Lecture 6: February 27, 2000

Today:

- Lower bounds of $\Omega(n \log n)$ for sorting
- Sorting in $O(n)$ time
  - Counting Sort
  - Radix Sort

Sorting Lower Bound

- limitation: only ordering operation allowed is **comparing** two elements, i.e. cannot look at elements' values!
  (covers all sorting algorithms so far)
- freebie: ignore all other operations (control, data movement etc)

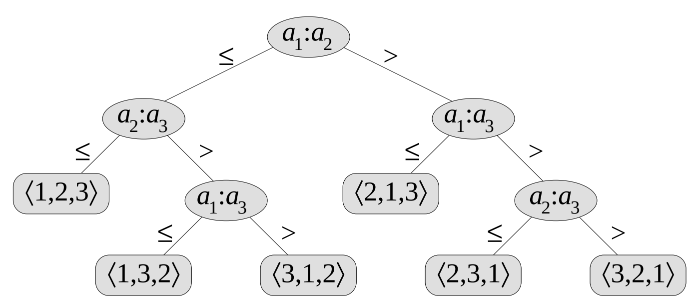
Model algorithm operation as **Decision Tree**:

- represents all comparisons for input of size n with elements $< a_1, a_2, ..., a_n >$
- control, data movement, and other aspects of algorithm ignored (for purposes of lower bound).

Decision Tree

- Each *internal* node holds $i : j$ for $i, j \in \{1, 2, \ldots, n\}$
  - Left subtree indicates subsequent comparisons if $a_i \leq a_j$
  - Right subtree indicates subsequent comparisons if $a_i > a_j$
- Each *leaf* holds a permutation $\langle \Pi(1), \Pi(2), \ldots, \Pi(n) \rangle$ indicating that the ordering $\left\langle a_{\Pi(1)}, a_{\Pi(2)}, \ldots, a_{\Pi(n)} \right\rangle$ has been established.

Decision Tree



Ex: 3 element sort $< a_1, a_2, a_3 >$
Note: all comparisons necessary!
(Adversary argument.)

Decision Tree

Decision tree can model a comparison sort:

- one tree for each distinct **input size** n
- view tree as if algorithm splits after each compare
- tree represents all possible execution traces
- algorithm running time = length of path

Worst-case running time = height of tree.
So... what's the height of the tree?

Lower Bound for Comparison Sorting

Theorem: Any decision tree that sorts n elements has height $\Omega(n \lg n)$.

Proof:

- The tree must have $\geq n!$ leaves (# of permutations $= n!$: n choices for 1st element, $n-1$ for the 2nd, etc.)
- A height h binary tree has at most 2^h leaves

Thus,

$$2^h \geq n! \geq (n/2)^{n/2}$$

and therefore

$$h = \Omega(n \log n)$$

Can we sort in $o(n \log n)$ time ?

E.g., how to sort $A[1 \ldots n]$ containing distinct elements from $\{1 \ldots n\}$?

for $i \leftarrow 1$ **to** k **do** $A[i] = i$

Sorting in linear time

- Counting sort - **no** key comparisons
- However, allowed to use key **values**!

Input: $A[1..n]$, where $A[j] \in 1, 2, \ldots, k$
(i.e., k is max. value found in $A[\cdot]$)

Output: $B[1..n]$, sorted

Uses: $C[1..k]$ auxiliary storage, size $O(k)$

Algorithm:

- count the number of times each element occurs (using $C[\cdot]$)
- find the position of each element in the sorted array (using $C[\cdot]$ again)
- permute accordingly (from $A[\cdot]$ to $B[\cdot]$)

Counting Sort

COUNTING-SORT(A, B, k)

1 **for** $i \leftarrow 1$ **to** k
2 **do** $C[i] \leftarrow 0$
3 **for** $j \leftarrow 1$ **to** *length*$[A]$
4 **do** $C[A[j]] \leftarrow C[A[j]] + 1$
5 $\triangleright$ $C[i]$ now contains # of elements = to i.
6 **for** $i \leftarrow 2$ **to** k
7 **do** $C[i] \leftarrow C[i] + C[i-1]$
8 $\triangleright$ $C[i]$ now contains # of elements $\leq i$.
9 **for** $j \leftarrow$ *length*$[A]$ **downto** 1
10 **do** $B[C[A[j]]] \leftarrow A[j]$
11 $C[A[j]] \leftarrow C[A[j]] - 1$

Counting Sort: Example

A:

4	3	5	8	1	8

C:

1	0	1	1	1	0	0	2

C':

1	1	2	3	4	4	4	6

B:

Why do we need line 11?

Counting Sort

Analysis

- $O(n+k)$ time
- If $n = o(k)$, then $O(k)$ total time.
  (Example: sort 10 numbers ranging from $1 \ldots 100,000$
- If $k = O(n)$, then $O(n)$ time.
  (Example: sort 100,000 numbers ranging from $1 \ldots 10$.
- Counting sort is **not** a comparison sort,
  since we use the *values* of sorted elements
- Useful property: sort is **stable**, i.e.,
  Input order is maintained among "equal" keys.

Radix Sort

- IBM card sorting algorithm run by
  machine **and** human operator
- multi-pass, **stable**, digit-by-digit sort
- each pass distributes one card stack into ten bins
- d passes required to sort numbers with d digits

Idea: Sort on *least* significant digit first:

Unsorted		Ones		Tens		Hundreds
329		720		720		329
457		355		329		355
657		436		436		436
839	$\rightarrow$	457	$\rightarrow$	839	$\rightarrow$	457
436		657		355		657
720		329		457		720
355		839		657		839

Radix Sort

Correctness: Induction on digit position

Assume numbers are sorted by low-order $t-1$ digits. Then, when we sort on digit t:

- Two numbers that differ in t^{th} digit are correctly sorted.
- Two numbers having same t^{th} digit are put in same order as they occur in the input to this pass, i.e., the correct order.

Radix Sort: Analysis

Running time depends on auxiliary stable sort

- If each digit lies in range 1 to k, and k is not too large, counting sort is obvious choice for auxiliary stable sort.
- Each pass over n d-digit numbers takes time $\Theta(n+k)$
- There are d passes, so total running time of radix sort is $\Theta(d(n+k))$
- When d is constant and $k = O(n)$, radix sort runs in time **linear** in n.

Radix Sort: Analysis

- Sort n words of b bits each.
- View each word as having $\frac{b}{r}$ digits of r bits each.

Ex: 32-bit word: | 8 bits | 8 bits | 8 bits | 8 bits |

$$b = 32, \quad r = 8, \quad \frac{b}{r} = 4 \text{ digits}$$

Therefore, radix sort makes

$$\frac{b}{r} = 4 \text{ passes.}$$

I.e., it processes each number 4 times.

Radix Sort: Analysis

- Each pass of radix sort takes $\Theta(n+2^r)$ time, since r bit values $\rightarrow$ keys in range $0, 1, \ldots, 2^r - 1$
- $T(n,b) = \Theta(\frac{b}{r}(n+2^r))$
- So, how should we choose r?
  - As we increase $r \rightarrow$ fewer passes are needed
  - But, as $r \gg \lg n$, counting sort time grows exponentially!
- Choose $r = \lg n$
  $\rightarrow T(n,b) = \Theta(\frac{bn}{\lg n})$

Again, where:
n is input size
b is number of bits in each word
r is number of bits in each digit

Radix Sort: Analysis

- Sorting numbers in range:

$$
\begin{array}{lll}
0 \text{ to } n-1 & (b = \lg n) & \text{in } \Theta(n) \text{ time.} \\
0 \text{ to } n^2-1 & (b = 2\lg n) & \text{in } 2\cdot\Theta(n) \text{ time.} \\
 & \vdots & \\
0 \text{ to } n^d-1 & (b = d\lg n) & \text{in } d\cdot\Theta(n) \text{ time.}
\end{array}
$$

Radix Sort: Example

- Sort 10^6 64-bit numbers:
  Radix sort: choose $r = \lg n = 20$; 4 passes (4 ops/#) Merge-, quicksort do $\Theta(n \lg n)$ work, $\Rightarrow$ 20 ops/#
- **In practice, radix sort is:**
  - fast for large inputs
  - simple to code

Concluding thoughts

- Model of computation is crucial!
- "Can only compare keys" implies $\Omega(n \log n)$ lower bound for sorting.
- Can look at digits of key, and key in small range yields $O(n)$ algorithm for sorting.
- In fact, can sort in $O(n \log\log n)$ time for **any** range (algorithm fairly complex)