Sorting

Yu-Tai Ching
Department of Computer Science
National Chiao Tung University

- A “list” a collection of records, each record having one or more fields.
- fields used to distinguish among the records are known as “keys”.
- A telephone directory, there are 3 fields, name, phone number, and address
- Most of the time, name is the key.
- If you have the key (in this case, name) sorted (according to the lexcographic order), you have chance to search fast.

- To search fast, binary search,
- Interpolation Search: Comparing k with $a[i]$,
- $i = ((k - a[1].key)/(a[n].key - a[1].key)) * n$.
- where $a[1].key$ and $a[n].key$ are the smallest and the largest key in the list.

- US Internal Revenue Service (IRS).
- a list from employer (money pay to individual), another list from emloyee
- verify the lists to find out the differences.
- If they are not sorted, for one record in the employer list, we check all in the employee list.
- if they are sorted, can be done much faster.

Insertion Sort

- Insert a new record into a sorted sequence of i records
- resulting sequence of size $i + 1$ is also ordered.
- initially, sorted list of length 1, $a[0]$.
- insert $a[1]$ make the sorted list of length 2,
- insert $a[2]$, make it length 3, . . .
- draw a picture

- In the worst case, ith insertion needs i comparisons.
- Total comparisons needed are $\sum_{i=1}^{n-1} i$ = $O(n^2)$.
- What if you have a sorted list as the input?
- The computing time depends on the relative disorder in the input list.
- Record R_i is “left out of order” (LOO) iff $R_i < \max_{1 \leq j < i}\{R_j\}$.
- If k is the number of LOO records, the computing time is $O(kn)$.
- If k is small, computing time will be almost linear.
- Or if the length is small, say $n \leq 30$,

Variations

1. Binary Insertion Sort:
   - number of comparisons can be reduced,
   - but the number of records moves remains unchanged.
   - Still $O(n^2)$ time in the worst case.
2. Linked Insertion Sort:
   - We don't need to move data around,
   - but the number of comparison unchanged,
   - still $O(n^2)$ time in the worst case. Considered good if size of record is large.

Note that, there are two kinds of cost in any sorting algorithm, comparisons and moving data.

Quick Sort

- Very good (the best) average behavior.
- Select a “pivot” records from the records to be sorted. Usually the pivot is the first one in the list.
- “Partition” records to be sorted are reordered so that records with smaller key are placed to the left of the pivot and records with larger keys are to the right of pivot.
- note that pivot is at it final position.
- We have two subproblems with smaller size to be sorted, recursively sort them.
- draw a picture

Analysis of QuickSort

- Worst-case, $O(n^2)$, unbalance partition, if pivot is the first and if the input is sorted
- Best case, balance partition, let the problem can be solved in $T(n)$ time (a function T of input size).
- Since it is a balance partition, $T(n) = 2T(n/2) + cn$.
- Solve the recursion obtain the computing time.

$$
\begin{aligned}
T(n) &\leq cn + 2T(n/2) \\
&\leq cn + 2(cn/2 + 2T(n/4)) \\
&\leq 2cn + 4T(n/4) \\
&\cdots \\
&\leq cn \log_2 n + nT(1) \\
&= O(n \log n)
\end{aligned}
$$

- or draw a tree like structure.
- what if not perfect partition, say always 1/9 partition?

Lemma 7.1: Let $T_{avg}(n)$ be the expected time for function $QuickSort$ to sort list with n records. Then there exists a constant k such that $T_{avg}(n) \leq kn \log_e n$ for $n \geq 2$.

Proof Assume that the rank of j can be any number with equal probability.

Let $T_{avg}(n)$ be the average time to quick sort list of size n. Let the pivot be placed at position j. Then we get two subproblems of size $j-1$ and $n-j$. The average time to solve these two subproblems are respectively $T_{avg}(j-1)$ and $T_{avg}(n-j)$. So the total time is

$$T_{avg}(n) \leq cn + T_{avg}(j-1) + T_{avg}(n-j).$$

We assumed that j can be any number with equal probability, we have

$$\begin{aligned} T_{avg}(n) &\leq cn + \frac{1}{n}\sum_{j=1}^{n}(T_{avg}(j-1) + T_{avg}(n-j)) \\ &= cn + \frac{2}{n}\sum_{j=0}^{n-1} T_{avg}(j). \end{aligned}$$

Assume $T_{avg}(0) \leq b$ and $T_{avg}(1) \leq b$ for some constant b. We shall show that $T_{avg} \leq kn\log_e n$ for $n \geq 2$ and $k = 2(b+c)$. Proof is by induction on n.

Induction base: For $n = 2$, $T_{avg}(2) \leq 2c + 2b \leq kn \log_e 2$.
Induction hypothesis: Assume that $T_{avg} \leq kn \log_e n$ for $1 \leq n < m$.
Induction Step: From $T_{avg} \leq cn + \frac{2}{n} \sum_{j=0}^{n-1} T_{avg}(j)$ and induction hypothesis, we have

$$T_{avg}(m) \leq cm + \frac{4b}{m} + \frac{2}{m} \sum_{j=2}^{m-1} T_{avg}(j) \leq cm + \frac{4b}{m} + \frac{2k}{m} \sum_{j=2}^{m-1} j \log_e j.$$

Since $j \log_e j$ is an increasing function of j, we have

$$
\begin{aligned}
T_{avg}(m) &\leq cm + \frac{4b}{m} + \frac{2k}{m}\int_2^m x \log_e x dx \\
&= cm + \frac{4b}{m} + \frac{2k}{m}\left[\frac{m^2 \log_e m}{2} - \frac{m^2}{4}\right] \\
&= cm + \frac{4b}{m} + km\log_e m - \frac{km}{2} \\
&\leq km\log_e m, \text{ for } m \geq 2.
\end{aligned}
$$

Discussions on Quick Sort

- It is fast, move data only when necessary.
- Bad input, if the input is sorted, either increasing or decreasing order.
- a variation, median of three, pivot is the median of the first, the last, and the middle one in the list.
- another variation, randomized approach, use a random number generator to generate the pivot.
- Do the variations have bad input?
- What is the worst case time complexity for the variations?

How Fast Can We Sort?

- How fast can you sort n records? Can you sort n numbers faster than $O(n \log n)$?
- Try to argue the "lower bound" for the computational problem, given n records, determine a permutation such that the records are arranged in accending order according to the key.
- Basic idea of the argument, each comparsion gets a bit of information, how many bits of information you need to determine the permutation.
- Any sorting algorithm compares and moves data, thus takes more time than the number of bits information needed.
- Linear Decision Tree model.

Linear Decision Tree Model

- A Binary Tree
- Internal node, a comparison $a \leq b$, branches to left if YES, otherwise branches to right.
- external node, an answer, i.e., a kind of permutation.
- Figure 7.2

Theorem 7.1: Any decision tree that sorts n distinct elements has a height of at least $\log_2(n!) + 1$.

Corollary: Any algorithm that sorts only by comparisons must have a worst case computing time of $\Omega(n \log n)$.

Merge Sort- Merge

- Merge two sorted list. There are two sorted list l_1 and l_2, merge them to to get a new sorted list. Suppose that both of the length of l_1 and l_2 are $n/2$.
- Need another array of lenght n.
- Need time $\Theta(n)$.

Iterated Merge Sort

- Originally, there are n sorted list of length 1.
- Pairwise merge, get $n/2$ sorted list of length 2. Takes time $2(n/2)$.
- Pairwise merge the sorted list of length 2, get $n/4$ sorted list of length 4. Takes time $4(n/4)$.
- ...
- Merge the two sorted list of length $n/2$, get the final sorted list of length n, takes time n.
- There are $\log n$ iterations, total time is $O(n \log n)$.
- Figure 7.4

Recrusive Merge Sort

$MergeSort(L)$
{
    if ($Length(L) == 1$) return;
    divide L into two halves L_1 and L_2;
    $MergeSort(L_1)$;
    $MergeSort(L_2)$;
    $Merge(L_1, L_2)$;
}

- Figure 7.5
- Time $T(n) = 2T(n/2) + cn$. $T(n)$ = $\Theta(n \log n)$ in both average and worst cases.

Heap Sort

- Insertion sort $O(n^2)$ worst case, the best case is $O(n)$, average case is $O(n^2)$.
- Quick sort $O(n^2)$ worst case, $O(n \log n)$ average case.
- Merge Sort is optimal $O(n \log n)$ in the worst case and average case. But it is not “in place”.
- Now the heap sort is also optimal and it is in place.

- Selection Sort, Select the largest and put it at the end of the array.
- Can selection sort be improved?
- Use a heap to select the largest.
- Suppose that we have a max heap, the largest is at the root,
- remove the largest, swap with the one at the end of the arrya (size of the heap is reduced by 1), update the heap in $O(\log n)$ time.
- iterate the process until the heap is size 0, $O(n \log n)$ time in total.
- figure 7.7

- Input may not be a heap. We have to initialize the input to build a heap.
- Suppose that we have a complete binary tree of n nodes, half of them are leaves. These leaves are max heap of one node.
- Suppose that x and y are leaves and they are sibling, their parent is denoted z.

- the subtree tree rooted at z meets the condition: left subtree and right subtree are max heap but the root may not be the largest.
- update the heap.
- iterate this process from $\lfloor n/2 \rfloor$ down to 1 (the root) and we build the heap.
- The rough bound $O(n \log n)$, a carefully analysis lead to $O(n)$.

Sorting on Several Keys

- Sorting records on several keys, K^1, K^2, $\cdots$, K^r.
- K^1 the most significant key and K^r the least significant.
- A list of records R_1, R_2, $\cdots$, R_n is said to be sorted with respect to the keys K^1, K^2, $\cdots$, K^r iff for every pair of records i and j, $i \leq j$ and $(K_i^1, K_i^2, \cdots, K_i^r) \leq (K_j^1, K_j^2, \cdots, K_j^r)$.
- The r-tuple $(x_1, x_2, \cdots, x_r)$ is less than or equal to the r-tuple $y_1, y_2, \cdots, y_r$ iff either $x_i = y_i$, $1 \leq i \leq j$ and $x_{j+1} < y_{j+1}$ for some $j < r$, or $x_i = y_i$, $1 \leq i \leq r$.

Sorting Card Deck

- Sorting according to two keys, the suit (K^1) and the face value(K^2),
- K^1: ♣ $<$ ♢ $<$ ♡ $<$ ♠
- K^2: 2 < 3 < 4 < $\cdots$ < 10 < J < Q < K < A
- to sort them, a possible way, put same suit in a pile (sort according to k^1), then sort each pile according to K^2 the face value.
- Most-significant-digit-first (MSD) sort.
- Another possible way is the least-significant-digit-first (LSD) sorting, sort according to K_2 first then sort according to K_1.

- To sort, we DONOT compare, we set up “bins”, four bins for K^1, 13 bins for K^2.
- To sort integers, say 3 digits integer, we say there are 3 keys, we need 10 bins for each key.
- For example, we sort $8_1, 3_2, 5_3, 9_4, 2_5, 3_6, 7_7, 8_8, 4_9, 3_{10}$, the subscript indicates the order or the key in the input list.
- put them in the bin, walk through the bins to get the sorted sequence.
- Stable, 8_1 is placed before 8_8 in the sorted list, ...
- to LSD sort 3 digits integers, fig 7.9.
- correctness

Discussions

- Sort large records, uses pointers, and try to minimize the number of moving data records.
- The above mentioned algorithms are for internal sort, that means the records can be placed into memory.
- Suppose that we are going to sort huge number of records, we partition the records to several small subsets,
- each subset is internal sorted (make it a run), then store in disk.
- finally, merge the sorted subsets (runs) to get a final sorted list.

Optimal Merging of Runs

- External Sort, we have had runs, length of runs may be different.
- merging two runs r_1 and r_2 obtain a new run, the length is the (length of r_1) + (length of r_2), and it needs this amount of comparisons.
- Determine the merging sequence so that the total number of comparisons is minimized.
- Figure 7.27.

- The number of mergers that an individual records is involved in is the distance of the corresponding external node from the root.
- Thus the total merge time is obtained by summing the products of the run lengths and the distance from the root to the external node.
- the sum is called the weighted external path length.
- Figure 7.27, (a) weighted external path length $2 \cdot 3 + 4 \cdot 3 + 5 \cdot 3 + 15 \cdot 1 = 43$,
- (b) weighted external path length $2 \cdot 2 + 4 \cdot 2 + 2 + 5 \cdot 2 + a5 \cdot 2 = 52$.

- to determine optimal merging sequence = to find a tree that has the least weighted external path length.
- greedy approach, each time we find the best possible solution at that time, we then get the optimal solution.
- Sort and form pairs, (or using a heap structure),
- An example, $q_1 = 2$, $q_2 = 3$, $q_3 = 5$, $q_4 = 7$, $q_5 = 9$, and $q_6 = 13$.
- the same as Huffman code for compression.