Problem 1:

According to the equation

$$u_i = \left\lfloor \frac{C + c_i - \sum_{j=1}^{n} c_j}{c_i} \right\rfloor$$

We can get the upper bound for each m_i. Thus we may know how many devices (D_i) there can be at most in module n. And then we can transform this problem into a knapsack problem:

There are n kinds of object. The ith kind object consists of u_i objects whose value (performance) is $p_{i,k} = 1 - (1 - r_i)^k$ and whose weight (cost) is $c'_{i,k} = c_i \cdot k$, where $1 \le k \le u_i$. The objects we can pick from looks like:

$$\begin{bmatrix} p_{1,k=1} \\ \cdots \\ p_{1,k=u_2} \end{bmatrix}, \begin{bmatrix} p_{2,k=1} \\ \cdots \\ p_{2,k=u_2} \end{bmatrix} \cdots \begin{bmatrix} p_{i,k=1} \\ \cdots \\ p_{i,k=u_i} \end{bmatrix} \cdots \begin{bmatrix} p_{n,k=1} \\ \cdots \\ p_{n,k=u_n} \end{bmatrix}$$

(Each column is the ith kind of objects)

We must pick one and only one of each kind object to our knapsack and try to maximum the value (performance) with the equation:

$$P_{\max} = \prod_{i=1}^{n} p_{i,k=m_i}$$

m_i is the number of device D_i in ith module which maximum system performance.

It is easy to see we can apply the theory of knapsack here: Suppose $P_{i,c}$ is the optimal solution for a system with i modules and cost constraint c, and p_{i,m_i} is the optimal performance for module i. Then $P_{i-1,c-c_i \cdot m_i}$ must be an optimal solution for a system with $i-1$ modules and cost constraint $c - c_i \cdot m_i$, as we can remove the ith module. In other words, $P_{i,c} = p_{i,m_i} \cdot P_{i-1,c-c_i \cdot m_i}$, for some m_i.

Thus the decomposition equation will be

$$P_{i,c} = \begin{cases} 0 & c \le c_i \cdot k \\ \max\left\{p_{i,k} \middle| 1 \le k \le u_i, c \ge c_i \cdot k\right\} & i = 0 \\ \max\left\{p_{i,k} \cdot P_{i-1,c-c_i \cdot k} \middle| 1 \le k \le u_i, c \ge c_i \cdot k\right\} & i \ge 1 \end{cases}$$

We can do the bottom-up algorithm by generating tables of all possible performances, which are sets of solutions to all possible sub-problems. Starting from the first module, we can calculate all $P_{1,c}$ where $1 \le c \le C$. Then for each module after the first one, we can get $P_{i,c}$ by applying the decomposition equation as we mentioned above. After all calculation is done, $P_{n,C}$ will be the optimal solution we are looking for.

So, define P[n-1][C] as the performance of module n with a cost of C here, the algorithm will be:

```
for (j = 1; j <= C; j++)
    for (i=0; i < n; i++)
        for (k = 0; k < u[i]; k++) {
            cost = c[i] * k;// current module's cost
            if (cost <= j) {
                p = 1 - pow(1-r[i],k);  // performance
                if (i>0)
                    p *= P[i-1][j-cost];
                if (p > P[i][j]) {// get a better answer
                    P[i][j] = p; // record p(i,c)
                    m[i][j] = k; // record m(i)
                }
            }
        }
```

Where P[n-1][C] will be the final optimal solution

And we can see the time complexity of this algorithm is $O\left(C \cdot \sum_{i=1}^{n} u_i\right)$.

Problem 2:

We can make sub-problem tables as following:

When n =1

c	3	6	9	12	15
$p_{i,c}$	0.9	0.99	0.999	0.9999	0.99999
$m_{i,c}$	1	2	3	4	5

When n =2 (start from n =2, the table is generated based on the previous table)

c	8	11	13	16	19	21	24	27	30
$p_{i,c}$	0.7200	0.7920	0.8640	0.9504	0.9590	0.9821	0.9910	0.9919	0.9920
$m_{i,c}$	1	1	2	2	2	3	3	3	3

When n =3

c	10	12	14	16	17	19	21	22	24	26	28	29
$p_{i,c}$	0.4320	0.6048	0.6739	0.7016	0.7413	0.8087	0.8419	0.8896	0.9261	0.9407	0.9465	0.9569
$m_{i,c}$	1	2	3	4	3	3	4	3	4	5	6	4

When n =4

c	14	16	18	20	21	22	23	25	26	27	29	30
$p_{i,c}$	0.3672	0.5141	0.5728	0.5963	0.6301	0.6588	0.6874	0.7246	0.7561	0.7905	0.8229	0.8696
$m_{i,c}$	1	1	1	1	1	2	1	2	1	2	2	2

From the last column of the last table, we can see our optimal solution (maximal system performance) is 0.869559 , which also shows $m_4 = 2$

Tracing back by calculating $c = C - c_i \cdot m_i = 30 - 4 \times 2 = 22$, we can find $m_3 = 3$ from table 3. Keep on digging, we will find $m_2 = 2$ and $m_1 = 2$.

To summarize, the maximum performance is 0.869559, when

$m_1 = 2, m_2 = 2, m_3 = 3, m_4 = 2$

Additional Problem : Program code

```
#include "stdio.h"
#include "malloc.h"
#include "math.h"

// define NULL as 0
#define NULL 0

// input int array
void inputIntArray(int * arr, int n, char arrName) {
    int i=0;
    for (i=0;i<n;i++){
        printf("Please input %c(%d): ", arrName, i+1);
        scanf("%d", (arr+i));
    }
}

// intput double array
void inputDoubleArray(double * arr, int n, char arrName) {
    int i=0;
    for (i=0;i<n;i++){
        printf("Please input %c(%d): ", arrName, i+1);
        scanf("%lf", (arr+i));
    }
}

// output int array
void outIntArray(int* arr, int n, char arrName) {
    int i=0;
    for (i=0;i<n;i++)
        printf("%c(%d): %d\n", arrName, i+1, arr[i]);
}

// output double array
void outDoubleArray(double* arr, int n, char arrName) {
    int i=0;
    for (i=0;i<n;i++)
        printf("%c(%d): %1.2lf\n", arrName, i+1, arr[i]);
}

// output formatted p(i) table
void outPiArrays(double **p, int **m, int n, int C) {
    int i=0,j=0;
```

```
    for (i=0;i<n;i++) {
        printf("\nn=%d ------------",i+1);
        printf("\nc");
        for (j=1;j<=C;j++)
            if (p[i][j]!=p[i][j-1])
                printf("\t%d", j);
        printf("\np",i+1);
        for (j=1;j<=C;j++)
            if (p[i][j]!=p[i][j-1])
                printf("\t%1.4lf",p[i][j]);
        printf("\nm");
        for (j=1;j<=C;j++)
            if (p[i][j]!=p[i][j-1])
                printf("\t%d",m[i][j]);
    }
}

// generate u(i) table basing on C and c(i)
void calculateUi(int *u, int *c, int C, int n){
    int i=0, sumC=0;
    for (i=0;i<n;i++)
        sumC += c[i];
    // calculate U(i)
    for (i=0;i<n;i++)
        u[i]=(C+c[i]-sumC)/c[i];
}

int main(){
    int C,n,i,j,k,cost;
    int *c=NULL,*u=NULL, **m=NULL, *M=NULL;
    double *r=NULL,**P=NULL, finalP, p;

    puts("Please input the number of modules (n)");
    scanf("%d", &n);

    if (n<0) {
        puts("n CANNOT be EQUAL or LESS then zero!");
        getchar(); getchar();
        return 0;
    }

    // try to allocate memory before program start
    c = (int *)malloc(sizeof(int)*n);    // c(i)
    r = (double *)malloc(sizeof(double)*n); // r(i)
```

```
    u = (int *)malloc(sizeof(int)*n);    // u(i)
    M = (int *)malloc(sizeof(int)*n);    // stores final result of m(i)
    m = (int **)malloc(sizeof(int*)*n); // stores m(i) for p(i,c)
    P = (double **)malloc(sizeof(double*)*n);    // p(i,c)
    // unable to allocate so many memory...
    if (c==NULL || r==NULL || u==NULL || M==NULL || m==NULL || P==NULL) {
        puts("Out of memory. Press any key to exit...");
        getchar(); getchar();
        return 0;
    }

    inputIntArray(c,n,'c');
    inputDoubleArray(r,n,'r');

    puts("Please input the cost constraint C");
    scanf("%d", &C);

    puts("Your input is :");
    printf("C=%d, n=%d\n", C, n);
    outIntArray(c,n,'c');
    outDoubleArray(r,n,'r');

    puts("Now calculating, please wait..");

    // now calculate u(i)
    calculateUi(u,c,C,n);
    outIntArray(u,n,'u');

    // check ui
    for (i=0;i<n;i++)
        if (u[i]<0) {
            printf("Wrong C or c(i). Causes u(%d) be %d.  Unable to continue. Press
any key to exit...\n", i, u[i]);
            getchar(); getchar();
            return 0;
        }

    for (i = 0; i < n; i++) {
        P[i] = (double *)malloc(sizeof(double)*(C+1));
        m[i] = (int *)malloc(sizeof(int)*(C+1));
        for (j = 0; j <= C; j++) {
            P[i][j] = 0;// init the performance as 0
            m[i][j] = 0;
        }
```

```
    }

    for (j = 1; j <= C; j++)
        for (i=0; i < n; i++)
            for (k = 0; k < u[i]; k++) {
                cost = c[i] * k;// current module's cost
                if (cost <= j) {
                    p = 1 - pow(1-r[i],k);  // performance
                    if (i>0)
                        p *= P[i-1][j-cost];// if this is not the first device, we need to multiply the previous device's performance
                    if (p > P[i][j]) {  // get a better answer
                        P[i][j] = p;// update p(i,j)
                        m[i][j] = k;// record m(i,j)
                    }
                }
            }
    finalP = P[n-1][C];

    // seek the solution path m(i)
    for (i=n-1, cost=C;i>=0;i--) {
        M[i]= m[i][cost];
        cost -= c[i] * m[i][cost];
    }

    outPiArrays(P,m,n,C);
    printf("\n\nFinal Optimal solution:\n");
    outIntArray(M,n,'m');

    printf("\nMax. System Performance : %lf", finalP);

    // free resource
    for (i=0;i<n;i++) {
        free(P[i]);
        free(m[i]);
    }
    free(c); free(r); free(u); free(P); free(m); free(M);

    puts("\nPress ENTER to exit the program..");
    getchar(); getchar();

    return 0;
}
```