Sorting Algorithms

Definition of a sorting problem

Input: A sequence of N numbers (a1,a2,...,aN)

Output: A permutation (a1′,a2′,...,aN′) of the input sequence such that a1′ <= a2′ <=... <= aN′.

Things to be considered in Sorting

These are the difficulties in sorting that can also happen in real life:

A. Size of the list to be ordered is the main concern. Sometimes, the computer memory is not sufficient to store all data. You may only be able to hold part of the data inside the computer at any time, the rest will probably have to stay on disc or tape. This is known as the problem of external sorting. However, rest assured, that almost all programming contests problem size will never be extremely big such that you need to access disc or tape to perform external sorting... (such hardware access is usually forbidden during contests).

B. Another problem is the stability of the sorting method. Example: suppose you are an airline. You have a list of the passengers for the day's flights. Associated to each passenger is the number of his/her flight. You will probably want to sort the list into alphabetical order. No problem... Then, you want to re-sort the list by flight number so as to get lists of passengers for each flight. Again, "no problem"... - except that it would be very nice if, for each flight list, the names were still in alphabetical order. This is the problem of stable sorting.

C. To be a bit more mathematical about it, suppose we have a list of items {xi} with xa equal to xb as far as the sorting comparison is concerned and with xa before xb in the list. The sorting method is stable if xa is sure to come before xb in the sorted list.

Finally, we have the problem of key sorting. The individual items to be sorted might be very large objects (e.g. complicated record cards). All sorting methods naturally involve a lot of moving around of the things being sorted. If the things are very large this might take up a lot of computing time -- much more than that taken just to switch two integers in an array.

Comparison-based sorting algorithms

Comparison-based sorting algorithms involves comparison between two object a and b to determine one of the three possible relationship between them: less than, equal, or greater than. These sorting algorithms are dealing with how to use this comparison effectively, so that we minimize the amount of such comparison. Lets start from the most naive version to the most sophisticated comparison-based sorting algorithms.

1. Bubble Sort

Speed: O(n^2), extremely slow

Space: The size of initial array

Coding Complexity: Simple

This is the simplest and (unfortunately) the worst sorting algorithm. This sort will do double pass on the array and swap 2 values when necessary.

```
BubbleSort(A)
for i ← length[A]-1 down to 1
for j ← 0 to i-1
if (A[j] > A[j+1]) // change ">" to "<" to do a descending sort
temp ← A[j]
A[j] ← A[j+1]
A[j+1] ← temp
```

Slow motion run of Bubble Sort (Bold == sorted region):

```
5 2 3 1 4
2 3 1 4 5
2 1 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5 >> done
```

2. Selection Sort

This algorithm splits the input array into sorted and unsorted parts, and with each iteration finds the smallest element remaining in the unsorted region and moves it to the end of the sorted region:

```
selection_sort(int s[], int n)
{
 int i,j; /* counters */
 int min; /* index of minimum */
 for (i=0; i<n; i++) {
  min=i;
  for (j=i+1; j<n; j++)
   if (s[j] < s[min]) min=j;
  swap(&s[i],&s[min]);
  }
}
```

Selection sort makes a lot of comparisons, but is quite efficient if all we count are the number of data moves. Only n-1 swaps are performed by the algorithm, which is necessary in the worst case; think about sorting a reversed permutation. It also provides an example of the power of advanced

data structures. Using an efficient priority queue to maintain the unsorted portion of the array suddenly turns O(n2) selection sort into O(n lg n) heapsort!

3.Insertion Sort

This algorithm also maintains sorted and unsorted regions of the array. In each iteration, the next unsorted element moves up to its appropriate position in the sorted region:

```
insertion_sort(int s[], int n)
{
  int i,j; /* counters */
  for (i=1; i<n; i++) {
   j=i;
   while ((j>0) && (s[j] < s[j-1])) {
     swap(&s[j],&s[j-1]);
     j = j-1;
    }
   }
}
```

Insertion sort is particularly significant as the algorithm which minimizes the amount of data movement. An inversion in a permutation p is a pair of elements which are out of order, i.e., an i, j such that i < j yet p[i] > p[j]. Each swap in insertion sort erases exactly one inversion, and no element is otherwise moved, so the number of swaps equals the number of inversions. Since an almost-sorted permutation has few inversions, insertion sort can be very effective on such data.

4. Quicksort

This algorithm reduces the job of sorting one big array into the job of sorting two smaller arrays by performing a partition step. The partition separates the array into those elements that are less than the pivot/divider element, and those which are strictly greater than this pivot/divider element. Because no element need ever move out of its region after the partition, each subarray can be sorted independently. To facilitate sorting subarrays, the arguments to quicksort include the indices of the first (l) and last (h) elements in the subarray.

```
quicksort(int s[], int l, int h)
{
  int p; /* index of partition */
  if ((h-l)>0) {
    p = partition(s,l,h);
    quicksort(s,l,p-1);
    quicksort(s,p+1,h);
    }
}
int partition(int s[], int l, int h)
```

```
{
  int i; /* counter */
  int p; /* pivot element index */
  int firsthigh; /* divider position for pivot */
  p = h;
  firsthigh = l;
  for (i=l; i<h; i++)
    if (s[i] < s[p]) {
      swap(&s[i],&s[firsthigh]);
      firsthigh ++;
     }
   swap(&s[p],&s[firsthigh]);
   return(firsthigh);
}
```

Quicksort is interesting for several reasons. When implemented properly, it is the fastest in-memory sorting algorithm. It is a beautiful example of the power of recursion. The partition algorithm is useful for many tasks in its own right. For example, how might you separate an array containing just 0's and 1's into one run of each symbol?

Sorting Library Functions

Whenever possible, take advantage of the built-in sorting/searching libraries in your favorite programming language:

• Sorting and Searching in C

The stdlib.h contains library functions for sorting and searching. For sorting, there is the function qsort:

```
#include <stdlib.h>
void qsort(void *base, size_t n, size_t width,
int (*compare) (const void *, const void *));
```

The key to using qsort is realizing what its arguments do. It sorts the first n elements of an array (pointed to by base), where each element is width-bytes long. Thus we can sort arrays of 1-byte characters, 4-byte integers, or 100-byte records, all by changing the value of width.

The ultimate desired order is determined by the function compare. It takes as arguments pointers to two width-byte elements, and returns a negative number if the first belongs before the second in sorted order, a positive number if the second belongs before the first, or zero if they are the same. Here is a comparison function for sorting integers in increasing order:

```
int intcompare(int *i, int *j)
{
  if (*i > *j) return (1);
  if (*i < *j) return (-1);
  return (0);
```

```
}
```

This comparison function can be used to sort an array a, of which the first cnt elements are occupied, as follows:

qsort((char *) a, cnt, sizeof(int), intcompare);

The name qsort suggests that quicksort is the algorithm implemented in this library function, although this is usually irrelevant to the user. Note that qsort destroys the contents of the original array, so if you need to restore the original order, make a copy or add an extra field to the record.

Binary search is an amazingly tricky algorithm to implement correctly under pressure. The best solution is not to try, since the stdlib.h library contains an implementation called bsearch(). Except for the search key, the arguments are the same as for qsort.
To search in the previously sorted array, try

bsearch(key, (char *) a, cnt, sizeof(int), intcompare); .

• Sorting and Searching in C++

The C++ Standard Template Library (STL), includes methods for sorting, searching, and more. Serious C++ users should get familiar with STL. To sort with STL, we can either use the default comparison (e.g., =) function defined for the class, or override it with a special-purpose comparison function op:

void sort(RandomAccessIterator bg, RandomAccessIterator end)

void sort(RandomAccessIterator bg, RandomAccessIterator end,BinaryPredicate op)

STL also provides a stable sorting routine, where keys of equal value are guaranteed to remain in the same relative order. This can be useful if we are sorting by multiple criteria:

void stable_sort(RandomAccessIterator bg, RandomAccessIterator end)

void stable_sort(RandomAccessIterator bg, RandomAccessIterator end, BinaryPredicate op)

Other STL functions implement some of the applications of sorting , including,

• nth element – Return the nth largest item in the container.

• set union, set intersection, set difference – Construct the union, intersection, and set difference of two containers.

• unique – Remove all consecutive duplicates.

• Sorting and Searching in Java The java.util.Arrays class contains various methods for sorting and

searching. In particular,

static void sort(Object[] a)

static void sort(Object[] a, Comparator c)

sorts the specified array of objects into ascending order using either the natural ordering of its elements or a specific comparator c. Stable sorts are also available. Methods for searching a sorted array for a specified object using either the natural comparison function or a new comparator c are also provided:

binarySearch(Object[] a, Object key)

binarySearch(Object[] a, Object key, Comparator c)

Linear-time Sorting

A. Lower bound of comparison-based sort is O(n log n)

The sorting algorithms that we see above are comparison-based sort, they use comparison function such as <, ≤, =, >, ≥, etc to compare 2 elements. We can model this comparison sort using decision tree model, and we can proof that the shortest height of this tree is O(n log n).

B. Counting Sort

For Counting Sort, we assume that the numbers are in the range [0..k], where k is at most O(n). We set up a counter array which counts how many duplicates inside the input, and the reorder the output accordingly, without any comparison at all. Complexity is O(n+k).

C. Radix Sort

For Radix Sort, we assume that the input are n d-digits number, where d is reasonably limited. Radix Sort will then sorts these number digit by digit, starting with the least significant digit to the most significant digit. It usually use a stable sort algorithm to sort the digits, such as Counting Sort above.

Example:
input:
321
257
113
622
sort by third (last) digit:
32**1**
62**2**
11**3**
25**7**
after this phase, the third (last) digit is sorted.
sort by second digit:
1**1**3
3**2**1

6**2**2
2**5**7
after this phase, the second and third (last) digit are sorted.
sort by second digit:
113
257
321
622
after this phase, all digits are sorted.

For a set of n d-digits numbers, we will do d pass of counting sort which have complexity O(n+k), therefore, the complexity of Radix Sort is O(d(n+k)).

Source:

http://www.learnalgorithms.in/#