Algorithms and Data Structures
Week 3: Algorithm Design Strategies

Abstract

Summary: This week we'll be concentrating on some common algorithm design patterns that crop up so frequently they have special names.

We'll have a look at some typical examples, of how these strategies can be applied.

- **Book reference**: Drozdek chapters 5 & 13.3;
- **Resources**: Week 3 archived eclipse program bundle.
- **Workshop exercises:** Submit answers to questions 2, 6 and 7 at the end of these notes.

  There is a submission box open in iLearn for these solutions. **Submission is entirely optional but strongly recommended.** Although there are no marks for submission, recall that the Week 6 in class test will be formed from a selection of the exercises you have been asked to submit in these workshops.
- **Extension questions:** Exercises 3 and 12 at the end of these notes. (This is extension only — it will not be part of the test.)

Design strategies

Divide and conquer

In our use of recursion we have seen how a method may be defined in terms of itself. A special case of this idea is called *divide and conquer* which occurs when the recursive called are applied to substructures comprising roughly half the size of the original substructure. When this works out it can result in a very efficient implementation. A typical use of this strategy is in e.g. binary search.

Estimating worst case complexity for recursive strategies

Given an algorithm which uses a recursive strategy:

```
Object function(Object A) {
  DIVIDE A INTO TWO HALVES, CALL THEM B, C;

   Object D= function(B);
   Object E= function(C);

   COMBINE D and E APPROPRIATELY TO GET Object F;

   return F;
}
```

Now let N be the original size of `A`, so that `B, C` each have sizes $N/2$, and let $T(N)$ be the big oh running time of `function`. We can compute a general recursive equation which, if we can solve it, tell us what $T(N)$ is.

$$T(N) = 2 \times T(N/2) + \textit{the time it takes to compute } \texttt{F}$$

In lectures we'll discuss how to do that for some examples, but here are some typical patterns that we'll come across as COMP225 progresses.

- $T(N) = 2 \times T(N/2) + O(N)$ – this is the pattern in mergesort. Solution: $O(n \log n)$.
- $T(N) = T(N/2) + O(1)$ – this is the pattern in binary search. Notice that we do not multiply $T(N/2)$ by 2 because in the algorithm we only process one half or other (i.e. not both) of the array. Solution: $O(\log n)$.
- $T(N) = 3 \times T(N/2) + O(N)$ – this is the pattern in integer multiplication. Notice that we multiply $T(N/2)$ by three because we call recursively the function on three instances of subproblem, each one about half the size of the original. Solution: $O(N^{1.59})$.

Integer multiplication

You might wonder why "number crunching" is really fast on computers. That's because the algorithm to multiply binary integers is not the one we were taught at school, but instead tries to optimise the time spent by using clever tricks based on its representation. Here's how it works in decimal notation. For binary representation, the ideas are exactly the same, but the numbers are harder to read.

Suppose we want to multiply two digit numbers together. Any such number is expressible as $10x + y$ where $0 \leq x, y < 10$.

A simple rule to multiply such numbers is summarised by this equation:

$$\begin{aligned} \texttt{mult}(10x + y, 10z + w) = \quad & 100\texttt{mult}(x, z) + \\ & 10(\texttt{mult}(x, w) + \texttt{mult}(y, z)) + \\ & \texttt{mult}(y, w) \end{aligned}$$

This would yield a recurrence relation $T(N) = 4T(n/2) + O(N)$, which has a solution $O(N^2)$. We can do better because we can rewrite the middle line of the above in terms of *three* simpler multiplications:

$$\texttt{mult}(x, w) + \texttt{mult}(y, z) = \texttt{mult}(x + y, w + z) - \texttt{mult}(x, z) - \texttt{mult}(y, w) \ ,$$

Which means overall we need to perform three smaller multiplications involving the digits and combine the answers with addition and multiplication by 10 and 100. All of these operations are fast.

This means that we can implement multiplication by doing three smaller multiplications and combining the answers with addition and subtraction. This

might seem like too much bother for 2-digit numbers and indeed you are right! However for large numbers with many digits the multiplication needs many of these steps, and many small savings translate into large overall savings.

The details are given in the program bundle for week three.

Dynamic programming

When recursion is used, if care is not taken, the result can be slower than we thought because many results are recalculated. Dynamic programming overcomes this problem by storing answers to subproblems so that they can be reused.

Computing fibonacci numbers

```
int fib(int n) {
 if (n <= 1) return 1;
 else return fib(n-1) + fib(n-2);
}
```

In general fib(2) will need to be computed exponentially-many times in the computation of fib(n). Similarly other values of fib(n) need to be computed many times with this implementation.

The Dynamic Programming solution uses a look up table.

```
int fast_fib(int n) {
  if (n <= 1) return 1;
  else {
     int lookUp[n+1];
     lookUp[0]= 1; lookUp[1]= 1;
     for (int j= 2; j < n+1; j++)
       lookUp[j]= lookUp[j-1] + lookUp[j-2];
  }
 return lookUp[n];
}
```

In this example we can do even better by saving only the last two items, thus we can do without the whole table altogether.

The longest common subsequence

A subsequence of a string is obtained by deleting any number of elements from any positions. A longest common subsequence of s1 and s2 is a subsequence of both whose length is maximal.

The longest common subsequence problem is to find the length of the longest common subsequence. (The algorithm can be adapted to find an actual longest common subsequence.)

The obvious brute force way is as follows, and is very inefficient.

1. Compute all the substrings of s1 and s2;

2. Compare each pair of substrings.

Let L1= length(s1) and L2= length(s2).

There are 2^{L1} substrings of s1, and 2^{L2} substrings of s2. Thus it takes at least 2^{L1+L2} comparisons. (This is exponential in the length of the strings!)

Problem analysis

Let `longest(s1, s2)` stand for a longest common subsequence of s1 and s2 and `lcss(i, j)` be the length of the longest common subsequence of the substrings of s1 and s2 respectively starting at the i'th and j'th characters respectively. We'll try to understand how to analyse this for various cases.

Rule 1: Either s1 or s2 is empty. In this case the longest common subsequence must be empty as well. So we can write:

`longest(s1, s2)= '' ''` and `lcss(i, j)=0`

Rule 2: Both s1 and s2 is begin with the same letter (eg "a", so that s1 = a(ss1) and s2 = a(ss2). This means that the longest common subsequence must begin with (that) letter (in this case "a"). So a way to compute the longest common subsequence is to strike off the first letter from both strings, work out the longest common subsequence of the remaining strings and assemble the answer to the original problem by putting the first letter back.

So we can write:

`longest(s1, s2)= a(longest(ss1,ss2)` ,

and

`lcss(i, j) = 1+ lcss(i+1, j+1)`

where the "a" we have used stands for the first letter.

Rule 3: Strings s1 and s2 begin with different letters (eg s1= a(ss1) and s2= b(ss2). This means that the longest common subsquence can be simplified by either striking out the first letter of s1 OR the first letter of s2, because it can't contain both first letters!

So we can write:

`longest(s1, s2)= EITHER longest(ss1, b(ss2)) OR longest(a(ss1), ss2)` ,

and

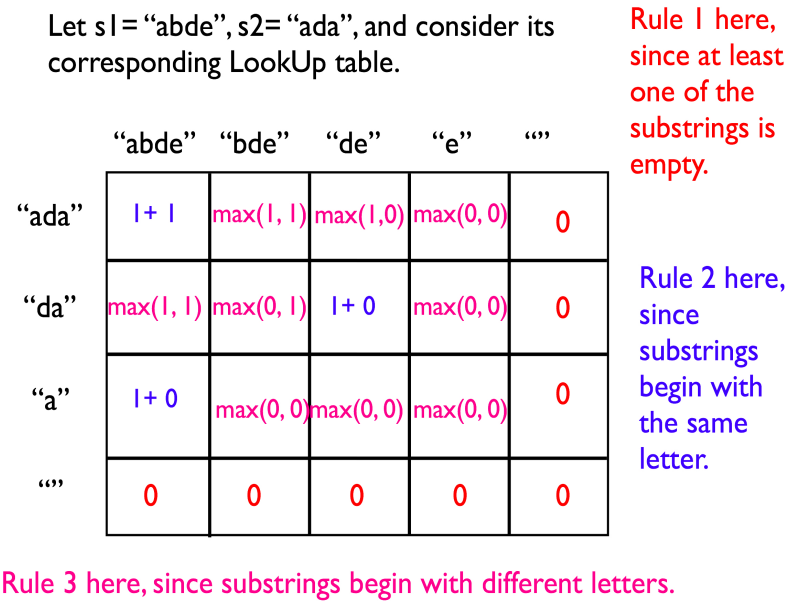
`lcss(i, j) = lcss(i+1, j) MAX lcss(i, j+1)`

i.e. just choose whichever is longest.

This analysis yields a recursive solution, and we note that instead of having to look at all subsequences, we only need to look at subsequences obtained by removing the initial letters, but even so it will still be very innefficient. (Why? Hint: think of our first try for the `fib` function.)

We can improve things by using Dynamic Programming! We'll build a table called `LookUp[i][j]` to store the results of `lcss(i, j)` in case they are needed again. We shall need a 2-dimensional table. (Why?) Here is an example with the strings "abde" and "ada".

Think of the `LookUp` table as being labelled by the various substrings computed by gradually removing the first item. In the table below along the top we see such substrings of "abde" and down the side the corresponding substrings of "ada". In this table we illustrate which Rules we applied at each entry to determine the length of the corresponding pair of substrings.



Based on the analysis above we use the following rules to fill up the table.

Rule 1 If either `i >= s1.length() || j >= s2.length()` (i.e. either substring is empty), then `LookUp[i][j]= 0`.

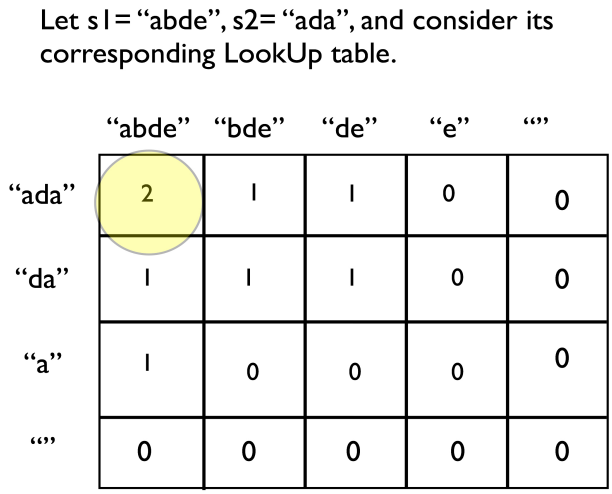
Rule 2 Both substrings begin with the same letter, i.e. `s1.charAt(i) == s2.charAt(j)` then

```
LookUp[i][j]= 1+ LookUp[i+1][j+1]
```

Rule 3 Both substrings begin with different letters, then

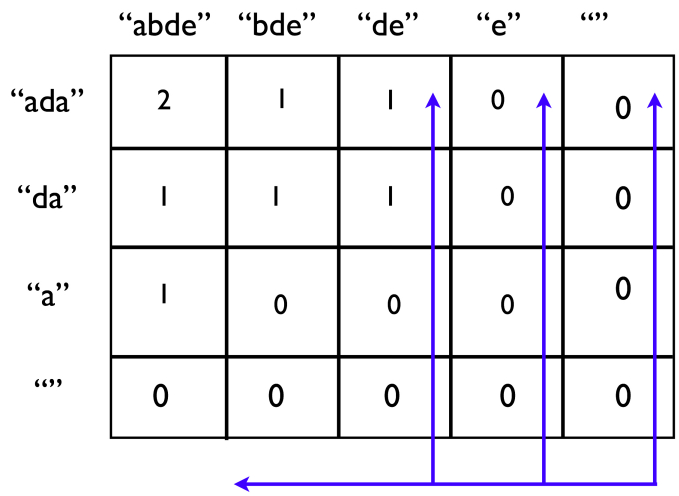
```
LookUp[i][j]= maximum(LookUp[i+1][j] AND LookUp[i][j+1])
```

And after solving the equations, the final table looks like this, where each entry is now the length of the longest common subsequence of the corresponding pair of substrings for each entry. The length of the longest common subsequence of the original strings "abde" and "ada" is situated in the top left corner, but to calculate it we needed to calculate the entire table!



Note that at this point were designing a program `lcss` to compute only the *length* of the longest common subsequence.

We can make one final improvement to simplify the implementation even further. We note that the shape of the rules mean that the `LookUp[i][j]` is computed from `LookUp[i+1][j+1]`, `LookUp[i+1][j]` or `LookUp[i][j+1]`. This means that we can fill up the `LookUp` entries from the bottom left and following the scheme depicted below.



Computing *the* longest common subsequence

Once we've built `LookUp` containing the longest lengths we can find the actual longest common subsequence by applying the above rules this time to the strings.

1 Exercises

1. Trace the `twoDigit` algorithm described in lectures using the numbers 13 and 24. The implemention given in the program bundle can be optimised further using the idea of dynamic programming. Say how and write down the improvement.

   [Note: be careful not to oversimplify — multiplying by 100 is easy but multiplying by 99 is not.]

2. Recall the fast method to raise an integer x to a power n. Observe the following property: if n is odd then $x^n = (x^2)^{n/2} \times x$, and is $(x^2)^{n/2}$ otherwise, where $n/2$ is computed using integer division. Use this to devise a *Divide and Conquer* recursive strategy to implement a fast power function.

   Give a sketch argument to convince me that the worst case complexity requires $O(\log n)$ recursive calls. What other algorithm design strategies can you use to ensure that your design uses the computing resources as efficiently as possible?

   If you want to be really keen, compare experimentally your design with the `fastPower` algorithm from Week 2.

3. Consider the problem of what strategy to take when moving your fingers over a keypad to dial a telephone number. You want to dial using two

fingers, which begin respectively on the $*$ and # keys. The objective is to devise a strategy so that the fingers take the least amount of effort overall for dialling a full n digit number. You may assume that the effort involved in moving one finger from one key to another is proportional to the distance between the two keys, and that a finger will remain hovering over a key until it is moved to the next key. (Use the distance between keys as the Euclidean distance, and round to the nearest 1/2 to make the arithmetic easy. Eg take the distance between keys next to each other as 1 unit if it is a vertically or horizontally next to, and 1.5 units if diagonally i.e. $\sqrt{2}$ rounded to the nearest 1/2.)

As for the longest common subsequence problem, the optimal strategy relies on being able to compute the least amount of distance travelled in dialling. Discuss how you would use dynamic programming to compute the minimal distance travelled. Your lookUp table would normally take into account the remaining digits to be dialled and the current position of the two fingers. Describe how you would calculate the entries.

[Hint: begin with the entries corresponding to no digits remaining to be dialled. Next consider how to calculate the total effort starting at some finger configuration and, say n digits left to dial; how would you calculate that if you already know the least amount of effort for any finger configuration and $n-1$ digits left to dial?]

Sketch a pseudo code algorithm for filling the table, and illustrate your algorithm by filling in the table for the 3 digit number, "165". What strategy is calculated with your method?

4. Professor I. Know tells his class that it is faster to square an n-bit integer than to multiply two n-bit integers. Should they believe him? [Hint: Try computing $(x+y) \times (x+y)$ using as few multiplications as possible.]

5. An array A[1...n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient

algorithm to tell whether the array has a majority element, and, if so, to find that element. The elements of the array are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form is A[i] bigger than A[j]?. (Think of the array elements as GIF files, say.) However you can answer questions of the form: is A[i] = A[j]? in constant time.

Consider splitting the array A into two arrays A_1 and A_2 of half the size. If you know the majority elements of A_1, A_2 and how many there are (in the original array), show how that helps you figure out the majority element of A.

Using this, construct a Divide and Conquer algorithm for finding the majority. Write down the difference equation, assuming that multiplication takes constant time. Use your difference equation to show that this strategy takes $O(n \log n)$ time.

In fact you've seen how to do even better with a problem like this: recall VLSI chip designs.

6. Trace the length of the longest common subsequence for the strings "abcab" and "abbca".

7. Implement a program that computes the longest common subsequence, given that the array for longest lengths has been computed. Make sure you include enough comments in your code so that your tutor is able to understand your strategy.

   [Hint: use the rules.]

8. Use the simplifying strategy outlined in lectures to implement an iterative algorithm for finding the length of the longest common subsequence.

9. Compare experimentally the two algorithms.

10. (Much harder.) Consider the following game played on the given square array, c[i][j].

```
  +---+---+---+---+---+
0 | 6 | 7 | 4 | 7 | 8 |
  +---|---|---|---|---+
1 | 7 | 6 | 1 | 1 | 4 |
  +---|---|---|---|---+
2 | 3 | 5 | 7 | 8 | 2 |
  +---|---|---|---|---+
3 | 2 | 6 | 7 | 0 | 2 |
  +---|---|---|---|---+
4 | 7 | 3 | 5 | 6 | 1 |
  +---+---+---+---+---+
    0   1   2   3   4
```

Each of the numbers in the squares represents an amount of money in dollars. A counter is first placed on the top left square. The player's objective is to move the counter vertically, or diagonally (always downwards) one square per move, in order to reach the bottom row. Thus if he is at position (i,j), then his possible moves are to move the counter to `(i+1, j-1), (i+1,j) or (i+1, j+1)`.

However, there is a snag: every time the counter lands on a square, the money is removed from that square on the board. When he reaches the bottom row, he is allowed to keep the remaining money on the board. Thus his objective is to maximise his winnings, which translates to finding a path whose total worth is as small as possible.

An obvious recursive solution to calculate the minimimum path value is as follows

```
int minCost(int i, int j) {
   if (i== 4)  return c[i][j];

    else
    return min( minCost(i+1, j),
                minCost(i+1, j-1),
                minCost(i+1, j+1)
                ) + c(i, j);
}
```

where "min" is the function that computes the minimum of its inputs. As with other examples we've seen, this is not very efficient, since the same values are recomputed repeatedly.

Use the ideas of dynamic programming to use a table LookUp[i][j], which stores the cost of the minimum path value from position (i, j) on the board to the bottom row `(i.e.  i==4)`. The following hints should help you.

If the current position is `(i,j)` with `(i==4)` (ie the bottom row) then the shortest path from there to the bottom row is `c[i][j]`. (Why?)

11. (Much harder still.) Consider the following problem for finding the longest strictly increasing subsequence. Given an array A of length n having integer values, determine the maximum length of a subsequence (not necessarily contiguous) in which the values in the subsequence form a strictly increasing sequence. (Recall that a subsequence of an array is obtained by "crossing out" elements.) (How many subsequences are there, and what does this tell you about the complexity of the brute-force method for computing this?)

    For example if the array is [2, 4, 3, 5, 3, 7] then the longest strictly increasing subsequence has length 4 (and it is [2, 4, 5,7]).

Now define the array L[] such that L[i] is the length of the longest strictly increasing subsequence A[0..i] which includes the item A[i].

Complete the following table.

```
    i     L[i] Example of the longest strict increasing
                          subsequence in A[0..i] including A[i]
=============================================================
    0      1 [2]
    1      2 [2,4]
    2      2 [2, 3]
    3
    4
    5
```

Write a program to compute L[]

```
    void seq(int A[], int n, int L[]);
    //PRE: A and L are arrays of length n
    //POST: For all j with 0 <= j < n, L[j] is the length
    //          of the longest increasing subsequence in A[0..j]
    //         which includes item A[j]
```

[Hint: If `A[j-1] < A[j]` the maximal subsequence at `j` does not include `A[j-1]`. This means `L[j] = 1+ L[k]` where `k` has `A[k] < A[j]`, and is maximal in `L[0..j-1]`.

Now show how you would use `L[i]` to solve the original problem. What is the overall complexity? State your reasons.

12. Optimal Strategy for a Game. Consider a row of n coins of values v(1) ... v(n), where n is even. We play a game against an opponent by alternating turns. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Sketch the pseudocode for a dynamic programming solution to determine the maximum possible amount of money we can definitely win if we move first. You must describe your lookUp table and explain how you fill it. Give an estimate of its worst case complexity.