Fundamental Data Structures and Algorithms 07 - Algorithm Design Techniques

Fundamental Data Structures and Algorithms 07 - Algorithm Design Techniques
- Unit 4: Algorithm Design Techniques
    - 4.1 Brute Force
    - 4.2 Divide-and-conquer
    - 4.3 Dynamic programming
        - 4.3.1 Memoization
        - 4.3.2 Tabulation
    - 4.4 Greedy Algorithms
    - 4.5 Space and Time Tradeoff

Unit 4: Algorithm Design Techniques

At a higher level of abstraction are design techniques which describe the design of algorithms, rather the design of data structures. These embody and generalize important design concepts that appear repeatedly in many problem contexts. They provide a general structure for algorithms, leaving the details to be added as required for particular problems. These can speed up the development of algorithms by providing familiar proven algorithm structures that can be applied straightforwardly to new problems. We shall see a number of familiar design techniques throughout these notes.

In general, we can discern three broad approaches to algorithm design. They are:

- Brute force
- Divide-and-conquer
- Dynamic Programming
- Greedy Algorithms

4.1 Brute Force

The brute force algorithmic design pattern is a powerful technique for algorithm design when we have something we wish to search for or when we wish to optimize some function. When applying this technique in a general situation, we typically enumerate all possible configurations of the inputs involved and pick the best of all these enumerated configurations.

In Chapter 1.3.2, we have seen how we can search an element in an array or list linearly. This is a classic example of a brute force method as the algorithm iterates through the entire dataset from start to the end without any consideration of optimization.

In general, brute force algorithms are suitable for most applications with small input sizes of n. However, at large input sizes, we will see how it can significantly slow down the running time. Consider a simplified password-cracking program that uses a brute force method to guess a password. Assume the passwords only consists of lower case letters $- a$ to z. Thus a password of 1 character long will have a total of 26 possible combinations. Also assume that the computing system is able to make 1000 guesses per second.

Below is a compilation of the time required to exhaustively guess a password for different password lengths consisting of only lower case letters:

length of password (only lower case letters)	**time required to crack password**
1	0.026 seconds
2	0.676 seconds
4	457 seconds or 7.6 minutes
6	309×10^6 seconds or 10 years
8	209×10^9 seconds or 6600 years
10	141×10^{12} or 4.5 million years
12	95×10^{15} or 3 billion years

Note: the age of the universe is around 13.8 billion years.

Brute force method is not suitable to guess passwords because the time required to crack the password scales exponentially to the length of the password. Therefore, other methods are required.

4.2 Divide-and-conquer

As the name suggests, the divide and conquer paradigm involves breaking a problem into smaller sub problems, and then in some way combining the results to obtain a global solution. This is a very common and natural problem solving technique, and is, arguably, the most commonly used approach to algorithm design.

The divide-and-conquer design consists of the following three steps:

1. *Divide*: If the input size is smaller than a certain threshold (say, one or two elements), solve the problem directly using a straightforward method and return the solution so obtained. Otherwise, divide the input data into two or more disjoint subsets.
2. *Conquer*: Recursively solve the subproblems associated with the subsets.
3. Combine: Take the solutions to the subproblems and merge them into a solution to the original problem.

In Chapter 2.2.2, we have seen how Merge Sort uses this design technique. With reference to Fig 2.3, we can see that to sort a sequence S with n elements using the three divide-and-conquer steps, the merge sort algorithm proceeds as follows:

1. *Divide*: If S has zero or one element, return S immediately; it is already sorted. Otherwise (S has at least two elements), remove all the elements from S and put them into two sequences, $left$ and $right$, each containing half of the elements of $S -$ that is, $left$ contains the first $\frac{n}{2}$ elements of S, and $right$ contains the remaining $\frac{n}{2}$ elements.
2. *Conquer*: Recursively sort sequences $left$ and $right$.
3. *Combine*: Put back the elements into S by merging the sorted sequences $left$ and $right$ into a sorted sequence.

4.3 Dynamic programming

This technique is similar to divide and conquer, in that a problem is broken down into smaller problems. In divide and conquer, each subproblem has to be solved before its results can be used to solve bigger problems. By contrast, dynamic programming does not compute the solution to an already encountered subproblem. Rather, it uses a remembering technique to avoid the recomputation.

As we have already described, in this approach, we divide a problem into smaller subproblems. In finding the solutions to the subprograms, care is taken not to recompute any of the previously encountered subproblems. This sounds a bit like recursion, but things are a little broader here. A problem may lend itself to being solved by using dynamic programming but will not necessarily take the form of making recursive calls.

A property of a problem that will make it an ideal candidate for being solved with dynamic programming is that it should have an overlapping set of subproblems. Once we realize that the form of subproblems has repeated itself during computation, we need not compute it again. Instead, we return the result of a pre-computed value of that subproblem previously encountered.

To avoid a situation where we never have to re-evaluate a subproblem, we need an efficient way in which we can store the results of each subproblem. The following two techniques are readily available: memoization and tabulation.

We will use the Fibonacci series to illustrate both memoization and tabulation techniques of generating the series.

4.3.1 Memoization

This technique starts from the initial problem set and divides it into small subproblems. After the solution to a subprogram has been determined, we store the result to that particular subproblem. In the future, when this subproblem is encountered, we only return its pre-computed result.

Let's generate the Fibonacci series to the fifth term: $1 \ 1 \ 2 \ 3 \ 5$

A recursive style of a program to generate the sequence is as follows:

```
def fib(n):
    if n <= 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

The code is very simple but a little tricky to read because of the recursive calls being made that end up solving the problem.

When the base case is met, the `fib()` function returns 1. If `n` is equal to or less than 2, the base case is met.

If the base case is not met, we will call the `fib()` function again and this time supply the first call with $n-1$ and the second with $n-2$: `return fib(n-1) + fib(n-2)`.

The layout of the strategy to solve the i^{th} term in the Fibonacci sequence is as follows:

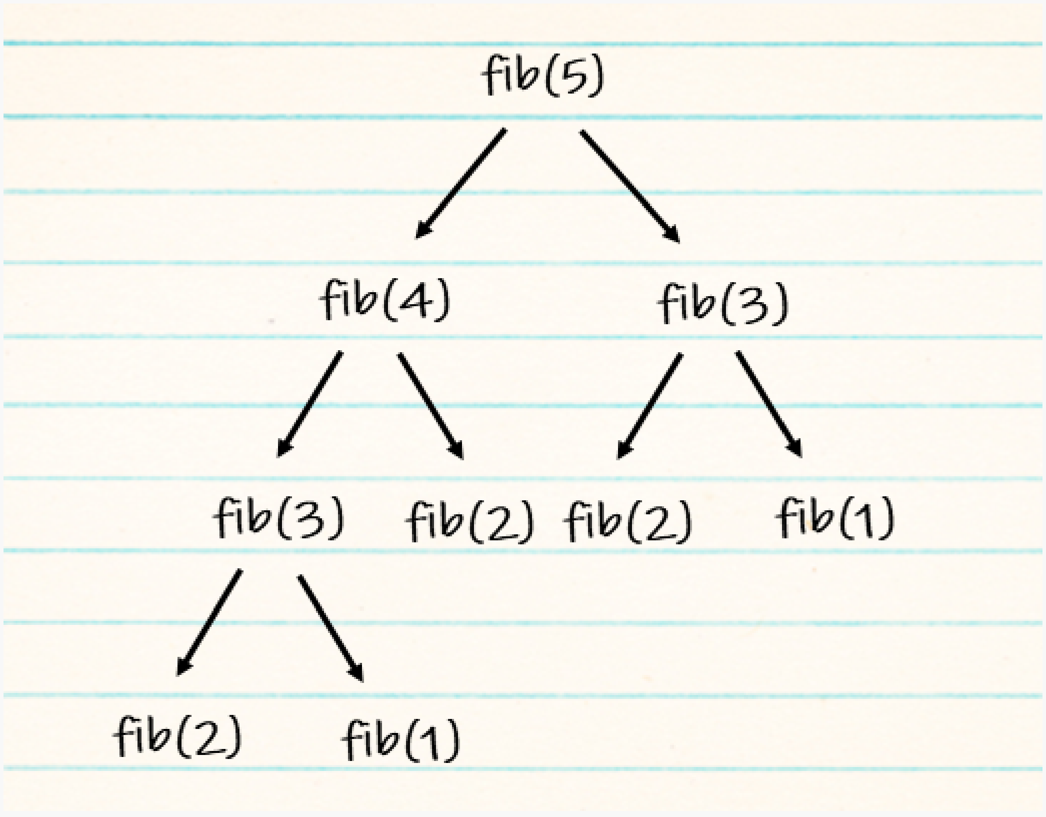

Fig 4.1. Fibionacci series recursive tree

A careful observation of the preceding tree shows some interesting patterns. The call to `f(1)` happens twice. The call to `f(1)` happens thrice. Also, the call to `f(3)` happens twice.

The return values of the function calls to all the times that `fib(2)` was called never changes. The same goes for `fib(1)` and fib(3). The computational time is wasted since the same result is returned for the function calls with the same parameters.

These repeated calls to a function with the same parameters and output suggest that there is an overlap. Certain computations are reoccurring down in the smaller subproblems.

A better approach would be to store the results of the computation of `fib(1)` the first time it is encountered. This also applies to `fib(2)` and `fib(3)`. Later, anytime we encounter a call to `fib(1)`, `fib(2)`, or `fib(3)`, we simply return their respective results.

The diagram of our `fib` calls will now look like this:

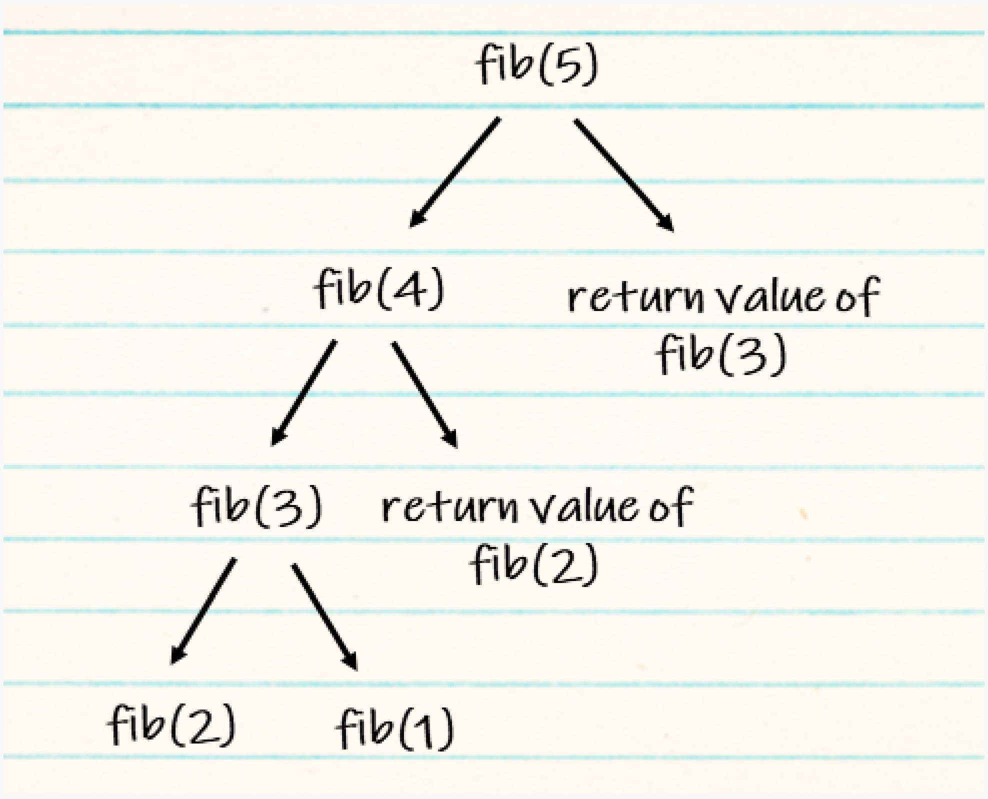

Fig 4.2. Dynamic Fibionacci series recursive tree

We have now completely eliminated the need to compute `fib(3)`, `fib(2)`, and `fib(1)`. This typifies the memoization technique wherein breaking a problem into its subproblems, there is no recomputation of overlapping calls to functions. The overlapping function calls in our Fibonacci example are `fib(1)`, `fib(2)`, and `fib(3)`:

```python
def dyna_fib(n, lookup):
    # This list will store the value of the computation of the various calls to
dyna_fib().
    # Any call to the dyna_fib() with n being less than or equal to 2 will
return 1.
    if n <= 2:
        lookup[n] = 1
    # When dyna_fib(1) is evaluated, we store the value at index 1 of lookup
    if lookup[n] is None:
        lookup[n] = dyna_fib(n-1, lookup) + dyna_fib(n-2, lookup)
    return lookup[n]
```

To create a list of 1,000 elements, we do the following and pass it to the `lookup` parameter of the `dyna_fib` function: `map_set = [None]*(10000)`.

We pass lookup so that it can be referenced when evaluating the subproblems. The calls to `dyna_fib(n-1, lookup)` and `dyna_fib(n-2, lookup)` are stored in `lookup[n]`. When we run our updated implementation of the function to find the i^{th} term of the Fibonacci series, we realize that there is considerable improvement. This implementation runs much faster than our initial implementation. Supply the value 20 to both implementations and witness the difference in the execution speed. The algorithm sacrificed space complexity for time because of the use of memory in storing the result to the function calls.

4.3.2 Tabulation

In tabulation, we settle on an approach where we fill a table of solutions to subproblems and then combine them to solve bigger problems. This approach solves the bigger problem by first working out a route to the final solution. In the case of the `fib()` function, we will develop a table with the values of `fib(1)` and fib`(2)` predetermined. Based on these two values, we will work our way up to `fib(n)`:

```
def fib(n):
    results = [1, 1]
    for i in range(2, n):
        results.append(results[i-1] + results[i-2])
    return results[-1]
```

The results variable is at index 0, and 1 the values, 1 and 1. This represents the return values of `fib(1)` and `fib(2)`. To calculate the values of the `fib()` function for higher than 2, we simply call the `for` loop appends the sum of the `results[i-1] + results[i-2]` to the list of results.

4.4 Greedy Algorithms

A greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. Some of the examples that uses greedy algorithms are Prim's and Kruskal's algorithms. Kruskal's algorithm qualifies as a greedy algorithm because at each step it adds an edge of least possible weight to the new minimum spanning tree. On the other hand, Prim's algorithm qualifies as greedy since at each step it adds to the tree an edge that contributes the minimum amount possible to the tree's weight.

Let's examine a very simple use of this greedy technique in a coin-counting problem. In some arbitrary country, we have the denominations 1 GC, 5 GC, and 8 GC. Given an amount such as 12 GC, we may want to find the least possible number of denominations needed to provide change. Using the greedy approach, we pick the largest value from our denomination to divide 12 GC. We use 8 because it yields the best possible means by which we can reduce the amount 12 GC into lower denominations.

The remainder, 4 GC, cannot be divided by 5, so we try the 1 GC denomination and realize that we can multiply it by 4 to obtain 4 GC. At the end of the day, the least possible number of denominations to create 12 GC is to get a one 8 GC and four 1 GC notes.

So far, our greedy algorithm seems to be doing pretty well. A function that returns the respective denominations is as follows:

```python
def basic_small_change(denom, total_amount):
    sorted_denominations = sorted(denom, reverse=True)
    number_of_denoms = []
    for i in sorted_denominations:
        div = total_amount / i
        total_amount = total_amount % i
        if div > 0:
            number_of_denoms.append((i, div))
    return number_of_denoms
```

This greedy algorithm always starts by using the largest denomination possible. `denom` is a list of denominations. `sorted(denom, reverse=True)` will sort the list in reverse so that we can obtain the largest denomination at index 0.

Now, starting from index 0 of the sorted list of denominations, `sorted_denominations`, we iterate and apply the greedy technique. The `for` loop will run through the list of denominations. Each time the loop runs, it obtains the quotient, `div`, by dividing the `total_amount` by the current denomination, `i`. `total_amount` is updated to store the remainder for further processing. If the quotient is greater than 0, we store it in `number_of_denoms`.

Unfortunately, there are instances where our algorithm fails. For instance, when passed 14 GC, our algorithm returns one 8 GC and four 1 GC. This output is, however, not the optimal solution. The right solution will be to use two 5 GC and two 1 GC denominations.

A better greedy algorithm is presented here. This time, the function returns a list of tuples that allow us to investigate the better results:

```
def optimal_small_change(denom, total_amount):
    sorted_denominations = sorted(denom, reverse=True)
    series = []
    for j in range(len(sorted_denominations)):
        term = sorted_denominations[j:]
        number_of_denoms = []
        local_total = total_amount
        for i in term:
            div = local_total / i
            local_total = local_total % i
            if div > 0:
                number_of_denoms.append((i, div))
        series.append(number_of_denoms)
    return series
```

The outer `for` loop enables us to limit the denominations from which we find our solution. Assuming that we have the list `[5, 4, 3]` in `sorted_denominations`, slicing it with `[j:]` helps us obtain the sub-lists `[5, 4, 3]`, `[4, 3]`, and `[3]`, from which we try to get the right combination to create the change.

4.5 Space and Time Tradeoff

A space–time tradeoff in computer science is a case where an algorithm or program trades increased space usage for reduced time or alternatively, lesser space in exchange for more time. Here, space refers to the data storage consumed in performing a given task (RAM, HDD, SSD, etc), and time refers to the time consumed in performing a given task i.e. computational time.

Most computers have a large amount of space, but not infinite space. Also, most people are willing to wait a little while for a big calculation, but not forever. So if your problem is taking a long time but not much memory, a space-time trade-off would let you use more memory and solve the problem more quickly. Or, if it could be solved very quickly but requires more memory than you have, you can try to spend more time solving the problem in the limited memory. The latter is more common in embedded systems where memory is quite limited.

The most common condition is an algorithm using a ***lookup table***. This means that the answers for some question for every possible value can be written down. One way of solving this problem is to write down the entire lookup table, which will let you find answers very quickly, but will use a lot of space. We have seen how dynamic programming improves computational time by using more memory in the Fibonacci example. Another way is to calculate the answers without writing down anything, which uses very little space, but might take a long time.

A space-time tradeoff can be used with the problem of data storage. If data is stored ***uncompressed***, it takes more space but less time than if the data were stored ***compressed*** (since compressing the data decreases the amount of space it takes, but it takes time to run the compression algorithm).

Larger code size can be used to increase program speed when using ***loop unwinding*** or ***loop unrolling***. This technique makes the program code longer for each iteration of a loop, but saves the computation time needed for jumping back to the beginning of the loop at the end of each iteration. Below is a simple illustration of loop unwinding:

normal loop	**after loop unwinding**

`import timeit`
`sum = 0`
`start = timeit.default_timer()`
 	
`for i in range(1000000):`
`        sum = sum + 1`
 	
`stop = timeit.default_timer()`
`print(sum)`
	`import timeit`
	`sum = 0`
	`start = timeit.default_timer()`
	
	`for i in range(0, 1000000, 10):`
	`        sum = sum + 1`
	`        sum = sum + 1`
	`        sum = sum + 1`
	`        sum = sum + 1`
	`        sum = sum + 1`
	`        sum = sum + 1`
	`        sum = sum + 1`
	`        sum = sum + 1`
	`        sum = sum + 1`
	`        sum = sum + 1`
	
	`stop = timeit.default_timer()`
	`print(sum)`

`1000000`
`Time: 0.07675480000000334`
	`1000000`
	`Time: 0.0476643000000081`

While we have covered many fundamental data structures and algorithms, there are many others out there. Each data structure has its own advantages and disadvantages and must be used in accordance to the needs of the application. As data structures are being used throughout the field of computer science, it is therefore necessary that the topic is carefully studied and understood by programmers to ensure the design of efficient and effective software and applications.