

6.046 Quiz 1 Review Handout

March 7, 2008

1 Disclaimer

Do not study by only looking at this handout. This handout makes no claims to fully cover everything that will come up on the exam!

2 6.046 Quiz 1

When Tuesday, March 11, 2008, 11:05am - 12:25 pm during normal class hours. Please show up early so that you can take advantage of the full time available. The exam is 80 minutes long.

Where 2-190

Crib sheet The quiz is closed book. You may, however, bring one handwritten crib sheet on an 8 1/2" x 11" or A4 paper. Preparing a crib sheet can be a useful study aid, so take time in selecting material for it. You may use both sides of the paper and write as small as you like, but you are allowed only one sheet. It must be handwritten and not photocopied or printed. Calculators and programmable devices are not allowed for this quiz. The crib sheet must be turned in with the quiz booklet. It will be returned with the graded quiz.

Material covered You are responsible for all material covered in the lectures, recitations, and problem sets through the recitation on Friday, March 7, with the exception of the topic of open addressing (covered at the end of lecture on March 4) and the topic of tree-based Union-Find (covered at the end of lecture on March 6). You are also responsible for the material covered in the corresponding sections in the textbook, as indicated by the reading assignments on the course calendar.

Practice quiz A practice quiz and its solutions are on the website.

Exam conflict If you are unable to attend the exam at the scheduled time, please talk to one of the TAs in advance (and as soon as possible).

3 Analysis of Algorithms

3.1 Running time $T(n)$

Measure of an algorithm's running time for input of size n .

- worst-case - maximum time for an algorithm
- average-case - expected time of algorithm over all inputs of size n ; need assumption of statistical distribution of inputs
- best-case - minimum running time for some input (bogus measure)

3.2 Asymptotic Notation

Tight asymptotic bounds

$O(g(n)) = \{f(n) \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$

$\Omega(g(n)) = \{f(n) \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$

$\Theta(g(n)) = \{f(n) \text{ there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0\}$

Strict asymptotic bounds

$o(g(n)) = \{f(n) \text{ for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}$

$\omega(g(n)) = \{f(n) \text{ for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}$

Intuitive definitions of the asymptotic bounds

$$\begin{aligned} f(n) = O(g(n)) &\approx f \leq g \\ f(n) = \Omega(g(n)) &\approx f \geq g \\ f(n) = \Theta(g(n)) &\approx f = g \\ f(n) = o(g(n)) &\approx f < g \\ f(n) = \omega(g(n)) &\approx f > g \end{aligned}$$

Not all nonnegative functions can be compared using asymptotic notation. For example, two functions are not comparable if one of them keeps oscillating between 0 and some function that is strictly larger than the other function. No matter how large n gets, you can't apply any of these definitions to the two functions.

3.3 Recurrences

3.3.1 Substitution Method

Essentially a guess-and-check method.

1. Guess the form of the solution. (e.g. $T(n) = O(f(n))$)
2. Verify by induction. (e.g. assume $T(k) \leq cf(n)$ for $k < n$)
3. Solve for constants. (e.g. prove $T(n) \leq cf(n)$)

3.3.2 Recursion Trees

Used for getting intuition about a recurrence. Cannot be used to prove a bound, but it can give you a guess that you can verify using the substitution method.

3.3.3 Master Method

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence $T(n) = aT(n/b) + f(n)$ where n/b means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$.

3.3.4 Some Tricks

- Substitution of variables for recurrences of the form $T(n) = aT(\sqrt{n}) + f(n)$. Set $m = \log n$ and $S(m) = T(2^m)$, use Master Method, then substitute everything back in.
- Ignoring lower order terms. For example, for $T(n) = aT(n/b + \log n) + f(n)$, you can ignore the $\log n$ because it's asymptotically smaller than the n/b .

3.3.5 Sloppiness Theorem

If the following are true:

1. $T(n), g(n)$ both monotonically increasing
2. $T(b^i) \leq g(b^i)$ for all integers $i > 0$
3. $g(n) = O(g(n/b))$ (i.e. $g(n)$ grows polynomially, not exponentially or such)

Then $T(n) = O(g(n))$ and you can ignore floors and ceilings.

3.4 Examples Covered

- Insertion sort - scan through the list and put each item in the correct place in the sorted list of already scanned items. This takes $\Theta(n^2)$ time.
- Merge sort - recursively sort the first and second halves of the list and merge them afterwards by scanning through the sorted lists in linear time. This takes $\Theta(n \log n)$ time.

3.5 Correctness

Use *induction* arguments to prove correctness of *recursive algorithms* and for proving loop invariants for *iterative algorithms*.

4 Divide and Conquer

The paradigm

1. Divide the problem into subproblems
2. Conquer the subproblems by solving them recursively
3. Combine subproblem solutions

Examples of divide and conquer algorithms: mergesort, binary search, recursive squaring, polynomial multiplication, matrix multiplication, Fibonacci numbers, quicksort, bucket sort
Can be analyzed with recurrences and the master method.

5 Probabilistic and randomized algorithms

Two types of randomized algorithms:

- Las Vegas: usually fast, sometimes slow.
- Monte Carlo: usually correct, sometimes wrong. matrix product checker

Examples: matrix product checker, quicksort

5.1 Indicator Random Variables

An *indicator random variable* X is the special case of *random variables* where we define:

$$X = \begin{cases} 1 & \text{if some event happens} \\ 0 & \text{otherwise} \end{cases}$$

Properties

- By definition, $E[X] = \Pr[X = 1]$.
- Linearity of expectation: the expectation of a sum of random variables is equal to the sum of their expectations, i.e. $E[\sum_i X_i] = \sum_i E[X_i]$. This is true even if the different events are not independent.
- If the variables are independent, then the expectation of the product of random variables is going to be equal to the product of their expectations, i.e. $E[\prod_i X_i] = \prod_i E[X_i]$.
- Indicator random variables can be used to calculate expected running times.

6 Sorting

Properties of sorts:

- In place - sort by modifying the existing array
- Stable - items that are equal stay in the same order as in the original list

6.1 Quicksort

General algorithm:

1. Divide: Partition the array into two subarrays around a pivot element x such that the lower subarray only contains elements $\leq x$ and the upper subarray only contains elements $\geq x$
2. Conquer: Recursively sort the two subarrays
3. Combine: Do nothing.

Notes:

- Partition takes linear time.
- In the worst case, one of the subarrays is empty, and it's insertion sort.
- Can randomly pick a pivot to get $\Theta(n \lg n)$ in average case.
- Can deterministically pick a good pivot using the linear time algorithm Select.

6.2 Lower bounds on comparison sorting

Comparison model: can only use comparison of two elements to determine relative ordering

Sorting takes $\Omega(n \lg n)$ in the comparison model.

Comparison sorts: insertion sort, merge sort, quicksort, heapsort

6.3 Linear-time sorting

We can beat the $\Omega(n \lg n)$ bound for easier sorting problems. For example, if input is all integers in a particular range, it cuts down a continuous range of infinitely many possible numbers to a much smaller finite set.

Examples: counting sort, radix sort, bucket sort

6.4 Summary

You should review the details of all the different sorting algorithms to know how they work. The following table is a good reference though.

Algorithm	Worst-case	Average-case	In place	Stable	Comments
Insertion	$\Theta(n^2)$		Y	Y	small constants
Merge	$\Theta(n \lg n)$		N	Y	stability depends on merge
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$	Y	N	assuming every permutation equally likely
randomized	$\Theta(n^2)$	$\Theta(n \lg n)$	Y	N	no adversary elicits worst case average behavior
median-partitioned	$\Theta(n \lg n)$		Y	N	large constants, not used in practice
Heapsort	$\Theta(n \lg n)$		Y	N	build heap in $O(n)$
Counting	$\Theta(n + k)$		N	Y	integers, k possible values
Radix	$\Theta(d(n + k))$		N	Y	integers; need stable auxiliary sort, d digits, k possible digits
Bucket	$\Theta(n^2)$	$\Theta(n)$	N	Y	real numbers, uniform over fixed range

7 Data Structures

7.1 Heaps

Nearly complete binary trees with the heap property: each item is bigger than all of its children (for a max heap). (Can also define min-heap by saying all parents are smaller than all their children.)

Operations (for a max-heap)

- Build-heap $O(n)$
- Insert $O(\lg n)$
- Heapify $O(\lg n)$
- Extract-max $O(\lg n)$

Applications

- priority queue
- sorting

7.2 Binary search trees

BST property: For any node x in a BST, all elements to the left of it are less than or equal to the value of x , and all elements to the right of it are greater than or equal to the value of x .

Operations:

- Insert

- Delete
- Search
- Predecessor
- Successor

If h is the height of the BST, it takes $O(h)$ time for each of these queries.

7.2.1 Balanced binary search trees

BSTs for which height stays $\Theta(\lg n)$, so queries run in $O(\lg n)$. Example: red-black trees.

Red-black trees use rotations to maintain red-black tree properties when insertions and deletions happen. Key property: on any path from a node down to the leaves, more than half of them will be black and there are the same number of them on each path.

7.2.2 Tree walks

These take $\Theta(n)$ time.

- inorder - process left subtree, then root, then right subtree
- preorder - process root before the child subtrees
- postorder process root after processing the child subtrees

8 Order statistics

Finding the i -th smallest element.

8.1 Randomized Select

Randomly pick pivot (as in quicksort), partition, and look in the right subarray that would contain the i -th smallest element, and recurse.

Worst case: $O(n^2)$ if all partitions are unlucky

Expected time: $O(n)$ (can be shown using indicator random variables)

Very useful in practice

8.2 Deterministic Select

This algorithm has big constants, so it's impractical, but it does guarantee $O(n)$ time.

1. Divide the n elements into groups of 5. Find the median of each group. This takes $\Theta(n)$ time.
2. Recursively select the median of the medians. This takes $T(n/5)$ time.
3. Use this median of medians as a pivot, partition, and recurse in the appropriate subarray (like in Randomized Select). This takes $\Theta(n) + T(7n/10)$ time.

Solving the recurrence $T(n) = T(n/5) + T(7n/10) + \Theta(n)$ gives $T(n) = \Theta(n)$.

9 Hashing

Hash function maps a large set of keys U to a (usually) smaller set of m values. Outcomes are usually stored in a hash table.

Collisions when more than one key is mapped to the same value. Can be resolved by keeping a linked list for that value.

Simple uniform hashing each key $k \in K$ of keys is equally likely to be hashed to any slot of table T , independent of where other keys are hashed.

Load factor $\alpha = n/m$, where n is the number of keys to be stored and m is the number of slots.

Examples of hash functions

- Division method: $h(k) = k \bmod m$, m is prime
- Multiplication method: $h(k) = (A \cdot k \bmod 2^w) \text{ rsh } (w - r)$, A is an odd integer between 2^{w-1} and 2^w and w is the machine word size. Fast to compute
- Dot-product method: $h_a(k) = \sum_{i=0}^r a_i k_i \bmod m$, m is prime, a is a random vector, and k and a are decomposed into $r + 1$ digits. Expensive to compute.

9.1 Universal hashing

If H is a finite collection of hash functions that map U to the m values $\{0, 1, \dots, m - 1\}$, it is universal if for all $x, y \in U$ where $x \neq y$, only $1/m$ of the hash functions in H result in a collision for x and y .

Theorem Let h be a hash function chosen uniformly at random from a universal set H of hash functions. Suppose h is used to hash n arbitrary keys into the m slots of a table T . Then, for a given key x , we have $E[\# \text{ collisions with } x] < n/m$.

Example: Dot-product method

9.2 Perfect hashing

Used when the set of keys you'll ever use doesn't change. Examples apparently include reserved words in a programming language or a set of file names on a CD-ROM.

Idea: Use universal hashing and deal with collisions by using another hash table rather than a linked list. There are thus two layers of hash tables. In total, this hashes n keys to a table of size $m = O(n)$, and queries will take $O(1)$ in worst case.

Let n_i be the number of keys that have a collision in the first level of hashing. If we choose to hash them again to a hash table of size $m = n_i^2$, then by universal hashing, we get less than $1/2$ a collision in expectation. In expectation, we can thus try two different random hash functions and get one that has no collisions at all. Similar to the bucket sort analysis that showed $O(n_i^2)$ work per bucket had overall work of $O(n)$ in expectation, storage is also $\Theta(n)$ in expectation. In the worst case, queries require hashing the item twice which takes $O(1)$ time.

10 Amortized analysis

Analysis of of the *total* running time taken by any sequence of m operations. *Not* the same as average case analysis, which assumes random distribution over the input or a random sequence of operations.

Example from class: Union-Find using linked lists. By merging smaller lists into larger lists, total cost of unioning is $O(m \log n)$ time for m unions, so we get an amortized running time of $O(\log n)$ per operation.

You aren't responsible for the details, but Union-Find can also be implemented with trees. By merging smaller trees into larger trees and doing path compression, we get an amortized running time of $O(\alpha(n))$, which grows incredibly slowly.

6.046 Recitation 5 Anonymous Survey and Feedback

How am I doing? Has there been anything lacking in the recitations? Anything that I should keep doing for sure?

Do you have any feedback on how lectures have been going? Anything that bugs you or makes you happy?

If you have come to any of our office hours, have you been satisfied with your experience?

Are there any topics that you thought were not well-explained? What has been the hardest part of the course so far?