**Assessing the Influence of Internships on Technical Knowledge of Graduating CS Students**

**By Tong Zou, April 10, 2011**

**Abstract**There are very few studies out there that can reliably measure the impact of doing an internship on a graduating computer science student. In this project, I attempted to assess the influence of an internship by giving out a questionnaire to two groups of graduating CS students, one group which had done PEY (University of Toronto’s internship program), and one group which had not, while establishing a consistent marking scheme across responses. The results indicated that the sampling data was too sparse to reliably measure any effect, but similar studies in the future are to be encouraged, in the hopes of a more consequential result.

**Introduction**The motivation for this project came about after a realization that the lack of studies that could quantitatively measure the effect of internship experience on graduating students. It would be interesting to graduating students to know how internship experience prepared them for securing a job in the real world, besides the obvious benefit of having fulfilled the experience requirements of entry level positions. I was looking for a way to discern if there really was a substantial difference in the way that graduating students with internships approached problem solving and knowledge questions in computer science versus those who hadn’t done internships. If there really was a significant difference, then the fruits of this research would have a substantial impact on employers, students, and internship departments. Thus, graduating CS students were sampled to observe and analyze differences.

**Method**Having established the context and background, I had to decide the main method for gathering data. I focused mainly on two different tools for gathering data: the interview and the survey/questionnaire, because of the nature of the project. Face to face interviews with a set of graduating CS students would result in a richer, more substantial data set, but would be costlier to participants, whereas questionnaires would result in a poorer data set, but at lesser cost to participants. I ended up settling on distributing a set of questionnaires online using Survey Monkey[[1]](#footnote-1). The surveys were inherently unsupervised since they were taken online, but I had to make a tradeoff between a smaller, richer data set involving interviews, which students would have not easily agreed to the middle of midterm season, and a larger, less substantial data set using the online questionnaire.

**Design**The design of the questionnaires were of particular importance because of the cost-benefit tradeoff: the more detailed and richer the questionnaire, the less chance that a student was willing to finish it; there had to be a balance so that the questions were challenging enough to provoke a thoughtful response, but not too difficult as to deter participants from answering. Since analyzing how participants fared on problem solving and knowledge questions in CS was important, the questions needed to be technical in nature. In addition, the opening questions needed to behavioral[[2]](#footnote-2) in order to extract the participants’ technical background and their experience with PEY.

This was done simply with the first three questions asking about the participants’ involvement with PEY, their final mark range in CSC108, and their opinion of their PEY experience, respectively. Their final mark range in CSC108 or in their first programming course should provide an approximate estimate of their abilities as a novice programmer, and their opinion of PEY experience was ranked by them, using a rating of 1 (PEY not helpful at all) to 5 (could not pass without PEY). The length of the questionnaire was restricted to 10 questions due to Survey Monkey’s free account limit, but having the questionnaire be any longer would have turned off participants.

The remaining questions had to be technical in nature, and test participants’ CS knowledge in a wide variety of domains relating to computer science, without being too trivial or too difficult. These questions were taken from common interview questions often asked by Google, Amazon, Microsoft, and other IT companies because they represented what an employer wanted a graduating student to know[[3]](#footnote-3). In addition, solutions for these questions were available to facilitate assessment[[4]](#footnote-4). From these questions, the main knowledge areas of CS that an employer wanted can be roughly divided into seven categories: Data Structures, Algorithms, Object Oriented Design, Testing, Databases, Networking, and Low level /System Design[[5]](#footnote-5). From these categories, Data Structures, Algorithms, Object Oriented Design and Low level/System Design were used because the other three categories are considered optional knowledge by University of Toronto standards[[6]](#footnote-6). Data Structures cover questions involving arrays, stacks, queues, hash tables, trees, graphs, and linked lists. Algorithms is a broad category, which covers different algorithms on searching, sorting, recursion, dynamic programming, mathematical, and large systems[[7]](#footnote-7). Object Oriented Design (OOD) is an interesting category because it implies that most employers currently value the OO paradigm over other paradigms (dynamic, procedural, functional, logical); this category also includes knowledge questions specifically pertaining to OO languages such as Java, Python, C++, etc. Low level/System Design covers questions on bit manipulation, physical & virtual memory, threads and locks.

From these categories, seven questions had to be chosen in such a way that the knowledge would be spread out, and worded in such a way as to provoke insight into how participants answered. It was decided that 4 questions be short technical teasers, which typically had only one right answer with additional room for more explanation, and the other 3 be longer technical questions which required algorithmic responses that may differ in explanation. Each question was selected such that they could be answered by someone who finished the “core” courses in computer science, and that each algorithm question was doable by students who had finished first year computer science[[8]](#footnote-8). One question each came from low level computing, sorting & complexity, data structures, systems design, linked lists, arrays & searching, and object oriented design. Appendix A contains the actual survey questions, along with sample solutions for each question.

**Marking and Assessment**In order to compare the different responses, a marking scheme had to be developed that could reasonably measure the quality of responses. Since the textbook solutions were already available, I decided to give a rating between 1 to 5 for responses; 1 indicates the answer has no correlation with the question, 2 indicates a poor answer with lots of mistakes, 3 indicates a decent response that goes in the right direction, 4 indicates a good answer with some details left out, and 5 indicates an excellent or innovative answer that is exactly in the line with the solution. Of course, this is still subjective, but I tried to maintain consistency across responses. The full marking scheme is provided in Appendix B.

**Data Collection and Analysis**16 responses by CS graduating students were received, 11 students were PEY, and 5 of which were not. The responses to the behavioral questions were somewhat surprising. About 81% of respondents indicated that they achieved a final mark of 83 or higher in their first year programming course. 73% of PEY respondents indicated that their PEY experience was not helpful in their courses, or only somewhat helpful in an unspecified sense. These responses imply that most participants were fairly competent in their programming level in first year, enough to advance to fourth year computer science, and that PEY experience was somewhat confined to the work environment rather than the abstract environment of the classroom.

Some interesting results were also gathered from the 4 short technical teaser responses. Please note that in the following sentences, citations of x% of respondents, mean out of all valid responses (taking out nonsense answers). On the low level computing question (Q4), 38% of respondents (all PEY) mentioned context switch or disk seek as the fastest operation, indicating that most respondents were unfamiliar with low level operations. On the sorting and complexity question (Q5), 50% of respondents (both PEY and non PEY) got both running times correct. On the deadlock (system design) question (Q6), 100% of respondents got the definition right, although there was rather a high number of nonsense responses, implying that some respondents did not take an operating systems course (CSC369), and those that did, retained their knowledge well. On the data structures question (Q7), responses varied, but 45% of respondents (both PEY and non PEY) mentioned using BST to preserve natural ordering over hash tables.

The three algorithmic questions also produced a variety of interesting responses. On the linked list question (Q8), 60% of respondents (both PEY and non PEY) used iteration in their algorithm, which the linked list data structure facilitates. On the array & searching question (Q9), 82% of respondents used a hash table of some sort in their algorithm, and PEY respondents especially liked to use a hash table with a Boolean flag to solve the question. Finally, on the object oriented design question (Q10), about the same amount of PEY respondents decided to use just classes as non PEY respondents who decided to use classes plus relevant methods, although this distinction is quite blurred as respondents weren’t asked to write complete pseudo code for this question. Furthermore, none of the respondents mentioned subclassing or design patterns, though a handful of PEY respondents (17%) mentioned using an interface.

When the marking scheme was applied across responses, a rather large amount of responses were nonsense answers, or answers that were not considered relevant to the question. These represent 38% of total responses, so it’s likely that a lot of participants found the survey to be too long or didn’t have the motivation or incentive to answer with effort. Adjusting for the nonsense answers (removing them), the average mark for non PEY was 3.73, and 3.4 for PEY respondents. Plotting these responses on a graph, an interesting pattern can be seen; about half of non-PEY respondents had poor answers, and the other half had great or above average answers, whereas PEY respondents exhibited more consistency, with most being around decent to above average. In short, non PEY respondents had a bi-modal distribution, and PEY respondents had a normal distribution.

Further analysis of the response data, including graphical charts, is in Appendix C. The raw data is provided in Appendix D.

**Conclusion**Overall, from the results gathered, several conclusions can be made about the study, and about the effects of PEY. The first thing of note is the number of responses. 16 students is a rather small sampling size to make many conclusions, and the large proportion of responses with a total average below 2.0 on the marking scheme (6 out of 16) add to the complication of making accurate implications from the data. From the responses, it’s clear that PEY knowledge is not necessarily applicable to classroom material from the opinions of the students. From the valid responses given on the technical questions and applying the marking scheme across them, there seems to be a little to no correlation between those who did PEY and those who didn’t. Moreover, those with PEY experience tended to give decent to above average responses (a normal distribution), and those who didn’t tended to give either excellent or poor responses (a bi-modal distribution). However, since the sample size was so small, the significance and reliability of these results is debatable.

Nevertheless, the data gathered was still interesting to analyze and I encourage further research to be done in this area of CS education; comparing technical knowledge of those with internship experience to those without. Potential problems that my study ran into was self-selection bias, since the survey was conducted informally, and those with a strong opinion about PEY may have been more likely to participate. Students had no incentive to complete these surveys since there were no rewards or participation marks for them, and they could have searched for answers online while completing the survey. Age could have been another factor, since PEY takes a year or more to complete, and this could have affected the responses. I recommend that future studies administer the survey to classrooms with additional incentives of prizes or course credit, and acquire a large sample size for which statistical analysis such as t-tests can be done. This would ensure a more statistically valid and therefore significant result can be obtained, from which students, employers and internship departments can all benefit.

**Appendix A: CS Questionnaire and Solutions**

**Behavioral questions:**

1. **Did you participate in the PEY program?**Choices: Yes, No
2. **What was your final mark range for your first programming course at UofT?**Choices: < 50, 51-57, 57-63, 64-69, 70-73, 74-77, 78-82, 83+
3. **How has your PEY experience helped you in your CS courses this year?**Choices: 1 (didn’t help), 2 (somewhat), 3 (helped with some material), 4 (helped with assignments), 5 (would fail without it)

**Technical teasers:**

1. ***[Low level Computing*] Rank these operations from fastest to slowest: Disk Seek, Context Switch, CPU access, Main memory access.**Solution: CPU access, Main memory, Disk Seek, Context Switch  
   The results indicated that most people had trouble determining how long a context switch takes. This is probably due to the unfamiliarity with context switching since the slowness of the operation is a big part of operating systems design.
2. ***[Algorithms – Sorting, Complexity]* What is the best or worst case runtime of Quicksort and when would you use it over Mergesort?**Solution: best case O(nlogn), worst case O(n^2). You would generally use Quicksort because its inner loop can be efficiently implemented in most architecture using caches and virtual memory, unless the data is close to being sorted which is Quicksort’s worst case.   
   The results indicated that most people got the running times right but few answered the second half of the question.
3. ***[System Design – Threads, Locks]* What is a deadlock and how would you prevent it?**Solution: Deadlock is when two processes are holding a lock the other one needs and both try to request the other. Good documentation, coding style, requiring a strict ordering be placed on the resource access, etc are all valid responses.   
   Most people got the definition right, but the responses to prevention varied. Some were very good such as imposing a total order over resource or lock acquisition, while others mentioned using a lock, which puzzled me because the issue of deadlock arises from using locks.
4. ***[Data Structures – Trees, Hash Tables]* When would you use a binary search tree over a hash table and why?**Solution: Binary trees are better for smaller data sets and data sets where a natural ordering is required and/or when a good hash function cannot be used and chance of collision is too high.  
   There are different correct responses to this question depending on the situation, with responses mentioning ordering, space/memory, and hash function collision being the most common.

**Technical questions:**

1. ***[Algorithms – Linked Lists, Recursion, Iteration]* How would you reverse a singly linked list? Explain your algorithm in plain English or in pseudo code.**Solution: This can be done recursively or iteratively. A way to do it recursively is to have a method that takes in head node and a second node. If the head node equals null, return the second node. Else create a temp node equal to head node.next, set head node.next equal to the second node, and recurse using the temp node as the new head node and the head node as the second node.   
    Although this question can be solved quite simply by using recursion, most answers used iteration to traverse the linked list, some using another data structure (extra space) or traversing it twice (extra time).
2. ***[Algorithms – Arrays, Recursion, Iteration, Searching*] Given an array of integers, all of which appear an even number of times except for one, how would you find that integer? Explain your algorithm in plain English or in pseudocode.**Solution: Many ways to do this, but the most efficient solution is to XOR all of the numbers and the result would be that integer.   
   No participant gave the XOR response, but interestingly, almost all the valid responses used a hash table of some form, usually Boolean or integer. This required additional space and time to construct and traverse the hash table. One variation used a stack to do this instead of a hash table.
3. ***[Object Oriented Design]* How would you design a generic card game using object oriented principles? We might want to implement more specific versions like Blackjack or Poker later. What classes, subclasses, methods, variables, and design patterns would you use? Just explain, pseudo code isn’t necessary.**Solution: Some classes needed might be Card, Deck, Number, Suit, Value. Useful design patterns could be Factory for creating, Template for running, Strategy for running, Observer for updating, etc. getCard, flipCard, bet, reset, etc might be good methods for a generic card game.   
   Almost all responses mentioned a Card class, but other objects such as Player, Hand, Game, and Deck were given as well, along with associated methods such as getCard, dealHand, shuffle, etc. This seems reasonable given the question context, but it’s interesting that none of the responses mentioned subclasses, modifiers, or design patterns, although some mentioned using an interface. My explanation for this is that participants didn’t want to go into a high level of detail, especially since I mentioned that pseudo code wasn’t necessary.

**Appendix B: Marking scheme – Response keys:   
  
1 – Candidate has not answered the question sufficiently or answers don’t make sense**Note: I mark a response 1 when the answer is blank and/or has nothing to do with the question.  
Example (Response #1 for Q6): Call the locksmith!  
 **2 – Candidate displays some knowledge of the questions but makes many errors and blatantly wrong responses**Note: I mark a response 2 when the answer is clearly wrong but the participant made an effort to answer it.   
Example (Response #7 for Q4): Disk seek, Context switch, Main memory access.CPU access  
 **3 – Candidate displays knowledge of the questions, makes some errors and algorithms are inefficient, but gets the idea.**Note: I mark a response 3 when the participant gets the idea and makes a good attempt but the answer is inefficient, has lots of errors and/or not detailed enough.   
Example (Reponse #4 for Q8): I would probably use recursion: func print\_list( node\* cur) print\_list(cur->next); print cur->data **4 – Candidate displays good knowledge of the questions, makes sensible efficient responses, but makes some minor mistakes.**Note: I mark a response 4 when the participant has a good answer but has some inefficiencies or minor details left out.   
Example (Response #2 for Q9): Create a hash table and then traverse the list for every element in the list: create a hash entry -> bool, start with 1 at first, then toggle its value whenever you see another key in the list at the end, only one key will have value of 1, others will have bool value of 0.  
 **5 – Candidate displays advanced knowledge of the questions, makes excellent detailed responses, and makes only one or two mistakes, if any at all.**Note: For answers that are innovative and/or efficient, or close to the textbook solution, I give the response a 5.  
Example (Response #2 for Q10): Some example classes with some of their plausible attributes and methods: - Game (attrs: players, deck, score) (methods: start, check\_victory\_conditions) - Player (attrs: name, hand) - Card (attrs: rank, suit, flipped) (methods: flip) - Deck (attrs: num\_cards, cards) (methods: shuffle) - Hand (attrs: cards) (methods: sort)

**To get an overall response mark, I add up all the individual response keys for each question and divide by 7 (total number of technical questions). Scores range from 1 to 5 based on this method.**

**Appendix C: Data Analysis**

**Q1. Did you participate in the PEY program?**

5 responses were non PEY (69%), and 11 responses were PEY (31%).   
In the future, hopefully additional non PEY responses could provide more insightful data.

**Q2. What was your final mark range for your first programming course at UofT?**

Only 3 participants got less than 83% as a final mark on their first programming course. Perhaps when students make it to 4th year, they were already quite bright to begin with, and were enthusiastic enough   
to pursue computer science into 4th year.

**Q3. How has your PEY experience helped you in your CS courses this year?**

73% of respondents indicated that their PEY experience was unhelpful or somewhat helped (but not in a specific way). This leads me to believe that internship experience is often contained within the confines of the workplace and doesn’t translate very well to the abstract and theoretical environment of the classroom.

***Q4. [Low level Computing*] Rank these operations from fastest to slowest: Disk Seek, Context Switch, CPU access, Main memory access.**   
The results indicated that most people had trouble determining how long a context switch takes. This is probably due to the unfamiliarity with context switching since the slowness of the operation is a big part of operating systems design. Most people seemed to recognize that disk seek was slow, which is a good sign. I considered putting context/disk seek first to be worse than putting CPU first, so the following graph is ranked from left (least correct) to right (most correct). From the data I collected, it seems that non PEY are actually consistently more correct than PEY responses.

***Q5. [Algorithms – Sorting, Complexity]* What is the best or worst case runtime of Quicksort and when would you use it over Mergesort?**  
The results indicated that most people got the running times right but few answered the second half of the question. There doesn’t seem to be a significant difference between PEY and non PEY with regards to responses, the PEY numbers seem double that of non PEY, but the sampling size was also twice as large.

***Q6. [System Design – Threads, Locks]* What is a deadlock and how would you prevent it?**   
Interesting thing to note here is that all responses that weren’t nonsense responses got the definition correct, but the responses to prevention varied. Some were very good such as imposing a total order over resource or lock acquisition, while others mentioned using a lock, which puzzled me because the issue of deadlock arises from using locks. The high amount of nonsense or skipped responses is most likely due to CSC369 not being a core course in second year, even though system design, and threads and locks in particular, is a subject that many employers will ask about. Again, both PEY and non PEY respondents got the definition correct, although the explanations which mentioned using locks to solve the problem came from the non PEY respondents.

***Q7. [Data Structures – Trees, Hash Tables]* When would you use a binary search tree over a hash table and why?**  
There are different correct responses to this question depending on the situation, with responses mentioning ordering, space/memory, and hash function collision being the most common. PEY respondents were most likely to mention ordering and space, while non PEY respondents tended not to mention space or memory.

***Q8. [Algorithms – Linked Lists, Recursion, Iteration]* How would you reverse a singly linked list? Explain your algorithm in plain English or in pseudo code.**  
 Although this question can be solved quite simply by using recursion, most answers used iteration to traverse the linked list, some using a stack (extra space) or traversing it twice (extra time). The high amount of iteration responses, and uniformity of responses across both PEY and non PEY respondents imply that most students think of linked lists as an iterative data structure (like arrays, whereas trees and graphs are often thought of recursively).

***Q9. [Algorithms – Arrays, Recursion, Iteration, Searching*] Given an array of integers, all of which appear an even number of times except for one, how would you find that integer? Explain your algorithm in plain English or in pseudocode.**  
No participant gave the XOR response, but interestingly, almost all the valid responses used a hash table of some form, usually Boolean or integer. This required additional space and time to construct and traverse the hash table. One variation used a stack to do this instead of a hash table, which was interesting. The most popular answer used by PEY respondents was the hash table with Boolean or int flag, whereas non PEY respondents tended to use a total count of the integers and then mod 2 to find the odd number at the end. Both methods require O (n) to traverse the table, but it is slightly less operationally intense to use the Boolean method (! instead of %).

***Q10. [Object Oriented Design]* How would you design a generic card game using object oriented principles? We might want to implement more specific versions like Blackjack or Poker later. What classes, subclasses, methods, variables, and design patterns would you use? Just explain, pseudo code isn’t necessary.**  
Almost all responses mentioned a Card class, but other objects such as Player, Hand, Game, and Deck were given as well, along with associated methods such as getCard, dealHand, shuffle, etc. This seems reasonable given the question context, but it’s interesting that none of the responses mentioned subclasses, modifiers, or design patterns, although some mentioned using an interface. My explanation for this is that participants didn’t want to go into a high level of detail, especially since I mentioned that pseudo code wasn’t necessary. In the following graph, classes only mean the respondents only mentioned a class with optional variables, whereas classes + methods means that both classes and several associated methods are mentioned. It’s interesting to note that only PEY respondents chose to mention using an interface as a part of class implementation, which is part of many design patterns, whereas non PEY respondents tended to mention using many classes, methods and attributes together.

**Marking results:**

Using the marking scheme, results were mixed. Taking results less than 2 out of consideration (nonsense responses), we have 3.73 as the average mark for non PEY, and 3.4 for PEY. For non PEY, about half as many respondents gave excellent answers as poor answers, but for PEY students this was more consistent, with the majority of the responses hovering around the decent to above average level. Additional sampling data is needed for a more accurate conclusion (note that we only have 10 responses that are not nonsense and worthy of consideration).

**Appendix D: CS Graduating Students Technical Questionnaire responses (raw data)**

Each response is sorted under those who did PEY and those who didn’t and gathered under each of the 10 questions. For multiple choice questions, totals are tallied after each choice.

**With PEY:**

**Q2 (CSC108 final mark range)**64-69: 1  
78-82: 1  
83+: 9

**Q3 (PEY experience rating)**1: 42: 4  
3: 1  
4: 2

**Q4 (Low level computing)**Me gusta.

Context switch, cpu access, main memory access, disk seek

Context switch, CPU access, Main memory access, Disk seek

Turtle, Rabbit, Lion

context switch, CPU access, Main memory access, disk seek

cpu access, context switch, main memory access, disk seek

Disk seek, Context switch, Main memory access.CPU access

Context switch, Main memory access, CPU access, Disk seek

CPU access, Main memory access, Context switch, Disk seek

1 CPU access, 2 Main memory access, 3 Disk seek, 4 Context switch

CPU access, Main memory access, Context switch, Disk seek

**Q5 (Algorithms – Sorting, Complexity)**Peanut butter and jelly.

O(nlogn) and O(n^2) respectively.

Best case: O(n) Worst case: O(mg)

The best case and worst case runtime of quicksort in big O notation is O(mg)

0(logn) and O(nlogn). not sure when you would use it over mergesort, but both are good.

n squared.

When i am too lazy to write merge sort

Ice cream.

worst: n^2 best: nlogn Memory space restriction. Quicksort can also be modified to become an efficient algorithm for selection. (Quick select)

best O(n log n) worst O(n^2)

Best case: O(nlogn) Worst case: O(n^2) You would use quicksort over mergesort when not sorting a huge amount of data.

**Q6 (System Design – Threads, Locks)**

Call the locksmith!

Two processes waiting on one another to release resource. Break by either preventing locks or mutual resource

Use deadlock protection.

Sleep

Didn’t take 369.

Don't know

A deadlock is when you have a lock and is dead

Deadlock is a cycle of dependencies where each resource is holding onto a lock and trying to acquire another lock. One way to prevent it is have a time limit for lock acquisition.

When multiple threads/processes are requesting and waiting on resources being held on by other threads/processes in the same situation. You can prevent it by: - Enforcing the program can only request resources in bulk. It may be forced to drop its current resources before it can make new resource requests. - Resource ordering

Two threads waiting on each other, ie T1 waits for T2 to release a resource and T2 waits for T1

A deadlock occurs when a process is waiting for feedback from another process, and this other process is also waiting for feedback from the previous process.

**Q7 (Data Structures – Trees, Hash Tables)**Save the trees for Mother Earth!

Use bst when there's a chance of a lot of collision in hashing algorithm when using hash table

Use a binary search tree when we have items that are multiples of 2. Use a hash table when you want things done faster.

Never

you use a binary search tree when you have too many elements in each row of the hash table

Less memory.

When I am on crack.

If you are a n00b, you use a BST over a hash table.

BST preserves index ordering whereas hashing tend not to. One example is printing out the order of nodes in BST. BST is memory efficient. You only need to allocate as much memory as the number of nodes, whereas you need to preallocate a large number of entries for the hash table. The number of entries in the hash table is also extremely dependent on the hash function. - Any situation where collision is common, you shouldn't use a hash table.

use BST if you need the sorted data to be available quickly for whatever reason; if it's a hash table you would have to sort the data first, taking Theta(n log n).

When the data is dispersed

**Q8 (Algorithms – Linked Lists, Recursion, Iteration)**Do it efficiently.

Traverse the list, for every element push it onto a stack then once the list has been traversed, pop the stack and construct the reversed linked list from there

reverse-list(list[]){ return []tsil; }

ham

in C, go to first element, then move onto the next, make a link to the previous, delete forward link. repeat until the end of the list. Done in O(n) time.

Have variable Node previous = null, and node next. Start from beginning. set next = node.next, node.next = previous (starts with null). previous = node. iterate to next node until next = null.

True

1,2,3,4

Iterate through each node, but make sure to remember the previous node. Set current node's next value to the previous node. For the first node in the original list, you need to set its next node to null.

p = head while (p != NULL) { // ie not at end of list p->next->next = p if (p == head) p->next = NULL p = p->next }

# first call of reverse is reverse(NULL,4) def reverse (node parent, child) if child.next == NULL child.next = parent else reverse(child,child.next) child.next = parent

**Q9 (Algorithms – Arrays, Searching, Recursion, Iteration)**Ask Francois Pitt. TM machines are good for this.

Create a hash table and then traverse the list for every element in the list: create a hash entry -> bool, start with 1 at first, then toggle its value whenever you see another key in the list at the end, only one key will have value of 1, others will have bool value of 0.

1. Look at the array 2. Count the elements in the array 3. Return elements where count mod 2 is 0

Strawberry

Delete first current integer and search array for its pair and delete the pair. keep deleting pairs until you delete a number that does not have a pair

Create boolean hashmap<int boolean>. Itererate through array. for each int update map as !boolValue. in the end itereate through map using keylist to find only value thats true.

Index out of bound

Step 1: Acquire array Step 2: ??? Step 3: Profit

Iterate through the entire array, store the number of times a number appears in a hash table, use the number as the index. Then, iterate through hash table to find the key corresponding to the odd value.

// hash\_table will store the counts for each array member for each number a[i] hash\_table[a[i]]++ for each k in hash\_table if hash\_table[k] % 2 != 0 return k

well you can just go create a map where the numbers in the array are the keys, and the value would be the number of occurrences of that key on the map. you go through the array and simply iterate over the map and check if any value % 2 == 1. and the answer would be the key to that value

**Q10 (Object Oriented Design)**

I would code it using good patterns.

Use the card class, a random generator class, and a player class

Big 2

Object class deck, object card. Each with their own properties such as suit, number, number of cards etc. the class deck would contain 52 card objects. You would have a game rules class that would be followed

implement classes card, gamestate. Card is basic card element. gamestate is generic current game status class. Any generic UI display classes needed also (the look of a card, etc). Then implement class Rule based on game type which contains victory condition checks, # of players, cards dealt per turn, etc.

Get a deck of cards

Pokemon.

For a blackjack game, my classes probably include Table, Dealer, Deck, Card, Spot, Player, ComputerPlayer. Where Deck has many cards, Table has spots, spots can be Player or Computer Player. The Table class keeps track of bets and player statistics, it also has a dealer which cares about dealing out cards. Some necessary methods used by the Player and ComputerPlayer include "hit", "bet", and "hold". Of course, a ComputerPlayer's implementation of the above methods are more complex and involve some basic AI.

- use some kind of card class which would vary the values depending on the game - i dont know all these card games, can’t assume everyone plays cards :P

There would be a Card class which has two values, the suit and the ordinal. there would be a Hand class, which contains a list of N Card objects. there would be an interface that had methods like draw so that the Card and Hand class could implement it

**Without PEY:**

**Q2 (CSC108 final mark range)**70-73: 183+: 4

**Q4 (Low level computing)**CPU access, Main memory access, Context switch, Disk seek

CPU access, Main memory access, Disk seek, Context switch (disk seek and context switch are debatable, it depends on how much data you read, hardware specs, OS design, etc.)

Thought this was a questionnaire not a test.

cpu access context switch disk seek main memory

CPU access, context switch, Disk seek, Main memory access

**Q5 (Algorithms – Sorting, Complexity)**Best case: O(nlogn) Worst case: O(n\*\*2) I don't really know why you would use it over Mergesort. I'm guessing that Quicksort gives empirically better average-case performance over typical input.

Best is n-log-n, worst is n^2. MergeSort requires recursive calls, which may be too much when working with a large data set, while QuickSort can be implemented in place.

I'll quote Albert Einstein - "Never memorize something that you can look up."

o/2 merge is better if the it's sorted data

O(log(n)) best case O(n^2) worst case When the data is quite large (large array)

**Q6 (System Design – Threads, Locks)**When progress can't occur because two processes are holding onto a resource that the other requires. We can prevent deadlock by putting a total order over all resources and requiring that processes acquire resources in order.

A deadlock is when a process blocks waiting on a resource held by another process, while that other process blocks waiting on a resource held by the original process. Neither of the processes can advance because they are not blocked, nor can they reach a later stage where they can release the resources they hold.

Design better code.

Deadlock is the case 2 programs have to access the same block in memory in the same time. We can prevent it by putting a lock in the place

When two processes are attempt to access a set of resources (say, 2 files or something), and they each have one, waiting forever for the other one to release control. Use some sort of a locking mechanism (like a mutex)

**Q7 (Data Structures – Trees, Hash Tables)**I'm sorry, I've forgotten everything I learned in 263. I know that hash tables have amortized O(1) lookups and insertions, and so they're good for contexts where that's important (say, representing an unordered set of elements that we're going to be looking up frequently). I guess a binary search tree is good for cases where ordering matters?

Binary tree is good when you need to retrieve a range of data, ie. all elements with key between 4 and 32, because it keeps everything sorted by key. Hash table is good for getting individual elements in constant time.

Use a hash table in autumn when the trees have no leaves.

if the data is huge

When the data set is small or when I can't get a good hash function for the data.

**Q8 (Algorithms – Linked Lists, Recursion, Iteration)**(define (reverse l) (foldl cons '() l)) Problem? (Human-readable description: make an empty list and, starting from the beginning of the input list, successively add elements to the front of the new list)

reverse(list) { previous = null; current = list; while(current) { next = current->next; current->next = previous; previous = current; current = next; } return previous; // head of reversed list }

Reverse the arrows: 4<-5<-2<-7

loop size begin temp=linkedlist1.first linkedlist.last.next=temp linkedlist.last=linkedlist.last.next linkedlist.firest.delete end

I would probably use recursion: func print\_list( node\* cur) print\_list(cur->next); print cur->data

**Q9 (Algorithms – Arrays, Searching, Recursion, Iteration)**def odd\_one\_out(l): odds = set() for ele in l: odds.add(ele) if ele not in odds else odds.remove(ele) return odds.pop() (Human-readable description: As we iterate through the list, keep a set of elements that have appeared an odd number of times so far. When we see a new element, we add it to the list if it's not there, or pop it if it is. At the end, if the list satisfies the precondition, there should be only one element in the set, so we pop it and return it.)

Suppose the integers are unsigned (cast them if they're not), and the largest one is MAX. Make a "bool test[MAX+1]" array, and initialize every entry to false. Then, for each x in the numbers array, test[x] = !test[x]. At the end, walk through the test array and if an entry is true, its index is an integer that appeared an odd number of times.

…

loop size begin if(mod(num,2)!=0) count++ end

Increment a counter for each number found then look at them all at the end

**Q10 (Object Oriented Design)**Some example classes with some of their plausible attributes and methods: - Game (attrs: players, deck, score) (methods: start, check\_victory\_conditions) - Player (attrs: name, hand) - Card (attrs: rank, suit, flipped) (methods: flip) - Deck (attrs: num\_cards, cards) (methods: shuffle) - Hand (attrs: cards) (methods: sort)

Class Card vars: suit, number, isTurned methods: getters/setters for the above vars Class Deck: vars: Collection<Card> cards methods: getters/setters for the above, shuffle, getCard(first, last, random), insertCard(first, last, random) Class Game: vars: Collection<Deck> decks methods: getters/setters for the above, addDeck, removeDeck

Card, Deck, Player, Game.

Card class - contained suit/value Deck class - bunch of cards, probably uses a stack under the covers, has a get\_top() method, shuffle() method, possibly a generate\_hand() method depending on the game Hand class - contains an array or vector or whatever of cards we have, has a print() method (or to\_string(), whatever is needed), has functions to modify the hand (add\_card(), discard\_card(), clear(), etc) Player class - contains money variable, cur\_hand variable of the Hand class

1. <http://www.surveymonkey.com> [↑](#footnote-ref-1)
2. Behavioral in this case, meaning non-technical questions, focused on their opinions and background. [↑](#footnote-ref-2)
3. Questions taken from *Cracking the Coding Interview* (2010), [www.glassdoor.com](http://www.glassdoor.com), [www.careercup.com](http://www.careercup.com) [↑](#footnote-ref-3)
4. *Cracking the Coding Interview*, Gayle Laakman (2010) [↑](#footnote-ref-4)
5. From this, it’s easy to see that employers often equate CS to be equal to Software Design. Web design or Artificial intelligence for example, are considered too specialized for common employers to ask. [↑](#footnote-ref-5)
6. One can graduate at UofT with a major in CS without knowing anything about Databases, QA Testing or Networking. [↑](#footnote-ref-6)
7. Large systems is the name for a group of algorithms that handle large amounts of data taking into account memory limits. [↑](#footnote-ref-7)
8. A student who had finished CSC148 should be able to handle these types of questions, so a fourth year student should be able to answer these in a few minutes each. [↑](#footnote-ref-8)