Data Structures and Algorithms: Lecture 8

Barbara Morawska

September 18, 2018

Optimal time bounds for sorting

Comparison based sorting

- Worst case $O(n \lg n)$: MERGE-SORT, HEAPSORT
- Average case $O(n \lg n)$: QUICKSORT

We will show:

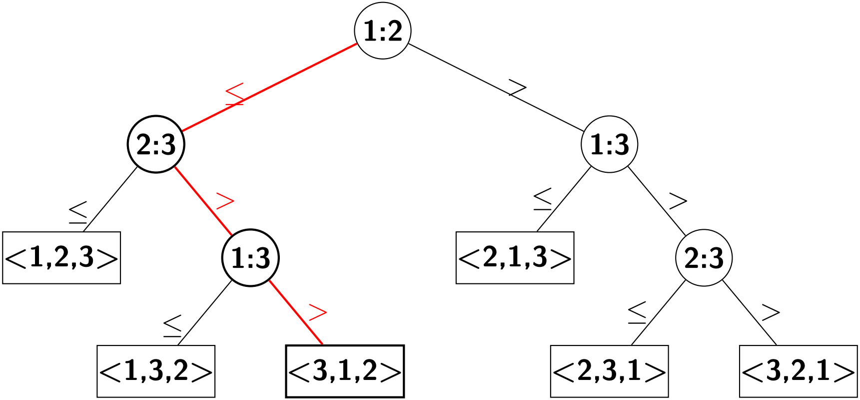
Any comparison based sorting has to use $\Omega(n \lg n)$ comparisong to sort n elements.

Hence these bounds are optimal for comparison based algorithms

Lower bound for comparison based sorting

- Assume all elements to be sorted are distinct.
- Assume only $a_i < a_j$ comparisons are made.
  (If this is false, then since $a_i \neq a_j$, we know that $a_i > a_j$.)
- View a run of a comparison based sorting algorithm as a decision tree...

Decision tree for Insertion Sort$(6, 8, 5)$



Decision tree model

Decision tree is:

- full binary tree
  A binary tree is a full binary tree iff each node has 2 or 0 children.
- internal nodes represent possible comparisons between the elements to be sorted,
- leaves represent the permutations of the sorted elements

Notice:

- Each of $n!$ possible permutations must appear in the leaves.
- Each of these leaves must be reachable from the root.

Lower bound for comparison based sorting

- The worst case: the largest number of comparisons,
- This is the longest path in the decision-tree, hence height of the tree.

Theorem

Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

Proof.

What is the height of a decision-tree in which all permutations of n elements are in the leaves?

- Let h be the height.
- Let l be the number of leaves. $l = n!$
- In the full binary tree $l = 2^h$. Hence $h = \lg l$
- $\lg n! = \Omega(n \lg n)$ (by Stirling's approximation)

□

Optimal algorithms

Corollary to Theorem 8.1

MERGE SORT and HEAPSORT are asymptotically optimal comparison sort algorithms.

Is it possible to sort in asymptotically linear time?

Counting sort

- Input: array of n numbers
- Assume that each number is in the range 0 to k, for some constant k
- Assume $k = O(n)$.
- Then we have sorting algorithm that runs in $O(n)$ time

Idea

- For each element x determine the number of elements smaller than x.
- Put x in the right place in the array.
- For example, if there are 17 elements smaller than x, put x in the slot 18'th, $A[18]$

Counting sort

We have to use additional space:

- $A[1..n]$ – input array,
- $B[1..n]$ – sorted output,
- $C[1..k]$ – temporary storage

Notice:

- Values in $A[1..n]$ are used as addresses in the array $C[1..k]$
- We assume that an address in an array is accessible in constant time!
- Addresses in the array C are sorted!
- There can be many elements in A with the same value.

Counting-Sort pseudocode

Procedure COUNTING-SORT(A, B, k)

Let $C[0..k]$ be a new array
for $i = 0$ *to* k **do**
    $C[i] = 0$
for $j = 1$ *to* $A.length$ **do**
    $C[A[j]] = C[A[j]] + 1$
    `//` $C[i]$ `contains the number equal to the number of elements equal to` i`, where` $0 \leq i \leq k$
for $i = 1$ *to* k **do**
    $C[i] = C[i] + C[i-1]$
`// Now` $C[i]$ `contains number of elements equal to the number of elements less than or equal to` i
for $j = A.length$ *downto* 1 **do**
    $B[C[A[j]]] = A[j]$
    $C[A[j]] = C[A[j]] - 1$

Running time

- Lines 1-3: $\Theta(k)$
- Loop populating C (lines 4–5) : $\Theta(n)$
- Loop counting elements (lines 7–8): $\Theta(k)$
- Loop populating B (lines 10–12): $\Theta(n)$

Overall the running time is: $\Theta(k+n)$ and
if $k = O(n)$, the running time is $\Theta(n)$.

Run COUNTING-SORT on $A =< 2, 5, 3, 0, 2, 3, 0, 3 >$, where $k = 5$.

Advantage of Counting Sort

Counting-Sort is stable

If in A elements with the same value appear, their relative order is preserved in the output B.

For example $A =< 1, 2, 1, 3 >$ then

- $A[1]$ will be copied to $B[1]$, $A[3]$ will be copied to $B[2]$ and
- not $A[1]$ to $B[2]$ and $A[3]$ to $B[1]$.

Radix sort

Idea

- To sort n binary numbers with k digits each
- (but can be used to records with sortable k fields, keys)
- First sort with respect to the least significant digit
- if input is binary numbers, then now you have the numbers with 0 at the end followed by those with 1 at the end
- Sort with respect to the next column, but use a stable sort, so that the sorted column remains relatively sorted.

Possible application

Use Radix-Sort to sort dates: Year, Month, Day

Pseudocode

- We assume that each element of an array A has d digits,
- and digit 1 is the lowest-order digit,
- and digit d is the highest-order digit

Procedure RADIX-SORT(A, d)

```
for i = 1 to d do
    Use a stable sort to sort A on digit i
```

Show how RADIX-SORT works on
$A =< 329, 457, 657, 839, 436, 720, 355 >$

Running time

Lemma 8.3

Given d-digit numbers in which each digit takes up to k possible values, RADIX-SORT correctly sorts these numbers in $\Theta(d(n+k))$ time, if the stable sort uses $\Theta(n+k)$ time.

Proof.

Obvious. $\square$

If d is a constant, and $k = O(n)$, then RADIX-SORT runs in linear time ($O(n)$)

How to break keys into digits?

Lemma 8.4

- Given n numbers, each b-bit long
- Let r be a positive integer, $r \leq b$.

RADIX-SORT correctly sorts the numbers in time

$$\Theta((b/r)(n + 2^r))$$

if the stable sort it uses takes time $\Theta(n + k)$, for inputs in the range 0 to k.

Proof.

- Each key has $d = \lceil b/r \rceil$ digits of r bits each.
- Each digit has r places, hence can represent a number from 0 to $2^r - 1$
- We can use COUNTING-SORT with $k = 2^r - 1$.

$\square$

Example

We have to sort 32-bit words.

- Each word has 8-bit digits
- $b = 32$, $r = 8$, $k = 2^8 - 1 = 255$, $d = b/r = 4$
- Each pass of COUNTING-SORT takes $\Theta(n + k) = \Theta(n + 255)$
- There are d passes.
- Hence $\Theta(d(n + 2^r)) = \Theta((b/r)(n + 2^r)) = \Theta(4(n + 255))$

Problem: when we use RADIX-SORT we want to minimize $(b/r)(n + 2^r)$

- Notice: if $b < \lfloor \lg n \rfloor$, then for any r ($r \leq b$):
  $n + 2^r \leq n + 2^{\lg n} = n + n = \Theta(n)$
- Hence if $b < \lfloor \lg n \rfloor$ choose $r = b$, then $b/b(n + 2^r) = \Theta(n)$.
- If $b \geq \lfloor \lg n \rfloor$, choose $r = \lfloor \lg n \rfloor$. Then
  $\Theta((b/r)(n + 2^r)) = \Theta((b/r)(n + n)) = \Theta((bn)/\lg n)$

Minimize $(b/r)(n+2^r)$

In the last case: If $b \geq \lfloor \lg n \rfloor$, choose $r = \lfloor \lg n \rfloor$. Then $\Theta((b/r)(n+2^r)) = \Theta((b/r)(n+n)) = \Theta((bn)/\lg n)$

- If $r > \lfloor \lg n \rfloor$ then 2^r increases and we have the running time: $\Omega((b/r)(n+2^r))$
- If $r < \lfloor \lg n \rfloor$ then b/r increases and we are back to $\Theta(n)$.

Compare Radix-Sort and Quicksort in practice.

Bucket-Sort

- Similar to Counting-Sort in that is uses a kind of "direct addressing"
- We are sorting numbers in an n element array A
- the numbers in A are all in the interval $[0, 1)$
- Like in Counting-Sort we use an auxiliary array $B[0..n-1]$ (array of buckets)
- Each element in B is a linked list.

Pseudocode

Procedure BUCKET-SORT(A)

```
n = A.length
Let B[0..n − 1] be a new array
for i = 0 to n − 1 do
    make B[i] an empty list
for i = 1 to n do
    insert A[i] into list B[⌊n · A[i]⌋]
for i = 0 to n − 1 do
    sort list B[i] with INSERTION-SORT
Concatenate the lists B[0], . . . , B[n − 1] together in this order
```

Run BUCKET-SORT on
$A =< .78, .17, .39, .26, .72, .94, .21, .12, .23, .68 >$, where $n = 10$.

Explanations

Consider $A[i]$ and $A[j]$ (for any indexes i, j, $i \neq j$) from the input array, such that $A[i] \leq A[j]$. Then:

- $\lfloor n \cdot A[i] \rfloor \leq \lfloor n \cdot A[j] \rfloor$
- Hence there are two cases:
  1. $A[i]$ and $A[j]$ are put into the same bucket ($\lfloor n \cdot A[i] \rfloor = \lfloor n \cdot A[j] \rfloor$)
  2. $A[i]$ is in the lower bucket
- In the first case $A[i]$ and $A[j]$ are put in order by INSERTION-SORT (line 7-8)
- In the second case they are concatenated in the right order (line 9).

Running time

- The lines 1,2 take constant time
- The loop in lines 3-4 takes $\Theta(n)$ time (initialization of B)
- The loop in lines 5-6 takes $\Theta(n)$ time (insertion of values of A into B)
- The loop in lines 7-8 requires n times Insertion-Sort
- Concatenation of lists in line 9 takes $\Theta(n)$ time.

Let n_i be a random variable that is a function from a basic event to the number of elements placed in $B[i]$.
Then running time is:

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

Average case for Bucket-Sort

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

Average time

$$\begin{aligned} E[T(n)] &= E[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)] \\ &= \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)] \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2])] \end{aligned}$$

What is the expectation of n_i^2

n_i - number of elements put in $B[i]$

- Let X_{ij} be an indicator variable for the event that $A[j]$ is put in bucket $B[i]$
- $X_{ij} = I\{A[j] \text{ falls in bucket } i\}$
- $X_{ij}(A) = 1$ with probability $1/n$
- In notation: $Pr\{X_{ij} = 1\} = 1/n$
- $n_i = \sum_{j=1}^{n} X_{ij}$

What is the expectation of n_i^2

Hence:
$$\begin{aligned} E[n_i^2] &= E[(\sum_{j=1}^{n} X_{ij})^2] \\ &= E[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}] \\ &= E[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\sum_{1\le k\le n, k\ne j} X_{ij}X_{ik}] \\ &= \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{1\le j\le n}\sum_{\substack{1\le k\le n,\\ k\ne j}} E[X_{ij}X_{ik}] \end{aligned}$$

Notice:

- $E[X_{ij}^2] = 1^2 \cdot 1/n + 0^2(1 - 1/n)$ (by definition)
- Since X_{ij} and X_{ik} are independent if $j \neq k$, $E[X_{ij}X_{ik}] = E[X_{ij}]E[X_{ik}] = 1/n \cdot 1/n = 1/n^2$
- Hence $E[n_i^2] = n \cdot 1/n + n(n-1) \cdot 1/n^2 = 1 + \dfrac{n-1}{n} = 2 - 1/n$

Average time

Hence average case running time of BUCKET-SORT is:

$$T(n) = \Theta(n) + n \cdot O(2 - 1/n) = \Theta(n)$$