

Flipped Lecture Concepts

Week 4

Ananda Guna

10.02.14

Focus this week

- Quicksort
 - 2-way and 3-way partitioning
 - Bad cases, average cases
 - Dealing with equal elements
 - Proof of average case
- Priority Queues
 - API
 - Implementation using binary heaps
 - Applications

Quick sort idea

- Good general purpose sort
- Works great for randomized data sets
- In-place but not stable (unless you use extra memory)
- Bad cases
 - All equal keys
 - Non-randomized data sets

Partitioning

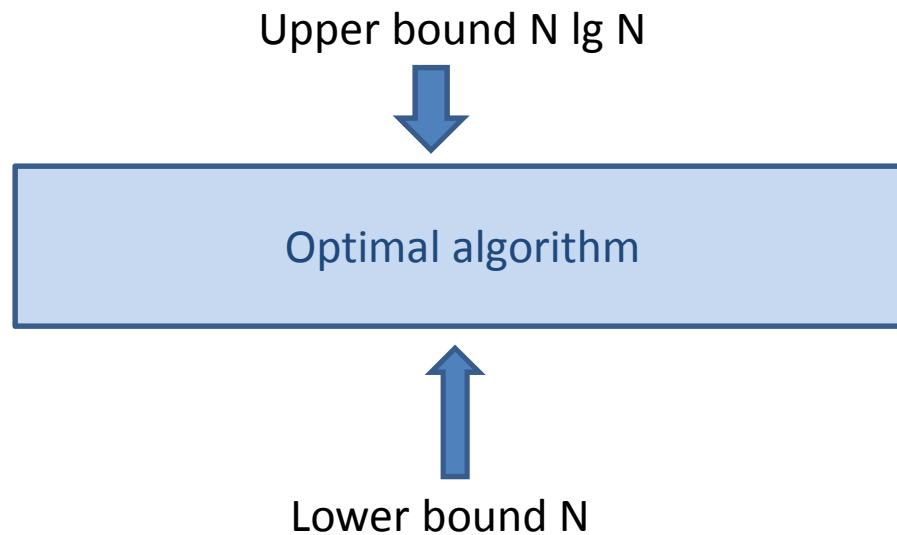
- 2-way partitioning
- 3-way partitioning (Dijkstra's)

Choosing the pivot

- First element in a randomized set
- Median of the three
- Double pivots

Idea of quick-select

- Given N items, find the k -th largest element



Quick-select is $\sim N$

Duplicate keys

Exercise: Show that standard 2-way partitioning algorithm behaves badly when it does not stop at equal keys

Dijkstra 3-way partitioning demo

- Let v be partitioning item $a[l_0]$.
- Scan i from left to right.
 - $(a[i] < v)$: exchange $a[l_t]$ with $a[i]$; increment both l_t and i
 - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$; decrement gt
 - $(a[i] == v)$: increment i

Just curious!

- Is there an efficient algorithm (something less than $N \lg N$) to sort N items where only k distinct keys exists?
 - If $k=N$ then we know the answer is NO!
 - What if there is some magic k where this can be true?

Best sorting algorithm to use?

Applications have diverse attributes.

- Stable?
- Parallel?
- Deterministic?
- Keys all distinct?
- Multiple key types?
- Linked list or arrays?
- Large or small items?
- Is your array randomly ordered?
- Need guaranteed performance?

Priority Queues

```
public class MaxPQ<Key extends Comparable<Key>>
```

```
    MaxPQ()
```

create an empty priority queue

```
    MaxPQ(Key[] a)
```

create a priority queue with given keys

```
    void insert(Key v)
```

insert a key into the priority queue

```
    Key delMax()
```

return and remove the largest key

```
    boolean isEmpty()
```

is the priority queue empty?

```
    Key max()
```

return the largest key

```
    int size()
```

number of entries in the priority queue

implementation

implementation	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1
goal	$\log N$	$\log N$	$\log N$

Binary Heaps

- Maintains the invariants
 - Shape invariant
 - Order invariant
- Implemented as a complete binary tree
 - Perfectly balanced except the last level
- Max is the root element
- Compact representation using an array
- Array[0] is unused
 - A[i] is parent, A[2i] and A[2i+1] are left and right children
- Height h of the binary heap with N elements
 - $2^h < N \leq 2^{h+1} - 1$

Operations

- deleteMax
 - Replace last element with root and sink
- Insert
 - Insert new as last element and swim up
- Heap shape Invariant violations (temporary)

Challenge

- Build a data structure that can do the following operations
 - find median in constant time
 - delete median in $\lg N$ time
 - insert an element in $\lg N$ time

Challenge problem

- Checkout line problem
 - 5 customers each requiring different times of service
 - Times required in original order (shown in mins)

1	2	3	4	5
18	5	25	2	10

- What is the average time to serve a customer?

Challenge problem ctd..

- Suppose the customers are placed in a minPQ
- What is the average time to serve a customer?

1	2	3	4	5
18	5	25	2	10

Heap Sort

- Two step process
 - Build a Max Heap
 - while (!heap.empty()) delete Max
- Complexity?

Sort summary

	inplace?	stable?	best	average	worst	remarks
selection	✓		$\frac{1}{2} N^2$	$\frac{1}{2} N^2$	$\frac{1}{2} N^2$	N exchanges
insertion	✓	✓	N	$\frac{1}{4} N^2$	$\frac{1}{2} N^2$	use for small N or partially ordered
shell	✓		$N \log_3 N$?	$c N^{3/2}$	tight code; subquadratic
merge		✓	$\frac{1}{2} N \lg N$	$N \lg N$	$N \lg N$	$N \log N$ guarantee; stable
timsort		✓	N	$N \lg N$	$N \lg N$	improves mergesort when preexisting order
quick	✓		$N \lg N$	$2 N \ln N$	$\frac{1}{2} N^2$	$N \log N$ probabilistic guarantee; fastest in practice
3-way quick	✓		N	$2 N \ln N$	$\frac{1}{2} N^2$	improves quicksort when duplicate keys
heap	✓		N	$2 N \lg N$	$2 N \lg N$	$N \log N$ guarantee; in-place