Вопросы к экзамену по курсу «Методы хранения и обработки информации» для студентов групп 2311 и 2361 (2004/2005 уч. год)

...в обработке Master'a...

17. Внутренние сортировки. Их основные характеристики. Теорема о нижней границе числа обменов для произвольного алгоритма внутренней сортировки. Примеры альтернативных более быстрых алгоритмов

Обычно методы сортировки подразделяют на два класса:
внутренние, когда все записи хранятся в быстрой оперативной памяти;
внешние, когда все записи в ней не помещаются.

Методы внуренней сортировки обеспечивают большую гибкость при построении структур данных и доступа к ним.

Теорема о нижней границе числа обменов для произвольного алгоритма внутренней сортировки:
пусть сложность алгоритма не меньше d и сортируется n различных эл-тов. Тогда:
$2^d \geq n! \Rightarrow d \geq n \log n$

Док-во:
Пусть 0—нет обмена, 1—есть обмен двух элементов. Тогда

1 набор 100000 ... 00
2 набор 110010 ... 01
... ...
k набор $\underbrace{101011 \ldots 11}_{2^d}$

Поскольку правильный алгоритм должен предусматривать все возможные порядки, то на вход можно подать не менее $n!$ наборов, в то время как при неизменной сложности d максимальное число рассматриваемых наборов равно 2^d. Отсюда
$2^d \geq n! \Rightarrow d \geq n \log n$

Алгоритм **сортировки подсчётом** применим, если каждый из n эл-тов сортируемой последовательности — целое положительное число в известном диапазоне (не превосходящее заранее известного k). Составляем массив натуральных чисел длины k, заполненный нулями. Затем для каждого эл-та x сортируемой последовательности проделываем следующую операцию: к значению ячейки массива чисел с индексом x прибавляем единицу. Таким образом, остаётся только переписать построенный массив и получить отсортированную последовательность.
Сложность алгоритма:
$T(n) = \Theta(n + k)$

Алгоритм **цифровой сортировки** устроен следующим образом: представляем сортируемые числа в k-ичной системе счисления, сортируем их по последнему разряду, затем по предпоследнему и т.д. В результате получаем отсортированную последовательность. Правильность алгоритма цифровой сортировки доказывается индукцией по номеру разряда. Для n чисел с d знаками от 0 до $k-1$ каждый проход

занимает время $\Theta(n+k)$; поскольку мы делаем d проходов, то время работы цифровой сортировки равно $\Theta(d(n+k))$.

18. Сортировка простыми включениями. Анализ сложности

Сортировка вставками удобна для сортировки коротких последовательностей. Именно таким способом обычно сортируют карты: держа в одной руке уже упорядоченные карты и взяв другой рукой очередную карту, мы вставляем её в нужное место, сравнивая с имеющимися отсортированными и идя справа налево (или слева направо - кому как больше нравится).

Запишем этот алгоритм в виде метода `InsertionSort`:

```
static void InsertionSort(int a[]) {
   for (int i = 1; i < a.length; i++) {
      int j = i;
      int B = a[i];
      while ((j > 0) && (a[j-1] > B)) {
         a[j] = a[j-1];
         j--;
      }
      a[j] = B;
   }
}
```

Последовательность сортируется «на месте», без дополнительной памяти (in place): помимо массива мы используем лишь фиксированное число ячеек памяти.

Число пересылок:

$M_{min} = 2(n-1),\ M_{max} = 2(n-1) + \frac{(n-1)n}{2}$

Число сравнений:

$C_{min} = n-1,\ C_{max} = \frac{(n-1)n}{2}$

Порядок роста:

$T(n) = \Theta(n^2)$

19. Сортировка простым выбором. Анализ сложности

Будем сортировать массив из n элементов так: просмотрим его и найдём минимальный элемент, который обменяем с первым элементом. Затем просмотрим массив снова и найдём следующий элемент, и так далее...

```
static void SelectionSort(int a[]) {
   for (int i = 0; i < a.length; i++) {
      int min = i;
      for (int j = i + 1; j < a.length; j++) {
         if (a[j] < a[min]) {min = j;}
      }
      int T = a[min];
      a[min] = a[i];
      a[i] = T;
```

```
        }
    }
```

Число пересылок:
$M_{min} = M_{max} = M_{cp} = 3n$

Число сравнений:
$C_{min} = C_{max} = C_{cp} = \frac{n(n-1)}{2}$

Порядок роста:
$T(n) = \Theta(n^2)$

20. Сортировка простым обменом. Анализ сложности

Пожалуй, наиболее очевидный способ обменной сортировки - сравнить два ключа и поменять их местами, если они расположены не в нужном порядке, и затем проделать то же самое с остальными парами.

```
static void BubbleSort(int a[]) {
   for (int i = a.length; --i >= 0; ) {
      for (int j = 0; j < i; j++) {
         if (a[j] > a[j+1]) {
            int T = a[j];
            a[j] = a[j+1];
            a[j+1] = T;
         }
      }
   }
}
```

Число пересылок:
$M_{min} = 0,\ M_{max} = \frac{3(n^2-n)}{2},\ M_{cp} = \frac{3(n^2-n)}{4}$

Число сравнений:
$C_{min} = C_{max} = C_{cp} = \frac{(n-1)n}{2}$

Порядок роста:
$T(n) = \Theta(n^2)$

21. Идея сортировки Шелла

```
static void ShellSort(int a[]) {
        int h = 1;
        /*
         * find the largest h value possible
         */
        while ((h * 3 + 1) < a.length) {h = 3 * h + 1;}
        /*
         * while h remains larger than 0
         */
```

```
        while( h > 0 ) {
            /*
             * for each set of elements (there are h sets)
             */
            for (int i = h - 1; i < a.length; i++) {
                /*
                 * pick the last element in the set
                 */
                int B = a[i];
                int j = i;
                /*
                 * compare the element at B to the one before it in the set
                 * if they are out of order continue this loop, moving
                 * elements "back" to make room for B to be inserted.
                 */
                for( j = i; (j >= h) && (a[j-h] > B); j -= h) {a[j] = a[j-h];}
                /*
                 *  insert B into the correct place
                 */
                a[j] = B;
            }
            /*
             * all sets h-sorted, now decrease set size
             */
            h = h / 3;
        }
}
```

22. Идея пирамидальной сортировки. Определение пирамиды на массиве. Пирамидальная сортировка и оценка её эффективности в худшем случае

Двоичной кучей (binary heap), или **пирамидой**, называют массив с определёнными свойствами упорядоченности. Чтобы сформулировать эти свойства, будем рассматривать массив как двоичное дерево. Каждая вершина дерева соответсвует эл-ту массива. Если вершина имеет индекс i, то её родитель имеет индекс $\lfloor i/2 \rfloor$ (вершина с индексом 1 является корнем), а её дети — индексы $2i$ и $2i + 1$. Будем считать, что куча может не занимать всего массива.

Алгоритм сортировки с помощью кучи состоит из двух частей. Сначала вызывается процедура `downheap (heapify)`, после выполнения которой массив становится кучей. Идея второй части проста: максимальный эл-т массива теперь находится в корне дерева ($A[1]$). Его следует поменять с $A[n]$, уменьшить размер кучи на 1 и восстановить основное свойство в корневой вершине. После этого в корне будет находится максимальный из оставшихся эл-тов. Так делается до тех пор, пока в куче не останется всего один элемент.

Время работы процедуры `HeapSort` составляет $\Theta(n \log n)$.

```
        static void HeapSort(int a[]) {
           int N = a.length;
           for (int k = N/2; k > 0; k--) {
```

```
      downheap(a, k, N);
   }
   do {
      int T = a[0];
      a[0] = a[N - 1];
      a[N - 1] = T;
      N = N - 1;
      downheap(a, 1, N);
   } while (N > 1);
}

static void downheap(int a[], int k, int N) {
   int T = a[k - 1];
   while (k <= N/2) {
      int j = k + k;
      if ((j < N) && (a[j - 1] < a[j])) {
         j++;
      }
      if (T >= a[j - 1]) {
         break;
      } else {
         a[k - 1] = a[j - 1];
         k = j;
      }
   }
   a[k - 1] = T;
}
```