CSE 326 Data Structures

Dave Bacon

Sorting

Logistics

- Survey on main web page!

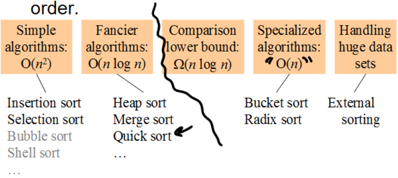
- Homework 6 (due on Friday)

- Project 3, Project 3, Project 3.

- Reading: finish Weiss Chapter 7, start Chapter 9

Sorting: *The* ***Big*** *Picture*

Given *n* <u>comparable</u> elements in an array, sort them in an increasing (or decreasing) order.



Simple algorithms: $O(n^2)$

- Insertion sort
- Selection sort
- Bubble sort
- Shell sort
- …

Fancier algorithms: $O(n \log n)$

- Heap sort
- Merge sort
- Quick sort
- …

Comparison lower bound: $\Omega(n \log n)$

Specialized algorithms: “$O(n)$”

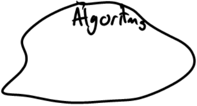
- Bucket sort
- Radix sort

Handling huge data sets

- External sorting

How fast can we sort?

- Heapsort, Mergesort, and Quicksort all run in O(N log N) best case running time
- Can we do any better?
- No, if the basic action is a comparison.



Sorting Model

- Recall our basic assumption: we can only compare two elements at a time
  - we can only reduce the possible solution space by half each time we make a comparison
- Suppose you are given N elements
  - Assume no duplicates a, b, c, d, e, . . .
- How many possible orderings can you get?
  - Example: a, b, c (N = 3)

Permutations

- How many possible orderings can you get?
  - Example: a, b, c (N = 3)
  - (a b c), (a c b), (b a c), (b c a), (c a b), (c b a)
  - 6 orderings = $3 \cdot 2 \cdot 1 = 3!$ (ie, "3 factorial")
  - All the possible permutations of a set of 3 elements
- For N elements
  - N choices for the first position, (N-1) choices for the second position, …, (2) choices, 1 choice
  - $N(N-1)(N-2)\cdots(2)(1)=$ N! possible orderings

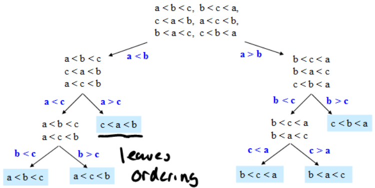
$N(N-1)(N-2) --- = N!$

N! grows fast.

Binary tree node—set of possible orderings

Decision Tree

edge=1-comparison



a < b < c, b < c < a,
c < a < b, a < c < b,
b < a < c, c < b < a

a < b

a > b

a < b < c
c < a < b
a < c < b

b < c < a
b < a < c
c < b < a

a < c

a > c

b < c

b > c

a < b < c
a < c < b

c < a < b

b < c < a
b < a < c

c < b < a

b < c

b > c

c < a

c > a

a < b < c

a < c < b

b < c < a

b < a < c

leaves ordering

The leaves contain all the possible orderings of a, b, c

Student Activity

Lower bound on Height

- A binary tree of height h has **at most** *how many* leaves?

$\boxed{L \le 2^h}$

$h=0 \quad L \le 1$

h

$L \le 2^{h-1}$

$L \le 2^{h-1}$

- A binary tree with L leaves has height **at least**:

$\log_2 L \le \log_2 2^h = h$ $\boxed{h \ge \log_2 L}$

- The decision tree has how many leaves:

$\boxed{N!}$

- So the decision tree has height:

$h \ge \log_2(N!) \leftarrow$ worst case

$\log(N!)$ is $\Omega(N\log N)$

$\log AB = \log A + \log B$

$$
\begin{aligned}
\log(N!) &= \log\bigl(N\cdot(N-1)\cdot(N-2)\cdots(2)\cdot(1)\bigr)\\
&= \log N + \log(N-1) + \log(N-2) + \cdots + \overset{\geq 0}{\underline{\log 2} + \underline{\log 1}}\\
&\geq \log N + \log(N-1) + \log(N-2) + \cdots + \log\frac{N}{2}\\
&\geq \frac{N}{2}\log\frac{N}{2}\\
&\geq \frac{N}{2}(\log N - \log 2) = \frac{N}{2}\log N - \frac{N}{2}\\
&\boxed{= \Omega(N\log N)}
\end{aligned}
$$

select just the first N/2 terms

each of the selected terms is ≥ logN/2

Ω(N log N)

- Run time of any comparison-based sorting algorithm is **Ω(N log N)**
- Can we do better if we don't use comparisons?

comparisons.

$a<b$

BucketSort (aka BinSort)

If all values to be sorted are *known* to be between 1 and *K*, create an array `count` of size *K*, **increment** counts while traversing the input, and finally output the result.

Example *K*=5. Input = (5,1,3,4,3,2,1,1,5,4,5)

↑↑↑↑↑↑↑↑↑↑↑

`count` array	
1	3 * ~~2~~
2	1
3	*2
4	*2
5	3 * ~~2~~

→ 1, 1, 1, 2, 3, 3, 4, 4, 5, 5, 5

initial scan
n

read → K

Running time to sort n items? no graph

$O(n+K)$

BucketSort Complexity: O(*n*+*K*)

k=1000   O(n+1000)

- Case 1: *K* is a constant
  - BinSort is linear time
- Case 2: *K* is variable ← K → f(n)
  - Not simply linear time
- Case 3: *K* is constant but large (e.g. 2^{32})
  - ???

↑

32 bit number

$O(n+2^{32})$

Fixing impracticality: RadixSort

- Radix = "The base of a number system"
  - We'll use 10 for convenience, but could be anything

- Idea: BucketSort on each **digit**,
  least significant to most significant
  (lsd to msd)

Radix Sort Example (1^{st} pass)

Input data

478
537
9
721
3
38
123
67

Bucket sort
by 1's digit

0	1	2	3	4	5	6	7	8	9
	721		3 123				537 67	478 38	9

After 1^{st} pass

721
3
123
537
67
478
38
9

This example uses B=10 and base 10 digits for simplicity of demonstration. Larger bucket counts should be used in an actual implementation.

Radix Sort Example (2nd pass)

After 1st pass

721
3
123
537
67
478
38
9

Bucket sort
by 10's
digit

0	1	2	3	4	5	6	7	8	9
03 09		721 123	537 38			67	478		

After 2nd pass

3
9
721
123
537
38
67
478

Radix Sort Example (3rd pass)

After 2nd pass

3
9
721
123
537
38
67
478

Bucket sort
by 100's
digit

0	1	2	3	4	5	6	7	8	9
003 009 038 067	123			478	537		721		

After 3rd pass

3
9
38
67
123
478
537
721

Invariant: after k passes the low order k digits are sorted.

Student Activity

RadixSort

- Input:126, 328, 636, 341, 416, 131, 328

BucketSort on lsd:

0	1	2	3	4	5	6	7	8	9

BucketSort on next-higher digit:

0	1	2	3	4	5	6	7	8	9

BucketSort on msd:

0	1	2	3	4	5	6	7	8	9

Radixsort: Complexity

- How many passes?

  $p = \log_K (\text{max number})$

- How much work per pass?

  $O(n + K)$

- Total time?

  $O(p(n+K))$

- Conclusion?

  p is large

  $\log_2 2^{32} = 32$

- In practice
  - RadixSort only good for large number of elements with relatively small values
  - Hard on the cache compared to MergeSort/QuickSort <

Internal versus External Sorting

- Need sorting algorithms that minimize disk/tape access time
- **External sorting** – Basic Idea:
  - Load chunk of data into RAM, sort, store this “run” on disk/tape
  - Use the Merge routine from Mergesort to merge runs
  - Repeat until you have only one run (one sorted chunk)
  - Text gives some examples

Graphs

Chapter 9 in Weiss

Graph… ADT?

Jar Jar
Darth Vade

- Not quite an ADT… operations not clear
- A formalism for representing relationships between objects

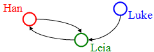
  Graph `G = (V,E)`

  - *Set of vertices*:

    $\mathbf{V} = \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$
  - *Set of edges*:

    $\mathbf{E} = \{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_m\}$

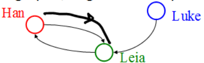
    where each $\mathbf{e}_i$ connects two vertices $(\mathbf{v}_{i1}, \mathbf{v}_{i2})$



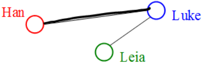
```
V = {Han, Leia, Luke}
E = {(Luke, Leia),
     (Han, Leia),
     (Leia, Han)}
```

Graph Definitions

In *directed* graphs, edges have a specific direction:



In *undirected* graphs, they don't (edges are two-way):



v is *adjacent* to **u** if **(u,v)** ∈ **E**

More Definitions: Simple Paths and Cycles

A *simple path* repeats no vertices (except that the first can be the last):

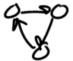
- p = {Seattle, Salt Lake City, San Francisco, Dallas}
- p = {Seattle, Salt Lake City, Dallas, San Francisco, Seattle}

A *cycle* is a path that starts and ends at the same node:

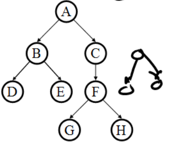
- p = {Seattle, Salt Lake City, Dallas, San Francisco, Seattle}
- p = {Seattle, Salt Lake City, Seattle, San Francisco, Seattle}

A *simple cycle* is a cycle that repeats no vertices except that the first vertex is also the last (in undirected graphs, no edge can be repeated)

Trees as Graphs



- Every tree is a graph!
- Not all graphs are trees!



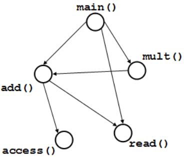
A graph is a tree if

- There are *no cycles* (directed or undirected)
- There is a *path* from the root *to every node*

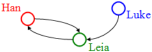
Directed Acyclic Graphs (DAGs)

DAGs are directed graphs with no (directed) cycles.

Aside: *If program call-graph is a DAG, then all procedure calls can be in-lined*



Graph Representations



0. List of vertices + list of edges
1. 2-D matrix of vertices (marking edges in the cells)
   "adjacency matrix" ←
2. List of vertices each with a list of adjacent vertices
   "adjacency list" ←

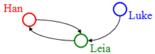
Vertices and edges may be labeled

Things we might want to do:

- iterate over vertices
- iterate over edges
- iterate over vertices adj. to a vertex
- check whether an edge exists

Representation 1: Adjacency Matrix

A `|V| x |V|` array in which an element `(u,v)` is true if and only if there is an edge from `u` to `v`



	Han	Luke	Leia
Han			
Luke			
Leia			

space requirements:

runtime: