CSE 202—Day 4

Divide and Conquer—Sorting

Neil Rhodes

UC San Diego
Overview

- Divide and Conquer
- Mergesort
- Quicksort
- Minimal cost comparison sorting
- Randomized quicksort
Divide and Conquer

Divide: the problem into one or more smaller problems

Conquer: Solve the smaller problem(s)

Combine: the solutions to the smaller problem(s)

The runtime is a recurrence of the form:

$$T(n) = a_1 T(n/b_1) + \ldots + a_k T(n/b_k) + D(n) + C(n)$$
Divide and Conquer

Example: Binary search

**Divide** the problem into one half the size by comparing the key against the middle element

**Conquer:** Binary search the half-size problem

**Combine** trivially by returning the answer to the half-size problem

For this recurrence, there’s one subproblem,

\[ D(n) = O(1), C(n) = O(1) \]

\[ T(n) = T(n/2) + O(1) + O(1)T(n) = O(\log n) \]
Mergesort

- Idea
- Code
- Runtime analysis
- Disadvantages
- Speed-up
Idea of Mergesort

**Divide**  the array into two equal-size sub-arrays

**Conquer:**  Sort the subarrays

**Combine:**  Merge the sorted subarrays
Code

MERGESORT(A, l, r))

1  if l < r
2  then
3         m ← ⌊(l + r)/2⌋
4         MERGESORT(A, l, m)
5         MERGESORT(A, m+1, r);
6         MERGE(A, l, m, r)
Merge

• Merge has two adjacent ranges of the array which are each sorted, and which it must merge.
• Copy the two ranges to temporary arrays, and copy in sorted order back to original array

\textsc{Merge}(A,l,m,r)

1\hspace{1em}A_1 \leftarrow A[l..m] \triangleright \text{Create two external arrays}
2\hspace{1em}A_2 \leftarrow A[m+1..r]
3\hspace{1em}next_1 \leftarrow next_2 \leftarrow 1
4\hspace{1em}\textbf{while} A_1, A_2 \text{ are not empty}
5\hspace{2em}\textbf{do}
6\hspace{3em}A[next] = \text{min}(A_1[next_1], A_2[next_2])
7\hspace{3em}next \leftarrow next + 1
8\hspace{3em}\text{Increment next}_1 \text{ or next}_2 \text{ appropriately}
9\hspace{1em}\text{Copy remainder of } A_1 \text{ and } A_2 \text{ to the end of } A
Merge

Merge can be simplified using *sentinels* (special values that remove boundary conditions)

\[
\text{MERGE}(A, l, m, r))
\]

1. \( A_1 \leftarrow A[l..m] \triangleright\) Create two external arrays
2. \( A_2 \leftarrow A[r..m] \)
3. Append \( \infty \) to end of \( A_1 \) and \( A_2 \)
4. \( next_1 \leftarrow next_2 \leftarrow 1 \)
5. \( \text{for } i \leftarrow l \text{ to } r \)
6. \hspace{1em} do
7. \hspace{2em} \( A[next] = \min(A_1[next_1], A_2[next_2]) \)
8. \hspace{2em} \( next \leftarrow next + 1 \)
9. \hspace{2em} Increment \( next_1 \) or \( next_2 \) appropriately
Merge

Takes $O(r - l)$ time.

- Copies $r - l + 1$ elements into two arrays
- Loop executes $r - l$ times
Mergesort runtime analysis

\[ D(n) = O(1) \]
\[ C(n) = O(n) \]
\[ T(n) = T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + C(n) + D(n) \]
\[ T(n) = 2T(n/2) + O(n) \]
\[ T(n) = O(n \log n) \]
Mergesort [dis]advantages

• Sorting is not \textit{in-place} (more than a constant number of array entries are copied to external storage)

• Sorting is \textit{stable} (items that compare equally keep their relative positioning)
Mergesort speedup

• Key idea: when there aren’t many elements left to sort, insertion sort is faster than merge sort (constants are much smaller)

• Clearly works when problem size is less than a constant. Can it work when problem size is bigger than that? (See CLRS, Problem 2-1)
Quicksort

- Idea
- Code
- Runtime analysis
- Disadvantages
- Speed-up
Idea of Quicksort

**Divide** Choose a splitter value (an element from the array). Partition the array into a left side, with values ≤ than the splitter value, and a right side, with values ≥ the splitter value. Put the splitter value in the middle.

**Conquer:** Sort the left side and the right side

**Combine:** return sorted left-side concatenated with splitter value concatenated with sorted right side (if Divide puts splitter in right spot to begin with, no work needed).

Mergesort spends O(1) time dividing and O(n) time combining. Quicksort spends O(n) time dividing and O(1) time combining.
Code

QUICKSORT(S)
1 Choose a splitter $a_i \in S$
2 for each element $a_j$ of $S$
3 do
4 \hspace{1em} if $a_j < a_i$
5 \hspace{2em} then
6 \hspace{3em} put $a_j$ in $S^-$
7 \hspace{2em} else
8 \hspace{3em} put $a_j$ in $S^+$
9 Quicksort($S^-$)
10 Quicksort($S^+$)
11 Output sorted $S^-$, $a_j$, sorted $S^+$
Quicksort Runtime Analysis

\[ D(n) = O(n) \]
\[ C(n) = O(1) \]

Worst-case: split into unequal-as-possible lists

\[ T(n) = T(1) + T(n-1) + C(n) + D(n) \]
\[ T(n) = T(n-1) + O(n) \]
\[ T(n) = O(n^2) \]
Quicksort Runtime Analysis

\[ D(n) = O(n) \]
\[ C(n) = O(1) \]

Best-case: split into as equal-as-possible lists

\[ T(n) = 2T(n/2) + C(n) + D(n) \]
\[ T(n) = O(n \log n) \]
Quicksort Runtime Analysis

\[ D(n) = O(n) \]
\[ C(n) = O(1) \]

OK-case: split a constant fraction, \( k \), of the elements

\[ T(n) = T\left(n\frac{1}{k}\right) + T\left(n\frac{k-1}{k}\right) + C(n) + D(n) \]
\[ T(n) = O(n \log_k n) = O(n \log n) \]
Quicksort Runtime Analysis

Average-case: we’ll see shortly
Quicksort Splitter

How to choose splitter:

- First item in array
- Last item in array
- Median item in array
- Best of 3
- Random item (yields a randomized algorithm)
Quicksort [dis]advantages

- Sorting is *in-place*
- Runs well in real world
- Sorting is *unstable* (items that compare equally can lose their relative positioning)
Quicksort [dis]advantages

- Worst-case is bad
  - In fact, McIlroy, *A Killer Adversary for Quicksort* shows that if sorting routine does "callback" to adversary for comparison, adversary can force $\Theta(n^2)$ time regardless of how splitter is chosen.

- Pseudo-random splitters can fail. Karloff & Raghavan show:
  for any standard linear congruential pseudo-random number generator (PRNG) (e.g. Unix’s RAND), there is a (carefully constructed) bad sorting instance that, averaged over all PRNG seeds has expected time $O(n^2)$
Comparison sorting bound

Decision tree

```
1:2

2:3

< 1,2,3 > 1:3 < 2,1,3 > 2:3

< 1,3,2 > < 3,1,2 > < 2,3,1 > < 3,2,1 >
```
Comparison sorting bound

Decision tree is a binary tree where:

- Internal nodes marked $i, j$ represent comparisons between $A_i$ and $A_j$ of the input list.
- Leaves marked with permutations of $1..n : \pi(1), \pi(2) \ldots \pi(n)$.
- A path from root to leaf shows which elements are compared. Leaf shows final ordering is: $A_{\pi(1)} \leq A_{\pi(2)} \leq \ldots \leq A_{\pi(n)}$. 

Comparison sorting bound

- Decision tree must contain all possible permutations as leaves.
- Thus, a comparison-based decision tree must have $\geq n!$ leaves.
- A tree of height $h$ can have at most $2^h$ leaves.
- A tree with $> 2^h$ leaves has a height of at least $\log h$.
- A decision tree with all possible permutations has height $\geq \log(n!) = \Omega(n \log n)$.
Randomized Quicksort

- When partitioning, choose a random element to partition on.
- Analysis to come...
Probability Review

- A sample space $S$ is a set of elementary events
  - Rolls of a die $= \{1, 2, 3, 4, 5, 6\}$ or flipping two coins $= \{HH, HT, TH, TT\}$
- A event is a subset of $S$.
  - Rolling better than a 4 $= \{5, 6\}$
  - Flipping at least one head $= \{HH, HT, TH\}$
- A probability distribution $Pr$ on a sample space is a function mapping events to real numbers $\in [0..1]$ such that:
  - $Pr\{S\} = 1$
  - $Pr\{A \cup B\} = Pr\{A\} + Pr\{B\}$ for mutually exclusive events
Probability Review

- If $S$ is finite or countably infinite, then the probability distribution is *discrete*.
- If $S$ is finite and each elementary event $e \in S$ has $Pr\{e\} = 1/|S|$, then we have a *uniform probability distribution*.
  - Probability distribution for rolling dice $\{1, 2, 3, 4, 5, 6\}$ is uniform
  - Probability distribution for scores on the GRE $\{400, 410, \ldots, 1590, 1600\}$ is not uniform.
A (discrete) random variable $X$ is a function from the elementary events in a (finite or countably infinite) sample space $S$ to the real numbers. Examples:

- Flip $k$ coins (sample space has $2^k$ elementary events). $X$ is the total number of heads flipped ($0 \leq X(e) \leq k$)
- Roll two fair dice. $S = \{(1,1), (1,2), \ldots, (6,6)\}$. $Y$ is the total shown on the dice
Probability Review

- The event $X = x$ is defined to be $\{s \in S : X(s) = x\}$
- $X + Y$ is a function. For example, 
  $(X + Y)(e) = X(e) + Y(e)$
- An *indicator random variable* $I(A)$ associated with event $A$ is 1 if $A$ occurs and 0 otherwise.
Probability Review

- The *Expectation* $E[X]$ is $\sum_{e \in S} X(e)Pr\{e\}$. That is, the weighted average of $X$.
  - Throw a die. $Y$ is the total shown on the die
    $$E[Y] = 1 \cdot Pr\{1\} + 2 \cdot Pr\{2\} + \ldots + 6 \cdot Pr\{6\} = 3.5$$

- *Linearity of Expectation* theorem: $E[X+Y] = E[X] + E[Y]$ (regardless of whether $X$ and $Y$ are dependent or independent)
  - Throw two dice. $Y + Y$ is the total shown.

- The expected value of an indicator random variable is the probability the associated event occurs.