Comparison Type Sorting Algorithms

Definition

A *comparison-based sorting algorithm* sorts an input sequence using only information obtained from comparisons between pairs of input elements.

Example

The comparison-based sorting algorithms:

Algorithm	Worst-case	Average-case	Best-case	In-Place
Selection Sort	$O(n^2)$	$\Theta(n^2)$	$\Omega(n^2)$	yes
Insertion Sort	$O(n^2)$	$O(n^2)$	$\Omega(n)$	yes
Merge Sort	$O(n \log n)$	$\Theta(n \log n)$	$\Omega(n \log n)$	no
Heap Sort	$O(n \log n)$	$\Theta(n \log n)$	$\Omega(n \log n)$	yes
Quick Sort	$O(n^2)$	$O(n \log n)$	$\Omega(n \log n)$	yes

Merge and heap sorts are asymptotically optimal because upper and lower bounds of their running times are of the same order.

Lower Bounds for Sorting

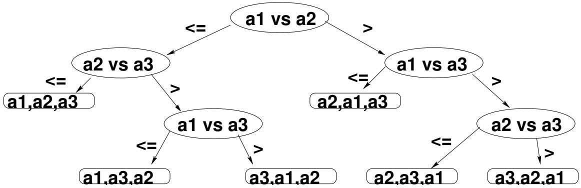
The question arises: *Can we find a faster comparison-based sorting algorithm to sort n elements $x_1, \ldots, x_n$ (say, of order $O(n)$)?*
The answer is: NO, since comparison sort type algorithms must make at least (lower bound) $n \log n$ comparisons in the worst case to sort a sequence of n elements.
To prove it, we need to view comparison-based sort algorithms abstractly in terms of a decision tree. This decision tree obeys the following rules:

1. Comparisons like $<$, $\leq$, $>$, $\geq$ are the only operations to be considered.
2. Each node of the decision tree contains only one comparison (of a pair of elements from an input sequence).
3. The left branch of the tree corresponds to one fixed comparison (say $\leq$) and the right branch of the tree corresponds to its negation (that is, $>$).
4. Each leaf node represents only one possible permutation of the input sequence (note that leaf nodes do not contain comparisons).

Decision Tree

Example

Consider insertion sort with a sequence $(a1, a2, a3)$ where $n = 3$. There are $3! = 6$ possible permutations, so we will have 6 different leaf nodes in the decision tree.



Facts

- For an n element sequence there are $n!$ different permutations (orderings).
- Every input permutation leads to a separate leaf output
- The height of a decision tree is equal to the length of the longest path from the root to a leaf.
- The number of comparisons corresponding to the height of the decision tree represents the worst case of the comparison-based algorithm.
- The height of a decision tree is at least $log_2(n!)$
- A lower bound on the height of the decision tree is a lower bound on the running time of any comparison-based sort algorithm.

The Lower-Bound Theorem for Comparison-Based Sorting

Theorem

Any decision tree for sorting n elements has height $\Omega(n \log n)$.

Proof.

- There must be $n!$ leaves; one for each possible ordering of n elements.
- The maximum number of leaf nodes of the binary tree is 2^h, where h is the height of the tree.

Consider a tree of height h that sorts n elements. There are $n!$ permutations of n elements so the decision tree must have exactly $n!$ leaves. A binary tree of height h has no more than 2^h leaf nodes. So we get the following when the Stirling inequality $(n! \geq (n/e)^n)$ is used:

$$\begin{aligned} 2^h &\geq n! \\ h &\geq \log_2(n!) \\ h &\geq n \log_2\left(\frac{n}{e}\right) = n \log n - n \log_2(e) = \Omega(n \log n) \end{aligned}$$

□

The Lower-Bound Theorem for Comparison-Based Sorting (cont)

Fact

The running time of any comparison-based algorithm for sorting n-element sequence takes at least $n \log_2 n$ (asymptotic notation $\Omega(n \log_2 n)$) comparisons in the worst case.

See the previous slide for a justification.

Non-Comparison Sort Algorithms

We are looking for sorting algorithms which are *not* of comparison-based type, and which can be faster than any algorithm of comparison-based type on a certain input sequences. Such non-comparison sort type algorithms are:

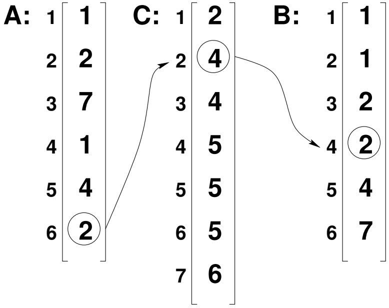
- counting sort
- radix sort
- bucket sort

They sort numbers by taking the information about the values to be sorted, or operate on “pieces” of input elements.
The running time of non-comparison sorting algorithms could be linear, $O(n)$.

Counting Sort

Counting sort requires integer input in the range 1 to k.
If $k = O(n)$ then the running time is $O(n)$.

The idea of the counting sort is to find the number of elements less than or equal to x (say m) for each input element x, and then put x in the $m+1$st spot in the output array. We need an input array $A[1..n]$, and two additional arrays: sorted output array $B[1..n]$, and temporary working storage array $C[1..k]$ (usually $k \geq n$).



Counting Sort (cont)

```
/* n is the length of A and B, k is the range of numbers */
void CountingSort(int *A, int *B, int n, int k){
   int i, j;
   /* C is a temporary storage initialized to 0 */
   int *C = new int[k+1];
   for (i = 0; i <= k; i++) C[i] = 0;
   /* C[i] now contains the number of elements equal to i */
   for (j = 0; j < n; j++) C[A[j]]++;
   /* C[i] now contains the number of elements <= i */
   for (i = 1; i <= k; i++) C[i] += C[i-1];
   for (j = n-1; j >= 0; j--) {/* B contains the sorted input */
      i = C[A[j]] - 1;    /* index for B */
      C[A[j]]--;/* decrease count */
      B[i] = A[j];  /* put A[j] in the right place */
   }
   delete[] C;  }
```

Running time:

- two loops are of size k
- two loops are of size n

Therefore the running time is $O(n + k)$. To get $O(n)$ we need to assume that $k = O(n)$. If it is not satisfied, then $\Theta(n \lg n)$ algorithms may be faster.

Stable Sorting Algorithms

Definition

A sorting algorithm is *stable* if equal elements in a sequence are in the same order in the input and output arrays. In other words, whichever element with the same key appears first in the input array, it must appear first in the output array.

Examples

1. Counting sort (see the implementation from the previous slide)
2. Insertion sort
3. Bubble sort

Radix Sort

Radix sort can be used for sorting integers or strings.

Facts

- It sorts on the least significant digit first (or the most right character first).
- For sorting it uses digits (or characters).
- If an input sequence consists of (at most) *d*-digit numbers, then radix sort needs *d* passes to sort these numbers.
- A sort algorithm used for sorting digits must be stable (otherwise we cannot obtain sorted sequences as output).

Example

$$\begin{array}{ccccccc}
128 & & 921 & & 921 & & 128 \\
345 & & 763 & & 324 & & 324 \\
763 & \Longrightarrow & 324 & \Longrightarrow & 128 & \Longrightarrow & 345 \\
921 & & 345 & & 345 & & 763 \\
324 & & 128 & & 763 & & 921 \\
 & & \uparrow & & \uparrow & & \uparrow
\end{array}$$

Ad Hoc Implementation of Radix Sort

```
const int base = 10;
int digit(int k, int num)
{  int r;
   r = num/(int)pow(base, k); /* integer division */
   return r % base;
}
/* we use counting sort for sorting digits */
void radix_sort(int *A, int n, int d)
{ int i, j, m;
  /* temporary storage */
  int *C = new int[base];
  int *B = new int[n];
  for (m = 0; m < d; m++) {
    for (i = 0; i < base; i++) C[i] = 0;
    for (j = 0; j < n; j++) C[digit(m, A[j])]++;
    for (i = 1; i < base; i++) C[i] += C[i-1];
    for (j = n-1; j >= 0; j--) {
      i = C[digit(m, A[j])]--;
      B[i] = A[j];
    } /* copy B -> A */
    for (j = 0; j < n; j++) A[j] = B[j];
  }
  delete [] B; delete [] C; }
```

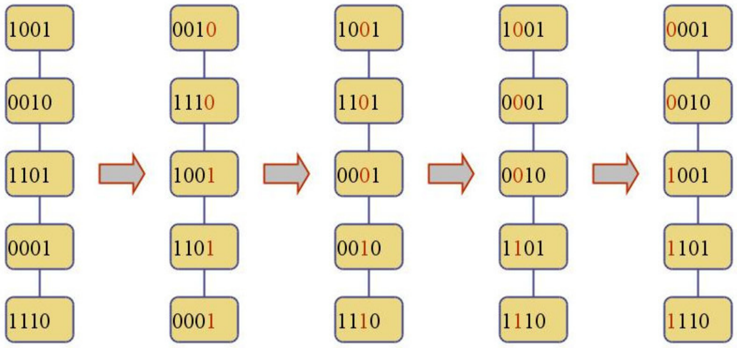
Running time: in the loop m we use the counting sort with the range $k = base$, whose running time is $O(n)$. Therefore the total time is $O(dn)$. If d is constant, then the running time of radix sort is $O(n)$.

A Low Level Efficient Implementation of Radix Sort

```
void radix_sort(unsigned int input, int size)
{
     long count[256];
     int shift = 0;
     int*  tmp_array = new int[size];
     while(shift < 32) {
        memset(count, 0, sizeof(count));
        for (int i = 0; i < size; ++i) {
           int bits = (input[i] >> shift) & 255;
           count [bits]++;
        }
        for(int i = 1; i < 256; ++i)
            count [i] += count [i] ;
        for( int i = size; i >= 0; i) {
          int bits = (input [i] >> shift) & 255;
          count[bits];
          tmp_array[count[bits]] = input [i];
        }
        memcpy(input, tmp_array, size*sizeof(input [0]));
        shift += 8 ;
    }
    delete [] tmp_array;
}
```

Sorting a Sequence of 4-bit Integers

Example



Bucket Sort

Bucket sort requires input data which are uniformly distributed over some range (often [0..1), and 1 is not included). The idea of bucket sort is to divide the given range into n equally-sized subranges (buckets) and then distribute the n input numbers into the buckets. Since the numbers are uniformly distributed, we can expect that only a few of them fall into each bucket. Then we sort the numbers in each bucket, and combine numbers into the sorted sequence collecting numbers from each bucket.

Example

A – input data, B – bucket. Bucket ranges: 0..9, 10..19, . . . , 90..99.

A	B				
12	0	$\rightarrow$	2		
36	1	$\rightarrow$	12		
53	2	$\rightarrow$	23	$\rightarrow$	28
28	3	$\rightarrow$	32	$\rightarrow$	36
91	4				
32	5	$\rightarrow$	53	$\rightarrow$	55
75	6				
23	7	$\rightarrow$	75		
2	8				
55	9	$\rightarrow$	91		

Problem Analysis

```
/* the range is 0..x, x is not included */
void BucketSort(int *A, int n, int x) {
   int i, j;
   AList B;     /* bucket as an array of linked-lists */
   B = CreateAList(n);  /* create n buckets */
   for (i = 0; i < n; i++) {
      j = (n*A[i])/x;
      InsertToList(A[i], B[j]);/*insert A[i] into list B[j]*/
   }
   for (i = 0; i < n; i++)
      InsertionSort(B[i]); /* use insertion sort */
   Collect(B, A, n);    /* concatenate the lists using A */
   FreeAList(B);        /* release buckets */
}
```

Running Time

An analysis of the running time for bucket sort:

- inserting elements to buckets takes $O(n)$
- insertion sort takes $O(1)$ since each bucket has only a few elements
- collecting elements and putting them back in A takes $O(n)$.

Therefore total expected running time is $O(n)$.
Note that the running time depends strongly on the uniform distribution of input elements. If input elements are not uniformly distributed, then in some buckets we can have more than only a few elements. And then sorting time can be considerable (even of order of n), and the total running time can be larger than $O(n)$.