Algorithms Example Sheet 2: Further Reading

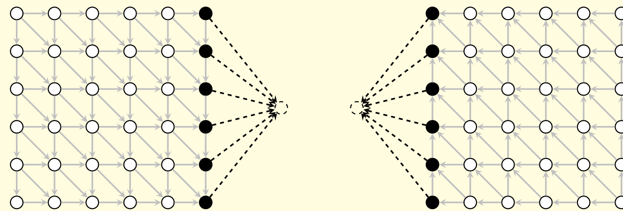
1 The longest common subsequence problem

Further reading: There are some an asymptotically more efficient algorithm which do not seem to be commonly implemented in practice. One such algorithm is the "Four Russian" algorithm which runs in $\mathcal{O}(nm/\log m)$ time and you can read about it here. It is an open problem to find an algorithm that computes the LCS between two strings of length n in $\mathcal{O}(n^{2-\epsilon})$ time for constant $\epsilon > 0$ (i.e. an asymptotically polynomial change instead of logarithmic). More recently, it has been shown that solving this problem in $\mathcal{O}(n^{2-\epsilon})$ time, would imply faster algorithms for a collection of problems (see for example here).

Project 1 [Hirschberg's algorithm] In this project, you will investigate a space efficient algorithm for recovering the LCS of two strings in $\mathcal{O}(\min(n,m))$ space using Hirschberg's algorithm. Read through these slides and answer the following questions:

(a) How can the LCS problem be interpreted as a longest path problem in a matrix?

(b) With the aid of the following diagram explain how you can compute the optimal in the column at $n/2$ by solving two instances of LCS for a string of length $n/2$ and a string of length m using linear memory.



(c) If you know the optimal path passes through $(n/2, x)$, how can you find the remaining path, by processing a matrix of total $mn/2$ size (and still using linear memory).

(d) Argue that the time complexity of the algorithm is $\mathcal{O}(mn)$ and the space complexity is $\mathcal{O}(\min(n,m))$. **Hint:** *Consider the sum* $mn + mn/2 + mn/4 + \ldots$.

Some other directions/variants for the LCS problem:

- Approximation algorithms for the LCS problem (e.g. here).
- LCS in the streaming setting (e.g. here).
- Beyond worst-case analysis for LCS problems (e.g. here).
- Analysis of the LCS problem in randomised setting (e.g. here).
- Multi-variate fine grained analysis (e.g. here).
- Counting the number of different LCS (e.g. here).

2 Longest Common Substring

Longest common substring can be efficiently solved using a suffix array in $\mathcal{O}(n \log n)$ time or using a suffix tree in $\mathcal{O}(n)$ time. You will learn more about these approaches in Part II Bioinformatics.

3 Longest palindromic substring

There is a more efficient algorithm for finding the longest palindromic substring in a given string (and actually any palindromic subtring). This algorithm is known as Manacher's algorithm [?] and you can read more about it here.

4 Speeding up dynamic programming

Project 2 [Faster DP algorithms for special cases] There are several techniques for speeding up Dynamic Programming algorithms. We have already seen some of these (e.g. matrix exponentiation to compute the Fibonacci numbers). Some more are described in this chapter (and also in these blog posts: here and here. In this project, you can pick one of these techniques and one example problem and describe: (a) the problem to be solved, (b) the normal DP algorithm and its complexity, (c) the technique to be applied, (d) the resulting algorithm and its complexity.

Project 3 [Faster subset sum] Recently, there has been a series of works that improve the running time for solving the subset sum problem. You can read more about these in this paper.

More divide and conquer

Project 4 [Karatsuba's multiplication algorithm] In this project, you will investigate an algorithm for integer multiplication that is asymptotically faster than the algorithm commonly taught at school. To simplify the analysis, we will assume that we want to multiply to n-bit positive integers with $n = 2^k$ (if it is not we can pad 0s in the beginning.

(a) (Warm-up) Show that classical method takes $\mathcal{O}\left(n^2\right)$ time.

(b) Argue that multiplication of an n-bit number by 2^ℓ takes $\Theta(\ell + n)$ time.

(c) Let $x = x_1 \cdot 2^{n/2} + x_0$ where x_1 and x_0 are two $n/2$-bit numbers. Then show that for $n > 1$,

$$x \cdot y = (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) = x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{n/2} + x_0 y_0.$$

(d) Show that a divide and conquer algorithm that performs 4 multiplications of $n/2$-bit numbers, takes $\mathcal{O}\left(n^2\right)$ time (i.e. no improvement over the classical method). *Hint:* It may be helpful to write out the recurrence relation)

(e) Show how $(x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0)$ can be computed using only 3 multiplications of roughly $n/2$-bit sized integers. *Hint:* Consider adding (or subtracting) two numbers before multiplying.

(f) (+) Solve the recurrence relation to show that the new divide and conquer algorithm performs $\Theta(n^{\log_2(3)})$. (*Hint:* If you get stuck, feel free to use the Master Theorem)

If you get stuck, you can consult Vazirani et al (section 2.1) or wikipedia.
Note: A similar technique can be applied for faster matrix multiplication, which is known as Strassen's algorithm.

Project 5 [Toom-Cook's multiplication algorithm] (++) (Only if you have completed Project 4) Read about Toom-Cook's multiplication method for getting $\mathcal{O}\left(n^{1+\epsilon}\right)$ time algorithm.

Project 6 [Fast Fourier Transform] One of the major applications of divide and conquer is in computing the Fast Fourier Transform. This algorithm is used in many applications, including fast integer multiplication, fast polynomial multiplication, string matching, in signal processing, in Computer Vision and more. Read about divide and conquer algorithm for computing the FFT from the Information Theory notes, Chapter 30 in CLRS3, or Section 2.6 in this book.

Furher reading on the $\sqrt{n}$ approach

The approach of splitting an array in $\sqrt{n}$ groups can be generalised to support several operations (such as range updates, range queries for different operations) as well as operations on trees and graphs. You can read about some of them here or here or here.

Further reading on the segment tree

Segment trees can be used to solve a lot of interesting problems:

- Can be generalised for other point updates/range queries, such as finding the min, gcd and more. What kind of condition should the function satisfy?
- Can be generalised for range updates/range queries for sum. Look at the lazy propagation technique.
- Can be made persistent, i.e. remember all previous states of the array (you get this for free if you implement segment trees in OCaml).
- Can be generalised to higher dimensions (see here).

You can read more about segment trees in following articles: [segment tree algorithm](https://www.topcoder.com/community/co programming/tutorials/range-minimum-query-and-lowest-common-ancestor/#Segment_Trees) for efficiently retrieving the sum of a subarray (and also updating an entry in the array).

Partial sums

If you are interested in the theoretical optimal times for answering the partial sum problem look at Chapter 4.3 from Patrascu's thesis or this more recent work.