Algorithms and Interviews

Job Interviewing for Programmers

Cody Jackson

April 22, 2021

Code used in this book can be found at https://github.com/crystalattice/Algorithms_and_Interviews.

Preface

This book is the culmination of many months of research. While all efforts have been made to ensure accuracy, some information may not be correct. I do not have a Computer Science background; my undergraduate degree is in Computer Engineering. Much of the technical part of this book required me to learn how the described algorithms work and confirm that the source code produces correct results.

In addition, my career is not normal. Much of this book comes from what I have personally experienced and what I have learned over the years. After retiring from the military, I moved into the world of government contracting, which is different from more traditional corporate work. While I have tried to account for the most common interview experiences, actual interviews will vary depending on your location, the company you're applying with, and the industry that company is part of.

Part I.

Prepping for Interviews

1. Programming Interviews

When it comes to job interviews, programming interviews are probably some of the scariest. Even seasoned programmers often have a case of the nerves because you never know what the interviewer(s) will ask. Will it be a discussion of your past projects and general knowledge, or will you be put on the spot and have to design a solution on a whiteboard? Even worse, what if it is something you are expected to know because of your background but, in fact, you have never used before in your projects?

This book will attempt to help prepare programmers, particularly Python programmers, for coding-specific interviews. It covers basic interview strategy and preparation, common algorithm-based questions and how to write answers using Python, and touches on some subjects that are commonly found in Computer Science classes.

In this chapter, we will cover the following material:

- Coding interviews
- The role and process of coding interviews

1.1. Coding Interviews

I'm going to put this out right now: I haven't had a normal programming interview, except once when I interviewed with one of the big tech companies. If you read my author's bio, you'll see that I have an eclectic background. I don't have a Computer Science degree and, while I did learn C, C++, and Java for my Computer Engineering degree, I never had the opportunity to use them while on active duty.

I taught myself Python around 2006 and made some hobby programs over the years. I wrote a book on Python programming while deployed to Iraq, as I was disappointed in the books I learned from. Jump to 2015 and my deputy department head came to me when he found out I had wrote a Python book. The command's sole Python programmer was getting ready to transfer so they needed me to replace him. No formal interview or anything; the fact that I was an author was good enough.

After I retired, I got a job as a government contractor. That was a phone interview and, again, the fact that I had a programming book on my resume pretty much gave me the job. The interviewers asked some perfunctory questions, like what a Python dictionary does, but I guess the assumption was that, if I wrote a book, I know how to code.

Since then, I have written several "legitimate" books, i.e. actually published by a publishing company and not self-published, created some programming videos, and I get contacted about once a week by headhunters.

There was only one interview I have had that followed the standard "programmer's interview". I conducted an initial phone interview to clarify my base tech knowledge, then I had an online, hands-on-keyboard interview via an online programming environment.

I will admit, I flubbed that interview. At that time, I had only been "professionally" programming about a year and a half, and even then it was only two projects that weren't for my own needs and used outdated software. I didn't have the muscle memory or mental memory to quickly create the short applications desired; at that time, I still needed to frequently look at a reference.

Long story short, if you have the credentials on your resume and a solid portfolio of work, you may not have to go through a formal interview process; your work will speak for itself. Plus, people will come to you to interview with their company, rather than you having to apply to the company.

1.1.1. Interview Process

Luckily, most interviews take the same approach, regardless of the industry and the skills required. Generally speaking, there will be a little chit chat to get you (a little) more comfortable, some basic technical knowledge questions, then the coding questions.

Of course, you have to get to the interview first. The following description doesn't apply everywhere, but it is fairly common. The first challenge is making it through the Human Resources (HR) filter machine. HR uses a variety of screening tools to reduce the number of resumes to manually look at. If someone one sends a resume to a company, it's automatically parsed to match the "keywords/tricky-phrases" listed in the job description. If there aren't a sufficient number of matches, then applicant is automatically rejected.

If your resume makes it past this hurdle, then your resume is placed into the recruiter's repository so a corporate headhunter or recruiter can contact you. This step can also happen automatically through job posting sites or social media sites like LinkedIn; your resume and job information is listed in the service's database and recruiters are able to search for keywords to find applicants.

This is not a foolproof system and it is easy to tell when you aren't actually being recruited with the human touch. Often, I will receive contact from a recruiter claiming that my information matched a need of the company, yet the skills they list are nowhere to be found on my resume or job history. They are simply looking for someone who matches the basic criteria of "programmer" and not necessarily the skills the company needs.

After the company has responded to you, you frequently have a phone interview to decide whether to proceed. These are normally pretty easy, depending on your background and the needs of the company. If you pass this test, then you'll probably be set up for a technical interview. The screening interview was simply to see if you match the basic needs of the company and whether it should invest time in moving forward. The technical interview is to determine your technical programming abilities, but it isn't the final interview. We'll talk about the technical interview in detail later on, as it's basically the purpose of this book.

Assuming you pass the technical interview, you'll be called in for a formal interview. At this point, you'll be asked the normal interview questions ("Tell me about your strengths."). You may get some more specific technical questions related to the general work of the company, or related to the work you will be expected to do.

During the formal interview, this is also your opportunity to ask questions about the company, and it is considered bad manners to not have any questions, as it can be interpreted as you have not researched the company.

At this point, assuming you pass this interview, what happens next depends on the company. You can have more interviews with various people from the company, such as your potential team members, your boss, or pretty much anyone else the company wants to include in the interviews. Ideally, however, this is the final interview and you'll either receive a job offer or become free to continue your job search.

Your first, formal interview may be with the actual hiring manager, or it could be with a designated "screening interviewer", someone who has a lot of experience in interviewing people. The first interview can act like the HR filter, determining whether to allow an applicant to enter the hallowed halls for more interviews with the actual hiring department.

It's not uncommon to hear about people who have 3-4 interviews before they are finally offered a job. Part of this is to see how serious applicants are about the job, but some of it is simply bad business practice. By this, I mean that company's seem to think that applicants are at the company's beck and call, not realizing (or not caring) that applicants often have to take a full day off work, plus travel time, to make it to an interview.

By the time you are offered a job (assuming it wasn't at the time of your very first interview), you should have a good understanding of what your duties will actually be, how it fits into the company overall, and what the corporate culture is. Think about it objectively and don't be dazzled by the name of the company, the job title, or even the salary being offered.

Smaller companies may not be as prestigious as big-name tech companies, but they often allow more free reign in your work duties, plus there more small-time jobs available than at big companies. Plus, if you're just starting out, working for a small company gives you a chance to hone your skills and decide on a specialty, e.g. web development, IoT development, etc.

1.1.1.1. Security Clearances

It should be noted that, if you have (or are eligible to receive) a security clearance, you can pretty much guarantee that you will find employment somewhere. Because there are only about 3-5% of the US population that has, or is eligible for, a clearance, there are obviously more jobs available than people to fill them. If you're not picky about where you live, i.e. somewhere near Washington D.C., you can always find an employer quickly. Outside of that area, jobs are fewer but still available, such as Colorado, Texas, et al.

If you have a clearance, the interview process is abbreviated, as I noted earlier. You won't have nearly as many interviews; typically, just a single one where you'll have a manager and a technical person. Depending on how strong your resume is, the manager will probably spend more time selling you on the company rather than asking you questions. The technical person will also probably spend more time telling you about the job and other work details than quizzing you on your knowledge. As long as you have the credentials the company needs to fulfill a contract, you're as good as hired.

Government contracting companies have to meet certain manning levels at certain points in their contract timeline. So it's not uncommon to receive a lot of recruiter contacts at the start of the new fiscal year and about six months later, as those are prime times to fill seats.

1.2. The Role and Process of Coding Interviews

Coding interviews ultimately look at applicants to see whether they actually know specific skills related to programming. You can find many horror stories online, such as TheyDailyWTF (https://thedailywtf.com/), that give examples of bad interviews, as well as bad coding examples. Just because someone graduates with a Computer Science (CS) degree doesn't necessarily mean they know how to program.

This is actually a key point. Most people assume that CS degrees teach students how to be programmers. In reality, Computer Science teaches programming as part of a larger picture: how to interact with computers and make them do things using software. CS, as a field, is primarily designed to teach people how to further the science of computers by developing new theories and technologies by teaching how to make computers do things via algorithms and higher math.

These are the people who create blockchain-based crypto currencies, machine learning algorithms, autonomous vehicles, and such mundane things as new file systems or database architectures. As such, depending on the curriculum, CS students don't necessarily learn how to program well, but learn enough to be able to interface with a computer.

Software engineering takes the ideas developed by the computer science field and applies engineering principles to it. You could almost say that a lot of programming best-practices were developed by software engineers to improve on the basic principles developed by Computer Science.

Programming, by and large, doesn't require formal training. There is nothing stopping someone from getting a Python programming book and learning how to make applications from that knowledge. Given enough time and practice, that person could conceivably be hired based on their knowledge, as basic business applications rarely require a lot of foundational Computer Science knowledge. That's one reason why there are so many programming boot camps nowadays.

Tying this back to my original thought, coding interviews are designed to show whether someone actually knows how to utilize their knowledge to create useful software, or if they just learned enough to get through school. Therefore, many questions are designed to test the applicant's knowledge in fundamental programming concepts and data structures, not necessarily to show whether you can actually implement them, but often to see whether you actually paid attention in class and know the basic concepts of how they are used.

1.2.1. Tips for Preparation

While I have often went to an interview cold, without preparing, it was usually because it was either a government contracting job or I really wasn't interested in the job but wanted to see what the market was offering, it's better to do a little preparation before going to an interview. Think about what you might be asked, and think about what you want to ask in return. It may not be your sole interview with the company, but if you don't do well in the first interview, you won't be asked back for any follow-ups.

1.2.1.1. Know how to FizzBuzz

One of the key things to help prepare for a programming interview is to be able to write FizzBuzz code. FizzBuzz (https://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/) was designed as a simple coding

exercise to quickly and easily identify whether someone can actually program or only looks good on paper. Given the following algorithm, it should be easy for a competent programmer to develop an answer in ten minutes or less:

Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

There is no guarantee that you will encounter a FizzBuzz question in your interview, but knowing how to answer it means you at least know how to write functioning code. Also, the idea of being able to handle FizzBuzz means you should be able to answer similar "easy questions".

1.2.1.2. Corporate Research

Moving on, you should learn a bit about the company you're applying for. Find out what they actually do, that is, what industry are they actually in. For example, is it a purely tech company, like Microsoft or Google, or is it more service oriented, offering SaaS (Software-as-a-Service) access to other businesses?

What are the long term plans of the company? That helps shape what your role in the company may morph into, as well as what things to put on your planner to learn.

What technologies and frameworks does the company use? Obviously, if they are using a software stack you're not familiar with, you have to decide whether it's worth even applying, with the possibility that you won't get to an interview. Or, potentially worse, you do get hired and are expected to get up to speed in a short amount of time.

Sometimes the tech stack doesn't really matter; they just want someone who knows how to program. Either they assume you can learn their tech stack, or they have other projects going on where you might be a better fit. This actually happened to me; I expected to interview for a regular programming job, but ended up interviewing for, and being hired for, a Linux modeling and simulation position as the company's programmer was retiring.

These aren't the only things to look up, but it should give you an idea. The main point is to demonstrate that you actually care about the company and what your role is within the overall mission. If you don't know what the company does, it makes it difficult to demonstrate how you add value to the bottom line.

1.2.1.3. First Impression

I won't hammer this point too much, as there are many resources available for interview tips. The main thing is to dress for success and convey the right attitude.

Your wardrobe should represent the position you want; at a minimum, you should be one "level" above your interviewer. For example, if khaki pants and polo shirts are the norm at the company, throw on a dress shirt and tie. If shirt and tie are the norm, then wear a full suit. It's better to be overdressed than under-dressed.

When greeting people, and during the interview, maintain eye contact, use open body language, and remain enthusiastic. Remember people's names and try to use them at least three times during a conversation. This helps reinforce the association between their name and face, as well as subtly improving their opinion of you.

1.2.1.4. Modest Boasting

There will be several opportunities to talk about your work, whether it comes from a question about your resume, or the stereotypical, "Tell me about yourself" question. This is the time to brag about your accomplishments, whether it is school projects or your work portfolio.

No one likes a braggart, so be aware of immodest boasting. Talk about the work you did, emphasizing the results, especially if there are numbers involved. Results could be money saved, time improvements, error reductions, etc. Generally speaking, programmers just need to demonstrate coding ability, but if you can put some results on your accomplishments, it helps reinforce your value to the company. It can also help when negotiating salary and perks.

1.2.1.5. Coding Exercises

Of course, you must be prepared to write code. As my experiences in the beginning of this chapter demonstrate, you may not have to actually code for an interview, but the possibility remains. Thus, be familiar with some of the most commonly used APIs and standard library calls, and know how to use them, in a general sense.

Interview coding is designed to be answered in 20-30 minutes, usually; take-home questions are rare. Be sure to ask questions about the scenario, such as assumptions and requirements. Write down everything; these questions are less about getting the right answer and are more about seeing how you approach problems. Draw diagrams, flow charts, and anything else that helps you think about the problem. As you write the code (on a computer or whiteboard), talk about what you're doing and why. If you're thinking, think out loud.

Don't worry about finding the most elegant solution; a brute force solution is acceptable, as long as you acknowledge that it should be refactored and provide possible changes to improve it. It's not unusual for a "real" programmer to use reference material, so don't be afraid to say something like, "I know this function call exists, but I don't recall how it is formatted", as long as it isn't a basic concept that everyone should know. For example, not knowing how to use a Python list comprehension or iterate through a dictionary probably won't instill confidence in the interviewer, but not being familiar with the *hashlib* library is probably acceptable.

It's important to note that some unscrupulous companies interview programmers with the intention of having the programmer provide "work for free". What this means is that the interviewer provides a problem that the company is currently dealing with; the interviewee's solution will end up being utilized by the company and the interviewee is not hired.

The best way to avoid this is to not provide details in your answers. Hedge your answers with phrases like, "This is the general concept, but I could provide a more detailed solution once I'm hired" or "This function provides these API capabilities" without explaining how the API internals operate.

1.2.1.6. Coding Test Sites

There are a number of programmer knowledge testing sites, such as CodeFights or HackerRank. Each person will gain different amounts of benefit from them. In my personal experience, I think they aren't that valuable in terms of programming tasks. I spent some time on one site, just to test my knowledge. I

eventually became disillusioned with the site, as the questions seemed to be more like programming homework questions rather than tasks you would actually have to do on a job.

Some sites are better, like LeetCode, in that the questions put on the site are from people who just completed an interview and so they post questions they encountered during the interview. In that sense, the questions are at least related to what you can expect for big name tech companies. Often, these sites also provide additional information about the companies beyond just programming questions.

In summary, if you want to practice your coding skills, try a variety to sites as some are better than others.

1.2.2. Resources

Below is a list of useful sites for interviews in general, and programming interviews specifically:

www.thebalancedcareers.com
www.monster.com (100 interview questions)
www.monster.com (job interviews)
www.programmerinterview.com
www.indeed.com
simpleprogrammer.com
www.thesoftwareguild.com
www.asktheheadhunter.com

1.3. Summary

In this chapter, we discussed the fact that, if you have a strong portfolio, you may be able to bypass much of the interview process. Then we discussed the interview process in general, and some special conditions for programmer interviews. We reviewed some tips for interview preparation, including links to helpful sites. We also mentioned online programmer test sites and how useful they can be.

2. Interview Preparation

Preparing for an interview starts long before you sit down in the interview chair. Preparation starts when you make the first draft of your resume, and continues through the application process until you are sitting at your desk.

In this chapter, we will cover the following topics:

- Applying for jobs
- Pre-interview preparations
- General interview process
- Resume creation

2.1. Applying for jobs

Looking for work is one of the most stressful things you can do. Not only do you have the stress of looking for jobs that you can (hopefully) meet the criteria for, you have to have references on stand-by, get your resume and cover letter looking good, practice answers to anticipated questions, ensure your wardrobe is able to make you look good, and many other things.

In this section, we will cover some of the traditional and non-traditional ways of looking for and applying for jobs.

2.1.1. Traditional methods

Traditional job hunting basically involves tweaking your resume to a one-size-fits-all theme that can be applied to nearly any position within your field. Then you upload it to job websites and maybe even several targeted companies. Then you wait for a response from someone at the company to say that they received your resume and will let you know if they have anything that meets your qualifications.

Maybe you'll be a bit more targeted and utilize LinkedIn or other social media to help facilitate the process. LinkedIn does work; the first job I received after I retired from the military came from a LinkedIn job posting. But, that was after receiving multiple contacts from recruiters about job postings who then ignored my requests for more information.

It's interesting to note that, in many cases, by the time a job posting is shown on a job site, the position has already been filled. This is often due to filling internally or the company already had someone in mind for the job, but they were legally obligated to post the opening.

This doesn't happen all the time, but consider this: if you're putting your resume out to many different job sites, companies are doing the same thing with their job openings. Therefore, the odds of someone else being selected for a particular job are higher than the odds of you being selected for that particular job. Quite simply, there are going to more applicants from the various job site notifications; unless you're lucky enough to be in the first batch of applicants, the odds of someone else being hired before they even review your application are small.

2.1.2. Non-traditional methods

Non-traditional job hunting is using your network of contacts to find a potential opening, as well as circumventing the HR filter to talk to the hiring manager directly.

While this is the most highly recommended way of getting a job, it does require some effort on your part, so most people take the easy route of shotgunning their resume across job boards and hope they get lucky.

The best job opportunities are the ones that aren't advertised. Maybe someone is getting ready to retire, but it's not in the window of posting a job opening. However, if you contact the right people, you could potentially be hired as the future-retiree's "apprentice" to learn the job while they are still around.

Or maybe you learn about a problem a particular company has, or specific needs they cannot fill. You create a proposal and approach the company about working for them to fill that hole. They may not have an immediate opening for you, but they will probably either make a position or bring you on as a contractor while they work to make an opening for you.

In short, you need to talk to a decision maker and demonstrate how you add value to the company. This isn't about filling a pre-designed role, but about showing what you can do for the company, both in filling a particular role as well as future capabilities.

2.1.3. Non-traditional jobs

A common theme I see on LinkedIn, Quora, and other forum-style sites is people asking how to get a job that pays X amount, or how to get on with a particular company, or questions along those lines. The common denominator among them is that people think of traditional corporate jobs, but there are many opportunities outside of the corporate office.

First, consider smaller companies. They may not be able to pay as much as the big companies, but it also means that you have a better chance to shine, as well as play a large part in their operations. That could include spearheading new projects, revitalizing legacy software, or expanding your knowledge by working with non-traditional tools.

Second, government work is often forgotten. Government civilians frequently dual-hat as both technicians and managers, with more managerial work as you increase in seniority. For example, if you work for the Department of Defense, you could be the civilian lead of a group of military personnel and civilian contractors. This expands your abilities by requiring you to learn project management and other managerial skills while honing your technical skills to match what the rest of the team provides.

Finally, government contracting can be lucrative, especially if you have a clearance. People like to talk about the six-figure salaries when working for Big Tech, like Google. But you're required to work in high cost-of-living areas like San Francisco, Austin, New York, etc.

Because software engineers with security clearances are comparatively rare, you can earn significantly more in a cheaper location, since government facilities are scattered around the world. Government contracting companies earn millions of dollars per contract, so they can afford to pay top dollar to programmers with the necessary skills who also have a security clearance.

To give one example, a senior programmer living in Texas can make $120,000-$140,000 per year. The equivalent salary in San Francisco would be $240,000+.

In addition, Texas doesn't have a state income tax, so that's an extra $10,000+ in annual income. And even if you don't currently have a clearance, many contractors will hire you with the expectation of acquiring your clearance within a few years.

In summary, don't get caught up in considering normal jobs. If you look hard enough, you can find a good job nearly anywhere.

2.2. Pre-Interview Preparation

Some people like to go to interviews "cold", not preparing (or not preparing much) as they expect each interview to be different. Others over-prepare, expecting every job interview to be "the one" that yields a job offer.

Regardless of your approach, there are some common things that can improve your chances of making a good first impression.

2.2.1. What to wear

Plan your interview outfit. For companies that have a business or business casual dress code keep your look basic and conservative for the first interview. If you don't know what to wear, you can ask someone who works there or even ask the recruiter or HR representative. Try to dress at least one level above normal; for example, if normal wear is a polo and khakis, then wear business casual (button-up shirt without a tie and dress slacks). Make sure you get your outfit cleaned, pressed, and tailored if necessary.

Don't forget about the little things. Shine your shoes, check for loose hems, and make sure your fingernails look good. These little details count, as they provide a subtle demonstration of your attention to detail.

Don't be afraid to pamper yourself, because looking your best helps you feel your best. Many people will get a fresh haircut or visit a spa to not only look their best, but also help them feel more relaxed and confident.

2.2.2. What to bring

Bring at least three printed copies of your resume. You never know who you'll be meeting with, and you want to have your resume ready to go in case you're asked for it.

Prepare a reference list, whether you think you'll be asked for it or not. Commonly, references aren't contacted until you are in the final stages of being considered for a position, but each company is different.

Prep an interview kit. It's recommended that it not be a separate bag but something that will fit in your purse or briefcase. It should include things such as extra resumes, a notepad, as well as an emergency kit with things like bandages, stain stick, needle and thread, breath mints, etc. While you're at it, clean out your bag. Everything you need should be neatly organized and readily accessible. The less you have to rifle through your bag, the better.

2.2.3. What to know

Spend time learning everything you can about the company, from as many sources as you can. Often, candidates just look at the public information about a company from websites and social media. Don't forget to look at the latest business news to see what internal or external events are affecting the company. Don't

forget sites like Glassdoor, where employees provide insight into the good and bad parts of a company.

Get a sense of "who" the company is and how to embody a similar personality during your interview. Essentially, you want to demonstrate how the mission statement and vision of the company applies to you, and how you will provide value to the company by embodying those principles.

No matter what role you're interviewing for, whenever possible, you should always use the product before your first interview. You should have an understanding of what the company provides to customers because, if hired, you'll be part of the process that creates value for customers. You may also have some suggestions for the product that will a) demonstrate you know what the product does and b) you care enough to think about improving it for customer use.

Before your interview, get a list of the people you're meeting with from the company. Learn about them; if they have a bio page on the company web site, use it to find out more about their background; if possible, find some common ground to make small talk about, if necessary. While you're at it, think of some potential questions to ask them if the opportunity presents itself.

Different companies use different types of interviews, so don't be afraid to ask what you'll be faced with. For example, big tech companies are famous for asking case questions or brain teasers while others will give a standard set of typical interview and leadership questions.

2.3. Interview Process

The job interview process can be long and tedious. Depending on the position, it can be nearly instantaneous, like in a matter of days, or it can drag out for weeks or months. Every process will be different, depending on the needs of the company.

For example, in a previous job, I went to the interview expecting to talk about a programmer position. However, the company's models & simulations engineer was retiring in six months and they needed a replacement. So, rather than discussing programming, we talked about my Linux background and how quickly I could get up to speed with new applications. The next week, I received the formal job offer and accepted.

In this section, we will talk about the possible interview workflow that you may find yourself in. This is essentially a worst-case scenario, as I have never personally had this many steps in the process but, from what I understand, it isn't uncommon for the programming community to encounter this.

2.3.1. Screening Interview

Your first interview will probably be with a recruiter, on the phone. The recruiter may ask some general tech questions to figure out what your basic capabilities are and whether you are ready to move to the next step in the process. It also gives you a chance to ask some questions about the job in a non-technical sense, e.g. where is the position located, what other opportunities exist at the company, etc.

2.3.2. Phone Interview

This interview is with someone within the hiring chain or from the shop where you would be working. This will be more in-depth than the screening interview,

and is often a video conference so the interviewer can see you and how you handle the interview process. Therefore, you should dress the part; maybe not suit & tie dress, but don't wear a grungy t-shirt. It's still a professional situation.

2.3.3. Tech Interview

In the programming world, a tech interview is frequently held over the phone, or more often video call. In this instance, you are often asked to talk about algorithms and other programming questions without the benefit of using a computer. For example, you may be asked to describe how a particular algorithm works and maybe even write pseudocode on a whiteboard. This is designed to see how you think, as well as whether you actually know how to program intelligently rather than relying on pre-built tools to do all the work.

2.3.4. In-Person Interview

This is frequently another tech interview, but this time in person. Again, you may find yourself writing code on a whiteboard and talking about code, but it will be at a higher level than before.

Often, you may spend an entire day at the company, meeting with various members of your potential team and answering a variety of questions. These include not just tech questions, but also meeting with managers and talking about non-technical aspects of your work history or describing how you would solve particular scenarios, both technical and managerial.

2.3.5. More Interviews

Apparently, it's not uncommon to go through three or more interviews for certain companies. These can be group interviews, with multiple job candidates meeting with a single interviewer, multiple candidates with multiple interviewers, or a single candidate with multiple interviewers. Or they can simply be more one-on-one interviews with different members of the company hierarchy.

In some situations, depending on the position applied for, you may find yourself at a lunch or dinner interview. This is supposed to be a more informal situation, allowing you to relax and "be yourself". But don't relax too much; they are trying to find out your true communication and interpersonal skills, as well as your social interactions in non-business settings.

2.3.6. Final Interview

The final interview is the one you want. This could be the first interview, if there are no other candidates, or it could be weeks after the first interview. In this interview, the company has pretty much made up its mind that it will hire you. However, you can still screw up it up at this point, so don't slack off. If there is anything that the interviewer asks for, or was requested in a previous interview, now is the time to bring it, such as lists of references, additional resumes, etc.

This is also a good time to ask any questions that you have thought of during your interview process. These can be clarifications of information learned during the process, or questions that haven't been answered yet and you couldn't find answer online. They could also be questions based on new things that have recently occurred, such as a new product or merger.

Recognize that, even if it's a done deal, it can still take some time for the company to formally offer you the job. Don't fret if you don't hear anything

immediately after the interview. However, if after a week you haven't heard anything, don't be afraid to contact the recruiter or your indicated point-of-contact to find out the status. It's not unusual for a company to hire someone else and neglect to tell the other candidates about it.

Also, until you have signed the paperwork, you don't have a job. Even if you pass all the interviews and they verbally offer you a job, it can still be rescinded prior to you starting. This happened to me once.

The day I was to sign the paperwork with HR, I was notified that the offer had been rescinded, leaving me to immediately start looking for another job. So always keep multiple options open, because you never know if your second or third choice may become your only choice.

2.4. Preparing your resume

Resume prep can be a fairly detailed process; there are books dedicated to creating the "best" resume, as well as different styles based on the potential audience. We won't cover everything in this section, but will highlight some of the most common recommendations for the "average" resume.

2.4.1. The Key Information

In this section, we will talk about the meat of your resume: the information that makes a resume a resume and not an employee bio.

2.4.1.1. Don't Include Everything

Your resume should not have every work experience you've ever had listed on it. Each resume you send in for a job should be tailored to that position. With that in mind, you'll want to highlight only the accomplishments and skills that are most relevant to the job at hand, even if that means you don't include all of your experience.

2.4.1.2. Keep a Master Resume

A master resume is one that documents all the work you have done since high school, or at least for the last seven years. The master resume is where you keep any information you've ever included on past resumes, such as old positions, bullet points tailored for different applications, special projects that only sometimes make sense to include. Then, when you're crafting each resume, it's just a matter of cutting and pasting relevant information together.

LinkedIn is a good place for this. Since it is a central location and has all the fields you need to worry about, I use it as my master resume. Then, when I apply for a job, I just take the latest version of my generic resume (which is also posted on LinkedIn), and change the resume based on the information I've added to my LinkedIn account since the last time I made a resume.

2.4.1.3. Put the Most Important Information First

Make sure your best experiences and accomplishments are visible on the top third of your resume. This top section is what the hiring manager is going to see first, and what will serve as a hook for someone to keep on reading.

2.4.1.4. Reconsider the Objective Statement

There was a time when common wisdom was to put an objective statement on your resume, much like a 3-4 line cover letter. Nowadays, the only occasion when an objective section makes sense is when you're making a huge career change and need to explain why your experience doesn't match up with the position you're applying to. Otherwise, consider whether an objective statement adds any value or if you would be better off using the space for more important information.

2.4.1.5. Keep the Standard

The traditional reverse chronological resume is still considered the best for most situations. Unless there is a reason to use something different, such as the functional resume, the tried-and-true should be your first choice.

2.4.1.6. Keep it to a Page

Again, common wisdom is to keep your resume to one page. Two pages is acceptable if you have a lot of relevant information to the job, such as previous projects, publications, etc. Even having 3-4 pages can be useful, depending on the type of job. But only use multiple pages if it is truly necessary; one page should be what you strive for.

2.4.2. Formatting

In this section, we will discuss how the resume should look, within certain guidelines, to ensure it remains professional but usable.

2.4.2.1. Keep it Simple

The most basic principle of good resume formatting is to keep it simple. Use a basic but modern font, like Helvetica, Arial, or Century Gothic. Make your resume easy on hiring managers' eyes by using a font size between 10 and 12 and leaving a healthy amount of white space on the page. In addition, recognize that your resume will most likely be scanned via OCR, so you need to ensure that the font type and size are easily processed by the OCR software. You can use a different font or typeface for your name, your resume headers, and the companies for which you've worked, but keep it simple and keep it consistent.

2.4.2.2. Make Your Contact Information Prominent

Keep your personal information off the resume, so don't include your address or other private information. However, do maintain a professional email address and phone number so that you can separate your personal life from your professional. If you're going to include social media links, then ensure that those profiles are strictly professional; you don't want your potential boss to see you passed out drunk from a weekend party.

2.4.2.3. Design for Skimming

The average resume gets 30 seconds of review before a decision is made to investigate further or move on. Ensure your resume is formatted to aid in skimming: put the important information at top, or separated with noticeable changes, e.g.

font type or size, and use a standard template format. Don't try to be clever or it may actually hurt you.

2.4.2.4. Professional Help

While it is highly recommended that you write your resume yourself, there are professional services that will help with the formatting and placement of information for the biggest impact. As a matter of fact, I used one when I was transitioning out of the military, and it was very helpful. I provided all the information in a generic resume format so the designer didn't have to try to make things up. He simply took my information and revised it for maximum impact in the limited amount of space of a resume.

2.4.3. Work Experience

Since most resumes are in the reverse chronological format, your work experience is a key factor in filling out your resume.

2.4.3.1. Keep it Recent and Relevant

Some sources indicate you should only show the most recent 10-15 years of your career history; my personal opinion is 5-7 years, simply based on my ability to fill out a page and a half with just five years of work history. Depending on your work history, you may have more or less time on your resume. For example, if you're worked at one location for ten years, then you may only list one job but with a dozen bullet points of accomplishments.

Include only the experience relevant to the positions to which you are applying. This way, it is easy to determine which items to remove if you're running into space issues. I always go with more information in the beginning, then start to remove information as necessary for space issues. Even if it's not necessarily relevant to the job, I like to show accomplishments (whether personal or professional) to help demonstrate capabilities beyond my normal work requirements; meeting expectations is easy, but exceeding expectations can be more difficult. Plus, you can always remove these items if necessary.

2.4.3.2. Transferable Knowledge

If you don't have experience that is directly related to the job, then consider other things you have done or know how to do. Transferable skills can be found anywhere. This is when a resume book comes in handy, as it can provide action words that can kick-start your thought process and make you realize that you have more abilities than you think you do.

2.4.3.3. Minimize Jargon

Recognize that the hiring manager may not be a technical expert. It isn't unusual for managers to be hired or promoted outside of the actual technical pipeline. Therefore, keep technical jargon to a minimum. For example, stating that you created a CMS for business intelligence analysis could be rewritten as "designed and implemented a content-management system for data analysis workflows", or something like that. You don't know who may see your resume, especially if your resume is passed around to different departments or even companies to find the right fit for you.

2.4.3.4. Numbers, Percentages, and Results

Use as many facts, figures, and numbers as you can in your bullet points. Quantify everything you can, especially when it comes to results, such as "Developed 7 command-line scripts that improved customer service operations by 30% and saved $35,000 in contractor costs."

Whenever possible, emphasize the results of your work. Sometimes, it's impossible to know the direct and indirect results of your work, especially if you leave before the final results are known. In these instances, your best guess is okay, especially if you have an idea of what the intended results were.

2.4.3.5. Use Different Phrases

If every bullet starts with "Responsible for..." or something similar, readers will get bored very quickly. Again, this is where having a list of action words helps, because you can say the same thing in different ways and no one will notice.

2.4.3.6. Use Keywords

Scan the job description, see what words are used most often, and make sure you've included them in your bullet points. This ensures that you're targeting your resume to the job, as well as making sure it gets noticed in applicant tracking systems.

2.5. Education

For some applicants, education is about all they have. For others, it's been years since they were in school. This is another case where your resume needs to be written to highlight the most important parts of "you".

2.5.1. Experience Trumps Education

Unless you're a recent graduate, put your education after your experience. Your last couple of jobs are more important and relevant to you getting the job than college, as practical experience is more important than academic learning. If you can't walk the walk, then talking the talk is meaningless.

That said, if you know the hiring manager or some other important individual is an alumnus of the same school you attended, you might want to highlight the school. Some people put a lot of emphasis on where you went to school, as demonstrated by wearing a class ring, for example. If you can "get in good" because of the school, that can bypass some of the interviewing.

2.5.2. Don't Worry About GPA

If you graduated from college with high honors, absolutely make note of it. Don't worry about putting a GPA; as long as you graduated with a degree, that's all that matters in the working world.

2.5.3. Include Continuing or Online Education

Any education you take that is relevant to the job should be included: continuing education, professional development coursework, or online courses. Don't forget certifications count as education as well.

2.6. Skills, Awards, and Interests

Once you have filled out your work history and education, you may have extra space available. On the other hand, you may have completely filled the paper. Either way, you should consider highlighting special things that make your resume unique among all the applicants.

2.6.1. List Out Relevant Skills

Be sure to add a section that lists out all the relevant skills you have for a position. This is relatively easy, as the job description should describe all the skills that are being looked for.

2.6.2. Awards and Accolades

Include awards and accolades you've received, even if they're company-specific awards. Consider that some awards may be industry-specific, especially military, and should be converted to general terms. For example, if you received a medal from a military commanding officer, annotate it on your resume as "received CEO-level award for ...".

2.7. Gaps and Other Issues

There are times when your job history may not be consistent. It's important to clarify what there are discrepancies in your resume.

2.7.1. Short-Term Jobs

If you stayed at a job for only a matter of months, consider eliminating it from your resume. If it is brought up in the interview, simply be honest. Of course, if the job had relevant experience for your current job prospect, then by all means, include it.

2.7.2. Dealing with Gaps

If you have gaps of a few months in your work history, don't list the usual start and end dates for each position; indicating just the years is fine. Honestly, I just put years on all my jobs, regardless of how long or short they are, mostly because it's easier to remember the years I worked at a place than the month I started or ended.

2.7.3. Explain Job Hopping

If you've job-hopped frequently, include a reason for leaving next to each position. Job hopping can indicate to an employer that you are never satisfied with your position and are continually looking for something better; therefore, the odds of you staying with the company are low. However, if you indicate that your job hopping was due to contract work, moving due to a spouse being in the military, etc., then it won't adversely affect you because it was due to situations out of your control.

2.7.4. Explain a Long Break in Jobs

If you have a long break in work, then consider including a summary statement at the top of the resume demonstrating your best skills and accomplishments. In the work history section, don't be afraid to talk about any part-time work, volunteer work, or anything else that demonstrates that you didn't simply "check out of life" for a period of time and sat on the couch all day.

2.8. Finishing Touches

Once you've drafted your resume, make sure you give a couple of reviews, and let someone else look it over to make sure you haven't missed anything.

2.8.1. "References Available Upon Request"

If a hiring manager is interested in you, you will be asked to provided references, if they are necessary. Not every job requires references, and not every hiring manager will bother checking references, so don't provide information that isn't necessary. However, do have a list of references in case you're asked for it during the interview, and also confirm with your references that they are willing to help you out.

2.8.2. Proofread, Proofread, Proofread

Make sure your resume is clear of typos. Don't rely on spell check and grammar check alone, as most office suites are lacking in their abilities. However, I will make a note that, if you're using Google Docs or have a Google plugin, it will check your spelling and grammar against Internet resources, rather than the built-in dictionary, and it is far more accurate.

Finally, ask family or friends to take a look at it for you, especially if they are managers or in your field. Having another pair of eyes looking at your resume is helpful because they can see things that were invisible to you because you've been looking at it too long.

2.8.3. Use the Correct File Type

If emailing your resume, make sure to always send a PDF rather than a Word file to ensure your formatting isn't corrupted. However, if the company requests the resume in a particular format, then make sure you use the format requested. Nothing gets your resume trashed faster than failing to follow instructions.

2.8.4. Name Your File Smartly

Make sure you save your file with a useful filename, such as "JSmith_resume.pdf". That makes it easier to locate it if the resume is removed from the application package or sent to someone else separately.

2.8.5. Living Document

At least once a year, but preferably every quarter, update your resume with your latest accomplishments. This goes back to what I said about using LinkedIn as a "living resume". All I do is keep LinkedIn updated and, when it comes time to

create a resume, it is very easy to copy and paste the relevant blocks and modify as needed, rather than having to recreate from scratch every time.

If you can't remember to do that, then keep your quarterly/annual evaluations. Those should have a fairly detailed record of your accomplishments, allowing you to pick and choose the most relevant items for inclusion in your resume.

In the next chapter, we will talk about what to do after you have had an interview.

3. After the Interview

How you handle the job hunting process after interviewing with a company can play a part in whether you get the job. Being too needing and constantly calling for a status is a turn off, and will most likely result in your application being moved to the side.

In this chapter, we will look at some of the things that you should do to maintain a good relationship with the company, including:

- How to follow-up
- What to include in the follow-up
- Tips for emails and letters
- Tips for phone calls
- Final thoughts

3.1. How to follow-up

After your interview, it's important to follow-up the next day by contacting the interviewer(s) and thank them for meeting with you. One reason for this, besides common courtesy, is to provide you with an opportunity to reinforce your qualifications for the job, demonstrate your enthusiasm, and provide any important information you or the interviewer(s) may have forgotten during the interview.

Because you may meet with a number of people during the interview process, make sure you collect everyone's business cards, or at least get their contact information. Don't forget that LinkedIn and company web sites may have that information as well.

In the worst case, you can always call the company and ask for that information. (On a side note, if you're applying for an information security position, this phone call can be one way to determine how well the employees are trained on social engineering tactics.)

3.2. What to include

When you send your thank-you note, consider whether email or a physical letter is more appropriate. While email is immediate, it can get lost in the sea of other messages in the person's inbox. A physical letter is more personal, but does take time to arrive, potentially after they have made a selection.

In my personal opinion, email is probably the best but, if it's a senior level position, I would do both. That way, you have the immediate response of email with the personal touch of a physical letter and, with a senior position, chances are your letter will arrive before they have made a selection.

In your message, make sure you restate your interest in the job and the company, reinforcing the relevant skills per the job's requirements. If you want to

add anything you didn't get a chance to mention during the actual interview, now is the time.

In addition, if you misspoke or otherwise messed up during the interview, the thank-you letter is a chance to rephrase your response and clarify what you actually wanted to say.

Don't forget to include your contact information. You can never have too many locations for this: resume, application, cover letter, thank-you letter, etc. It just makes it easier for someone to reach you if it is on every document you have provided.

3.3. Tips for emails and letters

First, a caveat. Some resume professionals say thank-you letters are not necessary anymore, mostly due to the amount of work business professionals have nowadays; sending a thank-you is just one more thing they have to deal with, especially if there were a lot of candidates interviewing. But it doesn't necessarily hurt you to do it and it might make the difference if you really want the job. You can always ask around to see if it's useful for the job you're applying for.

If you do send one, send it within 24 hours of the interview. That way, you name is fresh on everyone's mind and you beat out anyone who doesn't follow-up.

Make sure you send a personal response to everyone you talked to. If possible, take some notes on what you talked about during the day so you can personalize the message even more, especially if there were topics you want to cover.

When sending a letter, hand-write it. It makes it that much more personal, as few people write letters anymore and, in some instances, people don't even know how to write cursive. However, if you're handwriting is poor, a typed letter is adequate as well.

Regardless of the method, proofread your letter before sending it. Computers make it very easy to spellcheck and grammar check your messages, so use them. Make sure to spell people's name correctly; if in doubt, and you don't have a business card, check the company's web site.

3.4. Tips for phone calls

While it is easier and potentially less nerve wracking to send an email, calling someone to thank them for the interview is a good touch. It demonstrates your ability to communicate over the phone, allowing you to be more of yourself as opposed to the stress situation of an interview. In addition, there is immediate feedback, so if there are any questions or other information you want to provide, you don't have to wait for an email response.

When calling, always use your full name and remind the person what position you interviewed for, as well when you met. Remember, they are potentially interviewing dozens of candidates, so they will be hard pressed to remember one out of the crowd.

Make sure you use proper language, i.e. no slang, and keep the conversation professional. How you present yourself on the phone should represent how you would talk to a client.

If you end up being sent to voice mail, simply provide the same information you would if you had reached the individual: your name, the position interviewed and when you interviewed, your contact information, and reinforce the fact that they can contact you if they have any questions.

3.5. Final thoughts

After you have reached back to the company, don't rest. You need to contact your references (if you provided any) and alert them to the fact that they may receive a call from the company you interviewed with. Provide your references with a brief overview of the company and the position applied for so they are adequately prepared to provide the best possible reference for you.

Finally, keep looking. Just because you applied with one company doesn't mean you're guaranteed to get the job. Many times people have been told a job is waiting for them but, come the day to sign paperwork, the offer is rescinded, as happened to myself. Always have other positions in mind and keep sending out your resume. Keep your network of contacts up-to-date and, if necessary, inform them of your status, especially if you lose a job offer. You never know where your next job will come from.

Next chapter, we will take a look at data structures and algorithms, focusing on common problem solving techniques.

Part II.

Programming Review

4. Data Structures and Algorithms

Problem solving is the core of programming. Computer science is designed to teach users how to make computers do things, and software is the tool. Computer Science tends to focus on creating new theories, constructs, paradigms, etc.; computer scientists are the ones who develop new programming languages, artificial intelligence and machine learning, new files systems, etc.

Software engineers take what the scientists develop and apply engineering principles and best-practices to create practical applications. Software engineers aren't necessarily expected to know the same algorithms and mathematics as CS graduates, but more knowledge about how to program applications for an enterprise/corporate environment. Naturally, there is some overlap, but those are the basic ideas between the two fields.

In addition, few self-taught programmers will learn "responsible programming", best-practices, and other things that come from software engineering. As a matter of fact, from what I've learned over the past few years, many CS graduates don't learn these ideas either. For example, the host of a Python podcast (https://testandcode.com) mentioned that he never learned automated testing in his CS degree; he had to teach himself, after he was finally convinced that it was a good thing to do.

I see this also in my current job. We are responsible for enhancing an open-source project with a custom application and the open-source code has few in-code comments, no unit tests, and is nearly impossible to navigate in terms of logic flow. Variables and methods with the same names are common between different modules, and inherited classes with the same name are common as well. Thus, you may have three different modules that could provide a certain functionality because a class is using an inherited method with a "reused" name.

I mention all of this to tie it back to my main point. Computer science, software engineering, and programming in general all work to solve problems using computers. They do it in different ways using different tools and methodologies, and that can be important when it comes to interviews. Depending on the work, you may not need a complete CS background; a self-taught programmer is just as good at making web sites as a CS graduate, and possibly better. Yet interviewers will want to go through programming paradigms, algorithm construction, etc. with every candidate.

That's why, even if you have a strong portfolio, you need to be prepared to go through the same steps as everyone else. Hence, this book will cover topics, such as this chapter, that instill general principles from computer science, software engineering, and other programming areas so at least you have a chance if you're stuck in an interview.

In this chapter, we will cover:

- What are problem solving paradigms?
- What is problem modeling?
- What is complexity analysis?
- How to solve for Big-O?

4.1. What are problem solving paradigms?

As might be expected, there are several different ways to tackle solution development for programming problems. As might also be expected, while the different methods may work for different problems, certain types of problems lend themselves better to particular solution paradigms.

4.1.1. What is an imperative paradigm?

Imperative programming assumes a computer can maintain changes in a computation process via environment variables. The computation is performed through a series of steps, which reference or modify the variables. The step sequence is critical, as a particular step will provide differing results based on the current values of variables.

In other words, imperative programming uses statements to read and modify a program's state using variables. It focuses on how a program operates. Many popular programming languages use imperative design, partly because the imperative paradigm closely resembles the actual machine itself and because the imperative design was most efficient for widespread use for many decades.

Some examples of imperative languages include C, BASIC, ADA, FORTRAN, and COBOL. The imperative paradigm also includes object-oriented programming, but we will cover that as a separate section.

Procedural programming is also a type of imperative programming. However, procedural programming focuses on the use of subroutines (including functions), so the step series is less important since the subroutines can change the code flow arbitrarily.

Advantages of imperative programming include:

- Efficient
- Close to machine language
- Popular paradigm for many system logic languages
- Like above, it is a familiar paradigm to many programmers

Disadvantages include:

- Program semantics may be complex to understand, due to side effects
- Side effects also make debugging harder
- Limited abstraction compared to other languages
- Sequence order is crucial, so not all problems can be adequately defined

Figure 4.1.1 is an example of imperative programming.In the example above, we create a function and then print the results of the function. In the function, we have explicitly told the computer how to calculate the results: create an empty list, iterate through the input array and double each value, then append the doubled value to the list, then finally return the filled list.

In the next section, we will see an alternative way to accomplish this same task.

Figure 4.1.1.: Imperative programming example

4.1.2. What is a declarative paradigm?

Declarative programming focuses on *what* the program must do to accomplish its goals, rather than the *how*; the how is left to the actual language implementation. That means a declarative program defines the logic of the computations without describing the flow control.

Many languages that fall under declarative programming attempt to minimize or eliminate side effects. Side effects are the modification of a variable outside of its local environment; in other words, a separate effect is noted besides the return of a value to the invoking operation.

Declarative programming often treats programs like formal logic theories and the computations of the program as deductions in the logic space. This can help when writing parallel programs, and is also common in databases, templating, and configuration management.

Common declarative languages include SQL, Prolog, regular expressions, XSLT, and Make.

Advantages of declarative programming:

- Closer to natural English
- Abstraction reduces boilerplate code
- Code is easier to reuse for different purposes
- Easy to halt at first error rather than check for each condition

Disadvantages include:

- Domain-specific languages can increase program size, such as template systems
- Lack of structural familiarity between languages. For example, C is similar to C++, Java, and even Python (to an extent). Try to compare SQL to Prolog or regular expressions and you'll see that they look nothing alike.
- More complex than imperative languages

Figure 4.1.2 is an example of declarative programming. You'll note that this fundamentally the same program we previously saw in the imperative programming section: doubling a list of numbers. However, instead of explicitly telling Python *how* to process the actions, we simply told the interpreter *what* to do and the actions were handled behind the scenes.

Figure 4.1.2.: Declarative programming example

In this case, we created a simple function that simply multiplies a number by 2. Then we declared a list of numbers. A new variable captures the output of the *map()* function, which mapped each number in the list as input to the doubling function. Finally, we print the result of the mapping to the screen.

At no point did we tell the computer how to produce the output, only what the input numbers are and what action should occur to the numbers. We don't know how *map()* is implemented in its underlying C code; we just know the API call required to use it and collect the output.

4.1.3. What is a functional paradigm?

While functional programming technically falls within the declarative paradigm, there are enough functional languages to count as a separate paradigm. Functional programs treat computation as a mathematical function, avoiding changing states and mutating data. In addition, output of a function is solely dependent upon the input; calling a function multiple times will result in the same output every time.

While most common in academic-oriented languages, commercial languages include Scheme, Erlang, OCaml, Haskell, and F#. Some aspects of functional programming are found in other languages, including R, JavaScript, Perl, Python, and SQL.

Advantages of functional programming include:

- Lack of side effects
- Elimination of many error types
- Suitable for parallel processing

Disadvantages include:

- Less efficient
- Many variables or significant sequential operations can be harder to deal with

We saw one example of functional programming in the previous sections. When we created a list of input numbers, we never modified that particular list. The only thing that changed was the list of output numbers. If you'd want functions to be purely functional, then don't change the value of the input values or any data that exists outside the function's scope, i.e. global values.

Immutability is also a desired aspect of functional programming, as it helps to eliminate errors caused by side effects or inadvertently changing a variable. In Python, tuples can be used to provide this effect without requiring any additional work. Thus, if you want to ensure your values don't change accidentally, store them in a tuple.

Another aspect of functional programming that Python incorporates is using functions as arguments or return values. In functional programming, this is known as higher order functions. Figure 4.1.3 shows one way of doing this.

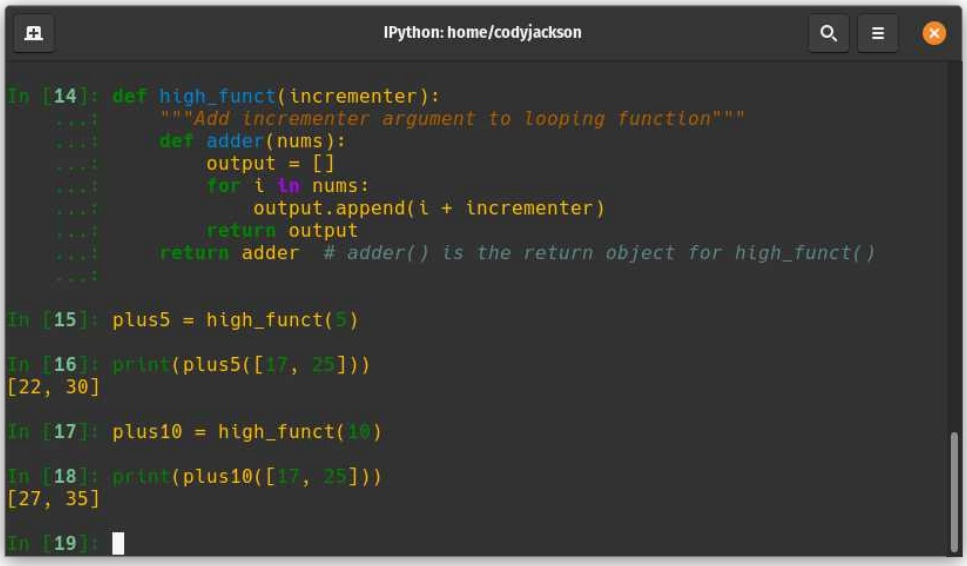

Figure 4.1.3.: Functional programming example

First, we create the higher order function *high_funct()*, which takes a number to pass to the inner function. While we only passed integers in the example, it will also work with floats. We don't have any input validation, so anything besides numbers will error out.

The nested, inner function is a simple iterator that adds the values in a list to the incremented value that was provided to *high_funct()*. The output values are stored in a separate list that is the return value for the inner function, and the inner function is the return value for the outer function.

In this way, you could easily create a program that performs the same actions every time, but change the input values. Also, note that this is similar to how Python decorators work. Decorators allow you to wrap one function within another function to extend the behavior of wrapped function without permanently modifying it.

4.1.4. What is an object-oriented paradigm?

Object-oriented programming (OOP) is based on objects that contain data attributes and methods that provide functionality to the objects. A common feature is that methods can access and modify attributes of the object they are associated with, and multiple instances of an object can exist simultaneously.

Most OOP languages use classes for the object containers; the other main type uses prototypes. JavaScript and Lua are examples of prototype-based OOP while Python, C++, C#, Java, et al. are examples of class-based languages.

Advantages include:

- Code reuse is encouraged through inheritance
- Multiple instances can exist at the same time
- Inheritance also encourages code extending rather than rewriting
- An object's interface is separate from its implementation

Disadvantages include:

- More complex than imperative languages
- Access to methods is typically only through an interface, which can cause problems if the interface doesn't provide a necessary connection.
- OOP can hide the transparency of a program's logic by shifting focus from data structures and algorithms to data types.

Figure 4.1.4 is a short example of OOP in Python. This program was kept intentionally short, but it shows most of the parts of the OOP paradigm. We create the class and define a method. In this case, we utilize the built-in method name $\_\_init\_\_()$, which automatically initializes instances of the class with any input arguments or other variables that need to be immediately available.

Figure 4.1.4.: Object-oriented programming example

Once the class is complete, we create an instance of the class in line 27. Part of the creation of the class is initializing the real and imaginary numbers for the complex number, so we pass in the real and imaginary parts for the instance. The class automatically assigns the arguments to this particular instance, so when we call the attributes for the numbers in line 28, Python knows what we are referring to and provides the corresponding information.

4.1.5. What is Python's paradigm?

As Python is the focus of code in this book, we will consider how Python fits into these paradigms. Python is an interesting language. At its core, it is an object-oriented; nearly everything within Python is an object and has a lot of hidden functionality tied to the object's class. As such, Python natively supports OOP using classes. However, Python can also be used as a traditional imperative language using functions. It also has features from functional programming, though it is not, by design, a functional language. In short, you can use Python

almost any way you like. It is not uncommon for a developer to start out using functions then, during refactoring, switch to OOP and convert the functions to methods. In fact, you can mix paradigms within the same program, incorporating classes with functions as desired.

4.2. What is problem modeling?

Since the purpose of programming is to solve problems, it's important to understand problem modeling. There are a number of different modeling paradigms available, depending on the type of problem and the desired outcome. For example, researchers may use ANOVA (analysis of variance) to determine the statistical differences between group averages in a sample, or could use t-tests if only two groups are considered. We won't go that far into models, but we will cover a general problem solving algorithm and discuss some of the commonly used problem solving models.

4.2.1. What is the process of problem solving?

When working with problems, it helps to have an established methodology. The methodology ensures a standardized process that is easily shared among collaborators and can help break down complex problems into smaller, easier to process pieces. The heart of project management as it applies to the software life cycle is defining problems, whether from internal or external customers, and then developing solutions to address the problems.

In this section, we will go through a six-step process to show one example how a problem solving algorithm can be standardized.

1. Define the problem
2. Determine the root cause(s)
3. Develop alternative solutions
4. Select a solution
5. Implement the solution
6. Evaluate the results

4.2.1.1. Define the problem

From a project management perspective, a project is, essentially, the work effort expended, within a definite length of time, to address a stated problem. The most important part of project management is the planning stage, including defining the problem, as without knowing what the results should be, you have no idea how to proceed.

Defining the problem includes knowing the context and background of the problem, any ancillary concerns, and what the project outcome is designed to accomplish. Recognize that multiple projects may be necessary to completely address the problem set, with each project addressing a particular aspect of the overall problem.

Various techniques can be used at this point, such as interviewing stakeholders, talking to subject-matter experts, and reviewing any relevant documentation. Once all the data is gathered and analyzed, the problem to be solved can be determined.

4.2.1.2. Determine the root cause

Combining all available information helps establish what the stated problem is, and realize that the actual problem may be different from the perceived problem. Understand that, often, the entity stating the problem may only be describing a symptom of the problem, not the actual problem that needs to be addressed. For example, if users are complaining that a web site is slow, the root cause could be multiple things: poor database design, improper MVC (model-view-controller) design in the web framework, network latency, or even poorly written client-side JavaScript that generates the dynamic web page.

Several different tools are available to assist with this step. Obviously, they can be used separately, but it is often most effective to use several tools to identify the true problem(s). We won't go into details of the different tools, but here is a short list of possible tools to consider:

- Fishbone diagrams: a diagram of major and minor items that affect a problem (see section 4.2.2.1 below).
- Pareto analysis: commonly known as the "80/20" rule (20% of causes determine 80% of problems), a person estimates the benefit provided by each action, then selects a number of the most effective actions that deliver a total benefit reasonably close to the maximum possible. In essence, Pareto analysis helps to identify the primary causes that need to be addressed to resolve the majority of problems.
- Affinity diagrams: used to organize ideas and data, the ideas and information are placed onto individual notes (whether physical or virtual), then sort the ideas that seem similar. This will result in a natural selection of ideas regarding a problem. The main groups that are created can then be analyzed again to create subgroups. Once the ideas have been collected into desired groupings, cause and effect analysis can utilize the ideas as input.
- Interrelationship diagrams: similar to fishbone diagrams, interrelationship diagrams display all interrelated cause and effect relationships, as well as the factors involved in a problem, and describes desired outcomes. The diagram helps to analyze the natural links between different aspects of a complex situation.
- Prioritization matrix: used to prioritize items and describe them in terms of weighted criteria. Utilization of both tree and matrix diagramming techniques create a pair-wise evaluation of items, helping to narrow down options to the most desired or most effective.

4.2.1.3. Develop alternative solutions

This can be a fun step, as it allows participants to brainstorm solutions. At this point, any possible solution should be considered, no matter how outrageous. However, each suggested solution should be looked at in terms of how it relates to the root cause of the problem. If possible, several suggestions should be merged to create a better answer to the problem.

4.2.1.4. Select a solution

After all the options are listed, it's time to pick one. Two main questions are asked:

- Which solution is most feasible?
- Which solution is most favored by those who will have to accept and implement it?

Feasibility can be quantified using things like time-to-implement, costs, return on investment, etc. Favorability is less quantifiable, as stated solutions may sound too radical or complex to the users or management.

4.2.1.5. Implement a solution

Once a solution has been decided upon, it has to be implemented. In project management, this is the longest part of the timeline. While finding the solution may or may not be part of the actual project, depending on different views, solution implementation is the core part of project management.

This is also the point where programmers get involved. Nearly everything prior to this does not involve programmers, though a team lead or other senior programmer should be involved to provide input regarding feasibility and timelines.

4.2.1.6. Evaluate the outcome

Part of project management is monitoring implementation to ensure the project is on-track towards completion while meeting all goals. In addition, after the project is completed, part of the software life cycle is to monitor the software and sustain it, meaning bugs are identified and patched, new features are added, unnecessary or obsolete features are deprecated, etc.

Very few, if any, programmers are fully satisfied with their work. Nearly everyone can think of a better way to do something after the they have completed a project, especially as they learn more about programming.

Below are some examples of thoughts after completing a project:

- Creating class instances for objects. A better, and more realistic, solution would be to create a database which will track the parameters, since databases are a natural choice when it comes to tracking parameters that frequently change.
- Brute-force solutions are fine for initial creation, but time needs to be spent refactoring the code into more elegant and efficient solutions. However, don't waste time trying to devise the optimum algorithm initially, especially if you aren't sure if the problem it addresses is even a problem.
- Unit tests are great and real time savers when it comes to refactoring. However, don't believe your results if things fail. Sometimes the tests requested the wrong data even though the actual simulation code was correct.

  In other words, if a program is ran manually with the correct input, you would get the expected output. But sometimes a unit test would use incorrect data input, resulting in failed tests. I found this to occur when I would create a class instance and then try to use that instance for multiple tests, such as attempting to toggle a variable's state between True and

False. Rerunning a test wouldn't always get new values, so errors would be generated, even though the actual code was fine.

This ties back to the first bullet: replacing class instances with a database. Had a database been used to hold object data, committing the changes to the database would ensure that a particular instance was being called multiple times and using incorrect values.

This type of analysis is useful as input to another project, such as the next revision of an application. This ties into the life cycle management, as the software has to be maintained until it is removed. When interviewing, you may not be asked about project management and software life cycle specifically, but general questions could be asked, as well as questions that relate to particular problem solving steps.

4.2.2. What are some problem solving models?

A common issue when solving problems is constraints. There may be conflicting objectives, time pressures, or just general "noise" when trying to figure out critical aspects. In addition, there is the Project Management Triangle, which essentially states, your solution can include any two of the following options: fast, good, and cheap. This applies to many different areas, and software development is no different. It isn't hard to identify fast and cheap programs that aren't any good, just like it isn't difficult to find software that is high-quality and efficient but is quite expensive.

When solving problems, there are several models available:

- Cause and effect: frequently used when multiple problems are involved, this helps identify any correlations and causations between problems, as well as helping identify solutions that have the biggest impact in problem resolution.
- Risk management: helps identify and assess any factors that play a factor in the success of a proposed solution.
- Brainstorming: also called "creative problem solving", formalized and structured brainstorming includes checklists, forced relationship analysis, structured relationship sets, and similar tools.
- DMAIC: derived from Six Sigma, this stands for Define, Measure, Analyze, Improve, and Control; this model helps improve processes by finding the cause of a problem and establishes a methodology for correction and improvement.

Other models are available, but regardless of the model selected, adherence to its process will ensure a successful outcome. When people ignore the process or the results don't align with their views, that's when solutions won't be resolved.

We will cover examples of the models listed above in the following sections.

4.2.2.1. Cause and effect modeling

The cause and effect problem solving model has been around since the 1960s, when it was created by Kaoru Ishikawa. The drawings utilized are known as Ishikawa diagrams, or fishbone diagrams due to the completed shape. You can

use this method to help identify the root cause of problems, find bottlenecks in a process, and figure out where and why a process isn't working.

Figure 4.2.1 is an empty version of a fishbone diagram, followed by the steps to create one.

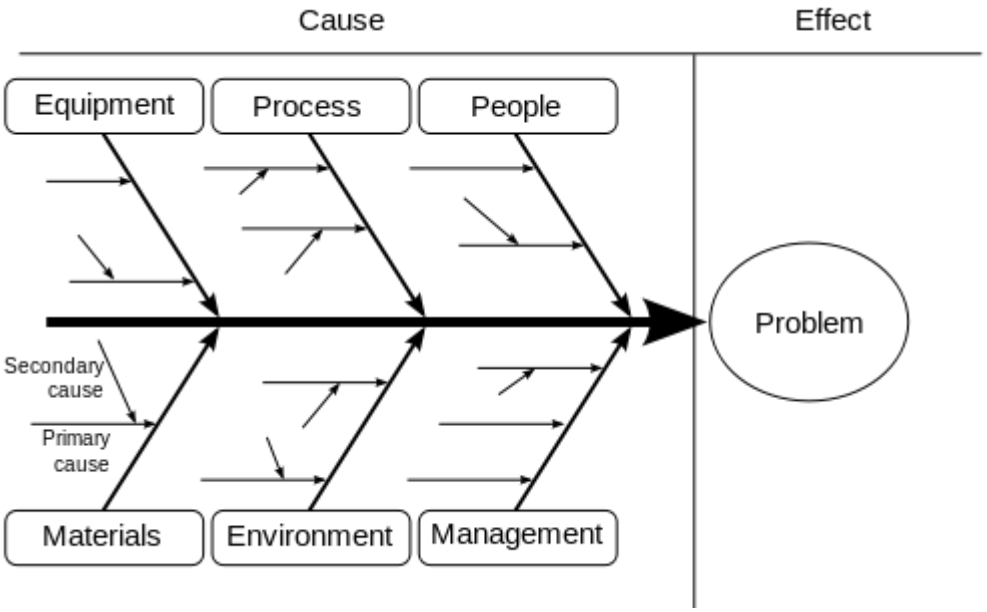

Figure 4.2.1.: Fishbone diagram

1. Identify the specific problem and, if possible, who is involved as well as when and where it occurs. This problem will become the root of the diagram, with a line extending horizontally to list ideas that contribute to the problem. To help define the problem, consider the problem from the perspective of customers, actors in the process, the transformation process, the overall world view, the process owner, and any environmental constraints.

2. Identify the major factors that may be part of the problem, such as systems, equipment, materials, external forces, management, etc. Include as many possibilities as makes sense, as there may be linkages between them that aren't readily apparent. These main factors will branch off the "spine" of the fishbone to hold individual factors.

3. For each primary factor, identify possible and probable causes that contribute to the primary factor. Label these primary causes as smaller lines coming off the primary factor lines.

4. If there are any secondary causes that affect the primary causes, list those as smaller lines connected to the primary cause lines. Thus, you should end up with a number of "ribs" coming off the spine of the diagram, with each rib further splitting into smaller branches of separate primary and secondary causes.

5. Once you have listed all the possible causes, you can start to consider which ones are most likely to be causing the overall problem. Further investigation may be required at this point, such as employee surveys, on-site inspections, etc. to ferret out the true causes, but with the diagram, it is easy to see which causes are unlikely and which are more likely to affect the problem.

4.2.2.2. Risk management

Risk management (RM) strives to minimize the number of surprises that will occur during a project. The short form concept is simply to identify any and all potential risks that may occur during the project, determine how they will affect the project, then develop solutions to counteract those risks, if necessary. While everyone performs RM, whether consciously or unconsciously, without a structured process, it is very easy to miss possible risks, misinterpret the impact of the risks, and prevent communication and transparency between team members and stakeholders.

The process to handle risk management is shown in figure 4.2.2. Risk management is an iterative and continuous process, conducted prior to, and during, project completion.

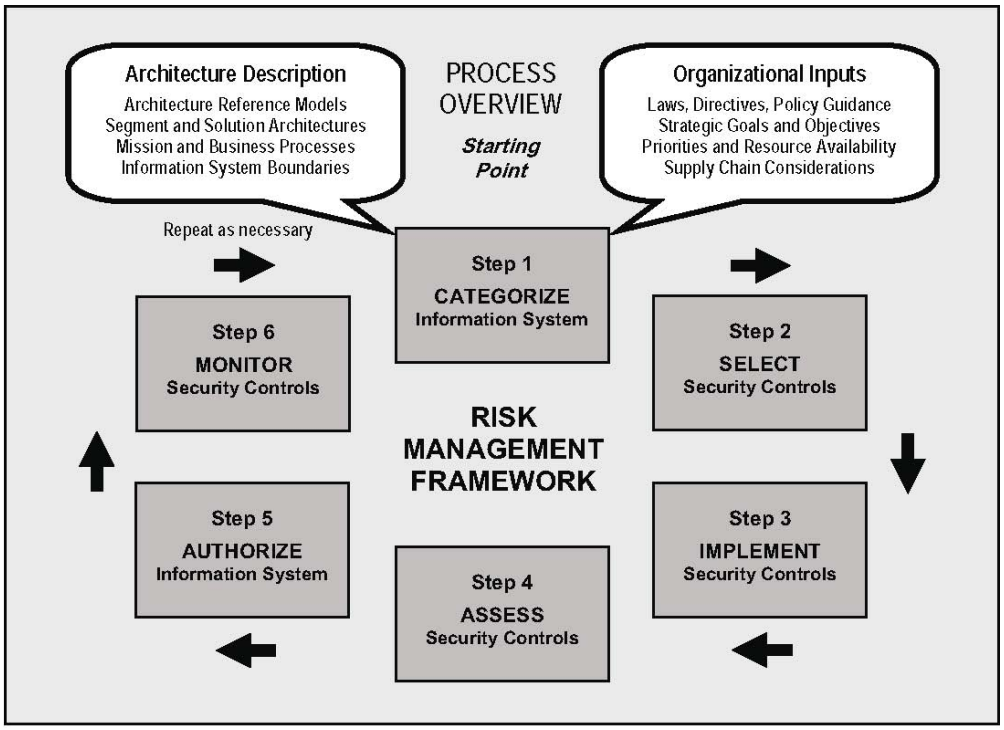

Figure 4.2.2.: Risk management process

Risk management plan An overall plan is drafted and approved during the project planning phase. The plan will identify things such as:

- Potential risk sources and categories
- Risk impact and probability
- Risk reduction and actions
- Contingencies
- Risk threshold and metrics

While an organization may have a generic RM plan template, it will be customized for each project due to the different circumstances possible in each project.

Risk analysis Risks are identified and dealt with as early as possible in the project's timeline. Due to the continuous nature of the RM process, risk identification occurs throughout the entire project life cycle, particularly at milestones. This is due to the changing nature of projects and external influences, especially in agile environments where customer feedback of sprint results, for example, helps dictate the work for the next sprint.

Risk status is reported at regular meetings and recorded on some sort of tracker, such as a spreadsheet, database, or even whiteboard. When identifying risk during project planning, it helps to look at the risks identified during previous projects, as certain risks are frequently common in every project, or type of project.

In addition, it's not just leadership and project managers who should be identifying risks. Subject matter experts and other members of the project teams should feel free to report risks that they know of or identify during the project. They may know of risks that aren't obvious to the rest of the project team.

Part of risk identification is categorizing risks. Common categories include:

- Technical: requirements, technology, interfaces, quality, performance, etc.
- Organizational: logistics, resources, budget, dependencies, etc.
- Management: planning, scheduling, controlling, communication, etc.
- External: customers, contracts, suppliers, etc.

Risk evaluation After all the known or perceived risks are identified, they need to be analyzed for impact and probability. A common tool is a risk matrix, where a particular risk is given a probability of occurring and an impact rating for the project. An example risk matrix is shown in table 4.1

Probability → Impact ↓	High (>80%)	Med-High (60-80%)	Med-Low (30-60%)	Low (<30%)
Catastrophic				
Critical				
Marginal				

Table 4.1.: Risk matrix example

When filling out a matrix for risks, it's important to consider the following factors that may affect risks:

- Customer importance
- If the project is critical to future dealings with a customer
- The risk is already identified by the customer
- Certain contract agreements with a customer

It's important to note that, when the term "customer" is used, it doesn't necessarily mean external customers. Internal customers, such as the Sales Department or even the CEO, are just as important to satisfy.

Risk control Once the project risks have been identified and analyzed, plans need to be drafted to control for those risks. In order of priority, the following list gives possible risk controls:

1. Completely eliminate the risk
2. Lower the probability of occurrence
3. Lower the impact on the project

To help judge the most effective action to take, and normalize between all the risks, each risk response should include the time and financial cost associated with it.

For example, it may take four months and $200,000 to completely eliminate a risk that has a 15% chance of occurring, but is 100% "fatal" to the project if it occurs. However, an alternative may take only one month and $75,000 to change it to a 50% "fatality" rating, but increase the likelihood of occurrence to 35%. Which one is a better option? That's up to the stakeholders and project manager to decide, based on the time and money available for the entire project.

This also illustrates the fact that each risk factor may have different plans available for mitigation, and there may not be a definite best-option. For example, what if insurance was available to cover the cost of a 50% "failure" of the project? How does that change the risk plans provided above?

Part of risk control is identifying the risk owner. This is the person or people responsible for monitoring risk triggers (indications that a particular risk is occurring or about to occur), and implementing actions to respond to the risk. While the project manager is overall responsible for the completion of the project and dealing with risks, the risk owners are frequently best positioned to identify and counteract risks.

Finally, as each phase of a project is completed, reassessment of the risk table should occur, removing any risks that are no longer factors and adding new risks that may arise. Then the process starts again.

4.2.2.3. Brainstorming

Brainstorming is actually a fairly structured process to solve problems. The concept is traced back to the late 1930s and focuses on two main principles: defer judgment and reach for quantity. In essence, these principles mean that there are no bad ideas during a brainstorming session, and everyone should feel free to contribute as much as they can to the problem.

In addition, four general rules help establish the boundaries of a session:

- Go for quantity: the more ideas generated, the more likelihood of a great solution arising
- Withhold criticism: focus on extending ideas and withhold judgment until later
- Welcome wild ideas: look at the problem with a new perspective and avoid assumptions
- Combine and improve ideas: no one idea may be perfect, but combining multiple ideas may get closer

Figure 4.2.3 demonstrates one way to streamline a brainstorming session. The diagram includes an introductory portion for inexperienced participants, then moves into the problem explanation. Note that this is a single problem, as brainstorming rarely works when multiple problems are attempted at one time. After the problem is presented, ideas are presented (which can be called out or move from person to person). Eventually, the best ideas are selected and improved upon until a final solution is derived.

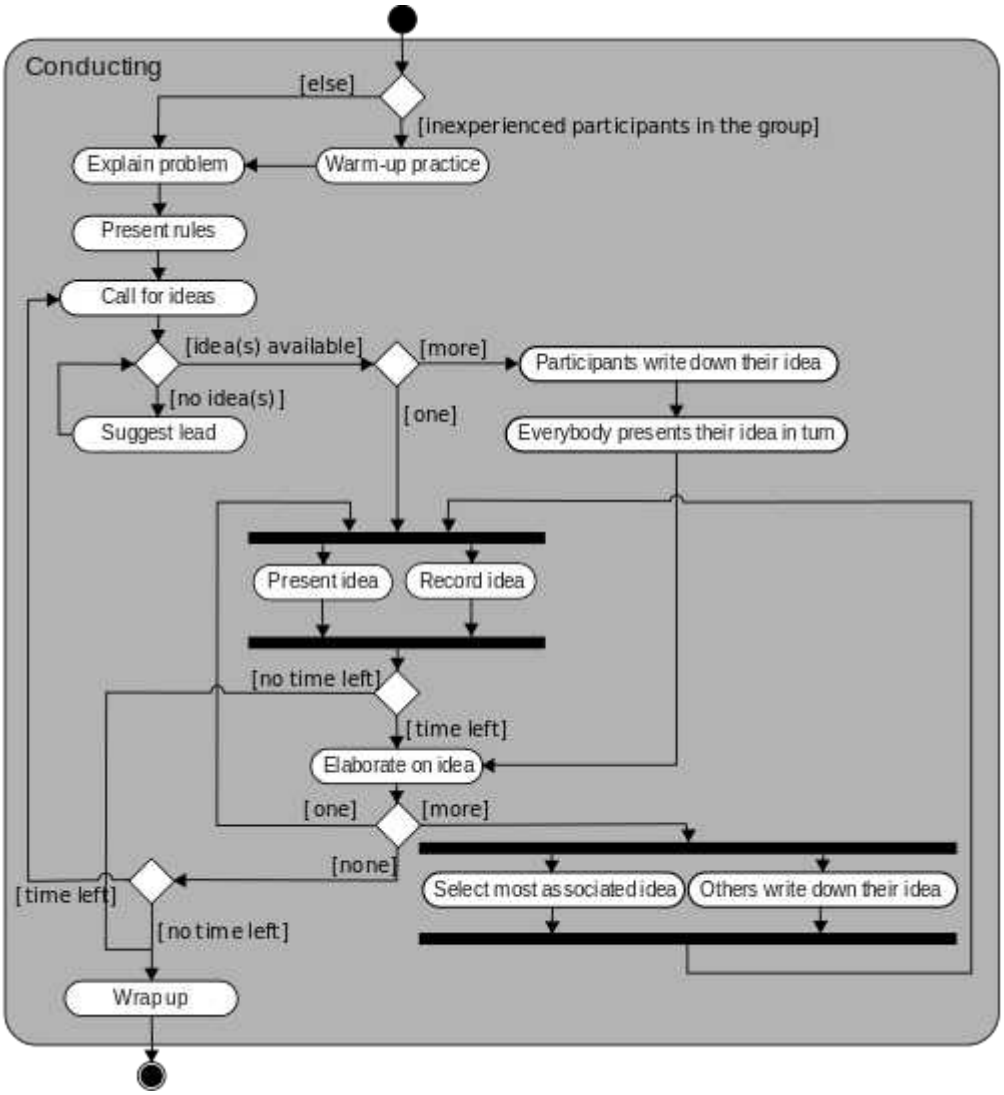

Figure 4.2.3.: Brainstorming example (Gwaur [CC BY-SA])

4.2.2.4. DMAIC

DMAIC (Define, Measure, Analyze, Improve, and Control) is an improvement framework commonly found in Six Sigma projects, but can be applied elsewhere. The DMAIC process is sequential and requires all steps to be followed, as listed below:

1. Define: Clearly identify the problem, desired outcome, resources available, scope, and projected timeline. This information is normally written in the project charter document. Define the following: problem, customer, voice of the customer (feedback), and critical process outputs.

2. Measure: Establish current baselines as the benchmarks for improvement. The baselines will be compared to the performance metric at the end of the project to determine whether significant improvement was accomplished. The team decides what should be measured and how.

3. Analyze: Identify, validate, and select the root cause of the problem. While a large number of root causes may be identified initially, perhaps via a fishbone diagram, the top three or so should be selected through some type of consensus tool, such as multi-voting. A data collection plan is created and data collected to establish the relative contribution each root cause has on the project metric. The process is continued until a valid root cause is found.

4. Improve: Identify, test, and implement a solution, whether in part or in whole, depending on the situation. Attempt to eliminate the root cause, if possible, or at least mitigate its effects. Rather than actually implementing a solution, this step can simply find possible solutions instead.

5. Control: Embed changes and ensure sustainability of the solution. This includes modifying work methods, procedures, and policies; quantifying benefits; monitoring improvement and sustainment; officially closing the project; and approving release of resources.

While not part of the DMAIC process, it should be considered to replicate the changes in other process and sharing knowledge to others.

4.3. What is complexity analysis?

While it is perfectly acceptable to formulate a project plan and start writing code, or just jump to writing code as some self-taught programmers do, one of the things that a CS degree teaches is analyzing algorithm complexity. The reason is because some solutions are more elegant and efficient than others. This is especially important if you use a "roll your own" language like C, where you are responsible for using primitives to make all your own data structures.

Conversely, in a language like Python, all that work has been done for you; as long as you use the built-in tools from the language, you don't have to worry about the inherent efficiency of the underlying code. Of course, it is still possible to make "bad code" on your own, but at least you can't blame the language itself for being inefficient.

4.3.1. What is time and space analysis?

Complexity analysis determines how efficient a particular algorithm is, as algorithm efficiency is a key part of quick computations. The two main types of complexity measures are *time* and *space* analysis.

Time complexity refers to how long an algorithm takes to process as a function of its inputs. Time is a relative term here, as it can mean actual time in seconds to the number of memory accesses, the number of loop executions, or pretty much anything defined by the programmer. Normally, time is a measurement of a unit related to the amount of work the algorithm will perform, rather than actual time, as total time to completion can be independent of the actual algorithm.

Space complexity refers to the amount of memory an algorithm takes up in terms of the amount of input. This is, essentially, the overhead RAM required

to perform the operations beyond the memory required to store the input itself. Space can be measured in bytes, but more often uses things like the number of integers used, number of fixed-size data structures, etc. Space is less often used compared to time, as for most calculations, space is either minimal or obvious. In some instances, such as data science, space can be more important than time when large data sets are input.

4.3.2. What is asymptotic notation?

When analyzing complexity, functions are derived that calculate performance as n input items are increased. For example, say we had an complexity equation of $6n + 4$, indicating that we have an algorithm that increases linearly ($6n$), plus a constant value. In asymptotic notation, all we care about are any values related to n; specifically, we are interested in the worst-case n value. In the previous equation, that means we drop the 4 and the 6, keeping only n. Thus, our algorithm complexity is $f(n) = n$.

Now assume we calculated a complexity formula of $f(n) = n^2 + 4n - 3$. First, we drop constants to get $f(n) = n^2 + n$. Then, since we really only care about the worst-case scenario, we can drop the linearly changing n and keep only the quadratic value: $f(n) = n^2$. This is because the dropped values contribute very little when n is a large value.

4.3.3. What is Big-O?

Using asymptotic notation, we can figure out the Big-O value for an algorithm. Big-O is a formal expression of asymptotic upper bounds: a way of bounding, from above, the growth of a function. In other words, Big-O is a hierarchy of algorithm complexity, allowing quick comparison of algorithm performance; ignoring constants and letting n get big enough, we know that a given function will not exceed another function.

Table 4.2 is a table of common Big-O values, from fastest to slowest. Note that not all Big-O values are listed.

Notation	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Linearithmic
$O(n^2)$	Quadratic
$O(n^c)$	Polynomial
$O(c^n)$when $c>1$	Exponential
$O(n!)$	Factorial

Table 4.2.: Common Big-O values

To make it a little easier to see, figure 4.3.1 shows how the different Big-O notations compare to each other. It shows the number of operations (N) required vs. input size (n).

Of course, to be useful, you have to determine the Big-O notation of your algorithm. Fundamentally, this requires you to enumerate the different operations your code does and determine how they relate to your inputs; don't forget to

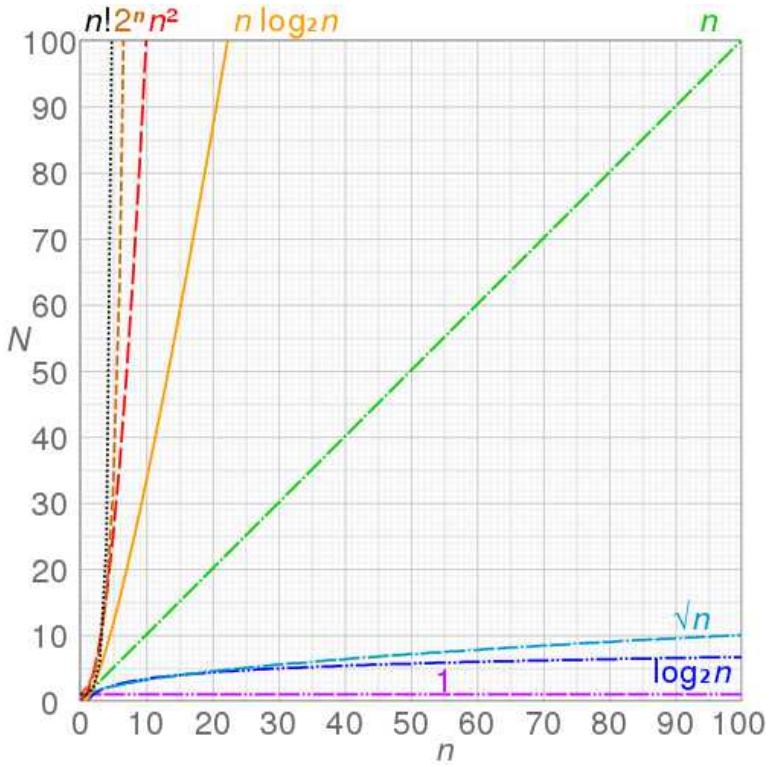

Figure 4.3.1.: Graphing Big-O (Cmglee/CC BY-SA)

account for external function calls. Once you have that information, you can formulate the results, drop constants, and look for the biggest n term.

A good cheat sheet for Big-O values for common data structures and sorting algorithms can be found at http://bigocheatsheet.com.

4.3.4. What are some other bounds?

In addition to Big-O upper bounds, lower bounds (Big-Ω) and tight bounds (Big-Θ) are available. While Big-O is for functions that grow faster than $f(n)$, Big-Omega (Ω) is for functions that grow slower than $f(n)$ and Big-Theta (Θ) is for functions that grow at about the same rate as $f(n)$; in other words, Big-Theta combines both Big-O and Big-Omega.

In layman terms, Big-O defines an algorithm with its, at worst, time-growth rate. Big-Omega defines it with its best-case time-growth rate. Big-Theta defines an algorithm's time growth as a function bounded by Big-O and Big-Omega, once n gets big enough.

Big-O is most commonly referenced, because worst-case scenarios are the goal of algorithm optimization. The other notations are used when more granularity is desired in describing program operations.

4.3.5. When to add or multiply multi-part algorithms?

When an algorithm has multiple steps, you need to account for those multiple steps by either adding or multiplying them in the Big-O formula. But how do you know which one to use?

The short answer is:

- If the algorithm requires the program to "do this and then that upon completion" (separate actions), add the components.
- If the algorithm requires the program to "do this every time you do that" (loops), multiply the components.

For example, you have the following code blocks:

```
for i in list1:
    print(i)
for j in list2:
    print(j)
```

In this case, we complete the first for loop before we move onto the second loop. Therefore, we add the times together to get $O(i + j)$.

We can do the same thing in the following example:

```
for i in list1:
    for j in list2:
        print(i, j)
```

In this case, we are performing the j loop in conjunction with the i loop, so we need to multiply them to get $O(i * j)$.

4.3.6. Why does complexity matter?

Apart from knowing you will be asked about it in an interview, knowing the complexity of your code can help you when trying to maximize efficiency and how to judge whether the code is making things better or worse.

4.3.6.1. Time complexity and file size

If we consider time complexity, think about a file transfer. For many people, they have asymmetric Internet connections, meaning their upload speeds are lower than than download speeds. For a while, many people backed up locally but, with cloud storage, it makes sense to have a secondary backup on a server somewhere.

For most file backups and small drive backups, it can take a few minutes to a few hours to perform a cloud backup. In this case, the complexity is given as $O(n)$, where n is the file size. In other words, and all things being equal, the time to transfer a file increases linearly as the file size gets bigger.

Now, consider a company. Even if they pay for symmetric Internet, there is still a limit to their upload speed. Companies have terabytes of data to backup. Even if a company paying for 1Gbps fiber is still going to be waiting hours for a normal daily backup, and even longer for weekly backups.

In this case, it is actually faster to back everything up to tape. Assuming you have LTO-8 magnetic tape drives, you can store up to 32TB per tape, at a rate of roughly 1GB/s. This is equivalent to nearly 9Gbps, so a data transfer rate of 9x faster than an Internet connection.

Now, if we fill up a truck with all these magnetic tapes and send them to an offsite storage facility, the time complexity is $O(1)$. That is, with respect to file size, the time to transfer all the backup data to storage is constant. Whether you're sending one tape or 400 tapes, it all gets there at the same time; there is no difference in time as the file sizes change.

What does that have to do with this chapter? If you know the complexity of your current algorithm, you will know whether or not it is worth the time to modify it. Maybe you can use a different solution to increase the speed, or maybe you can find a particular bottleneck to change the complexity from $O(n2)$ to $O(n \log n)$.

In regards to the example, if you tried to optimize your backup, such as improving compression to decrease the file size, that might make a difference when using the cloud, but it would do nothing in regards to shipping magnetic tapes. Therefore, there is little reason to spend resources trying to improve the compression algorithm if you're using tapes; you're already at the maximum efficiency.

This is pretty much how I program. I will brute-force a solution, as I want to get the logic down to generate the correct result. Once I've done that, I will go back and look at improving the efficiency of my code. I'll admit, the programs I write don't have Big-O calculations associated with them, but that's mostly because a lot of what I create is I/O dependent; it doesn't matter how efficient my algorithms are if they are waiting for data to process.

However, the concept is the same: do something, figure out its "cost" (time or space), and then determine where, and whether, to optimize. In planning sessions, the team can hash out suggested solutions based on what is know about the project and agree to a particular Big-O complexity. However, once programming starts and more is known about the details, it's alright to step back and reevaluate. Is the current Big-O value the best, based on what you now know? Is there a better solution that will provide a more optimized complexity? Where are the bottlenecks and how bad are they?

4.3.6.2. Revising code vs. rewriting code

Knowing how "bad" a bit of code is from a time complexity perspective is where Big-O shines. If you don't know what your current algorithm is producing (in terms of Big-O), then you don't know whether it is worth optimizing. It could be that you have a bottleneck that simply won't go away, e.g. I/O operations. Regardless of whether your algorithm is $O(n \log n)$ or $O(n)$, if the program is waiting for input, it can only function so fast.

On the other hand, if you are doing a lot of loops (which is common), it may be useful to determine the Big-O value for each code activity and find out where the slow down is. Perhaps as part of a $[O(\log n)]$ loop, it is searching within a $[O(n^2)]$ data structure. Because Big-O is concerned about worst-case scenarios, your overall complexity is $O(n^2)$.

If you wrote the code without thinking, you probably don't realize that there are a lot of different search algorithms that provide different benefits for different Big-O values. If you substituted a better algorithm, you could drop the searching portion to $O(n)$, thereby changing the overall complexity to $O(n)$.

Another thing to consider is that, sometimes, it may be better to completely revise the program to eliminate the bottleneck. For example, say you have a list of seven numbers, in random order, and you have to find the number pairs that give a difference of $k = 2$. The brute force method is to start at the first element and add/subtract 2 and compare to all the remaining numbers $(x_0, \pm k) = x_1$, then do it for the next element $(x_1, \pm k) = x_2$ and so on.

The bottleneck is repeatedly cycling through the remaining numbers each time. If the array was sorted first, then a binary search could be used to find the other half of the pair in $O(\log n)$ time.

Doing this creates a two-step process: sort the array, then find the difference. Since we have n elements within the list, the time is actually $O(n \log n)$ for each step; in other words, sorting becomes the new bottleneck.

To eliminate the bottleneck, we could simply rewrite the program to remove the need to sort the list first. This can be done using a hash table, which is an unsorted data structure optimized for quick searching. If the list was converted to a hash table, then the search could become a single step of finding the difference of values in the table.

Now, recognize that, with Python, a lot of this is not applicable. Python already optimizes a lot of functionality and provides for functions and methods that do a lot of this work for you, such as sorting a list or providing a hash table as a dictionary. But you still need to know it for your interview, so we will cover things like Big-O variations between examples in this book.

4.4. How to solve for Big-O?

When calculating the Big-O value for a program, it helps to consider each function separately. First, it allows you to breakdown each component separately and second, it helps to identify the bottlenecks.

4.4.1. Basic process

Assume we have the following code:

```
1 def count(arr_in):
2     total = 0
3     for element in arr_in:
4         if element == 5:
5             total += 1
6     return total
```

This is a simple incrementer that loops over a list. If the value of an element in the list is equal to five, the counter increases.

To find the Big-O value for the function, we consider each portion of it (excluding the definition line). The following list matches the line numbers in the example.

1. Ignore this line.

2. This line is $O(1)$, because setting a variable to a given value is constant time. We aren't setting the variable to another variable, or capturing the result of a function call. We are simply assigning the variable an integer value, so there is nothing that will change how long that takes, therefore it is constant time.

3. This line indicates that we are iterating over a sequence. Therefore, the time required is dependent upon how many elements (n) there are in the sequence, meaning this is $O(n)$ complexity.

4. This line is constant time, $O(1)$, because we are comparing whatever the list element is (assuming an integer) to a constant value (5). This is very similar to the first line, where we set a variable to a constant integer, except this is a comparison. There are no calculations involved, so the time to compare is constant.

5. Again, much like line 1, we are setting a variable to a constant integer. Though it is addition, there is not additional time required to complete the addition and set a variable; both time-to-complete operations are set by the CPU and operating system. Therefore, this line is $O(1)$.

6. Ignore this line.

Those are all the operations we need to consider. In this case, we have $O(1) + O(n) * [O(1) + O(1)]$; remember that, if an operation is sequential, it uses addition. If it is iterative for multiple elements (*for* loops), then it is multiplicative. Reducing it mathematically, we end up with $O(1) + O(2n)$.

Since we care about the worse case scenario, and $O(1)$ is the best case, we remove that term. And, since it doesn't matter what the coefficients are within these calculations (we only care about the approximation, and $2n$ is the same linear function as n, except the results are faster growing). Therefore, the final result for this function $O(n)$.

4.4.2. What is O(log n)?

If you look at the various Big-O values that are used, logarithmic values frequently show up. What does this exactly mean? The short answer is: if you see a problem where the number of elements is halved each cycle, you most likely have an $O(logn)$ type problem. The longer answer is demonstrated below.

Assume we conduct a binary search in a sorted array: [1, 3, 4, 8, 9, 12, 17, 23, 37, 45]. In a binary search, we split the array in half then look at the middle value. If it matches what we are searching for, we're done. If not, then we see if the value we're searching for is less than the median. If so, then we search the lower half of the array; if not, then we search the upper half. The splitting and searching continues until we've identified the location of the value.

The process looks like this:

1. Given [1, 3, 4, 8, 9, 12, 17, 23, 37, 45], is there an 8 in the list?

2. Split the list in half. For a list with an odd number of elements, the middle is easily found, as there will be an even number of elements in each half, leaving the middle value by itself. For a list with an even number of elements, we take the floor of *len(array)/2*. For our list, we have a length of 10, so the middle value is position 5, or the number 9.

3. We see if the checked value (8) is equal to the value of the middle element (9). It isn't, so we see if the checked value is greater than or less than the middle element. It is less, so we check the lower half of the list.

4. The new list is [1, 3, 4, 8]. The middle element is *len(list)/2* = 4/2 = 2. Therefore, we compare the checked value (8) to the second element (3). The checked value is not equal but is bigger, so we use the upper half of the list.

5. The new list is [4, 8]. The middle element is first position, so the value is 4. The checked value doesn't match but is greater, so we make a new list.

6. The new list is [8]. The middle element is the only element. The checked value matches the element's value, so we have found that yes, the checked value exists in the list.

It can be seen that, per the algorithm, regardless of the length of the array, it is perpetually being cut in half per search. It only stops when the value is found or we have only one element that doesn't match.

To determine the total runtime, we have to figure out how many divisions are required until we reach an element = 1. Given our example above, we have $N = 10$. Therefore, we have the following steps:

- $N = 10/2 = 5$
- $N = 5/2 = floor(2.5) = 2$
- $N = 2/2 = 1$

We end up with three division steps before we end up with $N = 1$. Now, we know that $x^k = N \equiv \log_x N = k$, and since we are talking computers, $x = 2$. In this instance, since $N = 10$ was the initial starting value, $\log_2 10 = 3.32$, which is rounded to 3 because we had three division steps.

But, for Big-O, we don't care what the base is since all constants are removed. That means the final outcome is $O(\log n)$. Hence, anytime you are dividing by two, you will have a $O(\log n)$ runtime.

4.4.3. What is Big-O for recursive functions?

Assume the following recursive function:

```
def funct(n):
    if n <= 1:
        return 1
    else:
        return funct(n - 1) + funct(n - 1)
```

If we consider it a branching tree, where n is the base and $n - 1$ is a branch, you can see that for each value n, there are two corresponding branch nodes that are one value less. Therefore, if $n = 4$, then we would have a total of 16 nodes split between all the branches. In other words, there are 2^n nodes for the tree.

Now, what is the complexity of this function? If the number of nodes is doubling for each new branch level, you would probably think it is something like $O(N^{n-1})$, where n is the initial number of recursive calls. Counterintuitively, the complexity for the entire function is actually $O(N)$, because at any given time, we only care about the value that is used as the argument into the function.

4.4.4. More Big-O examples

Below are selected examples of calculating the complexity for different functions.

1. Given the code example below, what is the runtime complexity?

   ```
   def funct(list_in):
       list_sum = 0
       product = 1
       for i in list_in:
           list_sum += list_in[i]
       for i in list_in:
           product *= list_in[i]
       print(f"sum {list_sum}; product {product}")
   ```

Because we are working with a sequence (*list_in*), the time factor is relative to the number of elements in the sequence. Therefore, this function is $O(n)$.

2. Consider the following nested loop:

```
def print_list_pairs(list_in):
    for i in list_in:
        for j in list_in:
            print(f"{i}:{j}")
```

Because both loops are using the same input list, the both run at $O(n)$ time. However, since the inner loop is called n times, that means the actual complexity is $O(n2)$.

3. Similar to the previous question, the following code only differs by using two input lists instead of one:

```
def print_double_lists(list1, list2):
        for i in list1:
                for j in list2:
                        if list1[i] < list2[j]:
                                print(f"{list1[i]}:{
                                    list2[j]}")
```

We can simplify this by assuming the *if* statement is time $O(1)$ because a comparison is constant time. That leaves just the two *for* loops. Since we have two lists to consider, this is not $O(n^2)$: for each element in *list1*, the inner loop has to go through j iterations in *list2*. Therefore, the complexity becomes $O(len(list1) * len(list2))$, or $O(ij)$.

4.5. Summary

In this chapter, we looked at a variety of tools and methods for solving problems. We covered four problem solving paradigms (imperative, declarative, functional, and object-oriented), a six-step problem solving approach (define the problem, determine the root cause(s), develop alternative solutions, select a solution, implement the solution, and evaluate the results), several different problem solving models (cause and effect, sensitivity analysis, risk analysis, brainstorming, and DMAIC), and how to use different complexity analysis tools to identify algorithm efficiency, particularly Big-O.

In the next chapter, we will look at Python's OOP paradigm and how Python utilizes different aspects of OOP to create programs.

5. Python Object-Oriented Programming

Though Python allows a developer to use functions, it also allows for object-oriented programming (OOP) development. This chapter will talk about the various aspects of OOP as defined by, and used with, Python programming.

Specifically, we will look at the following:

- How are classes defined and instantiated?
- What is inheritance?
- What is encapsulation?
- What is polymorphism?
- How do you extend and override classes?
- Interview-type questions

5.1. How are classes defined and instantiated?

Python uses classes for object-oriented programming. As it is entirely possible some readers of this book are unfamiliar with, or confused by, classes and OOP, I will provide a short analogy to clarify OOP principles. We will then use this analogy to create examples later on.

Assume you need to make a program that utilizes a number of vehicles. Traditionally, you would create each type of vehicle, such as a plane, a car, a truck, and a boat. Each vehicle would be a self-contained entity within the program, most likely with the same, or similar functions.

Using the OOP paradigm, we can create a variety of vehicles with little effort. For example, we could create a parent class *Vehicle*, that has attributes such as *speed*, *wheels*, *doors*, etc. It could also have methods such as *calculate_ acceleration()*, *calculate_ braking_ distance()*, etc.

From the *Vehicle* class, we can inherit these base parameters and create individual types of vehicles, each using the default parameters and either extending them with custom parameters per vehicle type, or overriding the inherited parameters, as necessary. Thus, from a single class, we can create a near infinite number of subclasses, each with their own special parameters while still utilizing the inherited parameters from the parent class.

With that in mind, let's actually implement some vehicle classes. First, we need to create the parent class the defines a generic vehicle. Naturally, we won't include all possible parameters, but some common features are shown in the following code example:

```
from math import pi

class Vehicle:
    """Generic land vehicle class"""
    def __init__(self, speed, rpm, wheels, doors, engine,
        weight, wheel_diam):
        self.vehicle_speed = speed # miles per hour
        self.wheel_rpm = rpm
        self.num_wheels = wheels
        self.num_doors = doors
        self.engine_size = engine # Liters
        self.vehicle_weight = weight # Pounds
        self.wheel_diam = wheel_diam # Inches
        self.stopping_distance = 0
        self.stopping_time = 0
        self.brake_power = 20 # feet/sec slowdown
        self.thinking_distance = 0  # time required for
            driver to react
        self.braking_distance = 0
```

In the code above, we first import the *math.pi* library prior to making the Vehicle class. The first method we create is the *__init__()* method, which initializes any instances of the class with the arguments provided. For items where an argument isn't provided, we set default values. For a parent class like this, default None values could be used, with each subclass defining their own initial values.

```
    def __repr__(self):
        """Return string representation of Vehicle."""
        return f"Speed:{self.vehicle_speed}, Wheel RPM:{
            self.wheel_rpm}, {self.num_wheels}
            wheels, {self.num_doors} doors, {self.
            engine_size} liter engine, {self.vehicle_weight
            } pounds curb weight, {self.wheel_diam} inch
            wheels"

    def calc_speed(self):
        """Determine vehicle speed based on outer diameter
            of wheel and RPM of tire.

        Base formula: diameter * Pi * RPM / 1056 = vehicle
            speed in mph
        """
        self.vehicle_speed = self.wheel_diam * pi * self.
            wheel_rpm / 1056
```

The next method is *__repr__()*, which creates a string representation of the class. If we didn't do this, we would simply get the memory location of the object when we attempted to print it. With this method, we define what text we want printed to the screen; in this case, we want all the parameters listed in a useful manner.

Next, we define the *calc_speed()* method. This is very simple; all it does is calculate the vehicle speed based on the size of the wheels and their rotational

speed. While we could have returned a value like a normal function call, the method simply updates the *vehicle_speed* parameter, which can be explicitly called later.

```
def calc_braking_distance(self):
        """Determine the distance required to stop vehicle
            at a given speed, including driver reaction
           time.

        Assumes 20 feet/sec braking power.
        Stopping distance is the distance (in feet) from
           vehicle to given object in front of it.
        Stopping time is time (in seconds) required to
           bring vehicle to complete stop.
        Thinking distance is the reaction time (in seconds
           ) to start braking.
        Braking distance is the total time required to
           fully stop vehicle once situation is noted by
           driver.
        """
        self.stopping_distance = 1.467 * self.
           vehicle_speed # ft/sec
        self.stopping_time = self.stopping_distance / self
           .brake_power
        self.thinking_distance = self.stopping_distance *
           2
        self.braking_distance = (0.5 * self.
           stopping_distance *
        self.stopping_time) + (2 * self.thinking_distance)

if __name__ == "__main__":
        car = Vehicle(0, 1000, 4, 2, 2.0, 3200, 19)
        print(car)
        car.calc_speed()
        print(f"{car.vehicle_speed} mph")
        car.calc_braking_distance()
        print(f"{car.braking_distance} feet")
```

Finally, we define the method *calc_braking_distance()*. This method ultimately provides the total distance required to stop the vehicle, based on speed and braking power. Again, rather than returning a value, we simply update the parameters.

The namespace line *if __name__ == "__main__":* runs the code that follows it if the file is called by itself, that is, if it is the main program. Here, we create an instance of Vehicle and print it to confirm the data that was provided. Then we call the *calc_speed()* method and print the car's speed. Finally, we call the *calc_braking_distance()* method and print that value.

Running the program provides the output shown in figure 5.1.1. Here, we see the output showing the parameters of the car instance, its calculated speed, and its calculated braking distance at that speed. Naturally, some changes would be ideal for actual use, such as changing the decimal places to less precision.

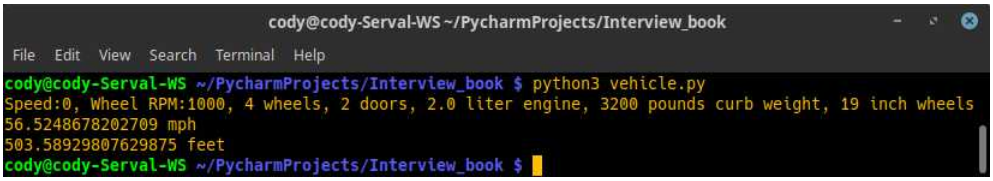

Figure 5.1.1.: Vehicle.py output

5.1.1. What are special class attributes?

We've seen how to make a class and how to instantiate it is an independent object. But what about the special methods we created: *__init__()* and *__repr__()*? What are they and how did we know to use them?

These, and other methods with leading and following double-underscores, are Python's "magic" methods. They are inherent to Python and help the language perform some of its special features. There are a lot of magic methods, but they are scattered around the Python documentation. The best reference I've found for magic methods is A Guide to Python's Magic Methods (https://rszalski.github.io/magicmethods/). As such, I won't recreate that information here, but highlight some of the more common magic methods that you may find.

- *__init__()*: Class initializer. It is called when the class is instantiated, and any arguments passed in during construction are provided to this method for variable assignment. Nearly every class will use this method. Note that this is not necessary in 3.7+ when the *dataclass* module was added to provide automatic initialization.

- *__del__()*: Class destructor. Contrary to first thought, this is not the same as the *del* statement. This defines object behavior when it is removed from memory due to garbage collection. Python normally handles objects in a common-sense way; this method is used for custom code that may require some special cleanup calls, such as open files or network connections.

- *__repr__()*: Defines how a class is represented when called. This is different from the *__str__()* method in that *__str__()* is strictly for human-readable output, whereas *__repr__()* can be used to create human-readable information, but could also be used to create machine-readable code.

- *__iter__()*: Used with container objects, dictates that the container should return an iterator on the container itself. These are most often used when the container acts as a sequence type object.

The special class attributes defined as magic methods provide a large amount of functionality to a programmer, effectively allowing creation of truly custom code. Rather than simply using the built-in objects that Python provides, new objects with unique capabilities can be created as well.

5.2. What is inheritance?

Inheritance is one of the primary things that make classes so useful and powerful. In essence, they allow you to inherit parameters and methods from a base class and customize subclasses from those inherited properties. Thus, you can change

how a program functions by modifying existing code and adding new components rather than having to rewrite the entire program.

If we look at the Vehicle class example from earlier, we can see how subclasses can inherit parameters and methods from a parent class, as shown in the code example below:

```
class Truck(Vehicle):
        """Subclass of Vehicle"""
        pass
if __name__ == "__main__":
        pickup = Truck(30, 1500, 4, 2, 4.0, 5000, 28)
        print(pickup)
        pickup.calc_speed()
        print(f"Vehicle speed: {pickup.vehicle_speed} mph"
            )
        pickup.calc_braking_distance()
        print(f"Braking distance: {pickup.braking_distance
            } feet")
```

For the *Truck* subclass, we inherit everything from the parent *Vehicle* class; thus, we use the *pass* statement within the body of the *Truck* class. We modified the "main" code to only work with the new *Truck* class. If we run this code, figure 5.2.1 is received.

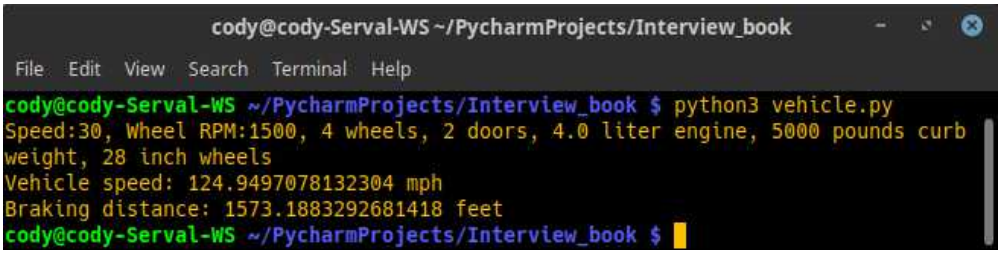

Figure 5.2.1.: Truck subclass of Vehicle

From this, we can see that the *pickup* instance of class *Truck* did, indeed, inherit all the parameters from *Vehicle*, as well as the parent methods. We created a subclass that can later be modified to account for characteristics associated with the new class.

Also, note that while we created the *pickup* instance with an initial speed of 30, after we run the *calc_speed()* method, a new speed is calculated that replaces the *vehicle_speed* parameter.

5.2.1. How does multiple inheritance work?

Some programming languages allow for a class to inherit from multiple parent classes, such as Python and C++. Others, like Java, do not support this. The reason is because multiple inheritance can be problematic and complex, leading to program issues that aren't easily resolved.

Over the course of several years, I have read a number of blogs and forum posts that essentially state, "If you need multiple inheritance, redesign your code." Personally, I have never needed multiple inheritance so I have never used it. However, it does make sense that it is really best for certain, niche uses and most of the time single inheritance will suffice. However, you may be asked to discuss how Python handles multiple inheritance, so this section will discuss it.

First, we will talk about the *diamond problem*. As shown in figure 5.2.2, a program may be written (purposefully or accidentally) so that a subclass (D) inherits from two separate superclasses (B and C) , each of which inherits from a common superclass (A).

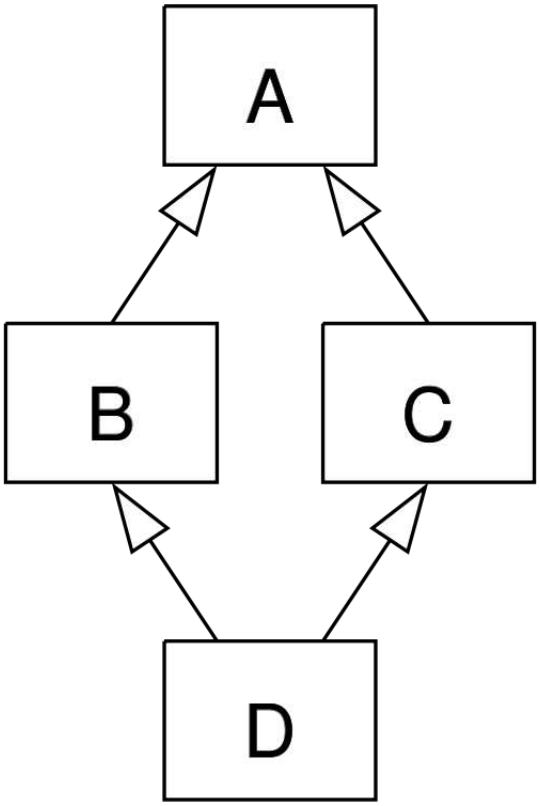

Figure 5.2.2.: Multiple inheritance diamond problem

The question becomes, if a method in A is overridden in B and/or C, which version of that method is inherited by D? Python uses the Method Resolution Order (MRO) to determine the order classes are browsed through.

In a multiple inheritance scenario, Python will look for the specified attribute within the current class. If not found, the search will move up the inheritance chain to the immediate superclasses, starting with the leftmost class first. The positioning is easily determined because Python automatically creates a sequence object as classes are inherited, i.e. the order in which inherited superclasses are listed when creating a class. In our example above, if a method isn't found in class D, Python will then check in class B, then C. If not found in either of those, it will look in class A.

Here is a basic, multiple inheritance program. While the subclasses may override the parent class method, the method only functions within its own namespace:

```
class A:
    def class_method(self):
        print("class_method␣of␣A␣called")

class B(A):
    def class_method(self):
        print("class_method␣of␣B␣called")
```

```
class C(A):
    def class_method(self):
        print("class_method of C called")

class D(B, C):
    def class_method(self):
        print("class_method of D called")

if __name__ == "__main__":
    inst1 = D()
    inst2 = C()
    inst3 = B()
    inst4 = A()
    inst1.class_method()
    inst2.class_method()
    inst3.class_method()
    inst4.class_method()
```

If we run this code, we get the output of figure 5.2.3. To confirm what MRO is doing, we can call the *mro()* method or call the *__mro__* attribute, as shown in figure 5.2.4.

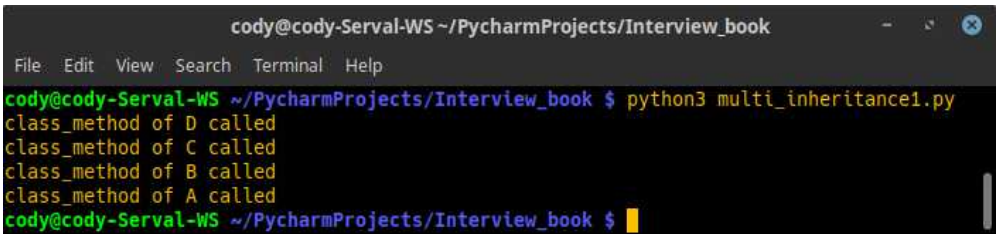

Figure 5.2.3.: Multiple inheritance output

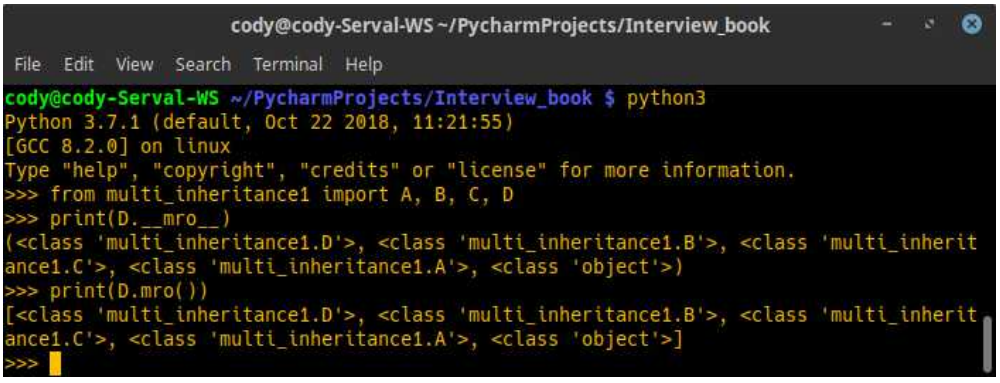

Figure 5.2.4.: Python MRO output

Notice that using the *__mro__* attribute creates a tuple, while the *mro()* method generates a list. Either way, you have a sequence that is used to define how Python will look for inherited objects.

What if we want each class to call the methods of its parent(s)? We could change the code to something like the example below:

```
class A:
    def class_method(self):
        print("class_method of A called")

class B(A):
    def class_method(self):
        print("class_method of B called")
        A.class_method(self)

class C(A):
    def class_method(self):
        print("class_method of C called")
        A.class_method(self)

class D(B, C):
    def class_method(self):
        print("class_method of D called")
        B.class_method(self)
        C.class_method(self)

if __name__ == "__main__":
    inst1 = D()
    inst1.class_method()
```

With these changes, we get the output in figure 5.2.5.

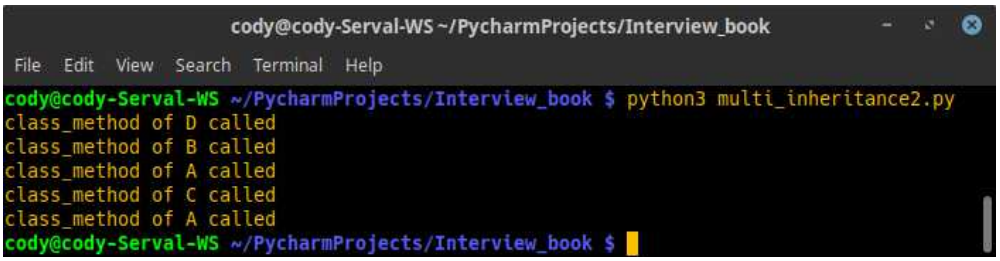

Figure 5.2.5.: Subclasses calling parent methods

Unfortunately, we have a problem. Because we have classes B and C both calling their parent class A's method, we get the output of *A.class_method()* twice. The best way to address this is via the *super()* function, which we saw earlier when used with the *__init__()* method. The revised code is shown below:

```
class A:
    def class_method(self):
        print("class_method of A called")

class B(A):
    def class_method(self):
        print("class_method of B called")
        super().class_method()
```

```
class C(A):
    def class_method(self):
        print("class_method of C called")
        super().class_method()

class D(B, C):
    def class_method(self):
        print("class_method of D called")
        super().class_method()

if __name__ == "__main__":
    inst1 = D()
    inst1.class_method()
```

Now, when we call this program, we get the correct, expected results shown in figure 5.2.6.

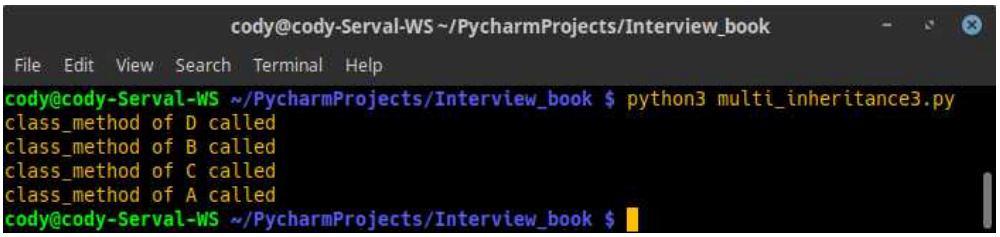

Figure 5.2.6.: *super()* call to parent class

The *super()* function returns a proxy object that delegates method calls to another class, either parent or sibling, but not a child class. Most commonly, it is used for the inheriting and extending methods from a parent, particularly the *__init__()* method.

5.3. What is encapsulation?

Encapsulation in Python is a means to prevent direct access to class members. Python doesn't have true private objects; it just uses pseudo-private to inform the programmer that a variable or method is not supposed to be accessed directly but through an API or other means.

When writing code, adding a single underscore prefix to a name indicates that it is private, such as *_myvariable*. While it can still be called, the underscore tells the user that this item should be considered private to the class and that another method, such as a property, provides access to it.

The reason private names exist in Python is because the implementation, naming scheme, or anything else is not guaranteed to remain constant with future updates. Thus, the user shouldn't be reliant on continued access in the future, or that the program will respond the same as in the past.

Another important point about private objects is that they are not automatically imported when using the *import* statement. This helps keep items private, as they are not included within the calling program's namespace.

Prefixing a name with double underscores, such as *__another-variable*, associates the object directly with the class, such that the only way to access the object is by providing the class name as well; this is known as *name mangling*. An attribute with double underscores is referenced by Python as *_Class-*

Name__attribute. This becomes the object's fully qualified name, much like a web page has a fully qualified domain name; it's the complete name of the object to ensure there is no confusion by the system. Associating the attribute directly with its class helps to prevent name shadowing or otherwise interfering with other objects that happen to have the same name.

The code below provides a simple example of how these concepts work in practice:

```
class Name:
    def __init__(self):
        self.name = "abc"
        self._name = "xyz"
        self.__name = "ABC"

if __name__ == "__main__":
    item = Name()
    print(item.name)
    print(item._name)
    print(item.__name)
    print(item._Name__name)
```

Figure 5.3.1 is generated when the program is ran. You can see that the program fails when trying to print *item.__name*. However, if we comment out that *print()* function, the program works correctly, as shown in figure 5.3.2.

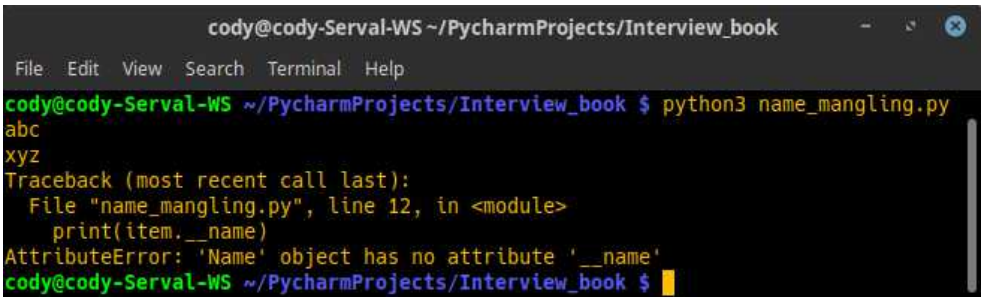

Figure 5.3.1.: Name mangling example

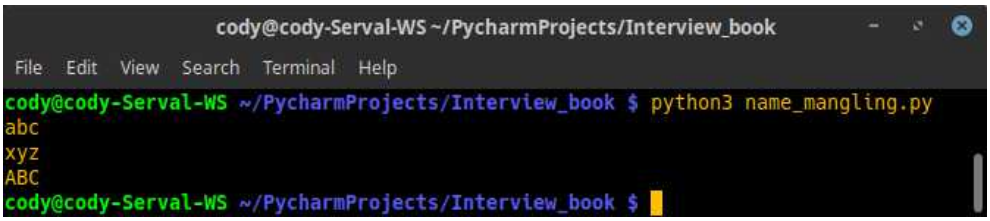

Figure 5.3.2.: Name mangling corrected

All of this just reinforces the fact that Python tries to prevent you from shooting yourself in the foot, but if you really want to, it will step out of the way if you force it.

5.4. What is polymorphism?

Polymorphism is the ability to leverage the same interface for different data types. While available for Python functions, it is more often found with classes.

The simplest way to demonstrate polymorphism is the code example figure 5.4.1.

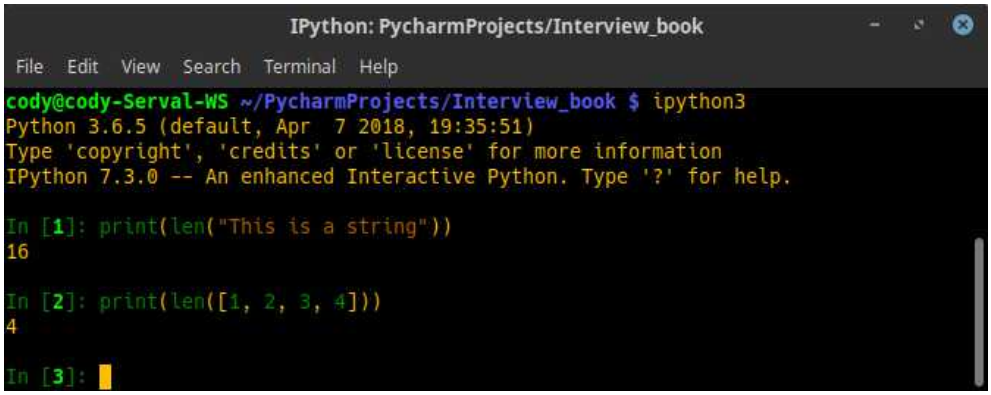

Figure 5.4.1.: Polymorphism example

Essentially, we are using the *len()* function to determine the length of two different data types: a string and a list. The back-end code that defines *len()* doesn't care what data type is passed into it; Python will automatically figure it out at runtime via duck typing.

Duck typing comes from the phrase, "If it walks like a duck and quacks like a duck, then it's probably a duck." In Python, this means that the interpreter doesn't care what actual data type an object is; all that is checked is what methods and properties the object has. In other languages, the actual data type is checked and, if it doesn't match the expectations, an error is generated.

This is one of the reasons for Python's ease-of-use; things just work because duck typing performs the data checks at runtime. It also means that custom data types can be processed without problems as long as they have inherited the correct methods and properties, such as *__iter__()*.

Polymorphism also comes into play with operator overloading. When adding integers, for example, the "+" sign will sum the numbers. However, you can use the exact same symbol with strings to concatenate them together. Again, Python is intelligent enough to recognize the different data types and, through polymorphism, the "+" operator is overloaded to perform addition or concatenation. Of course, there are limits; you can't try to combine different data types, such as an integer and a string, without an error, but as long as all the data types are compatible, Python should figure out what to do.

5.4.1. How do you extend and override classes?

In the *Truck* class example in figure 13.1.2 above, we simply inherited everything from the *Vehicle* class. This didn't do anything for us, except beyond demonstrating that subclasses automatically inherit properties from their superclass. Where inheritance really comes into play is when we extend and override superclass properties, as this leverages the entire idea of code reuse via subclassing.

To be most effective, a class hierarchy should have a generic superclass that has the parameters and methods that will be common to all subclasses. Meanwhile, the subclasses will have custom code that makes them special, such as new

parameters or methods that is unique between classes. That way, programmers don't have to recreate a lot of redundant code; the subclasses simply inherit the common properties, leaving the programmer to focus on the customization of subclasses.

To demonstrate this, we continue with our *Vehicle* example below:

```
class Car(Vehicle):
    """Subclass of Vehicle

    Inherits all base parameters, but adds engine type and
        acceleration.
    Default values set for number of wheels and doors.
    """
    def __init__(self, speed=0, rpm=0, wheels=4, doors=4,
        engine=0, weight=0, wheel_diam=0, engine_type="
        Gasoline"):
        super(Car, self).__init__(speed, rpm, wheels,
            doors, engine, weight, wheel_diam) # Use
            superclass parameters
        self.engine_type = engine_type # Set new
            parameters
        self.car_acceleration = 0
        self.eng_rpm = 0
        self.gear_ratio = 0
        self.axle_ratio = 0
```

For the *Car* subclass, we have both extended and overridden the inherited methods of *Vehicle*. While we inherited the *__init__()* method, we also extended it by adding two new parameters for it to deal with: *engine_type* and *car_acceleration*. Extending methods essentially takes the parent method and modifies it for purposes of the subclass' needs.

```
    def acceleration(self, start_speed, end_speed,
        time_required):
        """Calculate the average acceleration of a car

        Formula: a = dv/dt, where a = acceleration, dv =
            change in velocity in mph, and dt = time used
            in seconds
        """
        speed_delta = (end_speed - start_speed) * 5280 /
            3600 # Convert mph to ft/sec
        self.car_acceleration = speed_delta /
            time_required

        return self.car_acceleration

    def calc_speed(self, eng_rpm, gear_ratio, wheel_diam,
        axle_ratio):
        """Calculate vehicle speed based on engine
            parameters.
```

```
        Formula: speed in mph = (engine_rpm * min/hour *
            pi * 2 * wheel_diameter) / (inches/mile *
            gear_ration * axle_ratio)
        """
        self.eng_rpm = eng_rpm
        self.gear_ratio = gear_ratio
        self.wheel_diam = wheel_diam
        self.axle_ratio = axle_ratio
        min_hour = 60 # convert rpm to rph
        inch_to_mile = 63360
        self.vehicle_speed = (self.eng_rpm * min_hour * pi
            * 2 * self.wheel_diam) / (inch_to_mile * self.
            gear_ratio * self.axle_ratio)
```

Overriding inherited methods is basically a complete revision, keeping only the name while the results are completely different. In short, overriding methods is changing the default functionality of inherited methods. The catch is, they only apply to instances of that particular subclass. In the example above, we overrode the *calc_speed()* method to calculate the car's speed based on engine parameters and wheel dimensions, rather than wheel dimensions and wheel rotational speed.

We also added a new method: *acceleration()*. This method calculates the car's average acceleration based on change in velocity over time. You'll also note that, when we created the sedan instance, we ignored providing the number of wheels and doors, as those were set to default values for the *Car* class.

```
if __name__ == "__main__":
    sedan = Car(30, 1000, engine=1.5, weight=3200,
        wheel_diam=19)
    print(sedan)
    print(f"Acceleration: {sedan.acceleration(sedan.
        vehicle_speed, 60, 8)} ft/sec^2")
    sedan.calc_speed(eng_rpm=3000, gear_ratio=1.5,
        wheel_diam=12, axle_ratio=4)
    print(f"Vehicle speed: {sedan.vehicle_speed} mph")
    sedan.calc_braking_distance()
    print(f"Braking distance: {sedan.braking_distance}
        feet")
```

The main part of the program simply calls all the new methods and parameters to provide figure 5.4.2.

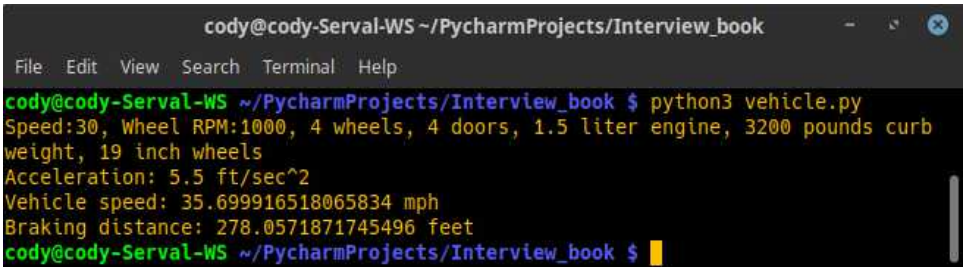

Figure 5.4.2.: Car subclass of Vehicle

It can be seen that the *Car* class worked as expected; it inherits properties from its parent class while also extending and overriding the methods.

5.5. Summary

In this chapter, we looked at object-oriented programming in Python. We talked about defining and instantiating classes, class inheritance, encapsulation, polymorphism, and extending and overriding inherited functionality.

In the next chapter, we will look at iteration and recursion, especially as they apply to Python programming.

Part III.

Algorithms

6. Iteration and Recursion

A fundamental part of programming is dealing with repetitive behavior. This can be handled in two main ways: iteration and recursion. Generally speaking, iteration is the repetition of a process in order to generate a sequence of outcomes, or to perform the same operation on a sequence of input arguments. Recursion happens when a function or method calls itself repeatedly to perform an operation.

In this chapter, we will discuss the following:

- How does iteration compare to recursion?
- How does recursion work?

6.1. How does iteration compare to recursion?

In Python, an object is considered iterable if it is either a physically stored sequence (like a *list*), or an object that produces one result at a time when used by an iteration tool like a *for* loop. Any object with a *__next__()* method automatically moves to the next result, and which raises the *StopIteration* exception at the end of the series of results, is considered an iterator in Python.

Any iteration tool, like a *for* loop, will know how to automatically call the *__next__()* method to get the next item to process, as well as stopping automatically when *StopIteration* is generated. Thus, when reading lines from a file, using a *for* loop to process each line is, at its heart, not actually reading the file explicitly. The loop simply calls the next line within the file for automatic processing.

An example of this file processing is shown in figure 6.1.1. The example uses the IPython interactive shell rather than the traditional Python interactive shell. We create a simple *for* loop in line 1 that opens one of the multiple inheritance scripts we wrote in chapter 3. The loop iterates over each line in the file and prints them to the screen. Note that we never explicitly read from the file using *readlines()* or some other function call. Looping is now considered the best way to read lines in Python, as it is quicker and doesn't use excess memory.

Recursion in Python relies on the programmer to make the function work; there are no built-in recursive operations like there are for iteration. While an iterative function stops when a particular condition is met, recursion continues until a base case is completed. If the operation is never able to reach that base case, then infinite recursion occurs.

Figure 6.1.2 is an example of a recursion function. In line 2, we define a simple, recursive summation function that takes a sequence of numbers and adds them together. It is called in line 3, with the output provided. In lines 4 & 5, we show what the actual recursive process is. For the list of integers, the function starts on the first entry and then adds each subsequent entry, slicing the list each time to remove the previously added value.

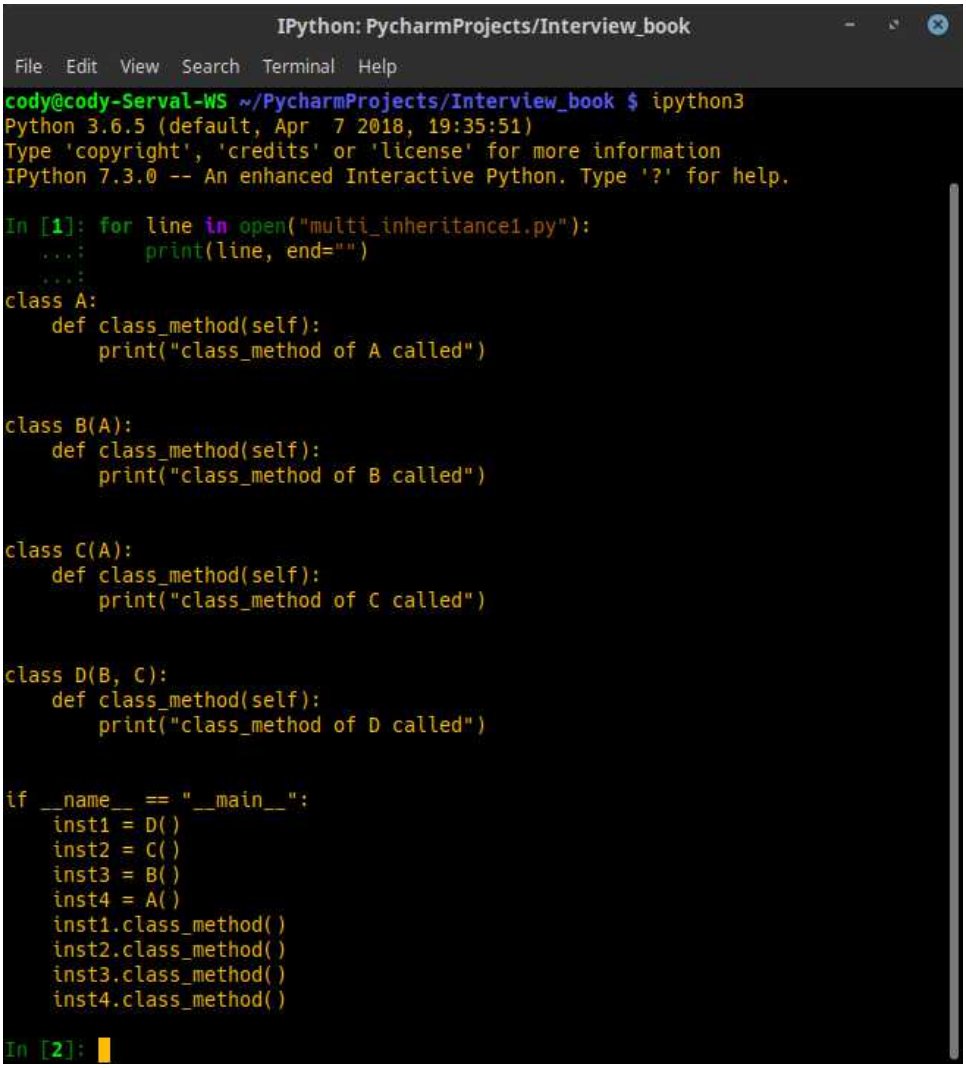

Figure 6.1.1.: Iteration example

It's important to note that recursion only occurs within a function or method, whereas iteration can apply to any set of instructions that are repeatedly executed.

6.2. How does recursion work?

Recursion is actually rarely used in Python, as Python's procedural loops are simpler and recursion ends up taking more time. In addition, nearly every recursive function can be made iteratively; it's just that some operations are better stated recursively. In general, if a problem can be discussed in a recursive fashion, then a recursive solution is probably best.

A key aspect of recursion is that a stack space has to be available for it to work. For each new launch of the recursive function, a new item is added to the stack. Figure 6.2.1 shows this process for a simple recursive factorial problem. Note that this isn't Python code, but is purely demonstrative.

For the factorial value 4!, there has to be four successive stack calls before the final answer is derived. This occurs for any recursive function: the stack space used increases as the number of recursive calls increases. Thus, recursive

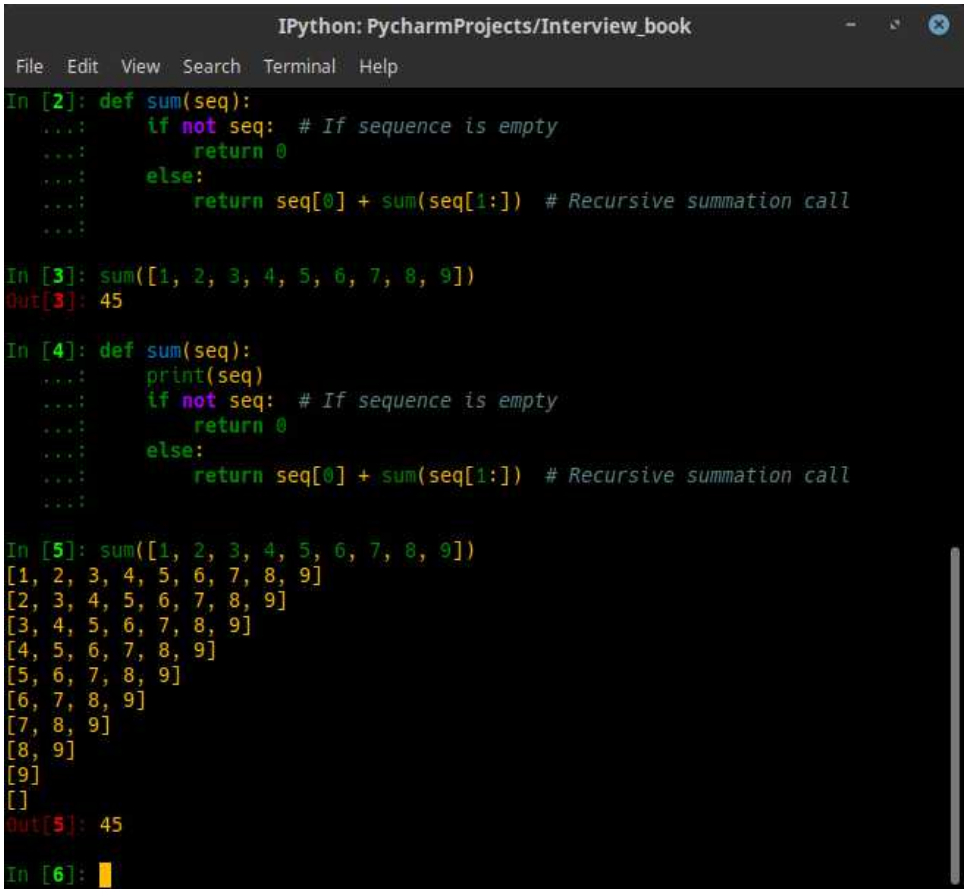

Figure 6.1.2.: Recursion example

functions have the potential for crashing a system as stack space fills up. Iterative functions simply slow down the system as more CPU cycles are used to process each iteration.

In addition, a recursive function has to ultimately converge on a base solution, otherwise it will become an infinite recursion. Iteration stops once a set condition is achieved. Recursive functions are similar to forking processes; each recursive call creates a new function with associated overhead. Iterative processes are one function that is looped over each time, so the actual function can be cached; this is one way that PyPy's Just-In-Time compiler improves Python's speed performance.

Python, however, prevents infinite recursion and overuse of stack space by limiting recursion to 1000 calls, as shown in the screenshot below:

You can overcome this limit by using *sys.setrecursionlimit()*, but you still have to be careful about complications. In some cases, recursion is useful, such as directory transversal, but in Python, loops and iteration are your best options.

6.2.1. Towers of Hanoi

A popular example of recursion in a real-world application is the Towers of Hanoi, also called the Tower of Brahma or Lucas' Tower and can be seen in both the singular "Tower" and plural "Towers" forms. It consists of three rods and a number of disks of different sizes, which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

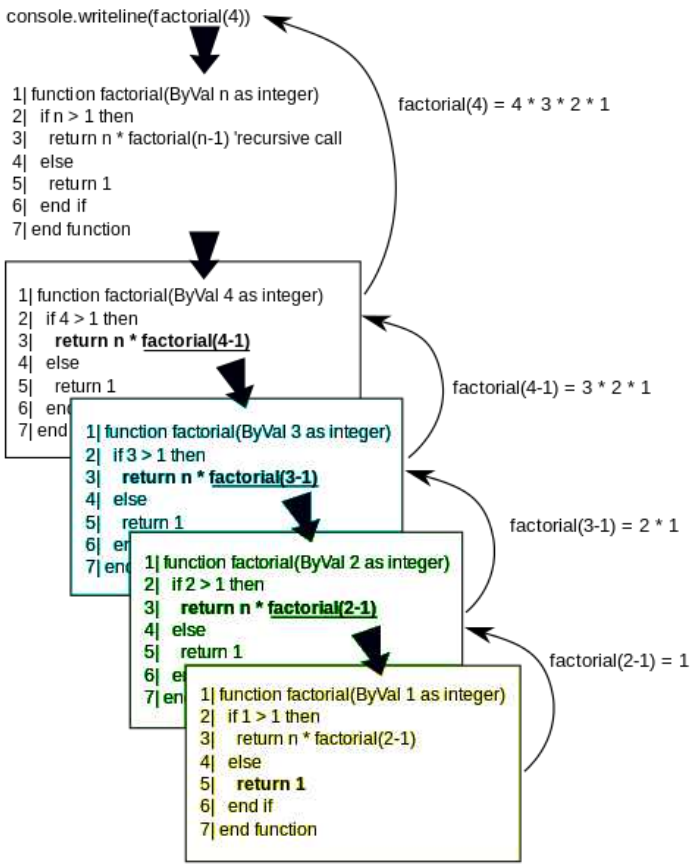

Figure 6.2.1.: Recursion example

Figure 6.2.3 shows how the Towers is constructed. Peg A represents the starting position of the discs. Peg B represents the final position of the Tower. Peg Via is used as a transitory holder for the discs, allowing the player to move discs to the side while also moving discs between pegs A and B.

The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
- No larger disk may be placed on top of a smaller disk.

With 3 disks, the puzzle can be solved in 7 moves. The minimal number of moves required to solve a Tower of Hanoi puzzle is $2^n - 1$, where n is the number of disks.

The code example below shows one solution to this problem[20]:

```
def hanoi_tower(n, from_peg, to_peg, temp_peg):
    if n == 1:
        print(f"Move disk 1 from {from_peg} to {to_peg}")
        return
    hanoi_tower(n - 1, from_peg, temp_peg, to_peg)
    print(f"Move disk {n} from {from_peg} to {to_peg}")
    hanoi_tower(n - 1, temp_peg, to_peg, from_peg)
```

```
cody@cody-Serval-WS ~/PycharmProjects/Transportation_model $ ipython3
Python 3.6.5 (default, Apr  7 2018, 19:35:51)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.3.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: def factorial(n):
   ...:     if n == 1:
   ...:         return 1
   ...:     else:
   ...:         return n * factorial(n-1)
   ...:

In [2]: print(factorial(3000))
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
<ipython-input-2-9eb6552901e5> in <module>
----> 1 print(factorial(3000))

<ipython-input-1-1de8b228879f> in factorial(n)
      3         return 1
      4     else:
----> 5         return n * factorial(n-1)
      6

... last 1 frames repeated, from the frame below ...

<ipython-input-1-1de8b228879f> in factorial(n)
      3         return 1
      4     else:
----> 5         return n * factorial(n-1)
      6

RecursionError: maximum recursion depth exceeded in comparison

In [3]:
```

Figure 6.2.2.: Python recursion error

```
if __name__ == "__main__":
    discs = 4
    hanoi_tower(discs, 'Peg A', 'Peg Via', 'Peg B')
```

The output of the program is shown in figure 6.2.4. The code uses the same names as the diagram of the Tower previously shown, for ease of confirmation. However, the results of the program put the final tower position on Peg Via, rather than Peg B. In most versions of the game, it doesn't matter where the tower is moved to, while some players like to specify a particular ending peg. It is left to the reader to rewrite the code so that the final position of the Tower is on Peg B.

6.3. Summary

In this chapter, we looked at iteration and recursion. Python prefers developers to use iterative functions over recursion, as the language has a variety of built-in features that improve iterative operations. Recursion can be used, but is recommended only in situations where it is most applicable and has appropriate "safety features" to ensure that infinite recursion doesn't occur.

Next chapter, we will look at divide and conquer algorithms, where a problem is broken down into sub-problems that are solved independently, then merged together for the final solution.

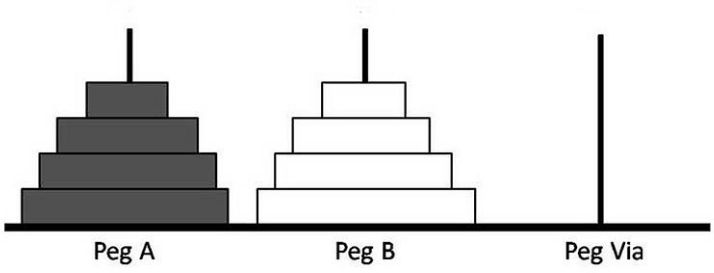

Figure 6.2.3.: Towers of Hanoi [Chaugule26 (CC BY-SA 4.0)]

```
codyjackson@pop-os: ~/PycharmProjects/Interview_book/recursion
codyjackson@pop-os:~/PycharmProjects/Interview_book/recursion$ python3 tower_of_
hanoi.py
Move disk 1 from Peg A to Peg B
Move disk 2 from Peg A to Peg Via
Move disk 1 from Peg B to Peg Via
Move disk 3 from Peg A to Peg B
Move disk 1 from Peg Via to Peg A
Move disk 2 from Peg Via to Peg B
Move disk 1 from Peg A to Peg B
Move disk 4 from Peg A to Peg Via
Move disk 1 from Peg B to Peg Via
Move disk 2 from Peg B to Peg A
Move disk 1 from Peg Via to Peg A
Move disk 3 from Peg B to Peg Via
Move disk 1 from Peg A to Peg B
Move disk 2 from Peg A to Peg Via
Move disk 1 from Peg B to Peg Via
codyjackson@pop-os:~/PycharmProjects/Interview_book/recursion$
```

Figure 6.2.4.: Towers of Hanoi output

7. Divide and Conquer

Divide and conquer is an algorithm paradigm, traditionally based on multi-branched recursion. Basically, it recursively breaks down a problem into multiple sub-problems several times, until these sub-problems become simple enough to be solved directly. When the sub-problems are solved, the results are combined to give a solution to the original problem.

In this chapter, we will talk about:

- What is a dividing and recurrence function?
- What is a divide and conquer algorithm?
- Examples

7.1. What is a dividing and recurrence function?

While divide and conquer (D&C) algorithms perform the actual operations to find the result, recurrence functions analyze the complexity of a D&C algorithm; that is, a recurrence relation is one way to describe how a function is defined without actually creating the formula itself. Once you have the recurrent function, you can use Big-O notation to determine what type of function it actually describes, such as linear, logarithmic, exponential, etc.

When developing the recurrent function for a D&C algorithm, the following steps are used:

1. Assume the D&C algorithm divides the problem with size n into a number of sub-problems.
2. Assume each sub-problem has a size of $\frac{n}{b}$.
3. Assume $f(n)$ operations are needed to combine the sub-problem solutions into the original problem solution.
4. If we let $T(n)$ be the number of operations required to solve for the problem, we get the following function:

   $T(n) = aT(\frac{n}{b}) + f(n)$ [Commonly known as the "Master Theorem"]
5. To make the recurrence well-defined, the $T(\frac{n}{b})$ term is actually either $T(\lceil \frac{n}{b} \rceil)$ or $T(\lfloor \frac{n}{b} \rfloor)$; that is, ceiling or floor of $\frac{n}{b}$.
   a) Ceiling means the smallest integer greater than or equal to $\frac{n}{b}$.
   b) Floor means the largest integer less than or equal to $\frac{n}{b}$.
6. Normally, you will see two functions: one with initial condition in the sequence, such as $T(1)$ or $T(0)$ with the number in parentheses indicating the initial index value in the sequence, and another with $T(n)$.

7.2. What is a divide and conquer algorithm?

Much like factorials are commonly used as examples of recursive algorithms, binary searches are common examples of D&C algorithms. In this example, we have a list of numbers and we want to search for a specific value. To do so, we have to split the list in half multiple times until we arrive at the desired value.

1. Sequence problem: 1, 4, 5, 11, 14, 15, 20 and the number we are looking for is 15.
2. Split the sequence in half to create two sub-problems: 1, 4, 5, 11 and 14, 15, 20.
3. Check highest value of the first half. Is it greater than or less than the desired value? In this case, the highest value in the first sub-problem (11) is less than the desired value (15), so we know that we can ignore everything in the first half.
4. Looking at the second half, we again divide it into separate sub-problems: 14, 15, and 20.
5. We look at the largest value of the first half and see that it (15) is equal to our desired value. We are done.
6. The recurrent functions are: $T(n) = T(\lfloor \frac{n}{2} \rfloor) + 2$ and $T(1) = 2$
   a) Because we have the floor of $\frac{1}{2}$, the largest integer less than $\frac{1}{2}$ is 0, so $0 + 2 = 2$.

7.3. Examples

Here are a few more examples to demonstrate the concept of divide and conquer, as well as recurrence functions.

7.3.1. How can you find the sequence minimum and maximum?

Consider the sequence $a_1, a_2, \ldots, a_n$.

- If $n = 1$, then a_1 is both the minimum and the maximum.
- If $n > 1$, split the sequence into two sub-sequences, either where both have the same number of elements or where one of the sequences has one more element than the other. The problem is reduced to finding the maximum and minimum of each of the two smaller sequences.
- The solution to the original problem results from the comparison of the separate maximum and minimum of the two smaller sequences.

Problem: Find a recurrence $T(n)$ that is the number of operations required to solve the problem of size n.

Solution: The problem of size n can be reduced into two problems of size $\lceil \frac{n}{2} \rceil$ and $\lfloor \frac{n}{2} \rfloor$. Two comparisons are required to find the original solution from those sub-problems:

- $T(1) = 1$
- $T(n) = 2T(\frac{n}{2}) + 2$

The main divide and conquer algorithm is shown below:

```
if y - x <= 1 then
    return (max(numbers[x], numbers[y]),
            min((numbers[x], numbers[y]))
else
    (max1, min1):= maxmin(x, /floor[(x + y)/2])
    (max2, min2):= maxmin(/floor[(x + y)/2) + 1)], y)
return (max(max1, max2), min(min1, min2))
```

When using Python, we get to cheat a little bit. Python has two built-in functions that handle all this work for you: *min()* and *max()*. We use them in the following example to show one of the reasons why Python is so popular for programmers.

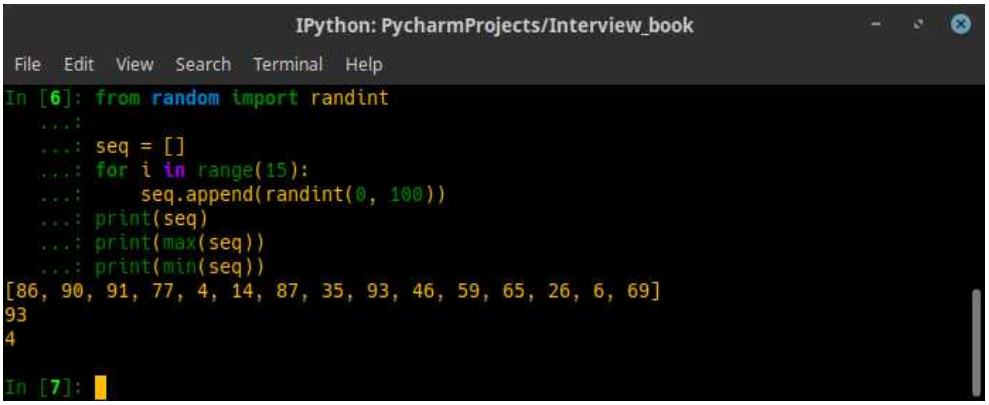

Figure 7.3.1.: Python min-max example

Figure 7.3.1 a simple IPython interactive session. In this case, we use random numbers to create a list of 15 items, then printed the maximum and minimum values within the sequence. In many cases, Python doesn't require the programmer to do a lot of unnecessary work, so knowing the built-in and third-party libraries that are available can save a lot of time in development.

7.3.2. What is a merge sort?

Consider an unsorted list of n elements.

- Continually divide the list until you have n sublists, each containing one element. Naturally, a list of one element is considered sorted.
- Compare each element with the adjacent list to sort and merge the two adjacent lists.
- Repeatedly merge the sublists to produce new sorted sublists until there is only one sublist remaining, which is the final, sorted list.

Regardless of the number of elements, n number of comparisons are required, i.e. the number of comparisons equals the number of elements. This is functionally the same operational sequence as the previous example; the only difference is that we don't know what the constant value is, so we have to use the variable n.

Therefore, the recurrence function becomes

- $T(1) = 0$
- $T(n) = 2T(\frac{n}{2}) + n$

Figure 7.3.2 is a diagram of this algorithm, followed by an analysis of the operations involved:

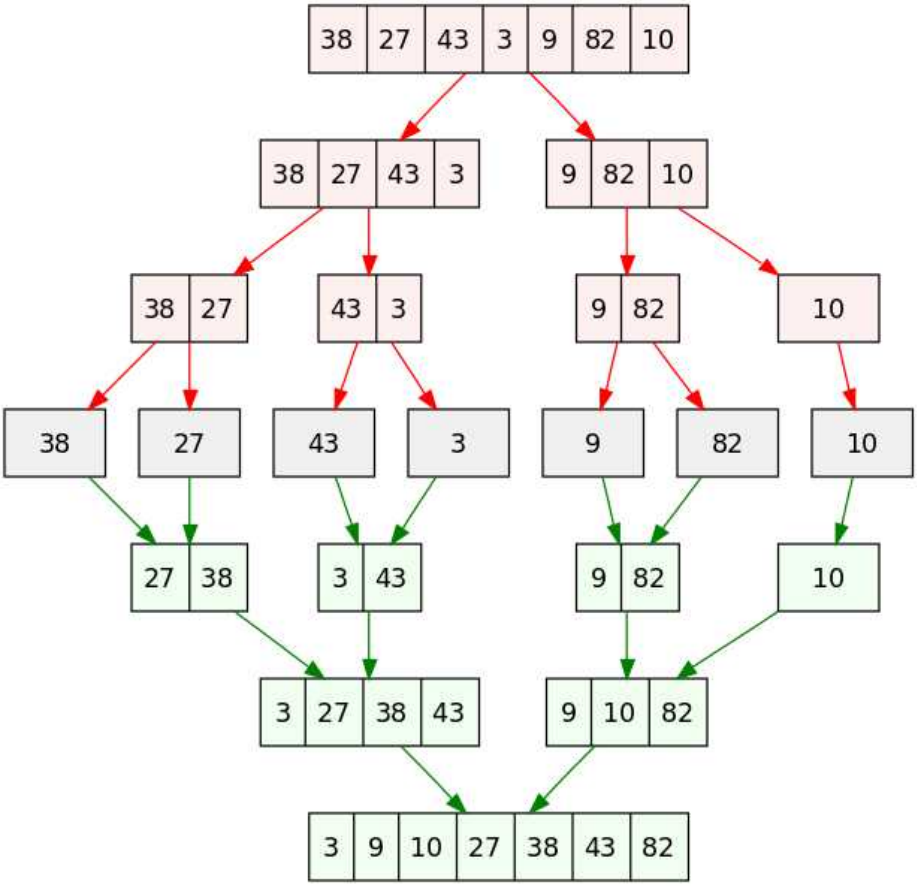

Figure 7.3.2.: Merge sort algorithm

1. We have an unsorted sequence.
2. We first divide it in half (as best as possible).
3. Each of the sub-sequences are divided into pairs.
4. Another division gives us the individual value for each element.
5. Each pair of elements is combined, with the lowest value first.
6. These pairs are combined, again with the lowest values first.
7. Both sub-sequences are combined into the final solution.

An example of this is shown in the code below[21]:

```
def merge_sort(array):
    """Takes input array and rewrites it to sorted list"""
    if len(array) > 1:
        midpoint = len(array) // 2 # Midpoint of the array
            in whole numbers
        left_side = array[:midpoint] # Find left half
        right_side = array[midpoint:] # Find right half

        # Sort the two sides
        merge_sort(left_side)
        merge_sort(right_side)

        left_counter = right_counter = main_counter = 0

        # Copy data to temp arrays left_side[] and
            right_side[]
        while left_counter < len(left_side) and
            right_counter < len(right_side):
            if left_side[left_counter] < right_side[
                right_counter]:
                array[main_counter] = left_side[
                    left_counter]
                left_counter += 1
            else:
                array[main_counter] = right_side[
                    right_counter]
                right_counter += 1
            main_counter += 1

        # Check if any element was left on either side
        while left_counter < len(left_side):
            array[main_counter] = left_side[left_counter]
            left_counter += 1
            main_counter += 1

        while right_counter < len(right_side):
            array[main_counter] = right_side[right_counter
                ]
            right_counter += 1
            main_counter += 1

if __name__ == '__main__':
    arr = [22, 45, 8, 1, 12, 99, 32] # This list will be
        overwritten
    print(f"Given array is {arr}")
    merge_sort(arr)
    print(f"Sorted array is {arr}")
```

When ran, we get the output in figure 7.3.3.

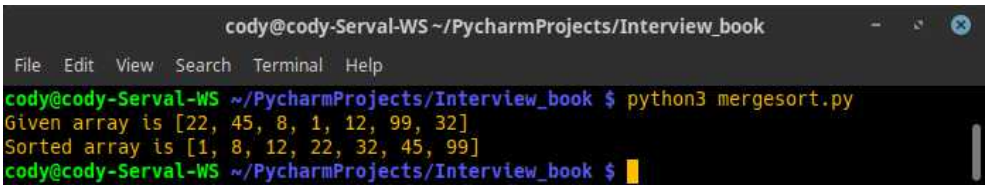

Figure 7.3.3.: Merge sort output

7.3.3. What is the largest sum in a subarray?

Assuming you have a one-dimensional list that has both positive and negative numbers, what is the largest sum available using a contiguous subarray of numbers?

Normally, you might approach this using two loops. The first loop selects the beginning number while the second loop calculates the maximum sum, given the first loop's number, and compares it to the overall maximum. This method has a complexity of $O(n^2)$.

Using divide-and-conquer, the complexity becomes $O(n \log n)$. The process is described in the following steps:

1. Divide the list into two halves
2. Calculate and return the maximum of the following:
   a) Left half sum via recursion
   b) Right half sum via recursion
   c) Sum across the midpoint. This is found by calculating the sum from midpoint to a point on the left half, then find the sum from midpoint + 1 to a point on the right half, then add those two sums together.

The following code provides an example of this[22]:

```
def sum_across_midpoint(input_array, first_value, midpoint
    , last_value):
    # Calculate left side sum
    current_sum = left_sum = 0
    for i in range(midpoint, first_value - 1, -1):
        current_sum = current_sum + input_array[i]
        if current_sum > left_sum:
            left_sum = current_sum

    # Calculate right side sum
    current_sum = right_sum = 0
    for i in range(midpoint + 1, last_value + 1):
        current_sum = current_sum + input_array[i]
        if current_sum > right_sum:
            right_sum = current_sum
    return left_sum + right_sum

def subarray_sum(input_array, first_value, last_value):
    # Only one element
```

```
    if first_value == last_value:
        return input_array[first_value]

    # Find middle point
    midpoint = (first_value + last_value) // 2 # Truncate
        division

    # Return maximum of: left side, right side, or across
        midpoint
    return max(subarray_sum(input_array, first_value,
        midpoint), subarray_sum(input_array, midpoint +
        1, last_value), sum_across_midpoint(input_array,
         first_value, midpoint, last_value))

if __name__ == '__main__':
    arr = [-2, -5, 6, -2, -3, 1, 5, -6]
    max_sum = subarray_sum(arr, 0, len(arr) - 1)
    print("Maximum contiguous sum is ", max_sum)
```

The output of the program is shown in figure 7.3.4. This result is easy to identify in the code example: you're going to have to include positive numbers if you want to have the highest sum. Therefore, the contiguous subarray will be *arr[2:7]*, or 6, -2, -3, 1, 5.

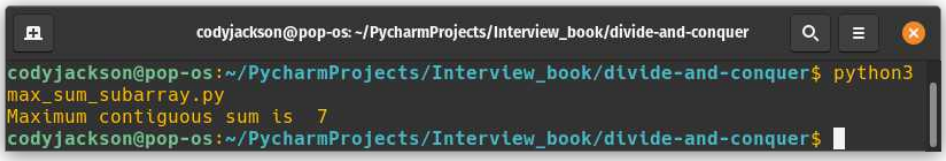

Figure 7.3.4.: Subarray largest sum

7.3.4. How to manually calculate x^n?

While it is simple enough to use a calculator or built-in Python math library to solve for powers, in this example, we will look at a normal divide-and-conquer solution, then look at one way to optimize the first solution.

Here is one way of writing a simple recursive divide-and-conquer program[23]:

```
def power(x, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return power(x, n // 2) * power(x, n // 2)
    else:
        return x * power(x, n // 2) * power(x, n // 2)

if __name__ == '__main__':
    base = 7
    pow = 2
    print(power(base, pow))
```

When ran, we get the output in figure 7.3.5. Note that this program does not account for negative powers. If you try to use a negative power, you will get a recursion error (figure 7.3.6).

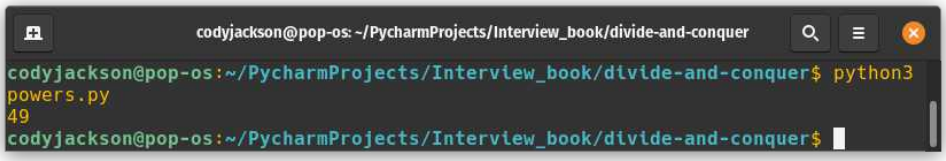

Figure 7.3.5.: Calculate powers

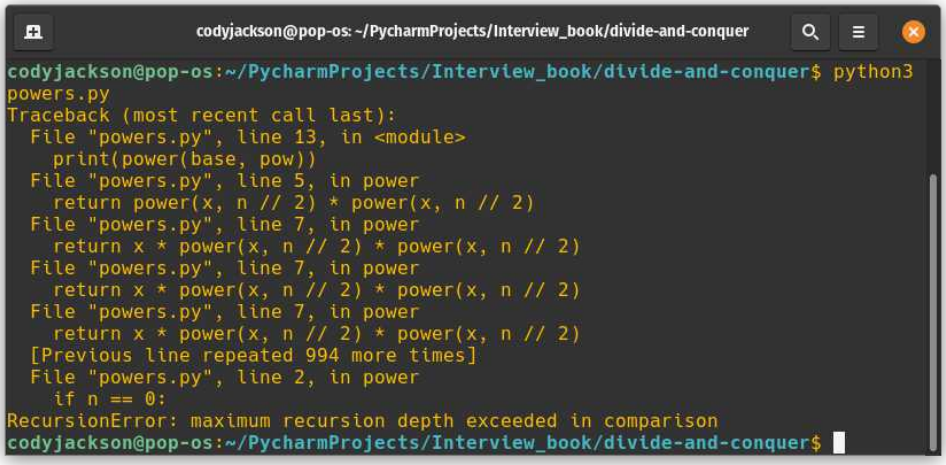

Figure 7.3.6.: Negative powers error

If we want to rewrite the code to handle negative powers and floating base values,we can do the following[23]:

```
def power(x, n):
    if n == 0:
        return 1
    temp = power(x, int(n / 2)) # Cast pow to integer

    if n % 2 == 0:
        return temp * temp
    else:
        if n > 0:
            return x * temp * temp
        else:
            return (temp * temp) / x

if __name__ == '__main__':
    base = 2.5
    pow = -3
    print(f"{power(base, pow)}")
```

When ran, we get the output in figure 7.3.7. Note that if you don't cast n to an integer and simply try to use integer division "//", you will get a recursion error (figure 7.3.8).

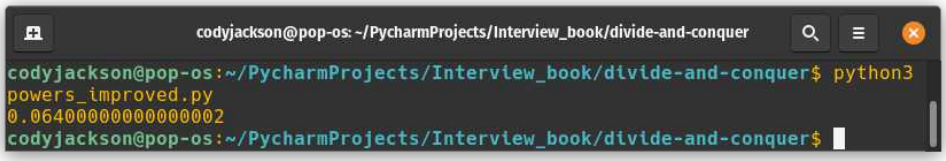

Figure 7.3.7.: Improved calculate powers

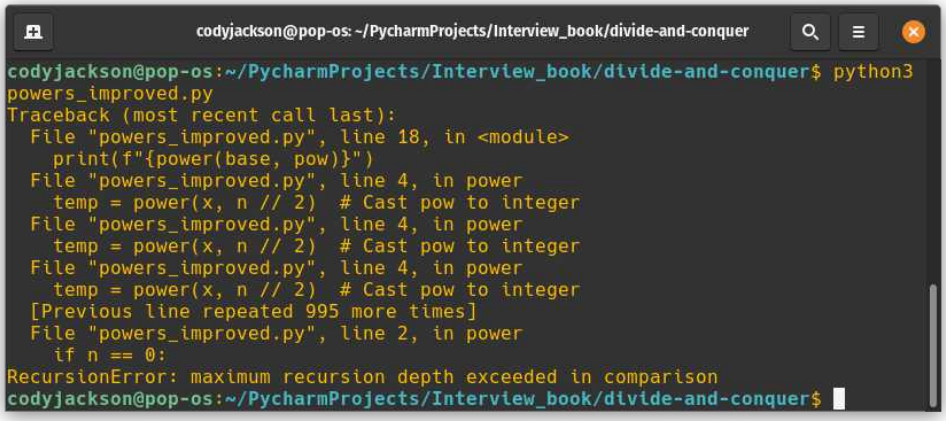

Figure 7.3.8.: Integer division error

In this case, implicitly trying to create an integer via integer division fails, while explicitly calling *int()* works fine. I can't definitively say why, but based on what I have been able to find, I suspect it is because integer division "//" is actually floor division, where the floating point portion of the value is removed by rounding down to the nearest whole number.

Therefore, when trying to run the program with integer division, the program is continuously trying to round the result of dividing the power by 2. When the quotient is explicitly converted to an integer using *int()*, there is no rounding; the floating point portion is simply dropped altogether, leaving only the integer portion of the quotient.

7.4. Summary

The D&C technique forms the basis of many efficient algorithms for a variety of problems, including sorting, multiplying large numbers, fast Fourier transforms, etc. Designing effective D&C algorithms requires a solid understanding of the underlying problem to be solved. The computational cost of a D&C algorithm is determined with recurrence functions.

In the next chapter, we will look at algorithm analysis, including a deeper dive into time and space complexity, as well as amortized analysis.

8. Algorithm Analysis

We touched on algorithm analysis in section §4.3 when we discussed time and space complexity. However, there are other ways to analyze algorithms, depending on the need of the project.

Mathematical analysis of algorithm performance can be a crucial part of software planning. Ideally, in the planning process, the developers, team leads, and other stakeholders will participate in the project development, where everyone provides their input for the best way forward. Part of that is deciding which algorithms to use, based on use case and time complexity, as well as the best return-on-investment for algorithm selection and use-cases.

There are a number of established software development concepts when it comes to narrowing down algorithm choices. In this chapter, we will look at the following:

- How to solve recurrence relations?
- What are asymptotic approximations?
- What is amortized analysis?
- What are cost models?
- How to analyze parallel algorithms?

8.1. How to solve recurrence relations?

Analyzing algorithms can be performed by expressing them as recursive or iterative procedures. Thus, we attempt to express the "cost" of a problem in terms of solving smaller problems. Recurrence relations represent a way to directly map a recursive representation of a program to a recursive representation of a function describing the program's properties.

Creating a recurrence relation that describes an algorithm's performance is a step forward, since the relation itself provides a fair bit of information about the problem. For example, particular properties of the algorithm, as related to the input model, are encapsulated within a much simpler mathematical expression. Of course, not all algorithms can be simplified this way, but luckily, many of the most important algorithms can be expressed recursively which, when analyzed, provide the recurrence relations that may describe the average case or worst-case performance.

When attempting to find a formula that defines a mathematical sequence, a common step is to find the n^{th} term as a function of earlier terms within the sequence. Ideally, having a closed form function of the n^{th} term is best, but sometimes all you have is the relationship between previous values in a sequence, such as how a Fibonacci sequence is the sum of the previous two terms.

The closed form simply means that the mathematical expression has a finite number of operations. In other words, a solution is closed form if it solves a problem in terms of functions that are generally-accepted and well-known. For

example, the equation $\frac{4+\log(x)}{\sqrt[3]{x}}$ is a closed form formula because it has a finite number of operations: a logarithm, an addition, a division, and a cube root. If a closed form is not known, a closed form formula can be deduced via recurrence.

8.1.1. What types of recurrences exist?

Recurrence relations are commonly classified by: how terms are combined, the coefficients used, and the number and nature of previous terms. While the following table is not exhaustive, it does provide a representative example of the most common recurrences in algorithm analysis.

Recurrence Type	Example
First Order Linear	$a_n = na_{n-1} - 1$
First Order Nonlinear	$a_n = \frac{1}{1+a_{n-1}}$
Second Order Linear	$a_n = a_{n-1} + 2a_{n-2}$
Second Order Nonlinear	$a_n = a_{n-1}a_{n-2} + \sqrt{a_{n-2}}$
Second Order Variable Coefficients	$a_n = na_{n-1} + (n-1)a_{n-2} + 1$
t^{th}Order	$a_n = f(a_{n-1}, a_{n-2}, \ldots, a_{n-t})$
Full History	$a_n = n + a_{n-1} + a_{n-2} + \ldots + a_1$
Divide-and-Conquer	$a_n = a_{\lfloor \frac{n}{2} \rfloor} + a_{\lceil \frac{n}{2} \rceil} + n$

Table 8.1.: Recurrence Types

8.1.2. What are linear first order recurrences?

This type of recurrence reduction is a simple iteration. The recurrence is applied to itself until only constants and initial values are left, leaving simplification for last. Being the simplest type of recurrence, first order recurrences immediately reduce to a product. Thus, the recurrence $a_n = na_{n-1} - 1$ (for all $n > 0$ and $a^0 = 1$) is equivalent to $a_n = \prod_{1 \leq k \leq n} x_k$. Therefore, if $x_n = n$, then $a_n = n!$ so that if $x_n = 2$, then $a_n = 2^n$ and so on.

The most common iterative transformation, however, is summation. In this instance, the recurrence $a_n = a_{n-1} + y_n$(for $n > 0$ and $a_0 = 0$) is equivalent to $a_n = \sum_{1 \leq k \leq n} y_k$. Therefore, if $y_n = 1$, then $a_n = n$, and if $y_n = n$–1, then $a_n = \frac{n(n-1)}{2}$, and so on.

8.1.3. What are common elementary discrete sums?

Since discrete sums are more commonly utilized, table 8.2 shows some of the discrete sums that are frequently encountered. This list is not comprehensive, but does highlight frequently seen summations for their respective categories.

8.1.4. What is an arithmetic relation?

Let us assume the following sequence: 3, 7, 11, 15, 19, ...

Now, consider the following steps:

1. We can see that each value is 4 more than the previous value, resulting in the following recurrence equations:
   a) $a_o = 3$ (The formula for the 0^{th} number)

Category	Equation
Geometric series	$\sum_{0 \leq k \leq n} x^k = \frac{1-x^n}{1-x}$
Arithmetic series	$\sum_{0 \leq k \leq n} k = \frac{n(n-1)}{2} = \binom{n}{2}$
Binomial coefficients	$\sum_{0 \leq k \leq n} \binom{k}{m} = \binom{n+1}{m+1}$
Binomial theorem	$\sum_{0 \leq k \leq n} \binom{n}{k} x^k y^{n-k} = (x+y)^n$
Harmonic numbers	$\sum_{1 \leq k \leq n} \frac{1}{k} = H_n$
Sum of harmonic numbers	$\sum_{1 \leq k \leq n} H_k = nH_n - n$
Vandermonde convolution	$\sum_{0 \leq k \leq n} \binom{n}{k} \binom{m}{t-k} = \binom{n+m}{t}$

Table 8.2.: Common discrete sums

b) $a_n = a_{n-1} + 4$ (The general formula for any value n)

2. Generalize the formula into an arithmetic sequence: $a_n = a_{n-1} + d$.
3. Derive a closed form formula for an arithmetic sequence: $a_n = a_0 + dn$. n is a variable that represents any unknowns.
4. Use the closed form to solve for any unknown term, based on how the sequence was first initialized. For example, the sequence in step 1 had 3 as the 0^{th} term ($a_0 = 3$), so the closed form equation is $a_n = 3 + 4n$. If you wanted 3 to be the 1^{st} term ($a_1 = 3$), the closed form equation would be $a_n = -1 + 4n$.

8.1.5. What is a geometric relation?

Let us assume the following sequence: 2, 6, 18, 54, 162, ...
Now, let's go through the following steps:

1. Each term is 3 times greater than the previous, leading to the following recurrences:
   a) $a_0 = 2$
   b) $a_n = a_{n-1} \times 2 = 2a_{n-1}$
2. The recurrence is a geometric sequence: $a_n = ra_{n-1}$.
3. The closed form formula for geometric sequences (with unknowns) is $a_n = a_0 r^n$.
4. Solve for the desired term, such as $a_0 = 2 \therefore 2 \times 3^n$.

8.1.6. What is a polynomial relation?

Assume the sequence: 5, 0, -8, -17, -25, -30, ..., with a recursion equation of $a_n = a_{n-1} + n^2 - 6n$.
The following steps detail the point further:

1. In a recursion where *p(n)* is any polynomial within n, the closed form formula is one degree higher than the degree of p. In other words, the degree of p in the equation of step 1, above, is n^2 ; therefore, the polynomial closed form is cubic: 2 + 1 = 3.
2. With that information, we create the general form of a polynomial with the required degree. Since the equation in step 1 is quadratic, we need a cubic to represent p: $a_n = c_3n^3 + c_cn^2 + c_1n + c_0$.
3. With a general cubic, four unknown coefficients are present, requiring us to use four terms from the sequence, such as the first four: 5, 0, -8, -17.
4. Set each term to a corresponding equation. The benefit of using the 0^{th} term is that we already know what the constant term of the polynomial is. For each term, the c_1 variable is incremented by one, and the subsequent c variables are multiplied by the previous value.
   a) $5 = c_0$
   b) $0 = c_3 + c_2 + c_1 + c_0$
   c) $-8 = 8c_3 + 4c_2 + 2c_1 + c_0$
   d) $-17 = 27c_3 + 9c_2 + 3c_1 + c_0$
5. We won't go into the math of linear recurrence equations of polynomials, but the final result is $a_n = \frac{1}{3}n^3 - \frac{5}{2}n^2 - \frac{17}{6}n + 5$.

8.1.7. What is a linear relation?

This method solves for any recurrence where the n^{th} term is a linear combination of the previous k terms.

1. Assume the sequence 1, 4, 13, 46, 157, ... and the equation $a_n = 2a_{n-1} + 5a_{n-2}$.
2. Determine the characteristic polynomial by replacing each a_n term with x^n and dividing by $x^{(n-k)}$:
   a) $a_n = 2a_{n-1} + 5a_{n-2}$
   b) $x^2 = 2x + 5$
   c) $x^2 - 2x - 5 = 0$
3. Solve the characteristic polynomial. The quadratic equation can be used in this example as we have a degree of 2:
   a) $x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$, where *a=1, b=-2, c=-5*
   b) $x = 1 \pm \sqrt{6} \therefore a_n = c_1(1 + \sqrt{6})^n + c_2(1 - \sqrt{6})^n$
4. Use the first two terms from the sequence to create the linear equations for the initial conditions
   a) *n=0* term is 1, therefore $c_1 + c_2 = 1$
   b) *n=1* term is 4, therefore $c_1(1 + \sqrt{6}) + c_2(1 - \sqrt{6}) = 4$
   c) $c_1 = \frac{2+\sqrt{6}}{4}$
   d) $c_2 = \frac{2-\sqrt{6}}{4}$
5. Insert $c_1$and c_2 into the general formula from step 4 for the solution: $a_n = \frac{2+\sqrt{6}}{4}(1 + \sqrt{6})^n + \frac{2-\sqrt{6}}{4}(1 - \sqrt{6})^n$.

8.1.8. What is a generating function relation?

The sequence 2, 5, 14, 41, 122, ... with the recurrence relation cannot be solved by any of the previous methods, so we will use a generating function to solve it. Generating functions are a formal power series where the coefficient of x^n is the n^{th} term of the sequence.

1. For this sequence, the generating function is $A(x) = \sum_{k=0}^{\infty} a_k x^k$.
2. Next, we need to find an equation that will solve for *A(x)*.
   a) Extract the initial term: $A(x) = 2 + \sum_{k=1}^{\infty} a_k x^k$
   b) Substitute the recurrence relation: $A(x) = 2 + \sum_{k=1}^{\infty} (3a_{k-1} - 1)x^k$
   c) Split the sum: $A(x) = 2 + \sum_{k=1}^{\infty} 3a_{k-1}x^k - \sum_{k=1}^{\infty} x^k$
   d) Extract constants: $A(x) = 2 + 3x \sum_{k=1}^{\infty} a_{k-1}x^k - \sum_{k=1}^{\infty} x^k$
   e) Use the formula for the sum of a geometric series: $A(x) = 2 + 3xA(x) - \frac{x}{1-x}$
3. Find the generating function *A(x)*
   a) $(3x-1)A(x) = \frac{x}{1-x} - 2$
   b) $A(x) = \frac{\frac{x}{1-x} - 2}{3x-1}$
   c) $A(x) = \frac{3x-2}{(3x-1)(1-x)}$
4. Find the coefficient of x^n in *A(x)*. Depending on what the actual equation of *A(x)* looks like, the methodology will vary. In this example, we use the method of partial fractions and the generating function of a geometric sequence.
   a) $A(x) = \frac{3}{2}(\frac{1}{1-3x}) + \frac{1}{2}(\frac{1}{1-x})$
   b) $A(x) = \frac{3}{2}(1 + 3x + 9x^2 + \ldots + 3^k x^k + \ldots) + \frac{1}{2}(1 + x + x^2 + \ldots + x^k + \ldots)$
   c) $A(x) = \sum_{k=0}^{\infty} (\frac{3^{k+1}+1}{2})x^k$
5. Finally, we write the formula for an by identifying the coefficient of x^n in the generating function: $a_n = \frac{3^{n+1}+1}{2}$.

8.1.8.1. More on generating functions

We just looked at how to derive the formula for a generating function, but there is more to them than solving for the formula. Generating functions are a principle part of algorithm analysis. A primary goal of analysis is to create a specific expression, given a set of values in a sequence. The expression should measure a performance parameter, since working with a single mathematical object is easier than trying to work with the entire sequence.

In the generating function section above, we started with the equation $A(x) = \sum_{k=0}^{\infty} a_k x^k$; it may also be shown as $A(x) = \sum_{k \geq 0} a_k x^k$. This is known as an ordinary

generating function (OGF). There are a number of different OGFs, depending on the sequence to be solved; the table below shows some of the most common sequences and their associated OGF.

Sequence	OGF
$1, 1, 1, \ldots, 1$	$\frac{1}{1-k} = \sum_{n\geq 0} k^n$
$0, 1, 2, 3, 4, \ldots, n$	$\frac{k}{(1-k)^2} = \sum_{n\geq 1} nk^n$
$0, 0, 1, 3, 6, 10, \ldots, \binom{n}{2}$	$\frac{k^2}{(1-k)^3} = \sum_{n\geq 2} \binom{n}{2} k^n$
$0, \ldots, 0, 1, m+1, \ldots, \binom{n}{m}$	$\frac{k^m}{(1-k)^{m+1}} = \sum_{n\geq m} \binom{n}{m} k^n$
$1, m, \binom{m}{2}, \ldots, \binom{m}{n}, \ldots, m, 1$	$(1+k)^m = \sum_{n\geq 0} \binom{m}{n} k^n$
$1, m+1, \binom{m+2}{2}, \binom{m+3}{3}, \ldots$	$\frac{1}{(1-k)^{m+1}} = \sum_{n\geq 0} \binom{n+m}{n} k^n$
$1, 0, 1, 0, \ldots$	$\frac{1}{1-k^2} = \sum_{n\geq 0} k^{2n}$
$1, c, c^2, c^3, \ldots, c^n$	$\frac{1}{1-ck} = \sum_{n\geq 0} c^n k^n$
$1, 1, \frac{1}{2!}, \frac{1}{3!}, \frac{1}{4!}, \ldots, \frac{1}{n!}$	$e^k = \sum_{n\geq 0} \frac{k^n}{n!}$
$0, 1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots, \frac{1}{n}$	$\ln \frac{1}{1-k} = \sum_{n\geq 1} \frac{k^n}{n}$
$0, 1, 1, 1+\frac{1}{2}, 1+\frac{1}{2}+\frac{1}{3}, \ldots, H_n$	$\left(\frac{1}{1-k}\right)\left(\ln \frac{1}{1-k}\right) = \sum_{n\geq 1} H_n k^n$
$0, 0, 1, 3\left(\frac{1}{2}+\frac{1}{3}\right), 4\left(\frac{1}{2}+\frac{1}{3}+\frac{1}{4}\right), \ldots$	$\left(\frac{1}{(1-k)^2}\right)\left(\ln \frac{1}{1-k}\right) = \sum_{n\geq 0} (H_n - 1) k^n$

Table 8.3.: Ordinary generating functions

There is a lot more to generating functions than we can cover in this book. However, you should recognize that, among all the things covered in this chapter, generating functions may be the most useful things to become familiar with.

8.1.9. What are harmonic numbers?

In number theory, the n^{th} harmonic number is the sum of the reciprocals of the first n natural numbers. Harmonic numbers are related to the harmonic mean because the n^{th} harmonic number is also n times the reciprocal of the harmonic mean of the first n positive integers.

The actual equation for harmonic numbers is $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k}$.

You may also see it expressed as $H_n = \sum_{1\leq k\leq N} \frac{1}{k}$.

We can see an application of harmonic numbers if we consider an analysis of the quicksort algorithm (see section 16.2.1). This analysis requires solving recurrence relations that directly mirror the recursive nature of the quicksort algorithm. However, the recurrences are based on probabilistic statements of the inputs.

If we assume C is a particular comparison, C_n is the average number of compares to sort n elements, and $C_0 = C_1 = 0$, the total average number of compar-

isons is found by adding the number of compares for the first partitioning stage *(n + 1)* with the number of compares for the subarrays after partitioning.

The probabilistic part comes when the partitioning element is the k^{th} largest; this occurs with the probability $\frac{1}{n}$ for each $1 \leq k \leq n$. In this instance, the subarrays after partitioning are sizes *k - 1* and *n - k*. Therefore, the derived equation is $C_n = n + 1 + \frac{1}{n} \sum_{1 \leq k \leq n} (C_{k-1} + C_{n-k})$, for all values of *n>1*.

Now that we have created a mathematical equation for the algorithm, we can work towards a solution.

1. First, change *k* to *n - k + 1* in the second part of the sum $C_n = n + 1 + \frac{1}{n} \sum_{1 \leq k \leq n} (C_{k-1} + C_{n-n-k+1})$ which becomes $C_n = n + 1 + \frac{2}{n} \sum_{1 \leq k \leq n} C_{k-1}$for $n > 0$.

2. Next, to remove the sum, we multiply by *n* and subtract *(n + 1)*: $nC_n - (n-1)\,C_{n-1} = 2n + 2C_{n-1}$for $n > 1$.

3. Rearrange the terms to get a recurrence: $nC_n = (n+1)C_{n-1} + 2n$ for $n > 1$.

4. Divide by sides by *n(n + 1)*: $\frac{C_n}{n+1} = \frac{C_{n-1}}{n} + \frac{2}{n+1}$ for $n > 1$.

5. Through iteration, we create the following summation to complete the proof, since $C_1 = 0$: $\frac{C_n}{n+1} = \frac{C_1}{2} + 2 \sum_{3 \leq k \leq n+1} \frac{1}{k}$.

8.2. What are asymptotic approximations?

It's important to recognize that there may be a trade-off between simplification and accuracy of a solution. Ideally, the solution will be both simple and accurate, but this may not always be the case. Sometimes, recurrences may not have an exact solution. However, we can probably develop an estimate of the rate of growth and, therefore, derive an asymptotic estimate that is reasonably accurate.

There are a variety of ways to estimate the solution, or at least approximating the exact solution. Due to space, we will not cover all of those ways here, but will cover several of them to demonstrate how they work.

One purpose of asymptotic approximation is that it allows you to "convert" between discrete mathematic functions, such as harmonic numbers or binomials, to more traditional functions, such as logarithms or powers. This is because these different function types are closely related, though certain functions are better suited for particular problems.

As problems get bigger in terms of general "size" or scope, they tend to become more accurate, much like the Law of Big Numbers. Or, at least, we are most interested in the the problems where approximations become more accurate. If an analysis problem can expressed with a single term, *N*, then it can be generally defined via asymptotic series *N* and $\log N$, and the initial terms are frequently very accurate estimates in a large number of algorithm analyses. This applies whether the overall series is convergent or divergent, which means asymptotic approximation is ideal for a wide range of analyses.

8.2.1. Asymptotic notation

The following definitions and expressions have been used for decades, so it is essential to understand them in the context of asymptotic analysis to ensure precise statements about function approximations.

Given the function $f(n)$:

- $g(N) = O(f(N))$ only if $\left|\frac{g(N)}{f(N)}\right|$is bounded from above as $N \to \infty$
- $g(N) = o(f(N))$ only if $\frac{g(N)}{f(N)} \to 0$ as $N \to \infty$
- $g(N) \sim f(N)$ only if $\frac{g(N)}{f(N)} \to 1$as $N \to \infty$

However, Big-O notation allows us to create approximations using a small set of basic algebra calculations, so it is comparatively rare that the definitions above are actually used to determine asymptotic values. Big-O is used to bound the error terms that have much smaller values than the main term; in normal practice, it is assumed that the error terms will be insignificant compared to *N*.

To approximate *g(N)* by calculating *f(N)*, we can use the equation $g(N) = f(N) + O(h(N))$, where the error is bounded above by *h(N)*. However, we can use the equation $g(N) = f(N) + o(h(N))$ to indicate that the error gets smaller compared to *h(N)* as *N* gets larger. Finally, the function $g(N) \sim f(N)$ is used to express the smallest, non-trivial approximation of $g(N) = f(N) + o(f(N))$.

8.3. What is amortized analysis?

Amortized analysis is an alternative take to time and space complexity analysis. Rather than looking at the worst-case run time per operation, amortized analysis considers the overall algorithms involved. In other words, it considers the fact that certain operations within an algorithm may be costly, while others are less so. Therefore, the gestalt algorithm may not be as bad as a time/cost analysis may indicate, as a single bad operation may not actually be that bad if it occurs infrequently.

This concept is actually fairly intuitive from normal life. For example, when baking bread, you effectively have two operations to consider: mixing the dough and cooking it. Compared to the baking time, making the dough is fast. Thus, most people would say the process is a moderately long process, all things considered.

If you expand the process from one loaf of bread to one hundred, you can still consider the process to be "medium speed". You can mix the dough for each loaf individually and bake each one individually, or you can make the dough for all one hundred at one time, then (assuming you have a small oven), bake each one individually. Regardless, you still have one or more fast operations followed by multiple slow operations.

If you take this back into the computer realm, there are a number of areas where this same concept can apply. Single-threaded vs. multi-threaded programming is a good example, especially if you have a shared memory block or have to wait on I/O operations. Even in a multi-threaded program, there are times when you simply can't move forward until another process is complete, which is equivalent to having all your dough ready but waiting on the oven.

8.3.1. What methods of amortized analysis exist?

Three methods are used to perform amortized analysis:

- aggregate method
- accounting method
- potential method

All three provide a correct answer, though they use different methodology to derive the answer. Thus, the method used depends on the user and the application, as one method may work better for the particular situation.

8.3.1.1. What is the aggregate method?

Aggregate analysis determines the worst case, upper bound *T(n)* for a sequence of n operations, resulting in $\frac{T(n)}{n}$ per operation, on average. Thus, in aggregate, each operation has the same cost.

If we consider a stack, it is a linear data structure that allows for *push()* and *pop()* operations. Each function call has a constant time for completion; a total of n operations results in the total time being *O(n)*.

If we add the functionality to allow multiple pop operations, we can determine what the new Big-O value is. If we assume *multipop(x)* will continue popping elements from the stack x times (assuming there are enough elements in the stack), we can see that the operation is not constant: *multipop()* can run, at most, n times (where n is the number of stack elements), so its worst-case time is *O(n)*.

We already know that we have to have pushed data to the stack before we can pop it, and *push()* is an *O(n)* operation. Therefore, *multipop(n)* is an $O(n^2)$ operation. However, this isn't the final answer. Because *multipop()* can only take *O(n)* time if there were n pushes to the stack, in the very worst case, we have *push(n)* (all constant time operations), and a single *multipop()* of *O(n)* time.

Therefore, for any value of n, any sequence of *push()*, *pop()*, and *multipop()* takes *O(n)* time. Using aggregate analysis, we calculate $\frac{T(n)}{n} = \frac{O(n)}{n} = O(1)$. Thus, the amortized cost is *O(1)* per operation, or constant time, regardless of the data set.

8.3.1.2. What is the accounting method?

The accounting method takes concepts from accounting, hence the term. Each operation receives an amortized cost, which may be different from its actual cost; operations may receive a amortized cost that is higher or lower than its actual cost. Amortized costs that are higher than their actual costs are credited to the process to "pay for" operations that have amortized costs that are lower than their actual costs.

Because credit starts at zero, and can never be negative, the actual cost of an operation sequence is the amortized cost minus the accumulated credit. This assigns the amortized cost as an upper bound on actual cost. Thus, a large number of small, short running operations can increase the credit, while rarer, long running operations will decrease it.

Each operation can have a different amortized cost, but the costs must always be the same for a given operation regardless of the sequence of operations. Some

operations will incur higher costs than others, but can be set to occur within the algorithm only when sufficient credit is available to pay for the operation.

For this example, we will consider our stack from the previous section, with *push()*, *pop()*, and *multipop()* functions.

The actual costs are as follows:

- *push() = 1*
- *pop() = 1*
- *multipop() = min(size, k)*

multipop() has a variable, actual cost because it will either be k when k is less than the number of stack elements, or the cost will be the total number of stack elements.

We will assign the following amortized costs to each function:

- *push() = 2*
- *pop() = 0*
- *multipop() = 0*

We use these values because we declare *push()* operations to cost more because more work is required compared to *pop()*, which isn't much more than a memory dereference. *multipop()* utilizes *pop()*, so while its actual cost is variable, its amortized cost is the same as *pop()*.

To push an item onto the stack, it costs 2 amortized units. However, the true cost is only 1 (after subtracting the actual cost), so we have a credit of 1 after a *push()* operation. At any point in time, each stack element has a credit of 1 associated with it.

When popping elements from the stack, the actual cost is 1 but the amortized cost is 0, for a net debit of -1. However, that -1 debit is nullified by the net credit of 1 for each stack element. Therefore, to pop an element from the stack, we have a total cost of $1 - 1 = 0$. This also applies to *multipop()*.

This way, our credit is never negative; since attempting to pop an element from an empty stack won't work, we cannot go negative. However, popping the final element will leave us with a credit balance of 0. Our next push will result in a positive credit balance of 1. The worst case scenario of n operations is *O(n)*, with an amortized cost of *O(1)*.

8.3.1.3. What is the potential method?

The potential method is similar to the accounting method, except that rather than cost and credit, work already completed is potential energy that can pay for future operations, much like storing potential energy to be used as work later. Another key point is that the potential method applies to the whole data structure, rather than an individual operations.

There are quite a few assumptions and definitions associated with the potential method. If we assume a data structure *(D)* starts at D_0, then a sequence of operations becomes D_1, D_2, D_3, ..., D_n. Assume c_i is the cost of the i^{th} operation and D_i is the data structure resulting from the i^{th} operation.

We define Φ to be the potential function that maps D to a number $\Phi(D)$, which is the potential associated with that particular data structure. Finally, the amortized cost of the i^{th} operation is $a_i = c_i + \phi(D_i) - \phi(D_{i-1})$.

Therefore, the total amortized cost over n operations is
$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (c_i + \phi(D_i) - \phi(D_{i-1}))$.
All $\Phi(D_i)$ terms except $\Phi(D_0)$ and $\Phi(D_n)$ cancel out, leaving the final function
$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} c_i + \phi(D_n) - \phi(D_0)$.
To ensure that the total amortized cost is an upper bound to the total actual cost for any sequence of length n, it is assumed that $\Phi(Di) \geq \Phi(D0)$ for all i. Typically, $\Phi(D_0)$ is defined to be zero and the potential is defined to always be non- negative.

What all this means is that, when a sequence of operations is performed, the i^{th} operation will have a potential difference of $\Phi(Di) - \Phi(Di-1)$. If this difference is positive, the amortized cost (a_i) is an overcharge for the operation and the potential energy increases. If the difference is negative, we have an undercharge and the potential energy decreases.

Applying this to our previous stack example, the potential function is the number of the items in the stack, defining $\Phi(D_0) = 0$, as there are no items in the stack. This also defines $\Phi(D_i)$ to always be non-negative.

If we calculate the potential difference for a *push()* operation, we come up with the following: $\phi(D_i) - \phi(D_{i-1}) = (stack.size + 1) - stack.size = 1$. Therefore, the amortized cost of a push operation is $a_i = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 + 1 = 2$.

The potential difference for a multipop is and the amortized cost is $\phi(D_i) - \phi(D_{i-1}) = -min(stack.size, k)$ and the amortized cost is $a_i = c_i + \phi(D_i) - \phi(D_{i-1}) = min(stack.size, k) - min(stack.size, k) = 0$. Since *multipop()* is similar to *pop()*, they have the same costs.

Ultimately, what this means is that all operations have an amortized cost of *O(1)*, so worst case scenarios take *O(n)* time.

8.4. What are cost models?

Program efficiency, as it pertains to time, is ultimately concerned with what a "program step" is defined as. The definition is key, because the time to perform a single step must be bounded above by a constant in order for the analysis to correspond with actual execution time.

In addition, a step must account for realistic operations. For example, a single step may be defined as adding two numbers. But if the numbers are arbitrarily large, or involved non-integers such as floats or complex numbers, the addition operation may no longer be assumed to be a constant value.

To address this, two operation cost models are normally used: uniform cost and logarithmic cost. However, it should be noted that the lower bounds for many published problems are often for a computation model that is more restricted than a set of real-world operations. Thus, some algorithms may actually be faster than perceived.

8.4.1. How does uniform cost differ from logarithmic cost?

The uniform cost model assigns a constant cost to each machine operation, regardless of the number size used. In other words, every instruction takes a single unit of time and every register requires a single unit of space.

The logarithmic cost model assigns a cost to every machine operation, proportional to the number of bits involved; that is, every instruction takes a logarithmic

number of time units, in terms of operand length, and every register takes a logarithmic number of units of space. Naturally, this model is much more complex to use correctly, so it is only implemented when necessary, such as for cryptography or other applications that have arbitrary-precision arithmetic.

An example of these two models first assumes we are adding two numbers, such as 256 + 32. In binary terms, 256 is one byte, or eight bits, while 32 is five bits. Since eight bits is greater than five, we have to perform the addition with eight bits. If we think about each bit representing a power of 2 column, i.e. 1s column, 2s column, 4s column, etc., then we have to make eight addition calculations.

In uniform cost, the entire set of eight calculations would be defined as taking one unit of time. With logarithmic cost, each of the column additions would be a unit of time, resulting in eight units to perform the addition.

Therefore, the exact same operation is effectively 8x longer when using logarithmic cost modeling. But you must remember that logarithmic cost is proportional to the number of bits involved, so the same operation with only five bits, say adding 32+32, would only be 5x longer than uniform cost modeling.

Needless to say, without having a normalized value to compare operations, logarithmic cost modeling is difficult to standardize between operations that use different quantities of bits. Uniform cost assumes a constant value for each operation, regardless of the number of bits.

8.5. How to analyze parallel algorithms?

Parallel algorithms, for the most part, use the same analysis tools as traditional, sequential algorithms, with the caveat that most parallel algorithms attempt to improve the computing time without necessarily attempting to improve other resource use.

8.5.1. What are the work and span laws?

For the following discussion, assume that p is the number of processors in a computer and T_p is the time between start and completion of a computation.

Computation *work* is the total number of primitive operations performed by p processors. If we assume that the synchronization communications overhead of the processors is zero, the work time is equal to the computation time of a single processor, T_1.

Computation *span* is the length of the longest sequential operation series necessary due to data dependencies, also known as the critical path. (This is similar to the critical path in project management, which is the minimum time required to complete a project.) Naturally, minimizing the span in a program is critical in parallel operations, as it determines the shortest possible execution time due to traditional, sequential processing.

Span is commonly defined as the time spent computing using an ideal machine with an infinite number of processors, T_∞.

Computation *cost* is the total time spent both computing and waiting, for all processors, pT_p.

With all these values, we can then define the work law and span law.

- Work law: Because p processors can only perform, at most, p operations in parallel, the computation cost is always, at least, $pT_p \geq T_1$.

- Span law: Because there is no such thing as unlimited processors, p processors will always be less than infinity, $T_p \geq T_\infty$.

8.5.2. What are the measures of parallel performance?

With the laws and definitions above, we can develop equations for parallel performance.

- Speedup: The speed increase using parallel operations compared to sequential operations, $S_p = \frac{T_1}{T_p}$. When the speedup is lower-bound $\Omega(n)$, the speedup is linear, since the work law implies $\frac{T_1}{T_p} \leq p$. An algorithm that has linear speedup is scalable. Perfect linear speedup is defined as $\frac{T_1}{T_p} = p$.
- Efficiency: The speedup per processor, $\frac{S_p}{p}$.
- Parallelism: The maximum speedup possible for any number of processors, $\frac{T_1}{T_\infty}$. By virtue of the span law, speedup is bounded by parallelism, by virtue of, if $p > \frac{T_1}{T_\infty}$ then $\frac{T_1}{T_p} \leq \frac{T_1}{T_\infty} < p$.
- Slackness: The ratio of one processor vs. multiple processors: $\frac{T_1}{pT_\infty}$. Per the span law, a slackness <1 indicates that perfect linear speedup is impossible with p processors.

8.6. Summary

In this chapter we looked at algorithm analysis, particularly in solving recurrence relations and amortized analysis to determine Big-O values. We also looked at cost analysis and parallel programming analysis.

This chapter should have been review for many people, particularly Computer Science graduates, as we reviewed such skills and concepts as mathematical and algorithmic relations, amortized analysis, calculating software costs, parallelization of tasks, etc. Obviously, if you aren't familiar with these things, such as being a self-taught programmer, make sure you understand the basics before you go to an interview.

This chapter was a little heavy in math but lacking in the actual calculations, so plan on brushing up on your math analysis to justify the answers presented here. Of course, the type of job you are applying for helps determine what type of interview questions you get.

While it's possible you will be asked about algorithm analysis, the odds are probably in your favor that you won't get one. Even if you do, it probably won't ask you to go this far into the details. For example, a simple web development job will probably ignore this area entirely, while a job developing a new encryption algorithm will almost certainly ask something about algorithm analysis.

In the next chapter, we will look at linear data structures, such as Python sequences, linked lists, stacks and queues, and similar structures.

9. Linear Data Structure

This chapter discusses linear data structures. Data structures are a way of organizing and storing data to more efficiently find and retrieve the data. Many data structures are designed to store the data sequentially, hence the term "linear". Other data structures link data in a non-linear way, but one which reflects the special relationship between the data.

In this chapter, we will talk a lot about non-primitive data structures. In computer science, a primitive data type is a basic building block used to create higher-level data structures. For example, in some languages, char is a primitive type that is used to build higher-level text strings. Thus, non-primitive data structures, such as arrays, linked lists, etc., are built on primitive data types.

In addition, we will look at common concepts in programming, such as linked lists, and how they are commonly utilized in many languages as well as how Python utilizes them. There is not always a direct match between Python and the basic concepts; sometimes you will have to know how to manually create the solution due to Python's alternative formulation.

Some other data structures we will cover are stacks vs. queues and when each type should be utilized, hash tables vs. Python dictionaries, and why deques are the best solution in some instances.

In this chapter, we will cover the following:

- What is an array?
- How do Python sequences work?
- What is a linked list?
- What is the difference between stacks and queues?
- What does deque do?
- What is a hash table?

9.1. What is an array?

An array is a data structure that stores homogeneous values in a sequential, contiguous fashion. The elements of the array are identified via an index value. While one-dimensional arrays are a single linear list of elements, multi-dimensional arrays can be created for different needs, such as storing the x, y, z values of a 3D object or storing data in a tabular form.

The array size must be stated prior to storing data. This is because elements in an array must have the same size and should use the same data representation. This facilitates the computation of element indices at run time, which allows a single, iterative statement to process array elements.

Arrays are typically implemented by array structures, but in some languages they can be created by hash tables, linked lists, search trees, etc. In Python, lists perform the functionality of arrays while providing for additional features, such as allowing for disparate data types.

Python does have an array data structure, but it is primarily used for storing numbers used in mathematical calculations; NumPy is a key library for array use. The key factor when using the Python array is that all data has to be the same type.

The primary differences between arrays and lists in Python is that arrays perform mathematical operations on each element, whereas a list is considered a single object, so you have to create a *for* loop to iterate over each element to perform the operation. Also, arrays tend to store data in a more compact fashion, as there isn't the metadata overhead that lists have.

9.2. How do Python sequences work?

A key point to Python sequences is that they are iterable, meaning they actually included the *__iter__()* method. In other words, an iterable object will return data, one value at a time. When combined with the *__next__()* method, they create the *iterator protocol*; together, those two methods are what make iterable objects work within Python.

Technically, a Python sequence is an iterable object that supports random access; you can call any element within the sequence without having to worry about the elements prior to the desired one. This capability is part of how slicing works.

Random access is supported by element indexing; you simply provide the index value for the desired element and it is returned. If the element access is provided by a key instead of position, you have a mapping structure; in Python, this is a dictionary.

If we want to create a *for* loop without actually using *for*, you have to utilize iterables, as shown in the following example:

```
def print_elements(iterable):
    iterator = iter(iterable)
    while True:
        try:
            item = next(iterator)
        except StopIteration:
            break # No more values
        else:
            print(item)

if __name__ == "__main__":
    print_elements([1, 2, 3])
```

When we run this code, we get figure 9.2.1. In essence, the example above performs the same functionality that is done in a *for* loop. The *while* loop continues operating as long as there is data within an iterable sequence. The *iter()* call actually gets the iterator, then the *next()* call pulls the next value in the sequence continuously until the *StopIteration* exception is generated.

In short, anything that you can loop over is an iterable. Thus, list comprehensions, tuple unpacking, *for* loops, etc. all utilize the iterator protocol.

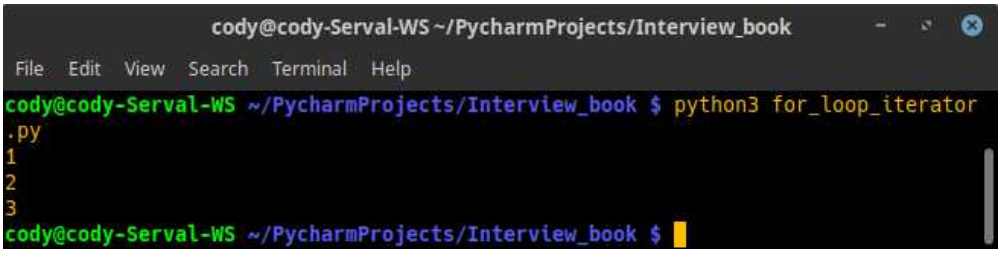

Figure 9.2.1.: Iterator loop

9.3. What is a linked list?

A linked list is similar to an array, in that each element is a separate object. However, each element (node) in a linked list is comprised of two items: the data itself and a reference to the next node.

Table 9.1 is a list of differences between arrays and linked lists:

Linked List	Array
Require iteration through each node to reach the desired one	Indexed, so you can simply call the desired element directly
Dynamic	Fixed-length
Insertions and deletions are quick	Overhead causes insertions and deletions to be slower
Slow traversal to desired node	Fast node access
Memory allocated at runtime	Memory allocated during compilation
Store data randomly	Store data sequentially
Extra memory required for pointers to other nodes	Only stores actual data
More efficient in data utilization	Less efficient data utilization

Table 9.1.: Linked Lists vs. Arrays

There are several types of linked lists, as discussed below:

- Singly linked list: every node stores its data and the address of the next node in line. The final node is NULL.
- Doubly linked list: every node stores its data and the addresses of both the previous and next node in line. The first and final nodes are NULL.
- Circular linked list: all nodes are connected in a loop, where the "last" node references the "first" node (any node can be designated "first"). There are no NULL nodes.

For clarification, Python lists are not related to linked lists. They are actually dynamic arrays, so they have fast insertions/deletions at the final element, but are slow elsewhere in the series, they implement random access and they are easily cached.

You can implement linked lists within Python, but from what I found while researching this, many people discussing Python linked lists admitted that most examples are purely academic and rarely found in the real world. There are times when a doubly linked list is useful, such as Raymond Hettinger's ordered set

recipe (http://code.activestate.com/recipes/576694-orderedset/), which creates a Python set that remembers the original insertion order, but for most Python programming tasks, it's best to use the data structures already provided by the language.

9.4. What is the difference between stacks and queues?

A stack is a non-primitive data structure, formed by an ordered list that accepts new items and removes existing items from only one end of the list; this end is referred to as the Top of the Stack (TOS). Because of its nature, the TOS is the only entry/exit point into the stack. Therefore, data is accessed in a Last In/First Out (LIFO) process; this is where pushing and popping data comes into play. You push data onto the stack, then pop it out from the top.

Figure 9.4.1 represents pushing/popping data from a stack:

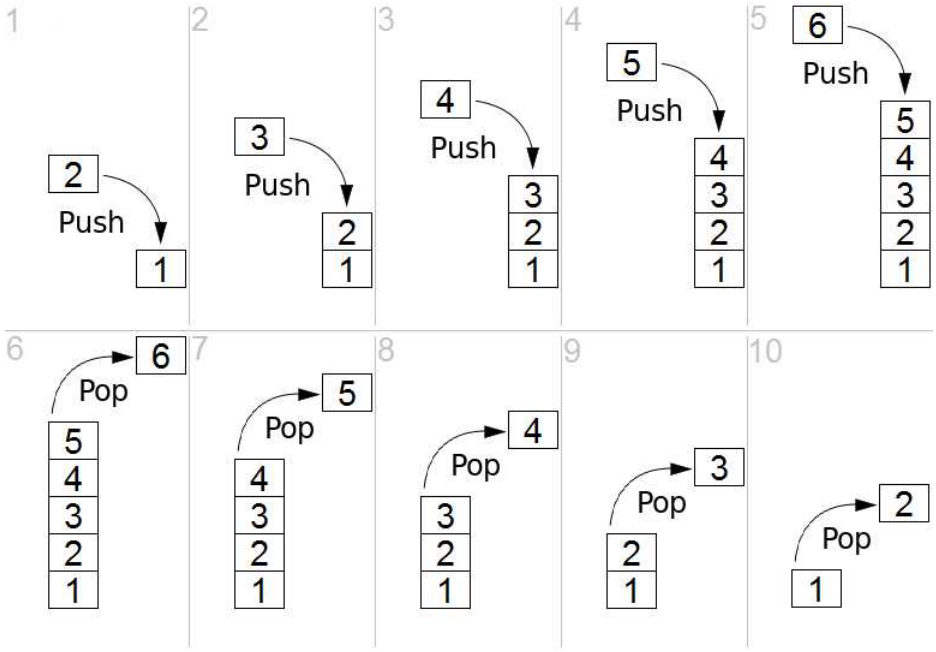

Figure 9.4.1.: Stack

A queue is another non-primitive data structure. Rather than having a single entry/exit point, you can think of it more like a tube: data comes in one side and out the other. Thus, a queue is a First In/First Out (FIFO) process. Where a stack has push and pop operations, a queue has *enqueue* and *dequeue*.

This is demonstrated in figure 9.4.2. With only one "open" location, only one address pointer is necessary to identify the TOS. However, a queue requires two pointers, one for the entrance and one for the exit of the queue. Another critical difference is that a stack only has one type, while there are multiple queue types: circular, priority, doubly ended, etc.

Finally, stacks and queues are used for different purposes, as described in the lists below.

- Stack
  - Compiler parsing

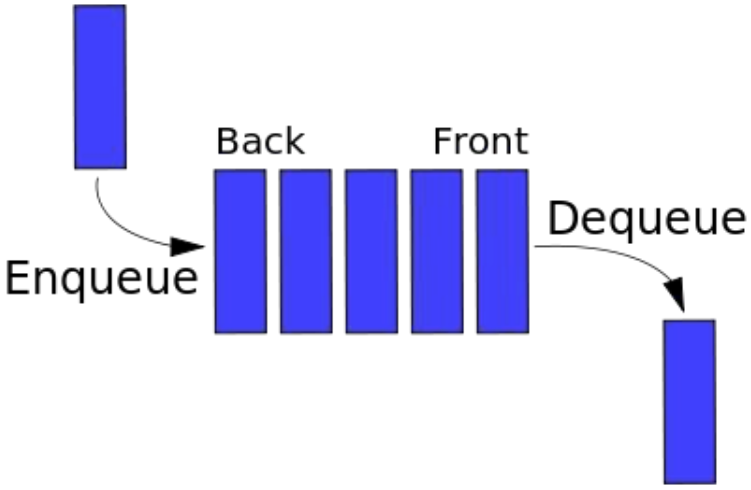

Figure 9.4.2.: Queue (Vegpuff/Wikipedia)

  - Application "undo" command
  - The "back" button of a browser
  - Compiler function calls

- Queue
  - Data buffers
  - Asynchronous data transfer
  - Resource requests
  - Enqueue
  - Dequeue
  - Traffic analysis

9.5. What does deque do?

Deque (pronounced "deck") is a double-ended queue. It is common in a number of languages, but since this book focuses on Python applications, we will look at how Python implements deque.

As it supports appending and popping from either end, it is preferred to Python lists if access to either end of the sequence is required. When performing operations, deque takes *O(1)* time, compared to *O(n)* for a normal list. Regular Python lists are optimized for fixed-length operations, meaning they assume the underlying data is relatively constant in terms of size and position. When attempting to pop or insert at the beginning of a list, things slow down because the entire list has to be modified to account for the changes in size and position of the data elements.

We won't cover all the methods available to deque, but the key ones are : *append(), appendleft(), pop(), popleft(), extend(), extendleft(),* and *rotate()*. Most of these are the same operations that a list has, but are available for both ends of the sequence. Of note, *extend()* and *extendleft()* allow the addition of values from an iterable argument, while *rotate()* rotates the data within the deque a given number of times; in other words, the data at one end is moved to the other end, kind of like popping an element from the end and moving it to the front.

9.6. What is a hash table?

In general computing terms, a hash table is a high-order data structure that implements an associative array that maps keys to values. A hash function computes an index for a key:value pair, effectively assigning a location for a value within the array while ensuring that the value's key is associated to it. In practical terms, a hash table stores the key as a hash digest which becomes the index of its value.

Python implements a hash table as the basis of the dictionary type, while a hash function is used with sets as well, though there is no association between keys and values within sets.

Because of the underlying algorithms, the order of elements is not fixed within a dictionary, as a way to prevent hash collisions. Collisions occur when the same hash digest is calculated with different input. If the hash value itself is the index of a key's value, then having a collision means the same value could be returned by different keys, or a particular key would generate an error due to the potential collision.

There are a number of different methods to prevent collisions, for example, open addressing. With an open address, a hashed key creates an initial index; if the index is already occupied, a probe sequence is implemented until an available slot is found; the "open address" simply means that the entry's location isn't determined by its hash digest.

Therefore, data stored in a dictionary isn't guaranteed to be in the same order it was inserted in versions prior to 3.7, unless you used OrderedDict from the collections library. Starting with v3.7, dictionary order is guaranteed using default dictionaries.

9.7. Summary

In this chapter, we looked at linear data structures in general terms and how they are implemented in Python. Arrays and linked lists are basic data structures common to nearly all programming languages; Python builds on those structures so that a lot of functionality is provided out of the box.

We looked at the difference between stacks and queues, seeing how they are implemented from a functional perspective, and looking at some examples of real world applications. In addition, we looked at deque, which is a double-ended queue, designed to make operations on either end of a queue faster than regular Python list.

We finished by looking at generic hash table functionality, and discussing how Python uses hash tables to implement dictionaries. We saw why, prior to v3.7, Python dictionaries did not store data in order of entry but learned that, starting with v3.7, that is no longer the case.

In the next chapter, we will take a look at the graph data type, commonly used in representing mathematical models.

10. Graphs and Trees

A graph is an abstract data type used to implement undirected and directed graphs from mathematical graph theory. Graph theory studies graphs, which are math structures that model pair relationships between objects.

A tree is an abstract data type (or data structure, depending on the implementation) that simulates a hierarchical tree structure, meaning it has a root value and a variety of branches where nodes are linked together.

Graphs are commonly used to represent network flow, data communications, and more. Because of their ability to show relationships, graphs are very useful in computer science. Similarly, trees can be used to represent hierarchical data, such as file systems, improve efficiency when searching for data, workflows for compositing digital images, and similar scenarios.

We will learn how graphs and trees differ, leading to the later use of recursion to solve these types of problems. We'll also cover how binary trees are considered a very useful data structure and some of the different implementations of binary trees.

In this chapter, we will discuss:

- What is a graph?
- What is a tree?
- What is a binary tree?
- What is special about a binary search tree?
- What other types of trees are there?

10.1. What is a graph?

Relationships between objects are often modeled using graph diagrams. The models show the relationships by using nodes (also called vertices) that are connected by lines (also called edges). A simple drawing of a graph is shown in figure 10.1.1:

There are two main types of graphs: undirected and directed. In computing, undirected graphs use unordered pairs of vertices while directed graphs use ordered pairs. Also, undirected graphs have standard edges while directed graphs have arrows, indicating the ordered direction.

Graphs are frequently used in computer science when developing algorithms, for example:

- Kruskal's algorithm: finds the edge of the least possible weight that connects any two trees in the forest.
- Prim's algorithm: finds a subset of edges forming a tree which includes every vertex where the total weight of all edges in the tree is minimized.
- Dijkstra's algorithm: finds the shortest path between nodes in a graph.

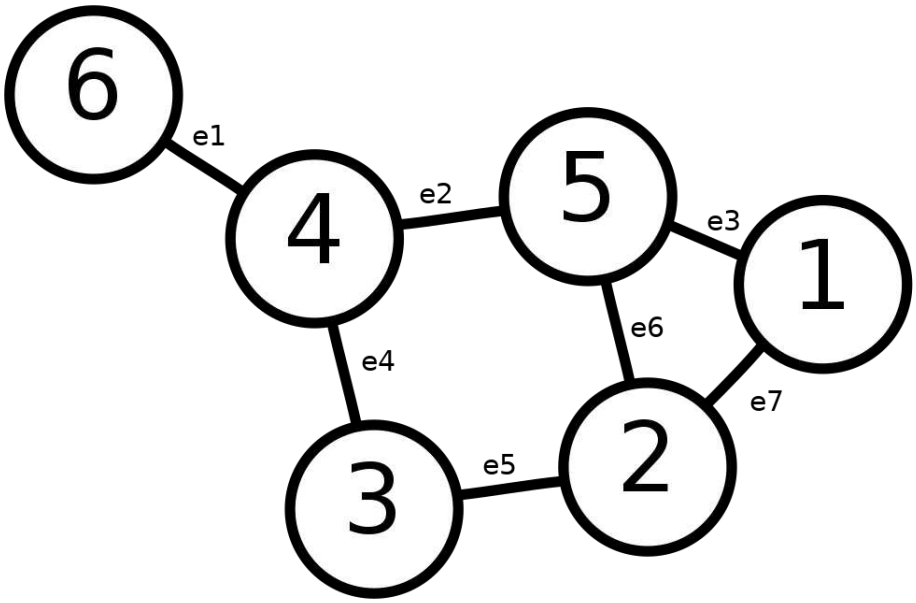

Figure 10.1.1.: Graph example

10.1.1. What graph operations are available?

The following operations are commonly found in graph data structures:

- *adjacent(x, y)*: tests whether an edge exists between vertices x and y
- *neighbors(x)*: lists all vertices that have an edge with x
- *add_vertex(x)*: if not already present, adds vertex x
- *del_vertex(x)*: if present, deletes vertex x
- *add_edge(x, y)*: if not present, adds an edge between vertices x and y
- *del_edge(x, y)*: if present, deletes the edge between vertices x and y
- *get_vertex_value(x)*: returns the value associated with x
- *set_vertex_value(x, v)*: sets the value of x to v

If the graph implementation supports, its getting and setting edge values are also available options.

10.1.2. Graph representation

As graphs are abstract, different data structures are used to represent different types of graphs, as discussed below.

10.1.2.1. What is an adjacency list?

An adjacency list is defined as a collection of unordered lists that represent a finite graph. Each list describes the neighboring set of vertices, as related to a particular vertex. For example, in the figure 10.1.1, we can see the following adjacency list representations:

- 6 is adjacent to 4
- 4 is adjacent to 6, 5, 3
- 5 is adjacent to 4, 2, 1
- 1 is adjacent to 5, 2
- 2 is adjacent to 3, 5, 1
- 3 is adjacent to 4, 5, 2

Each programming language has its own way of implementing adjacency lists, based on how they implement associations between vertices and collections, whether vertices alone are included or if both vertices and edges are tracked, and what types of objects are used to represent vertices and edges.

10.1.2.2. What is an adjacency matrix?

An alternative to adjacency lists is adjacency matrices, which use rows and columns to represent vertices and its Boolean values indicate whether an edge exists between the two vertices. In other words, the rows represent a source vertex and the columns represent the destination vertex, with a Boolean value indicating whether a connection between the two exists. Memory usage of an adjacency list is proportional to the number of edges and vertices in the graph, while an adjacency matrix would have space allocation based on the square of the number of vertices.

One benefit to using adjacency matrices is that they allow testing whether two vertices are adjacent to each other in *O(1)* time, while an adjacency list is in *O(n)* time.

An example of an adjacency matrix for figure 10.1.1 above is given below.

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

For those readers unfamiliar with matrices, coordinates are 1-6, starting in the top left corner, so the first line is vertices 1-1, 1-2, 1-3, 1-4, 1-5, and 1-6.

10.1.2.3. What is an incidence matrix?

This is a matrix that shows the relationship between vertices and edges, where rows are the vertices and columns are the edges. The intersections between rows and columns indicate whether the edge connects to the vertex.

Using the same graph example from above, an incidence matrix is provided below:

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Again, coordinates are 1-6 from the top left, so the first line would be show edge connections at 1-3 and 1-6.

10.1.3. What are some examples of graph types?

Depending on the number of vertices, edges, connections, and overall structure, a number of graph types are available. Some of the more common graph types are described below.

- Null graph: a graph that has no edges is a null graph.
- Trivial graph: a graph with a single vertex.
- Directed graph: the edges within the graph has a single direction from one vertex to another, indicated by an arrow.
- Undirected graph: the edges have no direction between vertices
- Simple graph: a graph with no loops and no parallel edges
  - The maximum number of edges in a single graph is ${}^nC_2$, where ${}^nC_2 = \frac{n(n-1)}{2}$
  - The number of simple graphs possible with n vertices is $2^{{}^nC_2}$
- Connected graph: a graph where every vertex pair has at least one edge; basically, when you start at a given vertex, you should be able to go through every edge and vertex and eventually end up at the starting point.
- Disconnected graph: a graph that does not contain at least two connected vertices.
- Regular graph: a graph where all vertices have the same degree, i.e. the same number of edge connections
- Complete graph: a simple graph where each vertex is connected to every other vertex; think of a mesh network diagram, where every node is connected to all the others.
- Cycle graph: the number of edges equals the number of vertices, such as a triangle or square.
- Wheel graph: a cycle graph that has a vertex in the center, like a pinwheel.
- Cyclic graph: a graph with at least one cycle.
- Acyclic graph: a graph with no cycles.

10.2. What is a tree?

A tree is an abstract data type that simulates a hierarchical tree structure. There is a base node (the "root"), which is often the main starting point for analysis. Children nodes create the various branches and are linked together to create subtrees.

Unlike many other data structures, a tree does not store data in a linear fashion. Instead, trees store data in a hierarchical structure. As a quick review, a hierarchical structure has a base node, with other nodes coming from it. The key is the base node, as it makes a tree different from a graph. The base node is the "root" of the tree; a graph doesn't necessarily have to have a starting point, and tracing a route could result in ending up back at the starting point.

An example of a tree is shown in figure 10.2.1. The root of the tree is actually at the top, with the branches coming off the root. The ends of each branch are considered the leaves of the tree.

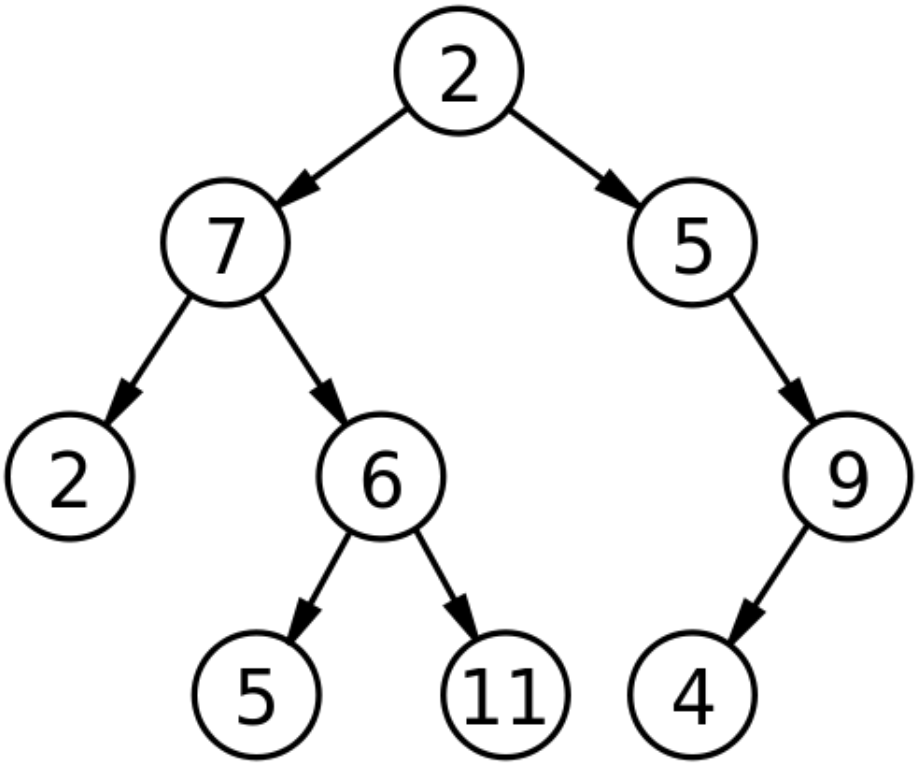

Figure 10.2.1.: Tree example

10.2.1. What properties do trees have?

As seen in figure 10.2.1, the root node (at the top) has no parent node, while every other node has a parent (linked by edges), which could be the root. Leaves are the solitary nodes at the bottom that have no children; leaves are also known as external nodes, while internal nodes have child nodes. Subtrees are created from each node, with the root containing all subtrees.

The height of a tree is the longest path to a leaf, while depth of a node is the path length from the node to the root. The root node has a depth of zero, leaf nodes have a height of zero, and a single-node tree has a height and depth of zero. By convention, a tree with no nodes (if allowed) has a height of -1.

Algorithms start at the root and move from parents to children (most common), while others start at the root, move to the leaves and work back towards the root.

10.2.2. What is tree traversal?

Moving through the nodes of a tree is called traversal, and there are two main types:

- Depth-first search (DFS): starting at the root, the search moves through all the nodes in a particular branch before backtracking to the next branch.

- Breadth-first search (BFS): starting at the root, each horizontal level is searched before moving to the next level.

10.2.2.1. What is depth-first search?

DFS has three sub-types of searching: pre-order, in-order, and post-order. Assuming we are printing each node that processed in the order it is searched, the algorithms work as follows:

- Pre-order
  1. Print the root node
  2. If left child exists, print it.
  3. Continue through all left children until no more exist, then backtrack to the parent.
  4. If right child exists, print it.
  5. Continue through all right children until no more exist, then backtrack to the root and move to the next child.
  6. In figure 10.2.1, this algorithm would result in 2-7-2-6-5-11-5-9-4.
- In-order
  1. If left child exists, print it
  2. Print the parent value
  3. If right child exists, print it
  4. Using figure 10.2.1, the result would be 2-7-5-6-11-2-5-4-9.
- Post-order
  1. If left child exists, print it
  2. If right child exists, print it
  3. Print the parent node.
  4. Using figure 10.2.1, the result would be 2-5-11-6-7-4-9-5-2.

Backtracking occurs when a branch is complete. The algorithm has to go back to the closest node and look for other branches. If present, it walks down those branches. If not, it moves back up to the next previous node and checks again.

10.2.2.2. What is breadth-first search?

BFS traverses the tree level-by-level and depth-by-depth. In other words, each level is processed completely before moving onto the next set of children. Queues are necessary to accomplish this, as each node in a level is placed into the queue, then returned when moving onto the next level. In the tree example previously shown, the output would be 2-7-5-2-6-9-5-11-4.

10.2.2.3. What is two-end BFS?

Two-end BFS is also called bidirectional search. Whereas a normal BFS searches for the shortest path in one direction, bidirectional search conducts a BFS from two directions: one from the initial state to the end, and one from the goal to the beginning. Where they meet, they will stop, indicating the shortest path has been found.

One benefit to this is that a single search graph can potentially grow exponentially, whereas a two-end search involves two, smaller graphs. Because the search ends when the two graphs intersect, there is an inherent limit to how big they will become. In Big-O terms, each of the two searches has a complexity of $O(b^{\frac{d}{2}})$, while the sum of the two searches is $O(b^{\frac{d}{2}} + b^{\frac{d}{2}})$, which is less than the complexity of a single search from the beginning: $O(b^d)$.

10.3. What is a binary tree?

Binary trees are fairly simple: each node has, at most, two children (left and right child). A child can be single, meaning a parent may only have a left or a right child, which has children below it. In the tree example above, node 9 has a single left-child while node 5 has a single right-child.

Binary trees are commonly used in two ways:

- Node access based on a value or label associated with each node. These types of trees are used to implement binary search trees and binary heaps, as well as efficient searching and sorting. In these instances, the designation of a single child as either left or right is important, particularly in binary search trees. However, the arrangement of the nodes within the tree carries no meaning: the nodes could be rearranged without damaging the inherent data structure.

- Data representation with a bifurcating structure that is relevant to data meaning. Thus, the relationship of nodes to each other, which nodes have children and whether they are left or right children, etc. is part of the data structure and rearrangement would damage that information. A family tree diagram is a good example of this. Each person has two biological parents, while each parent can have multiple children. The relationships are inherently part of the data structure of the tree; rearranging positions removes all meaning from the data.

10.3.1. What type of binary trees exist?

Definitions of the different binary tree types have not been standardized, so terms may vary depending on what you read.

- Rooted: a binary tree that has a root node and all nodes have at most two children.

- Full: also called a proper or plane binary tree, this is a tree where every node (except the leaves) has either zero or two children; no single children exist.

- Perfect: all interior nodes have two children and all leaves have the same level.

- Complete: every level (excluding the last) is completely filled and all single children in the last level are left-children.

- Infinitely Complete: every node has two children, and the tree will never end. An example of this is the Stern-Brocot tree, shown in figure 10.3.1.

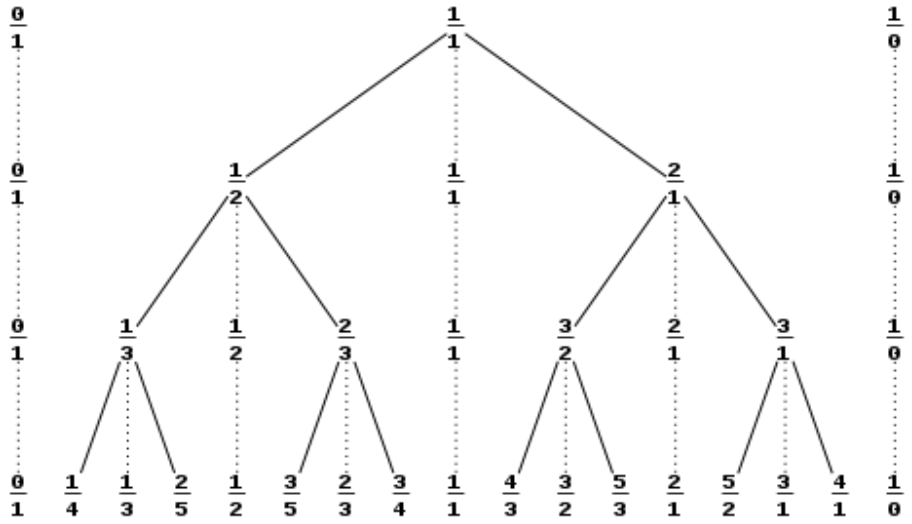

Figure 10.3.1.: Stern-Brocot infinitely complete tree (Aaron Rotenberg [CC BY-SA 3.0])

- Balanced: left and right subtrees of every node have a height difference ≤ 1.
- Degenerate: each parent node has only one child node. In practice, the tree will function like a linked list.

10.4. What is special about a binary search tree?

A binary search tree (BST) is a special type of binary tree, where all values are in a sorted order to allow searching, lookup, and other operations to be quick and efficient. A key factor is that the value of a BST node is larger than the value of the children of its left child, but smaller than the offspring of its right child. Figure 10.4.1 is an example of what this means.

It's easy to see that, moving to the right, numbers get larger. All the children to the left of the root node are less than the root node's value, while the children to the right are greater.

10.5. What other types of trees are there?

Depending on the application and need for alternative data structures, a variety of other data trees are available. The following list provides more information about some of these different trees.

- Interval tree: holds intervals (a range of numbers). These are optimized to identify which intervals overlap with a given interval.
- Segment tree: also called a statistic tree, these are used to store intervals (segments). These are optimized for identifying intervals that contain a given point.
- Range tree: store points and are optimized for identifying which points lie within a given interval.

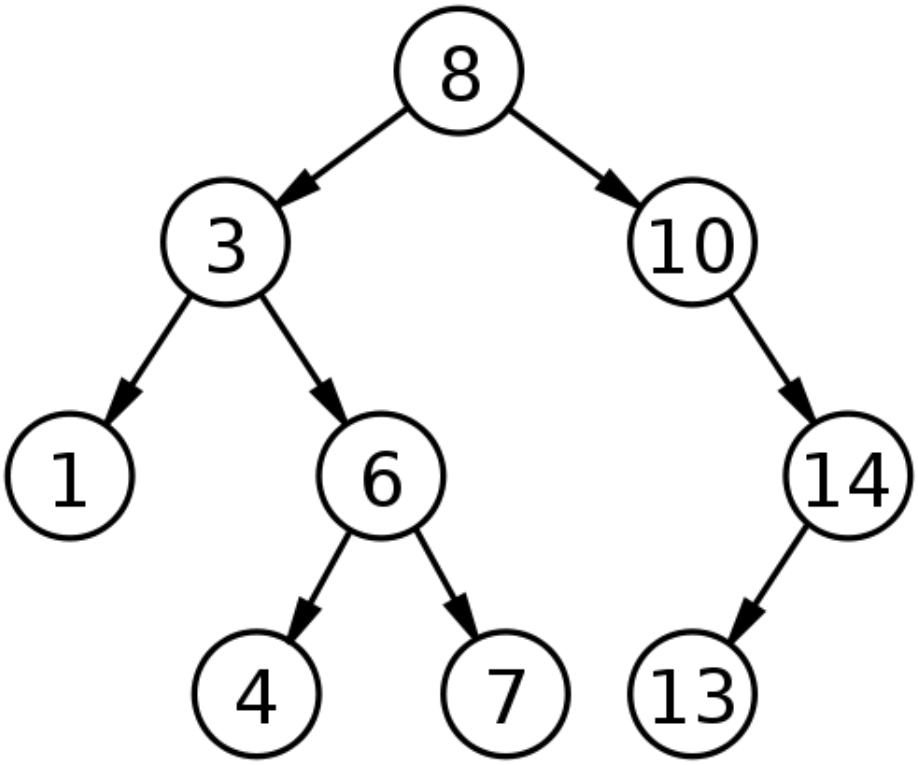

Figure 10.4.1.: Binary search tree

- Augmented tree: built from a simple ordered tree (such as a binary search tree), these are ordered by the low value of the intervals. However, an added note is placed with each node to indicate the maximum upper value of all the intervals from this particular node and its children. Therefore, when searching, you can skip all right-children that have a low value past the end of the given interval, and all nodes that have a maximum high value below the start of the given interval.
- Suffix tree: a compressed trie containing all the suffixes of a given text as their keys and text position as their values. A trie is an ordered tree where the node positions within the tree define their associated key, rather than storing the associated key within the node. A string suffix is any substring which includes the the last letter.
- Suffix array: an array created from a suffix tree by completing a DFS traversal of the tree. Any suffix tree-based algorithm can be replaced with a suffix array algorithm to provide additional information and solves the same problem in the same time complexity. Suffix arrays improve on suffix trees by providing improved space requirements, simpler linear time construction algorithms, and better local caching.

10.6. Summary

In this chapter, we talked about graphs and trees. Graphs model nodes and edges, while trees are essentially specialized graphs. As we saw in the chapter, graphs can be directed or undirected, depending on the data relationship between nodes.

Trees are commonly used to hold information that needs to be easily searched, so many tree formats are optimized for particular search patterns or data storage.

You should also have an understanding of depth-first searching vs. breadth-first searching in trees, matrix calculations for graphs, what makes binary trees special from regular trees, how binary search trees are optimized for searching and lookups, and have a basic understanding of the different types of graphs and trees you may encounter in different projects.

In the next chapter, we will talk about heaps and priority queues, which frequently make use of trees, such as binary search trees, for data queries.

11. Heap and Priority Queue

Priority queues are abstract data types that operate like a regular queue or stack, but there is a certain priority associated with them. Heaps are data structures used to implement priority queues, so, in this sense, they should not be confused with blocks of memory for dynamic allocation.

Heaps are useful to know as they are frequently used in queues where certain operations are processed before others. In addition, heaps are used when sorting and ordering, such as finding the k^{th} largest/smallest value in a list.

In this chapter, we will talk about the following:

- What is a priority queue?
- What is a heap?

11.1. What is a priority queue?

Priority queues (PQ) are abstract data types wherein higher priority elements are processed before lower priority ones. In some instances, the design allows for two items with the same priority to be processed in a "first come, first served" manner, while other instances, same-priority elements are effectively randomized in order of their process, regardless of order of enqueue. Priority queues are commonly implemented using heaps, but not always.

The properties for a PQ are as follows:

- Every item has a priority assigned
- An element with a higher priority is dequeued before a lower priority element
- Two elements with the same priority are processed in an order determined by the PQ implementation, but most often in a FIFO manner.

In addition, a normal PQ has the following operations:

- *insert(item, priority)*: insert an item with the given priority
- *get_highest_priority()*: return the element with the highest priority value assigned; essentially this is a *peek()* into the queue
- *pop_highest_priority()*: return and delete the highest priority element
- *is_empty()*: check whether the queue contains any elements

Some PQ implementations are focused on the lowest priority items, so the functions above would be directed towards the low priority side of the queue.

The code example below shows one way to implement a priority queue[24]:

```
class PriorityQueue:
    """Manual implementation of a priority queue. Worse
       performance than built-in heapq data structure."""
    def __init__(self):
        self.queue = []

    def __repr__(self):
        return " ".join([str(i) for i in self.queue]) #
           Iterate through queue list

    def check_empty(self):
        """Checks if queue is empty"""
        if len(self.queue) == 0:
            return True

    def insert(self, element):
        """Add element to queue"""
        self.queue.append(element)

    # for popping an element based on priority
    def pop(self):
        max_value = 0
        for i in range(len(self.queue)): # Iterate over
           the length of the queue
            if self.queue[i] > self.queue[max_value]: # If
               element > max_value, change max_value to
               element
                max_value = i
        item = self.queue[max_value] # Set item to
           max_value
        del self.queue[max_value] # Delete max_value
        return item # Return max_value item

if __name__ == '__main__':
    myQueue = PriorityQueue()
    myQueue.insert(12)
    myQueue.insert(1)
    myQueue.insert(14)
    myQueue.insert(7)
    print(myQueue) # Show all values in queue
    while not myQueue.check_empty(): # Continue while
       queue has values
        print(myQueue.pop()) # Pop out the highest value
```

The output is shown in figure 11.1.1. It should be noted that, with this implementation using OOP, you have to create two methods (*check_empty()* and *insert()*) that would not normally be required if using functions. In other words, because we have to use class instances, those instances cannot utilize direct *append()* or *len()* function calls, as the instance object has no idea what those are. That's why you have to take the additional steps of providing those methods, even though they are very simple. This also demonstrates that, at least

in Python, the ability to choose between OOP and procedural programming is often a trade-off of functionality.

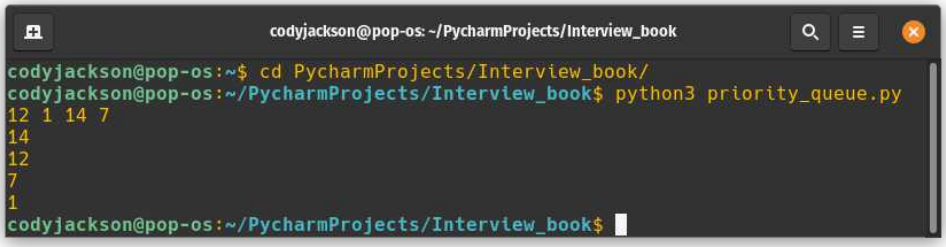

Figure 11.1.1.: Priority queue output

11.1.1. What is a basic priority of queue implementation?

Simple, inefficient PQ implementations are most commonly used to help a programmer understand the concepts of PQ operations. For example, an unsorted list could be searched each time for the element with the highest priority. This way, it doesn't matter what priority elements are appended to the end, as the search will always find the correct value.

Another example is a list that is sorted upon each query, resulting in the highest priority element being placed at the end. Therefore, a simple *pop()* operation can return the highest priority element. Naturally, this requires an extra step to be introduced: sorting the list prior to popping the element.

11.1.2. What is the normal priority queue implementation?

More commonly, to improve performance, a heap is used as the PQ implementation. In Big-O notation, a heap provides $O(\log n)$ performance for insertions and removals, and *O(n)* for the initial building of the queue. Alternatives to a basic heap are data structures such as pairing heaps or Fibonacci heaps that improve the bounds and time complexity for some operations.

Another approach is to use a self-balancing binary search tree, such as when data access is already provided via a library or API. Using a binary tree in this manner also provides $O(\log n)$ performance when inserting and removing data, but results in $O(n \log n)$ time when building from existing element sequences.

11.2. Heap

A heap is a specialized tree data structure. Heaps are implemented as priority queues; the association is close enough that priority queues are often called "heaps", regardless of whether they are implemented as heaps or not.

There are two primary heap types, max-heap and min-heap.

- Max-heap: The root node contains the highest value of all nodes. In other words, the children of the heap all have values less than the root node. In addition, all sub-trees have the same properties, i.e. moving from root node to leaves, every child is less than its parent.

- Min-heap: Basically, this is the opposite of max-heap. The root node is the smallest value of all nodes, and going towards the leaves, each child value is progressively larger than its parent.

Heaps are commonly used as data structures when the highest (or lowest) value priority is frequently removed. Binary heaps are common implementations, as they are simply complete binary trees (often stored as an array) in either max- or min-heap configuration. Common applications include array sorting, priority queues, and graph algorithms.

Other heap configurations include:

- 2-3 heap
- B-heap
- Beap
- Binomial heap
- Brodal queue
- d-ary heap
- Fibonacci heap
- K-D heap
- Leaf heap
- Leftist heap
- Pairing heap
- Radix heap
- Randomized meldable heap
- Skew heap
- Soft heap
- Ternary heap
- Treap
- Weak heap

11.2.1. Basic implementation

Heaps can be implemented via fixed or dynamic arrays, without pointers between elements. With binary heaps, the first or last element will be the root; the next two elements are the root's children; the next four elements are those nodes' children, and so on.

The result of this is that the children of any node n are at positions $2n+1$ and $2n+2$ (assuming a zero-based array). Therefore, any node within the tree can be identified using simple indexing. Heap balancing is accomplished by swapping the out-of-order elements via sift-up and sift-down.

When elements are added to the heap, they are frequently placed in the first available free space. This causes the heap to sift its nodes until properly sorted.

When removing the root node, the last element becomes the root and a sift-up is performed to rebalance the heap. Replacing is performed by removing the root and adding the new element as the root, followed by a sift-down. This eliminates the extra work of a *pop/push*, which would perform a sift-down of the last element followed by a sift-up of the new element.

11.2.1.1. What are some common heap operations?

Heaps have a number of operations that are commonly implemented, such as those shown in table 11.1:

Basic Operations	Creation Operations	Inspection Operations	Internal Operations
find-max	create-heap	size	increase-key
find-min	heapify	is-empty	delete
insert	merge (union)		sift-up
extract-max	meld		sift-down
extract-min			decrease-key
delete-max			
delete-min			
replace			

Table 11.1.: Common heap operations

More information for these items is provided below.

- *find-min/find-max*: finds the minimum item in a min-heap, or the maximum item in a max-heap
- *insert*: pushes a new key to the heap
- *extract-min/extract-max*: pops the node with the maximum value from a max-heap, or the node with the minimum value from a min-heap
- *delete-min/max*: removes the root node from a min-heap or max-heap, respectively
- *replace*: pops the root node and push a new key; more efficient than a pop->push, as there is need to balance the heap only once
- *create-heap*: makes a new, empty heap
- *heapify*: creates a heap out of an array
- *merge (union)*: joins two heaps together to form a new heap, while preserving the original heaps
- *meld*: joins two heaps together but destroys the original heaps
- *size*: returns the number of items in the heap
- *is-empty*: returns True if the heap is empty, otherwise False
- *increase/decrease key*: updates a key within a max-heap or min-heap, respectively
- *delete*: deletes an arbitrary node

- *sift-up/down*: moves a node up or down, respectively, until it reaches the correct location

11.2.2. Python's heapq library

Python provides built-in support for heaps via the *heapq* library. This library implements a min-heap, with each parent node less than or equal to its children. Therefore, using the *pop()* method returns the smallest heap element. The root is also zero-indexed. The beauty of this is that a Python heap can be considered as a normal Python list, with *heap[0]* being the smallest value, and *heap.sort()* being the one which maintains the heap invariant. In addition, an empty heap can be initialized with a simple [], and a populated list can be made into a heap via *heapify()*.

The code below shows *heapq* being used to create a heap, push a value into the heap, and popping the smallest value from the heap[25]:

```
import heapq
li = [5, 7, 9, 1, 3]
heapq.heapify(li) # Convert list to heap
print(f"The created heap is: {li}")
heapq.heappush(li, 4) # Push value 4 into heap
print(f"The modified heap after push is: {li}")
print(f"The popped and smallest element is {heapq.heappop(
    li)}") # Pop
smallest element
```

Figure 11.2.1 is the output of this code.

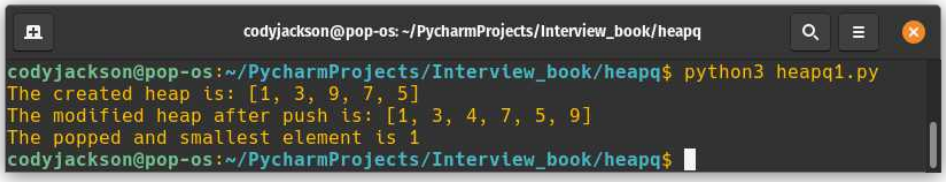

Figure 11.2.1.: Python heapq example 1

The code below shows two ways to push/pop values from a list: *heappushpop()* and *heapreplace()*[25].

```
import heapq
li1 = [5, 7, 9, 4, 3]
li2 = [5, 7, 9, 4, 3]
heapq.heapify(li1)
heapq.heapify(li2)
print(f"The popped item using heappushpop() is: {heapq.
    heappushpop(li1, 2)}")
print(f"The popped item using heapreplace() is: {heapq.
    heapreplace(li2, 2)}")
```

With *heappushpop()*, pushing and popping are combined in one statement, increasing efficiency and maintaining heap order. With *heapreplace()*, pushing and popping are also in one statement, but the element is popped first, then

another element is pushed. This allows a value larger than the pushed element to be returned.

The output is shown in figure 11.2.2. Notice that difference in return value between to the function calls.

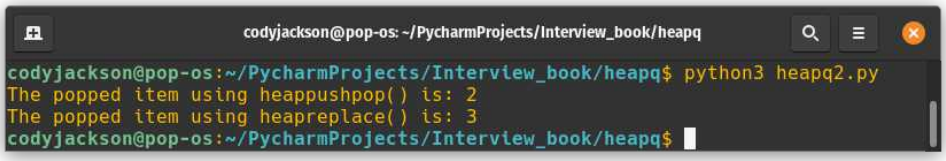

Figure 11.2.2.: Python heapq example 2

Finally, the following code shows how to get the largest and smallest values in the heap[25]:

```
import heapq
li = [6, 7, 9, 4, 3, 5, 8, 10, 1]
heapq.heapify(li)
print(f"The 3 largest numbers in list are: {heapq.nlargest
    (3, li)}")
print(f"The 3 smallest numbers in list are: {heapq.
    nsmallest(3, li)}")
```

The output is shown in figure 11.2.3. Notice that the values are returned in order of their function call, i.e., the largest numbers are returned highest to lowest, while the lowest numbers are returned lowest to highest.

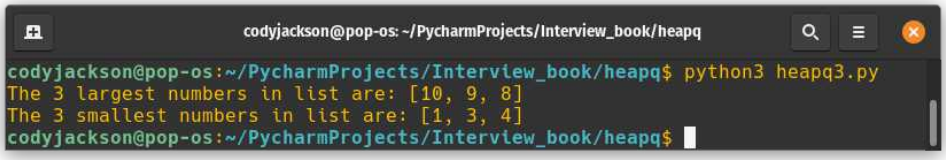

Figure 11.2.3.: Python heapq example 3

11.3. Summary

In this chapter, we looked at priority queues and heaps. We also looked at some of the common operations, as well as Python's implementation of the heapq library. You should recognize that priority queues are similar to regular queues, but have a certain priority associated with them. You should also understand by now that heaps are data structures used to implement priority queues.

In the next chapter, we will take a look at various searching algorithms, such as binary search and two-pointer search.

12. Linear Search

Data searching is one of the most important parts of computer science. Without it, data could not be retrieved, processed, or otherwise used. Search can be performed within data structures, databases, or even within the search space of a defined problem domain.

In this chapter, we will cover the following topics:

- What is linear search?
- What is binary search?
- What is two-pointer search?
- What is the sliding window algorithm?
- What is a pattern matching?

12.1. What is linear search?

Linear search is a methodology for finding an element within a list. By definition, each item is checked, in sequence, until a match is found or the list is exhausted. At worst, a linear search is O(n) time; as the number of items increases, the time to complete increases.

While one benefit of linear search is that data doesn't have to be pre-sorted, because linear search is essentially a brute-force method, there are better options for searching, so linear search is rarely used in practice. Better options, such as hash tables or binary search, are used in all but the smallest arrays.

12.2. What is binary search?

Binary search is an algorithm that requires a sorted array first. Algorithmically, binary search splits the array in half and compares whether the target value is in the upper or lower half (assuming the middle value doesn't match). Therefore, half the array is immediately eliminated from the search, as the target will either be in the upper or lower half. Then, the remaining portion is split again, with the target being checked to the upper or lower half. This continues until the target value is found.

Worst case, binary search takes $O(\log n)$ time, so except for small arrays, binary search is faster than linear search.

12.2.1. How does the Python *bisect* module work?

While not directly related to binary searches, Python's bisect module provides an array bisection algorithm that can be used to create binary searches. The module itself provides support for keeping a list sorted without having to sort it after each insertion, a nice feature when working with comparison operations on long lists.

The module supports the following functions. In these functions, x = desired item, a = target list, lo = bottom portion of list subset, and hi = upper range of subset.

- *bisect_left(a, x, lo=0, hi=len(a))*: locates the insertion point for x in a to maintain sorted order. By default, the entire list is used, though lo and hi can define a subset. If x is already present, the insertion point is moved to the left of the existing entry.
- *bisect_right(a, x, lo=0, hi=len(a))*: similar to bisect_left(), but existing entries result in the insertion point moving to the right.
- *bisect(a, x, lo=0, hi=len(a))*: same as bisect_right().
- *insort_left(a, x, lo=0, hi=len(a))*: insert x in a in sorted order, before any existing x entries.
- *insort_right(a, x, lo=0, hi=len(a))*: insert x in a in sorted order, after any existing x entries.
- *insort(a, x, lo=0, hi=len(a))*: same as insort_right()

12.2.1.1. *bisect* examples

As should be seen in the entries above, these functions are fine for identifying insertion locations to insert data, but there isn't an easily expressed way to actually use them for searching. Therefore, the following examples are provided from the official Python documentation (https://docs.python.org/3.7/library/bisect.html).

```
def index(a, x):
    'Locate the leftmost value exactly equal to x'
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError

def find_lt(a, x):
    'Find rightmost value less than x'
    i = bisect_left(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_le(a, x):
    'Find rightmost value less than or equal to x'
    i = bisect_right(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_gt(a, x):
    'Find leftmost value greater than x'
    i = bisect_right(a, x)
    if i != len(a):
        return a[i]
```

```
    raise ValueError

def find_ge(a, x):
    'Find leftmost item greater than or equal to x'
    i = bisect_left(a, x)
    if i != len(a):
        return a[i]
    raise ValueError
```

12.2.2. What is a rotated, sorted array?

A rotated, sorted array is simply a sorted array that has been shifted one or more positions, i.e. the end value becomes the beginning value. The common problem is "Given a rotated, sorted array, find the position of element x". This appears to be almost strictly an interview question, rather than an actual problem because, in my view, you would simply resort and perform a binary search, as the actual data is normally more important than its position within a data structure.

The solution to this problem is to determine the pivot point for the array to generate two sub-arrays, then search each sub-array for the result's index. With this knowledge, we can formulate a plan of attack.

The rotation amount is either less than or more than half of the elements within the array. If the array is rotated by more than half, then the elements from the start to the midpoint are already sorted. If the array is rotated by less than half, then the elements from midpoint to end are already sorted.

A quick check of the value of *array[0]* vs. *array[mid]* will tell you the rotation amount. If *array[0]* $<$ *array[mid]*, then the array is rotated by more than half of the elements; otherwise, the rotation is less.

With this knowledge, we can quickly perform a binary search of the sorted portion. If the desired element (x) is in the range $array[0] < x < array[mid]$, the search is performed in the first subarray. If not, we search in the second subarray, being aware that it is still a rotated, sorted array so we have to follow the same process as the original array.

In an attempt at clarification, this process is provided in the steps below:

1. Find the midpoint.
2. If the midpoint value is equal to the desired value, return the midpoint index.
3. If not, check whether $array[0] < array[mid]$
    a) If true, we know the left subarray is already sorted
        i. Check if $array[0] < x < array[mid]$
            A. If true, ignore everything from midpoint to end and continue with new subarray of *array[0]* to *array[mid - 1]*.
            B. If false, ignore everything from start to midpoint and continue with new subarray of *array[0]* to *array[mid + 1]*.
    b) If false, we know the right subarray is already sorted
        i. Check if $array[mid] < x < array[end]$
            A. If true, ignore everything from start to midpoint and continue with new subarray of *array[mid + 1]* to *array[end]*.

B. If false, ignore everything from midpoint to end and continue with new subarray of *array[0]* to *array[mid - 1]*.

The main point is, once you have separated the original array into subarrays, you can use a normal binary search once you identify the subarray the element your searching for is located in.

12.3. What is two-pointer search?

Two-pointer search (2PS) is, as its name suggests, searching an array with two different pointers. They can be used for a variety of different purposes, such as removing duplicates, reversing characters in a string, or even identifying elements that equal a given value.

Depending on implementation, 2PS can have two pointers working in the same direction (one faster moving than the other, maybe by an offset) or, possibly more common, starting at either end of the array. A key factor is that the array is pre-sorted, otherwise we cannot guarantee the results.

12.3.1. What are some two-pointer techniques?

Let's assume we have a sorted array and we want to find any pair of elements that, when added together, provides the sum x. If we brute-force this, one way to do that is shown in the code example below[26]:

```
def is_pair_sum(array_sum):
    array = range(20)
    for i in array:
        for j in array:
            print(f"i={i}") # Show the i value
            print(f"j={j}") # Show the j value
            if i + j == array_sum:
                return True
            if i + j > array_sum:
                break
    else:
        return False

if __name__ == "__main__":
    print(is_pair_sum(20))
```

If we run this program, we get figure 12.3.1. I've truncated much of the initial output, but you can see that the top of the screenshot that *i=0* and *j=19*, then it moves to *i=1* and *j=0*, so the loop starts over. As should be evident from the source code, we are simply looping over each value within the array and summing the i and j values, with *i=0* in the first go-around. When that doesn't work, i is incremented to 1 and we start the addition again through every possible j value. In this case, we get to *i=1* and *j=19* before we have a sum that matches $x = (20)$, so the loop finishes.

If we use 2PS and start from either end of the array, summing the elements. If the sum is smaller than x, the left pointer (i) is shifted right; if the sum is greater than x, then pointer j is shifted left. This continues until we get sum x. This code is shown below[26]:

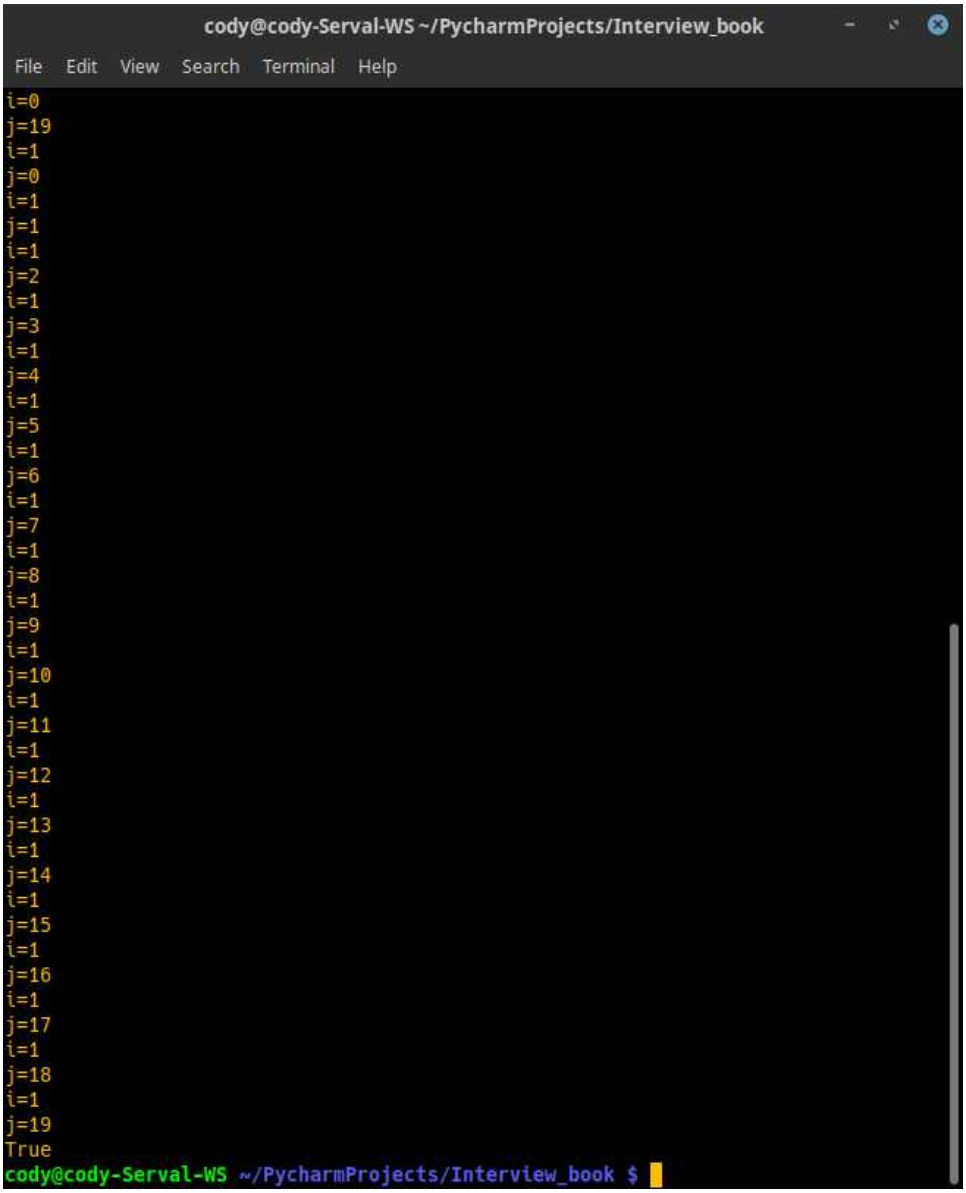

Figure 12.3.1.: Two-pointer search

```
def is_pair_sum(array_sum):
    array = range(20)
    i = 0
    j = array[-1]

    while i < j:
        print(f"i={i}")
        print(f"j={j}")
        if array[i] + array[j] == array_sum:
            return True
        elif array[i] + array[j] < array_sum:
            i += 1
        else:
            j -= 1
    return False
```

```
if __name__ == "__main__":
    print(is_pair_sum(20))
```

When ran, we get figure 12.3.2. It should be very obvious that this is much faster than the brute-force solution shown earlier. In fact, the brute-force solution is time $O(n^2)$ while the 2PS solution is time $O(n)$.

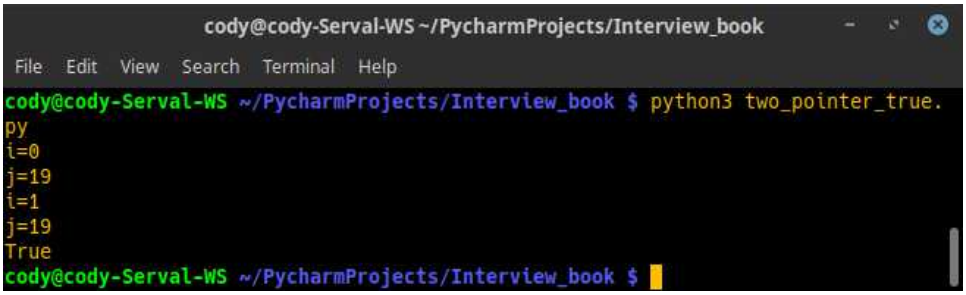

Figure 12.3.2.: Optimized two-pointer search

12.4. What is the sliding window algorithm?

A variation of 2PS is the sliding window. The idea is to think of an array as a window of length n, with a pane of glass that covers only a portion of the array, k. For example, if $n = 5$ and $k = 3$, at any time we can only work with three elements of the array. This concept is actually applied in the sliding window protocol utilized in network transmission via TCP and layer 2 transfers.

The main reason to use a sliding window is to convert a nested *for* loop to a single *for* loop, changing the time from $O(k \times n)$ to $O(n)$. We show a brute-force example of getting the maximum sum of k consecutive elements from array n in the code below[27]:

```
def max_sum(array, window):
    max_window_sum = 0
    for i in range(len(array) - window + 1):
        current_sum = 0
        for j in range(window):
            current_sum = current_sum + array[i + j]
        max_window_sum = max(current_sum, max_window_sum)
            # Update result if required.

    return max_window_sum

if __name__ == "__main__":
    arr = range(30)
    k = 4 # Window size
    print(max_sum(arr, k))
```

When we run this code, we get the result in figure 12.4.1. The result (110) is the sum of the last four values in the array (29+28+27+26).

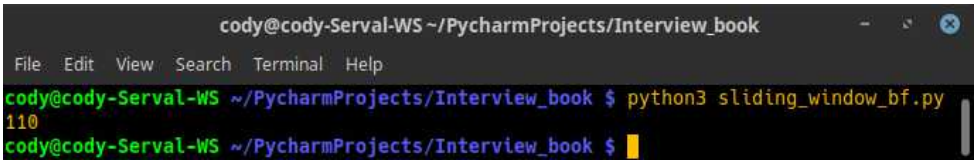

Figure 12.4.1.: Sliding window (brute force)

If we use a sliding window algorithm, we can see how the code is easier to work with, as shown below[28]:

```
def max_sum(array, window):
    max_window_sum = 0

    # Compute sum of first window
    window_sum = sum([array[i] for i in range(window)])

    # Compute sums of remaining windows by removing first
        element of previous window and adding last element
        of current window.
    for i in range(len(array) - window):
        window_sum = window_sum - array[i] + array[i +
            window]
        max_window_sum = max(window_sum, max_window_sum)

    return max_window_sum

if __name__ == "__main__":
    arr = range(30)
    k = 4 # Window size
    print(max_sum(arr, k))
```

If we run this code, we get figure 12.4.2, just like the brute-force method. The benefit, obviously, is that we only have to deal with a single loop rather than nested loops, which can devolve into chaos depending on how much work you have the multiple loops doing.

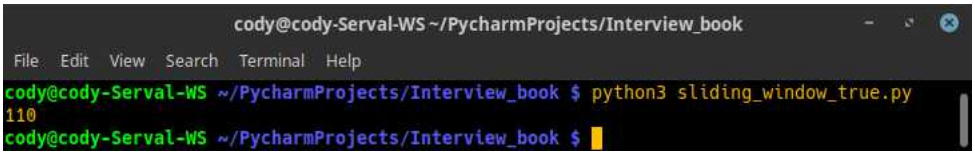

Figure 12.4.2.: Sliding window (algorithm)

12.5. What is pattern matching?

Pattern matching is similar to linear searching, as we are checking a token sequence for a particular pattern of tokens. With text strings, pattern matching is often performed using regular expressions.

Naive pattern matching moves through each token in the sequence, checking for a match. If a match is found, the index of the match is noted and the next token is checked to see if it continues with the matching pattern. If so, the subsequent tokens are checked. If not, the index is cleared until the next match is found.

An example of this is shown in the code below[29]:

```
def search(pattern, txt_string):
    pattern_length = len(pattern)
    string_length = len(txt_string)

    # A loop to slide pattern[] one by one
    for i in range(string_length - pattern_length + 1):
        j = 0

        # For current index i, check for pattern match
        for j in range(0, pattern_length):
            if txt_string[i + j] != pattern[j]:
                break

        if j == pattern_length - 1:
            print("Match found at index ", i)

if __name__ == "__main__":
    txt = "AABAACAADAABAAABAA"
    pat = "AABA"
    search(pat, txt)
```

When ran, we get the figure 12.4.2. Naive pattern matching has a couple of worst-case scenarios: when all the characters in the text and the pattern are the same, or when all the characters are the same except for the last character. In these scenarios, the time is $O(m \times (n - m + 1))$. However, other algorithms provide improved analysis, as discussed below.

12.5.1. Knuth-Morris-Pratt algorithm

Used for string searching, the KMP algorithm searches for a given word within a particular text string. It examines the comparison and, if a mismatch occurs, the word itself provides sufficient information to determine where the next match could begin, precluding the need to re-check previously matched characters.

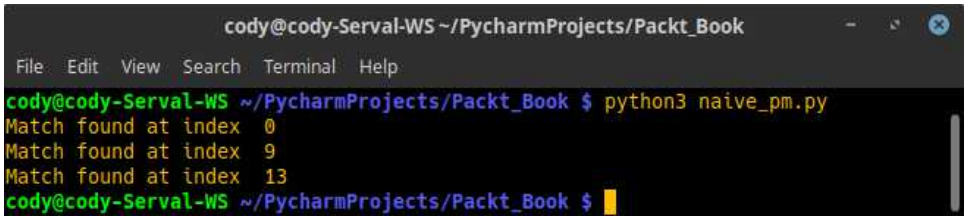

Figure 12.5.1.: Naive pattern matching

KMP provides a worst-case complexity of $O(n)$, but doesn't work well in similar cases to the naive algorithm above. However, because it doesn't match characters that we already know will match, it immediately ignores them and moves on to unmatched characters.

The comparisons are performed with windows (blocks of characters) to quickly identify matching characters. That way, only the characters within each window are compared; any previous, matching windows are ignored. If the pattern is pre-processed, an integer of the longest character count to skip is generated.

The term *lps* denotes the longest proper prefix which is also a suffix, and cannot be the entire string. For example, in the string "ABC", only "A" and "AB" are proper prefixes, while suffixes are "C", "BC", and "ABC".

As the string search is conducted and the pattern is matched, *lps* increments. As soon as a mismatch occurs, we already know how many index positions to skip due to the *lps* value.

An example of the KMP algorithm is provided below[30]:

```
def kmp_search(txt_pattern, txt_string):
    pattern_length = len(txt_pattern)
    string_length = len(txt_string)

    # create lps[] that will hold the longest prefix
        suffix values for pattern
    lps = [0] * pattern_length
    j = 0 # index for pattern[]

    # Preprocess the pattern (calculate lps[] array)
    compute_lps_array(txt_pattern, pattern_length, lps)

    i = 0 # index for txt[]
    while i < string_length:
        if txt_pattern[j] == txt_string[i]: # Match index
            values
            i += 1
            j += 1

        if j == pattern_length:
            print(f"Found pattern at index {str(i-j)}")
            j = lps[j-1]

        elif i < string_length and txt_pattern[j] !=
            txt_string[i]: # mismatch after j matches
            if j != 0: # Ignore lps[0..lps[j-1]]
                characters, they will match anyway
```

```
                j = lps[j-1]
            else:
                i += 1

def compute_lps_array(txt_pattern, pat_length, lps):
    prev_lps_length = 0 # length of the previous longest
        prefix suffix

    lps[0] # lps[0] is always 0
    i = 1

    while i < pat_length: # Calculate lps[i] for i = 1 to
        pattern_length - 1
        if txt_pattern[i] == txt_pattern[prev_lps_length]:
            prev_lps_length += 1
            lps[i] = prev_lps_length
            i += 1
        else:
            if prev_lps_length != 0:
                prev_lps_length = lps[prev_lps_length - 1]
            else:
                lps[i] = 0
                i += 1

if __name__ == "__main__":
    txt = "ABABDABACDABABCABAB"
    pat = "ABABCABAB"
    kmp_search(pat, txt)
```

The output is very simple, as shown in figure 12.5.2:

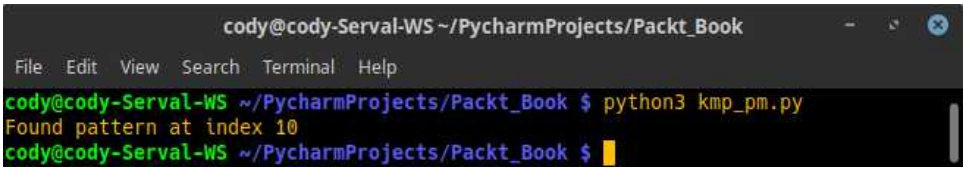

Figure 12.5.2.: KMP algorithm

12.5.2. Rabin-Karp algorithm

Like the naive algorithm, RK considers pattern matching by individual characters rather than windows. However, RK matches the hash digest of the pattern with the hash digest of the current text substring. If the hash values match, then it will consider the individual characters.

Therefore, the RK algorithm needs to calculate the hash digests of not only the pattern itself, but all the substrings within the text string. While we say that individual characters are checked, obviously we have to use a window of several characters to effectively function.

For example, if the string is "BADDAM", the windows might be "BAD", "ADD", "DDA", and "DAM". A hash is calculated for each window segment and compared against the pattern's hash. If the two hashes match, then the

algorithm will actually compare each character within the window against their corresponding index character in the pattern.

The reason for this is because the domain space for the hash is small compared to the possible number of character combinations, resulting in a higher number of collisions. Therefore, even if the hash digests match, individual character comparison must be performed to verify the match.

Worst case performance of the RK algorithm is $O(n \times m)$. This occurs when all characters of a pattern and string are the same hash values as all the substrings within the string, e.g. pattern = "XXX" and the string = "XXXXXX".

An example of this algorithm is below[30]:

```
chars = 256 # Number of characters in the input alphabet

def search(txt_pattern, txt_string, prime):
    patt_length = len(txt_pattern)
    str_length = len(txt_string)
    i = 0
    j = 0
    patt_hash = 0 # hash value for pattern
    str_hash = 0 # hash value for txt
    h = 1 # h = pow(d, M-1) % q

    for i in range(patt_length - 1):
        h = (h * chars) % prime

    # Calculate the hash values of pattern and first
       window of text
    for i in range(patt_length):
        patt_hash = (chars * patt_hash + ord(txt_pattern[i
            ])) % prime
        str_hash = (chars * str_hash + ord(txt_string[i]))
            % prime

    for i in range(str_length-patt_length+1): # Slide the
       pattern over text one by one
        if patt_hash == str_hash: # Compare hash values
            for j in range(patt_length): # Check for
                characters one by one
                if txt_string[i + j] != txt_pattern[j]:
                    break
            j += 1
            if j == patt_length: # if p == t and pat[0...M
                -1] = txt[i, i+1, ...i+M-1]
                print("Pattern found at index " + str(i))
        if i < str_length - patt_length: # Calculate hash
            value for next window of text
            str_hash = (chars * (str_hash - ord(txt_string
                [i]) * h) + ord(txt_string[i + patt_length
                ])) % prime
            if str_hash < 0: # If text_hash < 0, convert
                to positive
                str_hash = str_hash + prime
```

```
if __name__ == "__main__":
    txt = "SEEKING FOR SEEKS"
    pat = "SEEK"
    q = 101 # A prime number
    search(pat, txt, q)
```

The output of this program is shown in figure 12.5.3:

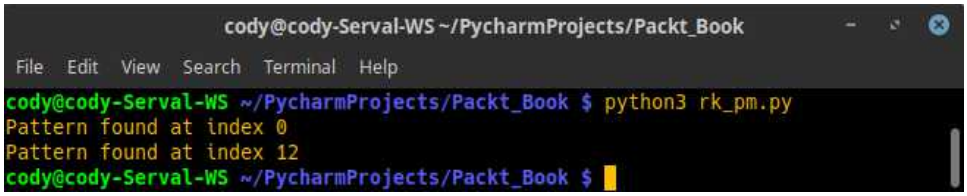

Figure 12.5.3.: RK algorithm

12.6. Summary

In this chapter, we looked at linear searching of sequences, including binary search, Python's bisect function, two-pointer searching, as well as the similar concept of pattern matching in text strings.

Next chapter, we will investigate advanced graph algorithms, including shortest-path designs.

13. Advanced Graph Algorithms

While we've talked about graphs at a basic level, because there are many applications of graphs in computing, a number of algorithms have been designed to work with graphs. In this chapter, we will review some of the more commonly used graph algorithms.

These algorithms will help when working with graphs that don't have images associated with them by allowing you to determine if they are connected or unconnected, directed or undirected, and weakly or strongly connected. These, and various other aspects, provide information about the graph that can be used for further analysis and refinement. Without knowing what you're working with at the start, you may find yourself with invalid data based on incorrect assumptions.

Specifically, we will cover the following topics:

- How Can You Find All the Connected Components in a Graph?
- What is Cycle Detection?
- What is Topological Sort?
- What are Minimum Spanning Trees?
- What are Single-Source Shortest Paths?
- What are All-Pairs Shortest Paths?

13.1. How Can You Find All the Connected Components in a Graph?

Connectivity is one of the basic concepts in graph theory. The ability to find the minimum number of elements to be removed to separate the remaining nodes into isolated sub-graphs reflects how robust the graph is, which is closely tied to network theory.

An undirected graph is connected when at least one vertex exists and there is a path between all vertex pairs. In other words, a connected graph exists if all nodes are reachable.

A directed graph is weakly connected if, by replacing all directed edges with undirected edges, an undirected, connected graph is created. It is strongly connected if there is a direct path between vertices a and b, as well as a directed path between b and a, for every pair of vertices a, b. It is unilaterally connected if there is a direct path, in either direction, between vertices a and b for every pair of vertices a, b.

13.1.1. How Can You Identify Connected Components in an Undirected Graph?

Given figure 13.1.1, how would you identify the connections between nodes?

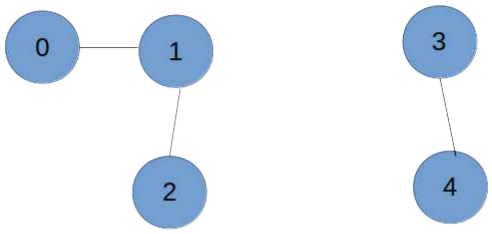

Figure 13.1.1.: Unconnected, undirected graph

The algorithm used depends on whether you want to use BFS or DFS, starting from every unvisited vertex.

In the algorithm listed below, we use DFS:

```
Initialize all vertices as "not visited"

At each vertex "v", do the following:
        If vertex is not visited yet, call dfs_util(v)
        If visited, pass

dfs_util(v)
        Mark vertex as "visited"
        Print vertex name
        For every adjacent vertex "u":
                If not visited, call dfs_util(u)
```

Using this idea, we can create one possible solution in the code below, where we create the *Graph* class and the initialization method[31]. Then we create a utility method that performs the DFS operation for us.

```
class Graph:
    def __init__(self, vertex):
        self.vertex = vertex
        self.adjacent = [[] for node in range(vertex)]

    def dfs_util(self, temp, v, visited):
        """Mark vertex as visited, repeat for all
           connected vertices"""
        visited[v] = True # Mark the current vertex as
           visited
        temp.append(v) # Store the vertex to list
        for i in self.adjacent[v]: # Repeat for all
           vertices adjacent to this vertex v
            if not visited[i]:
                temp = self.dfs_util(temp, i, visited) #
                   Update the list
        return temp
```

Next, we make a method to add the edges between nodes, and a method to get the nodes that have been connected.

```
    def add_edge(self, v, w):
        """Add edge between connected nodes"""
        self.adjacent[v].append(w)
        self.adjacent[w].append(v)

    def conn_nodes(self):
        """Get the connected nodes in undirected graph"""
        visited = []
        connections = []
        for i in range(self.vertex):
            visited.append(False)
        for v in range(self.vertex):
            if not visited[v]:
                temp = []
                connections.append(self.dfs_util(temp, v,
                    visited))
        return connections
```

Finally, we create the driver code to create a working graph.

```
if __name__ == "__main__":
    g = Graph(5)
    g.add_edge(0, 1)
    g.add_edge(1, 2)
    g.add_edge(3, 4)
    conns = g.conn_nodes()
    print("Below are the connected nodes:")
    for conn in conns:
        print(conn)
```

When run, we get figure 13.1.2. The output shows which nodes are connected. In this case, it is easily verified by looking at the actual images that nodes 0-2 are connected together, as are nodes 3 and 4, yet the two groups are separate from each other.

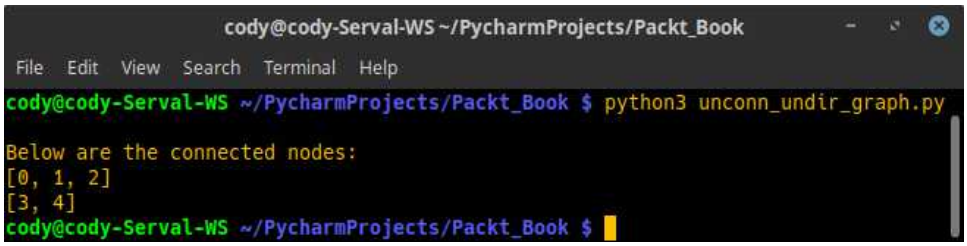

Figure 13.1.2.: Unconnected undirected graph output

13.1.2. How Can You Tell Whether a Directed Graph is Strongly Connected?

Like nearly everything else, there are multiple ways to check directed graph connections. We will demonstrate one use of Tarjan's algorithm in this section.

Tarjan's algorithm can be summed up as: a DFS starts at an arbitrary node. Since DFS visits every node only once, the collection of search trees is a spanning forest of the graph. The strongly connected components will be found as subtrees of the forest.

Assume we have the graph in figure 13.1.3:

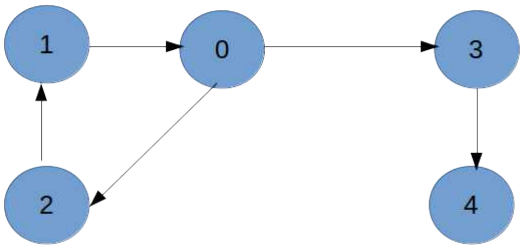

Figure 13.1.3.: Graph for Tarjan's algorithm

One possible solution to identify a graph's connection type is in the code examples below[31]. First, we import *defaultdict* from *collections* to provide default values when we create a dictionary. Then we create the *Graph* class and provide the initialization and edge addition methods.

```
from collections import defaultdict

class Graph:
    """A directed graph using adjacent list representation
       """
    def __init__(self, vertex):
        self.vert = vertex
        self.graph = defaultdict(list)
        self.time = 0

    def add_edge(self, next_vert, vert):
        self.graph[next_vert].append(vert)
```

Next is a utility method to find strongly connected components. The parameters *scc_util()* takes are

- next_vert: The vertex to be visited next
- low: earliest visited vertex that can be reached from subtree rooted with current vertex
- disc_time: discovery times of visited vertices
- stack_member: Boolean check whether a node is in the stack
- store: stores all the connected ancestors (parents, grandparents, etc. of the current node)

```
    def scc_util(self, next_vert, low, disc_time,
        stack_member, store):
```

```
        """Identify strongly connected components using
            DFS traversal"""
        disc_time[next_vert] = self.time
        low[next_vert] = self.time
        self.time += 1
        stack_member[next_vert] = True
        store.append(next_vert)

        for v in self.graph[next_vert]: # Go through all
            vertices adjacent to this
            if disc_time[v] == -1: # If not visited,
                recursive call
                self.scc_util(v, low, disc_time,
                    stack_member, store)
                low[next_vert] = min(low[next_vert], low[v
                    ]) # Check if the subtree has a
                    connection ancestor of 'u'
            elif stack_member[v]: # Update low value of 'u
                ' only if 'v' is still in stack
                low[next_vert] = min(low[next_vert],
                    disc_time[v])

        # Head node found, pop the stack and print an SCC
        stack_vert = -1 # Store stack extracted vertices
        if low[next_vert] == disc_time[next_vert]:
            while stack_vert != next_vert:
                stack_vert = store.pop()
                print(stack_vert, end=" ")
                stack_member[stack_vert] = False
            print() # Separate connections
```

In the final code block, we create the method that will act as a utility function to complete the DFS traversal.

```
    def dfs_traverse(self):
        """Perform DFS traversal"""
        disc_time = [-1] * self.vert
        low = [-1] * self.vert
        stack_member = [False] * self.vert
        store = []

        # Find articulation points in DFS tree with root
            of "i"
        for i in range(self.vert):
            if disc_time[i] == -1:
                self.scc_util(i, low, disc_time,
                    stack_member, store)

if __name__ == "__main__":
    g1 = Graph(5)
    g1.add_edge(1, 0)
    g1.add_edge(0, 2)
    g1.add_edge(2, 1)
```

```
g1.add_edge(0, 3)
g1.add_edge(3, 4)
print(g1.dfs_traverse())
```

When run, we get figure 13.1.4. Here, the output shows which nodes are separated with single edges (3 & 4) and which nodes are connected via circular edges (0-2).

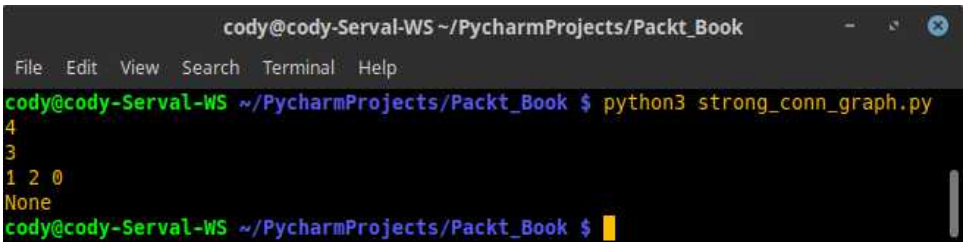

Figure 13.1.4.: Tarjan's algorithm output

13.2. What is Cycle Detection?

Cycle detection (also called cycle finding) is the method of algorithmically finding a cycle in a sequence of iterated function values.

For any function f that maps a finite set S to itself, and any initial value in S, the following sequence

$$x_0, x_1 = f(x_0), x_2 = f(x_1), \ldots, x_i = f(x_{i-1}), \ldots \; x_0, x_1 = f(x_0), x_2 = f(x_1), \ldots, x_i = f(x_{i-1}), \ldots$$

will eventually use the same value twice; in other words, there must be some pair of distinct indices i and j such that $x_i = x_j$. As soon as this occurs, the sequence must continue periodically, repeating the same sequence of values from x_i to $x_j - 1$. That is the essence of cycle detection: finding i and j, given f and x_0.

A number of algorithms exist for finding cycles quickly while using little memory. Floyd's Tortoise and Hare algorithm moves two pointers at different speeds through the sequence of values until they both point to equal values. An alternative take is Brent's algorithm, which is based on exponential search.

Both Floyd's and Brent's algorithms utilize a constant amount of memory. The number of function evaluations is proportional to the distance from the start of the sequence to the first repetition. Other algorithms trade larger amounts of memory for fewer function evaluations.

Cycle detection is utilized in

- Testing the quality of pseudorandom number generators and cryptographic hash functions,
- Computational number theory algorithms,
- Detection of infinite loops in computer programs and periodic configurations in cellular automata, and
- The automated shape analysis of linked list data structures.

When talking about cycle detection, two symbols are commonly used: λ is the loop length while μ is the smallest index value of the recurring sequence.

13.2.1. Explanatory Example

Assume the set $S = 0, 1, 2, 3, 4, 5, 6, 7, 8, x0 = 2$, and the mapping of x to *f(x)* shown in table 13.1:

x	*f(x)*
0	6
1	6
2	0
3	1
4	4
5	3
6	3
7	4
8	0

Table 13.1.: Map x to *f(x)*

If we repeatedly apply f, such that S is applied to itself, we see the following:

$$x0 = 2, f(2) = 0, f(0) = 6, f(6) = 3,$$
$$f(3) = 1, f(1) = 6, f(6) = 3, f(3) = 1, f(1) = 6, ...$$

Therefore, the cycle of this sequence is 6, 3, 1.

13.2.2. Floyd's Tortoise and Hare Algorithm

This algorithm has two pointers in the sequence, one (the tortoise) at x_i , and the other (the hare) at x_{2i}. As the algorithm operates, it increases i by one, moving the tortoise one step forward and the hare two steps forward in the sequence, and then compare the values at these two pointers. The smallest value of $i > 0$ for which the tortoise and hare point to equal values is the desired value v.

This process can be demonstrated in the following code examples, using the same data as in the example above (the special symbols λ and μ have been replaced with explanatory names for clarification)[32]:

```
def tort_hare(f, x0):
    tortoise = f(x0) # f(x0) is the element/node next to
       x0.
    hare = f(f(x0))
    while tortoise != hare:
        tortoise = f(tortoise)
        hare = f(f(hare))

    # Find the position of first repetition.
    first_rep = 0
    tortoise = x0
    while tortoise != hare:
        tortoise = f(tortoise)
```

```
        hare = f(hare) # Hare and tortoise move at same
            speed
        first_rep += 1

    # Find the length of the shortest cycle
    seq_length = 1
    hare = f(tortoise)
    while tortoise != hare:
        hare = f(hare)
        seq_length += 1

    return seq_length, first_rep
```

First, we define the main tortoise and hare function. The tortoise and hare markers are set to different positions within the sequence and will continue to perform their work until they have matching values.

```
def func(x):
    if x == 0 or x == 1:
        val = 6
    elif x == 2 or x == 8:
        val = 0
    elif x == 3:
        val = 1
    elif x == 4 or x == 7:
        val = 4
    elif x == 5 or x == 6:
        val = 3

    return val

if __name__ == "__main__":
    length, rep = tort_hare(func, 2)
    print(f"Position of first repetition: {rep}, Sequence
        length: {length}")
```

Next, we define a function that generates the appropriate value based on the input; this replicates the operations of the table in section 13.2.1 from above, just prior to this section.

Finally, we have the execution code to run the program. The output of this program is in figure 13.2.1:

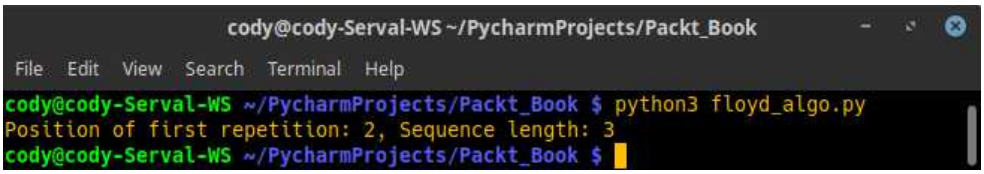

Figure 13.2.1.: Floyd algorithm output

It's not the most elegant program, and it doesn't actually show the sequence, but for most programs, all you need to know is the index position of the start of the cycle and how long the cycle sequence is.

13.2.3. Brent's Algorithm

Like Floyd's algorithm, Brent's algorithm utilizes two pointers in the sequence. However, it searches for the smallest power of two that is larger than λ and μ. The algorithm compares$x_{2^i} - 1$ with subsequent sequence values up to the next power of two, stopping once a match is found.

Compared to Floyd's algorithm, it finds the correct loop length of the cycle directly, rather than needing to search for it in another stage, and its steps involve only one evaluation of the function f rather than three.

An example of this algorithm is displayed below[32]:

```
def brent(f, x0):
    # Search successive powers of two
    power = lam_length = 1
    tortoise = x0
    hare = f(x0) # f(x0) is the element/node next to x0.
    while tortoise != hare:
        if power == lam_length: # Search for a new power
            of two
            tortoise = hare
            power *= 2
            lam_length = 0
        hare = f(hare)
        lam_length += 1

    # Find the position of the first repetition of
       lam_length
    tortoise = hare = x0
    for i in range(lam): # [0, 1, ... , lam-1]
        hare = f(hare) # Distance between the hare and
            tortoise is now lambda.
    mu = 0
    while tortoise != hare: # Hare and tortoise move at
       same speed until they agree
        tortoise = f(tortoise)
        hare = f(hare)
        mu += 1

    return lam_length, mu
```

We set up Brent's algorithm function in much the same way we did for Floyd's algorithm, except we are looking for powers of two.

```
def func(x):
    if x == 0 or x == 1:
        val = 6
    elif x == 2 or x == 8:
        val = 0
    elif x == 3:
        val = 1
    elif x == 4 or x == 7:
        val = 4
    elif x == 5 or x == 6:
        val = 3
```

```
    return val

if __name__ == "__main__":
    length, rep = brent(func, 2)
    print(f"Position of first repetition: {rep}, Sequence
        length: {length}")
```

Again, we set up a function to generate the table mapping and execution code. When run, we get figure 13.2.2:

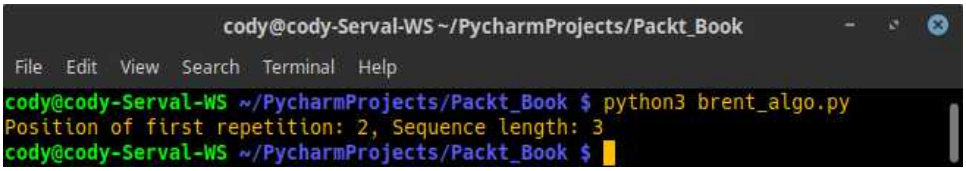

Figure 13.2.2.: Brent's algorithm output

You'll notice that the output is the same as Floyd's algorithm, as it should be. They are accomplishing the same goal, just in different ways.

13.3. What is Topological Sort?

Topological sorting is the linear ordering of a directed graph's vertices, so that every directed edge from vertex u to vertex v, u comes before v in the ordering. A practical example of this would be a graphical project manager, where each vertex represents a particular task and the edges constrain the time of task completion, such that subsequent tasks are completed sequentially.

Topological sorting is only possible when the graph has no directed cycles, i.e. it is a directed acyclic graph (DAG). The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).

Assume we have the DAG in figure 13.3.1.

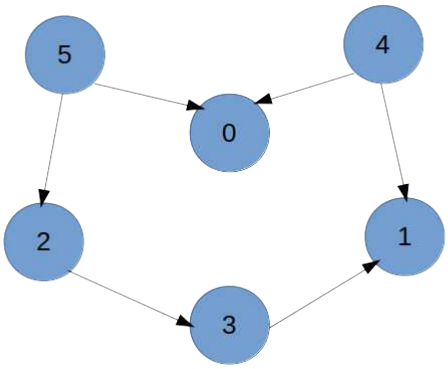

Figure 13.3.1.: Non-cyclic directed graph

The graph shows a directed (arrows) graph that is non-cyclic (nodes 4 & 5 are definite starting points, while nodes 0 & 1 are definite endpoints).

13.3.1. Depth-First Search

When we use depth-first searching (DFS) for topological sorting, we identify a vertex then recursively call DFS for adjacent nodes, modifying the normal DFS to ensure the topological sorting of the graph. In other words, once a node is identified, a depth-first search is performed to identify any adjacent nodes. While this happens, topological sorting is performed of the graph to ensure the proper output.

This process utilizes a temporary stack; the stack stores the vertices as they are identified. In other words, a vertex is pushed to the stack only when all its adjacent vertices, and their vertices and so on, are already in the stack.

One possible DFS solution is provided below[31]:

```
from collections import defaultdict

class Graph:
    def __init__(self, vertices):
        self.graph = defaultdict(list)  # Dictionary of
            adjacency List
        self.verts = vertices  # Number of vertices

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def recursive_topo_sort(self, v, visited, stack):
        """Find all vertices adjacent to this vertex"""
        visited[v] = True  # Mark the current node as
            visited.
        for i in self.graph[v]:
            if not visited[i]:
                self.recursive_topo_sort(i, visited, stack
                    )
        stack.insert(0, v)  # Push current vertex to stack

    def topo_sort(self):
        """Perform topological sort. Uses the recursive
            method to find adjacent vertices"""
        visited = [False] * self.verts # Mark all the
            vertices as not visited
        stack = []
        for i in range(self.verts):  # Call recursive
            method
            if not visited[i]:
                self.recursive_topo_sort(i, visited, stack
                    )
        print(stack)

if __name__ == "__main__":
    g = Graph(6)
    g.add_edge(5, 2)
    g.add_edge(5, 0)
    g.add_edge(4, 0)
```

```
g.add_edge(4, 1)
g.add_edge(2, 3)
g.add_edge(3, 1)
print("Topological Sort of the given graph")
print(g.topo_sort())
```

The output of this program is figure 13.3.2. The output shows the sorted values of the graph, based on the edges that were input into the algorithm. In short, the program determined where the nodes were located in conjunction to each other, based on the edge input. It then performed a topological sorting, from the highest value node to the lowest.

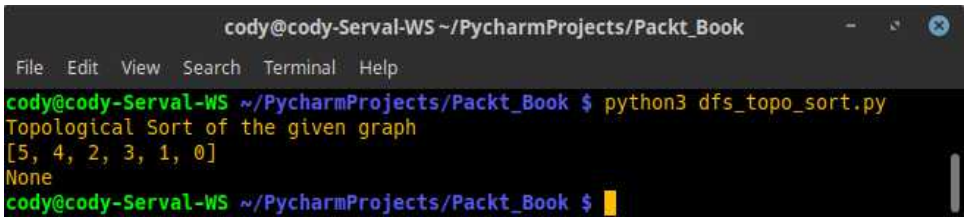

Figure 13.3.2.: DFS output

13.3.2. Kahn's Algorithm

An alternative to DFS for topological sorting is Kahn's algorithm:

1. Compute the in-degree (number of incoming edges) for each vertex. Initialize the visited node count to zero.
2. Pick all vertices with in-degree = 0 and add to the queue.
3. Remove a vertex from the queue.
   a) Increment the visited node count by one.
   b) Decrease in-degree by 1 for all neighboring nodes.
   c) If in-degree of neighboring nodes = 0, add it to queue.
4. Repeat step 3 until queue is empty.
5. If count of visited nodes isn't equal to the number of nodes in the graph, then topological sorting is not possible.

To find the in-degree of each node, two options are available. Assuming an array of in-degree counts exists:

1. Traverse the array of edges and increment the in-degree counter of the destination node by one.
2. Traverse the node list and increment the in-degree count of all nodes connected to it by one.

One possible code example for this problem is provided below[31]:

```
from collections import defaultdict

class Graph:
    def __init__(self, vertices):
        self.graph = defaultdict(list)  # Dictionary of
            adjacency List
        self.verts = vertices  # Number of vertices

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def topo_sort(self):
        in_degree = [0] * self.verts  # Store in-degrees
            of all vertices
        visited_verts = 0
        top_order = []  # Topological sorted vertices
        zero_queue = []  # Vertices with 0 in-degrees

        for i in self.graph:  # Traverse adjacent lists to
            fill in-degrees of vertices.
            for j in self.graph[i]:
                in_degree[j] += 1

        for i in range(self.verts):  # Enqueue in-degree =
            0 vertices
            if in_degree[i] == 0:
                zero_queue.append(i)
```

We are utilizing *defaultdict* again to generate default values for dictionary entries. The *Graph* class has the usual initialization and *add_edge()* methods, then we start the topological sorting method that performs most of the work in this program. After we set our working variables, we first check all the in-degree values of vertices and enqueue all vertices with an in-degree value of zero.

```
        while zero_queue:  # Dequeue vertices from zero_queue
            and enqueue adjacent if in-degree = 0
            u = zero_queue.pop(0)
            top_order.append(u)
            for i in self.graph[u]:  # Decrease adjacent nodes
                in-degree by 1
                in_degree[i] -= 1
                if in_degree[i] == 0: # If in-degree becomes
                    zero, add it to zero_queue
                    zero_queue.append(i)
            visited_verts += 1

            try:
                if visited_verts != self.verts:  # Check if
                    there was a cycle
                    raise ValueError
                else:
```

```
            print(top_order)
        except ValueError:
            print("There exists a cycle in the graph")

if __name__ == "__main__":
    g = Graph(6)
    g.add_edge(5, 2)
    g.add_edge(5, 0)
    g.add_edge(4, 0)
    g.add_edge(4, 1)
    g.add_edge(2, 3)
    g.add_edge(3, 1)
    # g.add_edge(0, 5)
    print("Topological Sort of the given graph")
    print(g.topo_sort())
```

Next, as long as there are items in the *zero_queue*, we dequeue those vertices and enqueue any adjacent vertices with an in-degree value of zero. Next, we compare the count of visited vertices to the total number of vertices; if they don't match, then we have a cycle. Otherwise, we print the topologically sorted list of vertices. Finally, we create the execution code to create the graph and run the program.

When run, we receive figure 13.3.3. The output has been sorted topologically. But this isn't the only option for output. The output has been sorted topologically. But this isn't the only option for output.

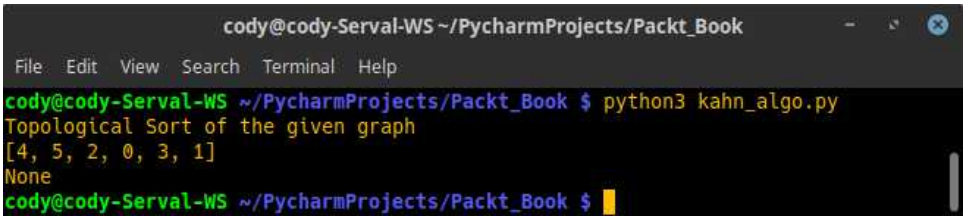

Figure 13.3.3.: Kahn algorithm output

Notice that, in the code example, we have added a test for the existence of a cycle in the graph. If we uncomment the final edge addition in the main code, we will see that the cyclical nature of the new graph is identified (figure 13.3.4). In this case, rather than returning a sorted list, we receive notification that there is a cycle within the graph.

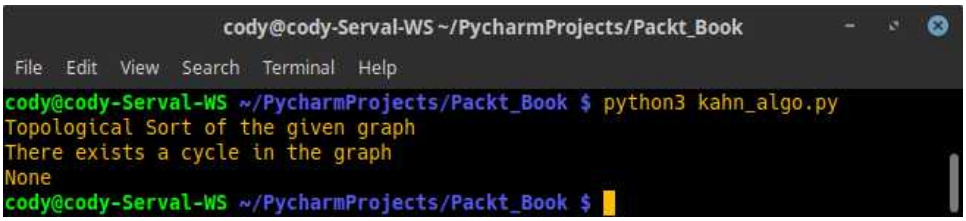

Figure 13.3.4.: Cyclical Kahn algorithm

13.4. What are Minimum Spanning Trees?

Minimum spanning trees (MST) are a subset of edges within a connected, edge-weighted, undirected graph. In an MST, all vertices are connected together with no cycles and with the minimum possible total edge weight. In other words, an MST is a spanning tree edge whose sum weight is the smallest possible. The union of the MSTs for all connected components within any edge-weighted undirected graph is the minimum spanning forest.

Possible uses include minimizing costs associated with construction. For example, reducing the cost of laying fiber within a neighborhood, utilizing existing roads or other infrastructure. The longer or deeper paths would be represented as edges with higher weights, as they would be more expensive. The MST for this problem would be the lowest cost of laying fiber while still ensuring all houses were connected.

There may be more than one MST for a given graph, as shown in figure 13.4.1. The first image is the graph itself, while the other two images show two possibilities for an MST. Note that the two possible images aren't the only possibilities, as there are also potential paths between A-E, B-D, and E-F.

13.4.1. Kruskal's Algorithm

A commonly used algorithm for finding an MST is Kruskal's algorithm, which is explained in the following steps.

1. Sort all edges in non-decreasing order, by weight.
2. Pick the smallest edge.
   a) Check whether the selected edge forms a cycle with the spanning-tree formed so far.
   b) If no cycle is formed, then include this edge. Otherwise, remove it.
3. Repeat step 2 until there are $(V-1)$ edges in the spanning tree. $(V-1)$ is the MST for the graph, where V is the number vertices in the graph.

The following code examples demonstrate this algorithm[31]. First, we go through the now-standard steps of creating a Graph class. After initialization and edge adding, we create a method to find the set of an element, then a method to perform a union between the two different sets.

```
class Graph:
    def __init__(self, vertices):
        self.verts = vertices # No. of vertices
        self.graph = []

    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])

    def find(self, parent, i):
        """Find the set of an element"""
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])
```

```
    def union(self, parent, rank, x, y):
        """Union, by rank, of two sets"""
        x_root = self.find(parent, x)
        y_root = self.find(parent, y)

        # Union by rank
        if rank[x_root] < rank[y_root]:
            parent[x_root] = y_root
        elif rank[x_root] > rank[y_root]:
            parent[y_root] = x_root
        else: # If ranks are same, then pick one as root
            and increment its rank by one
            parent[y_root] = x_root
            rank[x_root] += 1
```

The workhorse method of our program is defined next. We sort the edges in increasing order, based on weight, then create a number of single-element subsets that is equal to the number of vertices in the graph. After that, we compare the MST counter to the number of vertices and, as long as the counter is less than the number of vertices, we get the edge components from the graph, find the smallest edge, and check whether including the edge creates a cycle. If no cycle is created, we add the edge to our results list, otherwise it is discarded.

Finally, we print the edges of the MST and execute the program.

```
def kruskal_algo(self):
    """Find the MST. V is the number of vertices in graph.
       """
    result = [] # Store MST
    edge_counter = 0 # Sorted edges
    mst_counter = 0 # Used for result[]
    parent = []
    rank = []

    # Sort all the edges in non-decreasing order of their
       weight.
    self.graph = sorted(self.graph, key=lambda item: item
       [2])

    for node in range(self.verts): # Create V subsets with
        single elements
        parent.append(node)
        rank.append(0)

    while mst_counter < self.verts - 1: # Number of edges
       to be taken is equal to V-1
        u, v, w = self.graph[edge_counter] # Extract edge
            components from graph
        edge_counter = edge_counter + 1

        # Find smallest edge
        x = self.find(parent, u)
        y = self.find(parent, v)
```

```
            if x != y: # If edge doesn't create a cycle, add
                it to the result
                mst_counter = mst_counter + 1
                result.append([u, v, w])
                self.union(parent, rank, x, y)

        print("The following are the edges in the MST")
        for u, v, weight in result:
            print(f"{u} -- {v} == {weight}")

if __name__ == "__main__":
    g = Graph(4)
    g.add_edge(0, 1, 10)
    g.add_edge(0, 2, 6)
    g.add_edge(0, 3, 5)
    g.add_edge(1, 3, 15)
    g.add_edge(2, 3, 4)
    g.kruskal_algo()
```

The results are displayed in figure 13.4.2. In this output, we see the edges 2-3, 0-3, and 0-1, with their associated weights.

13.4.2. Prim's Algorithm

Another MST algorithm that is commonly used is Prim's algorithm. The short version of the algorithm is:

1. Initialize a tree with a single vertex, arbitrarily selected from the graph.
2. Grow the tree with one edge. Of the edges that connect to other vertices not yet in the tree, find the edge of minimum weight and add it to the tree.
3. Repeat step 2 for all vertices in the tree.

One way of writing this algorithm is shown in the following examples[33]. First, we create the Graph class, with a method to print out the MST and a method to find the vertex with the minimum distance.

```
import sys

class Graph:
    def __init__(self, vertices):
        self.verts = vertices
        self.graph = [[0 for column in range(vertices)]
            for row in range(vertices)] # Create a matrix
            of vertices

    def print_mst(self, parent):
        """Print MST construct in parent array"""
        print("Edge Weight")
        for i in range(1, self.verts):
            print(f"{parent[i]}-{i} {self.graph[i][parent[
                i]]}")
```

```
    def min_key(self, key, mst_set):
        """Find vertex w/ minimum distance value. Uses the
            set of vertices not yet included in shortest
           path tree."""
        min_dist = sys.maxsize # Maximum int size
        min_index = 0

        for v in range(self.verts):
            if key[v] < min_dist and mst_set[v] is False:
                min_dist = key[v]
                min_index = v

        return min_index
```

Next, we create the main method for the program to find the MST for the given graph. In this method, we determine the shortest path in a subtree, then perform the same operation for adjacent vertices. Then we execute the code.

```
    def mst_func(self):
        """Create MST for graph"""
        key = [sys.maxsize] * self.verts # Key values used
            to pick minimum weight edge in cut
        parent = [None] * self.verts # Store constructed
           MST
        key[0] = 0 # Create first vertex
        mst_set = [False] * self.verts

        parent[0] = -1 # First node is always the root

        for cout in range(self.verts):
            u = self.min_key(key, mst_set) # Pick the
                minimum distance vertex from vertices not
                yet processed.
            mst_set[u] = True # Put the min distance
                vertex in the shortest path tree

            for v in range(self.verts): # Update adj vert
                dist value dist > new distance
                if 0 < self.graph[u][v] < key[v] and
                    mst_set[v] is False: # Update the key
                    only if graph[u][v] is smaller than key
                    [v]
                    key[v] = self.graph[u][v]
                    parent[v] = u

        self.print_mst(parent)

if __name__ == "__main__":
    g = Graph(5)
    g.graph = [[0, 2, 0, 6, 0],
               [2, 0, 3, 8, 5],
               [0, 3, 0, 0, 7],
               [6, 8, 0, 0, 9],
```

```
                [0, 5, 7, 9, 0]]
g.mst_func()
```

When we run the program, we get figure 13.4.3. In the output, we have the edge between each node and its associated weight.

13.5. What are single-source shortest paths?

Shortest path problems involve finding a path between two vertices in a graph where the sum of the edge weights is minimized. It can be applied to undirected, directed, or mixed graphs.

In this section, we will talk about one of the most famous algorithms for this problem set: Dijkstra's algorithm. Dijkstra's algorithm is similar to Prim's MST algorithm and, in fact, Prim's algorithm can utilize Dijkstra's algorithm to improve the process. Dijkstra's algorithm is often used as the basis of other algorithms, such as Prim's, and is frequently used in artificial intelligence as uniform cost search as part of the idea of best-first search.

Dijkstra's algorithm creates a shortest path tree (SPT), with a given source as the root. Two sets are maintained; one set holds the vertices in the SPT while the other has vertices that have to be added to the SPT. At each step of the algorithm, we find the vertex that is not yet added and has a minimum distance from the source.

Assume we figure 13.5.1, with eight vertices and edge weights as marked. One implementation of Dijkstra's algorithm is demonstrated below[33]:

```
import sys

class Graph:
    def __init__(self, vertices):
        self.verts = vertices
        self.graph = [[0 for column in range(vertices)]
            for row in range(vertices)]

    def print_output(self, dist):
        print("Vertex Distance from Source")
        for node in range(self.verts):
            print(f"Vertex {node} is {dist[node]} units
                away")

    def min_dist(self, dist, spt_set):
        """Find vertex w/ min dist value"""
        min_dist = sys.maxsize # Initialize minimum
            distance for next node

        for v in range(self.verts):
            if dist[v] < min_dist and spt_set[v] is False:
                min_dist = dist[v]
                min_index = v

        return min_index
```

We make our usual Graph class and basic methods. We also include a minimum distance method to identify the vertex with the shortest distance. Next, we implement Dijkstra's algorithm and create a graph object in our execution code.

```
    def dijkstra(self, src):
        """Implement Dijkstra's algo"""
        dist = [sys.maxsize] * self.verts
        dist[src] = 0
        spt_set = [False] * self.verts

        for cout in range(self.verts):
            u = self.min_dist(dist, spt_set) # Get min
                distance from vertices not processed

            spt_set[u] = True # Add min dist vertex to SPT

            for v in range(self.verts): # If current dist
                > new distance, update dist value of adj
                vertices
                 if self.graph[u][v] > 0 and spt_set[v] is
                    False and dist[v] > dist[u] + self.
                    graph[u][v]:
                     dist[v] = dist[u] + self.graph[u][v]

        self.print_output(dist)

if __name__ == "__main__":
    g = Graph(9)
    g.graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],
               [4, 0, 8, 0, 0, 0, 0, 11, 0],
               [0, 8, 0, 7, 0, 4, 0, 0, 2],
               [0, 0, 7, 0, 9, 14, 0, 0, 0],
               [0, 0, 0, 9, 0, 10, 0, 0, 0],
               [0, 0, 4, 14, 10, 0, 2, 0, 0],
               [0, 0, 0, 0, 0, 2, 0, 1, 6],
               [8, 11, 0, 0, 0, 0, 1, 0, 7],
               [0, 0, 2, 0, 0, 0, 6, 7, 0]
               ]
    g.dijkstra(0)
```

When we run the program, we get figure 13.5.2. The output shows how many units each vertex is away from the source vertex.

Naturally, there are other algorithms that may be better for certain situations. For example, the Bellman-Ford algorithm is slower than Dijkstra's algorithm, but is more versatile in that it can handle graphs where edge weights are negative values. The key takeaway is that you need to have an idea of what's available, but don't beat yourself up trying to pre-optimize a solution. It's better to have a low-efficiency, brute force solution and refactor it than spend all your time in the planning phase.

13.6. What is all-pairs shortest paths?

While Dijkstra's algorithm is useful for finding the shortest path between two specific nodes, what if we want to know the shortest paths between all vertex pairs in a graph? One possible algorithm is the Floyd-Warshall algorithm, also called Floyd's algorithm but not to be confused with Floyd's Tortoise and Hare algorithm. (It is interesting to note that Floyd's algorithm is nearly the same as four other algorithms [Roy's, Warshall's, Kleene's, and Ingerman's], and all were published within a few years, or even months, of each other).

Floyd's algorithm works to find the shortest paths in a weighted graph with positive or negative edge weights, but no negative cycles. A solution matrix is initialized with the same data as the input graph matrix, and updated by considering all vertices as an intermediate vertex. The idea is to pick all the vertices, one by one, and update all the shortest paths which include the selected vertex as an intermediate vertex in the shortest path.

For every pair (i, j) of the source and destination vertices, and intermediate vertex k, there are two possible cases:

1. Vertex k is not an intermediate vertex in the shortest path from i to j. The distance between i and j is retained as the shortest distance.

2. Vertex k is an intermediate vertex in the shortest path. The distance between i and j becomes $dist[i][k] + dist[k][j]$ if $dist[i][j] > dist[i][k] + dist[k][j]$.

Assume we have figure 13.6.1:

One way to solve this problem is with the following implementation of Floyd-Warshall algorithm[34]:

```
import math

# Constants
VERTS = 4
INF = math.inf # This value will be used for vertices not
    connected to each other

def floyd_warshall(graph_matrix):
    """Initializing the solution matrix with same data as
       input graph matrix"""
    dist = [[j for j in i] for i in graph_matrix] # Output
        matrix

    for k in range(VERTS): # Add all vertices one by one
       to the set of intermediate vertices.
        for i in range(VERTS): # Select all vertices as
           source, one by one
            for j in range(VERTS): # Pick all vertices as
               destination for the above picked source
                dist[i][j] = min(dist[i][j], dist[i][k] +
                    dist[k][j]) # If k is on the shortest
                    path, update the dist
    print_output(dist)

def print_output(dist):
    print("Following matrix shows the shortest distances
       between every pair of vertices")
```

```
    print("Vert 0\tVert1\tVert2\tVert3")
    for i in range(VERTS):
        for j in range(VERTS):
            if dist[i][j] == INF:
                print("INF\t\t", end="")
            else:
                print(f"{dist[i][j]}\t\t", end="")

            if j == VERTS - 1:
                print("")

if __name__ == "__main__":
    graph = [[0, 5, INF, 10],
             [INF, 0, 3, INF],
             [INF, INF, 0, 1],
             [INF, INF, INF, 0]]
    floyd_warshall(graph)
```

In the program, we set the constant INF to indicate that we don't know what the value is for a distance. For all intents and purposes, it's infinite.

We create the output table (matrix) to initialize with the same data as the input. It will be updated as the algorithm operates. The main portion of the algorithm calculates that distance between each node, incrementing the previous values with the value of the current edge being considered. If the current value is less than the previous, that value is updated; otherwise, the previous value is kept as the shortest path.

The rest of the program is simply generating the output table, prepping the input data, and running the algorithm.When we run the program, we receive figure 13.6.2.

The output shows a table of vertices vs. distance. It would probably make more sense to have the same vertices listed at the start of each row, but you should understand what is being shown.

In essence, the algorithm populates the entire table with INF values, then fills it in as it moves around the graph. Let's have a deeper look :

- At Vert 0, the starting node, it's relation to itself is zero and all other vertices are INF.
- At Vert 1, it's the relation to itself is zero, but it has a distance of 5 to Vert 0. We haven't looked at any other nodes, so we don't know their distances, leaving them as INF.
- When the algorithm gets to Vert 2, it knows that the distance between itself and Vert 1 is 3, therefore the distance to Vert 0 is 5 + 3.
- Vert 3 is the final node to consider, and it simply adds the value of the edge between itself and Vert 2 to the other calculations, completing the table.

13.7. Summary

In this chapter, we discussed a variety of advanced graph concepts, including how to find all connected components, identifying cycles, sorting graphs topologically, creating minimum spanning trees, and finding the shortest paths in a graph. You

should have a better understanding of different ways to optimize code, both in initial estimates and during refactoring. A key factor is to recognize that there are many ways to solve the same problem and, while no one way is the best way, some choices are better than others.

Next chapter, we will look at dynamic programming, which utilizes recursion to solve sub-problems of a larger problem in order to simplify a complicated problem.

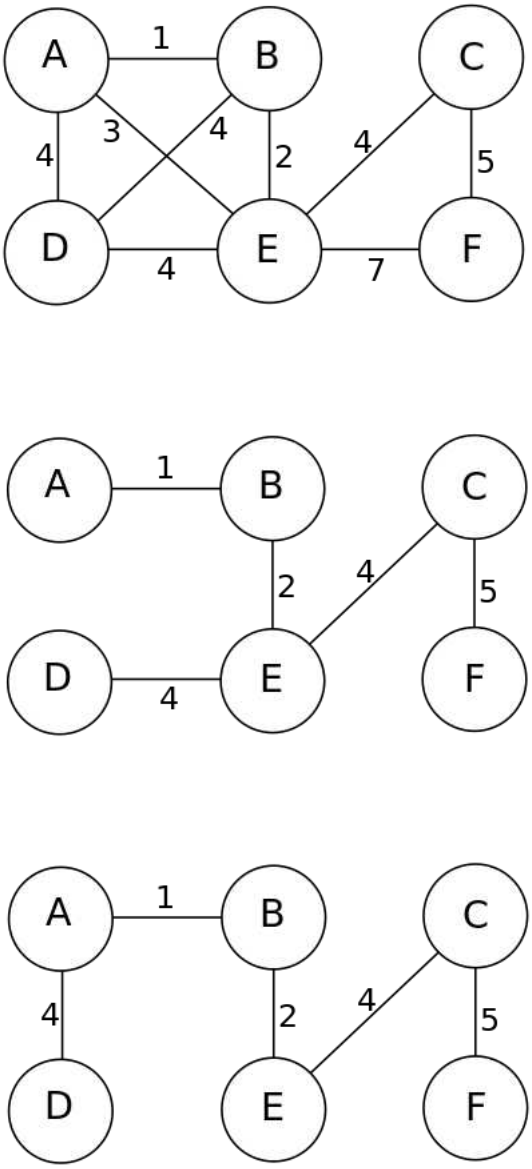

Figure 13.4.1.: Minimum spanning tree examples (Ftiercel [CC BY-SA 3.0])

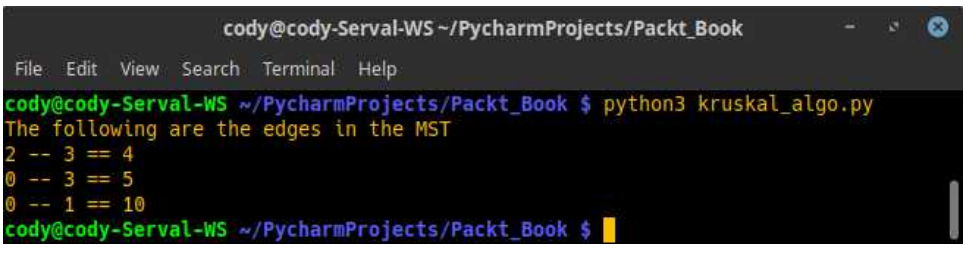

Figure 13.4.2.: Kruskal algorithm output

```
cody@cody-Serval-WS ~/PycharmProjects/Packt_Book
File  Edit  View  Search  Terminal  Help
cody@cody-Serval-WS ~/PycharmProjects/Packt_Book $ python3 prims_algo.py
Edge Weight
0-1 2
1-2 3
0-3 6
1-4 5
cody@cody-Serval-WS ~/PycharmProjects/Packt_Book $
```

Figure 13.4.3.: Prim's algorithm output

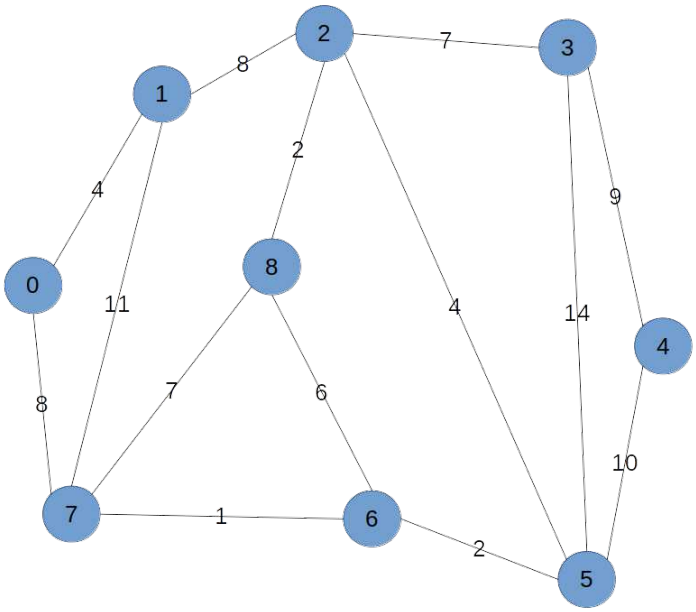

Figure 13.5.1.: Eight node graph

```
cody@cody-Serval-WS ~/PycharmProjects/Packt_Book
File  Edit  View  Search  Terminal  Help
cody@cody-Serval-WS ~/PycharmProjects/Packt_Book $ python3 dijkstra_algo.py
Vertex Distance from Source
Vertex 0 is 0 units away
Vertex 1 is 4 units away
Vertex 2 is 12 units away
Vertex 3 is 19 units away
Vertex 4 is 21 units away
Vertex 5 is 11 units away
Vertex 6 is 9 units away
Vertex 7 is 8 units away
Vertex 8 is 14 units away
cody@cody-Serval-WS ~/PycharmProjects/Packt_Book $
```

Figure 13.5.2.: Dijkstra's algorithm output

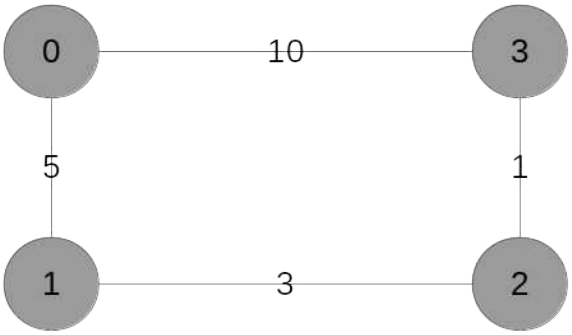

Figure 13.6.1.: Weighted graph

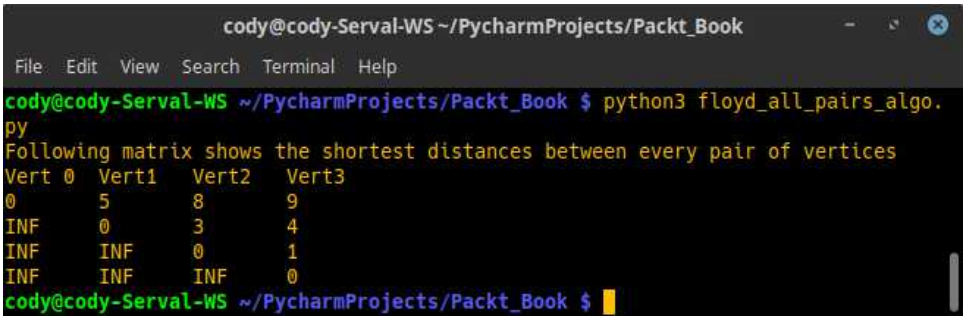

Figure 13.6.2.: Floyd algorithm output

14. Dynamic Programming

Dynamic programming is a programming methodology that utilizes optimization methods. The optimization comes from breaking down a complicated problem into smaller, sub-problems that can be handled recursively. Combining the sub-problem solutions together results in a solution for the overall problem.

By the end of this chapter, you should have a better understanding of memoization or tabulation to improve recursive solutions. These methods utilize cached values, rather than recomputing the solution every time. You should also better understand the differences between dynamic computing and divide and conquer, as well as having some useful examples to put into your programmer toolkit.

In this chapter, we will cover the following topics:

- How Does Dynamic Programming Compare to Divide and Conquer?
- What are some good things to know about dynamic programming?

14.1. How Does Dynamic Programming Compare to Divide and Conquer?

Two key factors make a problem solvable via dynamic programming: optimal substructure and overlapping sub-problems. An optimal substructure means that the solution to a problem can be found by combining the solutions to multiple sub-problems, normally via recursion.

Overlapping sub-problems means a recursive function should solve the same sub-problems again and again, rather than creating new sub-problems. If the sub-problems don't overlap but there is still an optimal substructure, then divide and conquer is more applicable.

14.1.1. How does simple recursion compare to dynamic programming?

In this section, we will look at two ways to solve a Fibonacci problem, one using simple recursion and one using dynamic programming.

14.1.1.1. Fibonacci with recursion

The code below solves the problem of finding the n^{th} number of a Fibonacci series using direct recursion.[35]

```
import sys

def Fibonacci(n):
    if n == 0:
        return 0 # First Fibonacci number is defined as 0
    elif n == 1:
        return 1 # Second Fibonacci number is defined as 1
    else:
```

```
        return Fibonacci(n - 1) + Fibonacci(n - 2)

if __name__ == "__main__":
    try:
        if len(sys.argv) > 1:
            fib_input = int(sys.argv[1])
            if fib_input < 0:
                raise ValueError
        else:
            fib_input = 9
        print(f"The value of Fibonacci {fib_input} is {
            Fibonacci(fib_input)}")
    except ValueError:
        print("Input must be at least zero.")
```

When we run the code, we receive figure 14.1.1. We can see that the program is able to calculate the Fibonacci value based on either an input value from the command line or, if none is provided, the default value in the program. The example code isn't perfect, as it only checks for negative numbers. Other input validation is left to the reader as a challenge.

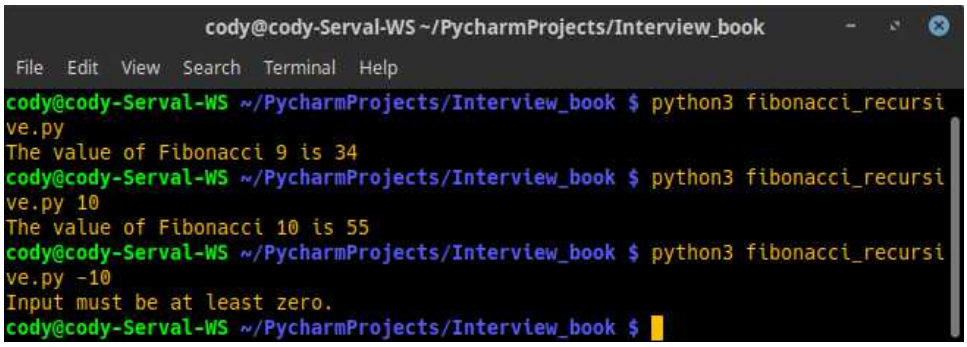

Figure 14.1.1.: Fibonacci recursion

14.1.1.2. Fibonacci with dynamic programming

So, how can be we improve on the recursive program we just saw? Dynamic programming allows us to store the numbers as they are generated. In the code example below, we use a simple list to maintain the generated values until we have the result.[35]

```
import sys

def Fibonacci(n):
    """Use an array to store Fibonacci numbers to prevent
       recalculation"""
    fib_array = [0, 1] # We already know the defined
       values for 0 and 1

    while len(fib_array) < n + 1: # Add results to array
       as long as we haven't exceeded 'n'
        fib_array.append(0)
    if n <= 1: # 0 and 1 are pre-defined
```

```
        return n
    else:
        # Add values to array
        if fib_array[n - 1] == 0:
            fib_array[n - 1] = Fibonacci(n - 1)

        if fib_array[n - 2] == 0:
            fib_array[n - 2] = Fibonacci(n - 2)

    fib_array[n] = fib_array[n - 2] + fib_array[n - 1]
    return fib_array[n]

if __name__ == "__main__":
    try:
        if len(sys.argv) > 1:
            fib_input = int(sys.argv[1])
            if fib_input < 0:
                raise ValueError
        else:
            fib_input = 9
        print(f"The value of Fibonacci {fib_input} is {
            Fibonacci(fib_input)}")
    except ValueError:
        print("Input must be at least zero.")
```

When ran, we receive the same output as before figure 14.1.2. This program gives the same results as the recursive program, as should be expected. The true test of dynamic programming will be shown in the next section, where we compare the execution profiles for each programming method.

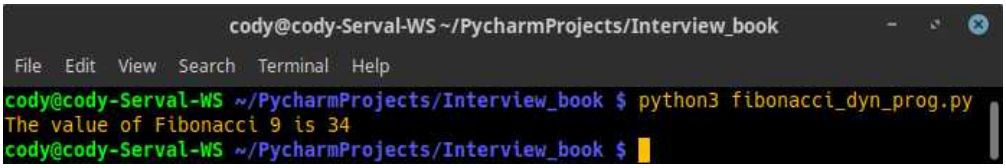

Figure 14.1.2.: Fibonacci dynamic

14.1.1.3. Comparing dynamic programming vs recursion

When we profile the two programs above using PyCharm, we get some interesting results. First, the recursion program in figure 14.1.3.

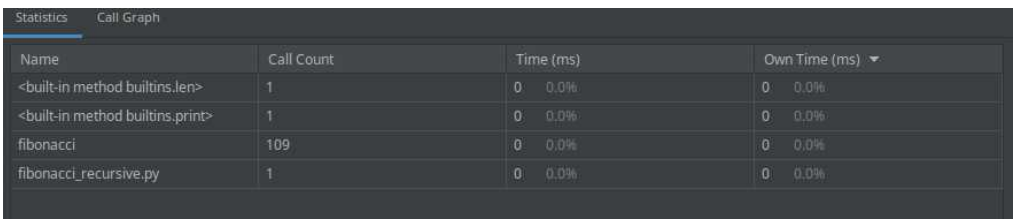

Figure 14.1.3.: Recursion profile

There are 109 calls to the Fibonacci function, and one call each for the built-in methods, but the program effectively took no time to run.

Next, we will look at the profile results for the dynamic program in figure 14.1.4.

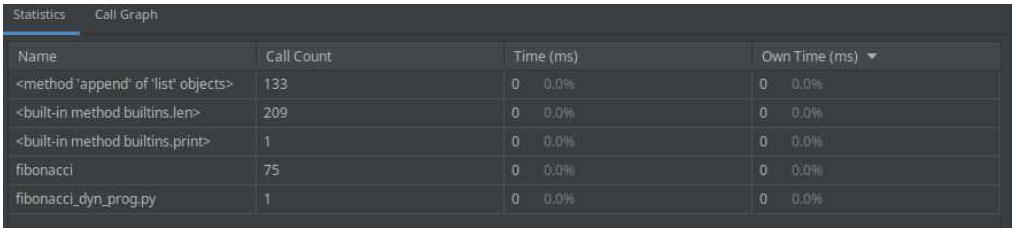

Figure 14.1.4.: Dynamic profile

This time, there were only 75 calls to the Fibonacci function, and hundreds of calls to built-in methods, which are already optimized. Again, the time to run was essentially instantaneous, but the overhead required to run the program was much less than standard recursion. Since the dynamic program already computed the preceding results as each Fibonacci number is calculated, it requires far fewer recursive calls.

14.1.2. What is longest increasing subsequence?

Longest increasing subsequence (LIS) is the subsequence of a given sequence where the subsequence elements are sorted (lowest to highest) and it is the longest subsequence possible. Note that the subsequence doesn't have to be unique nor contiguous within the given sequence. For example, in the sequence [13, 9, 20, 25, 5, 30, 50], the LIS is [13, 20, 25, 30, 50].

14.1.2.1. LIS using recursion

Below is a recursive way to find LIS[17]:

```
def lis_count(list_in, list_length):
    maximum = 1 # Max LIS length
    if list_length == 1: # Base result
        return 1
    max_lis = 1
    for length in range(1, list_length):
        result = lis_count(list_in, length)
        if list_in[length - 1] < list_in[list_length - 1]
            and result + 1 > max_lis:
            max_lis = result + 1

    maximum = max(maximum, max_lis) # Compare max_lis with
        overall maximum; update if needed

    return maximum

if __name__ == "__main__":
    input_list = [10, 22, 9, 33, 21, 50, 41, 60, 70, 5,
        71]
    list_len = len(input_list)
```

```
    print(f"Length of LIS is {lis_count(input_list, 
        list_len)}")
```

When ran, we get figure 14.1.5. The program tells us that the input array has an LIS of 7. The output of this program could be rewritten, as right now it only tells you how long the LIS is, but doesn't indicate what the input array is, nor what the values of the LIS are.

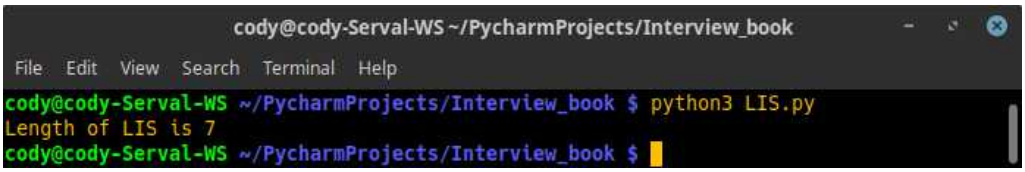

Figure 14.1.5.: LIS recursive output

14.1.2.2. LIS using dynamic programming

If we use dynamic programming, one potential solution is shown below[17]:

```
def lis(array):
    n = len(array)
    lis_array = [1]*n # Initialize LIS list
    for i in range(1, n):
        for j in range(0, i): if array[i] > array[j] and
            lis_array[i] < lis_array[j] + 1 :
             lis_array[i] = lis_array[j]+1

    maximum = 0 # LIS length

    for i in range(n):
        maximum = max(maximum, lis_array[i])

    return maximum

if __name__ == "__main__":
    arr = [10, 22, 9, 33, 21, 50, 41, 60, 70, 5, 71]
    print(f"Length of LIS is {lis(arr)}")
```

The results of this code are shown in figure 14.1.6. As expected, the dynamic programming solution provides the same results as the recursive solution. Again, the primary difference between the two comes down to their internal calls, as demonstrated in the next section using profile results.

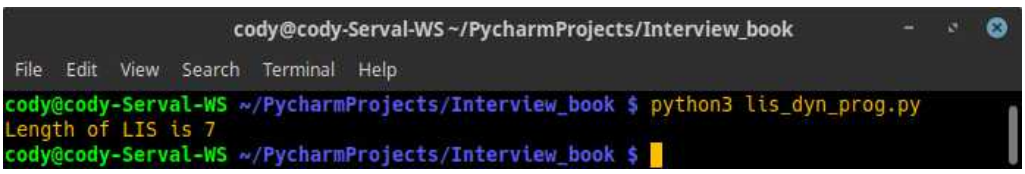

Figure 14.1.6.: LIS dynamic output

14.1.2.3. Comparing dynamic programming vs recursion

As we did before, we will look at the PyCharm profiles for the two examples. First, we look at the recursion code in figure 14.1.7. There were more than 1,000 calls to the LIS counting function, and most of those calls utilized the built-in *max()* method. In addition, we see that the execution time was 2ms.

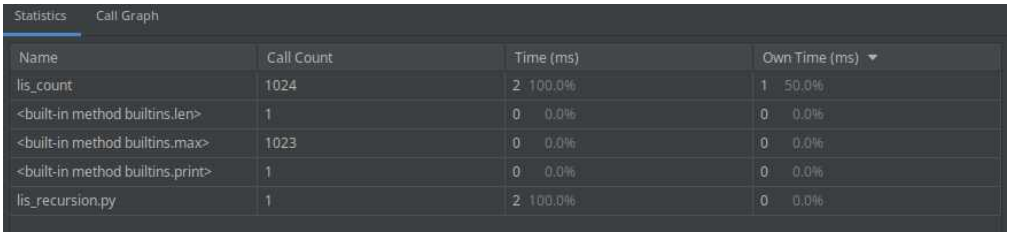

Figure 14.1.7.: LIS recursion profile

Next, we will look at the profile for the dynamic program in figure 14.1.8. In this case, there was only a single call to the LIS counting function, and 11 calls to *max()*. There was almost no delay in computing the results, clearly demonstrating the benefits of dynamic programming over regular recursion.

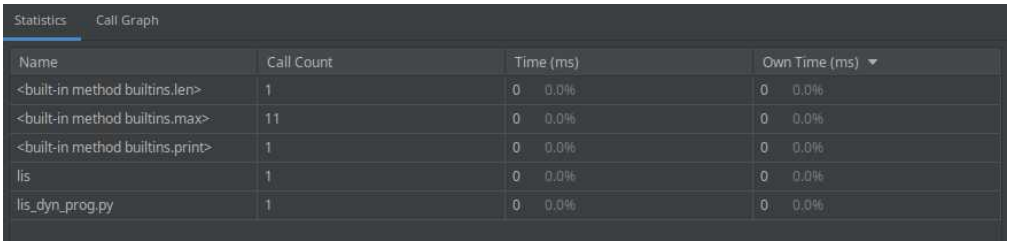

Figure 14.1.8.: LIS dynamic profile

14.2. What are some good things to know about dynamic programming?

Because dynamic programming is so useful, in terms of improving code efficiency, there are some useful things to know in order to utilize it most effectively. In this section we will look at the two ways to create dynamic programs, walk through the steps to solve dynamic programming problems, and then look at two different optimization problems.

14.2.1. What is top-down programming?

Top-down programming is based on memoization of a recursive problem: the results of previously developed problems are cached. When a new sub-problem is encountered, the cache table is checked to see if the sub-problem has already been solved. If so, the cached result is copied as the solution, otherwise the sub-problem is solved and the results stored to the table.

If only some of the sub-problems need to be solved to derive the solution for the overall problem, then memoization is preferred because sub-problems are solved lazily, i.e. only the calculations needed to be performed are made, minimizing resource use.

14.2.2. What is bottom-up programming?

Rather than checking whether a sub-problem has already been solved, bottom-up programming uses tabulation to populate a table. The sub-problems are solved first, iteratively, then their solutions are built up to provide the final solution, as small sub-problems are combined to make larger sub-problems and so on until the final problem is solved.

If a problem requires all sub-problems to be solved, then tabulation is the best choice because there is no overhead for recursion and memory can be preallocated for an array rather than dealing with a hash map.

14.2.3. What are the steps to solve a dynamic programming problem?

There are four primary steps to solve a dynamic programming problem, which will be discussed in detail here.

1. Is it a dynamic programming problem?
    a) Problems that check for maximum/minimum quantity, counting problems that specify counting under particular conditions, or probability problems are potential candidates for dynamic programming.
    b) If the problem is one of the previous candidates, then check if there are overlapping subproblems as well as optimal substructure. If so, then dynamic programming can be used.
2. What is the state?
    a) Dynamic programming problems are primarily about transitions of state. This is a vital step, as the transition depends on the choice of state definition used.
    b) The state is the set of parameters that uniquely identify a particular position in the problem. Ideally, this parameter set should be as small as possible to reduce the space required to calculate.
3. What is the relation between the states?
    a) Because the results of each subproblem will be combined to create the final solution, we need to find the relationship between the previous states to reach the current state.
    b) This is the hardest part of dynamic programming, as it requires intuition, educated guesses, and experience.
4. How can you add memoization or tabulation for the state?
    a) This part is easy, as we simply need to figure out how to best store a generated solution for a subproblem so it can be used again, rather than being recalculated.
    b) Depending on whether the solution is derived recursively or iteratively, you will use memoization or tabulation, respectively.

14.2.4. How can $O(2^n)$ problems be optimized?

Problems that have an $O(2^n)$ complexity are often subset sum problems. A classic $O(2^n)$ problem is the 0-1 knapsack problem: given a set of items (each

with a weight and value) and a backpack, determine the number of items to fill the backpack so that the total weight is less than or equal to the backpack's limit while keeping the value as large as possible. It should be obvious that this problem type is commonly utilized in resource allocation problems, such as finances, business, etc.

One possible solution is to consider all subsets of the items, then calculate the total weight and value of all subsets. Then, only consider the subsets where the total weight is less than the maximum allowed. Of those subsets, select the one with the largest value.

14.2.4.1. Basic 0-1 knapsack recursion solution

If we assume the following:

- item values = [50, 90, 120]
- item weights = [7, 23, 30]
- knapsack capacity = 50

A recursive solution might be like the following example[17]:

```
def knapsack_recur(max_weight, item_weight, item_value,
    item_quan):
    if item_quan == 0 or max_weight == 0:
        return 0
    if item_weight[item_quan - 1] > max_weight: # If
        weight > max_weight, exclude item
        return knapsack_recur(max_weight, item_weight,
            item_value, item_quan - 1)
    else: # Return the maximum of two cases: (1) item
        included, (2) item not included
        return max(item_value[item_quan - 1] +
            knapsack_recur(max_weight - item_weight[
            item_quan - 1], item_weight, item_value,
            item_quan - 1), knapsack_recur(max_weight,
            item_weight, item_value, item_quan - 1))

if __name__ == "__main__":
    values = [50, 90, 120]
    weights = [7, 23, 30]
    knapsack_capacity = 50
    print(f"Maximum value of items for capacity of {
        knapsack_capacity}: "
    f"{knapsack_recur(knapsack_capacity, weights, values,
        len(values))}")
```

When ran, we get figure 14.2.1. We see that, for a bag capacity of 50, the maximum value of items to place in it is 170. A profile of the program is shown in figure 14.2.2.

We can see that the recursive function was called a dozen times, and the built-in *max()* function five times. Overall processing time was negligible.

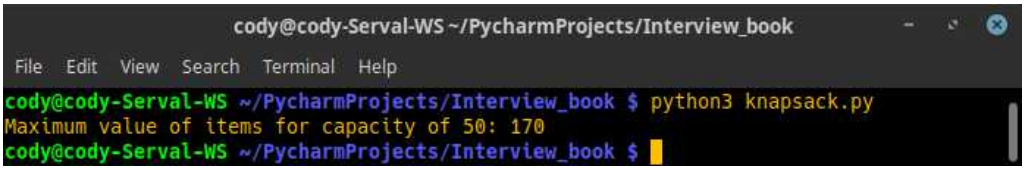

Figure 14.2.1.: Knapsack recursion

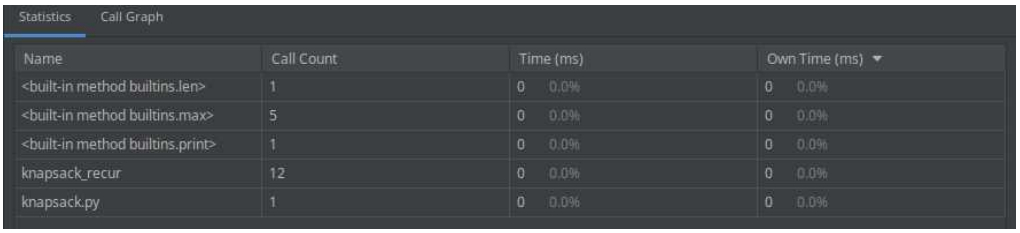

Figure 14.2.2.: Knapsack recursion profile

14.2.4.2. 0-1 Knapsack solution via dynamic programming

Using tabulation, we can create the following program[30]:

```
def knapsack_dp(capacity, item_weight, item_value,
    item_quan):
    max_value = [[0 for x in range(capacity + 1)] for x in
        range(item_quan + 1)]
    for quan in range(item_quan + 1):
        for weight in range(capacity + 1):
            if quan == 0 or weight == 0: # Base case if
                zero-values are provided
                max_value[quan][weight] = 0
            elif item_weight[quan - 1] <= weight: # Return
                the maximum of either item included or
                item not included
                max_value[quan][weight] = max(item_value[
                    quan - 1] + max_value[quan - 1][weight
                    - item_weight[quan - 1]], max_value[
                    quan - 1][weight])
            else:
                max_value[quan][weight] = max_value[quan -
                    1][weight] # If weight > max_weight,
                    exclude item
    return max_value[item_quan][capacity]

if __name__ == "__main__":
    values = [50, 90, 120]
    weights = [7, 23, 30]
    knapsack_capacity = 50
    print(f"Maximum value of items for capacity of {
        knapsack_capacity}: "
    f"{(knapsack_dp(knapsack_capacity, weights, values,
        len(values)))}")
```

When ran, we get figure 14.2.3. As expected, we receive the same results for this program as we did for the recursive program, as we are basically doing the same thing, except we are caching the results as they are created, rather than every time we loop through. When profiled, we get figure 14.2.4.

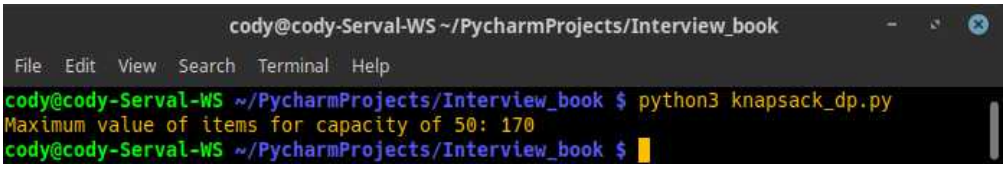

Figure 14.2.3.: Knapsack dynamic output

Figure 14.2.4.: Knapsack dynamic profile

When we compare the two different knapsack programs, we see that, while they essentially have the same time used (essentially zero), the recursive program is less efficient, as it calls the recursive function 12 times. The dynamic programming version only calls the function once, utilizing the built-in *max()* method to handle the "heavy lifting".

Since the built-in methods of Python are actually C code, they are better utilized than relying completely on your own code when possible. In this instance, dynamic programming presents a time complexity of $O(nW)$, where n is the number of items and W is the knapsack capacity.

14.2.5. How can $O(n^3)$ problems be optimized?

Matrix chain multiplication is an exponential complexity problem, or $O(c^n)$. The concept is: given a sequence of matrices, what is the most efficient way to multiply the matrices together? We aren't looking for the solution of the multiplications, just the most optimum procedure to perform the operations.

Matrix multiplication is associative: no matter how the parentheses are arranged, we get the same result every time. For example, $(ABC)D = (AB)(CD) = A(BC)D = \ldots$

The catch comes in the fact that how the parentheses are arranged can affect the underlying arithmetic operations to compute the product. For example, the following: $A = (10 \times 30), B = (30 \times 5), C = (5 \times 60)$. Simply moving the parentheses shows how the quantity of arithmetic operations changes, as shown below:

$$(AB)\,C = (10 \times 30 \times 5) + (10 \times 5 \times 60) = 1500 + 3000 = 4500\ operations$$

$$A\,(BC) = (30 \times 5 \times 60) + (10 \times 30 \times 60) = 9000 + 18000 = 2700\ operations$$

Thus, the problem is to determine which multiplication method is most efficient. The easiest, brute-force way is to simply place the parentheses in every possible position and then calculate the operation cost, tracking the minimum cost required to derive the solution. Because placing the parentheses is essentially recreating the original problem into multiple subproblems of smaller size, this is ideal for recursion and dynamic programming.

14.2.5.1. Matrix multiplication via recursion

The following is one recursion program to find the solution:

```
import sys

def chain_order(position, first_value, last_value):
    """Recursively move parentheses through matrices and
       determine which positions provide min number of
       operations"""
    min_count = sys.maxsize # Ensure we don't exceed the
       platform's space

    if first_value == last_value:
        return 0

    for curr_val in range(first_value, last_value):
        count = (chain_order(position, first_value,
           curr_val) + chain_order(position, curr_val + 1,
            last_value) + position[first_value - 1] *
           position[curr_val] * position[last_value])
        if count < min_count:
            min_count = count

    return min_count

if __name__ == "__main__":
    arr = [10, 2, 13, 9, 3]
    print(f"Most efficient number of operations is {
       chain_order(arr, 1, len(arr) - 1)}")
```

The output for this is shown in figure 14.2.5. Using recursion, we determine that the most efficient number is 348. Naturally, this value will change based on the input.

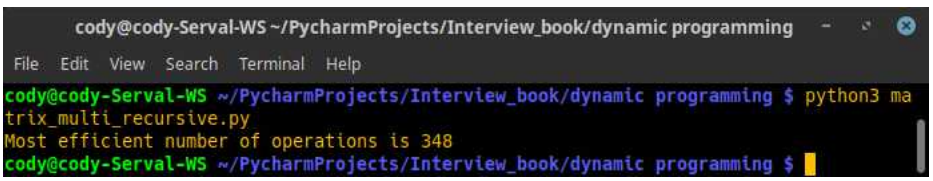

Figure 14.2.5.: Recursive matrix multiplication

The profile for this program is shown in figure 14.2.6. We can see that the recursive function was called 27 times but completion time was effectively nil.

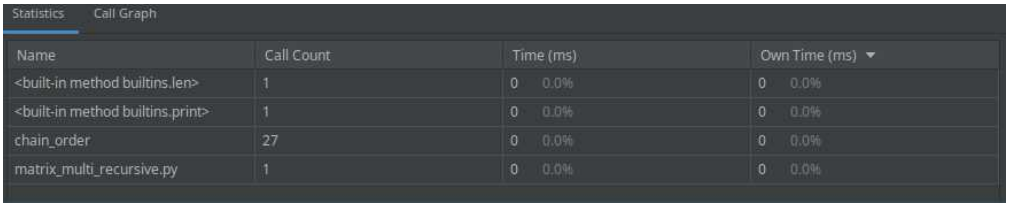

Figure 14.2.6.: Recursive matrix profile

14.2.5.2. Matrix multiplication via dynamic programming

Alternatively, we can use dynamic programming to perform the same operations and see what improvements we get, as shown in the code below[30]:

```
import sys

def matrix_chain(position, number):
    matrix = [[0 for x in range(number)] for x in range(
        number)] # Create initial matrix for initialization

    for first_val in range(1, number):
        matrix[first_val][first_val] = 0 # Zero count when
            multiplying only one matrix

    for chain_length in range(2, number):
        for first_val in range(1, number - chain_length +
            1):
            last_val = first_val + chain_length - 1
            matrix[first_val][last_val] = sys.maxsize #
                Ensure we don't exceed the platform's space
            for k in range(first_val, last_val):
                ops_cost = matrix[first_val][k] + matrix[k
                    + 1][last_val] + position[first_val -
                    1] * position[k] * position[last_val]
                if ops_cost < matrix[first_val][last_val]:
                    matrix[first_val][last_val] = ops_cost

    return matrix[1][number - 1]

if __name__ == "__main__":
    arr = [10, 2, 13, 9, 3]
    print(f"Most efficient number of operations is {
        matrix_chain(arr, len(arr))}")
```

When ran, we get figure 14.2.7. Again, as expected, we receive the same results as the recursive function. However, the profile (figure 14.2.8) shows something interesting, however.

Only one call is required to generate the solution, compared to the 27 function calls in the recursive program. The complexity for this solution is $O(n^3)$. As a reminder, the original recursive solution was $O(c^n)$, so we improved from an exponential complexity to a cubic complexity.

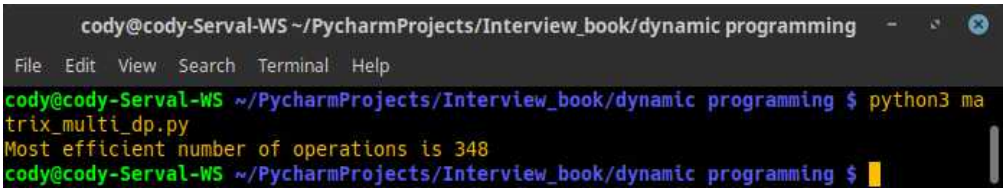

Figure 14.2.7.: Dynamic matrix output

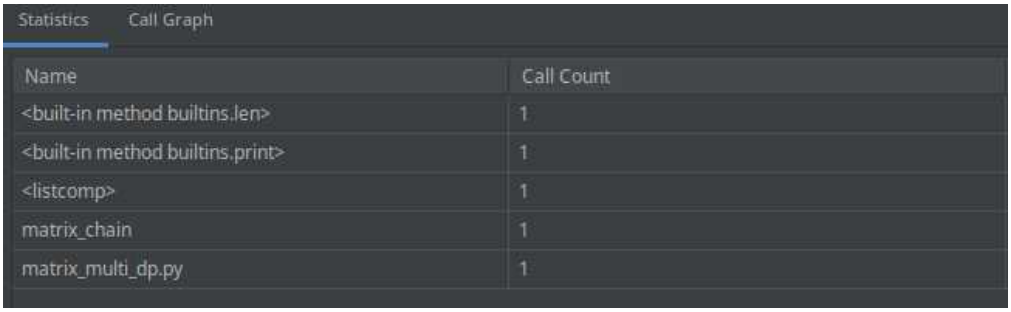

Figure 14.2.8.: Dynamic matrix profile

14.3. Summary

In this chapter, we looked at dynamic programming and how it compares to divide-and-conquer algorithms. We also compared a number of dynamic programming solutions vs. standard recursion. While recursion can make a solution clearer and more readable, it tends to use more memory than iterative solutions.

You should now have a better understanding of memoization or tabulation to improve recursive solutions. These methods utilize cached values, rather than recomputing each transaction. Memoization and tabulation skills are essential to dynamic programming and also make you a better programmer, as you can refactor a brute-force recursive program into something more efficient.

You should be able to better optimize code, depending on the particular situation, such as recursion, dynamic programming, or divide-and-conquer. Analysis of program operation, such as the time a particular function takes to complete, is another useful skill we demonstrated, as this is real-world performance as opposed to theoretical Big-O complexity.

In the next chapter, we will look at greedy algorithms, which improve on dynamic programming by finding the optimal solution at each stage, leading to the global optimum.

15. Greedy Algorithms

A greedy algorithm solves problems by determining the optimal choice at each stage, hopefully finding the global optimum. However, greedy algorithms don't always provide the most optimum solution, but having multiple locally optimal solutions may yield an approximate global solution that is close enough and in a reasonable amount of time.

In short, greedy algorithms give a reasonable answer in a reasonable amount of time, rather than a precise answer in an unreasonable amount of time. There are a lot of situations where stopping after a reasonable number of steps is better than the best solution.

In this chapter, we will cover the following topics:

- How Does Dynamic Programming Compare to Greedy Algorithms?
- What Makes Up a Greedy Algorithm?
- What are Some Greedy Algorithms?

15.1. How Does Dynamic Programming Compare to Greedy Algorithms?

Dynamic programming (DP) is, at its basic level, very simple: to solve a given problem, we solve different parts of the main problem via subproblems, then combine the separate solutions into the final solution for the overall problem. Typically, the brute-force/naive solution is to calculate the answers to the subproblems every time we start a new cycle. DP attempts to solve each subproblem only once, storing the result and referencing it on the next cycle. Thus, even as the number of subproblems expands, the time to complete and number of recursive calls doesn't appreciably expand as well.

Greedy algorithms (GAs) are similar, in that the overall problem is broken down into separate subproblems. The difference is that GAs make the best choice at each point in developing the solution. In other words, the best choice at the moment a subproblem is being solved is made, regardless of future outcome or all the solutions to the overall problem. Each subproblem's solution may be optimal for itself, and may be made based on previous decisions, but combined together, the individual solutions may not be the optimal solution for the overall problem. However, because they are individually optimal, the final solution is "good enough". Table 15.2 demonstrates the differences between DP and GA.

It should be emphasized that, while we say that GA subproblems are solved on their own, without looking back or revising previous solutions, the current subproblem may have been influenced by previous solutions. In other words, because the GA solves problems in a serial fashion, the solution for subproblem a may influence the setup for subproblem b, but the solution of subproblem b is its own solution, based on the conditions provided. If you were to go back and change the input conditions, the solution to subproblem b may be different the next time around.

Feature	Greedy Algorithm	Dynamic Program
Decision Making	At each step, we consider the current problem and solutions to previously solved subproblem to generate the optimal solution.	At each step, we make whatever choice makes the most sense at that moment, hoping it will help generate a globally optimal solution.
Optimal Solution	A global, optimal solution is guaranteed as all possible cases are considered and the best is selected.	A global, optimal solution isn't guaranteed, as the subproblem solutions are the best case solutions at that time.
Recursion	Recursion is minimized, as each subproblem's solution is cached for later reference.	Recursion may not even be necessary, as iterative solutions are possible.
Memory Use	Cached data (memoization or tabulation) requires additional memory, based on the quantity of subproblems to solve.	Memory efficient, as only the present problem is considered. Previous choices are not revised or looked at.
Time Complexity	Generally slow due to cache lookup, but faster than naive recursion.	Generally fast, as only the current subproblem is considered.
Operation	Solutions are solved via bottom-up or top-down operations by solving for smaller subproblems.	Solutions are developed in a serial fashion, where each subproblem is solved on its own, without looking back or revising previous solutions.
Example	0-1 Knapsack	Fractional Knapsack

Table 15.2.: Greedy vs. dynamic algorithms

In short, each subproblem is a self-contained problem that finds its own solution. However, taking each subproblem's solution and combining them together generates the general solution. Much like "The Butterfly Effect", changing the parameters may change the final outcome as the subproblem optimal solutions may change.

15.2. What Makes Up a Greedy Algorithm?

There are several factors that make up greedy algorithms as separate from other algorithms. It's important to note that GAs are most effective only on some mathematical problems, but not all. Therefore, they are not a panacea for other algorithms or solution processes.

For cases where a GA doesn't provide the best solution, be aware that they may actually produce the worst possible solution. This is because each subproblem isn't aware of the rest of the subproblems, and what is best for a particular subproblem may ultimately be the worst when combined with the others.

15.2.1. What are the components of greedy algorithms?

GAs need five components to function:

- A candidate set, from which the solution is created
- A selection function that selects the best candidate to add to the solution
- A feasibility function to determine if a candidate can contribute to the solution
- An objective function to assign a value to a (partial) solution
- A solution function that identifies when the complete solution has been found

15.2.2. What is the greedy choice property?

We have talked about this concept previously in this chapter, but we will clarify it here. The greedy choice property is the concept of selecting whatever choice makes the most sense at the moment of the subproblem is being solved; we don't care about any future subproblems nor all possible solutions to the subproblem.

Typically, the algorithm will iteratively make one greedy choice for each subproblem, never going back to reconsider the choices. Most problems where a GA will work include having a greedy choice property and an optimal substructure, like dynamic programming.

15.2.3. What is a matroid?

Matroids provide a link between graph theory, linear algebra, transcendence theory, combinatorial optimization, and semi-modular lattices.

A matroid is a mathematical structure that abstracts and generalizes the idea of linear independence in vector spaces. Put another way, a matroid is a finite set combined with a generalization concept (from linear algebra) that satisfies a set of properties for that generalized concept. For example, the set could be a matrix, while the generalized concept is the linear dependence/independence of any subset of rows within the matrix.

The significance of matroids is that, if an optimization problem is structured like a matroid, then an appropriate greedy algorithm will solve it optimally. In other words, a matroid problem is guaranteed to produce an optimal solution.

15.3. What are some Greedy Algorithm Examples?

Greedy algorithms are found in many areas of computer science and mathematics. This can easily be seen by considering the fact that a matroid is effectively a structure that guarantees a GA will always work, and as seen in the previous section, matroids are prevalent in many different areas. We will examine some of the more common GA applications in the following sections.

15.3.1. What is the Fractional Knapsack Problem?

The fractional knapsack problem is similar to the 0-1 knapsack problem from section 14.2.4.2. However, with the 0-1 problem, the items involved could not be broken apart, hence the 0-1 ("nothing or all") approach to the problem. The

fractional knapsack problem does allow the items to be broken up, making it easier to achieve maximum value.

In addition, the greedy approach is to determine the *value:weight* ratio for each item and sort the sequence based on that ratio. The items with the highest ratios are added first, continuing to add until we cannot add a whole item anymore, then move to the next item. Continue this process until we are left with a fractional part and add a fraction of an item to completely fill the bag.

One example of this process is shown in the example below[19]:

```
item_cost = []
def sort_items(array):
    """Calculate the value:weight ratio (cost) and sort,
       high to low, based on cost"""
    for tup in array:
        cost = tup[1] / tup[0]
        item_cost.append((tup[0], tup[1], cost))
        item_cost.sort()
    return item_cost

def fract_knapsack(capacity, items_arr):
    """Fill the bag with highest cost items first"""
    total_val = 0
    items_list = sort_items(items_arr) # Get sorted items
       list

    for item in items_list: # Separate tuples
        curr_weight = item[0]
        curr_val = item[1]
        if capacity - curr_weight >= 0: # Still room in
           bag?
            capacity -= curr_weight
            total_val += curr_val
        else: # Fill fractional capacity
            fract = capacity / curr_weight
            total_val += curr_val * fract
            capacity -= curr_weight * fract

    return total_val

if __name__ == "__main__":
items = [(10, 60), (40, 40), (20, 100), (30, 120)] #
   weight:value
sack_cap = 50
print(f"Max value in knapsack = {fract_knapsack(sack_cap, 
   items)}")
```

When ran, we receive figure 15.3.1. It's important to understand how the sorted list is created. To demonstrate this, here is the same program with notations of important values as the items are pulled from the list, just so you know what is being referenced:

```
item_cost = []
def sort_items(array):
```

Figure 15.3.1.: Fractional knapsack output

```
    """Calculate the value:weight ratio (cost) and sort,
       high to low, based on cost"""
    for tup in array:
        cost = tup[1] / tup[0]
        item_cost.append((tup[0], tup[1], cost)) item_cost
            .sort()
    return item_cost

def fract_knapsack(capacity, items_arr):
    """Fill the bag with highest cost items first"""
    total_val = 0
    items_list = sort_items(items_arr)

    for item in items_list:
        curr_weight = item[0]
        print(f"Current weight: {curr_weight}")
        curr_val = item[1]
        print(f"Current value: {curr_val}")
        print(f"Current capacity: {capacity}")
        print(f"Current total value: {total_val}")
        if capacity - curr_weight >= 0:
            capacity -= curr_weight
            print(f"New capacity: {capacity}")
            total_val += curr_val
            print(f"New total value: {total_val}")
        else:
            fract = capacity / curr_weight
            print(f"Fractional part: {fract}")
            total_val += curr_val * fract
            print(f"Total value after fraction: {total_val
                }")
            capacity -= curr_weight * fract
            print(f"Capacity after fraction: {capacity}")

    return total_val

if __name__ == "__main__":
items = [(10, 60), (40, 40), (20, 100), (30, 120)] #
    weight:value
sack_cap = 50
print(f"Max value in knapsack = {fract_knapsack(sack_cap,
    items)}")
```

When ran, we get figure 15.3.2. If no sorting of the items list is performed, i.e. we don't call *sort_ items()*, the result is much smaller figure 15.3.3. We can also see the results if we reverse the packing of the knapsack in figure 15.3.4.

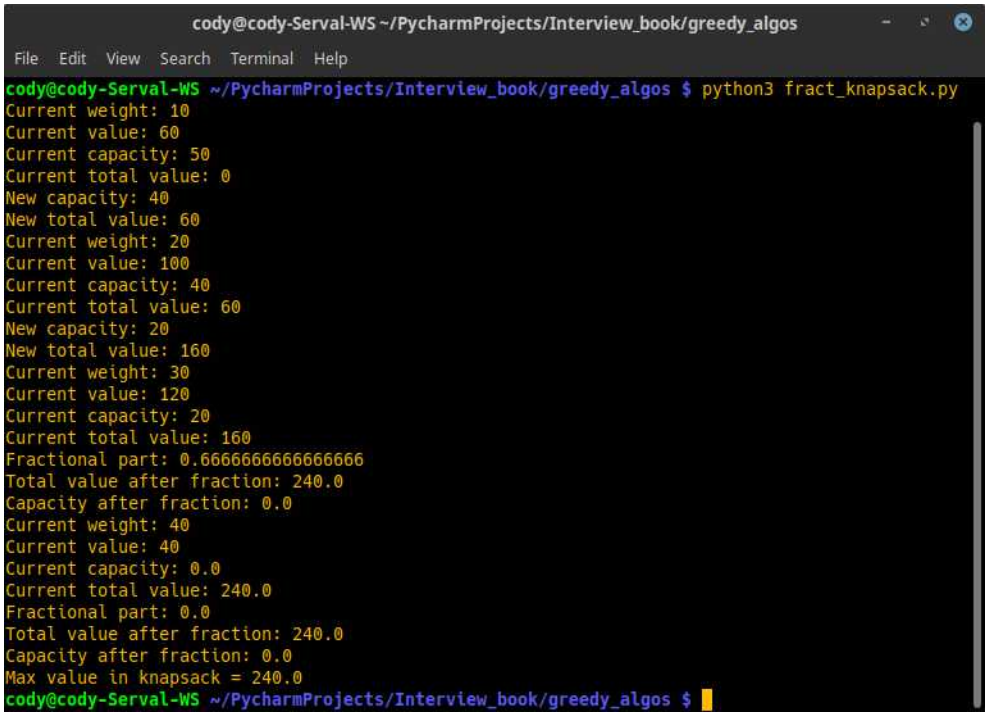

Figure 15.3.2.: Fractional knapsack step-by-step

You can easily see that the knapsack problem can be easily made incorrect if the sorting portion is not programmed correctly. Thus, it is important to understand the greedy choice property of the problem; in other words, the *value:weight* ratio. Otherwise, the answer you calculate won't be correct.

15.3.2. Job Sequencing Problem

Another common problem is sorting the order of tasks to maximize value. If we assume a sequence of jobs, where each job as a deadline and there is an associated profit for each job, how would we schedule the work list to maximize profit if only one job can be worked on at a time? This also assumes that the minimum amount of time for any given job is time unit = 1.

The following is the greedy algorithm for this problem:

1. Sort all jobs in descending order of profit
2. Initialize the result sequence as the first job in sorted jobs
3. For the remaining jobs, do the following: if the current job fits in the current result sequence without missing the deadline, add the job to the result. Otherwise, ignore the job.

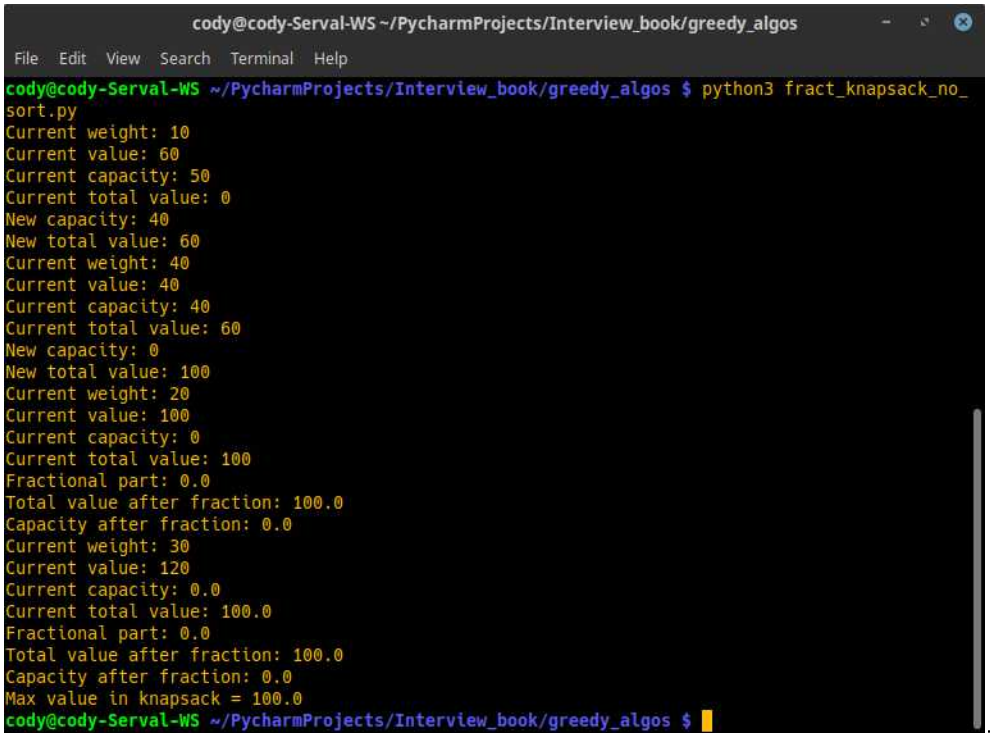

Figure 15.3.3.: Fractional knapsack no sort

The following is one example that implements this process[20]:

```
def job_scheduling(seq, num_jobs):
    """Schedule jobs in sequence, based on number of jobs
       to perform"""
    seq.sort(key=lambda x:x[-1], reverse=True) # Sort jobs
        in descending order of profit
    max_deadline = [False] * num_jobs # Max amount of time
        allowed
    job = ['-1'] * num_jobs # Sequence of jobs

    for i in range(len(seq)): # Iterate through all given
       jobs
        for j in range(min(num_jobs - 1, seq[i][1] - 1),
            -1, -1): # Find a free slot for this job,
            starting at end of 'job' list
            if max_deadline[j] is False:
                max_deadline[j] = True
                job[j] = seq[i][0]
                break
    return job

if __name__ == "__main__":
    # Job, deadline, profit
    arr = [['a', 2, 100],
           ['b', 1, 19],
           ['c', 2, 27],
           ['d', 1, 25],
```

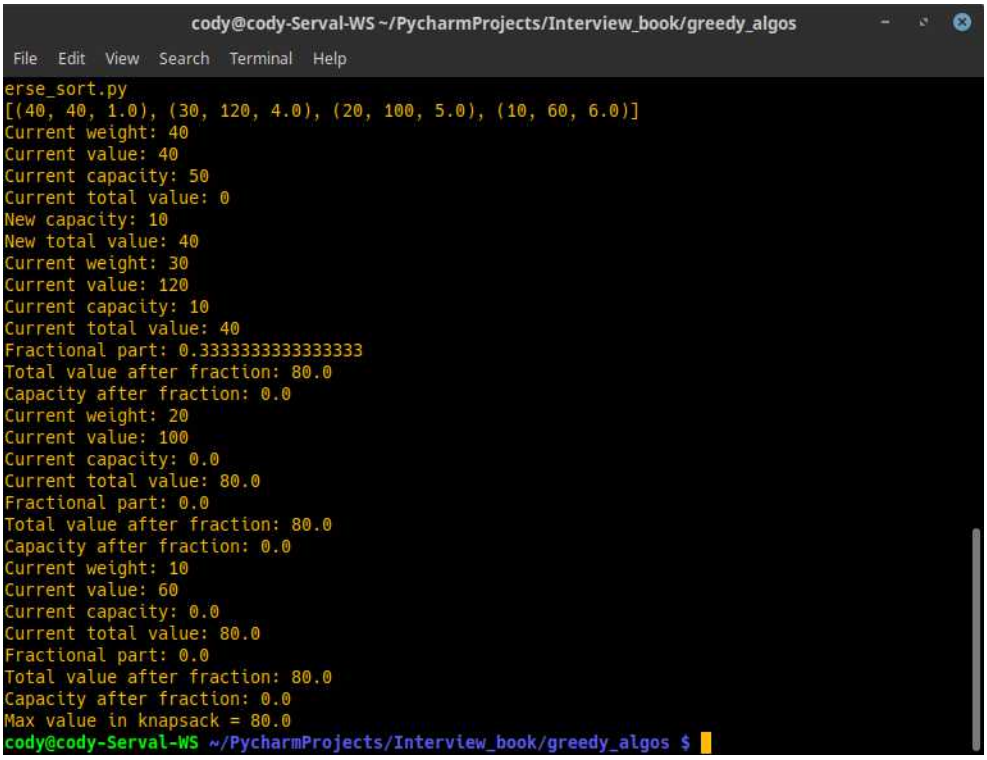

Figure 15.3.4.: Fractional knapsack reversed

```
            ['e', 3, 15]]
    print(f"For the following job matrix:")
    print(f"Job\t\tDeadline\tProfit")
    for item in arr:
        print(f"{item[0]}\t\t{item[1]}\t\t{item[2]}")
        print("\nThe following sequence of jobs that yield
            maximum profit:")
        print(job_scheduling(arr, 3))
```

Running this program results in figure 15.3.5. For this program, we show the input matrix of job profit vs. deadlines, then print the sequence that maximizes profit.

15.3.3. Policemen vs. Thieves

A common problem students are asked to solve using greedy algorithms is "policemen catching thieves"; finding the maximum number of thieves that can be caught by policemen. The problem is described as follows:

- Given an array of size n:
  - Each array element contains either a policeman or thief
  - Each policeman can capture only one thief
  - A policeman cannot capture a thief who is more than k units away

Solving it via brute force involves checking all possible combinations of police vs. thief and returning the maximum size set. Doing this, however, is an $O(c^n)$ complexity.

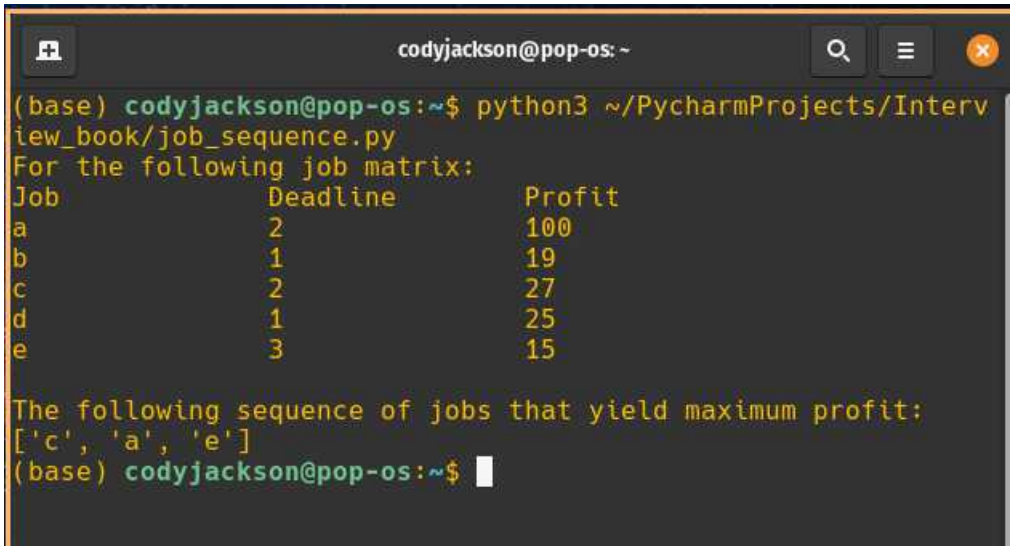

Figure 15.3.5.: Job sequence output

If we use a greedy algorithm, we can change it to an $O(n)$ solution. The trick is to identify the key greedy property of the problem: indices of the police and thieves:

1. Find the lowest indices of the policeman p and thief t. Increment a counter if $|p - t| \leq k$, then go to the next policeman and thief.
2. If not $|p - t| \leq k$, then just increment *min(p, t)* to the next p or t found.
3. Repeat steps 1 & 2 until the next p and t are found.
4. Return the counter value.

Below is one example of this algorithm[21]:

```
def cops_and_robbers(array, distance):
    i = 0
    thief_index = 0
    cop_index = 0
    result = 0
    array_length = len(array)
    thief_count = []
    cop_count = []

    # Store indices in list
    while i < array_length:
        if array[i] == "COP":
            cop_count.append(i)
        elif array[i] == "THIEF":
            thief_count.append(i)
        i += 1

    # Track lowest current indices
    while thief_index < len(thief_count) and cop_index <
        len(cop_count):
```

```
            if (abs(thief_count[thief_index] - cop_count[
                cop_index]) <= distance): # Thief can be caught
                result += 1
                thief_index += 1
                cop_index += 1
            elif thief_count[thief_index] < cop_count[
                cop_index]: # Increment the minimum index
                thief_index += 1
            else:
                cop_index += 1

        return result

if __name__ == "__main__":
    # problem = (array, k)
    problems = ((["COP", "THIEF", "THIEF", "COP", "THIEF"
        ], 2),
                (["THIEF", "THIEF", "COP", "COP", "THIEF",
                    "COP"], 2),
                (["COP", "THIEF", "COP", "THIEF", "THIEF",
                    "COP"], 3)
                (["THIEF", "THIEF", "COP", "COP", "THIEF",
                    "COP"], 1))
    for problem in problems:
        print(f"{problem[0]}; distance = {problem[1]}")
        print(f"Maximum thieves caught = {cops_and_robbers
            (problem[0], problem[1])}\n")
```

When ran, we get figure 15.3.6. We can visually verify whether the algorithm provided the expected results. In this case, we can confirm that, when $k = 2$ in the second array, three thieves were able to be captured. However, with the same arrangement (array 4) but $k = 1$, only two thieves were caught.

Figure 15.3.6.: Cops and thieves

15.4. Summary

In this chapter, we talked about greedy algorithms. We discussed how greedy algorithms differ from traditional divide-and-conquer algorithms, the components of a greedy algorithm, and then looked at some examples of greedy algorithms.

Next chapter, we will look at a variety of sorting and selection algorithms and compare the Big-O complexity of each type.

16. Sorting and Selection Algorithms

Sorting algorithms put items in a list into a particular order, most often alphanumeric characters sorted highest to lowest, or vice versa. Sorting is often conducted prior to using other algorithms, as it improves the efficiency of those follow-on algorithms; some algorithms can only be used if the data is sorted first.

Selection algorithms find the k^{th} number in a list; this number is known as the k^{th} *order statistic*. These algorithms are commonly used to find the minimum, maximum, and median elements within a list.

In this chapter, we will look at the following items:

- What are some $O(n^2)$ sorting algorithms?
- What are some $O(n \log n)$ sorting algorithms?
- What is an $O(n+k)$ counting sort?
- What are some $O(n)$ sorting algorithms?
- How does Python sorting work?

16.1. What are some $O(n^2)$ sorting algorithms?

We will cover some of the more commonly used $O(n^2)$ sorting algorithms in this section, starting with the classic bubble sort.

16.1.1. What is bubble sort?

Bubble sort is very simple: in a list of n elements, each k element is compared to the $k+1$ element that follows. If we are looking to sort from smallest to largest, then if the $k+1$ element is smaller than k, then the two elements swap positions. Otherwise, they are left in place and the $k+1$ element becomes the new k value for the next comparison. This continues through all elements in the list until all elements have been compared, leaving the final, sorted list.

One example of this is shown below[22]:

```
def bubble_sort(list_in):
    n = len(list_in)
    for i in range(n): # Traverse through all array
        elements
        for curr_elem in range(n-i-1): # Last i elements
            are already in place
            if list_in[curr_elem] > list_in[curr_elem +
                1]: # Current element > next element?
                list_in[curr_elem], list_in[curr_elem + 1]
                    = list_in[curr_elem + 1], list_in[
                    curr_elem] # Swap places

    return list_in
```

```
if __name__ == "__main__":
    arr = [34, 68, 11, 23, 90, 21, 0, 4]
    print(bubble_sort(arr))
```

When we run this code, we get figure 16.1.1

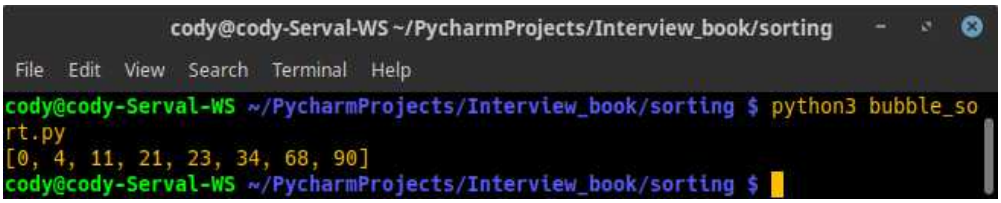

Figure 16.1.1.: Bubble sort output

The output list has been sorted correctly. Now, take a look if we eliminate the first *for* loop:

```
def bubble_sort(list_in):
    n = len(list_in)
    for curr_elem in range(n-1): # Last i elements are
        already in place
        if list_in[curr_elem] > list_in[curr_elem + 1]: #
            Current element > next element?
            list_in[curr_elem], list_in[curr_elem + 1] =
                list_in[curr_elem + 1], list_in[curr_elem]
                # Swap places

    return list_in

if __name__ == "__main__":
    arr = [34, 68, 11, 23, 90, 21, 0, 4]
    print(bubble_sort(arr))
```

This time, when we execute the code, we can an incorrect output figure 16.1.2. In this instance, the sorting algorithm has messed up, so make sure you have the correct loops needed to ensure proper sorting.

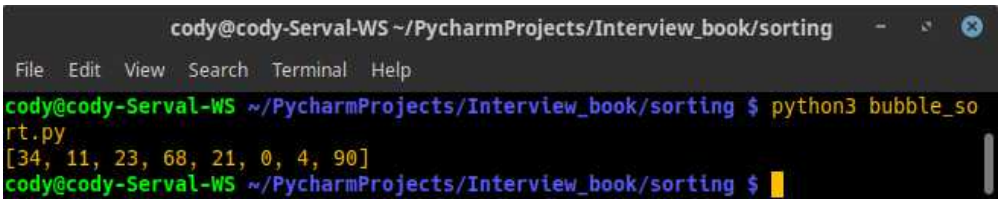

Figure 16.1.2.: Incorrect bubble sort

16.1.2. What is insertion sort?

Insertion sorting is similar to sorting playing cards: for each element in the list, check to see if it is less than the previous elements. If so, move it into a sorted list. If not, leave it in place. In this case, we are not checking two sequential elements. We are comparing the current element to the previous elements of the list and moving if necessary.

One example of this process is shown below[23]:

```
def insertion_sort(list_in):
    for val in range(1, len(list_in)): # '1' used because
        we have to compare to first element
        curr_val = list_in[val]
        comparator = val - 1
        while comparator >= 0 and curr_val < list_in[
            comparator]:
            list_in[comparator + 1] = list_in[comparator]
                comparator -= 1
        list_in[comparator + 1] = curr_val

    return list_in

if __name__ == "__main__":
    arr = [4, 8, 2, 1, 0, 13, 43, 22]
    print(insertion_sort(arr))
```

When executed, we see figure 16.1.3. We received the same type of sorted output as we did with the bubble sort, just with different values. When looking at their profiles (not shown here), they have the exact same parameters and execution time. This is to be expected, since they are both $O(n^2)$ algorithms.

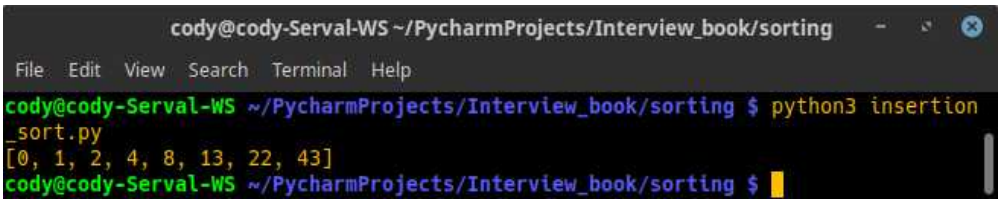

Figure 16.1.3.: Insertion sort output

16.1.3. What is selection sort?

Selection sort works by (assuming ascending order) repeatedly finding the minimum element in the unsorted portion and moving it to the beginning. Thus, two subarrays are created for the entire array: the sorted portion and the unsorted portion. Every iteration results in the smallest element in the unsorted subarray moving to the sorted subarray.

As usual, an example of this is shown below[24]:

```
def select_sort(input_list):
    for elem in range(len(input_list)): # Iterate over all
        array elements
        min_elem = elem
        for unsort_elem in range(elem + 1, len(input_list)
            ): # Find the minimum element in remaining
            unsorted array
            if input_list[min_elem] > input_list[
                unsort_elem]:
                min_elem = unsort_elem
```

```
            input_list[elem], input_list[min_elem] =
                input_list[min_elem], input_list[elem] # Swap
                elements

    return input_list

if __name__ == "__main__":
    list_in = [12, 34, 1, 3, 56, 8, 43, 99]
    print(select_sort(list_in))
```

When ran, we get figure 16.1.4figure 16.1.4:

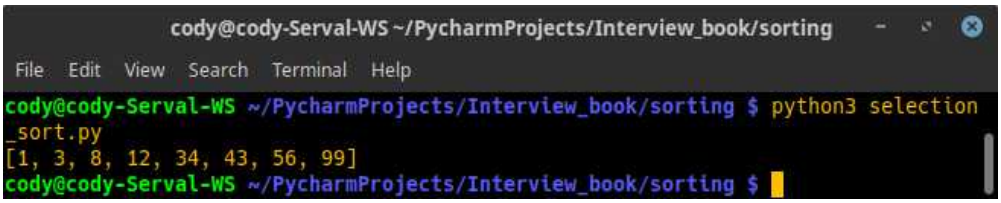

Figure 16.1.4.: Selection sort output

One thing to note about this algorithm, compared to bubble and insertion sorting, is that the built-in *len()* method is called nine times, as opposed to once for the other two examples. Granted, the time to completion is the same, as is the complexity, but it's something to be aware of if you are looking at maximum efficiency.

16.2. What are some *O(n log n)* sorting algorithms?

The algorithms we will cover in this section provide better performance than the algorithms covered in the previous section, as these algorithms grow in a superlinear fashion rather than polynomic. Merge sort and heap sort, discussed in section 7.3.2, are examples of this type of algorithm.

16.2.1. What is quick sort?

Like merge sort, quick sort is a divide-and-conquer algorithm. It selects an element as a pivot-point, then separates the array around that pivot. The pivot can be selected a variety of ways:

- Always pick first element
- Always pick last element
- Pick a random element
- Pick the median as the pivot

The partitioned array works by putting the pivot element in its correct position within a sorted array, then moves elements smaller than the pivot below the pivot and larger elements above it.

An example of this is shown below[25]:

```
def partition(array_in, start_index, end_index):
```

```
        """Use last element as pivot, move to correct position
            in sorted array, and place smaller and larger
           elements around it."""
        smallest_index = (start_index - 1)
        pivot = array_in[end_index]

        for element in range(start_index, end_index):
            if array_in[element] < pivot:
                smallest_index = smallest_index + 1
                array_in[smallest_index], array_in[element] =
                    array_in[element], array_in[smallest_index]
                     # Swap elements

        array_in[smallest_index + 1], array_in[end_index] =
            array_in[end_index], array_in[smallest_index + 1]

        return smallest_index + 1

    def quick_sort(input_array, start_index, end_index):
        if start_index < end_index:
            partition_index = partition(input_array,
                start_index, end_index)
            quick_sort(input_array, start_index,
                partition_index - 1) # Sort elements before
                partition
            quick_sort(input_array, partition_index + 1,
                end_index) # Sort elements after partition

        return input_array

    if __name__ == "__main__":
        arr = [23, 12, 78, 43, 0, 23, 1, 4]
        print(quick_sort(arr, 0, len(arr) - 1))
```

When ran, we get figure 16.2.1. It's interesting to note that Python's original *sort()* function was based on the quick sort algorithm. However, since version 2.3, it uses an adaptive merge sort algorithm.

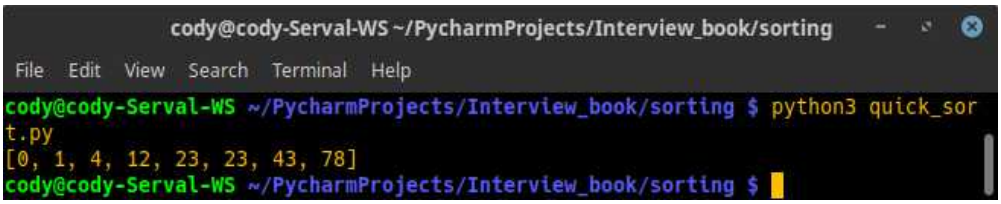

Figure 16.2.1.: Quick sort output

16.2.2. What is quickselect?

Quickselect is an algorithm to find the k^{th} smallest element in an unordered array; it is related to the quick sort algorithm. An example is shown below[26]:

```
def k_smallest(input_arr, small_index, big_index, k_index)
    :
    try:
        if 0 < k_index <= big_index - small_index + 1:
            pivot_position = partition(input_arr,
                small_index, big_index)

        if pivot_position - small_index == k_index - 1: #
            Position = k
            return input_arr[pivot_position]
        elif pivot_position - small_index > k_index - 1: #
            Position > k, recursive call left subarray
            return k_smallest(input_arr, small_index,
                pivot_position - 1, k_index)
        else: # Position < k, recursive call right
            subarray
            return k_smallest(input_arr, pivot_position +
                1, big_index,k_index - pivot_position +
                small_index - 1)

        if k_index > len(input_arr):
            raise ValueError
    except ValueError:
        return "ERROR: k value greater then list length"

def partition(array_in, left_index, right_index):
    right_value = array_in[right_index]

    for value in range(left_index, right_index):
        if array_in[value] <= right_value:
            array_in[left_index], array_in[value] =
                array_in[value], array_in[left_index]
            left_index += 1

    array_in[left_index], array_in[right_index] = array_in
        [right_index], array_in[left_index]

    return left_index

if __name__ == "__main__":
    arr = [12, 3, 5, 7, 4, 19, 26]
    k = 1
    if k == 1:
        suffix = "st"
    elif k == 2:
        suffix = "nd"
    elif k == 3:
        suffix = "rd"
    else:
        suffix = "th"
```

```
    print(f"The {k}{suffix} smallest value in list {arr}
        is {k_smallest(arr, 0, len(arr) - 1, k)}")
```

When we change the various k values in the main program, we get the figure 16.2.2 (based on the new k values).

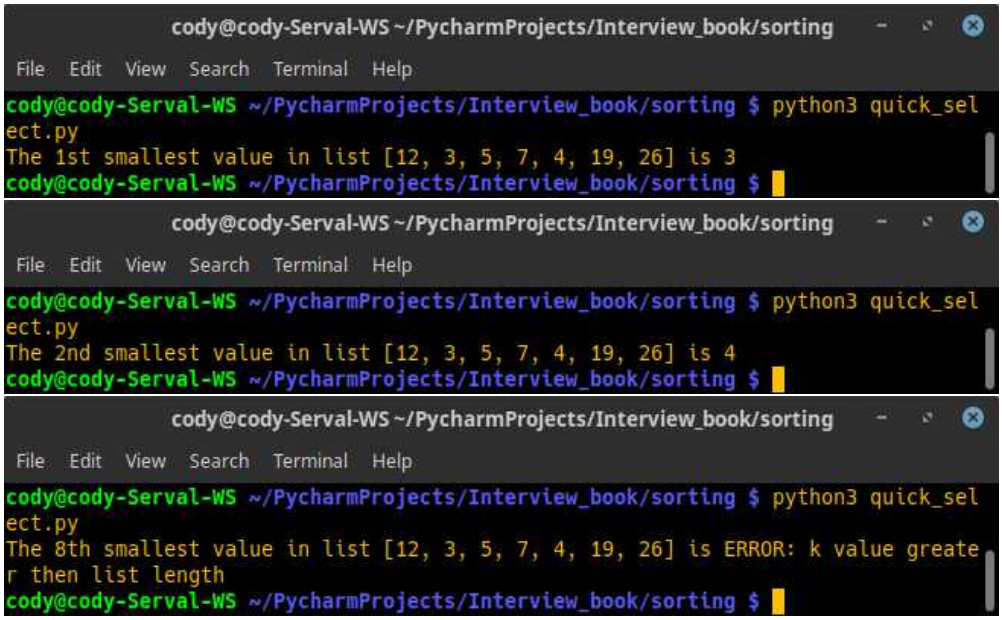

Figure 16.2.2.: Quick select output

First, you can see that the program provides the correct ordinal based on the input k value. Second, if the k value exceeds the number of elements in the array, an error is generated. Finally, you can also note that the correct k value is returned to match the requested input.

16.3. What is an $O(n + k)$ counting sort?

Counting sort is based on keys between a specified range. It counts the number of objects that have a distinct key value, similar to hashing. It then calculates the position of each object within the output sequence.

Counting sort is efficient as long as the input data range is not significantly greater than the quantity of objects to be sorted. Also, it is not comparison based sorting, therefore its complexity is $O(n)$ with space proportional to the range of data.

An example of this algorithm is shown below[17]:

```
def count_sort(arr):
    """Sort a string"""
    sorting_arr = [0 for i in range(256)]
    count_arr = [0 for i in range(256)] # Store count of
        individual characters
    answer_str = ["" for char in arr] # Store result as
        string is immutable

    for i in arr: # Store count of each character
        count_arr[ord(i)] += 1 # Get Unicode code point
            for string
```

```
    # Change count_arr[i] so it now contains actual
        position of this character in sorting_arr
    for i in range(256):
        count_arr[i] += count_arr[i-1]

    # Build the sorting character array
    for i in range(len(arr)):
        sorting_arr[count_arr[ord(arr[i])]-1] = arr[i]
        count_arr[ord(arr[i])] -= 1

    # Copy the sorting array to arr, so that arr now
        contains sorted characters
    for i in range(len(arr)):
        answer_str[i] = sorting_arr[i]

    return answer_str

if __name__ == "__main__":
    arr = "thisIsCamelCase"
    ans = count_sort(arr)
    print(f"When sorted, '{arr}' becomes '{''.join(ans)}'"
        )
```

When ran, we get figure 16.3.1. Notice that the sorting places capital letter before lowercase letters.

Figure 16.3.1.: Counting sort

16.4. What are some O(n) sorting algorithms?

$O(n)$ algorithms are slower than $O(\log n)$ algorithms, because they grow linearly in proportion to n, but are still very fast compared to all other algorithm complexities. In this section, we will talk about bucket sorting and radix sorting.

16.4.1. What is bucket sort?

Bucket sort (also called "bin sort") distributes array elements into several buckets, where each bucket is designated for a specific range of value. Each bucket is then sorted individually. Any algorithm can be used to sort the buckets.

In figure 16.4.1, the list of numbers at the top has been sorted into their appropriate buckets, i.e. bucket "0-9" contains the values 3 and 9 but, since there are no numbers between 10-19, that bucket is empty.

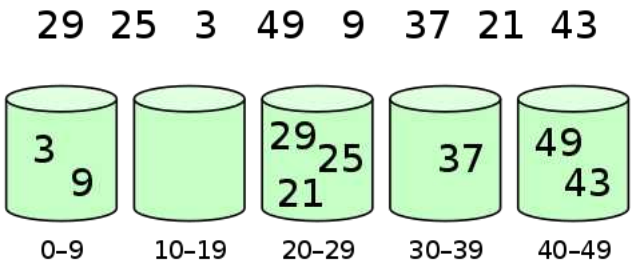

Figure 16.4.1.: Bucket sort initial sort (Zieben007 [CC BY-SA 4.0])

Next, in figure 16.4.2 the individual buckets are sorted. Once each bucket is sorted, the results from each bucket are combined into the final solution.

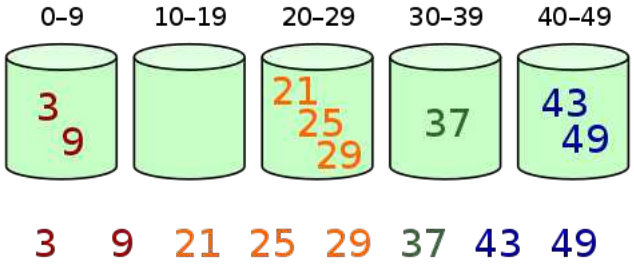

Figure 16.4.2.: Bucket sort final sort (Zieben007 [CC BY-SA 4.0])

An example of this is shown below[27]:

```
def bucket_sort(input_list):
    """Sort by bucket. Each bucket's size is 0.1, so first
       bucket's range is 0.0-0.1, second bucket is
       0.1-0.2, etc."""
    sorting_list = []
    bucket_num = 10

    for i in range(bucket_num):
        sorting_list.append([])

    for val in input_list: # Put array elements in
    different buckets
        bucket_index = int(bucket_num * val)
        sorting_list[bucket_index].append(val)

    for i in range(bucket_num): # Sort individual buckets
        sorting_list[i] = insertion_sort(sorting_list[i])
```

```
    k = 0
    for i in range(bucket_num): # Combine bucket results
        for val in range(len(sorting_list[i])):
            input_list[k] = sorting_list[i][val] k += 1

    return input_list

def insertion_sort(list_in):
    for val in range(1, len(list_in)): # '1' used because
        we have to compare to first element
        curr_val = list_in[val]
        comparator = val - 1
        while comparator >= 0 and curr_val < list_in[
            comparator]:
            list_in[comparator + 1] = list_in[comparator]
            comparator -= 1
        list_in[comparator + 1] = curr_val

    return list_in

if __name__ == "__main__":
    input = [0.321, 0.654, 0.651, 0.2135, 0.2065, 0.5216]
    print(f"Original list is {input}.\nSorted list is {
        bucket_sort(input)}")
```

When ran, we get figure 16.4.3, showing how the buckets are populated.

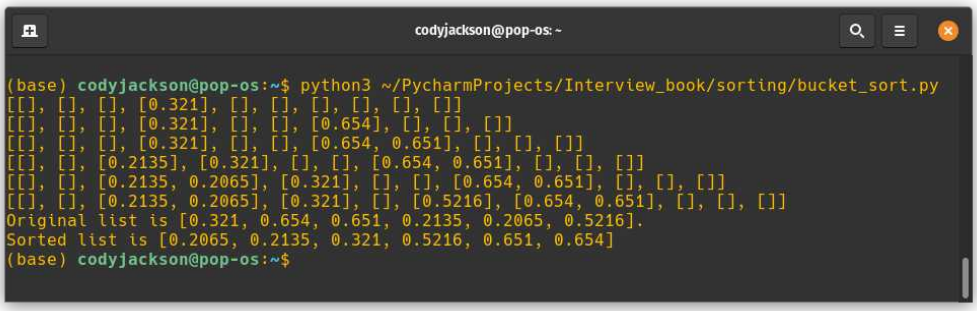

Figure 16.4.3.: Bucket sort output

You can see that we have created ten buckets within the enclosing list. Each bucket's size is 0.1, so first bucket's range is 0.0-0.1, second bucket is 0.1-0.2, etc. You can see that, in the third output line, the seventh bucket is populated with the values 0.654 and 0.651, as they are both within the same range value for that bucket.

Also note that the values placed in the buckets are not pre-sorted. In other words, they are placed in the buckets in the order they were put into the original list. Only after all elements in the list have been put into buckets is the actual sorting performed.

16.4.2. What is radix sort?

Radix sort is another non-comparative algorithm, like bucket sort. Also like bucket sort, it places items into buckets, but uses the radix (base) to sort them. For example, the decimal system has a radix of 10 while binary has a radix of 2.To further explain this, consider the following unsorted list: 170, 55, 7, 90, 802, 24, 2, 66. The algorithm to radix sort it is explained below:

1. Sort by least significant digit (1s place). Keep the original sequence in case of duplicate values. Add leading zeros as needed for missing digits. (Braces indicate buckets.)

   {170, 090}, {802, 002}, {024}, {055}, {066}, {007}

2. Sort by the 10s place. Again, keep original sequence for duplicate values.

   {802, 002, 007}, {024}, {055}, {066}, {170}, {090}

3. Sort by most significant digit (100s place).

   {002, 007, 024, 055, 066, 090}, {170}, {802}

4. Print final value: 2, 7, 55, 66, 90, 170, 802

An example of this is provided below[23]:

```
def exponent_sort(array_in, exponent):
    array_out = [0 for i in range(len(array_in))]
    count = [0] * (10) # Initialize count array to 0

    for i in range(len(array_in)): # Store occurrences in
        count array
        index = (array_in[i] // exponent) # Ensure integer
            created
        count[index % 10] += 1 # Get correct significant
            digit

    for i in range(1, 10): # Change count list to hold
        actual position of digit in output array
        count[i] += count[i - 1]

    i = len(array_in) - 1

    while i >= 0: # Build output list
        index = (array_in[i] // exponent)
        array_out[count[(index) % 10] - 1] = array_in[i]
        count[(index) % 10] -= 1
        i -= 1

    for i in range(len(array_in)): # Copy output list to
        sorted input list
        array_in[i] = array_out[i]

    return array_in

def radix_sort(arr):
```

```
    max_digits = max(arr) # Find the maximum number to
        know number of digits
    exp = 1

    while max_digits / exp > 0:
        result = exponent_sort(arr, exp)
        exp *= 10

    return result
if __name__ == "__main__":
    arr = [170, 90, 802, 2, 24, 55, 66, 7]
    print(f"Input list is {arr}. Sorted output is {
        radix_sort(arr)}")
```

The resultant output is shown in figure 16.4.4. The sorted output is the same as the algorithm example above, so the program worked correctly.

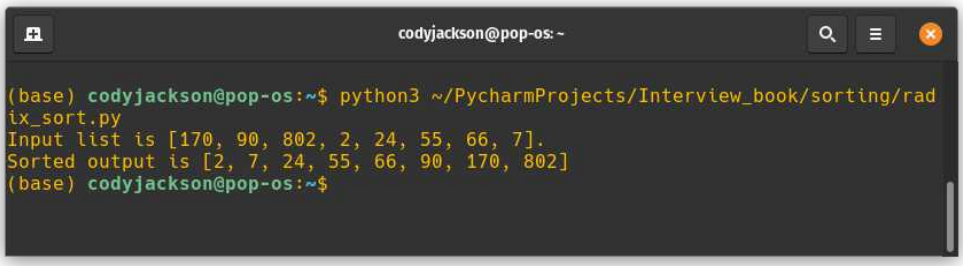

Figure 16.4.4.: Radix sort output

16.5. How does Python sorting work?

We've spent a lot of time talking about how to create various sorting algorithms for use in code. Luckily, Python provides a lot of this heavy lifting for us. All we have to do is call the correct sorting function and capture the results.

16.5.1. How does the Python *sort()* method work?

We briefly mentioned earlier that Python's built-in *sort()* method utilizes a merge sort algorithm. Let's compare how Python's optimized *sort()* method compares to the merge sort program we wrote in section 7.3.2.

As a refresher, below is the merge sort program we previously wrote:

```
def merge_sort(array):
    """Takes input array and rewrites it to sorted list"""
    if len(array) > 1:
        midpoint = len(array) // 2 # Midpoint of the array
            in whole numbers
        left_side = array[:midpoint] # Find left half
        right_side = array[midpoint:] # Find right half

        # Sort the two sides
        merge_sort(left_side)
```

```
        merge_sort(right_side)

        left_counter = right_counter = main_counter = 0

        # Copy data to temp arrays left_side[] and
            right_side[]
        while left_counter < len(left_side) and
            right_counter < len(right_side):
            if left_side[left_counter] < right_side[
                right_counter]:
                array[main_counter] = left_side[
                    left_counter]
                left_counter += 1
            else:
                array[main_counter] = right_side[
                    right_counter]
                right_counter += 1
            main_counter += 1

        # Check if any element was left on either side
        while left_counter < len(left_side):
            array[main_counter] = left_side[left_counter]
            left_counter += 1
            main_counter += 1

        while right_counter < len(right_side):
            array[main_counter] = right_side[right_counter
                ]
            right_counter += 1
            main_counter += 1

if __name__ == '__main__':
    arr = [22, 45, 8, 1, 12, 99, 32] # This list will be
        overwritten
    print(f"Given array is {arr}")
    merge_sort(arr)
    print(f"Sorted array is {arr}")
```

Next, we will create a simple sorted list program using Python's *sort()* method:

```
l = [3, 890, 23, 0, 23, 45, 12, 878, 56, 543, 276]

if __name__ == "__main__":
        print(f"The initial list is {l}")
        l.sort()
        print(f"The sorted list is {l}")
```

The length of our merge sort list is 7, compared to 11 in the Python sort program. We won't bother showing the output, as they are simply sorted lists. Yet, if we look at the profiles of each program, we see some interesting things.

In figure 16.5.1, we can see the output profile for our merge sort program. There were 72 calls to the *len()* method and 13 calls to the *merge_sort()* function that we wrote.

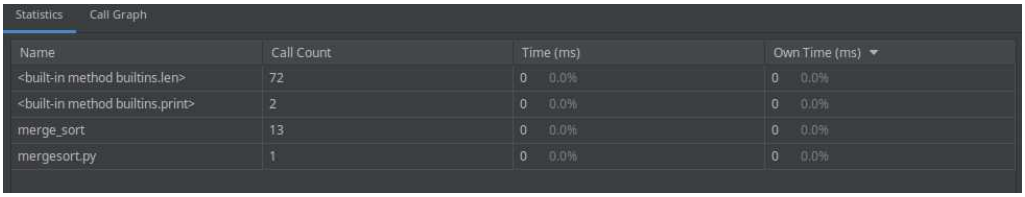

Figure 16.5.1.: Merge sort profile

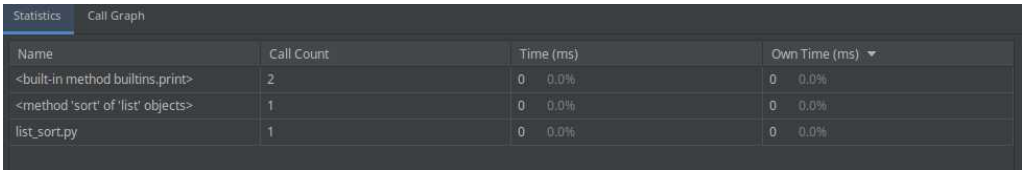

Figure 16.5.2.: Python sort profile

In figure 16.5.2, we can see that using Python's built-in sorting method was only called once. Granted, there was no time difference between the two, but obviously in terms of efficiency, using the built-in method is better.

The reason for this is primarily because Python's built-in methods are all C code underneath. They have been optimized over the years and, when called, actually call the compiled machine code from the underlying C language. In our program, it is pure Python code. Unless we rewrote the program to call an embedded C program, such as using ctypes or utilized Cython, using direct, interpreted Python code will result in more overhead.

This is one reason why "rolling your own" is not advised in Python. Python has been optimized to deliver best performance for nearly all built-in modules. Unless there is a bona fide reason to make your own version, it is better to simply rely on Python's implementations.

16.5.2. How does Python sort iterable objects?

While we're at it, let's take a look at sorting an iterable object. For any iterable, the generic *sorted()* function is available, as demonstrated below:

```
t = (3, 890, 23, 0, 23, 45, 12, 878, 56, 543, 276)

if __name__ == "__main__":
    print(f"The initial tuple is {t}")
    print(f"The sorted tuple is {sorted(t)}")
```

The output is shown in figure 16.5.3. As expected, the output is sorted. However, note that the actual object type that is returned is actually a list, rather than the original tuple. This applies to all iterable objects, including strings, as shown in figure 16.5.4.

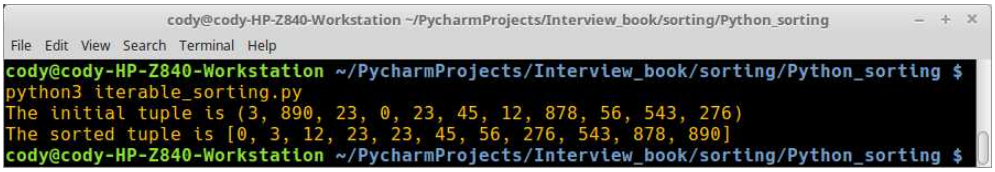

Figure 16.5.3.: Iterable sorting

Figure 16.5.4.: String sort

Again, the original input was a text string, but the *sorted()* function converted it to a list of string characters.

There are two main things to remember. First, *sorted()* takes an iterable object as an argument, while *sort()* is a method and called using dot nomenclature. Second, both ways of sorting generate a list object as the output type. Thus, you'll probably want to iterate over the actual output if you don't want the list object itself, such as "pretty printing" text to the screen.

On a final note, you could use *sorted()* with lists, but it's probably easier to use *sort()*. However, *sort()* works in-line; there is no output so you can't assign it to a variable but have to use the original list name.

16.5.3. How to sort a list of lists?

Sorting nested lists is nearly the same as normal sorting. Both *sort()* and *sorted()* provide for a key argument. The easiest way to handle nested lists is to use a lambda function as the key, as demonstrated below[28]:

```
student_tuples = [
('john', 'A', 15),
('jane', 'B', 12),
('dave', 'B', 10),
]
student_tuples.sort(key=lambda student: student[2]) # Sort
    by age
print(student_tuples)
```

The output is in figure 16.5.5. Even though the inner objects are actually tuples, we can still use the list *sort()* method, as the outer object is a list. If sorting a nested list of any sequence objects (including lists), you can also use the *sorted()* function as well.

Figure 16.5.5.: Sorting nested lists

16.5.4. What is the operator module?

The key-function operation shown in the last section is a very common action for programmers, so Python added the *operator* module. Rather than creating

a lambda function every time, you can use one of the methods in the operator module to perform the same actions, as demonstrated below[29]:

```
from operator import itemgetter, attrgetter, methodcaller

student_tuples = [
('henry', 'A', 15),
('jane', 'B', 12),
('dave', 'B', 10),
]

class Student:
    def __init__(self, name, grade, age):
        self.name = name
        self.grade = grade
        self.age = age

    def __repr__(self):
        return repr((self.name, self.grade, self.age))

    def weighted_grade(self):
        return 'CBA'.index(self.grade) / float(self.age)

    student_objects = [ Student('henry', 'A', 15),
                        Student('jane', 'B', 12),
                        Student('dave', 'B', 10),
                      ]

if __name__ == "__main__":
    print(sorted(student_tuples, key=itemgetter(0))) #
        Sort by index
    print(sorted(student_objects, key=attrgetter("age")))
        # Sort by name
    print(sorted(student_tuples, key=itemgetter(1, 2))) #
        Sort by grade, then age (index)
    print(sorted(student_objects, key=attrgetter("grade",
        "age"))) # Sort by grade, then age (name)
    print([(student.name, student.weighted_grade()) for
        student in student_objects]) # Calculate weighted
        grade
    print(sorted(student_objects, key=methodcaller('
        weighted_grade'), reverse=True)) # Sort by method,
        highest first
```

The output is shown figure 16.5.6. Compared to working with lambda functions, using the methods in the operator library are much easier to use, and make for cleaner code.

16.6. Summary

In this chapter, we looked at a variety of sorting and selection algorithms, some of which we touched on before in chapter 7 chapter. Of the algorithms we looked

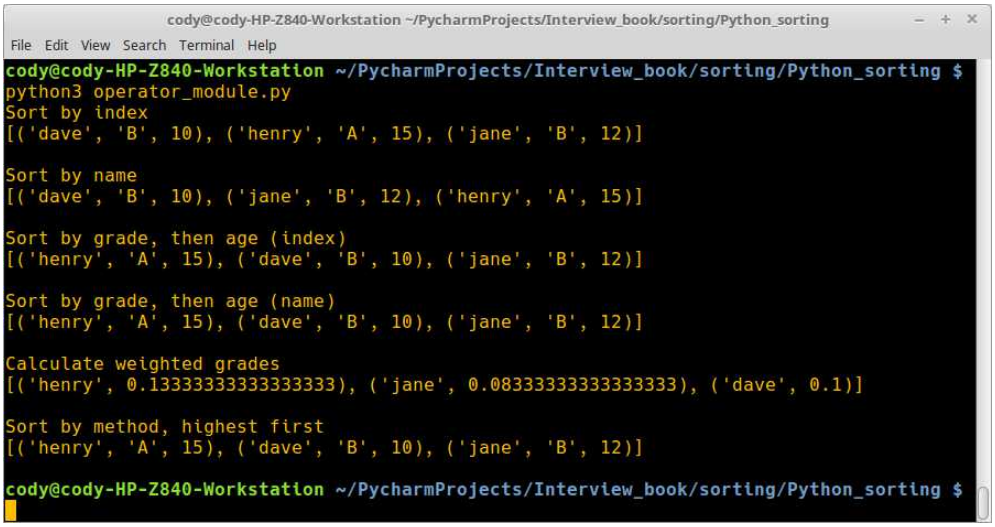

Figure 16.5.6.: Operator module

at here, we identified the Big-O complexity of each, seeing which ones performed better, as well as how the actual programs compared.

Finally, we looked at some of the built-in tools that Python provides for sorting, and found that the built-in tools are better optimized than most hand-written code, assuming you are using pure Python.

In the next chapter, we will look at bit manipulation, such as bitwise operators, built-in functions, combined bit operations, two's-complement binary.

17. Bit Manipulation

Bit manipulation (sometimes called bit twiddling or bit bashing) involves algorithmic modification of bits and other data that is shorter than the word-size for the computer. Common tasks include low-level device control, error detection and correction, compression, etc.

Programming languages often abstract out the actual manipulation, much like other interactions with the OS and low-level access. However, many programming languages do provide access to the actual bits via operators like AND, OR, NOT, XOR, and bit shifts.

In this chapter, we will discuss the following topics:

- What is Two's Complement?
- What are Python's Bitwise Operators?
- What are Python's Bit Manipulation Functions?
- What are Some Useful Combined Bit Operations?

17.1. What is Two's Complement?

Two's complement is mathematical operations on binary numbers; it is used in computing to represent signed numbers (positive and negative numbers). The definition of two's complement is: for any N-bit number, its complement is found with respect to 2^N.

The formula for calculating two's complement is is $\left(2^3 - n\right)_2$, where n is the integer. The subscript 2 identifies that it is base 2. Another way of calculating is to simply invert the binary value for the integer and add 1. For example, the integer number 3 has a binary value of 011. It's two's complement is 101.

Two's complement is the most common method for representing signed integers and fixed point binary values. In other words, if integer 3 is 011 in binary, its two's complement (101) is equal to -3. Thus, the sign of any integer can be changed by taking the two's complement of its binary value.

An advantage of two's complement is that arithmetic operations of addition, subtraction, and multiplication are identical for both signed and unsigned binary numbers, as long as the same number of bits are present. Any overflow bits are discarded. In addition, there is no representation for -0, so it is impossible to use it in an operation.

17.1.1. What About One's Complement?

On a side note, one's complement is the same process as two's complement, i.e. inverting all bits, but does not add 1 to the result. For most values, the one's complement is the opposite sign of the unsigned value. However, it does have problems with 0 and -1.

Looking at table 17.1, we can see that the binary values of the unsigned integers doesn't match the one's complement.

Unsigned Value	Binary	One's Complement
2	00000010	2
1	00000001	1
0	00000000	0
255	11111111	-0
254	11111110	-1
253	11111101	-2

Table 17.1.: One's compliment

Negative 1 and beyond are offset from their unsigned values, while 0 has both a positive and negative version. Negative 0 is commonly disregarded, but the standard for floating-point arithmetic does require both +0 and -0, because 1/+0 = +∞ and 1/-0 = -∞; thus, they are used when dealing with limits. The only time division is undefined is ±0/±0 and ±∞/±∞.

Another problem is arithmetic. Addition potentially results in an extra bit on the left side, extending past the 8 bits. When this happens, the carried bit performs an end-around carry and is added back in at the right-most bit.

Subtraction has a similar problem, except that borrows are taken from the left. If the borrow extends past the left-most bit, an end-around borrow occurs and the bit is actually subtracted from the right-most bit.

Because two's complement does not have any of these problems, it is the preferred method for bitwise operations.

17.2. What are Python's Bitwise Operators?

Bitwise operations in Python are performed using two's complement with an unlimited number of sign bits available. Bitwise operations can only be performed on integers, and in mathematical ordering, bitwise operations are lower than numeric operations but higher than comparisons. However, the bitwise unary operation ~ has the same priority as unary numeric operations + and -. Table 17.2 lists Python's bitwise operations, in descending order of priority.

Operation	Notes
x | y	bitwise *or* of x and y
x ^ y	bitwise *exclusive or* of x and y
x & y	bitwise a*nd* of x and y
x << n	x shifted left by n bits (same as multiplying by 2) *
x >> n	x shifted right by n bits (same as dividing by 2) *
~x	invert the bits of x

Table 17.2.: Python bitwise operations
*Negative shift counts are illegal and raise a *ValueError* exception

Let's look at how these actually operate. The following code shows these operators in use.

```
int1 = 60 # 60 = 0011 1100
int2 = 13 # 13 = 0000 1101

print(f"Binary value of {int1} = {bin(int1)}")
```

```
print(f"Binary value of {int2} = {bin(int2)}")
print("")
print(f"Bitwise AND of {int1} and {int2} = {int1 & int2}, or {bin(int1 & int2)}")
print(f"Bitwise OR of {int1} and {int2} = {int1 | int2}, or {bin(int1 | int2)}")
print(f"Bitwise XOR of {int1} and {int2} = {int1 ^ int2}, or {bin(int1 ^ int2)}")
print(f"Bitwise NOT of {int1} = {~int1}, or {bin(~int1)}")
print(f"Bitwise shift left by two bits of {int1} = {int1 << 2}, or {bin(int1 << 2)}")
print(f"Bitwise shift right by two bits of {int1} = {int1 >> 2}, or {bin(int1 >> 2)}")
```

When executed, we get figure 17.2.1. Note that, when displaying binary values, Python does not pad leading zeros to make even bytes. Also note that, when performing the NOT operation, Python includes the negative sign on the binary value.

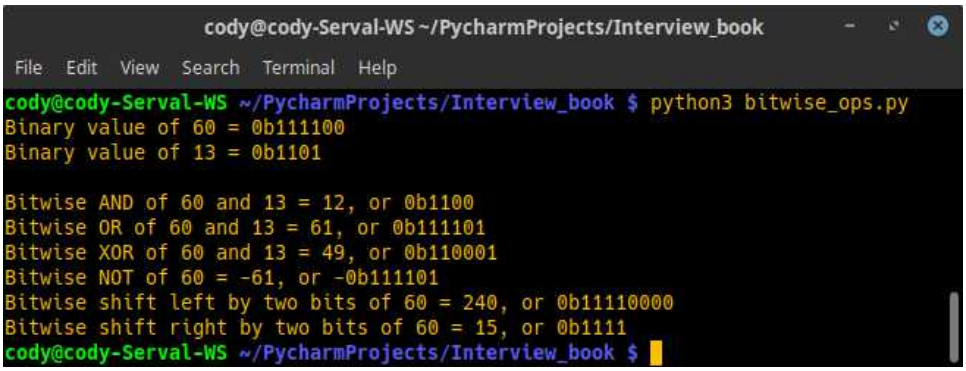

Figure 17.2.1.: Bitwise output

17.3. What are Python's Bit Manipulation Functions?

Python includes a number of built-in functions to help deal with bits, bytes, and general binary manipulation.

17.3.1. Conversions

Python can convert between integer, hexadecimal, octal, and binary values. The code below provides an example of this:

```
binary = 0b1110110
print(f"Binary value: {bin(binary)}")
print(f"Binary to integer: {binary}") # Auto conversion
print(f"Binary to hex: {hex(binary)}")
print(f"Binary to octal: {oct(binary)}")
print(f"Binary to string: {chr(binary)}")
print(f"String to integer: {ord('v')}")
print(f"String to binary: {bin(ord('v'))}")
print(f"Integer to binary: {bin(118)}")
print(f"Integer to hex: {hex(118)}")
```

```
print(f"Integer to octal: {oct(118)}")
print(f"Hex string to integer: {int('0x76', 16)}")
print(f"Integer to binary string: {str(bin(118))}")
print(f"Powers of 2\n"
      f"1 << 0 = {1 << 0}\n"
      f"1 << 1 = {1 << 1}\n"
      f"1 << 2 = {1 << 2}\n"
      f"1 << 3 = {1 << 3}\n"
      f"1 << 4 = {1 << 4}\n"
      f"1 << 5 = {1 << 5}")
```

When ran, we get figure 17.3.1. Notice that the second line automatically converts binary to integer. No special function call is required.

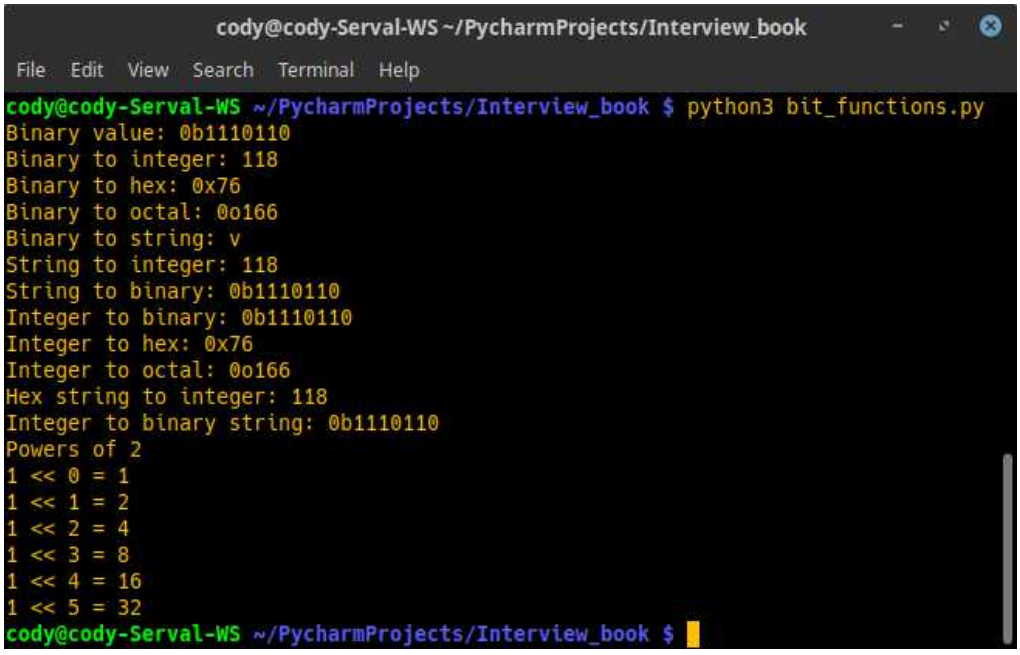

Figure 17.3.1.: Bit function output

17.4. What are Some Useful Combined Bit Operations?

With all these bit manipulation methods in mind, we will cover some useful and not-so-useful applications of bit hacking. The following examples are Python conversions of the public domain C language examples collected by Sean Anderson at http://graphics.stanford.edu/~seander/bithacks.html.

17.4.1. Detect if two integers have opposite signs

Using XOR, we can test two numbers to see if their signs differ:

```
def check_signs(x, y):
    if x ^ y < 0:
        return True
    else:
        return False
```

```
if __name__ == "__main__":
    print(check_signs(4, 8))
    print(check_signs(4, -8))
```

The output is shown in figure 17.4.1. In this case, False indicates that the two numbers have the same sign and True means they have different signs.

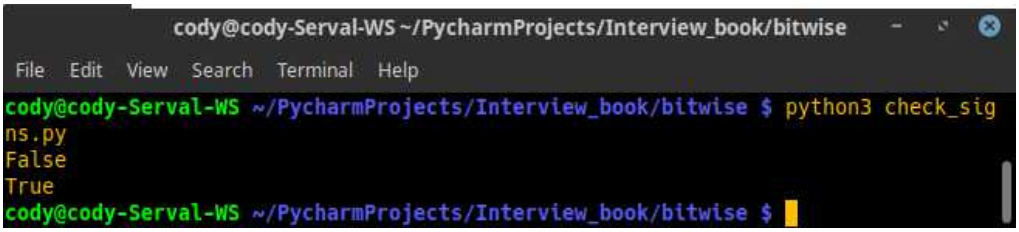

Figure 17.4.1.: Check signs output

17.4.2. Checking if a number is a power of 2

There are easy ways to check whether a number is a power of two, but the code below uses bitwise AND to do it:

```
def power_2(x):
    if x and not (x & (x - 1)): # Ensure 0 not considered
        True
        return True
    else:
        return False

if __name__ == "__main__":
    print(power_2(0))
    print(power_2(2))
    print(power_2(3))
    print(power_2(4))
```

The output is figure 17.4.2. If the number provided is a power of 2, *True* is returned. If not, or if the number is zero, *False* is returned.

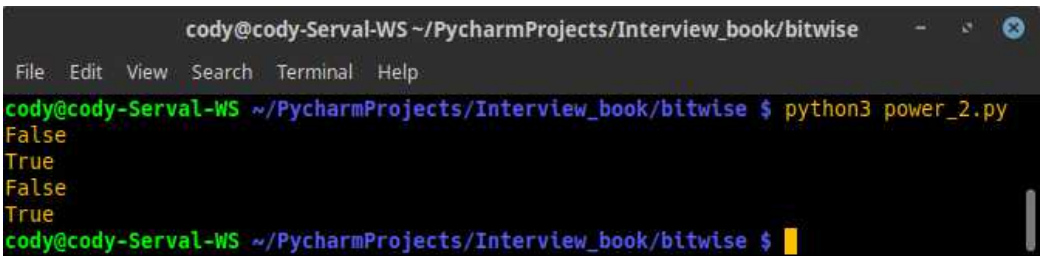

Figure 17.4.2.: Power of 2 output

17.4.3. Conditionally set or clear bits

In the following code, we have an input *word* and a *mask* value, both integers. The *flag* is a boolean value. Depending on the *flag* value, the *mask* will be com-

bined with the *word* or not; the return value will be the modified or unmodified word value.

```
def set_clear_bits(mask, word, flag):
    return (word & ~mask) | (-flag & mask)
if __name__ == "__main__":
    print(set_clear_bits(2, 1, True))
    print(set_clear_bits(2, 1, False))
```

When ran, we get figure 17.4.3. In the output, we can see that when the flag was set to True, the two integers were added together. However, when set to False, the numbers were not added and the unmodified value of word was returned.

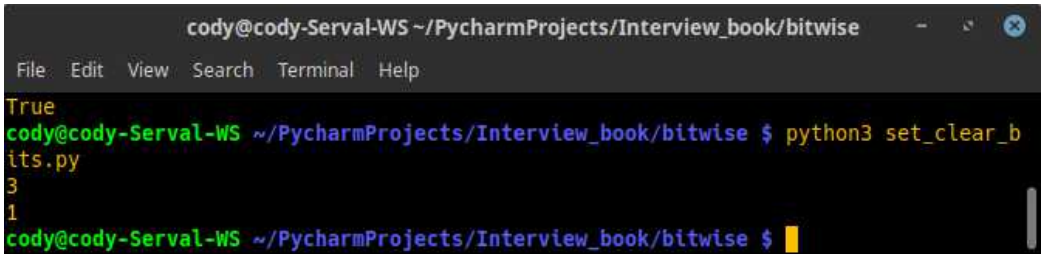

Figure 17.4.3.: Set or clear bits

17.4.4. Counting bits set

The following code counts the number of bits set in the input value. This is based on Brian Kernighan's algorithm, which only performs as many iterations as there are bits set. In other words, if a 32-bit word is provided that only has the highest bit set, it will only loop once.

```
def count_set_bits(word):
    counter = 0
    while counter <= word:
        word &= word - 1
        counter += 1

    return counter

if __name__ == "__main__":
    print(count_set_bits(4))
    print(count_set_bits(20))
    print(count_set_bits(434))
```

When we run this code, we get figure 17.4.4. We can quickly and easily confirm the output by inputting the integers into a programming calculator. In figure 17.4.5, we can see that the integer 4 has been converted to binary in the highlighted box. There is only a single bit set. In figure 17.4.6, the integer 20 has been converted to binary in the highlighted box. Only 2 bits are set. Again, in figure 17.4.7, we can see that integer 434 has been converted into its binary value in the highlighted area. Just as the Python program printed out, there are only 5 bits set.

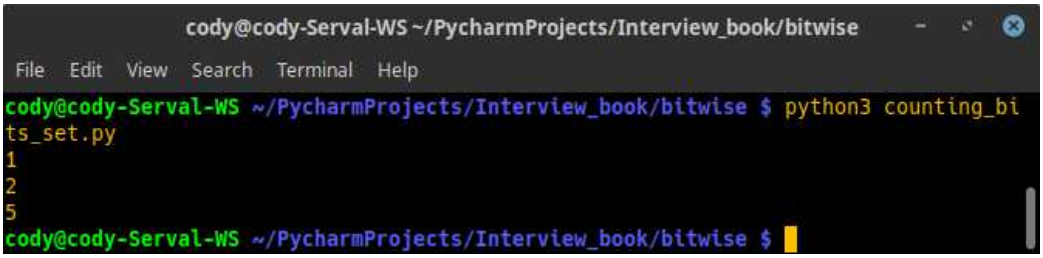

Figure 17.4.4.: Counting set bits

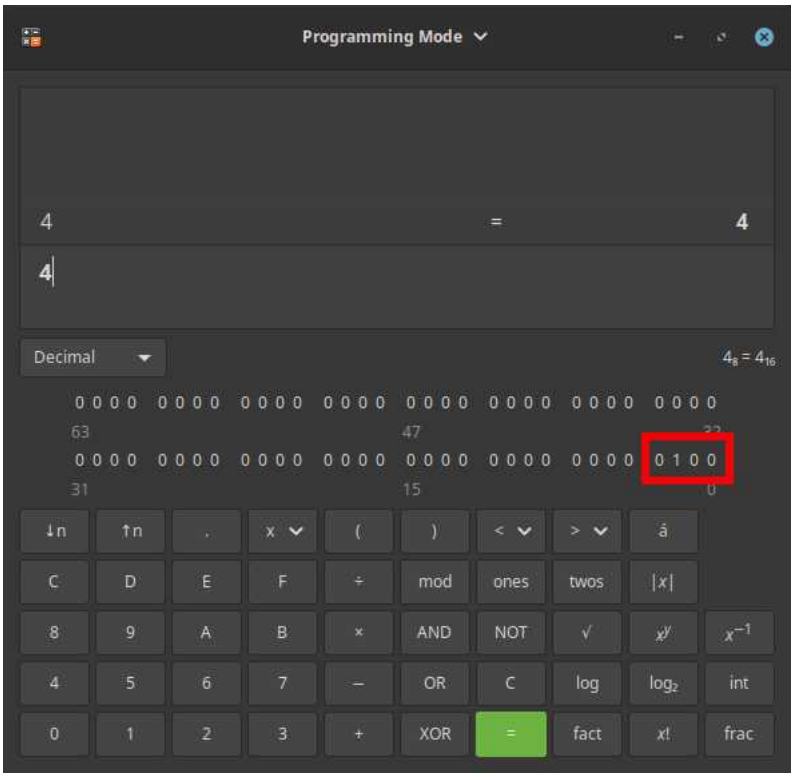

Figure 17.4.5.: Single bit set

17.4.5. Counting parity

A simple check of parity is to see whether the output is *True* or *False*, based on the number of bits set in the input value. If an odd number of bits are set (parity = 1), then we get a *True*. If there is an even number of bits set (parity = 0), then we get *False*.

An example of this code is shown below:

```
def parity(word, parity=False):
    while word:
        parity = not parity
        word = word & (word - 1)

    return parity

if __name__ == "__main__":
    print(parity(4))
    print(parity(1))
```

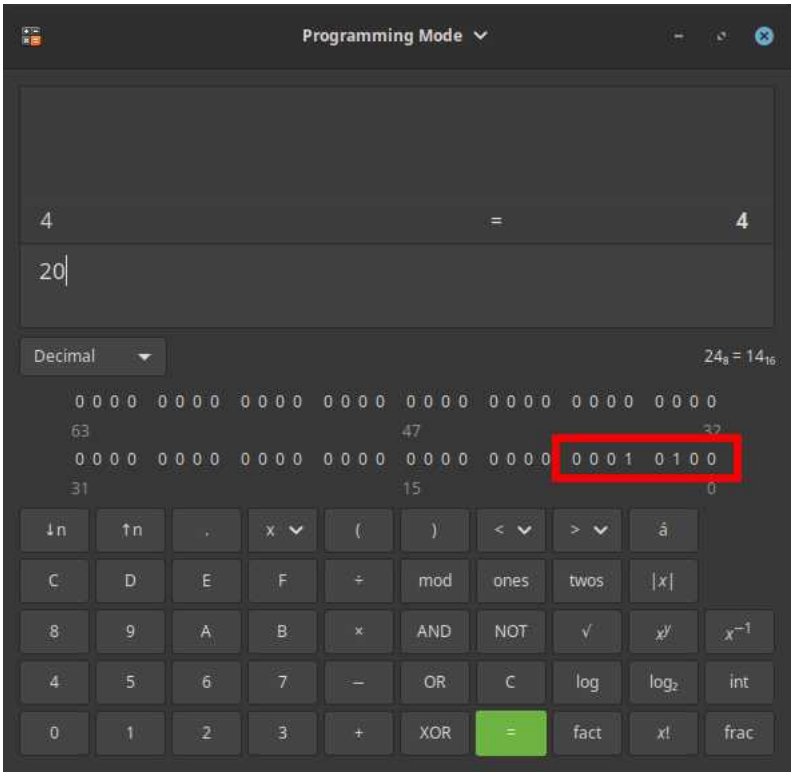

Figure 17.4.6.: Two bits set

```
print(parity(40))
print(parity(343))
```

When ran, we get figure 17.4.8. The parity values (1 or 0) are returned as Boolean values, which simply tells us whether or not the input values have an even or odd number of bits set.

17.5. Summary

In this chapter, we looked at what two's complement is and how it is used when dealing with bit calculations. We then looked at various bitwise manipulation operations available in Python which use two's complement for calculations. We ended by looking at several Python programs, based on C code, that performed bitwise operations.

In the next chapter, we will look at various math and probability exercises using Python code to calculate.

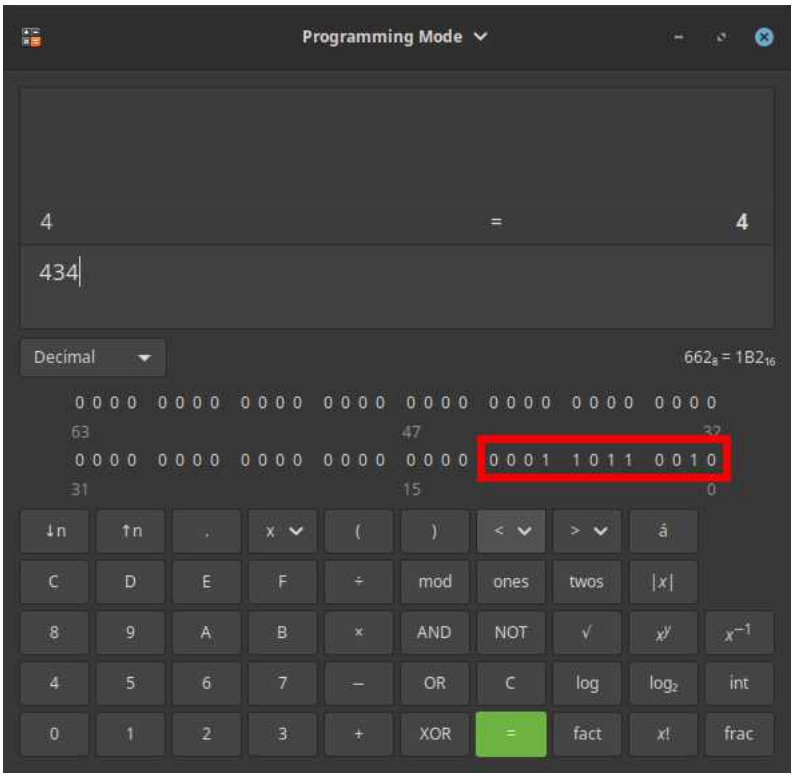

Figure 17.4.7.: Five bits set

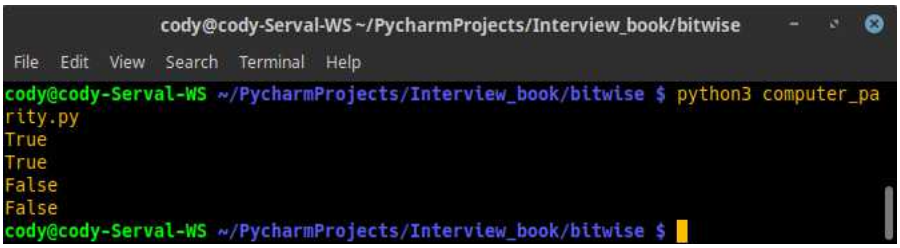

Figure 17.4.8.: Parity output

Part IV.

Mathematics

18. Math and Probability Problems

Math and probability are common problems and tools in programming. When computing first started, a lot of the initial computer creators, software developers, and hardware engineers either came from mathematics or had a strong math background, since computers were originally designed to help solve mathematical calculations.

In this chapter, we will look at the following topics:

- What is Number Theory?
- What is Probability Theory?
- What is Linear Algebra?
- What is Geometry?

18.1. What is Number Theory?

Number theory is a branch of pure mathematics dedicated to the study of integers. I know it sounds boring, but a lot of key elements in STEM fields actually derive from number theory: prime numbers, rational numbers, discrete mathematics, cryptography, quadratic equations, etc. We will cover only a small subset of number theory categories in this section.

18.1.1. What are prime numbers?

The definition of a prime number is a natural number, greater than one, that cannot be created by multiplying two smaller, natural numbers. A natural number is one that is used for counting, such as "6", or ordering, such as "third". Prime numbers are a central part of number theory because every natural number >1 is either prime or can be factored into prime numbers.

18.1.1.1. How to check if a number is prime?

Checking for prime numbers is very easy when using the modulus operation, as shown in the below example[30]:

```
def prime_check(n):
    try:
        if (n <= 1): # Can't check primes on numbers <2
            raise ValueError
        else:
            for i in range(2, n):
                if (n % i == 0):
                    return False
            return True
    except ValueError:
        return "Input can't be less than 2"
```

```
if __name__ == "__main__":
    for i in range(30):
        print(f"Is {i} prime? {prime_check(i)}")
```

The output of this code is figure 18.1.1. As shown, this program alerts the user that only numbers >2 are able to be checked as prime numbers. While the program itself is designed to work over a range of values, it could easily be modified to check only a single value, which is a more common scenario.

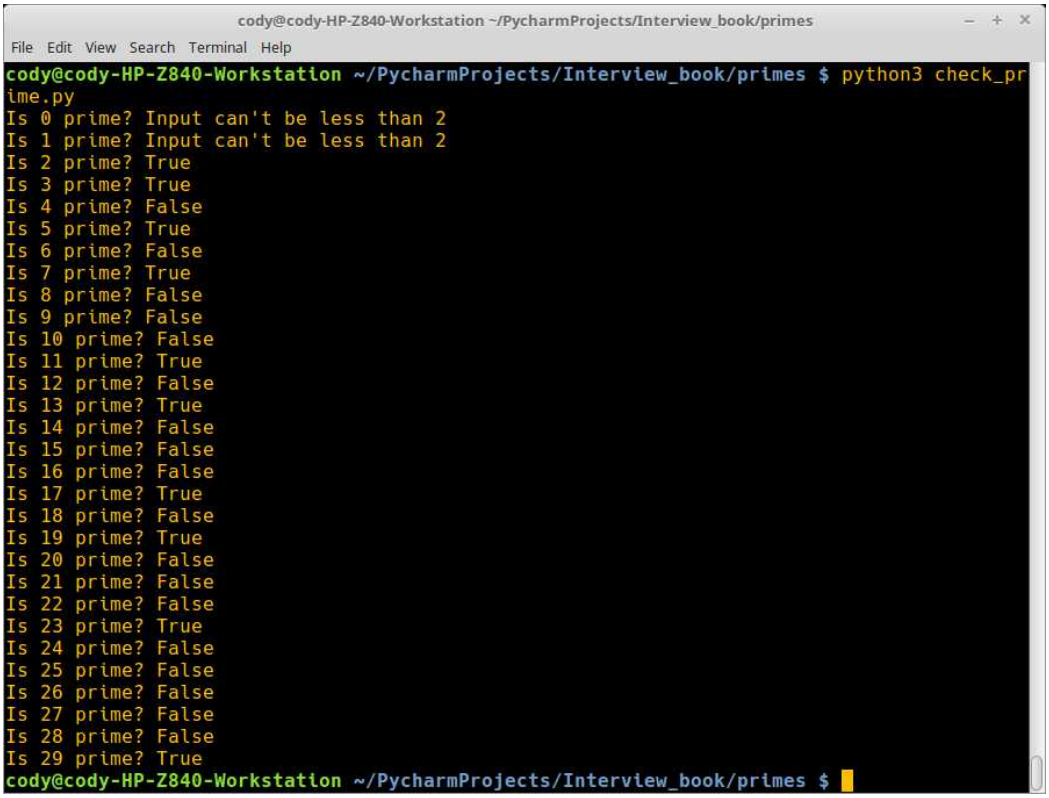

Figure 18.1.1.: Prime number check

18.1.1.2. What is the Sieve of Eratosthenes?

The Sieve of Eratosthenes is an ancient Greek algorithm to find all prime numbers to a given limit. It is best used, however, for values less than 10 million.

The Sieve works by iteratively marking as composite (not prime) the multiples of each prime, starting at 2. The multiples of a given prime are generated as a number sequence starting from that prime, with an arithmetic progression between them equal to the prime.

Restated, starting at 2, every even number after that is marked as composite, since they are divisible by 2. Then, the next prime number (3) is identified and all its multiples starting at 3 (9, 12, 15, etc.) are marked, because they are three numbers from each other. This continues until all unique, prime numbers have been found in the defined range.

The example below shows one way to solve this[31]:

```
def sieve(n):
    prime = [True for i in range(n + 1)] # Create list of
        True values
```

```
    p = 2
    primes = []
    while (p * p <= n): # Start at p^2 for next in series
        if prime[p]: # If value at [p] is True, it's prime
            for i in range(p * p, n + 1, p): # Update all
                multiples of p
                prime[i] = False # Change multiples to
                    False
        p += 1

    for p in range(2, n): # Get primes
        if prime[p]:
            primes.append(p)

    return primes

if __name__ == '__main__':
    n = 30
    print(sieve(n))
```

When ran, we get figure 18.1.2. The complexity for this algorithm is $O(n \log(\log n))$. It is also a common way to benchmark computer performance.

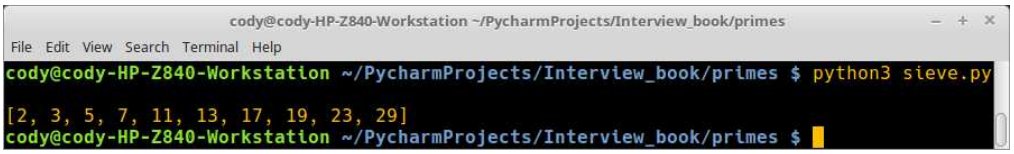

Figure 18.1.2.: Sieve of Eratosthenes

18.1.2. What are ugly numbers?

Ugly numbers are those numbers whose only prime factors are 2, 3, or 5. For example, the first ten ugly numbers are 1, 2, 3, 4, 5, 6, 8, 9, 10, and 12; by convention, 1 is included in the set.

Depending on the field, ugly numbers are also known as regular numbers, 5-smooth, Hamming numbers, or (in music theory) 5-limit pure intonation. You may also see the definition as: those numbers that evenly divide powers of 60, so $602 = 48 \times 75$, meaning 48 and 75 are ugly numbers.

The code below is an example of how to find the n^{th} ugly number, based on the premise: divide the input number by greatest divisible powers of 2, 3 and 5; if result = 1, then it is an ugly number.[32]

```
def find_ugly(input_num):
    i = 1
    ugly_counter = 1

    while input_num > ugly_counter:
        i += 1
        if ugly_check(i):
            ugly_counter += 1
    return i
```

```
def ugly_check(num):
    num = max_divisible(num, 2)
    num = max_divisible(num, 3)
    num = max_divisible(num, 5)
    if num == 1:
        return True

def max_divisible(val, prime):
    while val % prime == 0:
        val = val / prime
    return val

if __name__ == "__main__":
    nums = [7, 10, 150]
    for num in nums:
        print(f"The {num}th ugly number is {find_ugly(num)
            }")
```

The output is figure 18.1.3. A quick check with the sequence in the first paragraph of this section shows that the seventh and tenth values are correct, so it's a safe bet that the 150th value is correct as well.

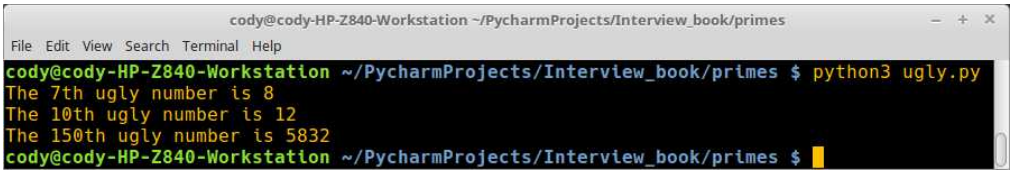

Figure 18.1.3.: Ugly numbers output

18.1.3. What are combinatorics?

Combinatorics is concerned about counting as a means and an end in obtaining results. Graph theory, for example, is one application of combinatorics. It is also used in the analysis of algorithms. There are two basic counting principles used in combinatorics: sum rule and product rule.

- Sum Rule: if there are a ways of completing a task and b ways of completing another task, and they both cannot be completed at the same time, there is a total of $a + b$ ways of to choose an action. A key factor in determining when to use the sum rule is if the question can be rephrased with the word *or*, as shown in the example below.

  For example, Jessica has a business meeting. Among the clothing in her wardrobe, she has 4 skirts and 2 slacks to wear for the meeting. How many different outfits could she create? Since she can wear only one of the skirts or one of the slacks, she had $4 + 2 = 6$ different options.

- Product Rule: if a task can be done in one of a ways and another task in one of b ways, there is a total of $a \times b$ ways of accomplishing both tasks. In this case, the key factor is whether the word *and* could be used in a rephrasing of the question, as demonstrated below.

For example, Jessica went shopping and bought 5 new blouses. Now, when she goes to a business meeting, she can choose only one blouse and either a skirt or slacks. How many different outfits could she create? Since we already know that there are 6 different options for skirts and slacks, we have to account for the blouses. Therefore, she has $5 \times 6 = 30$ different combinations. If we only considered different items, e.g. a = blouses, b = skirts, and c = shoes, then the calculation would be $a \times b \times c$.

An additional concept in combinatorics is the principle of inclusion-exclusion. This principle states that, in order to count only the unique ways of completing a task, we must count the ways to do it in one way, count the ways to do it another way, then subtract the ways that are common between them. Due to the subtraction process, this is also known as the subtraction principle.

More formally, the principle is mathematically expressed as $|A \bigcup B| = |A| + |B| - |A \bigcap B|$, and stated as the union of A and B is equal to A plus B minus the intersection of A and B.

18.1.3.1. Python's combinatoric tools

Python actually has several combinatoric functions built-in. The *itertools* module implements a number of iterator building blocks, based on constructs from other languages. The combinatoric iterators are:

- *product()*: returns the Cartesian product of input iterables
- *permutations()*: returns successive permutations of elements from an iterable
- *combinations()*: returns the subsequences of elements from an iterable
- *combinations_with_replacement()*: returns the subsequences of elements from an iterable while allowing individual elements to be repeated more than once

The example below (from the Python documentation) shows how the *product()* function can be used:

```
import random
from itertools import product

for val in product("ABCD", "xyz"):
    print(val)

for num in product(range(2), repeat=3):
    print(num)

def random_product(*args, repeat=1):
    "Random selection from itertools.product(*args, **kwds
        )"
    pools = [tuple(pool) for pool in args] * repeat
    return tuple(random.choice(pool) for pool in pools)

print(random_product("ABCD", "xyz"))
```

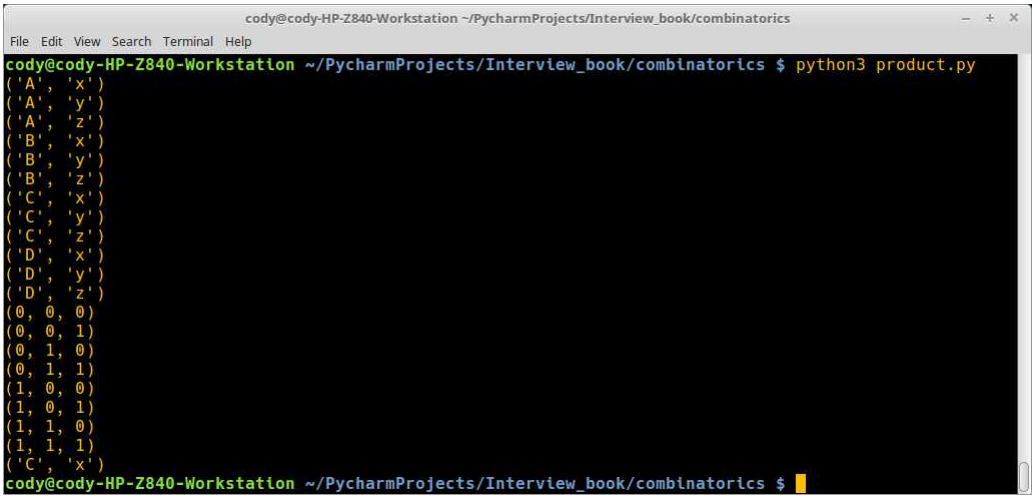

Figure 18.1.4.: Product combinator

The output of this is shown in figure 18.1.4. Note that your results from *random_product()* may be different.

In the output above, the first 12 entries are simply all the combinations of the input arguments, successively iterated over. The next eight entries are the products of two numbers (0 and 1) in a three-value array; note that the output is essentially binary counting. Finally, the last line shows a randomly selected product from the input arguments.

Also from the Python documentation, the code below shows examples of *permutation()*:

```
import random
from itertools import permutations

for chars in permutations('ABCD', 2):
    print(chars)

for val in permutations(range(3)):
    print(val)

def random_permutation(iterable, r=None):
    "Random selection from itertools.permutations(iterable
        , r)"
    pool = tuple(iterable)
    r = len(pool) if r is None else r
    return tuple(random.sample(pool, r))

print(random_permutation('ABCD', 2))
```

The output is shown figure 18.1.5: Similar to *product()*, the output shows all the possible combinations that can occur with a given input. The final line has only two values returned, as the *random_permutation()* function accepts an argument to limit the length of the permutation. If not provided, it would have provided a random permutation of the entire input argument.

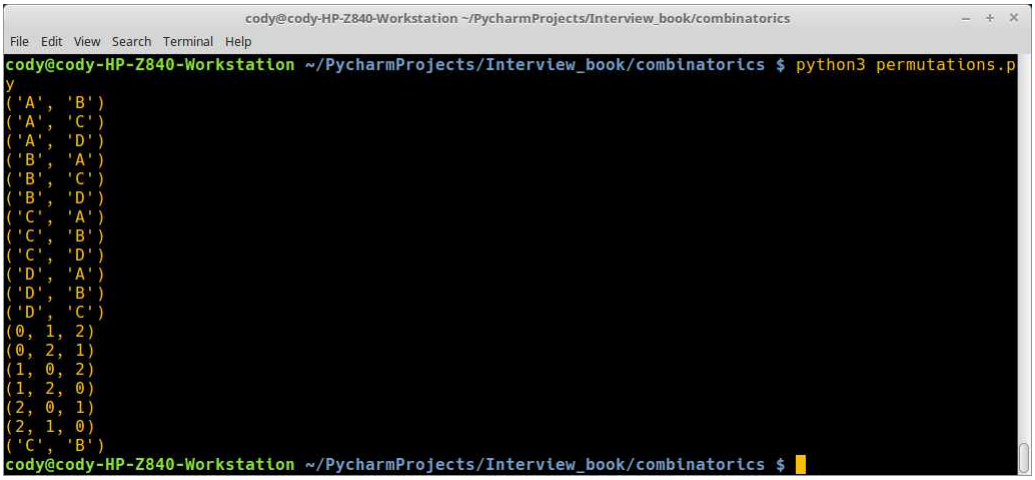

Figure 18.1.5.: Permutations combinator

18.2. What is Probability Theory?

Probability theory is the part of math that looks at probability, treating the concept in a rigorous mathematical manner via axioms. The axioms typically define probability as a probability measure between 0 and 1, within the probability space of options. The probability measure is assigned to a set of outcomes deemed the sample space. A specific subset of these outcomes is an event. In short, probability is the possibility of an event occurring.

18.2.1. Definitions

To ensure everyone is on the same page before we delve deeper into probability, we will define a number of basic probability terms below.

- *Probability*: If there are p possible outcomes and q of them are favorable to event A, the probability of event A occurring is denoted at $P(A)$ and determined by the formula $P(A) = \frac{q}{p}$.
- *Random variable*: A measurable function on a probability space. The distribution function defines the probability if different results.
- *Sample space*: The sample space (s) is the set of all possible outcomes of a random event. In the case of flipping a coin, the sample space is either heads or tails.
- *Event*: This is the subset of the sample space to which a probability can be assigned.
- *Elementary event*: Also called an atomic event or sample point, an elementary event is an event that has only one outcome. In a deck of cards, picking the eight of hearts is an elementary event, while picking any heart card is not.
- *Sure event*: An event that is guaranteed to occur; the probability = 1.
- *Impossible event*: An event that is guaranteed to never occur; the probability = 0.

- *Compound event*: An event that is the disjoint union of two or more elementary events.
- *Mutually exclusive events*: Also called complementary events, mutual exclusion is two or more events where, if one event occurs, the other(s) cannot occur. In other words, no two or more events can occur at the same time.
- *Exhaustive events*: The union of two or more events is the entire sample space when combined.

18.2.2. Theorems

Probability theory has a number of theorems associated with it. We will cover the most commonly used ones below.

18.2.2.1. Complementary Events

The probability of a complementary event A' for event A is given as $P(A') = 1 - P(A)$. This should be obvious, as probability is measured between 0 and 1. Therefore, if $A = 0.2$, then the probability of A' is$1 - 0.2 = 0.8$.

In formal terms, the events A and A' are mutually disjoint and together they form the whole sample space.

$$A \bigcup A' = S \Rightarrow P\left(A \bigcup A'\right) = P\left(S\right) \text{ or } P\left(A\right) + P\left(A'\right) = P\left(S\right) = 1 \Rightarrow P\left(A'\right) = 1 - P\left(A\right)$$

18.2.2.2. Impossible Events

The probability of an impossible event is 0. Again, this should be self-evident: if an event can never occur, and probability is a value between 0 and 1, then an impossible event must have a probability of 0.

In formal terms, let A be an impossible event and S be the sure event. $S = A'$ and $A = \Phi$.

$$P\left(\Phi\right) = P\left(A\right) = P\left(S'\right) = 1 - P\left(S\right) = 1 - 1 = 0$$

18.2.2.3. Subset Events

If A is a subset of B, then the probability of A occurring is less than or equal to the probability of B occurring. Yet another fairly obvious theorem, if A is a subset of B, meaning it is a component of B, then it can never be larger than B. Otherwise, B would be a subset of A or they would be independent items. This relationship is shown in figure 18.2.1.

This diagram can be described as A is a subset of B ($A \subseteq B$) or B is a superset of A ($B \supseteq A$). If there was a discrepancy between the two, in other words there was at least one element of B that is not part of A, then you could say A is a strict (or proper) subset of B ($A \subsetneq B$), or B is a strict (or proper) superset of A ($B \supsetneq A$).

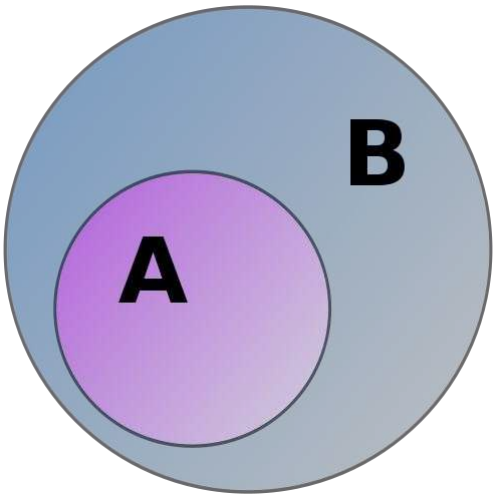

Figure 18.2.1.: Subset diagram

In formal terms, if A is subset B, then

$$A = B \bigcup \left(A \bigcap B'\right) \therefore P(A) = P\left[B \bigcup \left(A \bigcap B'\right)\right] =$$
$$P(A) = P(B) + P\left(A \bigcap B'\right) \therefore P(A) - P(B)$$

When A subset B, A and B $\bigcap$ A' are mutually exclusive events. Therefore,

$$P\left(A \bigcap B'\right) \geq 0 \therefore P(A) - P(B) \geq 0 \therefore P(B) \leq P(A)$$

18.2.2.4. Addition Theorem

If A and B are any two events and are not disjoint, then $P(A \bigcup B) = P(A) + P(B) - P(A \bigcap B)$. A and B are the subsets of sample space S. In figure 18.2.2, $A \bigcup B = A \bigcup (A' \bigcap B)$. In addition, A and $A' \bigcap B$ are mutually disjoint.

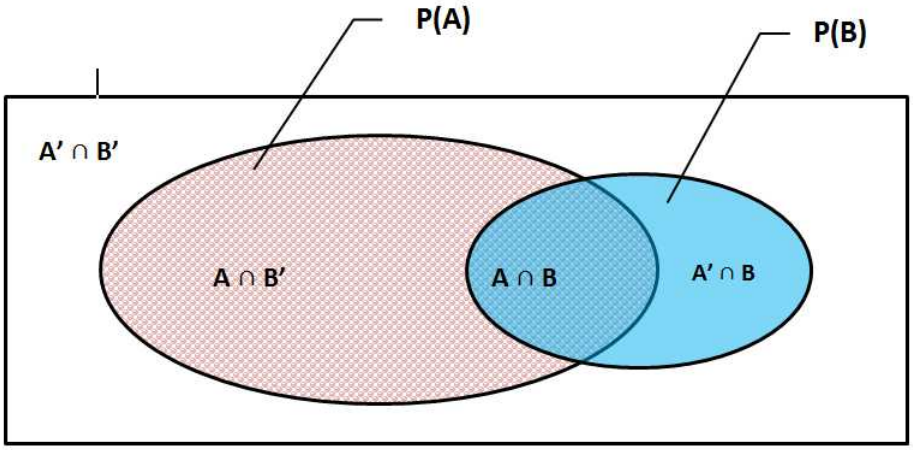

Figure 18.2.2.: Addition theorem graph (Andrew Batishchev [CC BY-SA 3.0])

From this, we know that $A' \bigcap B = B - (A \bigcap B)$ and that $A \bigcap B' = A - (A \bigcap B)$. Therefore,

$$P\left(A \bigcup B\right) = P\left[A \bigcup \left(A' \bigcap B\right)\right] = P(A) + P\left(A' \bigcap B\right) = P(A) + P(B) - P\left(A \bigcap B\right)$$

In plain speak, the sum of the probabilities of A and B is equal to probability of A plus the probability of B, minus the portion where A and B overlap.

18.2.2.5. Multiplication Theorem

If A and B are any two events such that $P(A) \neq 0$ and $P(B) \neq 0$ and if A and B are independent events, then $P(A \bigcap B) = P(B) \times P(A \mid B)$. $P(A \mid B)$ represents the conditional probability of occurrence of A when the event B had already happened.

In plain speak, the product of probabilities of A and B is equal to the probability of B occurring, times the probability of A occurring after B event has already occurred. You could also have $P(A \bigcap B) = P(A) \times P(B \mid A)$, with the same conditional probability of occurrence.

An example may make this clearer: an opaque container has 5 red, 7 blue, and 6 yellow balls. If three balls are pulled from the container, without replacement, what is the probability that all three balls are red?

The total number of balls is 18, and each time one is removed the total is decreased by one. Therefore,

$$P(B1) \times P(B2 \mid B1) \times P\left(B3 \mid B1 \bigcap B2\right) = \frac{5}{18} \times \frac{4}{17} \times \frac{3}{16} = \frac{5}{408} = 1.23\%$$

18.2.2.6. Law of Large Numbers

The law of large numbers (LLN) is a theorem that states that the average value of the sum of all results should be close to the expected value, and will tend to get closer to the expected value the more values are added. It is important to note that this only applies to a large number of observations; a small number cannot guarantee that the observations will be close the expected result or that a large quantity of one value will be balanced out by other values.

There is no set value to what a "large number" of observations is. This theorem simply implies that, the more values you have, the more likely you are to reach the optimum average. This is demonstrated by figure 18.2.3:

In this example, observations <200 show a relatively wide range of fluctuations, with no results at the theoretical average. At 400 observations, we have a closer measured mean vs. theoretical mean. By the time we have 700 or so measures, we are effectively at the theoretical mean.

Note that the graph above is showing the average values, not the actual dice rolls. The actual rolls on a graph would approach a bell curve, discussed below.

18.2.2.7. Central Limit Theorem

A prime tenant of mathematics, especially probability and statistics, the central limit theorem (CLT) explains the ubiquitous nature of the normal distribution (bell curve). The CLT states that the average value of randomly distributed,

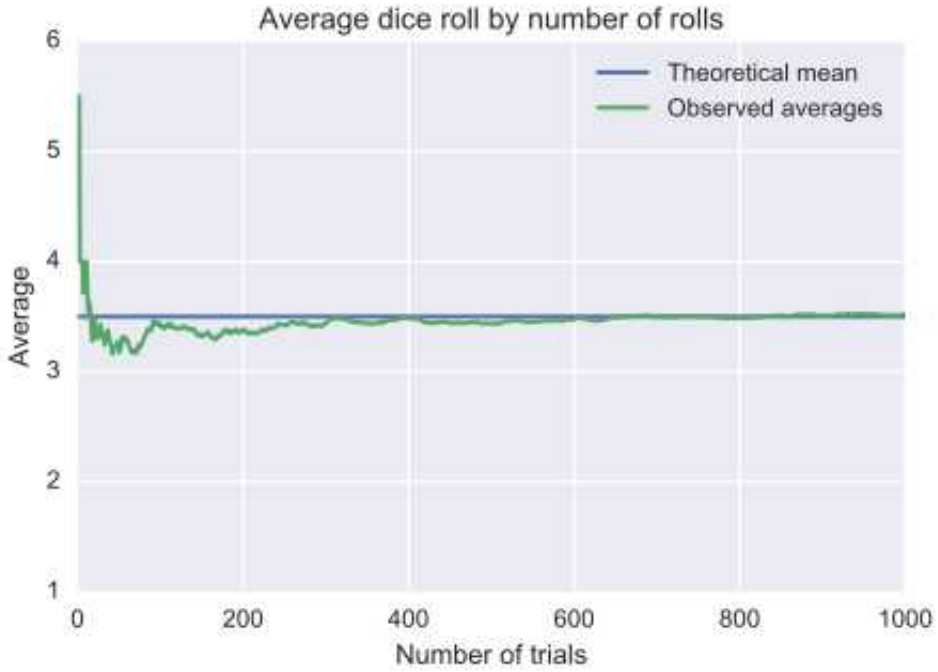

Figure 18.2.3.: Tending towards average

independent variables will tend towards a normal distribution, regardless of the distribution of the original, random variables.

In other words, as more variable values are added to a graph, the graph will tend to look like a bell curve. The average value will be the peak, with the two tails spreading out as the odds of a particular result get smaller as we move further from the mean. An example of this is shown figure 18.2.4:

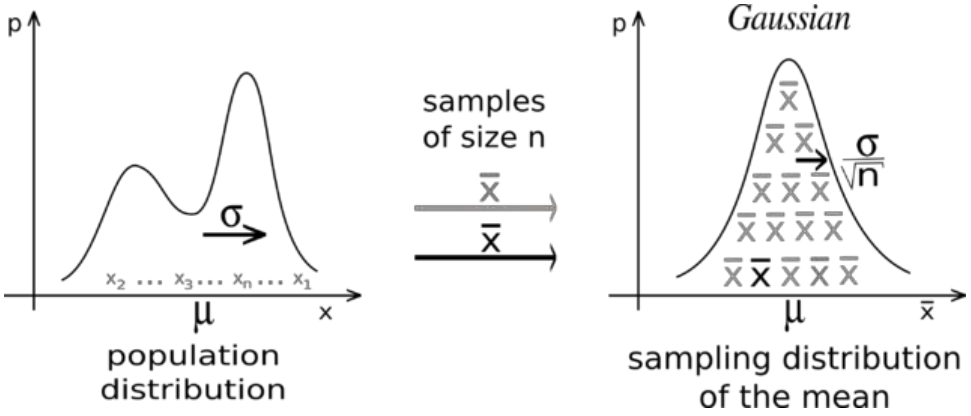

Figure 18.2.4.: Distribution curves

The entire population graph is not normal, but a random sampling of the population will tend to become normal as more samples are added to the graph results.

18.3. What is Linear Algebra?

Linear algebra is concerned with linear equations, linear functions, and their representations within matrices and vector spaces. It is a fundamental part of nearly all mathematics, such as geometry, as well as science and engineering, where it helps model natural phenomena or as first-order approximations for subsequent, non-linear models.

For programmers, linear algebra is almost a requirement to understand certain concepts and develop new ones. Linear algebra is commonly used in computer animation, 3D graphics, artificial intelligence, financial calculations, etc.

Even search engines utilize linear algebra as vector space models; for example, each document is represented as a vector in space, and the relationship between vectors indicates how good the match is. Google's PageRank algorithm ranks web pages according to an eigenvector of a weighted link matrix.

18.4. How Can Python Solve Linear Equations?

While NumPy and pandas are best suited for advanced math, vanilla Python can also be used. A Python list is the best representation of a vector and a matrix could be represented as a list of lists, for example:

```
matrix = [[1.0, 2.0], [3.0, 4.0]]
```

could represent the matrix

```
[1.0  2.0]
[         ]
[3.0  4.0]
```

The code example below demonstrates using list comprehension to create a variety of vectors and matrices and some associated operations.

```
def vector_format(vector):
    """Return a single vector list"""
    return f"{[x for x in vector]}"

def matrix_format(matrix):
    """Return a matrix of lists"""
    return "\n".join([vector_format(x) for x in matrix])

vect = [-1.0, 3.0]
num1 = [[1.0, 2.0],
        [3.0, 4.0]]
num2 = [[3.0, -2.0],
        [2.0, 1.0]]
num3 = [[1.0, 1.5, -2.0],
        [2.0, 1.0, -1.0],
        [3.0, -1.0, 2.0]]

print([[num1[i][j] + num2[i][j] for j in range(len(num2
    [0]))] for i in range(len(num1))]): # the matrix num1+
    num2
print("*" * 4)
```

```
print(sum(vect[i] * vect[i] for i in range(len(vect)))) #
    the dot product vect.vect
print("*" * 4)
print([sum(num1[i][j] * vect[j] for j in range(len(vect)))
     for i in range(len(num1))]) # the vector num1*vect
print("*" * 4)
print([[sum(num1[i][k] * num2[k][j] for k in range(len(
    num2))) for j in range(len(num2[0]))] for i in range(
    len(num1))]) # the matrix num1*num2
print("*" * 4)
print((vector_format(vect)))
print("*" * 4)
print((matrix_format(num2)))
```

When ran, we get figure 18.4.1. You can see that the default output is a one-line list for most items, except for the last element when we expressly called the formatting function *matrix_format()*.

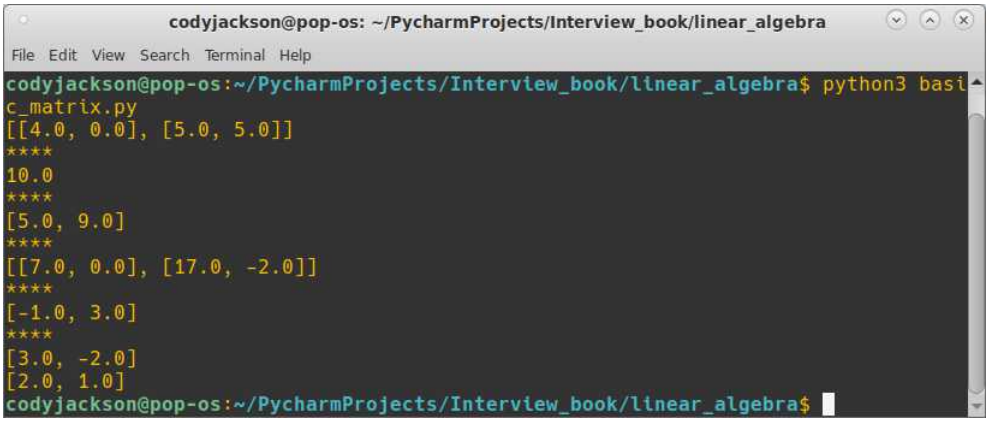

Figure 18.4.1.: Basic matrix

18.5. What is Geometry?

Geometry is math that deals with shapes, sizes, relative position, and properties in space. Obviously, there is a lot of application to computers, whether from video games, modeling and simulations, or calculating a space shot.

One thing to note about computational geometry is that floating point numbers have a limit to their precision. In addition, if a number can't be created from exact powers of two cannot be represented exactly as a floating point number.

For example, Python automatically truncates unnecessary values from a floating point result. But, if you add floating point numbers together, you may see the true value, as shown figure 18.5.1:

In this case, Python showed us the most useful representation of 0.1 without showing the actual floating point value. But when we added 0.1 three times, we see that the final value is not 0.3, as expected, but a nearly immeasurable amount off.

Naturally, if you are dealing with a lot of floating point numbers, those little differences add up. That is one of the base points about chaos theory: because we can never precisely measure something, small differences can lead to drastic

Figure 18.5.1.: Rounding error

changes later. This was discovered when trying to forecast the weather; when numbers with seemingly insignificant rounding were use, the imprecision of floating point numbers meant that the input values were never truly the same, leading to vastly differing results each time.

This is a subject all its own, so we will cover only a select number of geometric problems for demonstration purposes.

18.5.1. Does a line touch or intersect a circle?

Given the center coordinate of a circle and the equation of a line, and assuming the circle radius is > 1, how do you determine where the line is in relation to the circle?

The idea is to compare the perpendicular distance between the circle's center and the line using the circle's radius. The algorithm would be:

1. Find the perpendicular distance (d) between the center and the line.
2. Compare the distance with the radius (r)
   a) If $d > r$, the line is outside the circle
   b) If $d = r$, the line touches the circle
   c) If $d < r$, the line is intersects the circle

The distance of a line from a point is calculated with the formula $d = \frac{|ax_0+by_0+c|}{\sqrt{a^2+b^2}}$, where the equation for a line is $ax+by+c = 0$, where a, b and c are real constants with a and b not both zero, and the point is (x_0, y_0).

The example program below shows one way of solving this problem.[33]

```
from math import sqrt

def check_position(a, b, c, x, y, radius):
    dist = ((abs(a * x + b * y + c)) / sqrt(a * a + b * b)
        ) # Distance from line to center
    if radius == dist:
        return "Touches circle"
    elif radius > dist:
        return "Intersects circle"
    else:
        return "Outside circle"
if __name__ == "__main__":
    radius = 5
    x = 0
```

```
y = 0
a = 3
b = 4
c = 25
print(check_position(a, b, c, x, y, radius))

radius = 2
x = 0
y = 0
a = 3
b = 4
c = 25
print(check_position(a, b, c, x, y, radius))
radius = 7
x = 0
y = 0
a = 3
b = 4
c = 25
print(check_position(a, b, c, x, y, radius))
```

The output is shown figure 18.5.2. You'll notice that, for the input arguments, all we did was change the radius of the circle. Naturally, most problems won't be like this, and it's frequently good to have a visual representation for this. For more complex problems, or visualization, there are a number of Python libraries that are better equipped to handle mathematical calculations and displaying the results. However, we won't be covering those in this book.

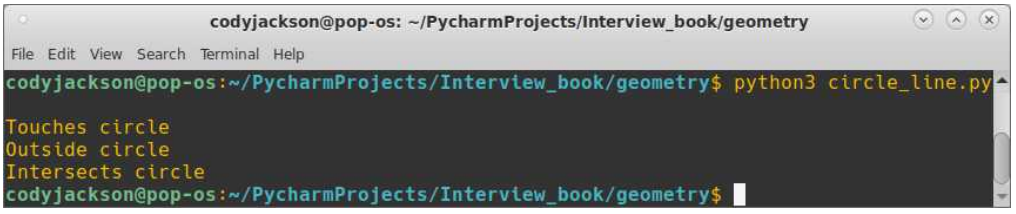

Figure 18.5.2.: Line intersection with circle

18.5.2. Does a point lie within a triangle?

Given the corner points of a triangle, how do you determine whether another given point is inside or outside of the triangle?

Assuming the corner points are $A = (x_1, y_1)$, $B = (x_2, y_2)$, and $C = (x_3, y_3)$ and the given point P is *(x, y)*, the algorithm is as follows:

1. Calculate the area (A) of the given triangle. Mathematically, this is $A = \frac{1}{2bh}$, where b is the length of the base and h is the height of the triangle. To make life easy as a programmer, you can also use the following formula: $A = \frac{x_1(y_2-y_3)+x_2(y_3-y_1)+x_3(y_1-y_2)}{2}$.
2. Next, calculate the area of P_{AB} as A_1.
3. Calculate the are of P_{BC} as A_2.
4. Finally, calculate the are of P_{AC} as A_3.

5. If P is inside the triangle, then $A_1 + A_2 + A_3 = A$.

The code example below shows one way of solving this problem[34]:

```
def area(x1, y1, x2, y2, x3, y3):
    """Area formula equates to 1/2 * base * height"""
    return abs((x1 * (y2 - y3) + x2 * (y3 - y1) + x3 * (y1
        - y2)) / 2.0)

def check_inside(x1, y1, x2, y2, x3, y3, x, y):
    """Determine if point (x, y) is inside the triangle"""
    total_area = area(x1, y1, x2, y2, x3, y3)
    area_1 = area(x, y, x2, y2, x3, y3)
    area_2 = area(x1, y1, x, y, x3, y3)
    area_3 = area(x1, y1, x2, y2, x, y)
    if total_area == area_1 + area_2 + area_3:
        return True
    else:
        return False

# A(0, 0), B(20, 0) and C(10, 30)
if __name__ == "__main__":
    if check_inside(0, 0, 20, 0, 10, 30, 10, 15):
        print('Inside')
    else:
        print('Outside')
```

When we run the code, we get figure 18.5.3. Pretty anti-climatic, but then not everything in programming has to be flashy. Obviously, you could modify the code to print all the points, and even graphically show how they relate to one another, but if you just need something to get the job done, this is sufficient.

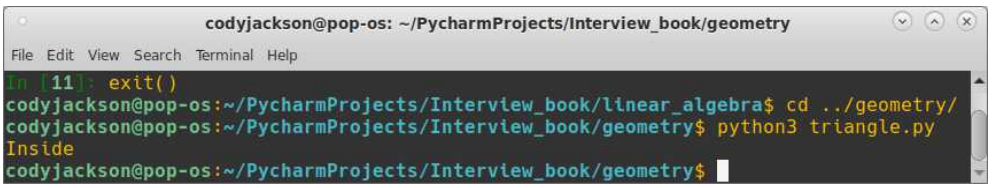

Figure 18.5.3.: Point in triangle

18.5.3. What is a sweep line?

A key technique in computational geometry is a sweep line. A sweep line is an imaginary line (commonly vertical) that moves (sweeps) across a plane, stopping at various points. Geometric operations are restricted to geometric objects that either intersect with, or are close to, the sweep line when it stops. Once the sweep line has moved across the plane and interacted with all objects, the final solution is derived.

A common sweep line problem is referred to as the "skyline problem", often stated as:

> Given n rectangular buildings in a 2-dimensional city, compute the skyline of these buildings, eliminating hidden lines. The main task

is to view buildings from a side and remove all sections that are not visible. All buildings share common bottom and every building is specified by an ordered triple (L_i, H_i, R_i) where L_i and R_i are left and right coordinates, respectively, of building i and H_i is the height of the building. The output is given as (L, H) and (R, H), where (L, H) is the left-hand x-coordinate and its height, and (R, H) is the right-hand coordinate and its height.

For example, a single building with (L, H, R) of (1, 11, 5) would have an output of (1, 11), (5, 0). Three buildings with ordered triples of (1, 11, 5), (2, 6, 7), and (3, 13, 9) would have an output of (1, 11), (3, 13), (9, 0). This is shown figure 18.5.4.

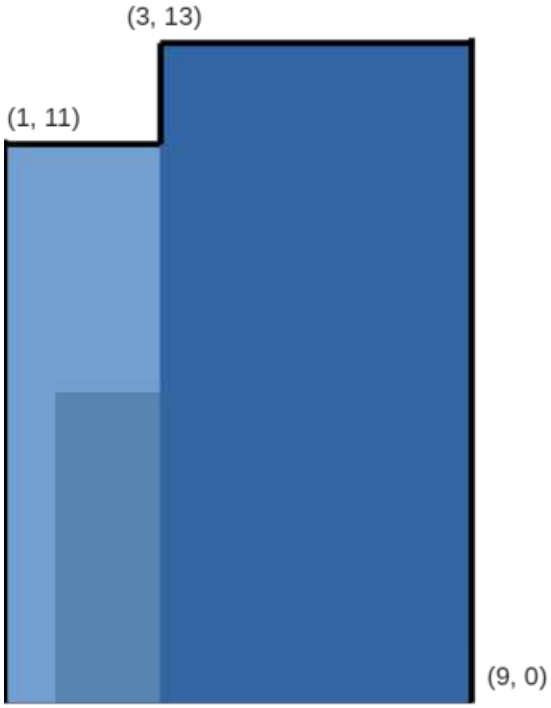

Figure 18.5.4.: Skyscraper problem

In the picture above, we can see the different buildings as different colored rectangles. As we move to the right, the previous building is covered up. The black line outlines what the skyline would look like if the building rectangles were removed. Because building 2 is smaller than the others, it doesn't have an effect on the skyline, leaving only buildings 1 and 3 to create the skyline.

One way of solving this problem is shown below, utilizing heaps. Another common way to solve this type of problem is via divide-and-conquer.[35]

```
from collections import defaultdict, namedtuple
from heapq import heappop, heappush

class Building:
    def __init__(self, height):
```

```
        self.height = height
        self.finished = False

    def __lt__(self, other):
        """Python uses min-heaps so, if we reverse order
           by height, when we store building_coordinates
           in a heap, the first building is the highest.
           """
        return other.height < self.height

Structure = namedtuple('Structure', 'start_end')  #
    Represents the buildings that start and end at a
    particular x-coordinate.

def skyline(building_coordinates):
    """Given an iterable of building_coordinates
       represented as triples (left, height, right),
       generate the coordinates of the skyline."""
    structures = defaultdict(lambda: Structure(start=[],
       end=[]))  # Map from x-coordinates to structure.
       defaultdict ensures that no errors are generated
       for keys that don't yet exist; a new Structure will
        simply be added to the dictionary.
    for left, height, right in building_coordinates:
        building = Building(height)
        structures[left].start.append(building)  # The
           left x-coordinate is added to the start entry
           for the building; if a new building has the
           same x-coordinate as the previous, the tallest
           height is used
        structures[right].end.append(building)  # The
           right x-coordinate is added to the end entry
           for the building

    placed = []  # Heap of building_coordinates currently
       placed.
    last_height = 0  # Last generated skyline height.

    for x, structure in sorted(structures.items()):  #
       Process structures in order by x-coordinate
        for building in structure.start:  # Update
           building_coordinates currently placed
            heappush(placed, building)  # Add building to
               heap
        for building in structure.end:  # If building not
           overlapped, then it is finished
            building.finished = True

        while placed and placed[0].finished:  # Pop any
           finished building_coordinates from the top of
           the heap.
```

```
            heappop(placed)

        if placed:  # Top of heap is the highest placed
            building, so its height is the current height
            of the skyline
            skyline_height = placed[0].height
        else:
            skyline_height = 0

        if skyline_height != last_height:  # Yield co-
            ordinates if the skyline height has changed.
            yield x, skyline_height
            last_height = skyline_height

if __name__ == "__main__":
    print(f"Directly overlapping buildings: {list(skyline
        ([(1, 3, 2), (1, 3, 2)]))}")
    print(f"Offset overlapping buildings: {list(skyline
        ([(1, 11, 7), (2, 9, 13), (3, 13, 15)]))}")
    print(f"Normal skyline: {list(skyline([(1, 9, 3), (1,
        11, 5), (2, 6, 7), (3, 13, 9), (12, 7, 16), (14, 3,
        25), (19, 18, 22), (23, 13, 29), (24, 4, 28)]))}")
```

The output of this code is shown in figure 18.5.5.

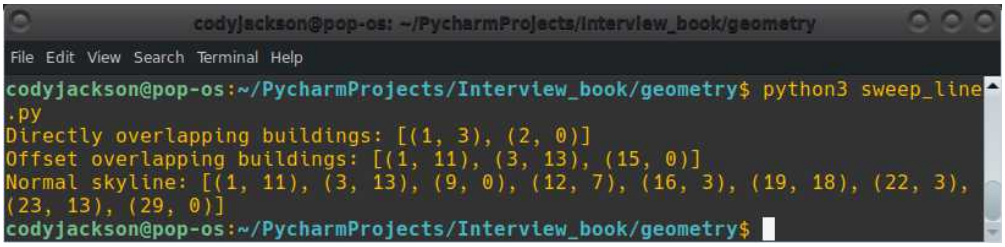

Figure 18.5.5.: Skyline sweep line output

I'll admit, this program isn't particularly intuitive, even with all the comments I added. Essentially, the program is looking at the triple for each building, figuring out where the left-side x-coordinate is as well as the building height, then comparing the right-hand x-coordinate to the next building's triple. If the next building's left-side value is less than or equal to the previous building's right-side value, then they overlap.

At that point, the program has to determine whether the next building is taller than the previous. If so, then it updates the height to the next building's height; otherwise it keeps the previous building's height. It also notes when the height changes, i.e. the building's left-side x-coordinate that creates the current height.

In addition, it has to determine when the previous building ends. If the next building is not taller, but the previous building ends, then the new height is equal to the now-lower height of the next building.

If we go back to the example drawing previously, if building 2 had an right-side x-coordinate > 9, such as 12, then the result would have been (1, 11), (3, 13), (9, 6), (12, 0).

18.6. Summary

In this chapter, we covered a number of mathematical issues that commonly arise in programming. Number theory is used nearly everywhere in mathematics, as is linear algebra. Probability theory is commonly used in statistics and probability, naturally enough. And geometry is more common than you might think if you consider both mathematical visualization, video games, and GUI development. Sweep lines are prevalent in a number of areas and can be programmed using a number of different ways, such as heaps, divide-and-conquer, or other algorithms.

That's it for this chapter, and that's it for this book. Apart from the information in the appendices, you now have an overview of the material commonly asked in programming job interviews. I hope you found this book worthwhile and I hope it serves you well in the future.

Part V.

Appendix

A. Common Interview Questions

Obviously, questions during an interview are the whole reason interviews exist. Employers want to know who can, and cannot, do the job. While there are a lot of steps to go through prior to getting to the interview, each one designed to weed out potential candidates, once you get to the interview you have to be prepared.

This appendix will cover two, broad topics:

- What are some traditional interview questions?
- What are some technical interview questions?

A.1. What are some traditional interview questions?

This section will provide a number of different questions that one can expect from nearly any interview. Some are open-ended, allowing the candidate to freely provide information. Others are a little more focused and generally require some specific information.

- Can you tell me a little about yourself?

  If you have heard of an elevator pitch, this is a good place to use it. In essence, an elevator pitch is something you would tell someone in an elevator as you are traveling to your floor. While it is normally used for pitching a proposal, it applies in this situation because you are proposing yourself as the ideal candidate.

  Start off with the 2-3 specific accomplishments or experiences that you most want the interviewer to know about, then wrap up talking about how that prior experience has positioned you for this specific role.

- How did you hear about the position?

  Pretty up-front question. If you found out about it through a friend or professional contact, name drop that person, then share why you were so excited about it. If you discovered the company through an event or article, share that. Even if you found the listing through a random job board, share why it caught your eye.

- What do you know about the company?

  You better have taken some time to read the company's "About Us" page. However, you need to be able to explain how you take the company's goals and mission to a personal level, along the lines of "I really believe in this approach because..." and share a personal example or two.

- Why do you want this job?

  Of course, you need the job to pay the bills. Everyone gets that. But what about this particular job offer makes it interesting enough to apply for? Identify a couple of key factors that make the role a great fit for you, then

share why you approached the company for this work, rather than someone else.

- Why should we hire you?

  In this answer, you need to answer three things: you can deliver great result, you're a good fit with the team and culture, and that you'd be a better hire than any of the other candidates. Be prepared for follow up questions to anything you say.

- What are your greatest strengths?

  Be truthful about your answers and ensure that the answers are relevant and specific to the job. After the initial response, provide an example of how you applied your strengths at work.

- What do you consider to be your weaknesses?

  Again, be honest but not so much that you answer yourself out of a job. Identify relevant weaknesses (a common one is public speaking) but how you are working to overcome that weakness (such as participating in Toast-Masters International or volunteering to run events to gain more practice).

- What is your greatest professional achievement?

  The main thing with this is to explain what you did, how you did it, and what the results were.

- Tell me about a challenge or conflict you've faced at work, and how you dealt with it.

  Answer this in the same manner as the "greatest achievement" question.

- Where do you see yourself in five years?

  A hiring manager wants to know a) if you've set realistic expectations for your career, b) if you have personal and professional goals, and c) if the position you're applying for aligns with your goals and growth. Think realistically about where this position could take you and answer along those lines.

- What's your dream job?

  Much like the previous question, identify what your dream job is in relation to the job your applying for. If you're interested in developing a new AI paradigm, you better have a good reason for applying for a sales position, for example.

- What other companies are you interviewing with?

  You don't have to provide actual names, but you should provide an answer. Something along the lines of "I'm applying at several companies that focus in web site and embedded system development for the automotive industry, as I hope to move into UI/UX applications for in-car infotainment systems."

- Why are you leaving your current job?

  Probably the best answer for this is "I felt it was time to move on." It doesn't indicate you left on bad terms, you were fired, or anything bad. It simply indicates that you had achieved what you wanted at your last job and felt it is time to do something new. Even if you're going to be doing the

exact same work, a new company offers new things to learn, from company culture to tool sets and projects.

- Why were you fired?

  If you do mention you were let go, you don't need to provide an elaborate explanation. Frequently, it's not your fault, especially when the economy is in a down-turn or the company merged and your position was made redundant. Try to frame it as an opportunity to move into something new.

- What are you looking for in a new position?

  Review the job posting for the position you're applying for. That should give you the answers you should be looking for.

- What type of work environment do you prefer?

  See the question above.

- What's your management style?

  Even if you aren't applying for a management position, you may be placed in charge of others at some point. The best managers are strong but flexible, and that's exactly what you want to show off in your answer.

- What's a time you exercised leadership?

  First, consider that leadership and management are two different things and know the difference. Then provide a story about how your leadership lead to great things for you, your team, and the company.

- What's a time you disagreed with a decision that was made at work?

  This can be tricky, as the question is really asking about how you deal with authority, particularly when you don't agree with leadership. You need to demonstrate that you have tact when dealing with these problems, as well as being confident enough to voice your concerns but ultimately accepting whatever the final decision may be.

- How would your boss and co-workers describe you?

  Consider any strengths or traits that you haven't already talked about.

- Why was there a gap in your employment?

  Be honest but, if possible, spin it as a good thing. Perhaps you took a sabbatical so you could earn a certification or become better educated, or maybe you left the country to volunteer overseas. Even if you were fired and couldn't find work for six months, explain what you did to keep busy during that time.

- Can you explain why you changed career paths?

  Give a few examples of how your past experience is transferable to the new role, especially if you can make seemingly irrelevant experience seem very relevant to the role.

- How do you deal with pressure or stressful situations?

  Again, be honest. The interviewer wants to know that you can handle stress without having an anxiety attack prior to briefing the CEO, or a similar high-pressure situation. Demonstrate that you can take the stress and harness it for good, such as working out, writing, etc.

- What are your salary requirements?

  Know what the salary range is for this position, as well as the average in your locality. Make sure the hiring manager knows that you're flexible. You're communicating that you know your skills are valuable, but that you want the job and are willing to negotiate.

  Be wary of low-balling yourself to make yourself look better. You know what you need to pay the bills and maintain your lifestyle; don't sell yourself short, as it can be difficult to make it up in the future. In addition, don't tell how much you earn now, as it is another opportunity to low-ball you by offering slightly more than you make now.

  The job pays a particular amount, frequently within a range based on experience. Aim high but be willing to accept lower, but not too low. Also consider what you would be making in your current job if you were to receive an annual pay raise; one of the reasons to change jobs is to get a higher salary than you would with a normal pay increase.

- What do you like to do outside of work?

  Interviewers ask this type of question to see if candidates will fit in with the corporate culture and give them the opportunity to open up to an open-ended question. Plus, people like to talk about themselves.

- If you were an animal, which one would you want to be?

  These types of questions come up in interviews because hiring managers want to see how well you can think on your feet. There's no wrong answer here, but naturally it helps if your answer demonstrates your strengths or personality or connect with the hiring manager.

- How many tennis balls can you fit into a limousine?

  Google and other companies are famous for these types of brainteasers, especially in quantitative jobs. The interviewer isn't looking for the actual answer, but wants to see how your thought process works and what steps you take in problem solving.

  For example, in this particular question, you could say something like, "Assume a tennis ball is 4 inches in diameter, and assume the interior volume of a limo is 200 cubic feet. While I don't recall the formula to calculate the volume of a sphere, which could then be divided into the volume of the limo to get a rough estimate, we could potentially solve it another way.

  "If we assume 4.5 feet width, 4.5 feet height, and 10 feet length for the limo interior, it's slightly more than 200 cubic feet, so we'll stick with 200. If a tennis ball is 4 inches across, and 12 inches x 4.5 feet = 54 inches, then we could fit about 13 tennis balls across the limo, as well as vertically. Ten feet long is 120 inches, which equals about 30 tennis balls.

  "So, we know that we will have 13 x 13 x 30 = 5070 tennis balls, based on the assumptions."

  Naturally, you can make whatever assumptions you want, but ensure that your thought process is logical. Ask for a whiteboard, pen and paper, or whatever you need, though calculators and computers will probably be off limit.

- Are you planning on having children?

  Questions about your family status, gender ("How would you handle managing a team of all men?"), nationality ("Where were you born?"), religion, or age, are illegal—but they still get asked. You can refrain from answering the question, or steer your answer towards work, such as "I'm considering having children in the future, but want to ensure I have a steady job before I take on that commitment."

- What do you think we could do better or differently?

  This is a common one at startups. Hiring managers want to know that you not only have some background on the company, but that you're able to think critically about it and come to the table with new ideas.

  This is actually your time to shine. Everything else is about your history. The key point of an interview is what you can do for the company. If you don't have enough information about the company to have an answer already, simply ask for an example problem they may have that is related to the work you will be doing.

  Take some time to think about the answer; thinking out-loud is a good way to show how your think about problems. If possible, formulate your answer by considering your strengths and background to demonstrate that you utilize your experiences and known information to provide your best answer for the situation.

- Do you have any questions for us?

  What do you want to know about the position? The company? The department? The team? Consider anything that you would like to know before you start work there, or need to know to compare job offers.

  You'll cover a lot of this in the actual interview, so have a few less-common questions ready to go. Good questions put the interviewer into thinking mode, such as "What's your favorite part about working here?" or the company's growth, such as "What can you tell me about your new products or plans for growth?"

A.2. What are some technical interview questions?

As these questions are relatively open-ended, so that candidates can demonstrate their knowledge of algorithms, thought process, and other programming areas, answers will not be provided. Many programming-oriented websites will have various different ways to arrive at the same answer, so knowing the general workflow is most important.

However, it is highly recommended to practice some of these questions using only a whiteboard, pseudocode, and even a simple text editor, as most interviews do not allow a full IDE to be used during the interview.

Recognize that, with Python, a lot of these can be solved simply by using built-in Python features. Therefore, it is recommended to know the algorithmic way of solving these questions, as well as how to do it in Python.

A.2.1. What are some array-based questions?

- How do you find the missing number in a given integer array of 1 to 100?

- How do you find the duplicate number on a given integer array?
- How do you find the largest and smallest number in an unsorted integer array?
- How do you find all pairs of an integer array whose sum is equal to a given number?
- How do you find duplicate numbers in an array if it contains multiple duplicates?
- How is an integer array sorted in place using the quicksort algorithm?
- How do you remove duplicates from an array in place?
- How are duplicates removed from an array without using any library?

A.2.2. What are some linked list questions?

- How do you find the middle element of a singly linked list in one pass?
- How do you check if a given linked list contains a cycle?
- How do you find the starting node of a linked list cycle?
- How do you reverse a linked list?
- How do you reverse a singly linked list without recursion?
- How are duplicate nodes removed in an unsorted linked list?
- How do you find the length of a singly linked list?
- How do you find the second node from the end in a singly linked list?
- How do you find the sum of two linked lists using Stack?
- How do you insert a node into a sorted linked list while maintaining the sort?
- How do you compare two strings that are represented as linked lists?

A.2.3. What are some string coding questions?

- How do you print duplicate characters from a string?
- How do you check if two strings are anagrams of each other?
- How do you print the first non-repeated character from a string?
- How can a given string be reversed using recursion?
- How do you check if a string contains only digits?
- How are duplicate characters found in a string?
- How do you count a number of vowels and consonants in a given string?
- How do you count the occurrence of a given character in a string?
- How do you find all permutations of a string?

- How do you reverse words in a given sentence without using any library method?
- How do you check if two strings are a rotation of each other?
- How do you check if a given string is a palindrome?

A.2.4. What are some binary tree questions?

- How is a binary search tree implemented?
- How do you perform preorder traversal in a given binary tree?
- How do you traverse a given binary tree in pre-order without recursion?
- How do you perform an inorder traversal in a given binary tree?
- How do you print all nodes of a given binary tree using in-order traversal without recursion?
- How do you implement a postorder traversal algorithm?
- How do you traverse a binary tree in post-order traversal without recursion?
- How are all leaves of a binary search tree printed?
- How do you count a number of leaf nodes in a given binary tree?
- How do you perform a binary search in a given array?

A.2.5. What are some graph questions?

- Given a graph and a source vertex in the graph, how do you find shortest paths from source to all vertices in the given graph?
- How is a breadth first search in a graph implemented?
- How is a depth first search in a graph implemented?
- Given a dictionary of words and a M x N board where every cell has one character, how would you find all possible words (from the dictionary) that can be formed by a sequence of adjacent characters? (Think of a Boggle game).

A.2.6. What are some dynamic programming questions?

- Given two sequences, how do you find the length of the longest sub-sequence present in both of them?
- How do you find the length of the longest sub-sequence of a given sequence such that all elements of the sub-sequence are sorted in increasing order?
- Given a distance value d, what is the total number of different ways to go from start to d if you can select between one step, two steps, and three steps?

A.2.7. What are some algorithm implementation questions?

- How is a bubble sort algorithm implemented?
- How is an iterative quicksort algorithm implemented?
- How do you implement an insertion sort algorithm?
- How is a merge sort algorithm implemented?
- How do you implement a bucket sort algorithm?
- How do you implement a counting sort algorithm?
- How is a radix sort algorithm implemented?

A.2.8. What are some number theory questions?

- Given three numbers x, y, and p, how would you calculate $(x^y)\%p$ using the modular property $(m \times n)\%$? Solve it both iteratively and recursively.
- Given a number n, how would you check if it is prime or not?
- Given a set of points in the plane, how would you find the convex hull of the set? (Convex hull is the smallest convex polygon that contains all the points of it.)

A.2.9. What are some miscellaneous questions?

- How do you swap two numbers without using the third variable?
- How do you check if two rectangles overlap with each other?
- The cost of a stock on each day is given in an array. For example, if the given array is [50, 160, 260, 313, 40, 535, 695], the maximum profit can earned by buying on day 0, selling on day 3. Again buy on day 4 and sell on day 6. If the given array of prices is sorted in decreasing order, then profit cannot be earned at all.
  - How would you find the maximum profit that can be made by buying and selling only once?
  - How would you find the maximum profit that can be made by buying and selling multiple times?

A.3. Summary

In this chapter, we provided some questions and answers to common interview questions, regardless of the type of job you will be applying for. We also provided a number of different questions that might show up during a programmer's job interview.

Recognize that there are many different web sites that provide additional questions, some of which are actual questions seen by applicants at particular companies. So while the questions here are a good start to get you thinking, be sure to look for potential questions that may be asked by the particular company you are applying with.

Bibliography

[1] https://auth.geeksforgeeks.org/user/dilipjain0504

[2] https://auth.geeksforgeeks.org/user/Mayank%20Khanna%202

[3] https://auth.geeksforgeeks.org/user/Nikita%20tiwari

[4] https://auth.geeksforgeeks.org/user/Smitha%20Dinesh%20Semwal

[5] https://auth.geeksforgeeks.org/user/simranjenny84

[6] https://auth.geeksforgeeks.org/user/shreyashagrawal

[7] https://auth.geeksforgeeks.org/user/maheswaripiyush9

[8] https://auth.geeksforgeeks.org/user/Mithun%20Kumar

[9] https://auth.geeksforgeeks.org/user/KyleMcClay

[10] https://auth.geeksforgeeks.org/user/princiraj1992

[11] https://auth.geeksforgeeks.org/user/Bhavya%20Jain

[12] https://auth.geeksforgeeks.org/user/Neelam%20Yadav

[13] https://en.wikipedia.org/w/index.php?title=Cycle_detection&oldid=1018282128

[14] https://auth.geeksforgeeks.org/user/Divyanshu%20Mehta

[15] https://auth.geeksforgeeks.org/user/mythri1020

[16] https://auth.geeksforgeeks.org/user/himanshukanojiya

[17] https://auth.geeksforgeeks.org/user/Nikhil%20Kumar%20Singh

[18] https://auth.geeksforgeeks.org/user/Aryan%20Garg

[19] https://auth.geeksforgeeks.org/user/vibhu4agarwal

[20] https://auth.geeksforgeeks.org/user/anubhavraj_08

[21] https://auth.geeksforgeeks.org/user/jahid_nadim

[22] https://www.geeksforgeeks.org/bubble-sort/

[23] https://auth.geeksforgeeks.org/user/Mohit%20Kumra

[24] https://www.geeksforgeeks.org/python-program-for-selection-sort/

[25] https://auth.geeksforgeeks.org/user/anush_krishna_v

[26] https://auth.geeksforgeeks.org/user/MuskanKalra1

[27] https://auth.geeksforgeeks.org/user/OneilHsiao

[28] https://www.geeksforgeeks.org/python-sort-list-according-second-element-sublist/

[29] https://www.geeksforgeeks.org/ways-sort-list-dictionaries-values-python-using-itemgetter/

[30] https://www.geeksforgeeks.org/analysis-different-methods-find-prime-number-python/

[31] https://www.geeksforgeeks.org/sieve-of-eratosthenes/

[32] https://www.geeksforgeeks.org/ugly-numbers/

[33] https://auth.geeksforgeeks.org/user/Sam007

[34] https://auth.geeksforgeeks.org/user/Danish%20Raza

[35] https://www.geeksforgeeks.org/the-skyline-problem-using-divide-and-conquer-algorithm/

About the Author

Cody Jackson is the author of *Programming in Python with Cody Jackson, Secret Recipes of the Python Ninja, Learning to Program Using Python,* and the novel *The Master's Keep.* He is currently the GUI team supervisor for the open-source OpenCPI project in support of GNU Radio.