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 11:12:20pm.
• 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.
• 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
• Algorithms, 4th edition, Sections 1.3–1.5,
  Chapter 2, and Chapter 3.
• Lectures 1–10.
• Programming assignments 1–4.

Concepts (so far) in a nutshell

List of algorithms and data structures:
- Quick find
- Analysis of Algorithms
- Quick union
- Stacks and Queues
- Binary search
- Elementary Sets
- Disjoint sets
- Merge sort
- Hash tables
- Quicksort
- Balanced search trees
- Priority queues

- Linear probing
- Hash tables
- 2-3 trees
- Hashing
- Insertion sort
- Hashing
- Selection sort
- Hashing
- Sequences search
- Hashing
- Binary search

- BSTs
- Weighted quick-union
- 2-way quicksort
- Direct access files
- Heapsort
- Lazy deletion
- Heapsort
- Left-leaning red-black BSTs
- Heapsort
- Linear probing

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

Question

True or False

1. Tilda notation includes the coefficient of the highest order term.
2. Tilda notation provides both an upper bound and a lower bound on the growth of a function.
3. Big O notation suppresses lower order terms, so it does not necessarily accurately describe the behavior of a function for small values of \( N \).

Count operations

- \( n \) count = 0;
- for (int i = 0; i < n; i++)
- for (int j = 0; j < n; j++)
- if (l[i] <= h[l]) (h[l] + 1) = (h[l] + 1) + 1;

Suppose that it takes 1 second to execute this code fragment when \( N = 1000 \). Using tilda notation, formulate a hypothesis for the running time (in seconds) of the code fragment as a function of \( N \).

Analysis of Algorithms

- Estimate the performance of an algorithm using
  - \( \tilde{O} \), \( \tilde{\Omega} \), \( \tilde{\Theta} \): order of growth
  - comparisons, array accesses, exchanges, memory requirements

- best, worst, average
  - Performance measure based on some specific inputs

Amortized analysis

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

Analysis of Algorithms

- More formally...
  - tilda notation

- Techniques:
  - count operations
    - Operations \( \rightarrow \) 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
Count operations

Example 1
```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= D; j++) {
        if (genomes[i-1].length > genomes[j].length())
            swap(genomes[i-1], genomes[j]);
        else break;
    }
}
```

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

Example 2
```
while (i <= N) {
    int current = all[i];
    int j = i;
    while (j > 0) {
        if (current < genomes[j])
            break;
        j--;
    }
    i = j;
}
```

Count operations

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
  - $\sum f(n) \sim \int f(n)$

useful formulas
```
\sum_{i=1}^{n} i = \frac{n(n+1)}{2}
\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}
\sum_{i=1}^{n} n^2 = \frac{n(n+1)}{2}
```

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.
```
let N = Integer.parseInt(args[0]);
for (int i = 0; i < N; i++) {
    in file = new In("genomicFile" + i + ".txt");
    genomes[i] = inFile.readString();
}
```

<table>
<thead>
<tr>
<th>$N$</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.15</td>
</tr>
<tr>
<td>2</td>
<td>0.14</td>
</tr>
<tr>
<td>4</td>
<td>0.29</td>
</tr>
<tr>
<td>8</td>
<td>0.41</td>
</tr>
<tr>
<td>16</td>
<td>0.65</td>
</tr>
<tr>
<td>32</td>
<td>1.66</td>
</tr>
<tr>
<td>64</td>
<td>3.38</td>
</tr>
</tbody>
</table>

The following runtimes were observed from an algorithm that reads a file of strings and sort them using insertion sort. The runtime analysis seems to suggest the algorithm is linear. Is this correct?
```
T(n) = aN + \text{power law}
```

Mathematically speaking
- write recurrences for many of the standard algorithms
  - linear search $T(n) = 1 + T(n-1)$
  - binary search $T(n) = 1 + T(n/2)$
  - merge sort $T(n) = 2T(n/2) + n$
  - quicksort $T(n) = T(n-i-1) + T(i) + n$
  - insertion sort $T(n) = i + T(n-i-1)$
- solve them using many of the techniques discussed
  - Repeated application with base case like $T(0) = 1$ or 0
  - $T(n) = 1 + T(n-1) = 1 + (1 + T(n-2)) = \ldots$
counting memory

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

```java
public class TreeThreeTreeKey extends Comparable<Key>, Value {
    private Node root;
    private class Node {
        private int count;  // subtree count
        private Key key1, key2;  // the one or two keys
        private Value val1, val2;  // the one or two values
        private Node left, middle, right;  // the two or three subtrees
        ...
    }
}
```

- How much memory is needed for a 2-3 tree object that holds N nodes?

Stack 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 deq)
  - design a queue using a resizable array
  - design a queue using two stacks

Data Structure Performance estimates (worst or amortized)

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>find</th>
<th>insert</th>
<th>delete</th>
<th>update</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ordered array</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>resizable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>linked list</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ordered linked list</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>queue</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stack</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary heap</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LLRB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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: enque and deque

Stacks and Queues
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 \( \frac{1}{4} \) full? What is a sequence of operations to justify your claim?

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

- is it possible to implement a FIFO queue using a single array, still have amortized constant time for enqueue and dequeue?
- Is it possible to solve the 3-sum problem in \( n \log n \) time?

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 an 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?

Why?

- 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

quick-union and quick-find

Weighted quick-union

Weighted Union-find question

Answer to union-find question

Sorting
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

**Standard or 2-way Quick sort**

- Randomize the array
- Find a pivot (A[j] usually)
- Partition the array to find a pivot position $j$ such that $A[j] = pivot$
  - Pointers stop and swap on equal keys to pivot
  - Recurse on subarrays leaving the pivot in-place
- Properties
  - Good general purpose $n \log_2 n$ algorithm
  - Partitioning 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**

< x | x | > x

I M I W A R F D T T O S D E E P
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
    - Invariant
      - \(< x \quad == \quad x \quad > x\)
- uses Dijkstra’s 3-way partitioning algorithm

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

3-way partitioning demo

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

Demo of 3-way quick sort

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]);
```

Sorting question

- Suppose you are sorting n equal keys labeled k₁, k₂, k₃, ..., kₙ
- Identify the number of compares (in terms of n) required when applying the following algorithms
  - Insertion sort
  - Selection sort
  - 2-way quicksort
  - 3-way quicksort
  - Mergesort
  - Heapsort

Priority Queues

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 ➔ linearithmic (insert and swim)
Heap questions
- Given a heap, find out which key was inserted last?
  - it must be along the path of the right most 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>Sequence search</th>
<th>Binary search</th>
<th>BST</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>N</td>
<td>log N</td>
<td>A</td>
</tr>
<tr>
<td>insert</td>
<td>N</td>
<td>N</td>
<td>A</td>
</tr>
<tr>
<td>min / max</td>
<td>N</td>
<td>N</td>
<td>A</td>
</tr>
<tr>
<td>floor / ceiling</td>
<td>N</td>
<td>log N</td>
<td>A</td>
</tr>
<tr>
<td>rank</td>
<td>N</td>
<td>log N</td>
<td>A</td>
</tr>
<tr>
<td>select</td>
<td>N</td>
<td>N</td>
<td>A</td>
</tr>
<tr>
<td>ordered iteration</td>
<td>N log N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

2-3 Trees

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

2-3 Tree operations

Red-black trees
- How to represent 3-nodes?
  - Regular BST with red “glue” links.

Balanced Trees
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 $O(\log N / \log \log N)$ 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 detail in design session.
See design notes on midterm site

Design problems

- Typically challenging
- There can be many possible solutions
  - Partial credit awarded
- Usually it is a data structure design to meet certain performance requirements for a given set of operations
  - Example, create a data structure that meets the following performance requirements
    - \( \text{findMedian} \) in \( \sim n \), \( \text{insert} \) \( \sim \log n \), \( \text{delete} \) \( \sim \log n \)
  - Example: A leaky queue that can remove from any point, that can insert to end and delete from front, all in logarithmic time or better
- Typical cues to look for
  - \( \log n \) time may indicate that you need a sorted array or balanced BST or some sort of a heap
  - Amortized time may indicate, you can have some costly operations once in a while, but on average, it must perform as expected

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()