17. The Right Choice (Choosing Algorithms)

Sometimes I sits and thinks, and sometimes I just sits.
Engraving on a bench, adapted from Punch, Volume 131, page 297 (1906)

Suppose I told you I wanted a pizza for lunch and asked you to sort it out. There are several different approaches you could use to solve the problem. You could for example phone for a take-away. You could take me to a pizza restaurant. You could buy a frozen one and heat it in the microwave, or you could cook one from scratch, base and all. How would you decide which to do? At different times any of these choices might be the best thing to do. Either you choose one at random, or you find out more information first. That is often the problem a software developer is in. The customer has said what they want, but not given enough detail to be sure of the best solution. The person who is to solve the problem must determine more about the problem first. Back to pizza, it is important to do two things: find out the advantages and disadvantages of each possible solution, and find out what other restrictions there are on your choice. For example, who is paying? What pizza topping is required – something straightforward like cheese and tomato pizza or something more exotic like banana and peanut butter pizza? How much time do you have to come up with the pizza? How much time will you have to pay for the pizza? There are many more questions that could be considered, the more that are considered, the better the final decision is likely to be. What then are the advantages and disadvantages of the different choices? An advantage and disadvantage of each is given below – no doubt you can think of many more.

Solution	**Advantages**	**Disadvantages**
Delivery	Low-hassle	May never arrive.
Restaurant	Good quality	Expensive
Frozen	Cheap	Taste like cardboard
Home-made	Any topping possible	May be inedible

If you compare the restrictions you have identified with the advantages and disadvantages, you are then in the position to solve the problem.

In the previous chapters we have looked at a whole range of different algorithms for doing the same thing, whether searching, sorting or whatever. How do you choose? As with pizza, in general there is rarely a single "best" way. Each solution may be the most appropriate in a given situation. The main considerations are how fast the algorithm is and how much space the data structure takes up. Sometimes it is space that matters the most, and sometimes it is speed. However, there can be other factors. For example, for some tasks the most important thing is that you do it accurately without making mistakes. In terms of software, how sure are you that you can write a program that implements the algorithm correctly? Sometimes what matters is the ease with which you work out how to do the task. There is little point spending hours working out the perfect solution to a problem if the worst solution would only have taken minutes to solve the problem adequately. Even if we are only concerned with speed, it may only be one part of the task that needs to be done quickly. We discuss some of these issues in this section.

Time to solve the problem

Often the most important thing is to get the task done quickly. However, this may not just mean using the fastest algorithm, though that is usually important. If the task must only be done once or twice then the time taken to decide which is the best algorithm may be the most important thing. It may take longer than that algorithm will take to complete the task once selected. It is for similar reasons that when politicians are bogged down in some controversial issue, they set up a committee to determine the best course of action. The idea is to use a form of decision making that is so slow, the solution will no longer be needed by the time it has been determined and can be quietly shelved. Usually we do want the solution to be used, however. It is therefore important to decide before we start how much time it is worth spending devising the solution.

If I am late for a class, perhaps because I over sleep, I could sit down with a map of the site and work out the fastest route through the maze of corridors to get from where I am to where I want to be. Alternatively, given the difference between different routes will only be a minute or so anyway, I could spend the time going to the class by the first route that comes into my head. In this situation there is little point spending much time coming up with the best route (algorithm) because the time that matters is the combined time to devise the algorithm and to execute it.

Critical Operations

When considering which of two algorithms to choose, a useful idea is that of critical operation – some important step of the algorithm that you wish to do least often.

We looked at a puzzle that involved a farmer getting across a river.

To cross the river:

1. The farmer crosses with the hen.
2. The farmer returns alone.
3. The farmer crosses with the corn.
4. The farmer returns with the hen.
5. The farmer crosses with the dog.
6. The farmer returns alone.
7. The farmer crosses with the hen.

I claimed that this was the fastest algorithm. However I have no idea how long it would take the farmer to cross the river each time. However I do not need to to be able to say it is faster than some other algorithm. I left you to work out the other equally fast algorithm: here it is.

To cross the river:

1. The farmer crosses with the hen.
2. The farmer returns alone.
3. The farmer crosses with the dog.
4. The farmer returns with the hen.
5. The farmer crosses with the corn.
6. The farmer returns alone.
7. The farmer crosses with the hen.

The farmer just takes the dog across before the corn. I claim this is just as fast, but how do I know without getting a farmer, a corracle, etc and timing them following the two sets of instructions? My claim is based on the fact that they both have the same number of steps. I am assuming that every time the farmer crosses the river it takes roughly the same time and that is what takes the most time. It may be that it is faster

to cross with the corn than the dog, or quicker to load the dog if there is no corn on the bank, but such time differences are negligible compared with the time to cross the river. Both the above algorithms require the same number of crossings so they are equally fast algorithms. If you come up with an algorithm that has 8 crossings, then I will deduce that it will take longer. I have decided that *crossing the river* is the critical operation. By counting it I get a measure of the efficiency of the algorithm that I can use to decide between two algorithms – and I can make the decision without ever going near a river. Similarly with algorithms that have been turned into programs, it is possible to decide that some step is the critical operation and work out which of two programs will be faster by comparing them. This can be done without running and timing the programs – just by looking at them.

Consider again the plight of Jean-Dominique Bauby and other people suffering from Locked-in Syndrome. Bauby had only the use of his eyelid to communicate due to his otherwise total paralysis. We argued that the method he used to communicate was inefficient. The person he was talking to recited the letters of the alphabet in turn until he blinked to indicate the correct letter had just been spoken (a form of linear search). We assumed that the number of questions asked should be kept as small as possible: it was being treated as the critical operation. We work out the number of questions that must be asked on average to find the correct letter. Here it is 26 questions in the worst case as we will run through the whole alphabet before finding the correct letter if the letter happens to be Z. Other algorithms considered were based on binary search methods: asking questions such as "Does it come before M in the alphabet?" that rule out half the remainder of the alphabet with each question. These methods take at most 5 questions. Thus the first algorithm takes 26 questions (critical operations) in the worst case whilst the binary ones take at most 5 questions. We conclude that the binary approaches are better, and that Bauby chose a poor algorithm.

Our reasoning above was based on an assumption: we assumed that what mattered most was the number of questions asked. However, given Bauby was paralysed it is possible that blinking took a monumental effort from him. Reading out letters of the alphabet on the other hand, though boring, would not be that difficult for his visitors. If this were so then the step he would want his communication algorithm to minimise, not the number of questions, but the number of blinks. It is blinks that are now the critical operation. The first algorithm required Bauby to blink just once per letter. The second required him in the worst case to give 5 blinks for each letter. This is because assuming a blink is used to mean "yes", the worst case is when the answer to each question is yes. 5 yes answers then require 5 blinks. Thus perhaps Bauby had not chosen a bad algorithm after all. Which algorithm was best for him would depend on what the critical operation was which in turn depends on how easy it was for him to blink. As with any problem solving, it is important to understand the problem in the sense of knowing as much as possible it if you are to devise a good solution.

Pre-processing

Another important consideration with respect to speed, is over which part of the algorithm you wish to be fast. A common feature of algorithms is known as pre-processing. This involves doing some extra work once at the start before doing the task you are really interested in. It is essentially the organisation of things in a way that makes the task easier. We have seen examples of this already. Linear search, as we saw, involves searching straight away. If I want to find a book on my bookshelf, I start at one end and work to the other checking each book. I do not make any attempt

to organise things first. That is fine as long as there are only a small number of books or I will be looking for books only occasionally. If there are lots of books it is worth pre-processing them: that is spending some time before doing any searching to organise them. That is what libraries and bookshops do. In both cases there are lots of books and lots of searches for books are being made. The time spent organising the books (which takes a very long time) is worth it because each time anyone searches for a book they save some time. Each person does not save the amount of time the librarian spent organising the books and writing out index cards. That is not the point however. When you add together all the time saved on all the searches made, time is saved in the long run. It is only worth it if there are lots of searches as then you are saving the small amount of time more often. Similarly if the number of books to search through is large, each time someone searches, the time saved on each search is larger. There is thus a point when the number of searches and number of books makes it worthwhile organising first. Below this number there is no point.

Compilation (translating a set of instructions from one language to another that can be executed) is a pre-processing task. We compile a program once and then can execute the code as many times as we wish without having to translate the original again. An interpreter on the other hand does not do this pre-processing. The translation is done every time the program is run as each step is executed. Interpreted languages consequently are slower to execute. When we discussed compilers and interpreters we used the example of my using Arabic directions to get to the market. If I am willing to wait and get the whole set of instructions translated before I start, then if I am to go to the market on several days, and so ultimately save time. If instead I am keen to set off immediately and stop at each junction to get the next instruction translated then on the first trip i will get there at the same time. However, on subsequent trips I would have to repeat the work and continue to stop at each junction. Thus subsequent trips will be slower than if I had had the whole document translated.

Many of the algorithms we have looked at involve some form of pre-processing. For example, binary search and related algorithms require the data to be sorted first – that is why the telephone directory is in alphabetical order. Bucket search requires all search questions to be pre-computed. For example, if we kept appointments as just a list, we would do limited pre-processing. On being told of each appointment we would just write it down on the end of the list. If we use a diary, however, we not only write it down, we need to search for the correct place to write it. Thus on being given new information we do extra work, but the pay-off is that it is now much quicker to look up if a given day is free. The pre-processing is in setting up the bucket array style data structure. In fact, if any special data structure is used, then putting the data in to that data structure could be considered as a pre-processing task.

In many cases, pre-processing involves moving tasks from the point when the task is to be done, to the point when the data is input. Most of the time my sink is surrounded by dirty pots – I usually wash the breakfast pots when I get in from work in the evening, as I do not have time in the morning – I would miss my train. I am thus organising the way I do the task in a way so that I can do the time-consuming part when I have most time. However, if I have porridge in the morning, then by the evening it is set on the dish like concrete and takes much longer to wash. I can solve this by quickly rinsing it under the tap in the morning when I put it by the sink. I am

spending a little extra time when the pots arrive at the sink, to speed the actual task of washing them when I finally get round to it.

One reason for pre-processing can be, not that it saves time overall, but that it saves time at a point when time is of the essence. A few years ago I ran a marathon. In the months before doing it I spent lots of time out running, to train myself so that I could do the actual marathon as fast as possible. The total time I spent running was thus much longer than if I had done no training, and taken 8 hours or so walking the 26 miles. In this instance, the total time running (including the training) was not the point. I had plenty of time in the months before hand to spend as long as I liked on the road. What mattered was that I was fast on one particular day. Computing applications are often like that. There is plenty of time to prepare. What matters is that when someone asks for a job to be done, at that point the computer is as fast as possible.

Speed versus Space
When choosing algorithms, there is often a trade-off between use of space and speed. One way to increase the speed with which a task is done is to use up more space. Diaries are an example of this – we are willing to waste many blank pages because it allows us to easily check what we are doing on a given day.

Different modes of transport can also provide a similar trade-off. I could go to work by car. However, the roads are very crowded. Next time you are in a traffic jam, count the number of people in each car. There is mainly only one person per car. Each person in the traffic jam is taking up a whole cars worth of road. I often go to work by Bus. It is usually crammed full with 40 or 50 people. Together we take up a fraction of the road space of the car drivers. If everyone went by bus, we would need fewer lanes on the roads (and there would be fewer traffic jams). Why do all those people prefer to go by car? One reason is the speed. Even with traffic jams it would still be quicker for me to drive to work. I am personally willing to take longer to get to work, in part because of the environmental benefit of using less road space. There is thus a trade-off between speed and space. For other people the speed is more important so they use a different solution to the problem of getting to work from me.

Accuracy
Often the accuracy of the solution is the most important thing. A slightly wrong answer is worse than for example, a slow answer. Returning to the pizza example considered above. Suppose I had asked for a vegetarian pizza. On looking at the options you discovered that using a frozen Meat Feast Pizza was the cheaper than any other form of pizza and could be obtained most quickly as you had one in your freezer already. Given this you decide to give me the Meat Feast. Your solution is close to one that is correct – it is after-all a pizza and in many situations (eg if I was not actually vegetarian might be a good solution). However, in this case accuracy in fulfilling my request was most important as I do not want to eat meat and am willing to pay for the accuracy.