Algorithms and Data Structures for Mathematicians

Lecture 5: Sorting

Peter Kostolányi
`kostolanyi at fmph and so on`
Room M-258

26 October 2017

Sorting Algorithms Covered So Far

Worst-case time complexity $\Theta(n^2)$:

- Insertion Sort
- Tree Sort ("basic" binary search trees)

Worst-case time complexity $\Theta(n \log n)$:

- Merge Sort
- Heap Sort
- Tree Sort (balanced binary search trees)

Quick Sort

Uses a divide and conquer strategy in a different way than merge sort:

Divide the array $\langle a[f], \ldots, a[l] \rangle$ as follows:

1. Choose a pivot, e.g., as $piv \leftarrow a[l]$
2. Rearrange the array so that for some $f \leq m \leq l$:
   - $a[i] \preceq piv$ for $i = f, \ldots, m-1$
   - $a[m] = piv$
   - $a[i] \succeq piv$ for $i = m+1, \ldots, l$

Conquer the subproblems by recursively applying the procedure to $\langle a[f], \ldots, a[m-1] \rangle$ and $\langle a[m+1], \ldots, a[l] \rangle$ (simply return for arrays with 1 or 0 elements)

Combine phase is not needed here

Quick Sort

QUICKSORT(a, f, l):

```
if f ≥ l then return
else
    m ← DIVIDE(a, f, l);
    QUICKSORT(a, f, m − 1);
    QUICKSORT(a, m + 1, l);
end
```

DIVIDE(a, f, l):

```
piv ← a[l]; j ← f;
for i ← f to l − 1 do
    if a[i] ⪯ piv then
        a[j] ↔ a[i];
        j ← j + 1;
    end
end
a[j] ↔ a[l];
return j;
```

Worst-Case Time Complexity of Quick Sort

- Suppose that $\langle a[1], \dots, a[n] \rangle$ is already sorted
- Then for each $f < l$, the array $\langle a[f], \dots, a[l] \rangle$ is divided as follows:
  - $m = l$
  - No elements of the array are exchanged
  - One recursive call for $\langle a[f], \dots, a[l-1] \rangle$ and one for an empty array
- Time complexity on such input: $T_s(n) = T_s(n-1) + T_s(0) + \Theta(n)$
- Hence, $T_s(n) = \Theta(n^2)$
- For worst-case complexity, this implies $T(n) = \Omega(n^2)$

Worst-Case Time Complexity of Quick Sort

- We shall prove that $T(n) = O(n^2)$ as well (hence, $T(n) = \Theta(n^2)$)
- By definition of worst-case complexity:
  $$T(n) \leq \max_{0 \leq k \leq n-1} (T(k) + T(n-k-1)) + c_1 n$$
  for some $c_1 > 0$
- Let $c_2 > 0$ be such that $T(0) \leq c_2$, $T(1) \leq c_2$, and $2c_2 n \geq 2c_2 + c_1 n$ for $n \geq 2$
- We shall prove that $T(n) \leq c_2(n^2 + 1)$ for all $n \geq 0$
- Induction on n:
  - $T(0) \leq c_2 \leq c_2(0^2 + 1)$ and $T(1) \leq c_2 \leq c_2(1^2 + 1)$
  - Now, let $n \geq 2$
  - By the induction hypothesis: $T(k) \leq c_2(k^2 + 1)$ for $k = 0, \ldots, n-1$
  - $T(n) \leq \max_{0 \leq k \leq n-1} \left(c_2(k^2 + 1) + c_2((n-k-1)^2 + 1)\right) + c_1 n$
  - $T(n) \leq \max_{0 \leq k \leq n-1} \left(c_2 k^2 + c_2(n-k-1)^2\right) + 2c_2 + c_1 n$
  - $c_2 x^2 + c_2(n-x-1)^2$ has positive second derivative on $(0, n-1)$
  - The maximum is attained for $x = 0$ and $x = n-1$
  - $T(n) \leq c_2(n-1)^2 + 2c_2 + c_1 n = c_2 n^2 - 2c_2 n + c_2 + 2c_2 + c_1 n \leq c_2(n^2+1)$

Why *Quick* Sort?

- The worst-case time complexity of quick sort is $\Theta(n^2)$
- Quick sort is much worse than insertion sort on already sorted inputs!

What is so quick about quick sort?

- Good average-case performance
- If a pivot is not chosen as $a[l]$, but randomly: no adversary can choose an input with bad running time (randomised quick sort)
- Quick sort is particularly well suited for practical implementation (cache efficiency, etc.)

Randomised Quick Sort

- "Classical" quick sort: pivot is set to $a[l]$
- Randomised quick sort: pivot is set to a randomly chosen element
- Randomised DIVIDE: randomly choose an element, exchange it with $a[l]$, and call the "ordinary" DIVIDE procedure

RQUICKSORT(a, f, l):

if $f \geq l$ **then return**
else
  $m \leftarrow$ RDIVIDE(a, f, l);
  RQUICKSORT($a, f, m-1$);
  RQUICKSORT($a, m+1, l$);
end

RDIVIDE(a, f, l):

$index \leftarrow$ RANDOM(f, l);
$a[index] \leftrightarrow a[l]$;
return DIVIDE(a, f, l);

Analysis of Randomised Quick Sort

- Let X be a random variable counting comparisons "$a[i] \preceq piv$" in DIVIDE during the execution of $\mathrm{RQUICKSORT}$
- The total running time is $O(n+X)$ (at most n calls of DIVIDE)

We shall compute the expected value of X

- Let $a = \langle a[1], \dots, a[n] \rangle$ contain elements $s_1 \preceq \dots \preceq s_n$
- For each $i \neq j$: s_i is compared with s_j at most once (each element can be a pivot at most once)
- Let $X_{i,j} = 1$ if s_i is compared with s_j and $X_{i,j} = 0$ otherwise

We obtain:

$$E[X] = E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{i,j}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{i,j}] =$$

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr[s_i \text{ is compared with } s_j]$$

Analysis of Randomised Quick Sort

- It remains to compute the probability that s_i is compared with s_j
- Let $i < j$ and let us denote $S[i, j] := \{s_i, \ldots, s_j\}$
- If s_i is a first element of $S[i, j]$ chosen as a pivot, then s_i is compared with $s_{i+1}, \ldots, s_j$
- If s_j is a first element of $S[i, j]$ chosen as a pivot, then s_j is compared with $s_i, \ldots, s_{j-1}$
- If some other element of $S[i, j]$ is a first chosen pivot, then s_i and s_j are not compared at all

$$\begin{aligned}
\Pr[s_i \text{ is compared with } s_j] &= \Pr[s_i \text{ is a first pivot chosen from } S[i,j]] + \\
&+ \Pr[s_j \text{ is a first pivot chosen from } S[i,j]] = \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}
\end{aligned}$$

Analysis of Randomised Quick Sort

As a result:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{t=1}^{n-i} \frac{2}{t+1} < \sum_{i=1}^{n-1} \sum_{t=1}^{n} \frac{2}{t} =$$

$$= \sum_{i=1}^{n-1} O(\log n) = O(n \log n)$$

- The expected time complexity thus is $O(n \log n)$
- It is not hard to prove that the best-case time complexity is $\Theta(n \log n)$
- Hence, the expected time complexity is $\Theta(n \log n)$

About Randomised Algorithms

Randomised quick sort is a Las Vegas randomised algorithm:

- It always produces a correct output
- The running time on a given input is a random variable

This is in contrast to Monte Carlo randomised algorithms:

- These may produce incorrect output
- The probability of error is typically small
- Usually can be made so small that it is irrelevant
- Often more efficient than their best known deterministic counterparts

Sorting by Comparison

The sorting algorithms described so far are comparison sorts

- The produced output depends solely on a sequence of comparisons (of type $a[i] \preceq a[j]$) between the array elements
- There might be some other actions than comparisons and exchanges, but these depend just on the comparisons made up to the point

We shall now prove the following lower bound:

- The worst-case time complexity of any comparison sort is $\Omega(n \log n)$
- For instance merge sort and heap sort are thus optimal (among comparison sorts)

Lower Bound for Sorting by Comparison

Let $\langle a[1], \ldots, a[n] \rangle$ be an array

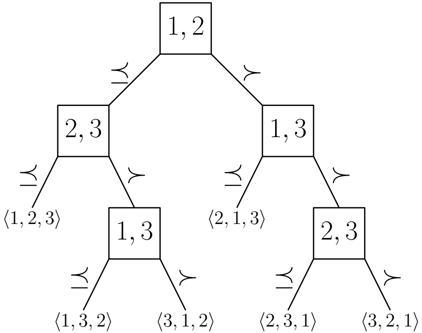
- A comparison sort makes a sequence of comparisons $a[i] \preceq a[j]$
- Comparisons $a[\cdot] \succeq a[\cdot]$, $a[\cdot] = a[\cdot]$, $a[\cdot] \prec a[\cdot]$, and $a[\cdot] \succ a[\cdot]$ can be "expressed in terms of" a constant number of $a[\cdot] \preceq a[\cdot]$

Decision trees:

- Interior nodes are pairs of indices of elements compared
- Leaves are sorted arrays produced on output

Lower Bound for Sorting by Comparison

Example of a decision tree for $n = 3$:



Lower Bound for Sorting by Comparison

If all input elements are distinct:

- A comparison sort has to be able to produce $n!$ different outputs
- Decision trees for inputs of length n need to have at least $n!$ reachable leaves
- A maximum depth of a reachable leaf has to be at least

$$\log(n!) = \log n + \log(n-1) + \ldots + \log 1 \geq \frac{n}{2} \log \frac{n}{2} = \Omega(n \log n)$$

- The worst-case time complexity is $\Omega(n \log n)$ as well

Sorting in "Linear Time"

We shall describe a sorting algorithm that:

- Is not comparison sort
- Makes relatively strong assumptions about the universe of possible array elements
- Runs in time that can be considered linear worst-case under some circumstances
- This is just an example, there are some other "similar" algorithms as well

Counting Sort

- Assumes that array elements are integers between 0 and some k
- For each i from 0 to k:
  - The algorithm counts the number of array elements equal to i
  - Then counts the number of elements less than or equal to i
  - Uses these values to "create" the sorted array

Counting Sort

COUNTINGSORT(a):

Create arrays $b = \langle b[1], \ldots, b[n] \rangle$ and $c = \langle c[0], \ldots, c[k] \rangle$;
for $i \leftarrow 0$ **to** k **do** $c[i] \leftarrow 0$;
for $j \leftarrow 1$ **to** n **do** $c[a[j]] \leftarrow c[a[j]] + 1$;
for $i \leftarrow 0$ **to** k **do** $c[i] \leftarrow c[i] + c[i-1]$;
for $j \leftarrow n$ **downto** 1 **do**
    $b[c[a[j]]] \leftarrow a[j]$;
    $c[a[j]] \leftarrow c[a[j]] - 1$;
end
return b;

- If elements have no "satellite data", the final for cycle can be executed in increasing order as well
- The pseudocode shown above results in a stable sorting algorithm
- Time complexity: $\Theta(n + k)$