0/1 Knapsack DP

```c
#include <stdio.h>
#include <limits.h>

void knapsack(int n, int m, int profits[], int weights[])
{
    int v[n+1][m+1];
    int result[n];

    for(int i=0; i<=m; i++)
        v[0][i] = 0;
    for(int i=0; i<=n; i++)
        v[i][0] = 0;

    for(int i=1; i<=n; i++) {
        for(int w=1; w<=m; w++) {
            if(w - weights[i-1] <0) {
                v[i][w] = v[i-1][w];
            } else {
                int val1 = v[i-1][w];
                int val2 = v[i-1][w-weights[i-1]] + profits[i-1];
                v[i][w] = (val1 > val2) ? val1 : val2;
            }
        }
    }

    int k=n, j=m;
    while (k>0 && j>0) {
        if(v[k][j] != v[k-1][j]) {
            result[k-1] = 1;
            j = j-weights[k-1];
        }
        else {
            result[k-1] = 0;
        }
        k--;
    }

    for(int i=0; i<n; i++)
        printf("%d ", result[i]);
}

int main()
{
    int n;              //No. of objects
    scanf("%d", &n);
    int m;              //Max capacity
    scanf("%d", &m);
```

```
    int profits[n], weights[n];

    for(int i=0; i<n; i++)
        scanf("%d", &profits[i]);
    for(int i=0; i<n; i++)
        scanf("%d", &weights[i]);

    knapsack(n, m, profits, weights);

    return 0;
}
```

LCS

```
#include <stdio.h>
#include <string.h>

char S1[20], S2[20];
int i, j, m, n, LCS_table[20][20];

int max(int a, int b) {
    return (a>b) ? a: b;
}


void lcsAlgo()
{
    printf("Reached\n");
    m = strlen(S1);
    n = strlen(S2);

    for(i = 0; i<=m; i++)
        LCS_table[i][0] = 0;
    for(i=0; i<=n; i++)
        LCS_table[0][i] = 0;

    for(i=1; i<=m; i++) {
        for(j=1; j<=n; j++) {
            if(S1[i-1] == S2[j-1])
                LCS_table[i][j] = LCS_table[i-1][j-1] + 1;
            else
                LCS_table[i][j] = max(LCS_table[i-1][j], LCS_table[i][j-1]);
        }
    }
```

```
    int index = LCS_table[m][n];
    char LCSAlgo[index+1];
    LCSAlgo[index] = '\0';

    printf("%d\n", index);
    i=m, j=n;
    while(i>0 && j>0)
    {
        if(S1[i-1] == S2[j-1]) {
            LCSAlgo[index-1] = S1[i-1];
            i--;
            j--;
            index--;
        } else {
            //max(LCS_table[i-1][j], LCS_table[i][j-1]);
            if(LCS_table[i - 1][j] > LCS_table[i][j - 1])
                i--;
            else
                j--;
        }
    }
    printf("%d\n", index);
    printf("The LCS is %s", LCSAlgo);
}

int main()
{
    scanf("%[^\n]s", &S1);
    scanf("%s", &S2);

    lcsAlgo();
}
```

Matrix Chain

```
#include <stdio.h>
#include <limits.h>

int MatrixChainMultiplication(int arr[], int n)
{
    int minMul[n][n];

    for(int i=1; i<n; i++)
        minMul[i][i] = 0;
    int j, q=0;

    for(int L=2; L<n; L++) {
```

```
        for(int i=1; i<n-L+1; i++) {
            j = i+L-1;
            minMul[i][j] = INT_MAX;
            for(int k=i; k<=j; k++) {
                q = minMul[i][k] + minMul[k+1][j] + arr[i-1]*arr[k]*arr[j];
                if(q<minMul[i][j])
                    minMul[i][j]=q;
            }
        }
    }

    return minMul[1][n-1];
}

int main()
{
    int n;
    scanf("%d", &n);

    int arrS[n][2];
    int arr[n+1];

    for(int i=0; i<n; i++) {
        scanf("%d %d",&arrS[i][0], &arrS[i][1]);
    }

    arr[0] = arrS[0][0];
    for(int i=1; i<=n; i++)
        arr[i] = arrS[i-1][1];

    printf("Minimum number of multiplications required for matrix
multiplication is %d\n", MatrixChainMultiplication(arr, n+1));
    getchar();
    return 0;
}
```

Assembly line scheduling

Code:

```
#include <stdio.h>
#include <stdlib.h>

#define min(a, b) ((a) < (b) ? (a) : (b))

int fun(int a[2][4], int t[2][4], int cl, int cs, int x1, int x2, int n);

int main() {
   int n = 4; // number of stations
   int a[2][4] = { { 4, 5, 3, 2 }, { 2, 10, 1, 4 } };
   int t[2][4] = { { 0, 7, 4, 5 }, { 0, 9, 2, 8 } };

   int e1 = 10;
   int e2 = 12;
   int x1 = 18;
   int x2 = 7;

   // entry from 1st line
   int x = fun(a, t, 0, 0, x1, x2, n) + e1 + a[0][0];
   // entry from 2nd line
   int y = fun(a, t, 1, 0, x1, x2, n) + e2 + a[1][0];
```

```
    printf("%d\n", min(x, y));

    return 0;
}

int fun(int a[2][4], int t[2][4], int cl, int cs, int x1, int x2, int n) {
    // base case
    if (cs == n - 1) {
        if (cl == 0) { // exiting from (current) line =0
            return x1;
        } else { // exiting from line 2
            return x2;
        }
    }

    // continue on same line
    int same = fun(a, t, cl, cs + 1, x1, x2, n) + a[cl][cs + 1];
    // continue on different line
    int diff = fun(a, t, !cl, cs + 1, x1, x2, n) + a[!cl][cs + 1] + t[cl][cs + 1];

    return min(same, diff);
}
```

Max Sub Array Sum

```c
#include <stdio.h>
#include <limits.h>

int *maxSubArraySum(int nums[], int start, int end, int *max_sum, int
*subarray_start, int *subarray_end)
{
    if(start == end) {
        *max_sum = nums[start];
        *subarray_start = start;
        *subarray_end = start;
        return &nums[start];
    }

    int mid= (start+end)/2;
    int left_subarray = maxSubArraySum(nums, start, mid, &max_sum,
&subarray_start, &subarray_end);
    int right_subarray = maxSubArraySum(nums, mid+1, end, &max_sum,
&subarray_start, &subarray_end);

    int left_sum=INT_MIN;
    int curr_sum = 0;
    int cross_start = mid;
    for(int i=mid; i>=start; i--) {
        curr_sum += nums[i];
        if(curr_sum >= left_sum) {
            left_sum = curr_sum;
            cross_start = i;
        }
    }

    int right_sum=INT_MIN;
    curr_sum=0;
    int cross_end = mid+1;
    for(int i=mid+1; i<end; i++) {
        curr_sum+=nums[i];
        if(curr_sum>= right_sum) {
            right_sum = curr_sum;
            cross_end = i;
        }
    }
}

int main()
{
    int n;
    scanf("%d", &n);
    int nums[n];
```

```
    for(int i=0; i<n; i++)
        scanf("%d ", &nums[i]);
    int max_sum, subarray_start, subarray_end;
    int *max_subarray =
        maxSubArraySum(nums, 0, n-1, &max_sum, &subarray_start,
&subarray_end);

    printf("The sub array [");
    for(int i=subarray_start; i<=subarray_end; i++) {
        if(i==subarray_end)
            printf("%d", nums[i]);
        else
            printf("%d ", nums[i]);
    }
    printf("] has the largest sum %d", max_sum);

    return 0;
}
```

KARATSUBA ALGORITHM

```
#include <stdio.h>
#include <string.h>

// Function to find the sum of larger
// numbers represented as a string
char* findSum(const char* str1, const char* str2)
{
	// Before proceeding further, make
	// sure length of str2 is larger
	if (strlen(str1) > strlen(str2)) {
		const char* temp = str1;
		str1 = str2;
		str2 = temp;
	}

	// Stores the result
	static char str[1000];

	// Calculate length of both strings
	int n1 = strlen(str1);
	int n2 = strlen(str2);

	// Reverse both of strings
	strrev(str1);
	strrev(str2);

	int carry = 0;
```

```
for (int i = 0; i < n1; i++) {

        // Find the sum of the current
        // digits and carry
        int sum
                = ((str1[i] - '0')
                + (str2[i] - '0')
                + carry);
        str[i] = (sum % 10) + '0';

        // Calculate carry for next step
        carry = sum / 10;
}

// Add remaining digits of larger number
for (int i = n1; i < n2; i++) {
        int sum = ((str2[i] - '0') + carry);
        str[i] = (sum % 10) + '0';
        carry = sum / 10;
}

// Add remaining carry
if (carry)
        str[n2] = carry + '0';
else
        str[n2] = '\0';

// Reverse resultant string
strrev(str);
```

```
        return str;
}

// Function to find difference of larger
// numbers represented as strings
char* findDiff(const char* str1, const char* str2)
{
        // Stores the result of difference
        static char str[1000];

        // Calculate length of both strings
        int n1 = strlen(str1), n2 = strlen(str2);

        // Reverse both of strings
        strrev(str1);
        strrev(str2);

        int carry = 0;

        // Run loop till small string length
        // and subtract digit of str1 to str2
        for (int i = 0; i < n2; i++) {

                // Compute difference of the
                // current digits
                int sub
                        = ((str1[i] - '0')
                        - (str2[i] - '0')
```

```
                - carry);

        // If subtraction < 0 then add 10
        // into sub and take carry as 1
        if (sub < 0) {
                sub = sub + 10;
                carry = 1;
        }
        else
                carry = 0;

        str[i] = sub + '0';
}

// Subtract the remaining digits of
// larger number
for (int i = n2; i < n1; i++) {
        int sub = ((str1[i] - '0') - carry);

        // If the sub value is -ve,
        // then make it positive
        if (sub < 0) {
                sub = sub + 10;
                carry = 1;
        }
        else
                carry = 0;

        str[i] = sub + '0';
```

```
	}

	// Reverse resultant string
	strrev(str);

	// Return answer
	return str;
}

// Function to remove all leading 0s
// from a given string
char* removeLeadingZeros(char* str)
{
	// Index to store the position of
	// the first non-zero digit
	int i, len = strlen(str);
	for (i = 0; i < len - 1; i++) {
		if (str[i] != '0')
			break;
	}

	// Shift all non-zero digits to the
	// beginning of the string
	for (int j = 0; j < len - i; j++)
		str[j] = str[i + j];

	// Null terminate the string
	str[len - i] = '\0';
```

```
        return str;
}

// Function to multiply two numbers
// using Karatsuba algorithm
char* multiply(const char* A, const char* B)
{
        const char* str1 = A;
        const char* str2 = B;

        if (strlen(A) > strlen(B)) {
                str1 = B;
                str2 = A;
        }

        // Make both numbers to have
        // same digits
        int n1 = strlen(str1), n2 = strlen(str2);
        while (n2 > n1) {
                n1++;
        }

        // Base case
        if (n1 == 1) {

                // If the length of strings is 1,
                // then return their product
                int ans = atoi(str1) * atoi(str2);
                sprintf(str1, "%d", ans);
```

```
        return str1;
}

// Add zeros in the beginning of
// the strings when length is odd
if (n1 % 2 == 1) {
        n1++;
        char temp[1000];
        strcpy(temp, str1);
        strcpy(str1, "0");
        strcat(str1, temp);
        strcpy(temp, str2);
        strcpy(str2, "0");
        strcat(str2, temp);
}

char Al[1000], Ar[1000], Bl[1000], Br[1000];

// Find the values of Al, Ar,
// Bl, and Br.
for (int i = 0; i < n1 / 2; ++i) {
        Al[i] = str1[i];
        Bl[i] = str2[i];
        Ar[i] = str1[n1 / 2 + i];
        Br[i] = str2[n1 / 2 + i];
}
Al[n1 / 2] = '\0';
Ar[n1 / 2] = '\0';
Bl[n1 / 2] = '\0';
```

```
Br[n1 / 2] = '\0';

// Recursively call the function
// to compute smaller product

// Stores the value of Al * Bl
char* p = multiply(Al, Bl);

// Stores the value of Ar * Br
char* q = multiply(Ar, Br);

// Stores value of ((Al + Ar)*(Bl + Br)
// - Al*Bl - Ar*Br)
char* r = findDiff(
        findSum(Al, Ar),
        findSum(Bl, Br));

// Multiply p by 10^n
for (int i = 0; i < n1; ++i)
        strcat(p, "0");

// Multiply s by 10^(n/2)
for (int i = 0; i < n1 / 2; ++i)
        strcat(r, "0");

// Calculate final answer p + r + s
char* ans = findSum(p, findSum(q, r));
```

Subset Sum

```c
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>

bool foundSolution = false;
int weights[10];
int solution[10];
int n;
int m;

void subsetSum(int currentSum, int index)
{
    if(index == n) {
        if(currentSum == m) {
            foundSolution = true;
            for(int i=0; i<n; i++)
                printf("%d ", solution[i]);
            printf("\n");
        }
        return;
    }

    solution[index] = 1;
    subsetSum(currentSum + weights[index], index+1);

    solution[index] = 0;
    subsetSum(currentSum, index + 1);

}


int main()
{
    //int n, m;
    scanf("%d", &n);
    scanf("%d", &m);

    //int weights[n];

    for(int i=0; i<n; i++)
        scanf("%d", &weights[i]);


    subsetSum(0, 0);

    if(!foundSolution)
        printf("No solution found.\n");
```

```
    return 0;
}
```

FRACTIONAL KNAPSACK GREEDY APPROACH

```
#include <stdio.h>
#include <stdlib.h>

// Structure for an item which stores weight and
// corresponding value of Item
struct Item {
        int profit, weight;
};

// Comparison function to sort Item
// according to profit/weight ratio
static int cmp(const void* a, const void* b)
{
        double r1 = (double)((struct Item*)b)->profit / (double)((struct Item*)b)->weight;
        double r2 = (double)((struct Item*)a)->profit / (double)((struct Item*)a)->weight;
        if (r1 > r2) return 1;
        else if (r1 < r2) return -1;
        return 0;
}

// Main greedy function to solve problem
double fractionalKnapsack(int W, struct Item arr[], int N)
```

```
{
        // Sorting Item on basis of ratio
        qsort(arr, N, sizeof(struct Item), cmp);

        double finalvalue = 0.0;

        // Looping through all items
        for (int i = 0; i < N; i++) {

                // If adding Item won't overflow,
                // add it completely
                if (arr[i].weight <= W) {
                        W -= arr[i].weight;
                        finalvalue += arr[i].profit;
                }

                // If we can't add current Item,
                // add fractional part of it
                else {
                        finalvalue
                                += arr[i].profit
                                * ((double)W / (double)arr[i].weight);
                        break;
                }
        }

        // Returning final value
        return finalvalue;
}
```

```
// Driver code
int main()
{
	int W = 50;
	struct Item arr[] = { { 60, 10 }, { 100, 20 }, { 120, 30 } };
	int N = sizeof(arr) / sizeof(arr[0]);

	// Function call
	printf("%.2lf\n", fractionalKnapsack(W, arr, N));
	return 0;
}
```

N QUENS

CODE :

```
// C program to solve N Queen Problem using backtracking

#define N 4
#include <stdbool.h>
#include <stdio.h>

// A utility function to print solution
void printSolution(int board[N][N])
{
	for (int i = 0; i < N; i++) {
		for (int j = 0; j < N; j++) {
			if(board[i][j])
				printf("Q ");
			else
				printf(". ");
		}
		printf("\n");
	}
}

// A utility function to check if a queen can
// be placed on board[row][col]. Note that this
// function is called when "col" queens are
// already placed in columns from 0 to col -1.
// So we need to check only left side for
```

```
// attacking queens
bool isSafe(int board[N][N], int row, int col)
{
	int i, j;

	// Check this row on left side
	for (i = 0; i < col; i++)
		if (board[row][i])
			return false;

	// Check upper diagonal on left side
	for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
		if (board[i][j])
			return false;

	// Check lower diagonal on left side
	for (i = row, j = col; j >= 0 && i < N; i++, j--)
		if (board[i][j])
			return false;

	return true;
}

// A recursive utility function to solve N
// Queen problem
bool solveNQUtil(int board[N][N], int col)
{
	// Base case: If all queens are placed
	// then return true
```

```
        if (col >= N)
                return true;

        // Consider this column and try placing
        // this queen in all rows one by one
        for (int i = 0; i < N; i++) {

                // Check if the queen can be placed on
                // board[i][col]
                if (isSafe(board, i, col)) {

                        // Place this queen in board[i][col]
                        board[i][col] = 1;

                        // Recur to place rest of the queens
                        if (solveNQUtil(board, col + 1))
                                return true;

                        // If placing queen in board[i][col]
                        // doesn't lead to a solution, then
                        // remove queen from board[i][col]
                        board[i][col] = 0; // BACKTRACK
                }
        }

        // If the queen cannot be placed in any row in
        // this column col then return false
        return false;
}
```

```
// This function solves the N Queen problem using
// Backtracking. It mainly uses solveNQUtil() to
// solve the problem. It returns false if queens
// cannot be placed, otherwise, return true and
// prints placement of queens in the form of 1s.
// Please note that there may be more than one
// solutions, this function prints one of the
// feasible solutions.
bool solveNQ()
{
	int board[N][N] = { { 0, 0, 0, 0 },
	                    { 0, 0, 0, 0 },
	                    { 0, 0, 0, 0 },
	                    { 0, 0, 0, 0 } };

	if (solveNQUtil(board, 0) == false) {
		printf("Solution does not exist");
		return false;
	}

	printSolution(board);
	return true;
}

// Driver program to test above function
int main()
{
	solveNQ();
```

```
	return 0;
}

// This code is contributed by Aditya Kumar (adityakumar129)
```

RANDOMIZED QUICK SORT

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int partition(int arr[], int low, int high)
{
	int pivot = arr[low];
	int i = low - 1, j = high + 1;

	while (1) {

		do {
			i++;
		} while (arr[i] < pivot);

		do {
			j--;
		} while (arr[j] > pivot);

		if (i >= j)
			return j;
```

```
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
        }
}

int partition_r(int arr[], int low, int high)
{
        srand(time(0));
        int random = low + rand() % (high - low);

        int temp = arr[random];
        arr[random] = arr[low];
        arr[low] = temp;

        return partition(arr, low, high);
}

void quickSort(int arr[], int low, int high)
{
        if (low < high) {
                int pi = partition_r(arr, low, high);

                quickSort(arr, low, pi);
                quickSort(arr, pi + 1, high);
        }
}
```

```
void printArray(int arr[], int n)
{
	for (int i = 0; i < n; i++)
		printf("%d ", arr[i]);
	printf("\n");
}

int main()
{
	int arr[] = { 10, 7, 8, 9, 1, 5 };
	int n = sizeof(arr) / sizeof(arr[0]);
	quickSort(arr, 0, n - 1);
	printf("Sorted array: \n");
	printArray(arr, n);
	return 0;
}
```