The comparison sorting problem

Assume we have \( n \) comparable elements in an array and we want to rearrange them to be in increasing order.

**Input:**
- An array \( A \) of data records
- A key value in each data record
- A comparison function (consistent and total)

**Effect:**
- Reorganize the elements of \( A \) such that for any \( i \) and \( j \), if \( i < j \) then \( A[i] \leq A[j] \)
- (Also, \( A \) must have exactly the same data it started with)
- Could also sort in reverse order, of course

An algorithm doing this is a comparison sort

---

**Sorting: The Big Picture**

Surprising amount of neat stuff to say about sorting:

- **Simple algorithms:** \( O(n^2) \)
- **Fancier algorithms:** \( O(n \log n) \)
- **Comparison lower bound:** \( \Omega(n \log n) \)
- **Specialized algorithms:** \( O(n) \)
- **Handling huge data sets**

**Insertion sort**
**Selection sort**
**Shell sort**
**Heap sort**
**Merge sort**
**Quick sort**
**Bucket sort**
**Radix sort**
**External sorting**

---

**Divide and conquer**

Very important technique in algorithm design

1. Divide problem into smaller parts
2. Independently solve the simpler parts
   - Think recursion
   - Or potential parallelism
3. Combine solution of parts to produce overall solution

---

**Divide-and-Conquer Sorting**

Two great sorting methods are fundamentally divide-and-conquer

1. **Merge sort**:
   - Sort the left half of the elements (recursively)
   - Sort the right half of the elements (recursively)
   - Merge the two sorted halves into a sorted whole

2. **Quick sort**:
   - Pick a "pivot" element
   - Divide elements into less-than pivot and greater-than pivot
   - Sort the two divisions (recursively on each)
   - Answer is sorted-less-than then pivot then sorted-greater-than

---

**Quick sort**

- A divide-and-conquer algorithm
  - Recursively chop into two pieces
  - Instead of doing all the work as we merge together, we will do all the work as we recursively split into halves
  - Unlike merge sort, does not need auxiliary space
- \( O(n \log n) \) on average \( \Theta \), but \( O(n^2) \) worst-case \( \Theta \)
- Faster than merge sort in practice?
  - Often believed so
  - Does fewer copies and more comparisons, so it depends on the relative cost of these two operations!
Quicksort Overview

1. Pick a pivot element
2. Partition all the data into:
   A. The elements less than the pivot
   B. The pivot
   C. The elements greater than the pivot
3. Recursively sort A and C
4. The answer is, “as simple as A, B, C”

Think in Terms of Sets

Example, Showing Recursion

Details

Have not yet explained:

- How to pick the pivot element
  - Any choice is correct; data will end up sorted
  - But as analysis will show, want the two partitions to be about equal in size
- How to implement partitioning
  - In linear time
  - In place

Pivots

- Best pivot?
  - Median
  - Halve each time

- Worst pivot?
  - Greatest/least element
  - Problem of size n - 1
  - $O(n^2)$

Potential pivot rules

While sorting $\text{arr}$ from $\text{lo}$ to $\text{hi}$ - 1 ...

- Pick $\text{arr[lo]}$ or $\text{arr[hi-1]}$
  - Fast, but worst-case occurs with mostly sorted input
- Pick random element in the range
  - Does as well as any technique, but (pseudo)random number generation can be slow
  - Still probably the most elegant approach
- Median of 3, e.g. $\text{arr[lo]}$, $\text{arr[hi-1]}$, $\text{arr[(hi+lo)/2]}$
  - Common heuristic that tends to work well
**Partitioning**

- Conceptually simple, but hardest part to code up correctly
  - After picking pivot, need to partition in linear time in place

- One approach (there are slightly fancier ones):
  1. Swap pivot with \( \text{arr}[\text{lo}] \)
  2. Use two fingers \( i \) and \( j \), starting at \( \text{lo}+1 \) and \( \text{hi}-1 \)
  3. while \( (i < j) \)
     - if \( \text{arr}[j] > \text{pivot} \) \( j-- \)
     - else if \( \text{arr}[i] < \text{pivot} \) \( i++ \)
     - else swap \( \text{arr}[i] \) with \( \text{arr}[j] \)
  4. Swap pivot with \( \text{arr}[i] \)

*skip step 4 if pivot ends up being least element

---

**Example**

- Step one: pick pivot as median of 3
  - \( \text{lo}=0, \text{hi}=10 \)

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
8 & 1 & 4 & 9 & 0 & 3 & 5 & 2 & 7 & 6 \\
\end{array}
\]

- Step two: move pivot to the \( \text{lo} \) position

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
6 & 1 & 4 & 9 & 0 & 3 & 5 & 2 & 7 & 8 \\
\end{array}
\]

---

**Quick sort visualization**


---

**Analysis**

- Best-case: Pivot is always the median
  - \( T(0)=T(1)=1 \)
  - \( T(n)=2T(n/2)+n \) -- linear-time partition
  - Same recurrence as merge sort: \( O(n \log n) \)

- Worst-case: Pivot is always smallest or largest element
  - \( T(0)=T(1)=1 \)
  - \( T(n)=T(n-1) + n \)
  - Basically same recurrence as selection sort: \( O(n^2) \)

- Average-case (e.g., with random pivot)
  - \( O(n \log n) \), not responsible for proof (in text)

---

**Cutoffs**

- For small \( n \), all that recursion tends to cost more than doing a quadratic sort
  - Remember asymptotic complexity is for large \( n \)

- Common engineering technique: switch algorithm below a cutoff
  - Reasonable rule of thumb: use insertion sort for \( n < 10 \)

- Notes:
  - Could also use a cutoff for merge sort
  - Cutoffs are also the norm with parallel algorithms
  - Switch to sequential algorithm
  - None of this affects asymptotic complexity
Cutoff pseudocode

```c
void quicksort(int[] arr, int lo, int hi) {
    if (hi - lo < CUTOFF)
        insertionSort(arr, lo, hi);
    else
        ...
}
```

Notice how this cuts out the vast majority of the recursive calls
- Think of the recursive calls to quicksort as a tree
- Trims out the bottom layers of the tree

How Fast Can We Sort?

- Heapsort & mergesort have $O(n \log n)$ worst-case running time
- Quicksort has $O(n \log n)$ average-case running time
- These bounds are all tight, actually $\Theta(n \log n)$
- Comparison sorting in general is $\Omega(n \log n)$
  - An amazing computer-science result: proves all the clever programming in the world cannot comparison-sort in linear time

The Big Picture

Surprising amount of juicy computer science: 2-3 lectures...

Simple algorithms: $O(n^2)$
- Insertion sort
- Selection sort
- Shell sort
Fancier algorithms: $O(n \log n)$
- Heap sort
- Merge sort
- Quick sort (avg)
Comparison lower bound: $\Omega(n \log n)$
Specialized algorithms: $O(n)$
- Bucket sort
- Radix sort
Handling huge data sets

Bucket Sort (a.k.a. BinSort)

- If all values to be sorted are known to be integers between 1 and $K$ (or any small range):
  - Create an array of size $K$
  - Put each element in its proper bucket (a.k.a. bin)
  - If data is only integers, no need to store more than a count of how times that bucket has been used
- Output result via linear pass through array of buckets

Bucket Sort (a.k.a. BinSort)

```
example:
K=5
input (5,1,3,4,3,2,1,5,4,5)
output: 1,1,2,3,3,4,4,5,5,5
```

Visualization


Analyzing Bucket Sort

- Overall: $O(n+K)$
  - Linear in $n$, but also linear in $K$
  - $O(n \log n)$ lower bound does not apply because this is not a comparison sort
- Good when $K$ is smaller (or not much larger) than $n$
  - We don’t spend time doing comparisons of duplicates
- Bad when $K$ is much larger than $n$
  - Wasted space; wasted time during linear $O(K)$ pass
- For data in addition to integer keys, use list at each bucket
Bucket Sort with Data

- Most real lists aren't just keys; we have data
- Each bucket is a list (say, linked list)
- To add to a bucket, insert in $O(1)$ (at beginning, or keep pointer to last element)

  • Example: Movie ratings;
  • scale 1-5: 1=bad, 5=excellent
  • Input:
    5: Casablanca
    3: Harry Potter movies
    5: Star Wars Original Trilogy
    1: Rocky V

  • Result: 1: Rocky V, 3: Harry Potter, 5: Casablanca, 5: Star Wars

  • Easy to keep 'stable'; Casablanca still before Star Wars

Radix sort

- Radix = "the base of a number system"
  - Examples will use 10 because we are used to that
  - In implementations use larger numbers
  - For example, for ASCII strings, might use 128

  • Idea:
    - Bucket sort on one digit at a time
    - Number of buckets = radix
    - Starting with least significant digit
    - Keeping sort stable
    - Do one pass per digit
    - Invariant: After $k$ passes (digits), the last $k$ digits are sorted

  • Aside: Origins go back to the 1890 U.S. census

Example

Radix = 10

Input: 478
537
9
721
3
38
143
67

Order now: 721
3
537
478
9
38
143
67

First pass: bucket sort by ones digit

Radix = 10

Example

Input:

0 1 2 3 4 5 6 7 8 9

Order was:

0 1 2 3 4 5 6 7 8 9

Second pass: bucket sort by tens digit

Order: 721
9
537
38
143
67
478
9
3

Visualization

Analysis

Input size: \( n \)
Number of buckets = Radix: \( B \)
Number of passes = "Digits": \( P \)
Work per pass is 1 bucket sort: \( O(B + n) \)
Total work is \( O(P(B + n)) \)

Compared to comparison sorts, sometimes a win, but often not

- Example: Strings of English letters up to length 15
  - Run-time proportional to: \( 15^2(52 + n) \)
  - This is less than \( n \log n \) only if \( n > 33,000 \)
  - Of course, cross-over point depends on constant factors of the implementations
    - And radix sort can have poor locality properties

Sorting massive data

- Need sorting algorithms that minimize disk/tape access time:
  - Quicksort and Heapsort both jump all over the array, leading to expensive random disk accesses
  - Merge sort scans linearly through arrays, leading to (relatively) efficient sequential disk access
- Merge sort is the basis of massive sorting
- Merge sort can leverage multiple disks

External Merge Sort

- Sort 900 MB using 100 MB RAM
  - Read 100 MB of data into memory
  - Sort using conventional method (e.g. quicksort)
  - Write sorted 100MB to temp file
  - Repeat until all data in sorted chunks (900/100 = 9 total)
- Read first 10 MB of each sorted chunk, merge into remaining 10MB
  - writing and reading as necessary
  - Single merge pass instead of \( \log n \)
  - Additional pass helpful if data much larger than memory
- Parallelism and better hardware can improve performance
- Distribution sorts (similar to bucket sort) are also used

Last Slide on Sorting

- Simple \( O(n^2) \) sorts can be fastest for small \( n \)
  - Selection sort, Insertion sort (latter linear for mostly-sorted)
  - Good for "below a cut-off" to help divide-and-conquer sorts
- \( O(n \log n) \) sorts
  - Heap sort, in-place but not stable nor parallelizable
  - Merge sort, not in place but stable and works as external sort
  - Quick sort, in place but not stable and \( O(n^2) \) in worst-case
    - Often fastest, but depends on costs of comparisons/copies
  - \( \Omega(n \log n) \) is worst-case and average lower-bound for sorting by comparisons
- Non-comparison sorts
  - Bucket sort good for small number of possible key values
  - Radix sort uses fewer buckets and more phases
- Best way to sort? It depends!