Algoritmos de Ordenación y Búsqueda

una vista ligera

Dr. Arno Formella

Departamento de Informática
Universidad de Vigo

21-22 de Mayo de 2009

Ordenar y buscar I

Dibujos y animaciones

Estas transparencias se acompaña con dibujos y animaciones de algoritmos en la pizarra en tiempo real.

¿Quién soy?

- Arno FORMELLA (alemán)
- hablo: castellano (cambiamos al inglés? o alemán?)
- desde 2000: Universidad de Vigo, España
  Departamento de Informática
  Escuela Superior de Ingeniería Informática (Ourense)
- antes: Universität des Saarlandes, Alemania

¿Qué es ordenar?

- necesitamos objetos distinguibles
- con una característica comparable
- de forma transitiva,
- por ejemplo: números $\{1, 2, 3, 4, 5, \ldots\}$, $\leq$
- $a \leq b$ y $b \leq c$ entonces $a \leq c$
- asumimos una relación total
- ¿Qué se puede hacer si la relación es parcial?

¿Por qué ordenamos?

- para tener un acceso más rápido a los datos, por ejemplo, con búsqueda binaria
- para encontrar el mediano o la pareja más próxima
- para identificar casos excepcionales estadísticamente
- para encontrar duplicados

¿Dónde se ordena también?

- algoritmos de informática gráfica
- algoritmos de bioinformática
- algoritmos de geometría computacional
- algoritmos de planificación y optimización
- algoritmos de simulación y visualización

¿Ya está hecho todo?

el mundo cambia continuadamente:

- los lenguajes de programación
- las arquitecturas de los ordenadores
- las aplicaciones de interés

Consecuencia:

Se necesita reprogramar y adaptar.

Ley de Moore

cada 18 meses:

- se duplica la velocidad de los procesadores
- se duplica la capacidad de la memoria
- se duplica el tamaño de las aplicaciones
- (se duplica el número de procesadores)

Consecuencia:

La misma aplicación necesita el doble (o más) del tiempo.

Ley para el ingeniero

Finalmente:

Hay que saber los algoritmos fundamentales y sus propiedades, porque son el *software* ultimativamente portable.

Notación O

- decimos una función $f(n)$ es de $O(g(n))$
- si existe una constante c y un n_0
- tal que: $f(n) \leq c \cdot g(n)$ para todos los $n > n_0$
- decimos una función $f(n)$ es de $\Omega(g(n))$
- si existe una constante c y un n_0
- tal que: $f(n) \geq c \cdot g(n)$ para todos los $n > n_0$

Evaluar lo que es interesante

- se concentra en la operaciones fundamentales
- se analiza: caso óptimo, caso medio, caso peor.
- se dice: algoritmo es óptimo, si orden del límite inferior es igual que el orden del límite superior

Taxonomía

Usamos una taxonomía en tres niveles:

Principio: describe cuál es la idea principal para lograr la ordenación

Método: describe una idea específica dentro de un principio para lograr la ordenación

Algoritmo: describe cuáles son los pasos concretos en una implementación de un método de ordenación

Principios de ordenación

- basados en comparaciones de claves
- basados en contar ocurrencias
- basados en conocimiento sobre los datos

Métodos de ordenación basados en comparaciones

- extracción aleatorio, inserción en orden
- extracción en orden, inserción directa
- partición de los datos y recursividad
- hay todo un zoo:
  InsertionSort, HeapSort, QuickSort, MergeSort,
  BubbleSort, SelectionSort, IntroSort, FlashSort, BurstSort,
  ShellSort, etc.

Complejidad de ordenación basados en comparaciones

- Se necesita $\Omega(n \log n)$ comparaciones.
- Argumento: profundidad mínima de un árbol de decisiones.
- hay $n!$ posibles hojas en el árbol
- tenemos para su altura:
  $h \geq \log n! \geq \log(n/e)^n \geq n \log n - n$

Métodos de ordenación basados en contar ocurrencias

- BucketSort
- CountingSort
- RadixSort
- (con estos seguimos mañana)

Algoritmos de ordenación basados en conocimiento

JordanSort: si se sabe que los datos por ordenar están a lo largo de una curva de Jordan, se puede ordenar en tiempo lineal, $O(n)$.

Propiedades de interés de algoritmos de ordenación

- tiempo de cálculo (dependencia de los datos)
- uso de memoria adicional
- ordenación estable
- número de comparaciones
- número de operaciones de intercambio de datos
- forma de acceso a la memoria
- comportamiento en jerarquía de memory
- paralelizable
- predicible (determinístico)

¿Qué es ordenación estable?

- dos elementos iguales mantienen el orden después de la ordenación
- especialmente interesante si se ordena elementos con campos diferentes
- por ejemplo:
  primero nombre, después tipo, después tamaño
- no funciona (todavía) en GUIs de sistemas operativos (ni Gnome, ni Windows)

InsertionSort

- extrae elementos en cualquier orden
  (aquí de izquierda a derecha)
- inserta en estructura ordenada
  (aquí en vector ordenado desde la derecha)

InsertionSort

```
int InsertionSort(IntArray& A) {
  for(unsigned int i(1);  i<A.size(); ++i) {
    const int value(A[i]);
    j=i-1;
    while(j>=0 && A[j]>value) {
      A[j+1]=A[j];
      --j;
    }
    A[j+1]=value;
  }
}
```

HeapSort

- construye una estructura llamado *heap* (montón)
- extrae mínimo e inserta en orden

HeapSort (C++)

```
void HeapSort(IntArray& A) {
  Heap heap(A);
  for(unsigned int k(A.size()-1); k >= 1; --k) {
    A[k]=heap.DeleteMin();
  }
}
```

HeapSort (C++)

```
class Heap {
  IntArray& A;
  unsigned int size;
public:
  Heap(IntArray& A);
  int DeleteMin(void);
  void FilterDown(unsigned int current);
  unsigned int LeftChild(
    const unsigned int padre
  ) const {
    return 2*padre+1;
  }
}
```

HeapSort (C++)

```
int Heap::Heap(IntArray& A_) :
  A(A_),
  size(A.size())
{
  unsigned int current((size-2)/2); // ultimo padre
  while(current>=0)
    FilterDown(current);
    --current;
  }
}
```

HeapSort (C++)

```
int Heap::DeleteMin(void) {
  const int min(A[0]);   // guardar
  A[0]=A[size-1]; // copiar ultimo como nueva raiz
  size--;
  FilterDown(0);  // reajustar
  return min;
}
```

HeapSort (C++)

```
void Heap::FilterDown(unsigned int current) {
  const int target(A[current]);
  unsigned int child(LeftChild(current));
  while(child<size) {
    const unsigned int right_child(child+1);
    if((right_child<size) &&
       (A[right_child]<=A[child])
    ) child=right_child;
    if(target<=A[child]) break;  // ya estan
    A[current]=A[child]; // mover contenido arriba
    current=child;       // bajar en el arbol
    child=LeftChild(current);
  }
  A[current]=target; // copiar a su posicion
}
```

HeapSort análisis

- `FilterDown` necesita tiempo proporcional a la altura donde comienza
- entonces la construcción se realiza en tiempo lineal
- la ordenación se realiza en tiempo $O(n \log n)$

QuickSort

- elige elemento pivote
- particiona los datos con el pivote
- ordena recursivamente

QuickSort (Java)

```
void QuickSort(Item a[], int left, int right) {
  if(right<=left) return;
  int i=left-1, j=right, Item pivot=a[right];
  for(;;) {
    while(a[++i]<pivot);
    while(pivot<a[--j]) if(j==i) break;
    if(i>=j) break;
    Exchange(a[i],a[j]);
  }
  Exchange(a[i],a[right]);
  QuickSort(a,left,i-1); QuickSort(a,i+1,right);
}
```

Problemas de QuickSort

- ¿Cómo escoger el pivote?
- mediano de tres: primero, medio, y último
- ¿Qué hacer con elementos iguales?
  - A: ¿Todo en un lado (1962)?
  - B: ¿Dejar donde están?
  - C: ¿Hay algo mejor (1993)?

QuickSort mejorado

- establecer la siguiente invariante después de la partición
- hay cuatro partes:
  iguales, más pequeños, más grandes, iguales
- transformar antes de la recursión en
  más pequeños, iguales, más grandes
- se ordena solamente las partes necesarias

QuickSort (mejorado)

```
void QuickSort(Item a[], int left, int right) {
  if(right<=left) return;
  int i=left-1, j=right, Item pivot=a[right];
  int p=left-1, q=right;
  for(;;) {
    while(a[++i]<pivot);
    while(pivot<a[--j]) if(j==i) break;
    if(i>=j) break;
    Exchange(a[i],a[j]);
    if(a[i]==pivot) Exchange(a[++p],a[i]);
    if(pivot==a[j]) Exchange(a[--q],a[j]);
  }
  Exchange(a[i],a[right]);
  for(int k=1;k<p;)       Exchange(a[k++],a[j--]);
  for(int k=right-1;k>q;) Exchange(a[k--],a[i++]);
  QuickSort(a,left,j); QuickSort(a,i,right);
}
```

MergeSort

- particiona los datos en partes
- ordena recursivamente
- intercala las partes ordenadas

Comparación

Algoritmo	media	peor	estable	mem
InsertionSort	$O(n^2)$	$O(n^2)$	si	$O(1)$
QuickSort	$O(n \log n)$	$O(n^2)$	no	$O(\log n)$
MergeSort	$O(n \log n)$	$O(n \log n)$	si	$O(n)$
HeapSort	$O(n \log n)$	$O(n \log n)$	no	$O(1)$

Abierto:

No se conoce ningún algoritmo que es óptimo tanto en tiempo como en memoria y al mismo tiempo estable.

Comparación

Algoritmo	paralizable	cache	acceso	listas
InsertionSort	?	(si)	lineal	bien
QuickSort	si	bien	lineal	mal
MergeSort	si	bien	lineal	bien
HeapSort	no	mal	aleatorio	muy mal

Abierto:

No se conoce ningún algoritmo paralelo que es óptimo y práctico.

Contar en vez de comparar

- se cuenta en *cubos* para cada clave cuantas veces un elemento se encuentra en los datos
- se enumeran los datos según contenido de los cubos BucketSort
- ¿Y si hay muchas claves?
- se subdivide la clave en partes y se ordena iterativamente RadixSort
- desde más significante a menos: MSB RadixSort
  desde menos significante a más: LSB RadixSort

C++ Standard Template Library

- implementación del *unstable sort* con IntroSort
- una combinación de
- median–of–3–QuickSort + HeapSort + delayed–InsertionSort
  tiempo peor en $O(n \log n)$

Java ordenación interna (*system sort*)

- QuickSort mejorado para tipos simples
- MergeSort para objetos

Ataque:

Es posible realizar un *ataque algoritmico* a una servidor en Java explotando la posibilidad de obligar a ordenar en tiempo en orden $O(n^2)$.

Banco de prueba para conjuntos de datos grandes

- banco de prueba: Gray benchmark
- hay que ordenar por lo menos 100 TB (tera byte)
- record 2009: 0.578 TB/min
- por: Hadoop, Owen O'Malley and Arun Murthy, Yahoo Inc.
- en concreto: 100 TB en 173 minutos
  3452 nodos x (2 Quadcore Xeons, 8 GB memory, 4 SATA)

¿Qué hacemos con los datos ordenados?

- hay que preguntarse que se hace con los datos ordenados
- a veces basta con algo *más simple*
- ejemplo: quiero los k valores más pequeños (y ordenados) de una lista de n (los *top* k)
- genera *heap* en tiempo $O(n)$ y estrae los k mínimos en tiempo $O(k \log n)$
- de hecho incluso es posible hacerlo en $O(n + k \log k)$, pero con una constante mucho más grande y si $k < n/\log n$ no vale la pena

¿Dónde se ordenan los datos?

- BurstSort: método específico para ordenar cadenas de caracteres explotando bien la estructura del caché del ordenador.
- GpuSort: RadixSort paralelizado (y implementado con CUDA) para su ejecución en tarjetas gráficas con muchos núcleos de procesamiento

¿Cómo se puede buscar?

- iterar linealmente sobre los datos hasta que se haya encontrado lo buscado
- ordenar primero los datos y buscar con un árbol como estructura de datos adicional
- el uso de un árbol requiere acceso por
  - índice, entonces se trabaja con arrays, o
  - puntero, entonces se trabaja con enlaces
- importante: los datos tienen que ser ordenables

¿Hay otras formas?

- usar tablas de hash
- acceso a un elemento en tiempo constante
- Cuckoo–hashing como método con todas las operaciones (insertar, localizar, borrar) en tiempo constante (esperado con alta probabilidad)

¿Qué más se quiere hacer?

enumerar rangos:

- funciona con listas y árboles
- no funciona con tablas de hash

mantener cola de prioridades:

- se mantiene un heap adecuado (Fibonacci)
- que permite insertar y bajar prioridad en tiempo constante
- `DeleteMin` y borrar sigue necesitar tiempo logarítmico

Colas de prioridad

	lista	lista ordenada	heap binario	heap binomial	heap fibonacci
insert	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$
delete-min	$O(n)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
decrease-key	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$
merge	$O(1)$	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$
find-min	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
delete	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
increase-key	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
make-queue	$O(n)$	$O(n \log n)$	$O(n)$	$O(n \log n)$	$O(n)$

extensión a dos (o más) dimensiones

- subdivisión binaria del espacio
- *kd*–árboles
- QuadTree
- diagramas de Voronoi
- ordenación y busqueda de intervalos

Nota:

Si se aumenta la dimensión, los tiempos de cálculo para construir la estructura de datos o la búsqueda suelen tener un factor que depende exponencialmente de $d - 1$, es decir, a partir de tres dimensiones, los algoritmos ya no funcionan en tiempo logarítmico.

Bibliografia I

- Th. Corman, C. Leiserson, R. Rivest. Indroduction to algorithms, Mc Graw Hill, 1990.
- D. Knuth. The art of computer programming: Sorting and searching, Vol. 3. Addison Wesley, 1973 (o ediciones más nuevas)
- material de R. Sedgewick y J. Bentley disponible en la red
- S. Merritt. An inverted taxonomy of sorting algorithms. Communications of the ACM, Vol. 28, 1985.

Bibliografia II

- N. Satish, M. Harris, M. Garland. Designing efficient sorting algorithms for manycore GPUs. IEEE International Parallel and Distributed Processing Symposium, 2009.
- K. Fung, T. Nicholl, R. Tarjan, C. Van Wyk. Simplified linear-time Jordan sorting and polygon clipping. Information Processing Letters 35, 1990.
- `http://en.wikipedia.org` secciones: sorting algorithms, searching algorithms