**COS 226**
**Midterm Review**
**Spring 2015**

Ananda Gunawardena  
(guna)  
guna@cs.princeton.edu  
guna@princeton.edu

---

**Time and location:**
- The midterm is during lecture on — Wednesday, March 11 from 11:12-12:20pm.
- The exam will start and end promptly, so please do arrive on time.
- The midterm room is either McCosh 10 or McDonnell A02, depending on your precept date.
  - Friday Precepts: McCosh 10.
  - Thursday Precepts: McDonnell A02.
- Failure to go to the right room can result in a serious deduction on the exam. There will be no makeup exams except under extraordinary circumstances, which must be accompanied by the recommendation of a Dean.

---

**Rules**
- Closed book, closed note.
- You may bring one 8.5-by-11 sheet (one side) with notes in your own handwriting to the exam.
- No electronic devices (including calculators, laptops, and cell phones).

---

**Materials covered**
- Lectures 1–10.
- Programming assignments 1–4.

---

**concepts (so far) in a nutshell**

- Intro: Union Find
- Analysis of Algorithms
- Stacks and Queues
- Elementary Sorts
- Quick sort
- Priority Queues
- Elementary Symbol Tables, BSTs
- Balanced Search Trees
- Hash Tables

---

**List of algorithms and data structures:**
- quick-find
- quick-union
- union-by-size
- binary search
- disjoint-set
- divide 2-stack
- selection 2-stack
- heap
- heapify
- heapify
- heap sort
- heap sort
- merge sort
- binary search
- sequential search
- 2-3 trees
- light bunking red-black BSTs
- separate chaining linear probing

- Recall as much as possible about each of the above topics
- Write down up to 5 most important things about each one
Analysis of Algorithms

- Estimate the performance of an algorithm using order of growth:
  - comparisons, array accesses, exchanges, memory requirements

  \[ \Theta(n^2), \Omega(n^2), O(n^2) \]

  - Best, worst, average

  - Performance measure based on some specific inputs

  \[ \Theta(n^2) \sim O(n^2) \]

Amortized analysis

- Measure of average performance over a series of operations
  - Some good, few bad

  \[ \sum_{i=1}^{n} \frac{1}{2} \leq \sum_{i=1}^{n} \frac{1}{i} \leq \int_{1}^{n+1} \frac{1}{x} dx \]

More formally....

- **Tilda notation**

<table>
<thead>
<tr>
<th>notation</th>
<th>( f(n) \sim g(n) ) means</th>
<th>( \frac{f(n)}{g(n)} \sim 1 ) as ( n \to \infty )</th>
</tr>
</thead>
</table>

Big Theta \( \Theta \)

- asymptotic order of growth
  - \( \Theta(n^2) \)
  - \( \Theta(n \log n) \)
  - \( \Theta(n) \)
  - \( \Theta(1) \)

Big O \( O \)

- \( O(n^2) \)
  - \( O(n \log n) \)
  - \( O(n) \)
  - \( O(1) \)

Big Omega \( \Omega \)

- \( \Omega(n^2) \)
  - \( \Omega(n \log n) \)
  - \( \Omega(n) \)
  - \( \Omega(1) \)

- Techniques:
  - count operations (reads, writes, compares)
  - derive mathematically
    - exploit the property of the algorithm
    - solve a recurrence formula to reach a closed form solution.
    - obtain upper/lower bounds

Analysis of Algorithms
Count operations

Example 1

```
for (int i = 1; i < N; i++) {
    for (int j = i; j > 0; j--)
        if (genomes[i].length() > genomes[j].length())
            m[i][j] = m[i][j - 1];
        else
            m[i][j] = m[i - 1][j] + 1;
    a[i][j] = m[i][j];
}
```

<table>
<thead>
<tr>
<th>compares</th>
<th>Array accesses</th>
<th>assignments</th>
<th>External method calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{1}{2}n^2$</td>
<td>$6 \cdot \frac{1}{2}n^2$</td>
<td>$3 \cdot \frac{3}{2}n^2$</td>
<td>$1 \cdot \frac{1}{2}n^2$</td>
</tr>
</tbody>
</table>

$N[i][j] = m[i][j] + a[i][j]$ = Length

Count operations

Example 2

```
for (int i = 0; i < N; i++)
    if (isSubstring(genomes[i], genomeFile))
```

<table>
<thead>
<tr>
<th>compares</th>
<th>Array accesses</th>
<th>assignments</th>
<th>External method calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

Runtime complexity

This is a method of describing behavior of an algorithm using runtime
Observations. Runtime of an algorithm depends on many factors
including language, compiler, input size, memory, optimizations etc.

```
int N = Integer.parseInt(args[0]);
String[] genomes = new String[N];
for (int i = 0; i < N; i++)
    if i = new In(genomeFile);
    genomes[i] = new String(IntStream.range(0, genomeLength)
        .mapToObj(i -> genome[i])
        .collect(Collectors.joining())
    );
}
```

The following runtimes were observed from an algorithm that reads a file of strings and
splits them using insertion sort. The runtime analysis seems to suggest the algorithm is
linear. Is this correct?

$T(N) = \theta(1)$

useful formulas

$$1 + 2 + \ldots + N = \sum_{i=1}^{N} i = \frac{N(N+1)}{2} = \frac{1}{2} N^2$$

$$1^2 + 2^2 + \ldots + N^2 = \sum_{i=1}^{N} x^2 = \frac{N(N+1)(2N+1)}{6}$$

```
1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{N} = \sum_{i=1}^{N} \frac{1}{i} 
```

Important when counting

- Do not assume two nested loops always give you $n^2$
  - Always read the code to see what it does

- When doubling or halving loop control variable, it can lead to log N performance
  - But analyze carefully

- Sometimes the sum of operations can be approximated by an integral

  $\sum f(n) \approx \int f(n) \, dx$

Mathematically speaking

- write recurrences for many of the standard algorithms
  - linear search $T(n) = 1 + T(n-1)$
  - binary search $T(n) = 1 + \log_2 n$
  - merge sort $T(n) = T(n/2) + n$
  - quicksort $T(n) = T(n-1) + T(1) + n$
  - insertion sort $T(n) = 1 + T(n-1)$

- solve them using many of the techniques discussed
  - Recurrence of the form $T(n) = n + T(n-1) = 1 + (1 + T(n-2)) = \ldots$
counting memory

- standard data types (int, bool, double)
- object overhead – 16 bytes
- array overhead – 24 bytes
- references – 8 bytes
- inner class reference – 8 bytes

\[ 3N + 8 + 16 \sim 8N \]

Stacks and Queues

- Amortized constant time operations
- implementation using
  - linked lists
  - resizable arrays
- many variations of stacks and queues asked in design questions
  - design a queue that allows removing a random element (in addition to dequeue)
  - design a queue using a resizable array
  - design a queue using two stacks

Resizing arrays

- Arrays are static, simple, random access data structures
- Arrays can be used in many applications
  - If resizing can be done efficiently
  - resizing by 1 is a bad idea (why?)
  - doubling the array is a good idea (why?)
  - can we get amortized constant performance in arbitrary insertion into an array?

Using resizable arrays

- Implement a stack
  - amortized constant time: pop and push
- Implement a queue with circular array
  - amortized constant time: enqueue and dequeue

Data Structure Performance estimates (worst or amortized)

<table>
<thead>
<tr>
<th></th>
<th>Find</th>
<th>Insert</th>
<th>Delete</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>unorderedMap</td>
<td>( n )</td>
<td>( n )</td>
<td>( n )</td>
<td>( n )</td>
</tr>
<tr>
<td>orderedMap</td>
<td>( \log n )</td>
<td>( n )</td>
<td>( n )</td>
<td>( \log n )</td>
</tr>
<tr>
<td>linkedList</td>
<td>( n )</td>
<td>( n )</td>
<td>( n )</td>
<td>( n )</td>
</tr>
<tr>
<td>unidirectionalList</td>
<td>( 1 )</td>
<td>( n )</td>
<td>( n )</td>
<td>( n )</td>
</tr>
<tr>
<td>Stack</td>
<td>( n )</td>
<td>( n )</td>
<td>( n )</td>
<td>( n )</td>
</tr>
<tr>
<td>Deque</td>
<td>( n )</td>
<td>( n )</td>
<td>( n )</td>
<td>( n )</td>
</tr>
</tbody>
</table>
Resizable array questions

- Resizing array by one gives amortized linear time per item (bad)
- Resizing array by doubling/halving gives amortized constant time (good)
- What if instead of doubling the size of the array, we triple the size? Good or bad?
- Resizing also includes shrinking the array by \( \frac{1}{2} \). When do we do that? When the array is less than half full or \( \frac{3}{4} \) full? What is a sequence of operations to justify your claim?

Possible/impossible questions

- We can build a heap in linear time. Is it possible to build a BST in linear time?
- Is it possible to find the max or min of any list in \( \log N \) time?
- Is it possible to create a collection where any item can be stored or found in constant time?
- Is it possible to design a max heap where find max, insertions and deletions can be done in constant time?

Possible/impossible questions

- Is it possible to sort a list of \( n \) keys in linear time, where only \( d \) (some small constant) distinct keys exists among \( n \) keys?
- Is it possible to find median in an unsorted list in linear time?

Possible/impossible questions

- Why do we ever use a BST when we can always use a hash table?
- Why do we ever use arrays when we can use linked lists?
- Why do we ever use a heap when we can always use a LLRB?
Union-find

Weighted quick-union

- Maintain heuristics
  - When merging two trees, smaller one gets attached to larger one – height does not increase
  - Height only increase when two trees are the same size

Quick-union and quick-find

Weighted Union-find question

Circle the letters corresponding to id[i] arrays that cannot possibly occur during the execution of the weighted quick-union algorithm.

- [ ] A. a[i]: 0 0 0 0 0 0 0 0
- [ ] B. a[i]: 0 1 0 0 0 0 0 0
- [ ] C. a[i]: 0 0 0 0 0 0 0 0
- [ ] D. a[i]: 0 0 0 0 0 0 0 0

What is the right approach to solving this?

Answer to union-find question

Circle the letters corresponding to id[i] arrays that cannot possibly occur during the execution of the weighted quick-union algorithm.

- [ ] A. a[i]: 0 0 0 0 0 0 0 0
- [ ] B. a[i]: 0 0 0 0 0 0 0 0
- [ ] C. a[i]: 0 0 0 0 0 0 0 0
- [ ] D. a[i]: 0 0 0 0 0 0 0 0

A B C

- A. The id[i] array contains a cycle: 0 → 2 → 4 → 0 → 8.
- B. The height of the forest is 4 or less.
- C. The size of tree rooted at the parent of i is less than twice the size of tree rooted at j.
- D. The following sequence of union operations would create the given id[i] array:
  2-0 3-2 3-8 6-8 0-1 2-8 3-8 6-8 0-6.
Typical question

Use the invariants to identify the sort algorithm.

Basic sorts

- **Insertion sort**
  - Invariant: $A[0...i-1]$ is sorted
  - Perform well in practice for almost sorted data
  - Can be used in quicksort and merge sort to speed things up

- **Selection sort**
  - Invariant: $A[0...i-1]$ is sorted and are the smallest elements in the array
  - Not used in practice much

Linearithmic sorts

Standard or 2-way Quick sort

- Randomize the array
- Find a pivot $A[i]$ usually
- Partition the array to find a pivot position $j$ such that $A[j] = \text{pivot}$
  - Pointer stop and swap on equal keys to pivot
- Recurse on subarrays leaving the pivot in-place
- Properties:
  - Good general purpose $\log n$ algorithm
  - Partition takes linear time
  - Not stable
  - In-place
  - Ideal for parallel implementations
  - Choosing a bad pivot can lead to quadratic performance
  - Works well when no duplicates

Demo of 2-way quick sort
3-way quick sort

- same as 2-way quicksort
- works well with duplicate keys
- same process
  - choose a pivot, say x
  - partition the array as follows

<table>
<thead>
<tr>
<th>&lt; x</th>
<th>== x</th>
<th>&gt; x</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Invariant

<table>
<thead>
<tr>
<th>&lt; x</th>
<th>== x</th>
<th>&gt; x</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- uses Dijkstra’s 3-way partitioning algorithm

3-way partitioning demo

Demo of 3-way quick sort

I M W R P D T T O S D E E P

Top-down merge sort

- facts
  - recursive
  - merging is the main operation
- performance
  - merging 2-sorted arrays takes linear time
  - merge sort tree is of height $\log N$
  - consistent linearithmic algorithm
- other properties
  - uses extra linear space
  - Stable
    - equal keys retain relative position in subsequent sorts

bottom-up merge sort

- facts
  - iterative
  - merges sub-arrays of size 2, 4, 8 ($\log N$ times) to finally get a sorted array
- performance
  - merging all sub arrays takes linear time in each step
  - merge continues $\log N$ times
  - consistent linearithmic algorithm
- other properties
  - no extra space
  - stable
    - merge step retains the position of the equal keys

Heap Sort

- build a max/min heap
- delete max/min and insert into the end of the array (if heap is implemented as an array) until heap is empty
- performance is linearithmic
- is heap sort stable?
Knuth shuffle

- Generates random permutations of a finite set
- algorithm

```java
for (int i=n-1; i > 0; i--) {
    j = random(0..i);
    exch(a[j], a[i]);
}
```

Problem 3 – sort matching

Sort Invariants
- Insertion sort - A[0..i] is sorted, A[i+1..n-1] is the orginal
- Selection sort - A[0..i] sorted and A[0..i] <= A[i+1..n-1]
- 2-way quicksort - first element is the pivot p and array is divided as A[<=p | p] <= p
- 3-way quicksort - A[<=p | p | >p]
- Mergesort (bottom-up) – pairs of elements (2’s, 4’s, 8’s etc) get sorted. Working on whole array at once
- Mergesort (top-down) – pairs of elements (2’s, 4’s, 8’s etc) get sorted. Working from left to right
- Heapsort - A[1..i] is a max heap and A[i+1..n-1] are sorted and are the largest elements
- Knuth shuffle - A[0..i] get shuffled first and display random form.

Binary heaps

- Invariant
  - for each node N
    - Key in N >= key in left child and key in right child
- good logarithmic performance for
  - insert
  - remove max
  - find max (constant)
- heap building
  - bottom-up ➔ linear time (sink each level)
  - top-down ➔ linear time (insert and swim)
**Heap questions**
- Given a heap, find out which key was inserted last?
  - It must be along the path of the rightmost leaf node in the tree
  - We always delete the root by exchanging that with the last leaf node
- Build a heap
  - Bottom-up
  - Top-down
- Applications
  - Can be used in design questions where delete, insert takes logarithmic time and find max takes constant time

**Ordered Symbol Tables**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Search</th>
<th>Binary Search</th>
<th>Insert</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>( \mathcal{O}(\log N) )</td>
<td>( \mathcal{O}(N) )</td>
<td>( \mathcal{O}(1) )</td>
<td>( \mathcal{O}(1) )</td>
</tr>
<tr>
<td>insert</td>
<td>( \mathcal{O}(\log N) )</td>
<td>( \mathcal{O}(N) )</td>
<td>( \mathcal{O}(1) )</td>
<td>( \mathcal{O}(1) )</td>
</tr>
<tr>
<td>max/min</td>
<td>( \mathcal{O}(1) )</td>
<td>( \mathcal{O}(1) )</td>
<td>( \mathcal{O}(1) )</td>
<td>( \mathcal{O}(1) )</td>
</tr>
<tr>
<td>floor/ceiling</td>
<td>( \mathcal{O}(\log N) )</td>
<td>( \mathcal{O}(\log N) )</td>
<td>( \mathcal{O}(1) )</td>
<td>( \mathcal{O}(1) )</td>
</tr>
<tr>
<td>rank</td>
<td>( \mathcal{O}(\log N) )</td>
<td>( \mathcal{O}(\log N) )</td>
<td>( \mathcal{O}(1) )</td>
<td>( \mathcal{O}(1) )</td>
</tr>
</tbody>
</table>

**2-3 Trees**

Two invariants:
- Balance invariant – each path from root to leaf node have the same length
- Order invariant – an inorder traversal of the tree produces an ordered sequence

**Balanced Trees**

**2-3 Tree operations**

**Red-black trees**
- How to represent 3-nodes?
  - Regular BST with red "glue" links.
Red-black tree properties

- A BST such that
  - No node has two red links connected to it
  - Every path from root to null link has the same number of black links
  - Red links lean left.

Red-black tree questions

- add or delete a key to/from a red-black tree and show how the tree is rebalanced
- Determining the value of an unknown node
  - Less than M, greater than G, less than L
- Know all the operations
  - Left rotation, right rotation, color flip
  - Know how to build a LLRB using operations
- Know how to go from 2-3 tree to a red-black tree and vice versa

Symbol Tables

Hashing

- Simple idea
- Given a key, find a hash function H(key) that computes an integer value.
- Create a table of size M and use H(key)%M to find a place.
- Hard to avoid collisions
  - Separate chaining
  - Linear probing
- Choose a good hash function
  - Easy to compute
  - Avoid collisions
  - Keep chain lengths to be $\Theta(\log N / \log \log N)$ using a random distribution of keys

Hashing type questions

- Given a set of keys, which table could result in?
  - Look for keys that are in the table corresponding to their hash values
    - They were inserted first
  - There must be at least one key that is in the position of the hash value (first key inserted)
- Know the value of a good hash function
- Know how collisions are resolved using
  - Separate chaining
  - Linear probing
- Know when to resize the hash table
Algorithm and Data Structure Design

Covered in details in design session. See design notes on midterm site.

Design problem #1

- Design a randomizedArray structure that can insert and delete a random item from the array. Need to guarantee amortized constant performance.
  - Insert(item item)
  - delete()