Хочу работать в Google: Read Me First!

О чем это все?

У меня давно была идея написать что-то вроде мануала для подготовки тем, кто готовится к интервью. Поскольку у меня нет свежего опыта подготовки (но зато есть много другого полезного опыта!) я сделала так: представила себе, что это мне надо готовиться к интервью, и попыталась как-то систематизировать план, который бы я составила для себя. Я не гарантирую, что это самый хороший план, но как минимум он вам даст основу для своего собственного плана.

Я выделила несколько основных категорий, которые встречаются на интервью:

- Algorithms и Coding;
- Concurrency и Design;
- Programming language и Fundamentals;
- Best practices и Experience.

Я приоритезировала категории следующим образом:

P0 > P1 > P2 > P3

Это значит, что если у вас совсем мало времени на подготовку, то вам надо сфокусироваться на **P0**, и затронуть остальные темы на поверхностном уровне. Даже в этом случае, потратьте на **P1** больше времени, чем на **P2**, а на **P2** больше, чем на **P3**.

Вторая призма, через которую можно смотреть на эти категории - это ваш опыт. Если вы только что выпустились из университета, то вам достаточно хорошо знать **P0**, умеренно **P1**, а все остальное - как получится. Никто от вас не ждет, что вы сможете глубоко рассуждать о том, что такое настоящий software engineer, и чем мастер отличается от не-мастера.

Зато если вы уже опытный профессионал (как минимум по резюме), то вы должны показать определенную глубину понимания, и даже собственную профессиональную философию. Вам нужно хорошо понимать свои сильные и слабые стороны, и иметь определенную профессиональную харизму. Идеальный вариант - нарабатывать ее, параллельно работая с коучем, ментором и терапевтом. Но это не делается за пару месяцев, тут предполагается долгосрочная работа. Плюс, не у всех есть возможность, время или желание для более глубокой работы над собой на многих уровнях. В этом случае вы можете просто почитать хорошие книги.

Ну и самый важный пункт - каждый следующий уровень строится поверх предыдущего. Если вы не умеете писать хороший код, то никого не будет волновать, что вы - гуру дизайна и у вас очень глубокие профессиональные ценности.

Про кучу ссылок

Внизу я приведу много ссылок на книги и материалы, которые мне кажутся интересными и полезными. Но это не значит, что вам надо прочитать все материалы - так у вас никакого времени на подготовку не хватит :)

Поэтому ориентируйтесь на свои цели и приоритеты, на свой стартовый уровень, и на количество времени, которое у вас есть. Скорее всего вам не надо читать 10 книг по алгоритмам. Вместо этого попробуйте те, которые вам покажутся наиболее подходящими. Остановитесь на одной, которая вам подойдет лучше всего. Выберите вторую "запасную", если вам не очень понравится, как какая-то тема объясняется в вашей "главной" книге.

У всех разные способы обучения и восприятия информации, и если какая-то книга для меня самая лучшая, то это не значит, что так же будет и для вас. Поэтому я не будут давать советов в ключе "Обязательно читайте именно эту книгу". Потратьте немного времени на то, чтобы выбрать ресурсы именно под себя.

И еще, я даю ссылки на книги на Амазоне, но многие из них доступны в интернете или на youtube в формате аудио бесплатно. Я не хочу давать ссылки для скачивания, так как книги часто удаляют и ссылка станет битой, а у меня нет времени их отслеживать и править. Но вы легко найдете большинство книг через Google или, в крайнем случае, на торрентах.

Перед тем, как вы начнете готовиться

Эффективная подготовка предполагает несколько вещей. Некоторые из них довольно очевидны и общеизвестны - вначале нужно составить представление о том, что вообще нужно изучить, примерно составить понимание о фронте работ и, может, даже написать какой-нибудь примерный план подготовки, который укладывается в отведенное вами время.

Тут я только хочу сказать, что обязательно - обязательно!!!! - поставьте себе довольно жесткий дедлайн. Идеально - подайтесь в компанию из вашего списка (про компании и список ниже), и договоритесь, что интервью будет не сразу, а, скажем, через месяц-два-три. На крайний случай жестко запланируйте когда вы начнете посылать резюме. Ни в коем случае не готовьтесь по принципу "когда я буду все знать и буду уверен на 100%, тогда и подамся" - так вы не пошлете никогда, да и основательно готовиться вряд ли будете.

Но есть и еще один важный фактор, который нужно обязательно упомянуть. А именно - сам процесс подготовки, и как готовиться эффективнее всего. К счастью для нас, тема эффективного обучения довольно хорошо изучена, и нам не надо изобретать колесо. Поэтому перед тем, как вы начнете готовиться, обязательно изучите, как именно готовиться эффективнее всего. Этим вы сэкономите себе кучу времени и сможете подготовиться намного лучше.

Ваша задача тут - освоить погружение в задачу и вхождение в состояние потока. Приготовьтесь к тому, что вначале будет сложно, но где-то за месяц упорных попыток вы освоите этот навык и будете себе очень за это благодарны.

Вторая задача - это войти в нужное эмоциональное состояние. Если быть более точным - то вам нужно поверить в себя, в свою цель и в то, что все достижимо. Вам нужно четко понимать, зачем вы собираетесь потратить большое количество времени на подготовку и подвергнуть себя некоторому стрессу.

Запишите эти причины на бумаге и читайте их каждый раз перед началом подготовки. Поставьте себе 30 days challenge - например, каждый день решать минимум 5 задач на алгоритмы (или сколько в вашей ситуации возможно). Постарайтесь измерять свой прогресс. Примите это как судьбоносный момент, поворот вашей судьбы, когда все должно поменяться на 180 градусов в лучшую сторону, и сейчас вам для этого надо поработать.

Помните, что подготовка и получение оффера от компании мечты - это марафон. Это не о том, насколько вы умный, какой у вас IQ, и сколько у вас опыта решения олимпиадных задач. Это о том, насколько эта цель поселилась внутри вас, о внутренней мотивации. И о том, насколько хорошо вы подготовились.

Если у вас "низкий старт" - мало опыта, не очень сильные знания - то это значит только то, что придется хорошенько поработать и многое выучить. Это будет нелегко, вам будет много раз хотеться все бросить. Вы узнаете не только про сортировки, написание кода и мьютексы, но и в какой-то мере заглянете внутрь себя, в свои желания, опасения, усталость, раздражение. Может быть, вы даже будете плакать, считая себя безнадежной балдой. Это пройдет, и в самом конце останется очень ценный опыт. И, возможно, отличный оффер от фирмы мечты.

Но это все - если вы выдержите. А для того, чтобы выдержать, вам нужно по-настоящему этого хотеть, хотеть изменить свою жизнь и бросить себе вызов. Поэтому загляните в себя и посмотрите - готовы ли вы?

Книги

- "Peak: Secrets from the New Science of Expertise"
- "Deep Work: Rules for Focused Success in a Distracted World"
- "So Good They Can't Ignore You: Why Skills Trump Passion in the Quest for Work You Love"
- "Accelerated Learning Techniques for Students: Learn More in Less Time"
- "The Power of Habit: Why We Do What We Do in Life and Business"

Ссылки и статьи

- 10 эффективных стратегий повышения продуктивности
- Эффективные методы подготовки к экзаменам
- What little habits made you a better software engineer

Алгоритмы

Приоритет: **P0**

Что нужно знать

Тут учтите, что если я пишу, что надо знать "Linked lists", например, то я не буду перечислять каждый алгоритм, связанный с этой темой. Но подразумевается, что вам надо самостоятельно изучить, какие у него есть операции и потренироваться писать удаление, вставку и другие важные вещи.

Вторая важная вещь - если у вас мало времени, и вы не успеваете покрыть все темы, то выбирайте те, которые вам встретятся с бОльшей вероятностью. Например, лучше изучить графы и деревья, чем алгоритмы шифрования. Но если у вас есть возможность, постарайтесь изучить все, хотя бы на уровне понимания Wikipedia.

- Базовые вещи
    - Сложность алгоритмов - что такое и как считается? Сложность временная и пространственная
    - Работа с битами - таблицы AND, OR, XOR. Сюда же - двоичная запись, и степени двойки
    - Основы теории вероятности
    - Основы комбинаторики и подсчет количества разных комбинаций
    - Основы евклидовой геометрии - например, как проверить лежат ли точки на одной прямой, по одну или разные стороны прямой
    - Основы криптографии
    - Основы компьютерной безопасности
- Основные структуры данных
    - Strings
        - ASCII, Unicode
        - Как строки реализованы в вашем языке программирования (например, есть ли у них максимальная длина)
        - Поиск подстрок (например алгоритм Рабина-Карпа)
        - Регулярные выражения
    - Arrays
        - Детали реализации в вашем языке программирования. Например, для C++ нужно знать реализацию с помощью указателей, и вектор. Для вектора тоже нужно знать, например, то, что он периодически делает resize, и другие похожие детали.
    - Linked lists
        - Singly linked list
        - Doubly linked list
        - Less common types are listed, for example, here
    - Stacks and Queues
    - Trees
        - DFS, BFS

        - Adding and removing elements
        - Less common tree types (e.g., red black trees, B-trees) - what are they, how they differ from the binary trees, basic complexities, and how they are used. No need to know all the rotations in the RB-tree, for example.
        - Tries
    - Heaps
        - Heap sort
        - Using heaps for tracking top-K
        - Allocating elements on a heap vs on a stack - what does it mean?
    - Graphs
        - DFS, BFS
        - Topological search
        - Shortest path
    - Hash
        - Hash functions
        - Universal hash
- Алгоритмы
    - Sorting
        - Especially make sure you know heapsort, mergesort and quicksort.
    - Searching
        - Binary search
        - Searching in linked lists, arrays, trees, graphs, dictionaries...
    - Dynamic programming
    - Greedy algorithms
    - Recursion
- Другие важные вещи
    - Master theorem
    - NP-complete problems
    - Discrete math (например).

Примечания

1. Вам нужно понимать trade offs каждой задачи. Например, поиск в HashMap обычно быстрее сортировки, но для него нужна дополнительная память. Так что в случаях, когда скорость не критична, а вот память да, может быть лучше отсортировать.
2. Для более редких структур данных - e.g., trie, B-tree, red black tree - необязательно помнить все ротации для удаления и вставки элемента, но нужно понимать как эти структуры работают, и где они используются.
3. Вам нужно понимать ограничения каждой задачи. Например, поиск элемента в массиве невозможно сделать меньше, чем за время O(n). Или что в общем случае сложность сортировки, которая использует сравнение элементов, не может быть меньше, чем O(n log n).
4. Что такое NP полнота. Это тип задач, которые можно решить только полным перебором всех вариантов. Знать, какие задачи - NP-полные.
5. Изучите стандартные приемы для решения задач. Например, heap размера K можно использовать для top-K большого массива для линейной сложности. Помните, что эти приемы отрабатываются решением задач.

Еще хороший ответ на вопрос "Что нужно знать" на Stackexchange: Which algorithms/data structures should I “recognize” and know by name?

Советы

1. Когда изучаете алгоритмы, задавайте себе вопросы сами. Это самый простой способ углубить свои знания. Например, когда сложность алгоритма log(n), то по какому основанию log и почему? Или что лучше купить, чтобы ускорить ваш код - море оперативной памяти или более шустрый процессор? Почему?

Книги по алгоритмам

- Introduction to Algorithms
- The Algorithm Design Manual
- Discrete Mathematics for Computing
- Algorithms (4th edition)
- Essential Algorithms by Rod Stephens

Ссылки и статьи

- Big-O Cheat Sheet - сложность всех самых важных алгоритмов в одном месте. Выучить наизусть! Перевод статьи на Хабрахабре
- Algorithms: Design and Analysis, Part 1 (Stanford), Part 2 - курсы алгоритмов Стэнфорда
- Algorithms, Part I (Princeton) - курс из Принстона
- Analysis of Algorithms (Princeton) - еще один курс из Принстона
- http://visualgo.net - множество визуализаций алгоритмов и структур данных
- Intro to Algorithms (Udacity)
- Видео от автора "Cracking the coding interview":
    - How Companies Evaluate Technical Interviews
    - How to Approach Behavioral Questions
    - 7 Steps to Solve Algorithm Problems
    - 3 Algorithm Strategies
- Mastering the Software Engineering Interview на Coursera

Код

Приоритет: **P0**

Я регулярно вижу кандидатов на hiring committee, которые умеют решать задачи, хорошо знают все алгоритмы, но не могут написать к ним код. Удивительно, но факт! Такой кандидат находит самое оптимальное решение, может хорошо рассказать почему оно оптимальное, и вообще, блестяще справляется с задачей в плоскости теории. Но когда дело доходит до того, чтобы написать к задаче код, у него случается затык, и он пишет долго, с ошибками, и не успевает до конца интервью. Таким кандидатам мы отказываем со словами "Они, конечно, умные и с потенциалом, но сразу видно, что нужно нарабатывать навык написания кода".

В общем, не будьте как такой кандидат. Обязательно тренируйтесь писать код к задачам. Обязательно! Вы должны уметь написать бинарный поиск или там быструю сортировку не приходя в сознание, если я вас разбужу посреди ночи!

Что нужно уметь

1. Быстро и без ошибок написать код стандартного алгоритма (бинарный поиск, сортировка, поиск в глубину и в ширину, удаление элемента из связного списка...)
2. Быстро и без ошибок написать код вашего решения алгоритмической задачи на интервью. Это значит, что если вы комбинируете несколько структур данных, например (ну, мало ли :), храните узлы графа в связном списке и что-то там с ними делаете в процессе поиска, то вы должны совершенно спокойно написать и код для графа, и код для связного списка, и код для поиска.
3. Писать код, который компилируется (по возможности избегайте псевдокода).
4. Писать аккуратный, элегантный код, который хорошо читается. Чем меньше строк, тем лучше, но не в ущерб читаемости. Если у вас есть несколько вариантов - например, более эффективный код, который, однако, хуже читается, то обсудите эти варианты с интервьюером.
5. Писать код, который работает с corner cases: null, пустые строки, ноль, отрицательные числа...
6. Писать код, который не просто работает, а работает эффективно. Например для C++, если вам надо передать строку в качестве параметра в функцию, правильно будет написать `f(const string& s)`, а не `f(string s)`.
7. Тестировать свой код без напоминания и находить все ошибки самостоятельно. Лучше сразу начинать с тест кейсов, хотя бы в простейшем виде: то есть вам поставили задачу, - напишите пример (или несколько) входных и выходных данных, прежде чем написать, собственно, код. Таким образом вы не просто подготавливаете себе почву для последующей отладки своего кода, но и убедитесь, что вы верно поняли поставленную интервьюером задачу.
8. При тестировании постарайтесь покрыть весь код - чтобы code coverage был 100%.

Что нужно знать

1. Стандартные библиотеки вашего языка программирования. Например, вы должны знать как удалить элемент из середины массива или найти элементы множества, которых нет в другом множестве без подглядывания в документацию. Для C++ это значит знать довольно много вещей из STL и std.

   Александр: *"Для лучшего понимания, как устроен STL и какой контейнер оптимально использовать в той или иной ситуации, мне помогла практика реализации "своего STL". Идея такая: реализовать несколько контейнеров STL односвязные и двусвязные списки, хеш мапы и др. Так же для контейнеров реализовать основные алгоритмы: вставка, удаление, сортировка, копирование, итераторы и др. После этого посмотреть, как устроен C++ STL внутри, и пробовать оптимизировать свою реализацию.*
   P.S.: Скорей всего это будет полезно если есть много времени на подготовку.
   P.P.S.: Не использовать "свой" STL в работе над реальным продуктом :)"

Советы

1. Учтите, что самое главное тут - практика, а не чтение книг и статей. В контексте процесса подготовки к интервью вы можете написать код решения, а потом посмотреть чужие решения этой же задачи и сравнить, насколько лучшие из них лучше вашего и чего не хватает вашему решению. Закройте чужое решение и перепишите ваше, чтобы оно стало более "элегантным". Делайте это с каждой задачей до тех пор, пока ваше решение не будет выглядеть, как одно из лучших на фоне остальных. Переписывать задачу используя чужое решение лучше на следующий день, тогда вы будете точно уверены в том, что вы действительно поняли, как работает более быстрое решение, а не просто скопировали из головы увиденный код.
2. Проговаривайте ваш ход мыслей. Интервьюер не знает, что происходит в вашей голове, если вы пять минут молча смотрите на доску. Может быть, вы уже продумали и отбросили 5 разных подходов, но интервьюер может подумать, что в голове вашей пусто. Поэтому надо говорить. Для большинства людей это неестественно. Практикуйтесь с товарищем.
3. Если у вас не получается сразу придумать оптимальное решение задачи, начните с brute force решения. Скажите, что вы понимаете, что оно не самое лучшее, но хотите с него начать, чтобы в последующем улучшить и оптимизировать. Возможно, интервьювер вас “подтолкнет” в нужном направлении, но если даже и нет, то лучше делать хоть что-нибудь, чем смотреть на чистую доску и выглядеть полным идиотом. =)
4. Для написания кода вам, скорее всего, не дадут IDE. А без Code Insight писать гораздо сложнее. Вас могут попросить написать код на доске маркером, на бумаге карандашом, или как вариант, если это удаленный phonescreen, в документе Google Doc. Практикуйтесь писать код на бумаге с товарищем, и пусть он вам потом даст отзыв, как это выглядело.
5. Никто не будет требовать от вас помнить все методы всех библиотечных классов и все параметры всех методов, но вы должны хотя бы в общих чертах помнить синтаксис. Имейте в виду, что писать маркером на доске в некотором отношении даже сложнее, чем ручкой на бумаге - потому что хотя у вас и есть возможность стереть свой код, маркеры обычно очень толстые и на досках быстро заканчивается место. (В некоторых источниках поэтому даже советуют приносить на собеседование свои маркеры, потоньше. Только не приносите несмываемые, а то в памяти интервьювера вы точно останетесь надолго, но не в самом лучшем контексте! =)

Совет практиковаться не одному, а с кем-то - один из самых трудно выполнимых, и соответственно, один из наиболее часто игнорируемых. Беда в том, что он одновременно и один из наиболее значительных. Поверьте, решать задачу самостоятельно и решать ее под бдительным оком интервьювера, в условиях стресса и ограниченного времени - это две большие разницы. Если вам совсем уж некого попросить и не с кем скооперироваться для оценки вашего интервью - на худой конец, запишите себя на камеру. Потом просмотрите и оцените. (Повторюсь, что это - вариант действительно на безрыбье, то есть в том случае, когда вам действительно совсем уж некого попросить, либо вы настолько стесняетесь, что прежде чем кого-то попросить, вам нужно потренироваться в одиночку.)

Книги

- "Clean Code: A Handbook of Agile Software Craftsmanship"

- "Code Craft: The Practice of Writing Excellent Code"
- "The Clean Coder: A Code of Conduct for Professional Programmers (Robert C. Martin Series)"
- "Beautiful Code: Leading Programmers Explain How They Think"
- Effective Java (2nd Edition) 2nd Edition by Joshua Bloch

Concurrency & Multithreading

Приоритет: **P1**

Многопоточность заслуживает своей отдельной категории. Если прямых вопросов на эту тему - вроде "А чем мьютекс отличается от семафора?" - обычно не задают, то все равно потоки, и все с ними связанное вылазит во многих областях, от алгоритмов до дизайна систем. Иногда интервьюеры после того как вы написали решение задачи могут спросить "How would you make your code thread safe?". Поэтому не поленитесь и хорошенько изучите эту тему.

Что нужно знать

1. Работа с потоками в вашем языке программирования - библиотеки, классы и их особенности.
2. Основные понятия - потоки, процессы, мьютексы, context switch, deadlock, race condition.
3. Синхронизация данных между потоками - какие особенности, на что нужно обратить внимание.
4. Тестирование многопоточных программ, основные признаки багов связанных с многопоточностью.
5. Попробуйте написать свой мьютекс-класс и/или shared pointer.

Книги

- "C++ Concurrency in Action: Practical Multithreading"
- "Java Concurrency in Practice"
- The Art of Multiprocessor Programming

Статьи

- Concurrency and multithreading interview questions
- Processes and Threads from A. Tannenbaum (один из моих любимых авторов в CS)
- Threads and concurrency

Design, large-scale systems

Приоритет: **P1**

Вопросы по дизайну крупных распределенных систем. Основная "проблема" таких вопросов в том, что у них часто нет "правильного" ответа. Они специально заточены под то, чтобы найти слабые

звенья в ваших знаниях, и готовиться к подобным вопросам - это, в какой-то мере пытаться охватить необъятное, особенно если у вас нет опыта работы с большими системами.

Такие вопросы могут начинаться вполне безобидно. Например: “У вас есть приложение, которое вы написали в одно лицо на своем компьютере, код хранится просто на жестком диске, и ваш компьютер хранит все данные и само приложение. Неожиданно ваше приложение начали покупать и использовать другие люди, и у вас уже десять пользователей. Что вам нужно сделать, чтобы обеспечить работоспособность программы для пользователей? Что меняется, когда у вас сто пользователей?... Та-ак, а теперь у вас уже тысяча пользователей, и вы заработали 10 тысяч долларов. Как меняется ваша архитектура и (или) инфраструктура? А теперь у вас 10 тысяч пользователей и вы заработали 100 тысяч долларов... А теперь у вас 100 тысяч пользователей и вы заработали 1 млн долларов...”

Имейте в виду, что если у вас появится соблазн сказать “я продаю бизнес, забираю деньги и линяю под пальму на Канары наслаждаться жизнью”, то это вам не поможет, отвечать на вопросы все равно придется =). Сюда же попадают вопросы по оптимизации performance, по устойчивости приложения, CAP theorem и т.д.

Хорошая новость для нас - это то, что не обязательно тратить годы на изучение всех деталей сетевых протоколов, архитектуры памяти итп.\. Есть довольно ограниченный (хотя и немаленький) набор приемов и подходов, которые вам достаточно изучить:

1. **Scaling**. Тут важно понимать, что затык чаще всего возникает в реляционной БД, как можно этого избежать при помощи sharding или используя различные NoSQL, типа MongoDB.
2. На уровне сервисов можно поговорить про **load balancing** и некоторые его стратегии. Любая система должна загружать сервера более-менее равномерно.
3.
4. **Fail-safe** (aka eliminate single point of failure, aka consistency). Любая нарисованная вами на доске архитектура для решения вопроса должна позволять любой своей части выйти из строя в любое время. Тут можно дополнительно рассказать про Netflix chaos monkey и как это круто =).
5. **Replication**. Любой stateful сервис, любая часть системы должна позволять перезагрузку и выход из строя в любой момент. Для этого нужно, чтобы данные были на это время были доступны где-то еще.
6. **Master election**. Если вы можете подробно рассказать про алгоритм выбора лидера при отказах и восстановлениях, например Raft или Paxos, то считайте, что прошли интервью.
7. **Performance** (обычно про caching и load balancing), рассказать несколько стратегий load balancing и про разные уровни кэша, рассказать про Memcached или Redis, и примеры их использования на практике, рассказать про географический load balancing, синхронизацию данных между несколькими датацентрами.
8. Очень важной частью распределенных систем является **обмен сообщениями и logging**, поэтому практическое знакомство с RabbitMQ, Amazon SQS/SNS, Kafka очень поможет.
9. Также помните про **мониторинг системы**. Какие данные вы хотите измерять, и какие критерии "здоровья" системы вы выберете?
10. **Deployment**. Контейнеры типа docker, управление ими при помощи mesos, проблемы deployment распределенных систем тоже могут пригодиться.

Книги

- Distributed Systems: Principles and Paradigms

Ссылки и статьи

- Distributed systems basics
- Scalable Web Architecture and Distributed Systems
- System Design for Tech Interviews - полезно тем, что там есть пример вопроса, ссылка на лекцию из Гарварда и ссылки на то, как сделаны дизайны настоящих продуктов. Этого материала точно должно хватить тем, у кого 5 и меньше лет опыта. Для более опытных кандидатов надо копать поглубже, но это тоже хороший старт.
- System Design на Interviewbit
- Вопросы на архитектуру систем (часть 1, часть 2)
- http://highscalability.com/
- https://github.com/checkcheckzz/system-design-interview
- http://blog.gainlo.co/index.php/category/system-design-interview-questions/ - тоже примеры вопросов. Довольно простенько, но может быть, на начальном уровне тоже будет полезно.

OOP & Design patterns

Приоритет: **P2**

В чистом виде сейчас такие вопросы не очень часто встречаются. Вряд ли у вас спросят, что такое инкапсуляция, полиморфизм или наследование. Однако же подобные вопросы вам могут встретиться как сопутствующие при дизайне систем. Например, если вас попросят спроектировать схему классов либо же сущностей данных для какой-либо системы, вас могут спросить, почему вы выбрали не наследование, а агрегацию (композицию). Или наоборот. И что вы вообще предпочитаете. И почему вы предпочитаете именно это. Объясняется ли это особенностями вашего языка программирования (например, java исключает множественное наследование)? Чем-то другим? Чем именно?

Что рекомендуется знать:

- Singleton. Самый известный паттерн, который тем не менее никто не отменял. Где встречается в реальной жизни, то есть в живом коде.
- Active Record. Chain of Responsibility. Data Transfer Object. Adapter. Decorator. Factory. Factory Method. Другие стандартные шаблоны программирования.
- Дизайн данных. В случае реляционных данных - table for class, table for all classes, table for each concrete class, что и почему. Реляционные БД все еще занимают очень значительное место.

Книги

- Patterns of Enterprise Application Architecture by Martin Fowler

Your programming language of choice

Приоритет: **P2**

Язык программирования - это ваш рабочий инструмент. И от вас, как от профессионала, ожидается, что вы будете не просто знать ограниченный набор функциональности, которым вы пользуетесь каждый день, но и более глубоко понимать ваш язык. Причем если новичку могут простить отсутствие глубокого понимания, например, Java garbage collection, то человеку, который пишет на Java много лет - вряд ли. Даже если по работе он с ним не сталкивается (что обычно правда, ведь все работает и так, смысл глубоко вникать?), то это создает впечатление человека, который особо не интересуется и относится без энтузиазма к своей профессии. Это не то впечатление, которое вы хотите произвести на работодателя.

Тут стоит учесть, что вопросы из этой части вам, скорее всего, начнут задавать, если у вас в резюме 5+ лет солидного опыта на этом конкретном языке. От вас будут ожидать, что раз уж вы на нем так много пишете, то вы должны неплохо знать ins & outs.

Если у вас 1-2 года опыта, вы недавно выпустились из университета и все это время писали, скажем, на Java, то не беспокойтесь сильно о тонкостях. Просто удостоверьтесь, что вы хорошо понимаете основы и принципы вашего языка.

Что нужно знать

1. Best practices вашего языка. Обязательно почитайте Effective Java или Effective C++, если это ваши языки. Погуглите "best practices <your_language>", уверена, что вы найдете достаточно хорошего материала.
2. Worst practices вашего языка. Например вещи, который будут работать, но будут работать медленно, или криво, или с риском завалиться при первой же возможности. Если погуглить "C++ avoid" или "C++ pitfalls", то вылазит много разных интересных вещей (например).
3. Особенности вашего языка - есть ли у него garbage collector, как он работает с памятью, как в нем реализованы виртуальные функции, чем он глобально отличается от других языков программирования, в каких ситуациях ваш язык лучше других и почему?
4. Как в вашем языке программирования делать разные полезные штуки, если это возможно. Например immutable класс, или класс, который нельзя скопировать.
5. Работа с памятью. Stack vs heap. malloc vs new (для C++).

Советы

1. Прочитайте Google style guide для вашего языка программирования. Там много полезной информации.
2. Прочитайте минимум одну книгу о вашем языке программирования ("Effective X" подойдет).

3. Поищите puzzles о вашем языке программирования. Когда написана маленькая программа и от читателя требуется понять, что программа выдаст. Например, для C++. Гуглится по "C++ quiz". Для Java.
4. Поищите interview questions по вашему языку программирования. Нормальные компании обычно не спрашивают вопросов типа "а что такое volatile в C++", но на всякий случай посмотрите их тоже.
5. Почитайте Documentation на StackOverflow для вашего языка.

Общий кругозор

Приоритет: **P2**

Большинство перечисленных в этой главе вещей нужно знать на общем уровне - иметь хорошее представление, но не обязательно сильно вникать в детали. Быть в состоянии ответить на базовые вопросы, вроде "Вот я написал программу в C++, что происходит дальше?". Тут надо понимать и про компиляторы, и про машинный код, и про инструкции процессора...

Исключение - если с какой-то из этих областей вы работали продолжительное время. В этом случае от вас будут ожидать более солидных знаний.

Что нужно знать

1. Что в реальности вещи работают не совсем так, как в теории. Например, если вы хотите хранить в памяти HashMap, то вы храните не только значения, но и саму структуру данных. Это называется overhead. Обычно это один или два дополнительных указателя для узла. Если значений очень много, и все они, например, числа, то overhead может оказаться довольно существенным.
2. Архитектура компьютера - память, процессоры, регистры, шины, диск, инструкции,...
3. Операционные системы - что такое поток и процесс (и чем они отличаются), какие в ОС есть компоненты, как процессор взаимодействует с памятью, откуда операционная система загружается при включении компьютера и т.п.
4. Сети и как работает Интернет.
5. Компиляторы в общих чертах.
6. Хранение данных, RAID.
7. Компьютерная безопасность, криптография.
8. Базы данных. NoSQL vs RDBMS (relational databases), что такое индекс, что такое key и attribute.
9. Регулярные выражения и автоматы/грамматики.

Книги

- Книги Andrew Tanenbaum. Они большие, но очень здорово написаны. Поэтому читайте скорее для общего развития.

- Список книг, по мнению участников Stackoverflow, которые нужно почитать каждому хорошему программисту. Список огромен, не стоит читать каждую книгу оттуда, но его можно рассматривать как мануал для подбора хороших книг.
- Рекомендуемая литература от Google - Google разрешил опубликовать список литературы, которые мы рекомендуем к прочтению сотрудникам.

Ссылки и статьи

- "What every computer science major should know". Очень большой список, не воспринимайте дословно. Там зато есть хорошие ссылки на материалы.
- “Programmer Competency Matrix” - огромный список знаний, разделенный по уровням.
- "What should every programmer know about X". Серия статей на Quora на разные темы.
- "Latency numbers every programmer should know"
- The Hardware/Software Interface (University of Washington) - курс по архитектуре компьютера

Нетехнические вопросы на интервью

Best practices, seniority, philosophy of professional growth.

Приоритет: **P3**

Что надо знать

1. Как отвечать на нетехнические вопросы - долгосрочные карьерные планы, слабые и сильные стороны, сложные ситуации на работе, примеры лидерства и т.п. Интервьюерам важно узнать вас как личность и понять, хотят ли они работать с вами в одной команде. Удобно подготовить ответы в виде историй в стиле STAR. Вам нужно четко знать, почему вы хотите работать в [[company]].
2. Какие вопросы вы зададите интервьюеру. Это стратегическая задача, кстати - 90% кандидатов просят интервьюера рассказать о его работе. Вы можете спросить то же, что и остальные 90%, а можете что-нибудь другое.
3. Подходы к тестированию кода - unit testing, regression testing, compatibility testing, canary testing.
4. Основные подходы к созданию проектов - scrum, agile.
5. Английский язык. Многие думают, что это не суть важно, если ты кодер-”звезда”. Важно. Вам на этом самом английском языке придется произвести на своих англоговорящих интервьюеров впечатление звезды. И это достаточно трудно сделать, если язык у вас на уровне десятиклассника.
6. Собственный уровень знаний и на какую позицию вы можете претендовать. Не пытайтесь показаться лучше, чем вы есть. Это позже выйдет боком.
7. Ваша собственная философия. Например, что для вас важнее - stability или velocity? То есть вы можете более основательно тестировать и выпускать более стабильный продукт. Либо тестировать несколько меньше, но зато быть более гибким и иметь возможность обновляться быстрее. Тут нет правильного ответа, есть разные предпочтения разных людей.

   Ларрр: *Заодно расскажу про нашего бывшего тимлида, одного из лучших инженеров, что я*

знаю. Он на любой запрос новой функциональности по умолчанию говорил "Нет". Его можно было убедить, но надо было постараться, и именно убеждать. Философия его была такова, что чем меньше в системе функциональности, тем проще ее поддерживать, оптимизировать и изменять. Поэтому он очень неохотно соглашался на новую функциональность, даже если его уговаривал хор продуктовых менеджеров, и предпочитал сосредоточиться на том, чтобы уметь делать меньше, но зато отлично.

Книги

- "On Managing Yourself"
- "How to answer the 64 toughest interview questions"
- "The Pragmatic Programmer: From Journeyman to Master"
- "Code Complete: A Practical Handbook of Software Construction, Second Edition"
- "The Mythical Man-Month: Essays on Software Engineering"
- "Soft Skills: The software developer's life manual"

Ссылки и статьи

- "What are the best questions to ask a potential employer at a job interview".

Профессиональный опыт

Приоритет: **P3**

Что нужно знать

1. Нужно хорошо знать технологии, с которыми вы много работали, судя по вашему резюме. Например, моя команда хоть сама напрямую и не пишет, но близко работает с базами данных и компиляторами. Поэтому если я буду готовиться к интервью, я обязательно включу базы данных и компиляторы в свою программу подготовки.
2. Детали больших проектов, с которыми вы работали. Проблемы и их решения. Например, если ваш проект работает с очень большим количеством запросов, то я могу у вас на интервью спросить "How did you implement throttling/rate limiting?" (пример).
3. Причем вы должны иметь хорошее представление не только о своих непосредственных проектах, но и о том, как работает проект в целом, какие в нем есть компоненты, какой у них дизайн и почему и т.п. С тем же вопросом про throttling - даже если вы не писали эту компоненту, я все равно буду ждать, что вы будете знать как она работает, раз уж она была на вашем проекте, пускай и не на супер-глубоком уровне.

Другие важные аспекты

Компании

Главная идея - не подавайтесь только в компанию вашей мечты. Подавайтесь сразу в несколько компаний, причем начинайте интервьюироваться с теми, которые не находятся в вашем топе. Почему?

1. Вам нужно время на "разогрев". С существенной вероятностью вы провалите первые интервью - либо будете сильно нервничать, либо не учтете какие-то вещи в вашей подготовке... И лучше провалить эти интервью в компаниях, куда вам не так, чтобы прямо сильно хочется.

2. Вам нужно протестировать уровень вашей подготовки и вменяемость вашего плана. Например, вы решили прийти на интервью в смокинге с бабочкой, чтобы выделиться из серой массы других кандидатов. Интервьюеры смотрели на вас как на полоумного. Значит это был плохой план, в следующую компанию надо надеть джинсы.

3. Если вы получите оффер в начальных компаниях, то, возможно, они сделают вам хороший оффер (особенно, если вы отлично пройдете интервью и дадите им знать, что вы планируете интервьюироваться и с другими компаниями). Это поднимет планку офферов в последующих кампаниях и поможет вам получить более выгодные предложения.

Тут только важно учесть такой момент, что после того, как компания вам сделала оффер, у вас будет что-то около недели или двух, чтобы его принять или отклонить. Поэтому постарайтесь сделать так, чтобы все офферы к вам пришли относительно в одно и то же время.

Как составить список компаний? Для начала, я бы посоветовала интервьюироваться только с теми компаниями, которые вам потенциально могут быть интересны (при наличии хорошего оффера). Если вы всю жизнь свято верите, что "M$ MUST DIE!!!", то Бога ради, не надо подаваться в Майкрософт! Не тратьте ни свое, ни их время. Считайте это профессиональным этикетом.

Дальше выбирайте несколько (3-5 хватит, если вы хорошо подготовились), которые вам симпатичны и в которых вы в принципе готовы работать, если все нормально сложится.

Обсуждение зарплаты

- Ten Rules for negotiating a job offer
- Salary Negotiations with Prospective Employers
- 5 Tips for Negotiating Your New Job Salary
- How to Negotiate Salary: 5 Expert Tips
- How to Negotiate A Starting Salary for a New Job
- How to Bomb Your Offer Negotiation

Резюме, рефералы

Выбор компаний - это только начало. Теперь вам надо составить резюме и, собственно, податься.

1. Составьте хорошее резюме [раз, два, три]. Я не претендую на идеал, но тут можно посмотреть мое резюме.

2. Найдите рефералов в компаниях, куда вы хотите подаваться. В большинстве компаний реферал не обязан знать вас лично, ему достаточно написать небольшое обоснование того, почему он считает, что вас надо рассмотреть. Этим обоснованием может быть "Он мне сам написал, я посмотрел его резюме, и у него интересный опыт в языках программирования с которыми работает Google. К тому же он окончил университет Зажопинска, а это довольно известный университет в нашей стране, они выпускают сильные кадры". Заметьте, ничего про то, что он вас лично знает. Искать реферала можно через знакомых, на linkedin, у меня в блоге…

3. В дополнение к предыдущему пункту, подготовьте небольшой параграф, который поможет рефералу принять решение вас порекомендовать. Около 5 пунктов, highlights of your experience. Примеры:
   "Results of my Master thesis were published in top Russian scientific magazines"
   "I grew from an intern role to being a TL in four years"
   "I have a solid knowledge in C++ (5+ years), and also can code in Java"
   "I launched a solid number of large and important projects in my company. I've always delivered on schedule and my projects didn't have any production issues afterwards."
   "I won XYZ award"
   "*I've earned top GPA at my university*"
   "I'm actively contributing to open source community"

4. Если вам кажется, что вам и написать про себя особо нечего, то, скорее всего, это не так. Достижения есть почти у всех, просто, по моим наблюдениям, у постсоветских граждан есть тенденция принижать свои достижения, думая "Это фигня, любой бы такого достиг на моем месте". В 100% случаев, когда мы вместе с ребятами копаемся в их резюме, оказывается, что достижений полно, намного больше пяти пунктов. Просто почему-то они не считают это достижениями. Impostor syndrome во всей красе.

5. Составьте письма для потенциальных рефералов. Обязательно включите: резюме в приложении; небольшой рассказ о себе; пункты о том, почему вы считаете, что вы хороший кандидат и вас надо рассмотреть (см. примеры выше).

6. Если найти реферала не получилось, попробуйте найти рекрутеров на LinkedIn и написать им напрямую. Через сайт подавайтесь в последнюю очередь, когда все варианты исчерпаны. Ну или если вам не очень сильно хочется работать в этой компании, и вы легко переживете, если вам не ответят.

7. Если вас позовут на интервью, узнайте про компанию, про её продукты и планы. Во-первых, это в ваших интересах. Вы не хотите идти в компанию без особых перспектив, у которой куча проблем на разных фронтах. Во-вторых, часто компаниям бывает важно увидеть хоть какой-то энтузиазм в кандидатах. Не нужно восторженно рассказывать, как вы всю жизнь мечтали там работать и вообще они компания вашей мечты. Но разумная беседа в ключе "Мне очень нравится ваш продукт X. До вас ничего подобного никто не делал, а сейчас этим продуктом пользуются очень многие. Включая меня, и мне он нравится тем и этим.". Будьте честны. Это должно быть несложно, так как у почти любой компании можно найти продукт, хоть один, который вам понравится. А если такого нет совсем, то вряд ли нужно с такой компанией связываться.

Книги для подготовки к интервью

- "Programming Interviews Exposed: Secrets to Landing Your Next Job"
- "Cracking the Coding Interview: 189 Programming Questions and Solutions"
- "Elements of Programming Interviews: The Insiders' Guide"
- "Programming Pearls"

Сайты с задачами

До интервью постарайтесь набрать 500-1000 задач, причем чтобы как минимум половина из них была medium/hard уровня. Решайте 10-15 задач в день. Кажется, что это очень много, но скорость сильно повышается со временем. Ту же задачу решить займет вначале час, а через месяц подготовки - 10-20 минут.

- HackerRank
  Там есть задачи уровня Easy, Medium, Hard. Начинайте с Easy (ну, разве что вы уже отлично решаете задачи и вообще олимпиадник-победитель по программированию), продолжайте через Medium и постепенно доходите до Hard.
    - Cracking the Coding Interview Tutorial - небольшой ликбез об алгоритмах и структурах данных от автора CTCI
- LeetCode
  Когда вы уже более-менее уверенно будете поднимать задачи уровня Medium на hackerrank, переходите на leetcode. У них там есть Mock Interview – решение случайной задачи по таймеру. Задачи там можно пропускать. Если вы задачу пропустите, то в другой раз она может попасться снова. Решайте задачи уровня Medium, иногда Easy когда устали, если получится – пробуйте решать и Hard. Продолжайте решать 10-15 задач в день (если получится больше – вообще круто).
- InterviewBit
  Когда вы уже неплохо решаете задачи на моках leetcode, переходите на interviewbit. Там задачи довольно сложные, сразу я бы за них браться не советовала, а вот после подготовки на предыдущих сайтах вполне можно.
- CareerCup
- GeeksForGeeks
- ProjectEuler
- TopCoder

- CodeWars
- Interviewcake
- CodeFights
- Sphere Online Judge
- Hiredintech

Сайты для тестовых интервью

- https://www.pramp.com - бесплатно
- https://www.careercup.com/interview (за деньги, и обсуждение стоит ли оно того)
- https://interviewing.io - бесплатно
- http://larrr.com/testovye-intervyu - бесплатно
- https://www.candidacy.io
- http://www.gainlo.co/#!/ - платное, но пишут, что они проводят интервью с “настоящими инженерами” из известных компаний. На себе не проверено.

Прочие полезности

- Mega Project List - a list of practical projects that anyone can solve in any programming language.
- Technical Interview Megarepo - study materials for technical interviews.
- Awesome Interviews - a curated list of lists of technical interview questions.
- Mastering the Software Engineering Interview - курс для подготовки к интервью на Coursera. Курс платный ($80), но есть программа financial aid, которая позволяет пройти его бесплатно и получить сертификат.
- Technical interview (Udacity) - бесплатный курс. Дает представление о том, как вообще выглядит интервью, для новичков, и несколько полезных алгоритмов в придачу.

Статьи

- Hiring & Interviews
- 7-Step Interview Prep Plan - Princeton University
- Guide to Job Hunting and Technical Interviews

Рассказы про прохождение интервью

Также хорошей подготовкой будет изучение опыта предыдущих кандидатов - как успешные, так и неудачные. Из успешных можно понять, что именно привело к успеху, из неудачных - учесть их ошибки и постараться не допустить подобные.

- Sobit Akhmedov - Amazon software engineer interview (оффер), перевод
- Jay Huang - My Amazon interview experience (не получил ответа), перевод
- Stepan Suvorov - Front-end Engineer собеседование от Amazon (отказ)
- Jason Clawson - Interviewing at Google, part 1 (перевод), part 2 (перевод) (оффер)
- Tom Goldenberg - My Google Interview and Lessons Learned (отказ)
- “Kevincav” - My experiences with interviews at Microsoft, Google and AT&T (оффер в Microsoft и Google, отказ в AT&T)

- Jon Guerrera - How I Gamified the Google Interview (оффер)
- Роман Ворушин - Мой опыт собеседования в Google (оффер)

Во время собеседования

- Не приступайте к решению задач до тех пор, пока не будет полного понимания, что от вас ожидается. Ведите себя на этих собеседованиях как на мозговых штурмах: общайтесь с интервьюером, задавайте вопросы, обсуждайте решения, и когда всем будет ясно, что надо делать, берите в руки маркер/карандаш/мышь и делайте.
- Есть хороший сайт для подготовки к интервью,https://www.hiredintech.com/. Там есть некий шаблон, называется The Algorithm Design Canvas. Он содержит 5 областей -- Constraints, Ideas, Complexities, Code, и Tests. По мнению авторов canvas облегчает решение задач при phone screen интервью -- вместо того, чтобы замыкаться и думать что делать дальше перед вами будет шаблон процессов следуя которому можно решать любую задачу. Ну и куча всего для подготовки, структурированная подача материалов.