☺😐☹ **MUHAMMAD FAISAL** ☺😐☹

MIT 4th Semester

Al-Barq Campus (VGJW01) Gujranwala

faisalgrw123@gmail.com

Reference Short Questions for MID TERM EXAMS

CS502– Design and Analysis of Algorithms

Divide and Conquer Strategy (Page#27)

The ancient Roman politicians understood an important principle of good algorithm design (although they were probably not thinking about algorithms at the time). You divide your enemies (by getting them to distrust each other) and then conquer them piece by piece. This is called divide-and-conquer. In algorithm design, the idea is to take a problem on a large input, break the input into smaller pieces, solve the problem on each of the small pieces, and then combine the piecewise solutions into a global solution. But once you have broken the problem into pieces, how do you solve these pieces? The answer is to apply divide-and-conquer to them, thus further breaking them down. The process ends when you are left with such tiny pieces remaining (e.g. one or two items) that it is trivial to solve them. Summarizing, the main elements to a divide-and-conquer solution are

Divide: the problem into a small number of pieces
Conquer: solve each piece by applying divide and conquer to it recursively
Combine: the pieces together into a global solution.

Selection Problem (Page#34)

Suppose we are given a set of n numbers. Define the rank of an element to be one plus the number of elements that are smaller. Thus, the rank of an element is its final position if the set is sorted. The minimum is of rank 1 and the maximum is of rank n.

Consider the set: {5, 7, 2, 10, 8, 15, 21, 37, 41}. The rank of each number is its position in the sorted order.

position	1	2	3	4	5	6	7	8	9
Number	2	5	7	8	10	15	21	37	41

Sieve Technique (Page#34)

The reason for introducing this algorithm is that it illustrates a very important special case of divide-and-conquer, which I call the sieve technique. We think of divide-and-conquer as breaking the problem into a small number of smaller subproblems, which are then solved recursively. The sieve technique is a special case, where the number of subproblems is just 1.

The sieve technique works in phases as follows. It applies to problems where we are interested in finding a single item from a larger set of n items.

Slow Sorting Algorithms (Page#39)

There are a number of well-known slow $O(n^2)$ sorting algorithms. These include the following:

Bubble sort: Scan the array. Whenever two consecutive items are found that are out of order, swap them. Repeat until all consecutive items are in order.

Insertion sort: Assume that A[1..i − 1] have already been sorted. Insert A[i] into its proper position in this sub array. Create this position by shifting all larger elements to the right.

Selection sort: Assume that A[1..i − 1] contain the i − 1 smallest elements in sorted order. Find the smallest element in A[i..n] Swap it with A[i].

Heaps (Page#40)

A heap is a left-complete binary tree that conforms to the heap order. The heap order property: in a (min) heap, for every node X, the key in the parent is smaller than or equal to the key in X. In other words, the parent node has key smaller than or equal to both of its children nodes. Similarly, in a max heap, the parent has a key larger than or equal both of its children Thus the smallest key is in the root in a min heap; in the max heap, the largest is in the root.

Quicksort (Page#46)

Our next sorting algorithm is Quicksort. It is one of the fastest sorting algorithms known and is the method of choice in most sorting libraries. Quicksort is based on the divide and conquer strategy.

Here is the algorithm:

```
QUICKSORT( array A, int p, int r)
1  if (r > p)
2     then
3          i ← a random index from [p..r]
4          swap A[i] with A[p]
5          q ← PARTITION(A, p, r)
6          QUICKSORT(A, p, q − 1)
7          QUICKSORT(A, q + 1, r)
```

In-place, Stable Sorting (Page#46)

An in-place sorting algorithm is one that uses no additional array for storage. A sorting algorithm is stable if duplicate elements remain in the same relative position after sorting.

9 | 3 | 3′ | 5 | 6 | 5′ | 2 | 1 | 3″	unsorted
1 | 2 | 3 | 3′ | 3″ | 5 | 5′ | 6 | 9	stable sort
1 | 2 | 3′ | 3 | 3″ | 5′ | 5 | 6 | 9	unstable

Bubble sort, insertion sort and selection sort are in-place sorting algorithms. Bubble sort and insertion sort can be implemented as stable algorithms but selection sort cannot (without significant modifications). Mergesort is a stable algorithm but not an in-place algorithm. It requires extra array storage. Quicksort is not stable but is an in-place algorithm. Heapsort is an in-place algorithm but is not stable.

Counting Sort (Page#57)

We will consider three algorithms that are faster and work by not making comparisons. Counting sort assumes that the numbers to be sorted are in the range 1 to k where k is small. The basic idea is to determine the rank of each number in final sorted array.

Recall that the rank of an item is the number of elements that are less than or equal to it. Once we know the ranks, we simply copy numbers to their final position in an output array.

Fibonacci Sequence (Page#73)

Suppose we put a pair of rabbits in a place surrounded on all sides by a wall. How many pairs of rabbits can be produced from that pair in a year if it is supposed that every month each pair begets a new pair which from the second month on becomes productive? Resulting sequence is 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, . . . where each number is the sum of the two preceding numbers.

This problem was posed by Leonardo Pisano, better known by his nickname Fibonacci (son of Bonacci, born 1170, died 1250). This problem and many others were in posed in his book Liber abaci, published in 1202. The book was based on the arithmetic and algebra that Fibonacci had accumulated during his travels. The book, which went on to be widely copied and imitated, introduced the Hindu-Arabic place-valued decimal system and the use of Arabic numerals into Europe. The rabbits problem in the third section of Liber abaci led to the introduction of the Fibonacci numbers and the Fibonacci sequence for which Fibonacci is best remembered today.

This sequence, in which each number is the sum of the two preceding numbers, has proved extremely fruitful and appears in many different areas of mathematics and science. The Fibonacci Quarterly is a, modern journal devoted to studying mathematics related to this sequence. The Fibonacci numbers Fn are defined as follows:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

Figure shows four levels of recursion for the call fib(8):

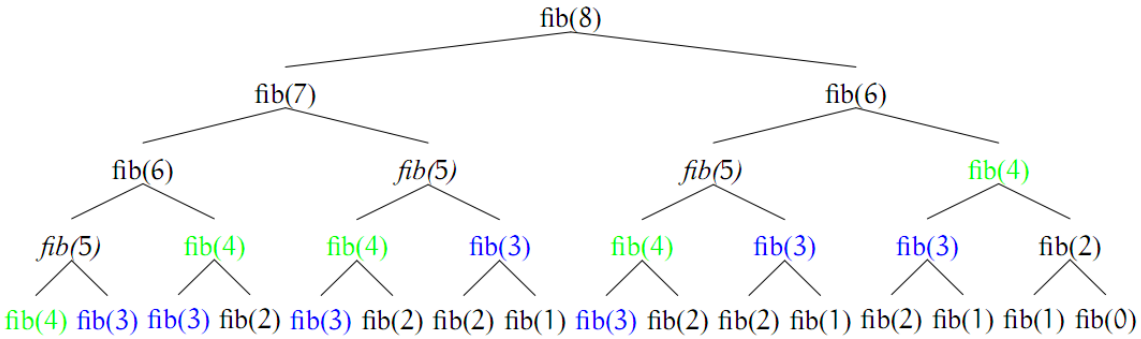


Figure 6.1: Recursive calls during computation of Fibonacci number

Dynamic Programming (Page#75)

Dynamic programming is essentially recursion without repetition. Developing a dynamic programming algorithm generally involves two separate steps:

• **Formulate problem recursively.** Write down a formula for the whole problem as a simple combination of answers to smaller subproblems.

• **Build solution to recurrence from bottom up.** Write an algorithm that starts with base cases and works its way up to the final solution.

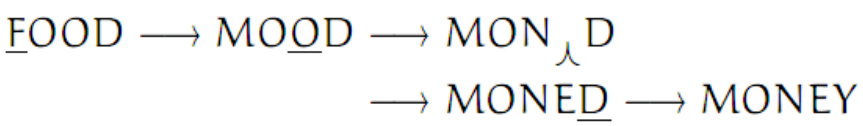
Dynamic programming algorithms need to store the results of intermediate subproblems. This is often but not always done with some kind of table. We will now cover a number of examples of problems in which the solution is based on dynamic programming strategy.

Edit Distance (Page#75)

The words “computer” and “commuter” are very similar, and a change of just one letter, p-¿m, will change the first word into the second. The word “sport” can be changed into “sort” by the deletion of the ‘p’, or equivalently, ‘sort’ can be changed into ‘sport’ by the insertion of ‘p’. The edit distance of two strings, s1 and s2, is defined as the minimum number of point mutations required to change s1 into s2, where a point mutation is one of:

- change a letter,
- insert a letter or
- delete a letter

For example, the edit distance between FOOD and MONEY is at most four:



Edit Distance: Applications (Page#76)

There are numerous applications of the Edit Distance algorithm.

Here are some examples:

1. Spelling Correction
2. Plagiarism Detection
3. Computational Molecular Biology
4. Speech Recognition

Edit Distance Algorithm (Page#77)

A better way to display this editing process is to place the words above the other:

S	*D*	*I*	*M*	*D*	*M*
M	A	_	T	H	S
A	_	R	T	_	S

The first word has a gap for every insertion (I) and the second word has a gap for every deletion (D). Columns with two different characters correspond to substitutions (S). Matches (M) do not count. The Edit transcript is defined as a string over the alphabet M, S, I, D that describes a transformation of one string into another.

In general, it is not easy to determine the optimal edit distance. For example, the distance between ALGORITHM and ALTRUISTIC is at most 6.

A	L	G	O	R	_	I	_	T	H	M
A	L	_	T	R	U	I	S	T	I	C

CHAIN MATRIX MULTIPLY (Page#85)

There is considerable savings achieved even for this simple example. In general, however, in what order should we multiply a series of matrices A1A2 . . .An. Matrix multiplication is an associative but not commutative operation. We are free to add parenthesis the above multiplication but the order of matrices can not be changed. The Chain Matrix Multiplication Problem is stated as follows:

Given a sequence A1,A2, . . . ,An and dimensions p0, p1, . . . , pn where Ai is of dimension $p_{i-1} \times p_i$, determine the order of multiplication that minimizes the number of operations.

We could write a procedure that tries all possible parenthesizations. Unfortunately, the number of ways of parenthesizing an expression is very large. If there are n items, there are $n-1$ ways in which outer most pair of parentheses can placed.

$$
\begin{array}{ll}
 & (A_1)(A_2A_3A_4\ldots A_n) \\
\text{or} & (A_1A_2)(A_3A_4\ldots A_n) \\
\text{or} & (A_1A_2A_3)(A_4\ldots A_n) \\
\ldots & \ldots \\
\text{or} & (A_1A_2A_3A_4\ldots A_{n-1})(A_n)
\end{array}
$$

We do not want to calculate m entries recursively. So how should we proceed? We will fill m along the diagonals. Here is how. Set all $m[i,i]=0$ using the base condition. Compute cost for multiplication of a sequence of 2 matrices. These are $m[1,2], m[2,3], m[3,4], \ldots, m[n-1,n]$. $m[1,2]$, for example is

$$m[1,2] = m[1,1] + m[2,2] + p_0 \cdot p_1 \cdot p_2$$

For example, for m for product of 5 matrices at this stage would be:

$m[1,1]$	$\leftarrow m[1,2]$ $\downarrow$			
	$m[2,2]$	$\leftarrow m[2,3]$ $\downarrow$		
		$m[3,3]$	$\leftarrow m[3,4]$ $\downarrow$	
			$m[4,4]$	$\leftarrow m[4,5]$ $\downarrow$
				$m[5,5]$

Knapsack Problem (Page#91)

A thief goes into a jewelry store to steal jewelry items. He has a knapsack (a bag) that he would like to fill up. The bag has a limit on the total weight of the objects placed in it. If the total weight exceeds the limit, the bag would tear open. The value of of the jewelry items varies for cheap to expensive. The thief's goal is to put items in the bag such that the value of the items is maximized and the weight of the items does not exceed the weight limit of the bag. Another limitation is that an item can either be put in the bag or not - fractional items are not allowed. The problem is: what jewelry should the thief choose that satisfy the constraints?

Formally, the problem can be stated as follows: Given a knapsack with maximum capacity W, and a set S consisting of n items Each item i has some weight w_i and value value v_i (all w_i , v_i and W are integer values) How to pack the knapsack to achieve maximum total value of packed items? For example, consider the following scenario:

Item i	Weight w_i	Value v_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

Figure 6.3: Knapsack can hold $W = 20$

The knapsack problem belongs to the domain of optimization problems. Mathematically, the problem is

$$\text{maximize} \sum_{i \in T} v_i$$

$$\text{subject to} \sum_{i \in T} w_i \leq W$$

The **steps** in the **dynamic programming strategy**: (Page#92)

1) Simple Subproblems
2) Principle of Optimality
3) Bottom-up computation

Construction of optimal solution

Knapsack algorithm (Page#96)

```
KNAPSACK(n, W)
 1  for w = 0, W
 2  do V[0, w] ← 0
 3  for i = 0, n
 4  do V[i, 0] ← 0
 5     for w = 0, W
 6     do if (w_i ≤ w & v_i + V[i − 1, w − w_i] > V[i − 1, w])
 7           then V[i, w] ← v_i + V[i − 1, w − w_i]; keep[i, w] ← 1
 8           else V[i, w] ← V[i − 1, w]; keep[i, w] ← 0
 9  // output the selected items
10  k ← W
11  for i = n downto 1
12  do if (keep[i, k] = 1)
13        then output i
14             k ← k − w_i
```

----------Wish U Best of L|U|C|K for EXAMS ----------