Design & Analysis of Algorithms
COMP 482 / ELEC 420
John Greiner

Sorting
Should already know some sorting algorithms:
E.g., Insertion sort, Selection sort, Quicksort, Mergesort, Heapsort

We’ll concentrate on ideas not seen in previous courses:
• Lower bounds on general-purpose sorting
• Quicksort probabilistic analysis
• Special-purpose linear-time sorting

To do:
[CLRS] 7-8
#3

Comparison Trees
Comparisons used to determine order of 3 distinct elements. Leaves correspond to all possible permutations.

How do Comparison Trees Relate to Sorting?
• Any sorting algorithm must be able to reorder any permutation.
• Sorting algorithm’s behavior corresponds to some comparison tree.

Lower Bounds on Sorting
How many leaves in a comparison tree?
How many levels?

So, any general sorting algorithm must make \( \Omega(n \lg n) \) comparisons on at least some inputs.

Quicksort – Immutable
qsort(A):
if \(|A| \leq 1\)
    return A
else
    pivot = some element of A
    L = [x in A : x < pivot]
    G = [x in A : x > pivot]
    return qsort(L) + pivot + qsort(G)
Quicksort – Mutable

Algorithm in [CLRS] 7.

Advantages:
- Constant factor less time.
- Constant factor less space.

Disadvantage:
- More difficult to understand.

Quicksort Pivot Best Case

Time depends on a good (lucky?) choice of pivot.

? Best case? ?

? What is resulting recurrence & bound? ?

T(n) = 2T(n/2) + Θ(n) = Θ(n lg n)

Quicksort Pivot Worst Case

Time depends on a good (lucky?) choice of pivot.

? Worst case? ?

? What is resulting recurrence & bound? ?

T(n) = T(n-1) + Θ(n) = Θ(n²)

Quicksort Average Case Intuition

Average case is more like the best case than the worst case.

Two interesting cases for intuition:

1. Any sequence of partitions with the same ratios, such as 1/2:1/2 (the best case), 1/3:2/3, or even 1/100:99/100.
   - As have previously seen, the recursion tree depth is still logarithmic, which leads to the same bound.
   - Thus, the “good” cases don’t have to be that good.

Quicksort Worst Case

For deterministic pivot-choosing algorithms:
- O(1): Can be unlucky → O(n²) quicksort worst case
- O(n): Can find median (as we’ll see soon) → O(n log n) quicksort

We took shortcut by assuming the 0 & all-but-1 split is worst.

Intuitively obvious, but could prove this, e.g.,

T(n) = max_{q=0..n-1} (T(q) + T(n-q-1)) + Θ(n)

...which can be solved to...

T(n) = Θ(n²)

Quicksort Average Case Intuition

2. Sequence of alternating worst case and best case partitions.
   - Each pair of these partitions behaves like a best case partition, except with higher overhead.

   0 All-but-1
   -Half

   All-but-1
   -Half

- Thus, can tolerate having some bad partitions.
Quicksort Average Case Overview

Already have \( \Omega(n \log n) \) bound.

Want to obtain \( O(n \log n) \).

Can overestimate in analysis.

Always look for ways to simplify!

Quicksort Average Case: Partitioning

Observe: Partitioning dominates Quicksort’s work.

- Partitioning includes the comparisons – the interesting work.
- Every Quicksort call partitions – except the \( O(1) \)-time base cases.
- Partitioning more expensive than joining.

\(? \) How many partitions are done in the sort? \( ? \)

\[ n - 1 = O(n) \]

Observe: Comparisons dominate partitioning work.

- Each partition’s time \( \propto \) that partition’s #comparisons.

So, concentrate on time spent comparing.

Quicksort Average Case: Comparisons

\# of comparisons in partitions in \( n \)-element Quicksort

\[ C(n) = \ldots \]

\# of comp. in this partition?

\[ n - 1 \]

\# of comp. in two recursive calls?

\[ C(i) + C(n - i - 1) \]

Average this over all partition sizes.

\[ \frac{1}{n} \sum_{i=0}^{n-1} (C(i) + C(n - i - 1)) \]

\[ C(n) = n - 1 + \frac{1}{n} \sum_{i=0}^{n-1} (C(i) + C(n - i - 1)) \]

\[ = O(n \log n) \]

Quicksort Average Case: Comparisons

Let \( X_{ij} = I(z_i \text{ is compared to } z_j) \)

Total comparisons

\[ X = \sum_{i=1}^{n} \sum_{j=1}^{i} X_{ij} \]

Each pivot is selected at most once, so each \( z_i, z_j \) pair is compared at most once.

\[ E[X] = \sum_{i=1}^{n} \sum_{j=1}^{i} E[X_{ij}] \]

\[ = \sum_{i=1}^{n} \sum_{j=1}^{i} \text{Pr}(X_{ij}) \]

by linearity of expectation

\[ = \sum_{i=1}^{n} \sum_{j=1}^{i} \text{Pr}(z_i \text{ is compared to } z_j) \]

by definition

Quicksort Average Case: Comparisons

What is this probability?

Consider arbitrary \( i, j \) and corresponding \( Z_i \).

\( Z_i \) need not correspond to a partition executed during the sort.

Claim: \( z_i \) and \( z_j \) are compared \( \leftrightarrow \) either is the first element in \( Z_i \)

to be chosen as a pivot.

Proof: Which is first element in \( Z_i \) to be chosen as pivot?

- If \( z_i \), then that partition must start with at least all the elements in \( Z_i \).
- Then \( z_i \), compared with all the elements in that partition (except itself), including \( z_i \).
- If \( z_j \), similar argument.
- If something else, the resulting partition puts \( z_i \) and \( z_j \) into separate sets (without comparing them), so that no future Quicksort or partition call will consider both of them.
**Quicksort Average Case: Comparisons**

Now, compute the probability:

\[
\Pr(z_i \text{ is compared to } z_j) = \Pr(z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij})
\]

- \[= \Pr(z_i \text{ is first pivot chosen from } Z_{ij}) \text{ just explained} \]
- \[= \Pr(z_j \text{ is first pivot chosen from } Z_{ij}) \text{ mutually exclusive possibilities} \]

\[
= \frac{1}{(j-i+1)} + \frac{1}{(j-i+1)} = \frac{2}{(j-i+1)}
\]

**Quicksort Average Case: Total**

Plug this back into the sum:

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

- \[\leq \sum_{i=1}^{n-1} \sum_{k=2}^{n-i} \frac{2}{k} \text{ Simplify with a change of variable, } k = j-i+1. \]
- \[< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \text{ Simplify and overestimate by adding terms.} \]

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

**Quicksort Input Distribution**

Are all inputs equally likely in practice?

**Linear-Time Sorting**

In limited circumstances, can avoid comparison-based sorting, & thus do better than previous lower bound!

Must rely on some restriction on inputs.

**Counting Sort**

Limit data to a small discrete range: e.g., 0,….5.

Let \(m\) = size of range.

```
Input= 3 0 0 2 3 3
Count instances of each possible element: Count= 2 1 2 1 0 1
Produce Count[i] instances of each i: Output= 0 1 2 2 1 3
```

Limited usefulness, but the simplest example of a non-comparison-based sorting algorithm.

**Counting Sort**

\(csort(A,n):\)

- \(^*\) Count number of instances of each possible element. \(^*/\)
- \(\text{Count}[0...m-1] = 0\)
- \(\text{For index } i = 0 \text{ to } n-1\)
- \(\text{Count}[A[index]] += 1\)

- \(^*\) Produce Count[i] copies of each i. \(^*/\)
- \(\text{index } = 0\)
- \(\text{For copies } = 0 \text{ to Count[A[i]]}\)
- \(A[\text{index}] = i\)
- \(\text{index += 1}\)

\[\Theta(m) = \Theta(n) \text{ time, when } m \text{ taken to be a constant.}\]
Bucket Sort

Limit data to a continuous range: e.g., [0,6).
Let m = size of range.

Input: 2.1, 5.3, 0.5, 2.0, 1.9, 0.3

Create n buckets, for equal-sized subranges

<0.86
<1.71
<2.57
<3.43
<4.29
<5.14
<6.00

Buckets:

For each
• Calculate bucket = \[d \cdot \frac{n}{m}\]
• Insert to bucket's list.

2.1 \rightarrow \text{Bucket} \lfloor 2.1 \cdot \frac{7}{6} \rfloor = \lfloor 2.45 \rfloor = 2

Bucket Sort Analysis

<table>
<thead>
<tr>
<th>Worst Case</th>
<th>Best Case</th>
<th>Average Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Items per Bucket</td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td>Total Time</td>
<td>O(n^2)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

Radix Sort

Limit input to fixed-length numbers or words.
Represent symbols in some base b.
Each input has exactly d “digits”.

Sort numbers d times, using 1 digit as key.
Must sort from least-significant to most-significant digit.
Must use any “stable” sort, keeping equal-keyed items in same order.

Radix Sort Example

Input data:

\[a b a b a c a c b a \ b a c a c b a a b c b a a b c a c b a\]

Pass 1: Looking at rightmost position.

Place into appropriate pile.

a b c

Radix Sort Example

Pass 1: Looking at rightmost position.

Join piles.

a b c
Radix Sort Example

Pass 2: Looking at next position.

Place into appropriate pile.

a  b  c

Radix Sort Example

Pass 2: Looking at next position.

Join piles.

a  b  c

Radix Sort Example

Pass 3: Looking at last position.

Place into appropriate pile.

a  b  c

Radix Sort Example

Pass 3: Looking at last position.

Join piles.

a  b  c

Radix Sort Algorithm

\[ \text{rsort}(A,n): \]
\[
\begin{array}{l}
\text{For } j = 0 \text{ to } d-1 \\
\quad \text{/* Stable sort } A, \text{ using digit position } j \text{ as the key. */}
\text{For } i = 0 \text{ to } n-1 \\
\quad \text{Add } A[i] \text{ to end of list } ((A[i] >> j) \mod b)
\end{array}
\]
\[ A = \text{Join lists } 0...b-1 \]
\[ \Theta(dn) \text{ time, where } d \text{ is taken to be a constant.} \]