Advanced Topics in Sorting

ducnh-fit@mail.hut.edu.vn

Sorting applications

Sorting algorithms are essential in a broad variety of applications

- Organize an MP3 library.
- Display Google PageRank results.
- List RSS news items in reverse chronological order.
- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.
- Data compression.
- Computer graphics.
- Computational biology.
- Supply chain management.
- Load balancing on a parallel computer.
- . . .

Sorting algorithms

Many sorting algorithms to choose from

Internal sorts

- Insertion sort, selection sort, bubblesort, shaker sort.
- Quicksort, mergesort, heapsort, samplesort, shellsort.
- Solitaire sort, red-black sort, splaysort, Dobosiewicz sort, psort, ...

External sorts

- Poly-phase mergesort, cascade-merge, oscillating sort.

Radix sorts

- Distribution, MSD, LSD.
- 3-way radix quicksort.

Parallel sorts

- Bitonic sort, Batcher even-odd sort.
- Smooth sort, cube sort, column sort.
- GPUsort.

Which algorithm to use?

Applications have diverse attributes

- Stable?
- Multiple keys?
- Deterministic?
- Keys all distinct?
- Multiple key types?
- Linked list or arrays?
- Large or small records?
- Is your file randomly ordered?
- Need guaranteed performance?

Cannot cover all combinations of attributes.

Case study 1

Problem

- Sort a huge randomly-ordered file of small records.

Example

- Process transaction records for a phone company.

Which sorting method to use?

1. Quicksort: YES, it's designed for this problem
2. Insertion sort: No, quadratic time for randomly-ordered files
3. Selection sort: No, always takes quadratic time

Case study 2

Problem

- Sort a huge file that is already almost in order.

Example

- Re-sort a huge database after a few changes.

Which sorting method to use?

1. Quicksort: probably no, insertion simpler and faster
2. Insertion sort: YES, linear time for most definitions of "in order"
3. Selection sort: No, always takes quadratic time

Case study 3

Problem: sort a file of huge records with tiny keys.

Ex: reorganizing your MP3 files.

Which sorting method to use?

1. Mergesort: probably no, selection sort simpler and faster
2. Insertion sort: no, too many exchanges
3. Selection sort: YES, linear time under reasonable assumptions

Ex: 5,000 records, each 2 million bytes with 100-byte keys.

- Cost of comparisons: 100 x 50002 / 2 = 1.25 billion
- Cost of exchanges: 2,000,000 x 5,000 = 10 trillion
- Mergesort might be a factor of log (5000) slower.

Duplicate keys

Often, purpose of sort is to bring records with duplicate keys together.

- Sort population by age.
- Finding collinear points.
- Remove duplicates from mailing list.
- Sort job applicants by college attended.

Typical characteristics of such applications.

- Huge file.
- Small number of key values.

Mergesort with duplicate keys: always ~ N lg N compares

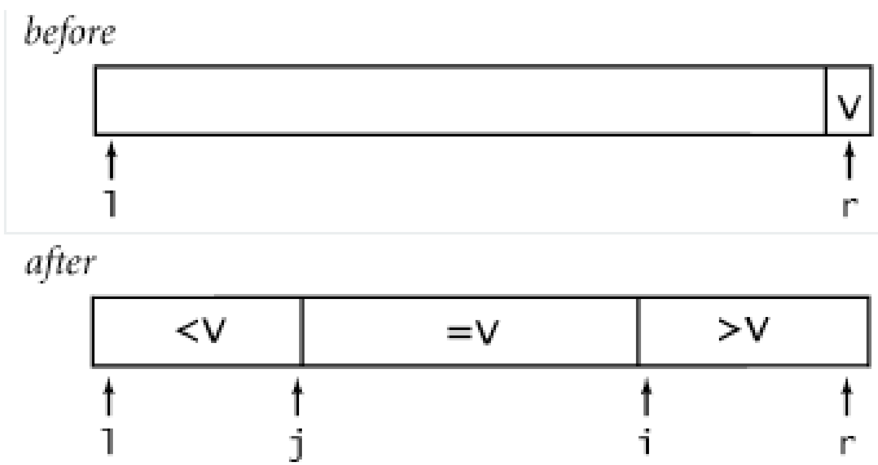
Quicksort with duplicate keys

- algorithm goes quadratic unless partitioning stops on equal keys!
- 1990s Unix user found this problem in qsort()

3-Way Partitioning

3-way partitioning. Partition elements into 3 parts:

- Elements between i and j equal to partition element v.
- No larger elements to left of i.
- No smaller elements to right of j.

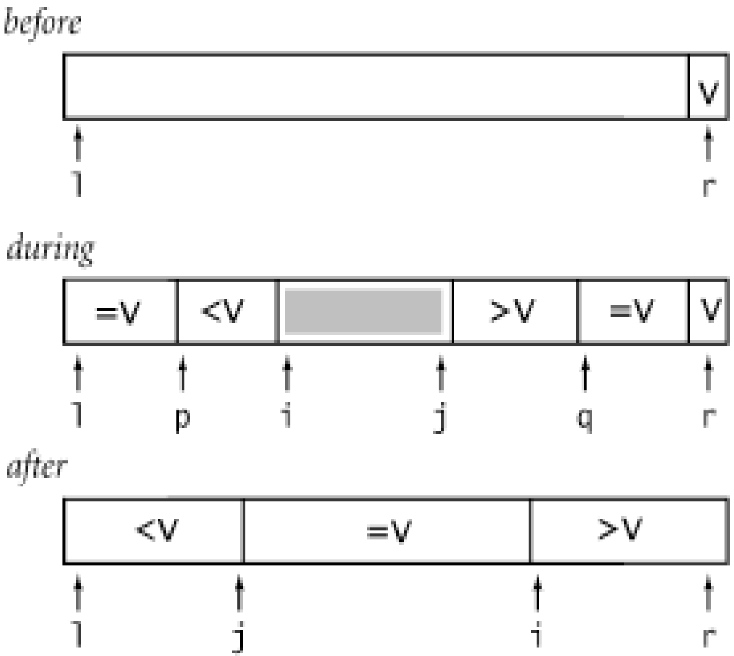


Implementation solution

3-way partitioning (Bentley-McIlroy): Partition elements into 4 parts:

- no larger elements to left of i
- no smaller elements to right of j
- equal elements to left of p
- equal elements to right of q

Afterwards, swap equal keys into center.



Code

```
void sort(int a[], int l, int r) {
   if (r <= l) return;
   int i = l-1, j = r;
   int p = l-1, q = r;
   while(1)    {
        while (a[++i] < a[r]));
        while (a[r] < a[--j])) if (j == l) break;
        if (i >= j) break;
        exch(a, i, j);
        if (a[i]==a[r]) exch(a, ++p, i);
        if (a[j]==a[r]) exch(a, --q, j);
   }
   exch(a, i, r);
   j = i - 1;
   i = i + 1;
   for (int k = l ; k <= p; k++) exch(a, k, j--);
   for (int k = r-1; k >= q; k--) exch(a, k, i++);
   sort(a, l, j);
   sort(a, i, r);
}
```

Demo

- demo-partition3.ppt

Quiz 1

- Write two quick sort algorithms
  - 2-way partitioning
  - 3-way partitioning
- Create two identical arrays of 10 millions randomized numbers having value from 1 to 10.
- Compare the time for sorting the numbers using each algorithm

Instructions (1)

- Write a function to create new data stored in a dynamic memory. The array's size is passed as a parameter.
  - int * createArray(int size);
- Call rand() function to generate a random number in the range 0 to RAND_MAX
  - #include <stdlib.h>
  - i = rand();
- Write a function to help duplicating data from an existing array.
  - int * dumpArray(int *p, int size);

Instructions (2)

- Call memcpy() function to copy data from an array to another array.
  - memcpy(void* dest, void* src, size_t size);
- Write 2 sorting algorithms in 2 functions:
  - void sort2way(int a[], int l, int r);
  - void sort3way(int a[], int l, int r);
- Write a main() function where we can firstly verify the correctness of the sorting functions on a small data and then check their performance on a huge volume data.

Instructions (3)

```
#define SMALL_NUMBER 20
#define HUGE_NUMBER 10000000
main() {
    int* a1, a2;
    a1 = createArray(SMALL_NUMBER);
    a2 = dumpArray(a1, SMALL_NUMBER);
  sort2way(a1, 0, SMALL_NUMBER-1);
    /* print data in a1 */
    sort2way(a2, 0, SMALL_NUMBER-1);
    /* print data in a2 */
    free (a1);
    free (a2);
    a1 = createArray(HUGE_NUMBER);
    a2 = dumpArray(a1, HUGE_NUMBER);
    /* compare the time to execute sorting */
}
```

Instructions (4)

- How to check the performance

```
#include <time.h>
#include <stdio.h>
time_t start,end;
volatile long unsigned t;

start = time(NULL);

/* your algorithm to check the performance */

end = time(NULL);

printf("Run in %f seconds.\n", difftime(end, start));
```

Generalized sorting

- In C we can use the qsort function for sorting

```
void qsort(
      void *buf,
      size_t num,
      size_t size,
      int (*compare)(void const *, void  const *)
);
```

- The qsort() function sorts *buf* (which contains *num* items, each of size *size*).
- The *compare* function is used to compare the items in *buf*. *compare* should return negative if the first argument is less than the second, zero if they are equal, and positive if the first argument is greater than the second.

Example

```
int int_compare(void const* x, void const *y) {
   int m, n;
   m = *((int*)x);
   n = *((int*)y);
   if ( m == n ) return 0;
   return m > n ? 1: -1;
}
void main()
{
   int a[20], n;
   /* input an array of numbers */
   /* call qsort */
   qsort(a, n, sizeof(int), int_compare);
}
```

Brief on function pointer

- Declare a pointer to a function
  - int (*pf) (int);
- Declare a function
  - int f(int);
- Assign a function to a function pointer
  - pf = &f;
- Call a function via pointer
  - ans = pf(5); // which are equivalent with ans = f(5)

- In the qsort() function, *compare* is a function pointer to reference to a compare the items

Quiz 2

- How to use qsort() to sort an array in ascendant or descendant order?
- Rewrite the program in Quiz 1 to compare the performance of your algorithm with the one of qsort().
- Let a file contain the data of a phone book (records of name and phone numbers). Write a program to read the phone book's data and sort the records by name using qsort().