.org/citation.cfm?id=351827.384259

  (h) Colas de Prioridades en Memoria Secundaria: cota inferior?

    - Cota inferior para dictionaries es una cota inferior por

- No. La reduccion es en la otra direccion.
- Cual es la cota inferior mas simple que se puede imaginar?

  - $\Omega(n/B)$

3. *REFERENCES*

- http://en.wikipedia.org/wiki/Priority_queue
- http://en.wikipedia.org/wiki/Heap_data_structure
- "An experimental Study of Priority Queues in External Memories" by Brengel, Crauser, Ferragina and Meyer

1.3.4 Ordenamiento en memoria secundaria: Mergesort. Cota inferior.

1. Un modelo mas fino

   (a) Cuantos paginas quedan en memoria local?

   - no tan importante para busqueda
   - muy importante para applicaciones de computacion con mucho datos.

   (b) Nuevas notaciones

   - B = Tamano pagina
   - N = cantidad de elementos en total
   - n = cantidad de paginas con elementos = N/B
   - M = cantidad de memoria local
   - m = cantidad de paginas locales = M/B
   - mnemotechnique:
     - N, M, B en cantidad de palabras maquinas (=bytes?)
     - n, m en cantidad de paginas
     - $n << N$, $m << M$

   (c) En estas notaciones, usando resultos previos:

   - Insertion Sort (en un B-Arbol)
     - usa dictionarios en memoria externa
     - $N \lg N / \lg B = N \log_B N$
     - Heap Sort
       * usa colas de prioridades en memoria externa

        * $N \lg N / \lg B = N \log_B N$
        * Eso es optimo o no?

2. Cotas Inferiores en Memoria Secundaria

    - para buscar en un diccionario?
        - en modelo RAM? (de comparaciones)
            * $\lg N$
        - en modele Memoria Externa? (de comparaciones)
            * $\lg N / \lg B = \log_B N$ (ajustado)
            * para fusionar dos arreglos ordenados?
        - en modelo RAM?
            * N
        - en modelo Memoria Externa con paginas de tamano B?
            * $N/B = n$ (ajustado)
            * para fusionar k arreglos ordenados?
        - en modelo RAM?
            * N
        - en modelo de Memoria Externa con M paginas de tamano B?
            * $N/B = n$ (si $M > kB$)
            * para Ordenar

- en modelo RAM de comparaciones
  - $N \lg N$
- en modelo Memoria Externa con n/B paginas de tamano B
  - $\Omega(N/B \frac{\lg(N/B)}{\lg(M/B)})$
  - que se puede notar mas simplamente $\Omega(n \lg_m n)$
- Prueba:
  - en vez de considerar el problema de ordenamiento, supponga que el arregla sea una permutacion y considera el problema (equivalente en ese caso) de identificar cual permutacion sea.
  - inicialemente, pueden ser $N!$ permutaciones.
    - supponga que cada bloque de B elementos sea ya ordenado (impliqua un costo de al maximo $n = N/B$ accessos a la memoria externa).
    - queda $N!/((B!)^n)$ permutaciones posibles.
    - para cada accesso a una pagina de memoria externo,
  - con M entradas en memoria primaria
  - B nuevas entradas se pueden quedar de $\binom{M}{B} = \frac{M!}{B!(M-B)!}$ maneras distintas
  - calcular la union de los $M + B$ elementos reduce la cuantidad de permtuaciones por un factor de $1/\binom{M}{B}$
  - despues de t accessos (distintos) a la memoria externa, se reduci la cuantidad de permutaciones a $N!/((B!)^n \binom{M}{B}^t)$
  - cuanto accessos a la memoria sean necesarios para que queda al maximo una permutacion?

        * $N!/((B!)^n \binom{M}{B}^t)$ debe ser al maximo uno.
        * usamos las formulas siguientes:
            · $\log(x!) \approx x \log x$
            · $\log \binom{M}{B} \approx B \lg \frac{M}{B}$

$N!$	$\leq$	$(B!)^n \binom{M}{B}^t$
$N \lg N$	$\leq$	$nB \lg B + tB \lg \frac{M}{B}$
t	$\geq$	$\frac{N \lg N - nB \lg B}{B \lg(M/B)}$
	$\geq$	$\frac{N \lg(N/B)}{B \lg(M/B)}$
	$\geq$	$\frac{n \lg n}{\lg m}$
	$\geq$	$n \log_m n$

            · BONUS: Para ordenar strings, un caso particular (donde la

- $\Omega(N_1/B \log_{M/B}(N_1/B) + K_2 \lg_{M/B} K_2 + N/B)$
- donde
    - N_1 es la suma de los tamanos de las caldenas mas cortas que B
    - K_2 es la cuantidad de caldenas mas largas que B

3. Ordenar en Memoria Externa N elementos (en $n = N/B$ paginas) http://en.wikipedia.org/wiki/External_*sorting*
    - Usando dictionarios o colas de prioridades en memoria externa
        - $N \lg N / \lg B = N \log_B N$
        - No es "ajustado" con la cota inferior
        - impliqua
            * o que hay un mejor algoritmo

        * o que hay una mejor cota inferior
        * Queda un algoritmo de ordenamiento: MergeSort
    - usa la fusion de $m-1$ arreglos ordenados en memoria externa:
        (a) carga en memoria principal $m-1$ paginas, cada una la primera de su arreglo.
        (b) calcula la union de estas paginas en la pagina m de memoria principal,
            * botando la pagina cuando llena
            * cargando una nueva pagina (del mismo arreglo) cuando vacilla
        (c) La complejidad es n accessos.
    - Algoritmo:
    (a) ordena cada de las n paginas $\rightarrow$n*accessosCadanodocalculalauniond*e*marreglosyescril*
    (b) Analisis:
        * Cada nivel de recurencia costa n accessos
        * Cada nivel reduce por $m-1$ la cantidad de arreglos
        * la complejidad total es de orden $n \log_m n$ accessos. (ajustado)

4. **BONUS** cota inferior para una cola de prioridad?
    - una cola de prioridad se puede usar para ordenar (con N accessos)
    - hay una cota inferior para ordernar de $n \log_m n$
    - entonces????

1.3.5 Resultados de Aprendisajes de la Unidad Dos

- Comprender el modelo de costo de memoria secundario
- Conocer algoritmos y estructuras de datos basicos que son eficientes en memoria secundaria,
- y el analisis de su desempeno.

1.4 3. Tecnicas avanzadas de diseno y analisis de algoritmos (4 semanas = 8 charlas)

1.4.1 Presentacion de la Unidad 3

1. Dominios discretos y finitos
   (a) Interpolacion
   (b) hash
   (c) hash en memoria Secundaria
   (d) algoritmos de ordenamientos con universo finito
   (e) Analisis amortizada
      - analisi completo,
      - contabilidad de costos,
      - funcion potencial
   (f) Algoritmos en linea
      - “ski renting” problema
      - analisis competitiva (”Competitive Analysis”)

1.4.2 Uso de dominios discretos y finitos en el diseno de algoritmos y estructuras de datos

Algorimos de Busqueda (Interpolation Search, Extrapolation Search)

1. Interpolacion

   (a) Introduccion:

   - Hagaria busqueda binaria en un anuario telefonico para el nombre "Barbay"? En un diccionario para la ciudad "Zanzibar"?
   - ojala que no: se puede provechar de la informacion que da la primera letra de cada palabra

   (b) Algoritmo

   - Interaccion

   (c) Analisis

   - La analisis **en promedio** es complicada: conversamos solamente la intuicion matematica (para mas ver la publicacion cientifica de SODA04, Demaine Jones y Patrascu):
     - si las llaves son **distribuidas uniformamente**, la distancia en promedio de la posicion calculada por interpolacion **lineal** hasta la posicion real es de $\sqrt{r-l}$.
     - entonces, se puede reducir el tamano del subarreglo de n a $\sqrt{n}$ cada (dos) comparaciones
     - la busqueda por interpolacion
       * en promedio,
       * si las llaves son **distribuidas uniformamente**,
       * toma $O(\lg \lg n)$ comparaciones
       * La analisis **en el peor caso**

- Interaccion.

(d) Variantas

i. Interpolacion non-lineal

- en un anuario telefonico o en un diccionario, las frecuencias de las letras **no** son uniformes

ii. Busqueda por Interpolacion Mixta con Binaria

- Se puede buscar en tiempo
  - $O(\lg n)$ en el peor caso Y
  - $O(\lg \lg n)$ en el caso promedio?
- Solucion facil
- Solucion mas compleja

iii. Busqueda por Extrapolacion

- Tarea 3

iv. Busqueda por Extrapolacion Mixta con Doblada

- Tarea 3

2. Discussion:

- Porque todavia estudiar la complejidad en el modelo de comparaciones?
  - Cuando el peor caso es importante
  - Cuando la distribucion no es uniforme o no es conocida
  - cuando el costo de la evaluacion es mas costo que una simple comparacion (en particular para la interpolacion non lineal)

Estructuras de Datos de Busqueda

From Tries to Suffix Arrays

Tries o Arboles Digitales Ordenamos usualmente como pre-computacion para buscar despues. En el caso donde n es demasiado grande, ordenar puede ser demasiado carro. Consideramos alternativas para buscar.

- Ejemplo de trie
- Insertar los nodos siguiente en un trie
  - hola
  - holistico
  - holograme
  - hologramas
  - ola
  - ole
- .
  - h
    - o
      - l
      - a $\rightarrow$ *holai* $\rightarrow$ *holistica*

· o

· g

· r

· a

· m

· a

· $\$ \rightarrow holo gramas \rightarrow holo gramas$

- o

  * l

    1. a $\rightarrow olae \rightarrow ole$

2. Busqueda

   - con arreglos de tamano σ en cada nodo:

     – O(l) tiempo, pero O(Lσ) espacio

   - con arreglos de tamano variables en cada nodo:

   – O(l lg $sigma)tiempo(busquedabinaria), O(L)espacio(optima)$.

   - con hashing

     1. O(l) tiempo en promedio, O(L) espacio.
     2. Insercion

   - Insertar "hora" en el arbol precedente

     – .

       * h

         · o

         · l

         · a $\rightarrow holai \rightarrow holistica$

· o

· g

· r

· a

· m

· a

· \$ → *hologramas* → *hologramas*

· r

· a → **hora**

∗ o

· l

· a → *olae* → *ole*

• Insertar “holistico” en el arbol precedente

– .

∗ h

· o

· l

· a → *holai*

· s

· t

· i

· c

· a → *holisticao* → *holistico*

· o

· g

· r

· a

· m

· a

· \$ → *hologramas* → *hologramas*

· r

· a → **hora**

* o

· l

· a → *olae* → *ole*

- Borrar “hola” y “holistica”

  1. (TAREA)
  2. Tiene de “limpiar”, pero no costo mas que un factor constante de la busqueda.
  3. BONUS: PAT Trie

- Comprime las ramas de nodos de grado uno en una sola arista.
- La caldena (”string”) etiquetando la arista se guarda en el nodo hijo de la arista.
- Superio tan en tiempo que en espacio en practica.

Arboles y Arreglos de Sufijos

1. Arbol de Sufijos
2. Espacio $O(n)$

3. Construccion $O(n)$
4. Busqueda $O(m)$
5. expresion regular O(nλ) donde $0 \leq \lambda \leq 1$
6. Arreglo de Sufijos
    - Lista de sufijos ordenados
    - busqueda de patrones = dos busquedas binarias, donde cada comparacion costa $\leq m, resultandoenunacomplejidaddeO(m \lg n)$
7. BONUS: Rank en Bitmaps
    - `rank`(B, i) = cantidad de unos en $B[1, i]$
    - consideramos
        - B estatico
        - se puede almacenar $\lg n$ bits.
    - Solucion de Munro, Raman y Raman:
        - $b = 1/2 \lg n$ y $s = \lg^2 n$
        - Dividimos el index de B en
            * s Superbloques de tamano $n/s \lg n$ bits
            * b Mini bloques de tamano $n/b \lg s$ bits

$$\frac{n}{1/2 \lg n} \lg(\lg^2 n) = \frac{4nn}{\lg n} \in o(n)$$

            * un diccionario con todos los bit vectores de tamano

$$\sqrt{n} \lg n/2n \in o(n)$$

Hashing

Introduccion

- Motivaciones
- Mejor tiempo **en promedio**
- Uso de todos la herramientas que tenemos
  - dominio de las valores
  - distribuciones de probabilidades de las valores
  - Terminologia
- Tabla de Hash
  - Arreglo de tamano N
  - que contiene n elementos a dentro de un universo [1..U]
- Funccion de Hash $h(K)$
  - $h : [1..U] \rightarrow [0..N-1]$
  - se calcula rapidamente
  - distribue uniformemente (mas o menos) las llaves en la tabla en el caso ideal, $P[h(K) = i] = 1/N, \forall K, i$
- Collision
  - cuando $h(K_{1))=h(K_2)}$
  - la probabilidad es alta: "Paradoxe del cumpleanos"
  - Cual es la probabilidad que en una pieca de n personas, dos tiene la misma fecha de cumpleanos (a dentro de 365 dias)?
    - probabilidad que cada cumpleanos es unico:

$$\frac{364!}{(365-n)! \times 365^{n-1}}$$

* Probabilidad que hay al menos un cumpleanos compartido:

$$\frac{1 - 364!}{(365 - n)! \times 365^{n-1}}$$

n	Proba
10	.12
23	.5
50	.97
100	.9999996

Hashing Abierto

1. Idea principal:

   - resolver las colisions con caldenas

2. Ejemplo: (muy irealistico)

   - $h(K) = K10$
   - Secuencia de insercion $52, 18, 70, 22, 44, 38, 62$

     – Insertando al final (si hay que probar por repeticiones)

0	70
1	
2	52,22,62
3	
4	44
5	
6	
7	
8	18,38
9	

     – Insertando al final (si no hay que probar por repeticiones)

0	70
1	
2	62,22,52
3	
4	44
5	
6	
7	
8	38,18
9	

3. Analisis:

   - factor de carga es $\lambda = \frac{n}{N}$
   - Rendimiento en el peor caso: $O(n)$

Hashing Cerrado

1. Idea principal:

   - resolver las colisiones con busqueda, i.e.
     - $(h(K) + f(i))N$
   - differentes tipos de busqueda:
     - lineal $f(i) = i$
       - primary "clustering" (formacion de secuencias largas)
     - cuadratica $f(i) = i^2$
       - secundario "clustering" (si $h(K_1) = h(K_2)$, la secuencias son las mismas.
     - doble hashing $f(i) = i.h'(K)$
       - $h'(K)$ debe ser prima con N

2. Ideal Hashing

   - Imagina una funcion de hash que genera una secuencia que parece aleatoria.
   - cada posicion tiene la misma probabilidad de ser la proxima
     - Probabilidad λ de elegir una posicion ocupada
     - Probabilidad $1 - \lambda$ de elegir une posicion libre
     - la secuencia de prueba puede tocar la misma posicion mas que una vez.
     - llaves identicas todavia siguen la misma secuencia.
   - Cual es el costo promedio u_j de una busqueda negativa con j llaves en la tabla?
     - $\lambda = \frac{j}{N}$
     - $u_j = 1(1-\lambda) + 2\lambda(1-\lambda) + r\lambda^2(1-\lambda) + \ldots$
     - $= 1 + \lambda + \lambda^2 + \ldots$
     - $= \frac{1}{1-\lambda}$
     - $= \frac{1}{1-j/N}$
     - $= \frac{N}{N-j} \in [1..N]$
   - Cual es el costo promedio de una busqueda positiva s_i para el i-th elemento insertado?
     - $s_i = \frac{1}{1-i/N} = \frac{N}{N-i}$
     - $s_n = 1/n \sum \frac{N}{N-i}$
     - $= \frac{N}{n} \sum_{i=0}^{n-1} \frac{1}{N-i}$
     - $= \frac{1}{\alpha} \sum_{i=0}^{n-1} \frac{1}{N-i}$ con $\alpha = \frac{n}{N}$

- $< \frac{1}{\alpha} \int_{N-n}^{N} 1/x dx$
- $= 1/\alpha \ln \frac{N}{N-n}$
- $= 1/\alpha \ln \frac{1}{\alpha}$

- Para $\alpha = 1/2$, el costo promedio es 1.387
- Para $\alpha = 0.9$, el costo promedio es 2.559

Universal Hashing

- $h(K) = ((aK + b)p)N$
- $a \in [1..p-1]$ elegido al azar
- $b \in [0..p-1]$ elegido al azar
- p es primo y mas grande que N
- N no es necesaramente primo

Hashing en memoria externa

- Que pasa si la tabla de hashing no queda en memoria?
- IDEA: Simula un B-arbol de altura dos
    1. organiza el dato con valores de hash
    2. guarda un index en el nodo raiz
    3. usa solamente **una parte** de la valor de hash para elegir el sobre-arbol
    4. extiende el index cuando mas dato es agregado
    5. Descripcion

1. B - cantidad de elementos en una pagina
2. h - funcion de hash $\rightarrow [0..2^k - 1]$
3. D - **profundidad general**, con $D \leq k$
    - la raiz tiene 2^D punteros a las paginas horas

- la raiz es indexada con los D primeros bits de cada valor de hash.

4. d_l - **profundidad local** de cada hora l

   (a) Las valores de hash en l tienen en comun los primeros d_l bits.

   (b) Hay 2^{D-d_l} punteros a la hora l

   (c) Siempre, $d_l \leq D$

   (d) Ejemplo

5. $B = 4, k = 6, D = 2$

$$\begin{array}{ll} d_l = 2 & 000100 \\ & 001000 \\ & 001011 \\ & 001100 \end{array}$$

$$\begin{array}{ll} d_l = 2 & 010101 \\ & 011100 \end{array}$$

$$\begin{array}{ll} d_l = 1 & 100100 \\ & 101101 \\ & 110001 \\ & 111100 \end{array}$$

6. Algoritmos

   - Buscar
   - Insertar
   - Remover

7. Analisis

- Buscar, insertar remover
  - 1 acceso a la memoria secundariaa si el index se queda
- cantidad Promedio de paginas para tener n llaves
  - $\frac{n}{B \lg 2} \approx 1.44 \frac{n}{B}$
  - paginas son llenas a 69% mas o menos.

Algoritmos de Ordenamiento (Counting Sort, Bucket sort, radix sort, string sort)

Counting Sort $O(\sigma + n)$

1. for $j = 1$ to σ do $C[j] \leftarrow 0$
2. for $i = 1$ to n do $C[A[i]] + +$
3. $p \leftarrow 1$
4. for $j = 1$ to σ do
   - for $i = 1$ to $C[j]$ do
     - $A[p + +] \leftarrow j$

       Este algoritmo es bueno para ordenar multi conjuntos (donde cada elementos puede ser presente muchas veces), pero pobre para diccionarios, para cual es mejor usar la extension logica, Bucket Sort.

Bucket Sort $O(\sigma + n)$

1. for $j = 1$ to σ do $C[j] \leftarrow 0$
2. for $i = 1$ to n do $C[A[i]] + +$
3. $P[1] \leftarrow 1$
4. for $j \leftarrow 2$ to σ do
   - $P[j] \leftarrow P[j-1] + C[j-1]$

5. for $i \leftarrow 1$ to n

   - $B[P[A[i]]++] \leftarrow A[i]$

     Este algoritmo es particularmente practica para ordenar llaves asociadas con objetos, donde dos llaves pueden ser asociadas con algunas valores distintas. Nota que el ordenamiento es **estable**.

Radix Sort $O(n \lg_n \sigma) = O(cn)$

- Considera un arreglo A de tamano n sobre alfabeto σ
- si $\sigma = n$, se recuerdan que bucket sort puede ordenar A en $O(n)$
- si $\sigma = n^2$, bucket sort puede ordenar A en $O(n)$:
  - 1 ves con los $\lg n$ bits de la derecha
  - 1 ves con los $\lg n$ bits de la izquierda (utilizando la estabilidad de bucket sort)
- si $|A| = n^c$, bucket sort puede ordenar A
  - en tiempo $O(cn)$
  - con espacio $2n + \sigma \approx 3n$ ($\sigma \approx n$ a cada iteracion de bucket sort)

    El espacio se puede reducir a $2n + \sqrt{n}$ con $\lg n/2$ bits a cada iteracion de Bucketsort, cambiando la complejidad solamente por un factor de 2.

    En final, si A es de tamano n sobre un alfabeto de tamano σ, radix sort puede ordenar A en tiempo $O(n\lceil \frac{\lg \sigma}{\lg n} \rceil)$

BONUS Provechando de las repeticiones en el modelo de Comparaciones

- Se puede o no?
- Ordenar en nH_i comparaciones

BONUS String Sort

- Problema: Ordenar k strings sobre alfabeto $[\sigma]$, de largo total $n = \sum_i n_i$.
- Si $\sigma \leq k$, y cada string es de mismo tamano.
  - Si utilizamos bucket-sort de la derecha a la izquierda,

podemos ordenar en tiempo $O(n)$, porque $O(n\lceil \lg \sigma / \lg l \rceil)$ y $\sigma < n$.

- Si $\sigma \in O(1)$
  - Radix Sort sobre c simboles, donde c es el tamano minima de una string, y iterar recursivamente sobre el restos de la strings mas grande con mismo prefijo.
  - En el peor caso, la complejidad corresponde a la suma de las superficias de los bloques, aka $O(n)$.

1.4.3 Analisi amortizado de algoritmos y estructuras de datos

Analisis amortizada

- Costo Amortizado:
  - Se tiene una secuencia de n operaciones con costos $c_1, c_2, ..., c_n$.
  - Se quiere determinar $C = \sum c_i$.
  - Se puede tomar el peor caso de $ci \leq t$ para tener una cota superior de $C \leq tn$.
  - Un mejor analisis puede analizar el costo amortizado, con varias tecnicas:
    - analisis agragada
    - contabilidad de costos
    - funcion potencial.
    - Ejemplo simple: Incremento binario

· Incrementar n veces un numero binario de k bits,

· e.g. desde cero hasta $2^k - 1$, con $n = 2^k$.

· costo $\leq kn$ (brute force)

· costo $\leq n + n/2 + ... \leq 2n$ (costo amortizado)

· La tecnica usada aqui es la contabilidad de costos:

· un flip de 0 a 1 cuesta 2

· un flip de 1 a 0 cuesta 0

· cada incremento cuesta 2.

· Funcion Potencial para contar el costo

- $\phi_0 = 0$
- ϕ_i es el tamano de la bolsa luego de ejecutar $c_1, \ldots, c_i$
- $\Delta\phi_i = \phi_i - \phi_{i-1}$
- costo amortizado $= \overline{c_i} = c_i + \Delta\phi_i$
- $\sum \overline{c_i} = \sum c_i + \phi_n - \phi_0 \geq \sum c_i$
    * Ejemplo : la analisis del algoritmo para min y max al mismo tiempo.
    * En el ejemplo de Incrementacion binaria:
- $\phi =$ cantidad de unos en el numero
- $\phi_0 = 0$
- $c_i = l + 1$ cuando hay l unos
- $\Delta\phi_i = -l + 1$
- $\sum \overline{c_i} = \sum c_i + \phi_n - \phi_0 \geq 2$
- Ejemplo: realocacion amortizada
    * considera el tipo “Vector” en Java.

* de tamano fijo n
* cuando accede a $n+1$, crea un otro arreglo de tamano $2n$, y copia todo.
* cual es el costo amortizado si agregando elementos uno a uno?

Colas de Prioridades

1. REFERENCIAS:
   - http://www.leekillough.com/heaps/
   - Problema: Dado un conjunto (dinamica) de n tareas con
2. operaciones basicas:
   - M.build({e1,e2,...,2n})
   - M.insert(e)
   - M.min
   - M.deleteMin
   - operaciones adicionales ("Addressable priority queues")
     - M.insert(e), volviendo un puntero h ("handle") al elemento insertado
     - M.remove(h), remudando el elemento especificado para h
     - M.decreaseKey(h,k), reduciendo la llave del elemento especificado para h
     - M.merge(Q), agregando el heap Q al heap M.
     - Soluciones (conocidas o no):
3. Colas de prioridades binarias ("Binary Heaps")
   - La solucion tradicional

- un arreglo de n elementos
- hijos del nodo i en posiciones $2i$ y $2i+1$
- la valor de cada nodo es mas pequena que las valores de su hijos.
- Complejidad: Espacio n+O(1), y

- Detalles de implementacion (mejorando las constantes)
  - Cuidado de no implementar M.build({e1,e2,...,2n}) con n inserciones (sift up $\rightarrow$O(nlg n)), pero con $n/2$ sift-down ($\rightarrow$ $O(n)$).
  - En M.deleteMin(), algunas variantes de implementacion (despues de cambiar el min con $A[n]$):
    (a) dos comparaciones en cada nivel hasta encontrar la posicion final de $A[n]$
      * $2 \lg n$ comparaciones en el peor caso
      * $\lg n$ copias

    (b) una comparacion en cada nivel para definir un camino de tamano lg n, y una busqueda binaria para encontrar la posicion final
      * $\lg n + O(\lg \lg n)$ comparaciones en el peor caso
      * $\lg n$ copias en el peor caso

    (c) una comparacion en cada nivel para definir un camino de tamano lg n, y una busqueda **secuencial** up para encontrar la posicion final
      * $2 \lg n$ comparaciones en el peor caso
      * $\lg n$ copias en el peor caso

    * pero en practica y promedio mucho mejor.
    * Flexibilidad de estructuras

- Porque la complejidad es mejor con un heap que con un arreglo ordenado?
    * Para cada conjunto, hay solamente un arreglo ordenado que puede representarlo, pero muchos heaps posibles: eso da mas flexibilidad para la mantencion dinamica de la estructura de datos.

- Colas de prioridades con punteros ("Addressable priority queues")
    * Algun que mas flexible que los arreglos ordenados, las colas de prioridades binarias todavia son de estructura muy estricta, por ejemplo para la union de filas. Estructuras mas flexibles consideran un "bosque" de arboles. Ademas, estas estructuras de arboles son implementadas con puntadores (en ves de implementarlos en arreglos).
    * Hay diferentes variantes. Todas tienen en comun los puntos siguientes:
        · un puntero **minPtr** indica el nodo de valor minima en el bosque, raiz de alguno arbol.
        · **insert** agregas un nuevo arbol al bosque en tiempo $O(1)$
        · **deleteMin** remudas el nodo indicado par minPtr, dividiendo su arbol en dos nuevos arboles. Buscamos para el nuevo min y fusionamos algunos arboles (los detalles diferencian las variantes)
        · **decreaseKey(h,k)** es implementado cortando el arbol al nodo indicado por h, y rebalanceando el arbol cortado.

· **delete()** es reducido a **decreaseKey(h,0)** y **deleteMin**

4. "Pairing Heaps"

   - Malo rendimiento en el peor caso, pero bastante buena en practica.
   - rebalancea los arboles solamente en deleteMin, y solamente con pares de raises (i.e. la cantidad de arboles es reducida por dos a cada deleteMin).

Operacion	Amortizado
Insert(C,x)	$O(1)$
Merge	$O(1)$
ExtractMin	$O(\lg n)$
decreaseKey(h,k)	$\Omega(n \lg n \lg \lg n)$

5. Colas de prioridades binomiales ("Binomial Heaps")

   http://en.wikipedia.org/wiki/Binomial$_h eap$ o p136 de Melhorn y Sanders

   - Definicion
     - Un **arbol binomial** (de orden k) tiene exactamente k hijos de orden distintos $k-1, k-2, ..., 0$. [Un arbol binomial de orden 0 tiene 0 hijos.]
     - Un **bosque binomial** es un conjunto de arboles binomiales de orden **distintas** (i.e. hay cero o uno arboles de cada orden).
     - Propiedades
       * Para cada arbol T
         · $h(T) \leq lg|T|$
         · $|T| \geq 2^{h(T)}$
       * Para el bosque

· $\forall n$ hay solamente uno bosque binomial con n nodos.

· al maxima tiene $\lfloor \lg(n+1) \rfloor$ arboles.

· la descomposicion del bosque en arboles de orden k corresponde a la descomposicion de n en base de dos.

· Definicion

* una **cola binomial** es un bosque binomial donde cada nodo almacena una clave, y siempre la clave de un padre es inferior o igual a la clave de un hijo.

* Operaciones

- Union

  – Union de dos arboles binomiales de mismo orden:

    * agrega T_2 a T_1 si T_1 tiene la raiz mas pequena.

  – Union de dos bosques binomiales:

    * si hay uno arbol de orden k, es lo de la union

    * si hay dos arboles de orden k, calcula la union en un arbol de orden $k+1$

    * la propagacion es similar a la suma de enteros en binario.

  – Complejidad

    * $O(\lg n)$ en el peor caso

    * Insert

  – agrega un arbol de orden 0 y hace la union si necesitado

- Complejidad $O(\lg n)$ en el peor caso
- Puede ser $O(1)$ sin corregir el bosque, que tiene de ser corregido mas tarde, que puede ser en tiempo $O(n)$ peor caso, pero sera $O(\lg n)$ en tiempo amortizado.
- Minima
    * lei la lista de al maxima $\lfloor \lg(n+1) \rfloor$ raices
    * Complejidad $O(\lg n)$
    * Puede ser $O(1)$ si precalculando un puntero al minima, que tiene de ser corregido (en tiempo $O(\lg n)$) a cada modificacion.
    * DeleteMin
        · encontra el min
        · remuda el min de su arbol (la raiz)
        · reordena su hijos para su orden, en un bosque binomial
        · hace la union con el bosque binomial original, menos el arbol del min
        · complejidad $O(\lg n)$
        · DecreaseKey
        · sigue el camino abajo hasta que la condicion del heap es corregida.
        · cada arbol tiene altura lg n, entonces la complejidad es $O(\lg n)$ en el peor caso.
        · Delete
        · reducido a DecreaseKey+DeleteMin

6. Colas de prioridades de Fibonacci ("Fibonacci Heaps")

   http://en.wikipedia.org/wiki/Fibonacci$_h eap$ o pagina 135 de Melhorn y Sanders.

- Diferencia con la cola binomial:
  - relax la estructura de los arboles (heap-forma), pero de forma controlada.
  - el tamano de un sub-arbol cual raiz tiene k hijos es al maxima $F_k + 2$, donde F_k es el k-esimo numero de Fibonacci.
  - Operaciones

7. Overview (copy de http://en.wikipedia.org/wiki/Fibonacci$_h eap$)

	Linked List	Binary Tree	(Min-)Heap	Fibonacci Heap	Brodal Qu
insert	$O(1)$	$O(\lg n)$	$O(\lg n)$	$O(1)$	$O(1)$
accessmin	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
deletemin	$O(n)$	$O(\lg n)$	$O(\lg n)$	$O(\lg n)^*$	$O(\lg n)$
decreasekey	$O(1)$	$O(\lg n)$	$O(\lg n)$	$O(1)^*$	$O(1)$
delete	$O(n)$	$O(n)$	$O(\lg n)$	$O(\lg n)^*$	$O(\lg n)$
merge	$O(1)$	$O(mlog(n+m))$	$O(mlog(n+m))$	$O(1)$	$O(1)$

8. BONUS Heapsort
   - in place
   - $O(n \lg n)$ con cualquiera de estas variantes.

1.4.4 Analisi adaptativa de algoritmos y estructuras de datos

Analisi en el peor caso: a dentro de que?

- peor caso a dentro de las instancias de tamano fijo
  - el tamano puede ser multidimensional (e.g. grafos)
- peor caso a dentro de las instancias de tamano de resultado fijado
  - cantidad infinita
- peor caso a dentro de las instancias de **dificultad** fijada
  - cantidad infinita
- peor caso a dentro de las instancias de **tamano fijo y dificultad** fijada

Busqueda Doblada: $1+2\lceil \lg p \rceil$ **comparaciones** El algoritmo de busqueda doblada

- encuentra en $1+\lceil \lg p \rceil$ comparaciones un intervalo de tama$/2quecontienex, encuentrapen1+\lceil \lg p \rceil$ comparaciones adicionales (usando la segunda variante de la bsqueda binaria),
- por un total de $1 + 2\lceil \lg p \rceil$ comparaciones

1.4.5 Algoritmos en linea. Competividad. Adaptividad.

List Accessing REFERENCIA: Capitulo 1 en "Online Computation and Competitive Analysis", de Allan Borodin y Ran El-Yaniv

1. "List Accessing"
   - Considera la secuencia de busqueda de tamano n, en un diccionario de tamano σ: "1,1,1,1,1,1,2,2,2,3,3,3,4,4,4,5,5,5,6,6,6,..."
   - Cual es el rendimiento de una estructura de diccionario **estatica** (tal que AVL) en este secuencia?
     - $n \lg \sigma$
   - Se puede mejorar?
     - si, utilizando la **localidad** de las consultas, en **estructuras de datos dinamicas**.
2. Soluciones
   (a) MTF ("Move To Front"):
   - pone las llaves en un arreglo desordenado
   - buscas secuencialmente en el arreglo
   - muda la llave encontrada en frente

   (b) TRANS ("Transpose"):

- pone las llaves en un arreglo desordenado
- buscas secuencialmente en el arreglo
- muda la llave encontrada de una posicion mas cerca del frente

(c) FC ("Frequency Count"):

- mantiene un contador para la frecuencia de cada elemento
- mantiene la lista ordenada para frecuencia decreciente.

(d) Splay Trees (y otras estructuras con propiedades de localidad) (http://www.dcc.uchile.cl/ cc30a/apuntes/Diccionario/8b)

3. Estos son "Algoritmos en Linea"

- algoritmo de optimizacion
- que conoce solamente una parte de la entrada al tiempo t.
- se compara a la competitividad con el algoritmo offline que conoce toda la instancia.
- Como se puede medir su complejidad?
  - cada algoritmo ejecuta $O(n)$ comparaciones para cada busqueda en el peor caso!!!!
  - tiene de considerar instancies "faciles" y "dificiles"
  - una medida de dificultad
  - e.g. el rendimiento del *mejor algoritmo "offline"*

4. Competitive Analysis: instancias "dificiles" o "faciles"

- Las estructuras de datos dinamicas pueden aprovechar de secuencias "faciles" de consultas: eso se llama "online".
- pero para muchos problemas online, todas las heuristicas se comportan de la misma manera en el peor caso.

- Por eso se identifica una medida de dificultad de las instancias, y se comparan los rendimientos de los algoritmos sobre instancias que tienen una valor fijada de este medida de dificultad.
- Tradicionalmente, esta medida de dificultad es el rendimiento del mejor algoritmo "offline": eso se llama **competitive analysis**, resultando en el **competitive ratio**, el ratio entre la complejidad del algoritmo ONLINE y la complejidad del mejor algoritmo OFFLINE.
  - por ejemplo, veamos que MTF tiene un competitive ratio de 2
- Pero todavia hay algoritmos con performancia practicas muy distintas que tienen el mismo competitive ratio. Por eso se introduce otras medidas de dificultadas mas sofisticadas, y mas especialidades en cada problema.

5. Competitividad
   - Optimizacion/aproximacion
     - A es $k(n)$ **competitiva** para un problema de **minimizacion** si
     $$\exists b \forall n, x,\ |x| = n,\ C_A(x) - k(n)C_{OPT}(x) \leq b$$
     - A es **k(n) competitiva** para un problema de **maximizacion** si
     $$\exists b, \forall n, x,\ |x| = n,\ C_{OPT}(x) - k(n)C_A(x) \leq b$$
     - Competitiva Ratio
       * an algoritmo en linea es **c-competitiva** si
       $$\exists \alpha, \forall I ALG(I) \leq cOPT(I) + \alpha$$
       * an algoritmo en linea es **estrictamente c-competitiva** si
       $$\forall I ALG(I) \leq cOPT(I)$$

6. Sleator-Tarjan sobre MTF

- costo de una busqueda negativa (la llave NO esta en el diccionario)
  - σ
- costo de una busqueda positiva (la llave esta en el diccionario)
  - la posicion de la llave, no mas que σ
  - en promedio para una distribucion de probabilidad fijada:
    - $MTF \leq 2OPT$,
  - Prueba: (from my notes in my CS240 slides)
    How does MTF compare to the optimal ordering?
    - Assume that:
      - the keys $k_1, \ldots, k_n$ have probabilities $p_1 \geq p_2 \geq \ldots \geq p_n \geq 0$
      - the list is used sufficiently to reach a steady state.
    - Then:
      $$C_{MTF} < 2 \cdot C_{OPT}$$
    - Proof:
      - $C_{OPT} = \sum_{j=1}^{n} j p_j$
      - $C_{MTF} = \sum_{j=1}^{n} p_j$(cost of finding k_j)
      - $C_{MTF} = \sum_{j=1}^{n} p_j$(1+number of keys before k_j)

* To compute the average number of keys before k_j:

$$\Pr[\ k_i \text{ before } k_j] = \frac{p_i}{p_i + p_j} E(\text{ number of keys before } k_j)$$

$$\sum_{i \neq j} \frac{p_i}{p_i + p_j}$$

k_i is before k_j if and only if k_i was accessed more recently than k_j.

– Consider the last time either k_i or k_j was looked up. What is the probability that it was k_i?

$$P(k_i \text{ before } k_j) = \frac{P(k_i \text{ chosen })}{P(k_i \text{ or } k_j \text{ chosen })}$$

$$\frac{p_i}{p_i + p_j}$$

Therefore,

$$\begin{aligned}
C_{MTF} &= \sum_{j=1}^{n} p_j (1 + \sum_{i \neq j} \frac{p_i}{p_i + p_j}) && \text{(Joining both previous formulas.)} \\
&= 1 + 2 \sum_{j=1}^{n} \sum_{i<j} \frac{p_i p_j}{p_i + p_j} && \text{(By reordering the terms.)} \\
&\leq 1 + 2 \sum_{j=1}^{n} p_j (\sum_{i<j} 1) && \text{(Because } \tfrac{p_i}{p_i + p_j} \leq 1.) \\
&= 1 + 2 \sum_{j=1}^{n} p_j (j-1) \\
&= 1 + 2C_{OPT} + 2 \sum_{j=1}^{n} (-p_j) \\
&= 2C_{OPT} - 1. && \text{(Because } \textstyle\sum_{j=1}^{n} (p_j) = 1.)
\end{aligned}$$

- Applicaciones a la compression de textos
    - Bentley, Sleator, Tarjan and Wei proponieron de comprimir un texto utilizando una lista dinamica, donde el codigo para un simbolo es la posicion del simbolo en la lista.
        - Experimentalmente, se compara a Huffman:
            * a veces mucho mejor
            * nunca mucho peor.
            * Burrows and Wheeler proponieron una transformacion
    - Experimentalmente, 6% mejor que GZip, que es enorme!

Paginamiento Deterministico REFERENCIA: Capitulo 2 en "Online Computation and Competitive Analysis", de Allan Borodin y Ran El-Yaniv

1. Paginamiento
    - Definicion:
        - elegir cual paginas guardar en memoria, dado
            * una secuencia online de n consultas para paginas, y
            * un cache de k paginas.
    - Politicas:
        - LRU (Least Recently Used)
        - CLOCK (1bit LRU)
        - FIFO (First In First Out)
        - LFU (Least Frequently Used)

- LIFO = MRU (Most Recently Used)
- FWF (Flush When Full)
- LFD (Offline, Longuest Forward Distance)

- Ustedes tienen una idea de cuales son las peores/mejores?

2. Relacion con "List Accessing"

   (a) Cada "List accessing" algoritmo corresponde a un algoritmo de paginamiento:

   - cada miss, borra el ultimo elemento de la lista y "inserta" el nuevo elemento.

   (b) No hay una reduccion tan clara en la otra direccion.

3. Offline analysis

   - LFD performa O(n/k) misses
   - Cualquier algoritmo Offline performa Omega(n/k) en el peor caso.

4. Online analisis: resultados basicos

   (a) $\forall A$ online, hay una entrada con n fallas.

   - Estrategia de adversario.

   (b) No algoritmo online puede ser mejor que k competitivo.

   - Obvio, comparando con LFD.

   (c) MRU=LIFO NO es competitivo

   - Considera S= p_1,p_2,...,p_k,$p_{\{}k+1\}, p_k$,$p_{\{}k+1\}, p_k$,$p_{\{}k+1\}, p_k$,...

- despues las k primeras consultas, MRU va a tener un miss cada consulta, cuando LFD nunca mas.

(d) LFU no es competitivo

- Considera $l > 0$ y $S = p_1^l, p_2^l, \ldots, p_{k-1}^l, (p_k, p_{k+1})^{l-1}$
- Despues de las $(k-1)l$ primeras consultas, LFU va a tener un miss cada consulta, cuando LFD solamente dos.

(e) Que tal de FWF?

- MRU=LIFO es un poco estupido, su mala rendimiento no es una sorpresa.
- FWF es un algoritmo muy ingenuo tambien, pero vamos a ver que no tiene un rendimiento tan mal "en teoria".

5. BONUS: Competive Analysis: Algoritmos a Marcas

(a) k-fases particiones

Para cada secuencia S, partitionala en secuencias $S_1, \ldots, S_\delta$ tal que

- $S_0 = \emptyset$
- S_i es la secuencia Maxima despues de S_{i-1} que contiene al maximum k consultas distintas.
- Llamamos "fase i" el tiempo que el algoritmo considera elementos de la subsecuencia S_i.
- Nota que eso es independiente del algoritmo considerado.

(b) Algoritmo con marcas

- agrega a cada pagina de memoria lenta un bit de marca.
- al inicio de cada fase, remuda las marcas de cada pagina en memoria.

- a dentro de una fase, marca una pagina la primera vez que es consultada.
- un algoritmo a marca ("marking algorithm") es un algoritmo que nunca remuda una pagina marcada de su cache.

(c) Un algoritmo con marcas es k-competitiva

- En cada fase,
  - un algoritmo ONLINE con marcas performa al maximum k miss.
  - Un algoritmo OFFLINE (e.g. LFD) performa al minimum 1 miss.
- QED

(d) LRU, CLOCK y FWF son algoritmos con marcas

(e) LRU, CLOCK y FWF tienen un ratio competitivo OPTIMO

6. Mas resultados:

(a) La analisis se puede generalizar al caso donde el algoritmo offline tiene h paginas, y el algoritmo online tiene $k \geq h$ paginas.

- Cada algoritmo **con marcas** es $\frac{k}{k-h+1}$-competitiva.

(b) Definicion de algoritmos (conservadores) da resultado similar para FIFO (que no es con marcas pero es conservador).

(c) En practica, sabemos que LRU es mucho mejor que FWF (for instancia). Habia mucha investigacion para intentar de mejorar la analisis por 20 anos, ahora parece que hay una analisis que explica la mejor rendimiento de LRU sobre FWF, y de variantes de LRU que pueden saber x pasos en el futuro (Reza Dorrigiv y Alex Lopez-Ortiz).

"Ski Renting" http://en.wikipedia.org/wiki/Ski$_r ental_p roblem$

1.4.6 Conclusion Unidad 3

- Resultados de Aprendisajes de la Unidad
  - Comprender las tecnicas de algoritmos de
    * costo amortizado,
    * uso de finitud, y
    * algorimos competitivos
  - Ser capaz de disenar y analzar algoritmos y estructuras de datos basados en estos principios.
  - conocer algunos casos de estudio relevantes
  - Principales casos de estudio:
    * estructuras para union-find,
    * colas binomiales
    * splay trees,
    * busqueda por interpolacion
    * radix sort
    * arboles de van Emde Boas
    * arboles de sufijos
    * tecnica de los cuatro rusos,
    * paginamiento
    * busqueda no acotada (unbounded search, doubling search)

1.5 4. Algoritmos no convencionales (5 semanas = 10 charlas)

:LOGBOOK:

- State “PAUS” from “TODO” [2010-10-07 Thu 13:44]
- State “TODO” from “” [2010-10-07 Thu 13:43]

:END:

1.5.1 Descripcion de la Unidad

:LOGBOOK:

- State "DONE" from "" [2010-10-14 Thu 22:34]

:END:

Resultados de Aprendisajes de la Unidad :LOGBOOK:

- State "DONE" from "" [2010-10-07 Thu 13:43]

:END:

- Comprender el concepto de algoritmos
  - aleatorizados,
  - probabilisticos,
  - aproximados
  - paralelos
- y cuando son relevantes
- ser capaz de disenar y analizar algoritmos de estos tipos
- Conocer algunos casos de estudio relevantes

Principales casos de estudio: :LOGBOOK:

- State "DONE" from "" [2010-10-07 Thu 13:43]

:END:

- primalidad
- Karp Rabin para busqueda en strings
- numero mayoritario
- arboles binarios de busqueda aleatorizados
- quicksort

- hashing universal y perfecto
- aproximaciones para recubrimiento de vertices
- vendedor viajero
- mochila
- ordenamiento paralelo
- paralel prefix

1.5.2 Aleatorizacion (1 semana = 2 charlas)

:LOGBOOK:

- State "DONE" from "ACTF" [2010-10-18 Mon 19:45]
- State "ACTF" from "PAUS" [2010-10-18 Mon 15:36]
- State "PAUS" from "DONE" [2010-10-14 Thu 23:28]
- State "DONE" from "PAUS" [2010-10-14 Thu 22:34]
- State "PAUS" from "" [2010-10-14 Thu 15:05]

:END:

- REFERENCIA:
  - Capitulo 1 en "Randomized Algorithms", de Rajeev Motwani and Prabhakar Raghavan.

Definiciones

- **Algoritmos deterministico**
  - algoritmo que usa solamente instrucciones **deterministicas**.
  - algoritmo de cual la ejecucion (y, entonces, el rendimiento) depende solamente del input.
- **Algoritmos aleatorizados**

- algoritmo que usa una instruccion **aleatorizada** (potentialemente muchas veces)
- una distribucion de probabilidades sobre una familia de algoritmos deterministicos.

- **Analisis probabilistica**
  - analysis del rendimiento de un algoritmo, en promedio sobre
    * el aleatorio de la entrada, o
    * el aleatorio del algoritmo, o
    * los dos.
  - veamos que son nociones equivalentes.

El poder de un algoritmo aleatorizado Nota: Trae los juguetes/cajas de colores, con un tesoro a esconder a dentro. Ejemplos de algoritmos o estructuras de datos aleatorizados

1. **hidden coin**
   - Decidir si un elemento pertenece en un lista desordenada de tamano k, o si hay una moneda a dentro de una de las k caja.
   - cual son las complejidades determinisitica y aleatorizada del problema de encontrar **una** moneda, con c la cantidad de monedas,
     - si $c = 1$?
     - si $c = n - 1$?
     - si $c = n/2$?
2. Respuestas:
   - Si una sola instancia de la valor buscada
     - k en el peor caso deterministico

  - $k/2$ en (promedio y en) el peor caso aleatorio
    * con una direccion al azar
    * con $\lg(k!)$ bits aleatorios
- Si r instancias de la valor buscada
  - $k - r$ en el peor caso deterministico
  - $O(k/r)$ en (promedio y en) el peor caso aleatorio

3. Decidir si un elemento pertenece en una lista ordenadas de tamano n
   - $\Theta(\lg n)$ comparaciones en ambos casos, deterministico y probabilistico.

4. **problema de union**
   - Decidir si un elemento pertenece en una de las k listas ordenadas de tamano n
   - Si una sola lista contiene la valor buscada
     - k busquedas en el peor caso deterministico, que da $k \lg(n)$ comparaciones
     - $k/2$ busquedas en (promedio y en) el peor caso aleatorio, que da $k \lg(n)/2$ comparaciones
     - Si $r < k$ listas contienen la valor buscada
       * $k - r$ busquedas en el peor caso deterministico, que dan $(k - r) \lg(n)$ comparaciones
       * k/r busquedas en (promedio y en) el peor caso aleatorio, que dan $(k/r) \lg(n)$ comparaciones
       * Si $r = k$ listas contienen la valor buscada

· k busquedas en el peor caso deterministico, en promedio y en el peor caso aleatorio, que dan $k \lg n$ comparaciones

5. **problema de interseccion**

   - dado k arreglos ordenados de tamano n cada uno, y un elemento x.
   - cual son las complejidades determinisitica y aleatorizada del problema de encontrar **un** arreglo que no contiene x (i.e. mostrar que la interseccion de $\{x\}$ con $\cap A$ es vacilla)?
     - si $c = 1$ arreglo contiene x?
     - si $c = n - 1$ arreglos contienen x?
     - si $c = n/2$ arreglos contienen x?

6. Eso se puede aplicar a la interseccion de posting lists (Google Queries).

Aleatorizacion de la entrada

- Independencia de la distribucion de la entrada
  - Si el input sigue una distribucion non-conocida, el input perturbado tiene una distribucion conocida (para una perturbacion bien elegida)
  - Ejemplo:
    * flip b de una bit con probabilidad p que puede ser distinta de $1/2$.
    * suma-lo modulo 1 con un otro bit aleatorizado, con probabilidad $1/2$ de ser uno.
    * la suma es igual a uno con probabilidad $1/2$.
    * Estructuras de datos aleatorizadas
      · **funciones de hash**: estructura de datos aleatorizada, donde (a, b) son elegidos al azar.
      · **skiplists**: estructura de datos aleatorizada, que simula en promedio un arbol binario

SkipLists :LOGBOOK:

- State "DONE" from "" [2010-10-07 Thu 10:54]

:END:

1. Estructuras de datos para diccionarios

    - [ ] Arreglo ordenado
    - [ ] "Move To Front" list (did they see it already?)
    - [ ] Arboles binarios
    - [ ] Arboles binarios aleatorizados
    - [ ] Arboles 2-3 (they saw it already?)
    - [ ] Red-Black Trees (they saw it already?)
    - [ ] AVL
    - [ ] Skip List
    - [ ] Splay trees

2. Skip Lists

    (a) Motivacion

    - un arbol binario con entradas aleatorizadas tienen una altura $O(\lg n)$, pero eso supone un orden de input aleatorizados.
    - El objetivo de las "skip lists" es de poner el aleatorio a dentro de la estructura.
    - tambien, es el equivalente de una busqueda binaria en listas via un resumen de resumen de resumen...

    (b) Definicion

- una skip-list de altura h para un diccionario D de n elementos es una familia de lists $S_0, \ldots, S_h$ y un puntero al primero elemento de S_h, tal que
  - S_0 contiene D;
  - cada S_i contiene un subconjunto aleatorio de S_{i-1} (en promedio la mitad)
  - se puede navegar de la izquierda a la derecha, y de la cima hasta abajo.
- se puede ver como n torres de altura aleatorizadas, conectadas horizontalmente con punteros de la izquierda a la derecha.
- la informacion del diccionario estan solamente en S_0 (no se duplica)

(c) Ejemplo

4	X	-	-	-	-	-	-	->	X
3	X	-	->	X	-	-	-	->	X
2	X	-	->	X	X	->	X	->	X
1	X	X	->	X	X	->	X	->	X
0	X	X	X	X	X	X	X	X	X
	$-\infty$	10	20	30	40	50	60	70	∞

(d) Operaciones

- Search(x):
  - Start a the first element of S_h
  - while not in S_0
    - go down one level
    - go right till finding a key larger than x
    - Insert(x)

- Search(x)
- create a tower of random height ($p = 1/2$ to increase height, typically)
- insert it in all the lists it cuts.
- Delete(x)
  - Search(x)
  - remove tower from all lists.

(e) Ejemplos

- Insert(55) con secuencia aleatora (1,0)

(f) Insert(25) con (1,1,1,1,0)

5	**X**	-	-	-	-	-	-	-	-	**->**	**X**
4	X	-	**->**	**X**	-	-	-	-	-	->	X
3	X	-	**->**	**X**	X	-	-	-	-	->	X
2	X	-	**->**	**X**	X	X	-	->	X	->	X
1	X	X	**->**	**X**	X	X	->	X	X	->	X
0	X	X	X	**X**	X	X	X	X	X	X	X
	$-\infty$	10	20	**25**	30	40	50	55	60	70	∞

(g) Delete(60)

5	X	-	-	-	-	-	-	-	->	X
4	X	-	->	X	-	-	-	-	->	X
3	X	-	->	X	X	-	-	-	->	X
2	X	-	->	X	X	X	-	-	**->**	X
1	X	X	->	X	X	X	->	X	->	X
0	X	X	X	X	X	X	X	X	X	X
	$-\infty$	10	20	25	30	40	50	55	70	∞

(h) Analisis

- Espacio: Cuanto nodos en promedio?

- cuanto nodos en lista S_i?
  - $n/2^i$ en promedio
- Summa sobre todos los niveles
  - $n \sum 1/2^i < 2n$
  - Tiempo:
- altura promedio es $O(\lg n)$
- tiempo promedio es $O(\lg n)$

Paginamiento al Azar :OPTIONAL: :LOGBOOK:

- State "DONE" from "" [2010-10-07 Thu 10:53]

:END:

- REFERENCIA:
  - Capitulo 3 en "Online Computation and Competitive Analysis", de Allan Borodin y Ran El-Yaniv
  - Capitulo 13 en "Randomized Algorithms", de Rajeev Motwani and Prabhakar Raghavan, p. 368

Tipos de Adversarios (cf p372 [Motwani Raghavan)]

- Veamos en el caso deterministico un tipo de adversario offline, como medida de dificultad para las instancias online. Para el problema de paginamiento con k paginas, el ratio optima entre un algoritmo online y offline es de k (e.g. entre LRU y LFD).

- DEFINICION:

  En el caso aleatorizado, se puede considerar mas tipos de adversarios, cada uno definiendo una medida de dificultad y un modelo de complejidad.

1. Adversario "Oblivious" ("Oblivious Adversary")

   El adversario conoce A pero no R: el elija su instancia completamente al inicial, antes de la ejecucion online del algoritmo.

2. Adversario Offline adaptativo

   Para este definicion, es mas facil de pensar a un adversario como un agente distinto del algoritmo offline con quien se compara el algoritmo online.

   El adversario conoce A en total, pero R online, y le utiliza para generar una instancia peor I. Este instancia I es utilizada de nuevo para ejecutar el algoritmo offline (quien conoce el futuro) y producir las complejidades (a cada instante online) con cual comparar la complejidad del algoritmo online.

3. Adversario Online adaptativo

   En este definicion, el algoritmo conoce A en total, construir la instancia I online como en el caso precedente, pero tiene de tiene de resolverla online tambien (de una manera, no se ve en el futuro).

Comparacion de los Tipos de adversarios.

- Por las definiciones, es claro que
  - el adversario offline adaptativo
    - es mas poderoso que
  - el adversario online adaptativo
    - es mas poderoso que
  - el adversario oblivious

Competitiva Ratios

- Para un algoritmo online A, para cada tipo de adversario se define un ratio de competitividad:

- C_A^{obl}: competitivo ratio con adversario oblivious
- C_A^{aon}: competitivo ratio con adversario adaptativo online
- C_A^{aof}: competitivo ratio con adversario adaptativo offline

- Para un problema el ratio de competitividad de un problema es el ratio de competitividad minima sobre todos los algoritmos correctos para este problema:

$$C^{obl} \leq C^{aon} \leq C^{aof} \leq C^{det}$$

donde C^{det} es el competitivo ratio de un algoritmo online deterministico.

Arboles Binarios de Busqueda aleatorizados :LOGBOOK:

- State "DONE" from "" [2010-10-07 Thu 10:53]

:END:

- Conrado Martinez. Randomized binary search trees. Algorithms Seminar. Universitat Politecnica de Catalunya, Spain, 1996.
- Conrado Martinez and Salvador Roura. Randomized binary search trees. J. ACM, 45(2):288–323, 1998.
- http://en.wikipedia.org/wiki/Treap, seccion "Randomized$_{\text{binary}}$_search_tree".

Complejidad Probabilistica: cotas inferiores :LOGBOOK:

- State "DONE" from "" [2010-10-07 Thu 10:53]

:END:

- REFERENCIA:
  - Capitulo 1 en "Randomized Algorithms", de Rajeev Motwani and Prabhakar Raghavan.
  - Problema
    * Strategia de adversario no funciona

* En algunos casos, teoria de codigos es suficiente (e.g. busqueda en arreglo ordenado). Eso es una cota inferior sobre el tamano del certificado.

* En otros caso, teoria de codigos no es suficiente. En particular, cuando el precio para verificar un certificado es mas pequeno que de encontrarlo. En estos casos, utilizamos otras tecnicas:

  · teoria de juegos (que vamos a ver) y equilibro de Nash

  · cotas sobre la comunicacion en un sistema de "Interactive Proof"

– Algunas Notaciones Algebraicas

  Sea:

  * A una familia de n_a algoritmos deterministicos

  * a un vector $(0, ..., 0, 1, 0, ..., 0)$ de dimension n_a

  * α una distribucion de probabilidad de dimension n_a

  * B una familia de n_n instancias

  * b un vector $(0, ..., 0, 1, 0, ..., 0)$ de dimension n_b

  * β una distribucion de probabilidad de dimension n_b

  * M una matriz de dimension $n_a \times n_b$ tal que $\mathrm{M}_{\{a,b\}} es el costo del algoritmo a sobre la insta$

    · $a^t M b = M_{a,b}$

    · $\alpha^t M b$ es la complejidad en promedio (sobre el aleatorio del algoritmo α) de α sobre b

    · $a^t M \beta$ es la complejidad en promedio (sobre la distribucion de instancias β) de a sobre β

    · α^t M β es la complejidad en promedio del algoritmo aleatorizados α sobre la distribucion de instancia β.

– von Neuman's theorem: infsup = supinf = minmax = maxmin

1. OPCIONAL Existencia de $\tilde{\alpha}$ et $\tilde{\beta}$

   Dado ϕ y ψ definidas sobre $\mathcal{R}^m$ y $\mathcal{R}^n$ por

   $$\phi(\alpha) = \sup_{\beta} \alpha^T M \beta \text{ y } \psi(\beta) = \inf_{\alpha} \alpha^T M \beta$$

   Entonces:

   * $\phi(\alpha) = \max_\beta \alpha^T M \beta$
   * $\psi(\beta) = \min_\alpha \alpha^T M \beta$
   * hay estrategias mixtas
     - $\tilde{\alpha}$ por A
     - $\tilde{\beta}$ por B
   * tal que
     - ϕ es a su minima en $\tilde{\alpha}$ y
     - ψ es a su maxima en $\tilde{\beta}$.

2. Resultado de von Neuman:

   Dado un juego Γ definido por la matrica \$M\$~:

   $$\min_{\alpha} \max_{\beta} \alpha^T M \beta = \max_{\beta} \min_{\alpha} \alpha^T M \beta$$

3. Interpretacion:

   * Este resultado significa que si consideramos ambos distribuciones sobre algoritmos y instancias, no importa el orden del max o min:
     - podemos elegir el mejor algoritmo (i.e. minimizar sobre los algoritmos aleatorizados) y despues elegir la peor distribucion de instancias para el (i.e. maximizar sobre las distribuciones de instancias), o al reves

· podemos elegir la peor distribucion de instancias (i.e. maximizar sobre las distribuciones de instancias), y considerar el mejor algoritmo (i.e. minimizar sobre los algoritmos aleatorizados) para este distribucion.

* ATENCION!!!! Veamos que

· El promedio (sobre las instancias) de las complejidades (de los algoritmos) en el peor caso

· no es igual

· al peor caso (sobre las instancias) de la complejidad en promedio (sobre el aleatorio del algoritmo)

· donde el segundo termo es realmente la complejidad de un algoritmo aleatorizados.

* Todavia falta la relacion con la complejidad en el peor caso b de un algoritmo aleatorizados α:

$$\max_{b} \min_{\alpha} \alpha^T M b$$

– Lema de Loomis

* Dado una estrategia aleatoria α, emite una instancia b tal que α es tan mal en b que en el peor β.

$$\forall \alpha \exists b, \max_{\beta} \alpha^T M \beta = \alpha^T M b$$

* Dado una distribucion de instancias β, existe un algoritmo deterministico a tal que a es tan bien que el mejor algoritmo aleatorizados α sobre la distribucion de instancias β:

$$\forall \beta \exists a, \min_{\alpha} \alpha^T M \beta = a^T M \beta$$

* Interpretacion:

· En frente a una distribucion de instancias especifica, siempre existe un algoritmo deterministico optima en comparacion con los algoritmos aleatorizados (que incluen los deterministicos).

· En frente a un algoritmo aleatorizados, siempre existe una instan ca tan mal que la pero distribucion de instancias.

– Principe de Yao

* Del leima de Loomis podemos concluir que

$$\max_{\beta} \alpha^T M \beta = \max_{b} \alpha^T M b$$

$$\min_{\alpha} \alpha^T M \beta = \min_{a} a^T M \beta$$

* Del resultado de von Neuman sabemos que maxmin=minmax (sobre α y β):

$$\min_{\alpha} \max_{\beta} \alpha^T M \beta = \max_{\beta} \min_{\alpha} \alpha^T M \beta$$

* Entonces

$$\min_{a} lpha \max_{b} \alpha^T M b = (Loomis) \min_{\alpha} \max_{\beta} \alpha^T M \beta = (vonNeuman) \max_{\beta} \min_{\alpha} \alpha^T M \beta =$$

* Interpretacion

$\min_\alpha$	$\max_b$	$\alpha^T M b$
		La complejidad
del mejor algoritmo aleatorizado		
	en el peor caso	

es igual a

$\max_\beta$ $\min_a$ $\alpha^T M$

La co

del mejor algoritmo deterministico

sobre la peor distribucion de instancias

"El peor caso del mejor algoritmo aleatorizado corresponde a la peor distribucion para el mejor algoritmo deterministico."

* Ejemplos de cotas inferiores:

  - Decidir si un elemento pertenece en una lista ordenadas de tamano n
  - Cual es el peor caso b de un algoritmo aleatorizado α?
  - Buscamos una distribucion β_0 que es mala para todos los algoritmos deterministicos a (del modelo de comparaciones)
  - Consideramos la distribucion uniforma.
  - Cada algoritmo deterministico se puede representar como un arbol de decision (binario) con $2n+1$ hojas.
  - Ya utilizamos para la cota inferior deterministica que la altura de un tal arbol es al menos $\lg(2n+1) \in \Omega(\lg n)$. Esta propiedad se muestra por recurrencia.
  - De manera similar, se puede mostrar por recurrencia que la altura en promedio de un tal arbol binario es al menos $\lg(2n+1) \in \Omega(\lg n)$.
  - Entonces, la complejidad promedio de cada algoritmo deterministico a sobre β_0 es al menos $\lg(2n+1) \in \Omega(\lg n)$.
  - Entonces, utilizando el principie de Yao, la complejidad en el peor caso de un algoritmo aleatorizado en el modelo de comparaciones es al menos $\lg(2n+1) \in \Omega(\lg n)$.

- El corolario interesante, es que el algoritmo deterministico de busqueda binaria es **oprima** a dentro de la clase mas general de algoritmos aleatorizados.

- Decidir si un elemento pertenece en un lista desordenada de tamano k

- Si una sola instancia de la valor buscada ($r = 1$)

- Cotas superiores

- k en el peor caso deterministico

- $(k + 1)/2$ en el peor caso aleatorio

- con una direccion al azar

- con $\lg(k!)$ bits aleatorios

- Cota inferior

- Buscamos una distribucion β_0 que es mala para todos los algoritmos deterministicos a (en el modelo de comparaciones).

- Consideramos la distribucion uniforma (cada algoritmo reordena la instancia a su gusto, de toda manera, entonces solamente la distribucion uniforma tiene sentido): cada posicion es elegida con probabilidad $1/k$

- Se puede considerar solamente los algoritmos que no consideran mas que una ves cada posicion, y que consideran todas las posiciones en el peor caso: entonces cada algoritmo puede ser representado por una permutacion sobre k.

- Dado un algoritmo deterministico a, para cada $i \in [1, k]$, hay una instancia sobre cual el performe i comparaciones. Entonces, su complejidad en promedio en este instancia es $\sum_i i/k$, que es $k(k + 1)/2k = (k+1)/2$. Como eso es verdad para todos los algoritmos deterministicos, es verdad para el mejor de ellos tambien.

- Entonces, utilizando el principio de Yao, la complejidad en el peor caso de un algoritmo aleatorizado en el modelo de comparaciones es al menos $(k+1)/2$.
- Si r instancias de la valor buscada
- Cotas superiores
- $k - r$ en el peor caso deterministico
- $O(k/r)$ en (promedio y en) el peor caso aleatorio
- Cota inferior
- Buscamos una distribucion β_0 que es mala para todos los algoritmos deterministicos a (en el modelo de comparaciones).
- Consideramos la distribucion uniforma (cada algoritmo reordena la instancia a su gusto, de toda manera, entonces solamente la distribucion uniforma tiene sentido): cada posicion es elegida con probabilidad $1/k$
- Se puede considerar solamente los algoritmos que no consideran mas que una ves cada posicion.
- De verdad, no algoritmo tiene de considerar mas posiciones que $k - r + 1$, entonces hay menos algoritmos que de permutaciones sobre k elementos. Para simplificar la prueba, podemos exigir que los algoritmos especifican una permutacion entera, pero no vamos a contar las comparaciones despues que un de las r valores fue encontrada.

- Decidir si un elemento pertenece en k listas ordenadas de tamano n/k
- Cotas superiores
- Si una sola lista contiene la valor buscada
- k busquedas en el peor caso deterministico, que da $k \lg(n/k)$ comparaciones

· $k/2$ busquedas en (promedio y en) el peor caso aleatorio, que da $k \lg(n/k)/2$ comparaciones

· Si $r < k$ listas contienen la valor buscada

· $k - r$ busquedas en el peor caso deterministico, que dan $(k - r) \lg(n/(k - r))$ comparaciones

· k/r busquedas en (promedio y en) el peor caso aleatorio, que dan $(k/r) \lg(n/k)$ comparaciones

· Si $r = k$ listas contienen la valor buscada

· k busquedas en el peor caso deterministico, en promedio y en el peor caso aleatorio, que dan $k \lg(n/k)$ comparaciones

· Aplicacion:

algoritmos de interseccion de listas ordenadas.

· Conclusion:

· Relacion fuerte entre algoritmos aleatorizados y complejidad en

– El peor caso de un algoritmo aleatorio corresponde a la peor distribucion para un algoritmo deterministico.

– Otras aplicaciones importantes de los algoritmos aleatorizados

* "Online Algorithms", en particular paginamiento.

* Algoritmos de aproximacion

* Hashing

Relacion con Problemas NP-Dificiles

- Ejemplos de Problemas NP-Dificiles

1. **Maxcut**

– dado un grafe $G = (V, E)$

- encontrar una partition (L, R) tq $L \cup R = V$ y que **maximiza** la cantidad de aristas entre L y R
- el problema es NP dificil
- se aproxima con un factor de dos con un algoritmo aleatorizado en tiempo polinomial.

2. **mincut**
   - dado un grafe $G = (V, E)$
   - encontrar una partition (L, R) tq $L \cup R = V$ y que minimiza la cantidad de aristas entre L y R
   - el problema es NP dificil
   - Relacion con la nocion de NP:
     * El arbol representando la ejecucion de un algoritmo

- una parte de **decisiones** non-deterministica (fan-out)
- una parte de **verificacion** deterministica (straight)
- Si una solamente de las $2^{p(n))}$ soluciones corresponde a una solucion valida del problema, la aleatorizacion no ayuda, pero si una proporcion constante (e.g. 1/2,1/3,1/4,...) de las ramas corresponden a una solucion correcta, existe un algoritmo aleatorizado que resuelve el problema NP-dificil en tiempo polynomial **en promedio**.

Complejidad de un algoritmo aleatorizado

- Considera algoritmos con comparaciones
  - algoritmos deterministicos se pueden ver como arboles de decision.
  - algoritmos aleatorios se pueden ver (de manera intercambiable) como
    * una distribucion sobre los arboles de decision,

        * un arbol de decision con algunos nodos "aleatorios".

    - La complejidad en una instancia de un algoritmo aleatorio es el promedio de la complejidad (en este instancia) de los algoritmos deterministicos que le compasan:

      $\mathrm{C}((\mathrm{A}_r)_r, I) = E_r(\ \mathrm{C}(\mathrm{A}_r,\mathrm{I})\ )$

    - La complejidad en el peor caso de un algoritmo aleatorio es el peor caso del promedio de la complejidad de los algoritmos deterministicos que le composan:

      $\mathrm{C}((\mathrm{A}_r)_r) = \max_I\ \mathrm{C}((\mathrm{A}_r)_r, I) = \max_I\ \mathrm{E}_r(\ \mathrm{C}(\mathrm{A}_r,\mathrm{I})\ )$

1.5.3 Algoritmos tipo Monte Carlo y Las Vegas

- Formalizacion

Algoritmos clasicos	Non clasicos
Siempre hacen lo mismo	aleatorizados
Nunca se equivocan	Monte Carlo
Siempre terminan	Las Vegas

- Clasificacion de los algoritmos **de decision** aleatorizados

  1. Probabilistico: Monte Carlo

     - $P(error) < \epsilon$
     - Ejemplo:

       * deteccion de cliquas

     - Se puede considerar tambien variantes mas fines:

       * two-sided error (=> clase de complejidad BPP)

         · $P(accept|negative) < \epsilon$

         · $P(refuse|negative) > 1 - \epsilon$

         · $P(accept|positive) > 1 - \epsilon$

· $P(refuse|positive) < epsilon$

* One-sided error

· $P(accept|negative) < epsilon$

· $P(refuse | negative) > 1 - \epsilon $

· $P(accept|positive) = 1$

· $P(refuse|positive) = 0$

2. Probabilistico: Las Vegas

   – Tiempo es una variable aleatoria

   – Ejemplos:

   * determinar primalidad [Miller Robin]

   * busqueda en arreglo desordenado

   * interseccion de arreglos ordenados

   * etc...

   * Relacion

   · Si se puede verificar el resultado en tiempo razonable,

1.5.4 Primalidad

- REFERENCIAS:

  – Primalidad: Capitulo 14.6 en "Randomized Algorithms", de Rajeev Motwani and Prabhakar Raghavan.

  – http://en.wikipedia.org/wiki/Primality$_testComplexity$

  – Algoritmo "Random Walks" para SAT

  * eliga cualquieras valores $x_1, \ldots, x_n$

    * si todas las clausilas son satisfechas,
        · accepta
    * sino
        · eliga una clausula non satisfecha (deterministicamente o no)
        · eliga una de las variables de esta closula.
    * Repite es r veces.
    * PRIMES is in coNP

- PRIMES is in NP (hence in NP$\cap$ coNP)

  In 1975, Vaughan Pratt showed that there existed a certificate for primality that was checkable in polynomial time, and thus that PRIMES was in NP, and therefore in NP coNP.

- PRIMES in coRP

  The subsequent discovery of the Solovay-Strassen and Miller-Rabin algorithms put PRIMES in coRP.

- PRIMES in $ZPP = RP \cap coRP$

  In 1992, the Adleman-Huang algorithm reduced the complexity to $ZPP = RP \cap coRP$, which superseded Pratt's result.

- PRIMES in QP

  The cyclotomy test of Adleman, Pomerance, and Rumely from 1983 put PRIMES in QP (quasi-polynomial time), which is not known to be comparable with the classes mentioned above.

- PRIMES in P

  Because of its tractability in practice, polynomial-time algorithms assuming the Riemann hypothesis, and other similar evidence, it was long suspected but not proven that primality could be solved in polynomial time. The existence of the AKS primality test finally settled this long-standing question and placed PRIMES in P.

- $PRIMES \in NC?$ $PRIMES \in L?$

  PRIMES is not known to be P-complete, and it is not known whether it lies in classes lying inside P such as NC or L.

1.5.5 Clases de complejidad aleatorizada :BONUS:

RP

- "A **precise** polynomial-time bounded nondeterministic Turing Machine", aka "maquina de Turing non-deterministica acotadada polinomialemente *precisa*", es una maquina tal que su ejecucion sobre las entradas de tamano n toman tiempo p(n) **todas**.
- "A *polynomial Monte-Carlo Turing machine*", aka una "*maquina de Turing de Monte-Carlo* polynomial" para el idioma L, es una tal maquina tal que
  - si $x \in L$, al menos la mitad de los $2^{p(|x|)}$ caminos acceptan x
  - si $x \notin L$, **todas** los caminos rechazan x.
- La definicion corresponde exactamente a la definicion de algoritmos de Monte-Carlo. La classe de idiomas reconocidos por una **maquina de Turing de Monte-Carlo** polynomial es RP.
- $P \subset BP \subset NP$:
  - un algoritmo en P accepta con todos sus caminos cuando una palabra x es en L, que es "al menos" la mitad.
  - un algoritmo en NP accepta con al menos un camino: un algoritmo en RP accepta con al menos la mitad de sus caminos.

ZPP

- $ZPP = RP \cap coRP$
- $Primes \in ZPP$

PP

- Maquina que, si $x \in L$, accepta en la mayoridad de sus entradas.
- PP probablemente no en NP
- PP probablemente no en RP

BPP

$$BPP = \left\{ L, \forall x, \left\{ \begin{array}{l} x \in L \Rightarrow 3/4 \text{ de los caminos acceptan } x \\ x \notin L \Rightarrow 3/4 \text{ de los caminos rechazan } x \end{array} \right\} \right.$$

$RP \subset BPP \subset PP$
$BPP = coBPP$

1.5.6 Nociones de aproximabilidad (2 semanas = 4 charlas)

:LOGBOOK:

- State "DONE" from "" [2010-10-07 Thu 10:50]

:END:

- problemas que son o no aproximables

Intro: Aproximacion de problemas de optimizacion NP-dificiles

- Que problemas NP dificiles conocen?
  - colorisacion de grafos
  - ciclo hamiltonian
  - Recubrimiento de Vertices (Vertex Cover)
  - Bin Packing
  - Problema de la Mochila
  - Vendedor viajero (Traveling Salesman)
- Que hacer cuando se necessita una solucion en tiempo polinomial?
  - Consideramos los problemas NP completos de decision, generalmente de optimizacion. Si se necesita una soluciono en tiempo polinomial, se puede considerar una aproximacion.

$p(n)$-aproximacion: Definicion

- Dado un problema de minimizacion, un algoritmo A es un **$p(n)$-aproximacion** si $\forall n \max \frac{C_A(x)}{C_{OPT}(x)} \leq p(n)$
- Dado un problema de maximizacion, un algoritmo A es un **$p(n)$-aproximacion** si $\forall n \max \frac{C_{OPT}(x)}{C_A(x)} \leq p(n)$
  - Notas:
    1. Aqui consideramos la cualidad de la solucion, NO la complejidad del algoritmo. Usualmente el problema es NP-dificil, y el algoritmo de aproximacion es de complejidad polinomial.
    2. Las razones estas ≥ 1 (elijamos las definiciones para eso)
    3. A veces consideramos tambien $C_A(x) - C_{OPT}(x)$ (minimizacion) y $C_{OPT}(x) - C_A(x)$: eso se llama un "esquema de aproximacion polinomial" y vamos a verlo mas tarde.

Ejemplo: Bin Packing (un problema que es 2-aproximable)

- DEFINICION

  Dado n valores $x_1, \ldots, x_n$, $0 \leq x_i \leq 1/2$, cual es la menor cantidad de cajas de tamano 1 necesarias para empaquetarlas?
- Algoritmo "Greedy"
  - Considerando los x_i en orden,
  - llenar la caja actual todo lo posible,
  - pasar ala siguiente caja.
- Analisis
  - Este algoritmo tiene complejidad lineal $O(n)$.
  - Greedy da una 2-aproximacion.

- (se puede mostrar facilamente en las instancias donde el algoritmo optima llene completamente todas las cajas.)

- Ademas, hay mejores aproximaciones,

  1. Best-fit tiene performance ratio de 1.7 en el peor caso

with Random Order (1997), Kenyon Best-fit is the best known algorithm for on-line binpacking, in the sense that no algorithm is known to behave better both in the worst case (when Best-fit has performance ratio 1.7) and in the average uniform case, with items drawn uniformly in the interval [0; 1] (then Best-fit has expected wasted space O(n 1=2 (log n) 3=4)). In practical applications, Best-fit appears to perform within a few percent of optimal. In this paper, in the spirit of previous work in computational geometry, we study the expected performance ratio, taking the worst-case multiset of items L, and assuming that the elements of L are inserted in random order, with all permutations equally likely. We show a lower bound of 1:08 : : : and an upper bound of 1:5 on the random order performance ratio of Best-fit. The upper bound contrasts with the result that in the worst case, any (deterministic or randomized) on-line bin-packing algorithm has performance ratio at least 1:54 : : :. 1

  2. Un otro paper donde particionan los x_i en tres classes, placeando los x_i mas grande primero, buscando el placamiento optimal de los x_i promedio, y usando un algoritmo greedy para los x_i pequenos. [Karpinski?]

  3. la distribucion de los tamanos de las cajas hacen la instancia dificil o facil.

Ejemplo: Recubrimiento de Vertices (Vertex Cover)

- DEFINICION:

  Dado un grafo $G = (V, E)$, cual es el menor $V' \subseteq V$ tal que V' sea un vertex cover de G. (o sea V' mas pequeno tq \$$\forall$ e=(u,v)$\in E, u \in V' ov \in V') Algoritmodeaproximacion$:

- – $V' \leftarrow \emptyset$

  – while $E \neq \emptyset$

        * sea $(u, v) \in E$
        * $V' \leftarrow V' \cup u, v$
        * $E \leftarrow E\ (x, y), x = uox = voy = uoy = v$
    - return V
- Discusion: es una 2-aproximacion o no?
- LEMA: El algoritmo es una 2-aproximacion.
    - PRUEBA:
        * Cada par u, v que la aproximacion elige esta conectada, entonces u o v estas en cualquier soluciono optima de Vertex Cover.
        * Como se eliminan las aristas incidentes en u y v, los siguientes pares que se eligen no tienen interseccion con el actual, entonces cada 2 nodos que el algoritmo elige, uno pertenece a la solucion optima.
        * quod erat demonstrandum, (QED).

Ejemplo: Vendedor viajero (Traveling Salesman)

- DEFINICION:

  Dado $G = (V, E)$ dirigido y $C : E \rightarrow R^+$ una funcion de costos, encontrar un recorrido que toque cada ciudad una vez, y minimice la suma de los costos de las aristas recorridas.
- LEMA: Si c satisface la desigualdad triangular $\forall x, y, zc(x, y)+x(y, z) \geq c(x, z)$, hay una 2-aproximacion.
    - PROOF:
        * Algoritmo de Aproximacion (con desigualdad triangular)
            · construir un arbol cobertor minimo (MST)

· se puede hacer en tiempo polinomial, con programacion lineal.

· $C_{MST} \leq C_{OPT}$

· producimos un recorrido en profundidad DFS del MST:

· $C_{DFS} = 2C_{MST} \leq 2C_{OPT}$

· (factor dos porque el camino de vuelta puede ser al maximo de tamano igual al tamano del camino de ida)

· eliminamos los nodos repetidos del camino, que no crece el costo

· $C_A \leq 2C_{OPT}$

* quod erat demonstradndum, QED.

- LEMA: Si c no satisface la desigualdad triangular, el problema de vendedor viajero no es aproximables en tiempo polinomial (a menos que P=NP$).

  – PROOF:

    * Supongamos que existe una p(n)-aproximacion de tiempo polinomial.

    * Dado un grafo G=(V,E)

    * Construimos un grafo G'=(V,E') tal que

      · E'=V^2 (grafo completo), y

    · c(u,v) = 1 si (u,v)$\in$ $Enp(n)sino$

    * Si no hay un Ciclo Hamiltonian en E,

      · todas las soluciones usan al menos una arista que no es en E

· entonces todas las soluciones tienen costo mas que $np(n)$

* Si hay un Ciclo Hamiltonian en E

· tenemos una aproximacion de una soluciono de costo menos que $(n-1)p(n)$ QED

Ejemplo: Vertex Cover con pesos

- DEFINICION

  Dado G=(V,E) y c:V^+, $sequiereunV^* \subseteq VquecubraEyqueminimice\sum_{\{}v \in V^*\}c(v).EsteproblemaesNPCompleto.$

- LEMA: Vertex Cover con pesos es 2-aproximable
- PROOF:

1. Sea variables $x(v) \in 0, 1, \forall v \in V$
   - el costo de V^* sera $\sum x(v)c(v)$
   - objetivo: $\min \sum_{v \in V} x(v)c(v)$ donde $0 \leq x(v) \leq 1 \forall v \in V$
   - $x(u) + x(v) \geq 1 \forall u, v \in E$
2. $x(v) \in Z$ sere programacion entera, que es NP Completa $x(v) \in R$ sere programacion lineal, que es polinomial el valor $\sum x(v)c(v)$ que produce el programo lineal es inferior a la mejor solucion al problema de VC.
3. Algoritmo
   - resolver el problema de programacion lineal
     * nos da $x(v_1), \ldots, x(v_n)$
   - $V^* \leftarrow \emptyset$
   - for $v \in V$

        * si $x(v) \geq 1/2$
            · $V^* \leftarrow V^* \cup v$
    - return V^*
4. Propiedades
    - V^* es un vertex cover:
        * si $\forall (u,v) \in E, x(u) + x(v) \geq 1$
        * entonces, $x(u) \geq 1/2ox(v)/geq1/2$
        * entonces, $u \in V^*ov \in V^*$
    - V^* es una 2-aproximacion:
        * $c(V^*) = \sum_v y(v)c(v)$ donde $y(v) = 1six(v) \geq 1/2$, y 0 sino
        * entonces $y(v) \leq 2x(v)$
        * $\sum y(v)c(v) \leq \sum 2x(v)c(v) \leq 2OPT$
        * $C(V^*) \leq 2OPT$
5. QED

PTAS y FPTAS: Definiciones

- Esquema de aproximacion poliniomial **PTAS** Un **esquema de aproximacion polinomial** para un problema es un algoritmo A que recibe como input una instancia del problema y un para-metro $\varepsilon > 0$ y produce una $(1+\varepsilon)$ aproximacion. Para todo ε fijo, el tiempo de A debe ser polinomial en n, el tamano de la instancia.
- Ejemplos de complejidades: Cuales tienen sentidos?
    - [ ] $O(n^{2/3})$
    - [ ] $O(\frac{1}{\varepsilon^2}n^2)$
    - [ ] $O(2^{\varepsilon}n)$
    - [ ] $O(2^{1/\varepsilon}n)$
    - Esquema de aproximacion completamente polinomial **FPTAS**

Ejemplo: Problema de la Mochila

- Definicion
  - Dado
    - n pesos $p_1, \ldots, p_n \geq 0$,
    - n valores $v_1, \ldots, v_n \geq 0$,
    - un peso total maximo P
  - Queremos encontrar $S \subset [1..n[$ tal que
    - $\sum_{i \in S} p_i \leq P$
    - $\sum_{i \in S} v_i$ sea maximal.
- Solucion Exacta
  - Dado $L = \{y_1, , \ldots y_m\}$,
    - definimos $L + x = \{y_1 + x, \ldots, y_m + x\}$.
  - Algoritmo:
    - $L \leftarrow \{\emptyset\}$
    - for $i \leftarrow 1$ to n
      - $L \leftarrow$ merge $(L, L + x_i)$
      - prune(L, P) (remudando las valores $> P$)
    - return max(L)
- Solucion aproximada (inspirada del algoritmo exacto)
  - Operacion "Recorte"

* Definimos la operacion de **recorte** de una lista L con parametro δ:
    - Dado $y, z \in L$, z **represente** a y si
    - $y/(1+\delta) \leq z \leq y$
* vamos a eliminar de L todos los y que sean representados por alguno z no eliminado de L.
* Recortar(L, δ)
    - Sea $L = \{y_1, \ldots, y_m\}$
    - $L' \leftarrow \{y_1\}$
    - $Last \leftarrow y_1$
    - for $i \leftarrow 2$ to m
    - si $y_i > Last(1+\delta)$
    - $L \leftarrow L'.\{y_i\}$
    - $Last \leftarrow y_i$
    - return L'

– Algoritmo de Aproximacion:

* $L \leftarrow \{\emptyset\}$
* for $i \leftarrow 1$ to n
    - $L \leftarrow merge(L, L + x_i)$
    - $prune(L, t)$ (remudando las valores $> t$)
    - **$L** $\leftarrow recortar(L,\epsilon/$**2n)$$**
* return max(L)

- Analysis
    - El resultado es una $(1+\epsilon)$-aproximacion:
        1. retorne una solucion valida, tal que
            - $\sum(S') \leq t$ para algun $S' \subset S$
        2. en el paso i, para todo $z \in L_{OPT}$, existe un $y \in L_A$ tal que z representa a y.
            - Luego de los n pasos, el z^* optimo en L_{OPT} tiene un representante $y^* \in L_A$ tal que

$$z^*/(1+\epsilon/2n)^n \leq y^* \leq z^*$$

        3. Para mostrar que el algoritmo es una $(1+\epsilon)$ aproximacion,
            - hay que mostrar que
            - $z^*/(1+\epsilon) \leq y^*$
            - entonces, debemos mostrar que
            - $(1+\epsilon/2n)^n \leq 1+\epsilon$
            - Eso se muestra con puro calculo:
            - $(1+\epsilon/2n)^n \leq ?1+\epsilon$
            - $e^{n\lg(1+\epsilon/2n)}$
            - $\leq e^{n\epsilon/2n}$
            - $= e^{\epsilon/2}$
            - $\leq ?e^{\ln(1+\epsilon)}$
            - eso es equivalente a elegir ϵ tal que $\epsilon/2 \leq \ln(1+\epsilon)$

- i.e. cualquier tal que $0 < \epsilon \leq 1$

4. El algoritmo es polinomial (en todos los parametros)

   - despues de recortar dedos $y_i, y_{i+1} \in L$ se cumple $y_{i+1} > y_i(1+\delta)$ y el ultimo elemento es $\leq t$
   - entonces, la lista contiene $0, 1$ y luego a lo mas $\lfloor \log_{(1+\delta)} t \rfloor$
   - entonces el largo de L en cada iteracion no supera $2 + \frac{\log t}{\log(1+\varepsilon/2n)}$
   - Nota que $\ln(1+x)$
   - $= -\ln(1/(1+x))$
   - $= -\ln((1+x-x)/(1+x))$
   - $= -\ln(1-x/(1+x))$
   - $= -\ln(1+y) \geq -y$
   - $\geq -(-x/(1+x)) = x/(1+x)$

   - Entonces
   - $2 + \frac{\ln t}{\ln 1+\epsilon/2n}$
   - $\leq 2 + ((1+\epsilon/2n)2n \ln t)/\epsilon$
   - $= (2n \ln t)/\epsilon + \ln t + 2$
   - $= O(n \lg t/\epsilon)$

   - Entonces cada iteracion toma $O(n \lg t/\epsilon)$ operaciones
   - Las n iteraciones en total toman
   - $O(n^2 \lg t/\epsilon)$ operaciones

1.5.7 Algoritmos paralelos y distribuidos (2 semanas = 4 charlas)

- Medidas de complejidad
- Tecnicas de diseno

REFERENCIA: Chap 12 of "Introduction to Algorithms, A Creative Approach", Udi Manber, p. 375

REFERENCIA: http://www.catonmat.net/blog/mit-introduction-to-algorithms-part-thirteen/

Modelos de paralelismo

- Instrucciones
  - SIMD: Single Instruccion, Multiple Data
  - MIMD: Multiple Instruccion, Multiple Data
  - Memoria
    * compartida
    * distribuida
    * 2*2 combinaciones posibles:

	Memoria compartida	Memoria distribuida
SIMD	PRAM	redes de interconexion (weak computer units) (hipercubos, meshes, etc...)
MIMD	Threads	procesamiento distribuido (strong computer units), Bulk Synchronous Process, etc...

    * En este curso consideramos en particular el modelo PRAM

Modelo PRAM

1. Mucha unidad de CPU, una sola memoria RAM

   cada procesador tiene un identificador unico, y puede utilizarlo en el programa

2. Ejemplo:

   - if p mod 2 = 0 then
     - A[p] += A[p-1]
   - else
     - A[p] += A[p+1]
   - b $\leftarrow A[p]; A[p] \leftarrow b;$

- Problema: el resultado no es bien definido, puede tener

3. EREW Exclusive Read. Exclusive Write
4. CREW Concurrent Read, Exclusive Write
5. CRCW Concurrent Read, Concurent Write En este caso hay variantes tambien:

   - todos deben escribir lo mismo
   - arbitrario resultado
   - priorizado
   - alguna f() de lo que se escribe

Como medir el "trade-off" entre recursos (cantidad de procesadores) y tiempo?

- DEFINICION:
  - $T^*(n)$ es el **Tiempo secuencial** del mejor algoritmo no paralelo en una entrada de tamano n (i.e. usando 1 procesador).
  - $T_A(n, p)$ es el **Tiempo paralelo** del algoritmo paralelo A en una entrada de tamano n usando p procesadores.

- El **Speedup** del algoritmo A es definido por

  $S_A(n,p) = \frac{T^*(n)}{T_A(n,p)} \leq p$

  Un algoritmo es mas efectivo cuando $S(p) = p$, que se llama **speedup perfecto**.

- La **Eficiencia** del algoritmo A es definida por

  $E_A(n,p) = \frac{S_A(n,p)}{p} = \frac{T^*(n)}{pT_A(n,p)}$

  El caso optima es cuando $E_A(n,p) = 1$, cuando el algoritmo paralelo hace la misma cantidad de trabajo que el algoritmo secuencial. El objetivo es de **maximizar la eficiencia**.

  (Nota estas definiciones en la pisara, vamos a usarlas despues.)

PROBLEMA: Calcular Max(A[1,. . . ,N])

1. Solucion Secuencial

   - Algoritmo:

     - $m \leftarrow 1$
     - for $i \leftarrow 2$ to n

       $*$ if $A[i] > A[m]$ then $m \leftarrow i$

     - return $A[m]$
     - Se puede ver como un arbol de evaluacion con una sola rama

   - Complejidad:

     - tiempo $O(n)$, con 1 procesador, entonces:
     - $T^*(n) = n$.

2. Solucion Parallela con n procesadores

   - Algoritmo:

- $M[p] \leftarrow A[p]$
- for $l \leftarrow 0$ to $\lceil \lg p \rceil - 1$
    - if $p2^{l+1} = 0$ y $ p+2^l<n$
        - $M[p] \leftarrow \max(M[p], M[p + 2^l])$
- if $p = 0$
    - $max \leftarrow M[2]$
    - Se puede ver como un arbol balanceado de altura $\lg n$ con

- Complejidad:
    - tiempo $O(\lg n)$ con n procesador, i.e. en nuestra notaciones:
    - $T(n, n) = \lg n$
    - $S(n,n) = n\frac{}{\lg n$$E(n,n)=\frac{n}{n \lg n}=\frac{1}{\lg n}$}
    - Nota: no se puede hacer mas rapido, pero hay mucho

3. Solucion general con p procesadores

- Idea:
    - reduce la cantidad de procesadores, y hace "load balancing" sobre $n/\lg n$ procesadores.
    - Divida el input en $n/\lg n$ grupos,
    - asigna cada grupo de $\lg n$ elementos a un procesador.
    - En la primera fase, cada procesador encontra el max de su grupo
    - En la segunda fase, utiliza el algoritmo precedente.

- Complejidad:
    - tiempo $O(\lg n)$ con n procesador, i.e. en nuestra notaciones:
        - $T(n, \frac{n}{\lg n}) = 2 \lg n \in O(\lg n)$
        - $T(n,p) = \frac{n}{p} + \lg p$
        - $S(n,p) = \frac{n}{\frac{n}{p}+\lg p} = p(1 - \frac{p \lg p}{n + p \lg p}) \to p$ si $n \to \infty$ y p
    - \$E(n,p) = n$\frac{}{\frac{n}{\lg n} \lg n = 1/2\$Discusion:}$

- El parametro de $\lg n$ procesadores es optima?
    - Para que? Que significa ser optima?
        - en energia
        - en el contexto donde los procesadores libres pueden ser usados para otras tareas.
    - Si, es optimo para la eficiencia, se puede ver estudiando el grafo en funcion de p.

- Eso es un algoritmo EREW, CREW, o CRCW?
    - EREW (Exclusive Read. Exclusive Write): no dos procesadores lean o escriben en la misma cedula al mismo tiempo.

- Nota:
    - Hay un algoritmo CRCW que puede calcular el max en $O(1)$ tiempo en paralelo, ilustrando el poder del modelo CRCW (y el costo de las restricciones del modelo EREW) [REFERENCIA: Section 12.3.2 of "Introduction to Algorithms, A Creative Approach", Udi Manber, p. 382]]

LEMMA de Brent, Trabajo y Consecuencias

LEMA de Brent El algoritmo previo illustra un principo mas general, llamado el "Lemma de Brent":

Si un algoritmo

- consigue un tiempo T(n,p)=C, entonces
- consigue tiempo T(n,p/s)= sC $\forall$ s>1
- (bajo algunas condiciones, tal que hay suficientamente memoria para cada procesador)

DEFINICION "Trabajo"

- Usando el Lema de Brent, podemos exprimir el rendimiento de los algoritmos paralelos con solamente dos medidas:
  - $T(n)$, el tiempo del mejor algoritmo paralelo usando cualquier cantidad de procesadores que quiere.

    Nota las diferencias con
    - $T^*(n)$, el tiempo del mejor algoritmo secuencial, y
    - $T_A(n,n)$, el tiempo del algoritmo A con n procesadores.
  - $W(n)$, la suma del total trabajo ejecutado por todo los procesadores (i.e. superficia del arbol de calculo, a contras de su altura (tiempo) o hancho (cantidad de procesadores).
- INTERACCION: Cual son estas valores para el algoritmo de Max?
  - $T(n)$ =?
  - $W(n)$ =?
- INTERACCION: Puedes ver como desde $T(n)$, $W(n)$ se puede deducir las valores de
  - $T(n,p)$? (solucion en el corolario)
  - $S(n,p)$? (trivial desde $T(n,p)$)
  - $E(n,p)$? (solucion en el corolario)
    - *

COROLARIO

- Con el lema de Brent podemos obtener:
  - $$T(n,p) = T(n) + \frac{W(n)}{p}$$
  - $$E(n,p) = \frac{T^*(n)}{pT(n) + W(n)}$$

EJEMPLO

- Para el calculo del maximo:
  - $T(n) = \lg n$
  - $W(n) = n$
  - Entonces
    - se puede obtener
      - $T_B(n,p) = \lg n + \frac{n}{p}$
      - $$E(n,p) = \frac{n}{p \lg n + n}$$
      - (Nota que eso es solamente una cota superior, nuestro

PROBLEMA: Ranking en listas

1. DEFINICION
   - dado una lista, calcula el rango para cada elemento.
   - En el caso de una lista tradicional, no se puede hacer mucho mejor que lineal.
   - Consideramos una lista en un arreglo A,

- donde cada elemento $A[i]$ tiene un puntero al siguiente elemento, $N[i]$, y
- calculamos su rango $R[i]$ en un arreglo R.

2. DoublingRank()
   - $R[p] \leftarrow 0$
   - if $N[p] = null$
     - $R[p] \leftarrow 1$
   - for $d \leftarrow 1$ to $\lceil \lg n \rceil$
     - if $N[p] \neq NULL$
       - if $R[N[p]] > 0$
         - $R[p] \leftarrow R[N[p]] + 2^d$
       - $N[p] \leftarrow N[N[p]]$
3. Analisis
   - $T(n) = \lg n$
   - $W(n) = n + W(n/2) \in O(n)$
   - $T(n,p) = T(n) + W(n)/p = \lg n + n/p$
   - $p^*\ T(n) = W(n)/p^*\ pn/\lg n$
   - $E(n,p^*) = \frac{T^*(n)}{p^*T(n)+W(n)} = \frac{n}{n/\lg n \lg n+n} \in \Theta(1)$
4. El algoritmo es EREW o CREW?
   - es EREW si los procesadores estan sincronizados, com en RAM aqua.

PROBLEMA: Prefijos en paralelo ("Parallel Prefix")

- DEFINICION: Problema "Prefijo en Paralelo"
  Dado $x_1, \ldots, x_n$ y un operador asociativo $\times$, calcular
  - $y_1 = x_1$
  - $y_2 = x_1 \times x_2$
  - $y_2 = x_1 \times x_2 \times x_3$
  - ...
  - $y_n = x_1 \times \ldots \times x_n$
  - Solucion Secuencial

Solucion paralela 1

- Concepto:
  - Hipotesis: sabemos solucionarlo con $n/2$ elementos
  - Caso de base: $n = 1$ es simple.
  - Induccion:
    - recursivamente calculamos en paralelo:
      - todos los prefijos de $\{x_1, \ldots, x_{n/2}\}$ con $n/2$ procesadores.
      - todos los prefijos de $\{x_{n/2}, \ldots, x_n\}$ con $n/2$ procesadores.
    - en paralelo agregamos $x_{n/2}$ a los prefijos de $\{x_{n/2}, \ldots, x_n\}$
    - Observacion: en cual modelo de parallelismo es el ultimo paso?
    - ParallelPrefix1(i,j)
  - if $i_p = j_p$

        * return $x_{\{i_p\}}$

    - $m_p \leftarrow \lfloor \frac{i_p + j_p}{2} \rfloor$;
    - if $p \leq m$ then

        * algo(i_p, m_p)

    - else

        * algo(m+1, j_p)
        * $y_p \leftarrow y_m \cdot y_p$
        * Otra forma de escribir el algoritmo (de p. 384 de

- ParallelPrefix1(left,right)

    - if $(right - left) = 1$

        * $x[right] \leftarrow x[left].x[right]$

    - else

        * $middle \leftarrow (left + right - 1)/2$
        * do in paralel

            · ParallelPrefix1(left,middle) {assigned to ${$P_1$ to $P_{\{n/2\}}$$}$}$ParallelPrefix1(mid$
            $1, right)\{assignedto$\${$P_{\{n/2+1\}}toP_n$\}\$\}$

        * for $i \leftarrow middle + 1$ to $right$ do in paralel

            · $x[i] \leftarrow x[middle].x[i]$

            · Notas:

            · este solucion **no** es EREW (Exclusive Read and Write), porque

- este soluciono es CREW (Concurrent Read, Exclusive Write).
- Complejidad:
  - $T_{A_1}(n,n) = 1 + T(n/2, n/2) = \lg n$

    (El mejor tiempo en paralelo con cualquier cantidad de procesadores.)
  - $W_{A_1}(n) = n + 2W_{A_1}(n/2) - n \lg n$
  - $T_{A_1}(n,p) = T(n) + W_{A_1}(n)/p = \lg n + (n \lg n)/p$
  - Calculamos p^*, la cantidad optima de procesadores para minimizar el tiempo:
    - $T(n) = W_{A_1}(n)/p^*$
    - $p^* = \frac{W(n)}{T(n)} = n$
  - Calculamos la eficiencia
    - \$E_{A_1}(n,p*)=$\frac{T^*(n)}{p^*T(n)+W(n)}=\frac{n}{n\lg n}$={1/lg n}\$$espocoeficiente$=(

* Podriamos tener un algoritmo con
  - la eficiencia del algoritmo secuencial
  - el tiempo del algoritmo paralelo?

Solucion paralela 2: mismo tiempo, mejor eficiencia

- Idea:

  El concepto es de dividir de manera diferente: par y impar (en vez de largo o pequeno).
- Concepto:

  1. Calcular en paralelo $x_{2i-1}.x_{2i}$ en x_{2i} para $\forall i, 1 \leq i \leq n/2$.
  2. Recursivamente, calcular todos los prefijos de $E = \{x_2, x_4, \ldots, x_{2i}, \ldots, x_n\}$

3. Calcular en paralelo los prefijos en posiciones impares, multiplicando los prefijos de E por una sola valor.

- algo2(i,j)
  - for $d \leftarrow 1$ to $(\lg n) - 1$
    - if $p = 02^{d+1}$
      - if $p + 2^d < n$
      - $x_{p+2^d} \leftarrow x_p.x_{p+2^d}$
  - for $d \leftarrow 1$ to $(\lg n) - 1$
    - if $p = 02^{d+1}$
      - if $p - 2^d > 0$
      - $x_{p-2^d} \leftarrow x_{p-2^d}.x_p$

- visualizacion

1.

0,1

2.

2,3 [0,3]

3.

4,5

4.

6,7 [4,7] [0,7]

5.

8,9

10,11 [8,11]

-

12,13

-

14,15 [12,15] [8,15] [0,15]

- Notas:
  - Este algoritmo es EREW
- Analisis
  - $T_2(n) = 2 \lg n$
  - $W(n) = n + W(n.2) = n$
  - $T_2(n,p) = T(n) + W(n)/p = 2 \lg n + \frac{n}{p}$
  - Calculamos la cantidad optima de procesadores para obtener el tiempo optima:
    - $T_2(n) = W(n)/p^*$ entonces
    - $p^* = \frac{W(n)}{T(n)} = \frac{n}{\lg n}$
  - $E_2(n,p^*) = \frac{T^*(n)}{p^* T(n) + W(n)} \in O(1)$

Conclusion del Parallelismo:

1. Cual es la consecuencia del Lemma de Brent?
   - Concentrarse en $T(n)$ y $E(n)$
   - Pero saber aplicar el Lemma de Brent para programmar $T(n,p)$
2. Cual (otra) tecnica veamos?
   - Mejorar la efficiencia con una parte secuencial (en parallelo) del algoritmo.

1.5.8 Conclusion Unidad

- Vimos
  - Aleatorizacion
  - Aproximabilidad
  - Paralelizacion / Distribucio
  - Contexto
    - son **extenciones** del contenido del curso
    - hay muchas otras
      - cryptografia
      - quantum computing
      - parameterized complexity
      - ...
    - La metodologia entre todas tiene una parte en comun:
      - Formalismo
      - Cotas superiores
      - Cotas inferiores
      - Adecuacion a la practica.

1.6 CONCLUSION del curso

- Unidades:
  - Conceptos basicos
  - Memoria Secundaria
  - Tecnicas Avanzadas

- Extensiones
- Temas
    * Implementacion y Experimentacion.
    * cotas inferiores
        · lemma del ave
        · minimo y maximo de un arreglo
        · busqueda en un arreglo con distintas probabilidades de acceso
        · lemma del minimax (para aleatorizacion)
        · theorema de Yao-von Neuman
    * analisis
        · en promedio
        · de algoritmos deterministicos
        · de algoritmos aleatorizados
        · “adaptativa” para
        · torre de Hanoi y “Disk Pile” problema
        · busqueda ordenada (y codificacion de enteros)
        · problemas en linea
        · en Memoria Externa
        · Diccionarios
        · Colas de Prioridades
        · Ordenamiento

- en dominio discreto y finito (afuera del modelo de comparacion)
- inter/extra polacion
- skiplists
- hash
- radix sort

- de algoritmos en linea ("online")
- "competitive analysis"

- de esquemas de aproximacion
- "bin packing"
- "Vertex Cover"
- "Traveling Salesman"
- "Backpack"

- amortizada
- enumeracion de enteros en binario
- min max

- en el modelo PRAM
- max
- ranking en listas
- prefijos

^{2} FOOTNOTE DEFINITION NOT FOUND: 0