Design and Analysis of Algorithms

UNIT-III

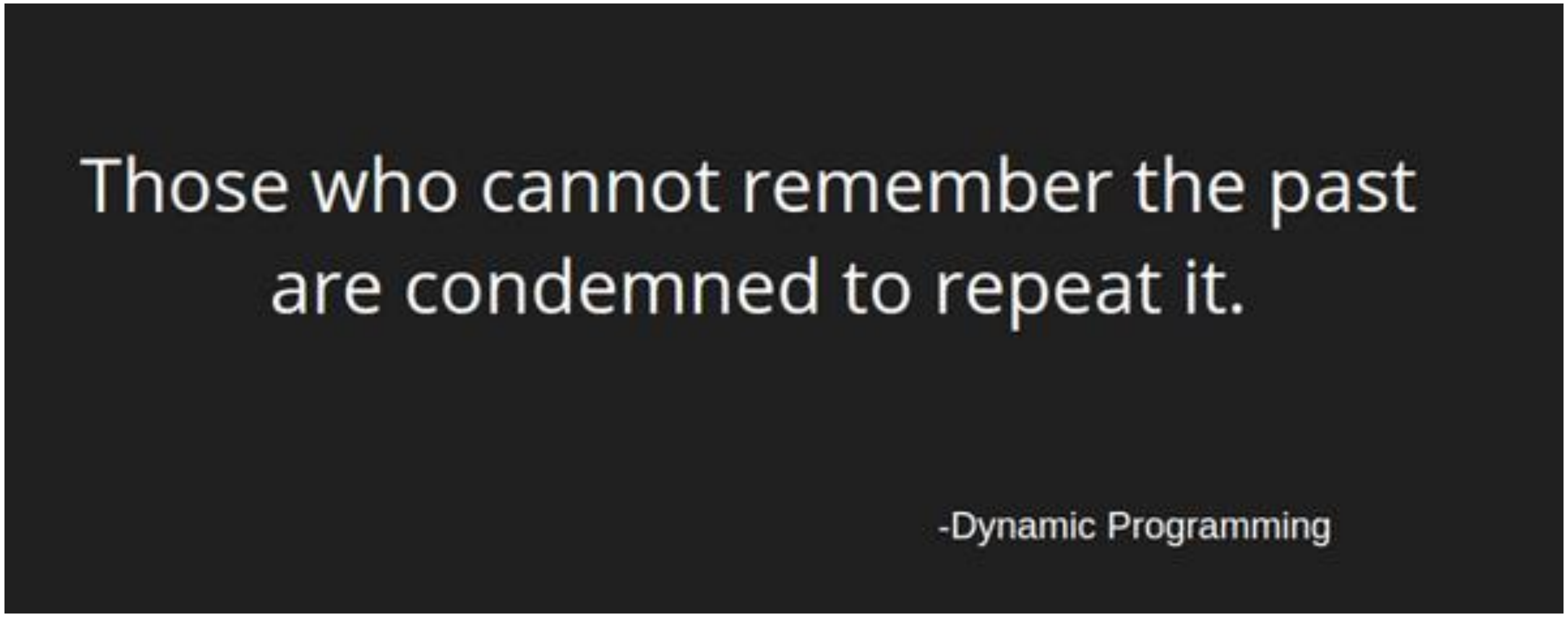
Dynamic Programming

Dr G. Kalyani

Topics

- General method
- Travelling sales person problem
- All pairs shortest Path problem
- 0/1 knapsack problem
- Reliability Design problem

Dynamic programming



Those who cannot remember the past are condemned to repeat it.

-Dynamic Programming

Dynamic programming

- 1+1+1+1+1+1+1+1+1+1+1+1+1+1 =?
- "What's that equal to?“

Dynamic programming

- 1+1+1+1+1+1+1+1+1+1+1+1+1+1 =?
- "What's that equal to?"
- Counting “Fourteen!"
- Then 1+1+1+1+1+1+1+1+1+1+1+1+1+1 +1 =?
- "What about that?"

Dynamic programming

- 1+1+1+1+1+1+1+1+1+1+1+1+1+1 =?
- "What's that equal to?"
- Counting "Fourteen!"
- Then 1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 =?
- "What about that?"
- It is "Fifteen"
- How'd you know it was fifteen so fast?

Dynamic programming

- 1+1+1+1+1+1+1+1+1+1+1+1+1+1 =?
- "What's that equal to?"
- Counting “Fourteen!"
- Then 1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 =?
- "What about that?“
- It is “Fifteen”
- How'd you know it was fifteen so fast?
- So you didn't need to recount because you remembered the previous result as fourteen!
- Dynamic Programming is just a fancy way to say remembering stuff to save time later!"

Dynamic programming

- is repeating the things for which you already have the answer, a good thing ?
- No
- That's what Dynamic Programming is about.
- Divide the problem as sub problems and
- *always* remember answers to the sub-problems you've already solved.
- Use answers of the sub problems in solving the main problem

Dynamic programming

- Majority of the Dynamic Programming problems can be categorized into two types:
  - **1. Optimization problems.**
    2. Combinatorial problems.
- The optimization problems expect you to select a feasible solution, so that the value of the required function is minimized or maximized.
- Combinatorial problems expect you to figure out the number of ways to do something, or the probability of some event happening.

Dynamic programming

- **Dynamic Programming** is a general algorithm design technique for solving problems defined by or formulated as **recurrences with overlapping sub instances.**

- Invented by American mathematician **Richard Bellman to solve optimization problems** .

- **Main idea:**
  - set up a recurrence relating a solution to a larger instance with solutions of some smaller instances
  - solve smaller instances once
  - record solutions in a table
  - extract solution to the initial instance from that table

Dynamic Programming

- **Dynamic programming** is a way of improving on inefficient *divide and-conquer algorithms.*

- By "*inefficient*", *we mean that **the same recursive call is made over and over.***

- If **same *subproblem is solved several times, we can use table to*** store result of a subproblem the first time it is computed and thus never have to recompute it again.

- Dynamic programming is applicable when the subproblems are dependent, that is, when subproblems share sub subproblems.

- **"Programming" refers to a tabular method**

Characteristics of Dynamic Programming

- DP is used to solve problems with the following characteristics:

- Simple sub problems
  - We should be able to break the original problem to **smaller sub problems that have the same structure**

- Optimal substructure of the problems
  - The **optimal solution to the problem contains optimal solutions to its sub problems.**

- Overlapping sub-problems
  - there exist some places where we solve the **same sub problem more** than **once.**

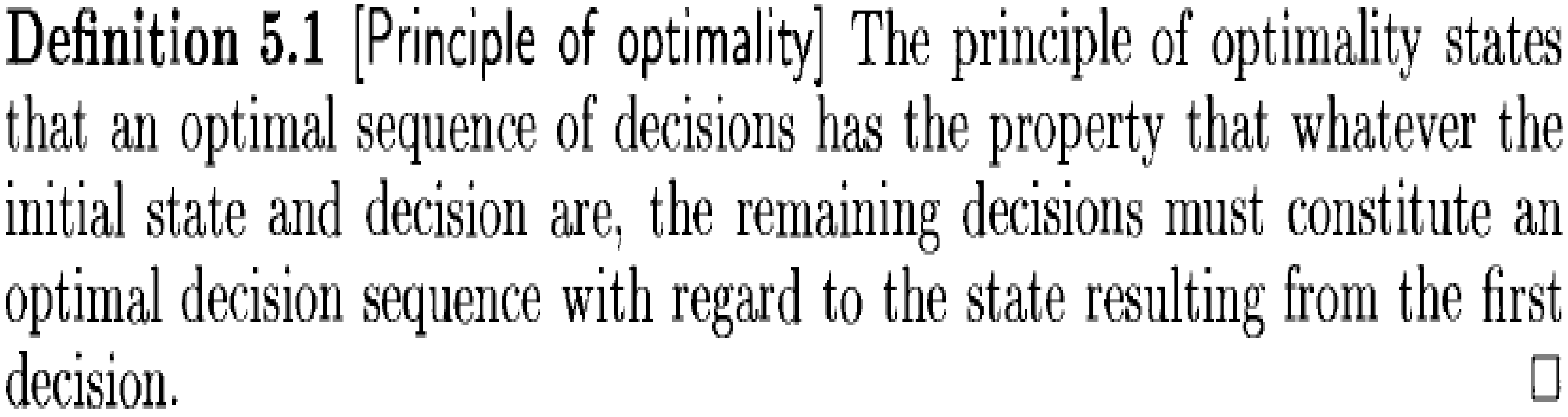
Dynamic Programming: Top Down Vs Bottom Up

- **Bottom Up:**
  - - I'm going to learn programming.
  - Then, I will start practicing.
  - Then, I will start taking part in contests.
  - Then, I'll practice even more and try to improve.
  - After working hard like crazy, I'll be an amazing coder.
- Bottom up approach starts with small problems and go on to large problem.

- **Top Down:**
  - I will be an amazing coder. How?
  - I will work hard like crazy. How?
  - I'll practice more and try to improve. How?
  - I'll start taking part in contests. What I have to do?
  - I'll start practicing. How?
  - by learning programming.
- Top down approach will try to solve large first, if any small is required it will try to solve that and use it.

Principle of Optimality

- The dynamic programming works on a principle of optimality.



Definition 5.1 [Principle of optimality] The principle of optimality states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision. □

Dynamic Programming Vs. Divide & Conquer

Divide & Conquer	Dynamic Programming
1. Partitions a problem into independent smaller sub-problems	1. Partitions a problem into overlapping sub-problems
2. Doesn't store solutions of sub-problems. (Identical sub-problems may arise - results in the same computations are performed repeatedly.)	2. Stores solutions of sub-problems: thus avoids calculations of same quantity twice
3. Top down algorithms: which logically progresses from the initial instance down to the smallest sub-instances via intermediate sub-instances.	3. Bottom up algorithms: in which the smallest sub-problems are explicitly solved first and the results of these used to construct solutions to progressively larger sub-instances

Dynamic Programming vs. Greedy Method

Dynamic Programming	Greedy Method
1. Dynamic Programming is used to solve optimization and combinatorial problems.	1. Greedy Method is used solve optimization problems only.
2. In Dynamic Programming, we choose at each step, but the choice may depend on the solution to sub-problems.	2. In a greedy Algorithm, we make whatever choice seems best at the moment and then solve the sub-problems arising after the choice is made.
3. It is guaranteed that Dynamic Programming will generate an optimal solution using Principle of Optimality.	3. In Greedy Method, there is no such guarantee of getting Optimal Solution.
4. Dynamic programming computes its solution bottom up or top down by synthesizing them from smaller optimal sub solutions.	4. The greedy method computes its solution by making its choices in a serial forward fashion, never looking back or revising previous choices.
5. Example: 0/1 Knapsack	5. Example: Fractional Knapsack

Topics

- General method
- Travelling sales person problem
- All pairs shortest Path problem
- 0/1 knapsack problem
- Reliability Design problem

The Travelling Salesperson Problem

- A **traveler** needs to **visit all the cities from a list**, where **distances** between all the cities are known and each city should be **visited just once**.

- What is the **shortest possible route** that he visits each city exactly once and **returns to the origin city**?

- Travelling salesman problem is the most **notorious computational problem**. We can use **brute-force approach** to evaluate **every possible tour** and **select the best one**. For **n number of vertices** in a graph, there are **(n - 1)! number of possibilities.**

- Instead of brute-force using **dynamic programming approach**, **the solution can be obtained in lesser time.**

The Travelling Salesperson Problem

Problem Definition:

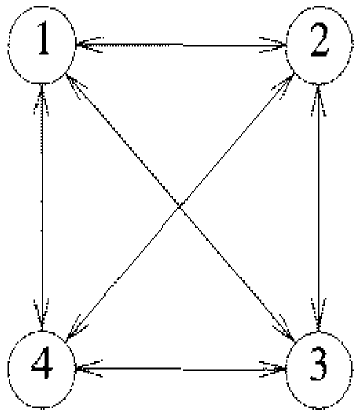
- Let **G (V, E)** be a directed graph with **edge cost $c_{i,j}$** is defined such **that $c_{i,j} > 0$ for all i and j and $c_{i,j} = \propto$, if $\langle i, j\rangle \notin$ E.**
- Let **V = n** and assume n>1.
- The traveling salesman problem is to find a tour of **minimum cost.**
- A tour of **Graph G is** a directed cycle that include **every vertex in V.**
- The **cost of the tour** is **the sum of cost of the edges** on the tour.
- The tour is the shortest path that **starts and ends at the same vertex.**

DP Solution to Travelling Salesperson Problem

The function **g(i, S)** is the **length of an optimal tour**.

$$g(i, S) = \min_{j \in S}\{c_{ij} + g(j, S - \{j\})\}$$

Example-TSP



$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

$| s | = 0.$

$$g(2,\Phi) = c_{21} = 5$$
$$g(3,\Phi) = c_{31} = 6$$
$$g(4,\Phi) = c_{41} = 8$$

$|S| = 1$

$g(2,\{3\}) = c_{23} + g(3,\Phi) = 9+6 = 15$

$g(2,\{4\}) = c_{24} + g(4,\Phi) = 10+8 = 18$

$g(3,\{2\}) = c_{32} + g(2,\Phi) = 13+5 = 18$

$g(3,\{4\}) = c_{34} + g(4,\Phi) = 12+8 = 20$

$g(4,\{2\}) = c_{42} + g(2,\Phi) = 8+5 = 13$

$g(4,\{3\}) = c_{43} + g(3,\Phi) = 9+6 = 15$

$|S| = 2$

$g(2,\{3, 4\}) = \min\{ c_{23} + g(3, \{4\}),\ c_{24} + g(4, \{3\}) \}$

$\min\{ 9+20,\ 10+15\}$

$\min\{29, 25\} = 25$

$g(3, \{2, 4\}) = \min\{c_{32} + g(2, \{4\}), c_{34} + g(4, \{2\})\}$

$\min\{13+18, 12+13\}$

$\min\{31,25\} = 25$

$g(4, \{2, 3\}) = \min\{c_{42} + g(2, \{3\}), c_{43} + g(3, \{2\})\}$

$\min\{8+15, 9+18\}$

$\min\{23, 27\} = 23$

$|S| = 3$

$g(1, \{2, 3, 4\}) = \min\{c_{12} + g(2, \{3, 4\}),$

$c_{13} + g(3, \{2, 4\}),$

$c_{14} + g(4, \{2, 3\}) \}$

$\min\{10+25, 15+25, 20+23\}$

$\min\{35, 40, 43\} = 35$

optimal cost is 35

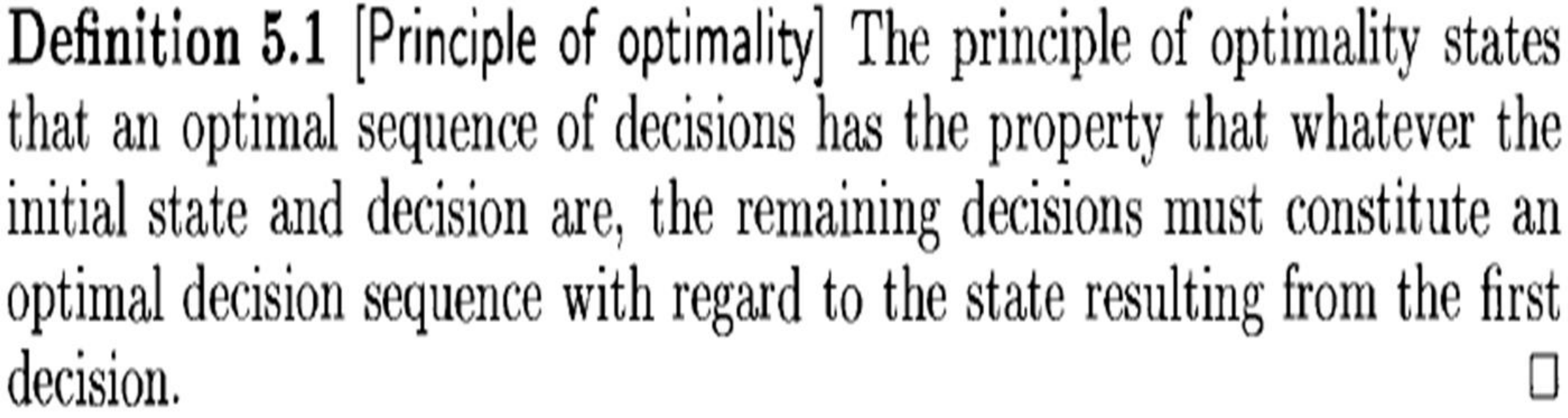
the shortest path is,

$g(1,\{2, 3, 4\}) = c_{12} + g(2, \{3, 4\})$ => 1 -> 2

$g(2,\{3, 4\}) = c_{24} + g(4,\{3\})$ => 1->2->4

$g(4,\{3\}) = c_{43} + g(3,\{\Phi\})$ => 1->2->4->3->1

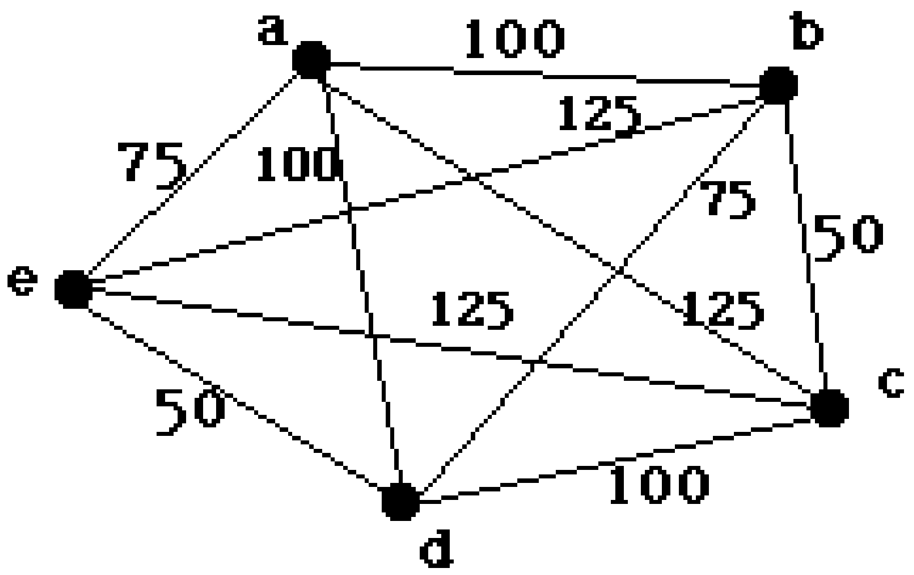
so the optimal tour is 1 → 2 → 4 →3 → 1



Definition 5.1 [Principle of optimality] The principle of optimality states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision. □

Example 2

An Instance of the
Traveling Salesman Problem



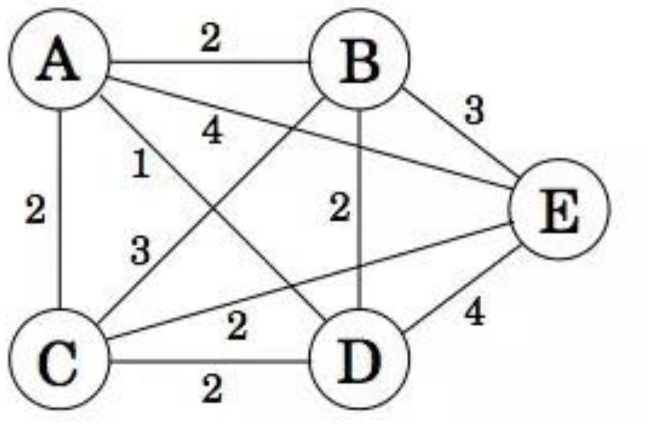
Time Complexity Analysis

- An **algorithm** that proceeds to find an optimal tour will require **$\theta(n^2 2^n)$ time** as the computation of g(i,S) with |S|= k requires k= 1 comparisons when solving (2).

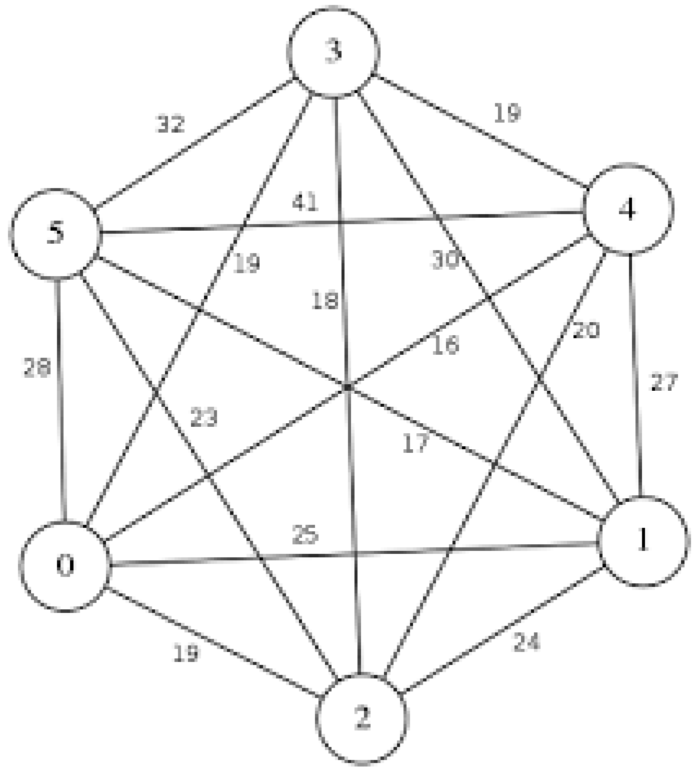
- This is better than enumerating all **n! different tours to find the best one**. The most serious drawback of this **dynamic programming solution is the space needed, $0(n2^n)$.**

- This is too large even for modest values of n.

Problems for practicing



Find the tour for TSP by considering "A" as the starting point



Find the tour for TSP by considering "3" as the starting point

Topics

- General method
- Travelling sales person problem
- All pairs shortest Path problem
- 0/1 knapsack problem
- Reliability Design problem

All Pairs Shortest Path Problem

- Let G = (V, E) be a directed graph with n vertices.
- Let cost be a adjacency matrix for G such that cost(i,i)=0,1< i < n.
- The cost(i,j) is the length (or cost) of edge(i,j)
  - cost(i, j)=x if (i,j) $\in$ E(G) and
  - cost(i,j)= ∞ if i $\neq$ j and (i,j) $\notin$ E(G).
- The all-pairs shortest-path problem is to determine a matrix A such that A(i,j) is the length of a shortest path from i to j.

All Pairs Shortest Path Problem

- The matrix A can be obtained by solving n single-source problems using the algorithm Shortest Paths of Greedy method.

- Since each application of this procedure requires $0(n^2)$time, the matrix A can be obtained in $0(n^3)$time.

- We obtain an alternate $O(n^3)$ solution to this problem using the principle of optimality.

- Our alternate solution requires a weaker restriction on edge costs than required by Shortest Paths.

- Rather than require cost(i,j)> 0, for every edge(i,j),we only require that G have no cycles with negative length.

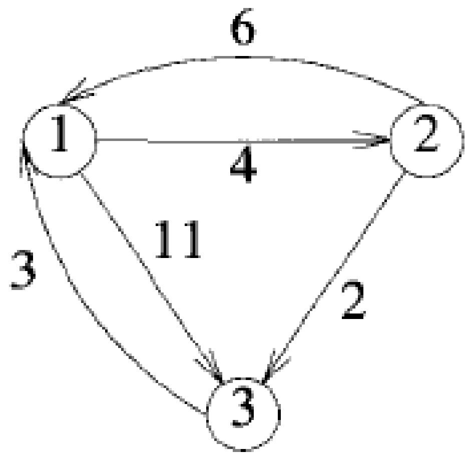
Recurrence relating for APSP problem

- Using $A^k(i,j)$ to represent the length of a shortest path from i to j going through no vertex of index greater than k.

$$A^k(i, j) = \min\{A^{k-1}(i,j), (A^{k-1}(i,k) + A^{k-1}(k,j))\}$$

$$A^0(i,j) = cost(i,j),\ 1 \le i \le n,\ 1 \le j \le n.$$

Example



A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

Example

A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

(b) A^0

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

(c) A^1

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

(d) A^2

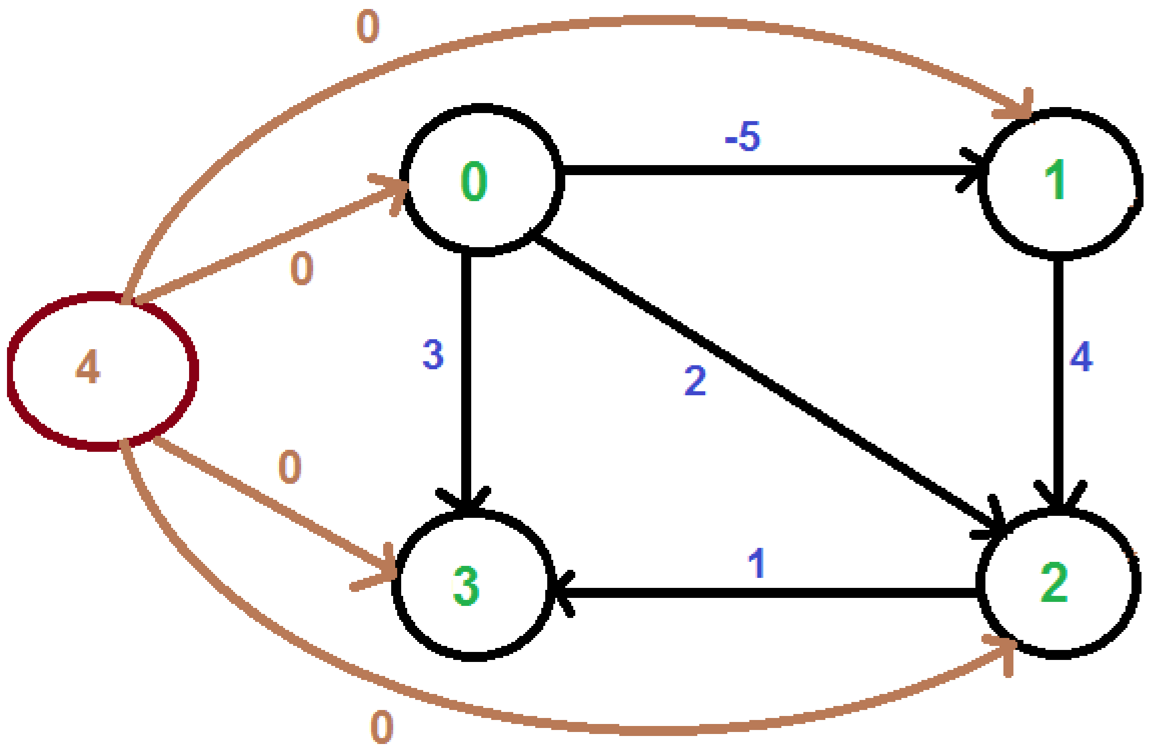
A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

(e) A^3

Algorithm for All Pairs Shortest Path Problem

```
Algorithm AllPaths($cost, A, n$)
// $cost[1:n, 1:n]$ is the cost adjacency matrix of a graph with
// $n$ vertices; $A[i,j]$ is the cost of a shortest path from vertex
// $i$ to vertex $j$. $cost[i,i] = 0.0$, for $1 \leq i \leq n$.
{
    for $i := 1$ to $n$ do
        for $j := 1$ to $n$ do
            $A[i,j] := cost[i,j]$; // Copy $cost$ into $A$.
    for $k := 1$ to $n$ do
        for $i := 1$ to $n$ do
            for $j := 1$ to $n$ do
                $A[i,j] := \min(A[i,j], A[i,k] + A[k,j])$;
}
```

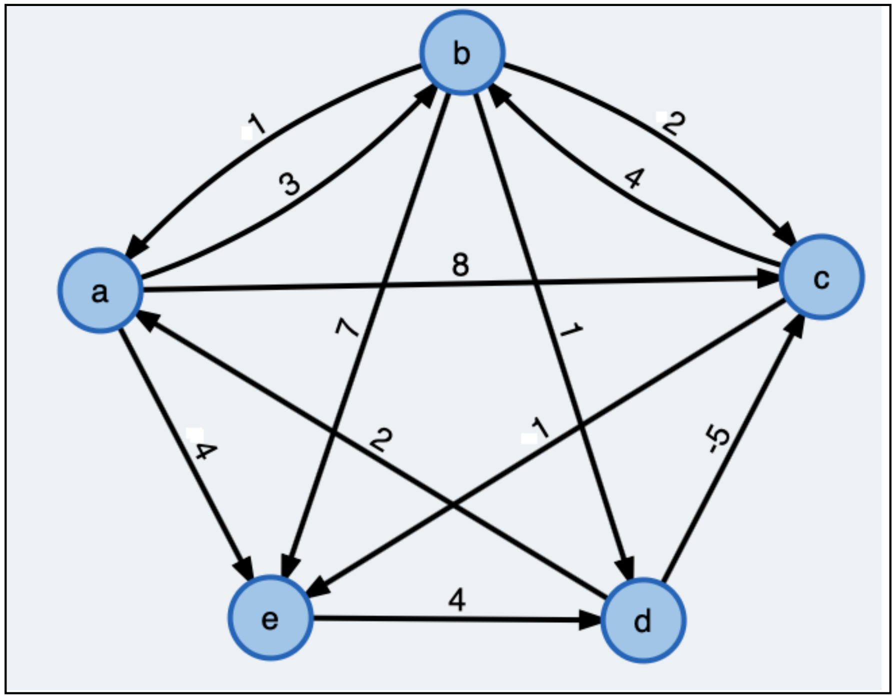
Example 2

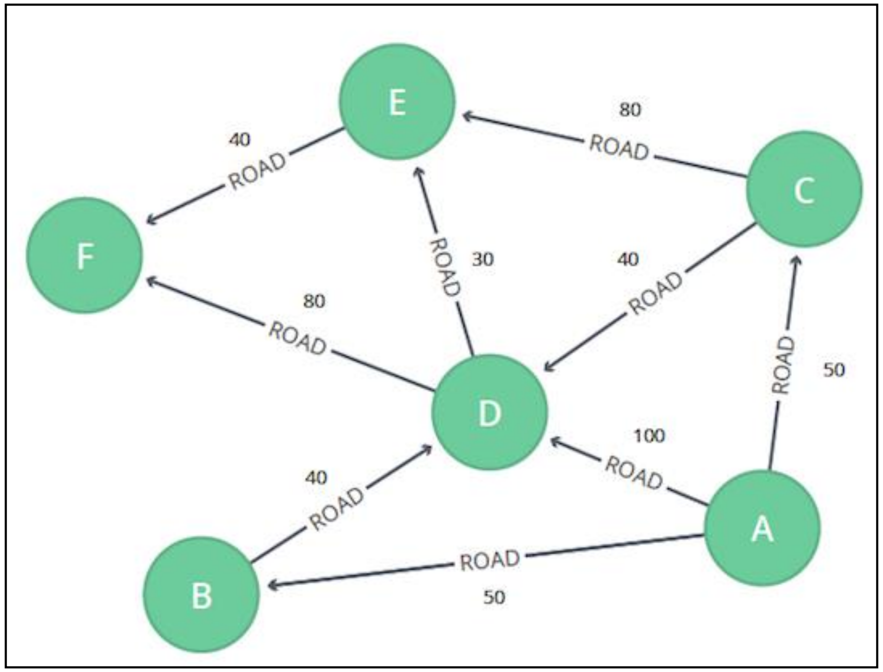


Time Complexity

- All pair shortest path algorithm also known as Floyd-Warshall Algorithm.
- The time needed by All Paths (Algorithm 5.3) is especially easy to determine because the looping is independent of the data in the matrix A.
- Line 11 is iterated n^3 times, and so the time for AllPaths is $O(n^3)$.

Problems for Practice





Topics

- General method
- Travelling sales person problem
- All pairs shortest Path problem
- 0/1 knapsack problem
- Reliability Design problem

0/1 or 0-1 Knapsack Problem

- Earlier we have discussed Fractional Knapsack problem using Greedy approach.
- We have shown that Greedy approach gives an optimal solution for Fractional Knapsack problem.
- In 0/1 or 0-1 Knapsack, items cannot be broken which means the we should take the item as a whole or should leave it.
- Hence, in case of 0-1 Knapsack, the value of $\boldsymbol{x_i}$ can be either ***0*** or ***1***, where other constraints remain the same.

Fractional Vs 0/1 Knapsack Problem

Fractional Knapsack Problem

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i \text{ ----------A}$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m \text{ --------B}$$

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n \text{ ------C}$$

0/1 Knapsack Problem

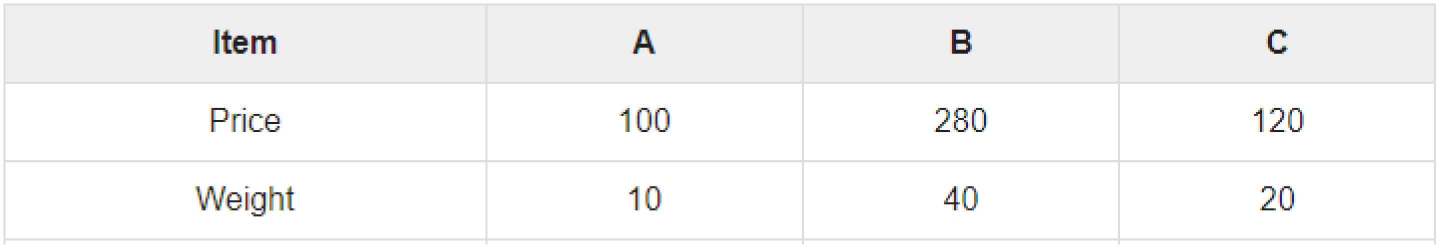
$$\text{maximize} \sum_{1 \le i \le n} p_i x_i \text{ ----------A}$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m \text{ --------B}$$

$$\text{and } \underline{\hspace{4cm}} \, 1 \le i \le n \text{ -----C}$$

0/1 Knapsack Problem

- 0/1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution.
- Ex: Consider the following data with knapsack capacity m=60



Item	A	B	C
Price	100	280	120
Weight	10	40	20

0/1 Knapsack Problem

- 0-1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution.
- Ex: Consider the following data with knapsack capacity m=60

Item	A	B	C
Price	100	280	120
Weight	10	40	20
Ratio	10	7	6

- XA=1 ➔ m=60-10=50
- WB<=m ➔XB=1 ➔ m=50-40 = 10
- WC<=m failed.so stop the process
- Hence the optimal solution as per greedy method is (1,1,0) with total profit of 100+280=380.
- But this is not an optimal solution for the problem. The optimal solution for the problem is ____________ with total profit of __________.

0/1 knapsack problem with Dynamic Programming

- $S^i = S^{i-1} \cup S_1^{i-1}$
- S^i is the set of all possibilities up to ith object
- $S_1^{i-1} = \{ (P,W)/(P-p_i, W-w_i) \in S^{i-1} \}$
- $S^0 = (0, 0)$
- compute S^1 to S^n

Example for 0/1 Knapsack Problem

Ex: n=3; m=6; w[]={2,3,4}; p[]={1,2,5}

$$S^i = S^{i-1} \cup S_1^{i-1}$$

- Start with S^0 =(0,0) [pair is (P,W)]
- Compute S^1 to S^n
- $S^1 = S^0 \cup S_1^0$
  - S_1^0 =(1,2) ➔ S^1 ={(0,0),(1,2)}
- $S^2 = S^1 \cup S_1^1$
  - S_1^1 = {(2,3),(3,5)} ➔ S^2 ={(0,0),(1,2),(2,3),(3,5)}
- $S^3 = S^2 \cup S_1^2$
- S_1^2 ={(5,4),(6,6),(7,7),(8,9)}
- ➔ S^3 = ={(0,0),(1,2),(2,3),(3,5), (5,4),(6,6),(7,7),(8,9)}
- S^3 = ={(0,0),(1,2),(2,3),~~(3,5)~~,(5,4),(6,6),(7,7),(8,9)}

Example continuation

- Hence finally S^3 = ={(0,0),(1,2),(2,3),(5,4),(6,6)}
- Because of highest profit pair is (6,6), it means you will highest profit of 6 with a total weight in the knapsack 6.
- But what is the solution for this highest profit?
- The selected pair is (6,6) from S^3 .
- Since $S^3 = S^2 \cup S_1^2$ -> Check whether (6,6) is in S^2 or not.
- (6,6) $\notin S^2 \Rightarrow$ x3=1
- (6,6)- (5,4)= (1,2) Check whether in S^1 or not
- (1,2) $\in S^1 \Rightarrow$x2=0
- (1,2) Check whether in S^0 or not
- (1,2) $\notin S^0 \Rightarrow$ x1=1
- Hence the solution is (1,0,1) with a total profit of 6.

Example 2

- **Ex:** n=6; m=165;
- w[]= P[]= {100,50,20,10,7,3};

Profit of 6

eg:- $n=6$ $(P_1, P_2, P_3, P_4, P_5, P_6) = (w_1, w_2, w_3, w_4, w_5, w_6)$

$= (100, 50, 20, 10, 7, 3)$

$M = 165$

$S^0 = \{0\}$ $S_1^0 = 100$

$S^1 = \{0, 100\}$ $S_1^1 = \{50, 150\}$

$S^2 = \{0, 50, 100, 150\}$ $S_1^2 = \{20, 70, 120, 170\}$

$S^3 = \{0, 20, 50, 70, 100, 120, 150, \cancel{170}\}$

$S_1^3 = \{10, 30, 60, 80, 110, 130, 160\}$

$S^4 = \{0, 10, 20, 30, 50, 60, 70, 80, 100, 110, 120, 130, 150, 160\}$

$S_1^4 = \{7, 17, 27, 37, 57, 67, 77, 87, 107, 117, 127, 137, 157, 167\}$

$S^5 = \{0, 7, 10, 17, 20, 27, 30, 37, 50, 57, 60, 67, 70, 77, 80, 87, 100, 107, 110, 117$

$120, 127, 130, 137, 150, 157, 160, \cancel{167}\}$

$S_1^5 = \{3, 10, 13, 20, 23, 30, 33, 40, 53, 60, 63, 70, 73, 80, 83, 90, 10$

$110, 113, 120, 123, 130, 133, 140, 153, 160, 163\}$

- The selected pair is (163,163) from S^6
- Check whether (163,163) is in S^5 or not.
- $(163,163) \notin S^5 \Rightarrow$ x6=1
- (163,163)-(3,3)= (160,160) Check whether in S^4 or not
- $(160,160) \in S^4 \Rightarrow$ x5=0
- (160,160) Check whether in S^3 or not
- $(160,160) \notin S^3 \Rightarrow$ x4=1
- (160,160)-(10,10)= (150,150) Check whether in S^2 or not
- $(150,150) \in S^2 \Rightarrow$ x3=0
- (150,150) Check whether in S^1 or not
- $(150,150) \notin S^1 \Rightarrow$ x2=1
- (150,150)-(50,50)= (100,100) Check whether in S^0 or not
- $(100,100) \notin S^0 \Rightarrow$ x1=1
- Hence the solution is (1,1,0,1,0,1) with a total profit of 163.

Problems for Practice

Problem-1:

Find solution to the Knapsack using Dynamic programming n = 4, m = 7, $(p_1, p_2, p_3, p_4) = (1, 4, 5, 7)$ and $(w_1, w_2, w_3, w_4) = (1, 3, 4, 5)$. **7M**

Problem-2:

Example: Solve knapsack instance M =8 and N = 4. Let $P_i =$ *and* W_i are as shown below.

i	P_i	W_i
1	1	2
2	2	3
3	5	4
4	6	5

Topics

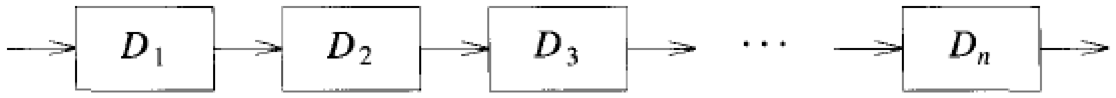
- General method
- Travelling sales person problem
- All pairs shortest Path problem
- 0/1 knapsack problem
- Reliability Design Problem

What is Reliability

- Reliability is the likelihood(probability) that a product will
  - perform its intended function
  - within specified tolerances
  - under stated conditions
  - for a given period of time

What is Reliability Design Problem

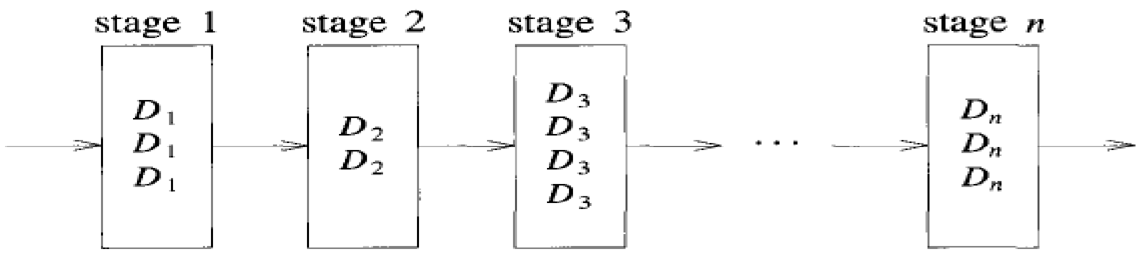
- The problem is to design a system that is composed of several devices connected in series.



- Let r_i be the reliability of device D_i (that it is the probability that device i will function properly).
- The reliability of the entire system is πr_i.
- Even if one device failed, the entire system will be failed.
- Hence, it is desirable to duplicate devices.

What is Reliability Design Problem

- Multiple copies of the same device type are connected in parallel.

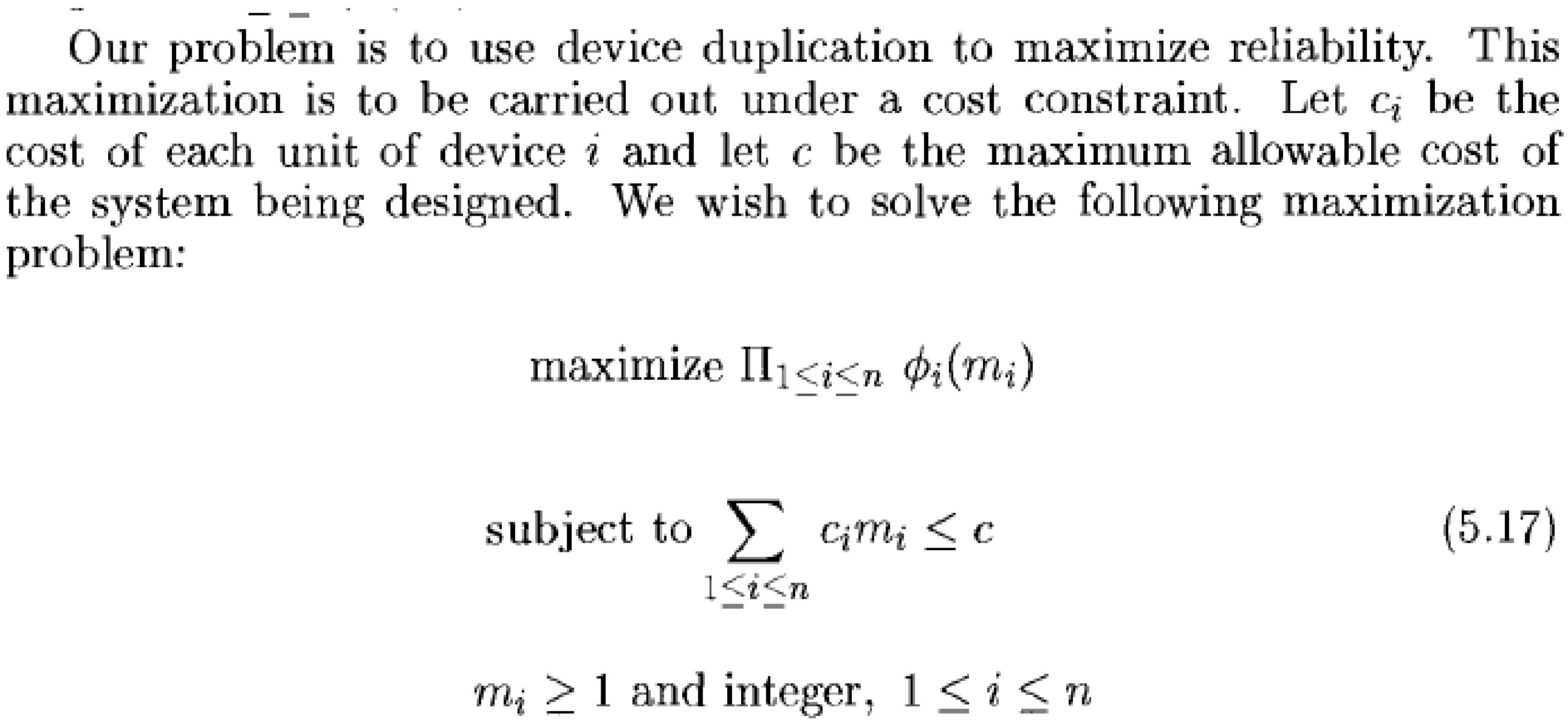


If stage i contains m_i copies of device D_i, then the probability that all m_i have a malfunction is $(1 - r_i)^{m_i}$. Hence the reliability of stage i becomes

$1 - (1 - r_i)^{m_i}$.

Let us assume that the reliability of stage i is given by a function $\phi_i(m_i)$, $1 \leq n$. (It is quite conceivable that $\phi_i(m_i)$ may decrease after a certain value of m_i.) The reliability of the system of stages is $\Pi_{1 \leq i \leq n} \phi_i(m_i)$.

What is Reliability Design Problem



Our problem is to use device duplication to maximize reliability. This maximization is to be carried out under a cost constraint. Let c_i be the cost of each unit of device i and let c be the maximum allowable cost of the system being designed. We wish to solve the following maximization problem:

$$\text{maximize } \Pi_{1 \leq i \leq n}\ \phi_i(m_i)$$

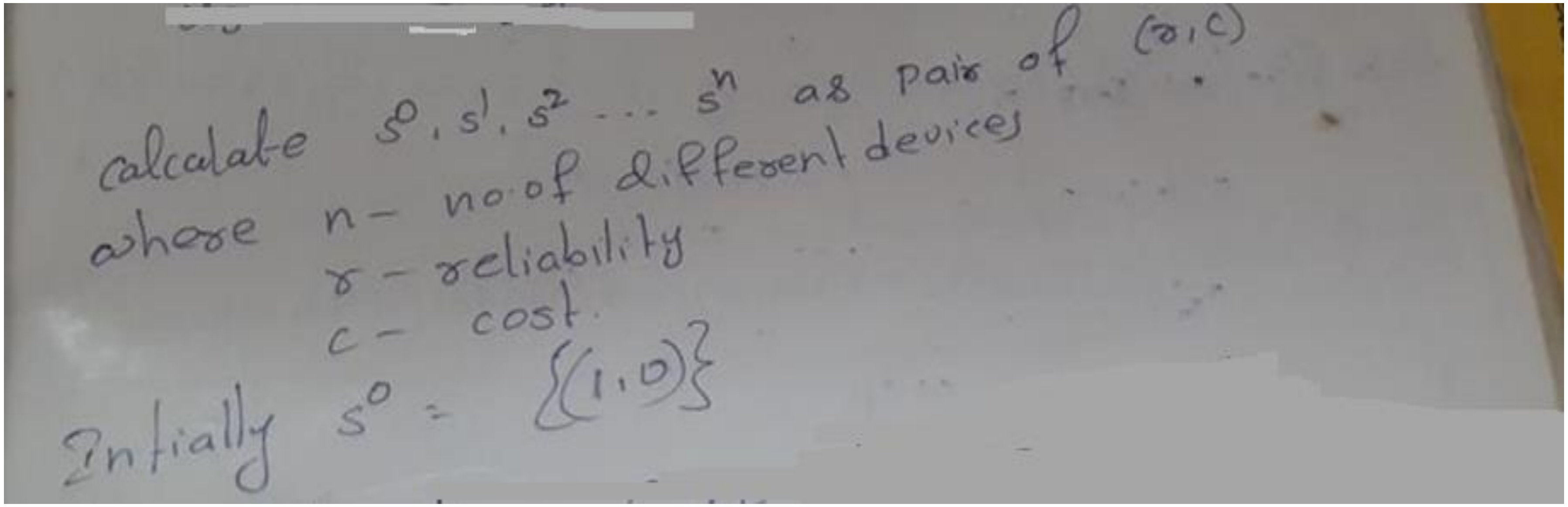
$$\text{subject to } \sum_{1 \leq i \leq n} c_i m_i \leq c \tag{5.17}$$

$$m_i \geq 1 \text{ and integer, } 1 \leq i \leq n$$

What is Reliability Design Problem

A dynamic programming solution can be obtained in a manner similar to that used for the knapsack problem. Since, we can assume each $c_i > 0$, each m_i must be in the range $1 \leq m_i \leq u_i$, where

$$u_i = \left\lfloor \left(c + c_i - \sum_{1}^{n} c_j\right) / c_i \right\rfloor$$



calculate $S^0, S^1, S^2 \dots S^n$ as pair of (r, c)

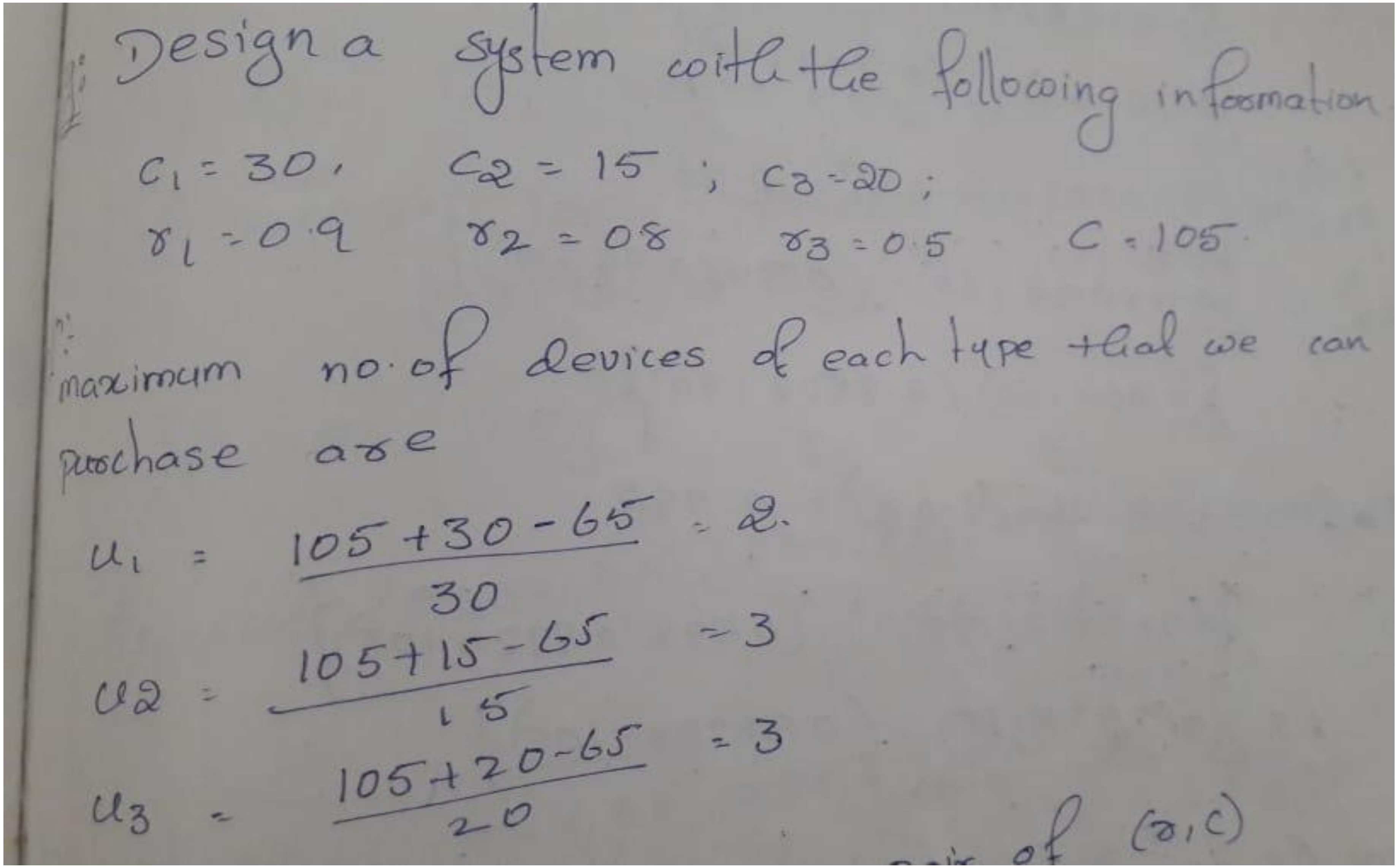
where n – no of different devices

r – reliability

c – cost.

Initially $S^0 = \{(1, 0)\}$

Example for Reliability Design Problem



Design a system with the following information

$c_1 = 30$, $c_2 = 15$; $c_3 = 20$;

$r_1 = 0.9$ $r_2 = 0.8$ $r_3 = 0.5$ $C = 105$.

maximum no. of devices of each type that we can purchase are

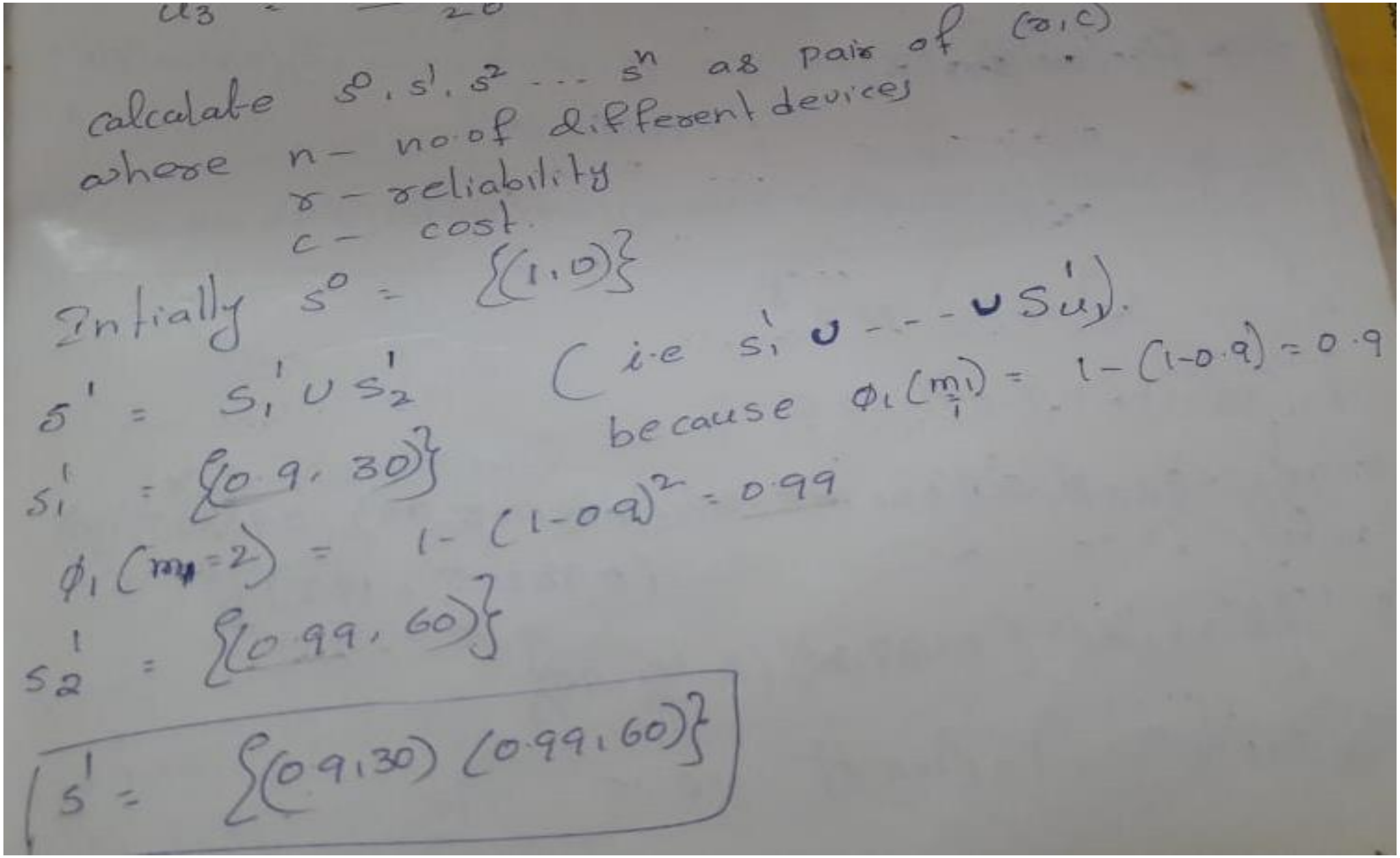
$$u_1 = \frac{105+30-65}{30} = 2.$$

$$u_2 = \frac{105+15-65}{15} = 3$$

$$u_3 = \frac{105+20-65}{20} = 3$$

of (r, c)

Example for Reliability Design Problem



a_3 = 20

calculate $S^0, S^1, S^2 \dots S^n$ as pair of (r, c)

where n – no. of different devices

r – reliability

c – cost.

Initially $S^0 = \{(1, 0)\}$

$S^1 = S_1^1 \cup S_2^1$ (i.e $S_1^1 \cup \dots \cup S_{u_1}^1$).

because $\phi_1(m_1) = 1-(1-0.9) = 0.9$

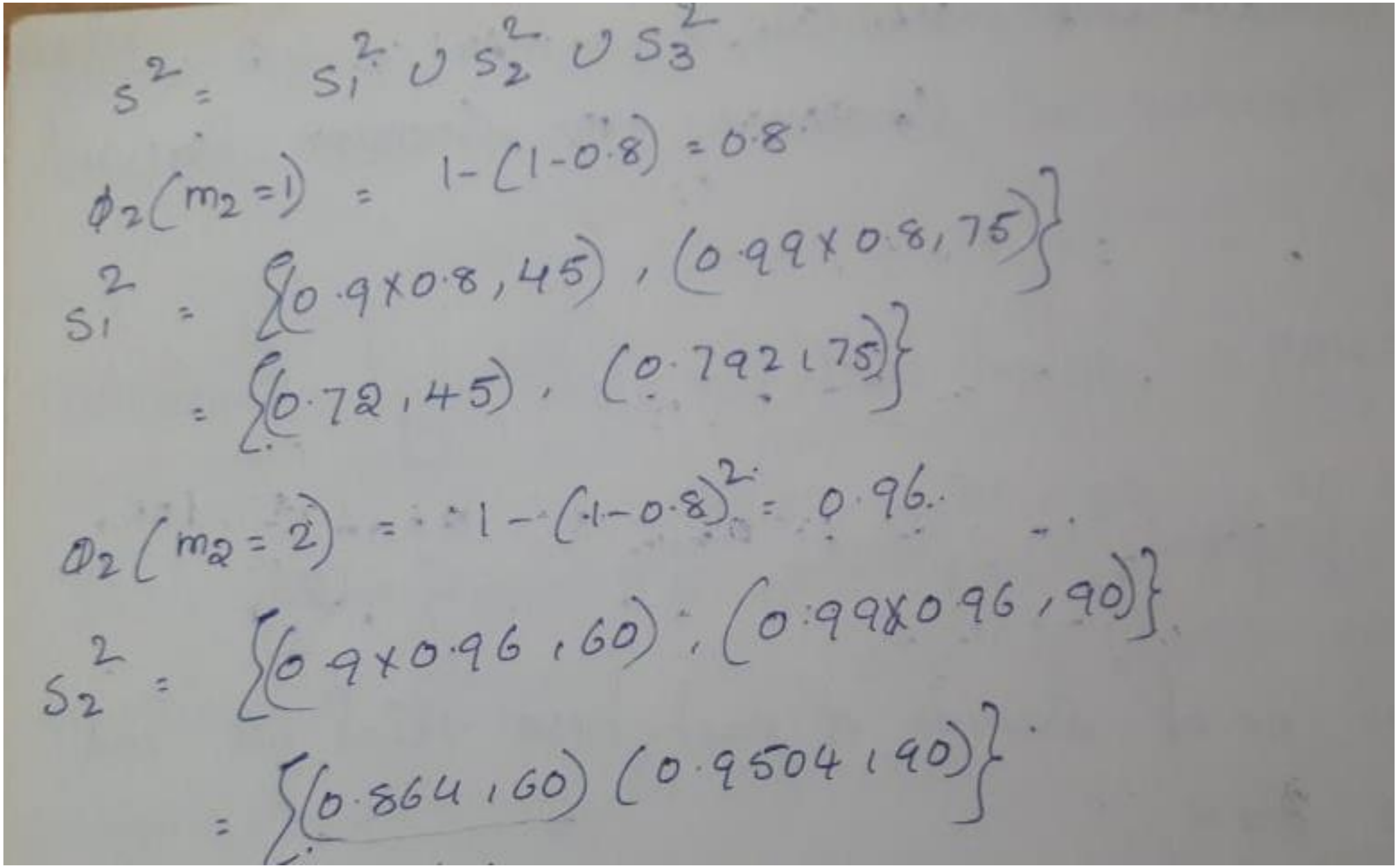
$S_1^1 = \{(0.9, 30)\}$

$\phi_1(m_1=2) = 1-(1-0.9)^2 = 0.99$

$S_2^1 = \{(0.99, 60)\}$

$\boxed{S^1 = \{(0.9, 30)(0.99, 60)\}}$

Example for Reliability Design Problem



$S^2 = S_1^2 \cup S_2^2 \cup S_3^2$

$\phi_2(m_2=1) = 1-(1-0.8) = 0.8$

$S_1^2 = \{(0.9\times0.8, 45), (0.99\times0.8, 75)\}$

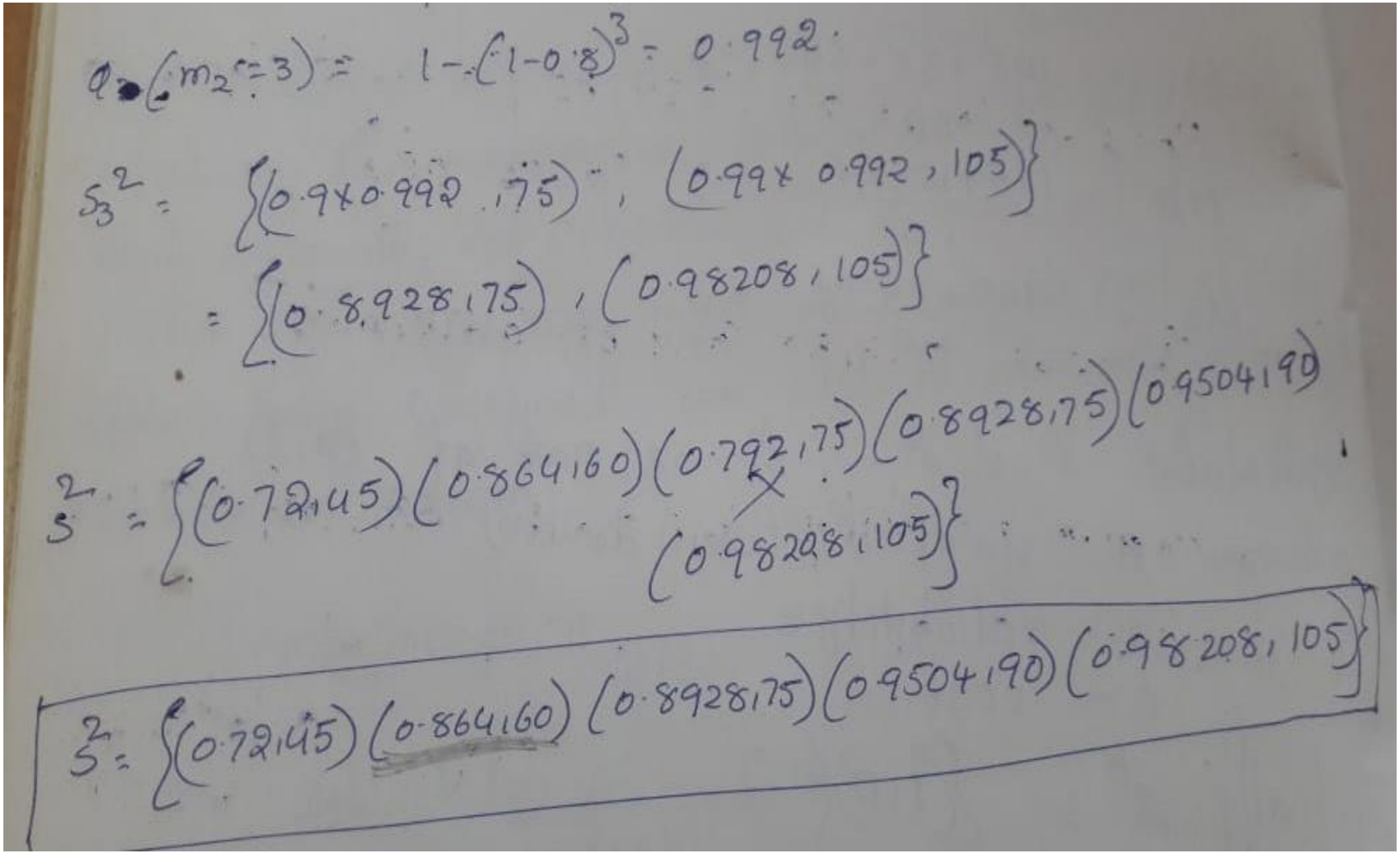
$= \{(0.72, 45), (0.792, 75)\}$

$\phi_2(m_2=2) = 1-(1-0.8)^2 = 0.96$

$S_2^2 = \{(0.9\times0.96, 60), (0.99\times0.96, 90)\}$

$= \{(0.864, 60)(0.9504, 90)\}$

Example for Reliability Design Problem



$\phi_2(m_2=3) = 1-(1-0.8)^3 = 0.992$

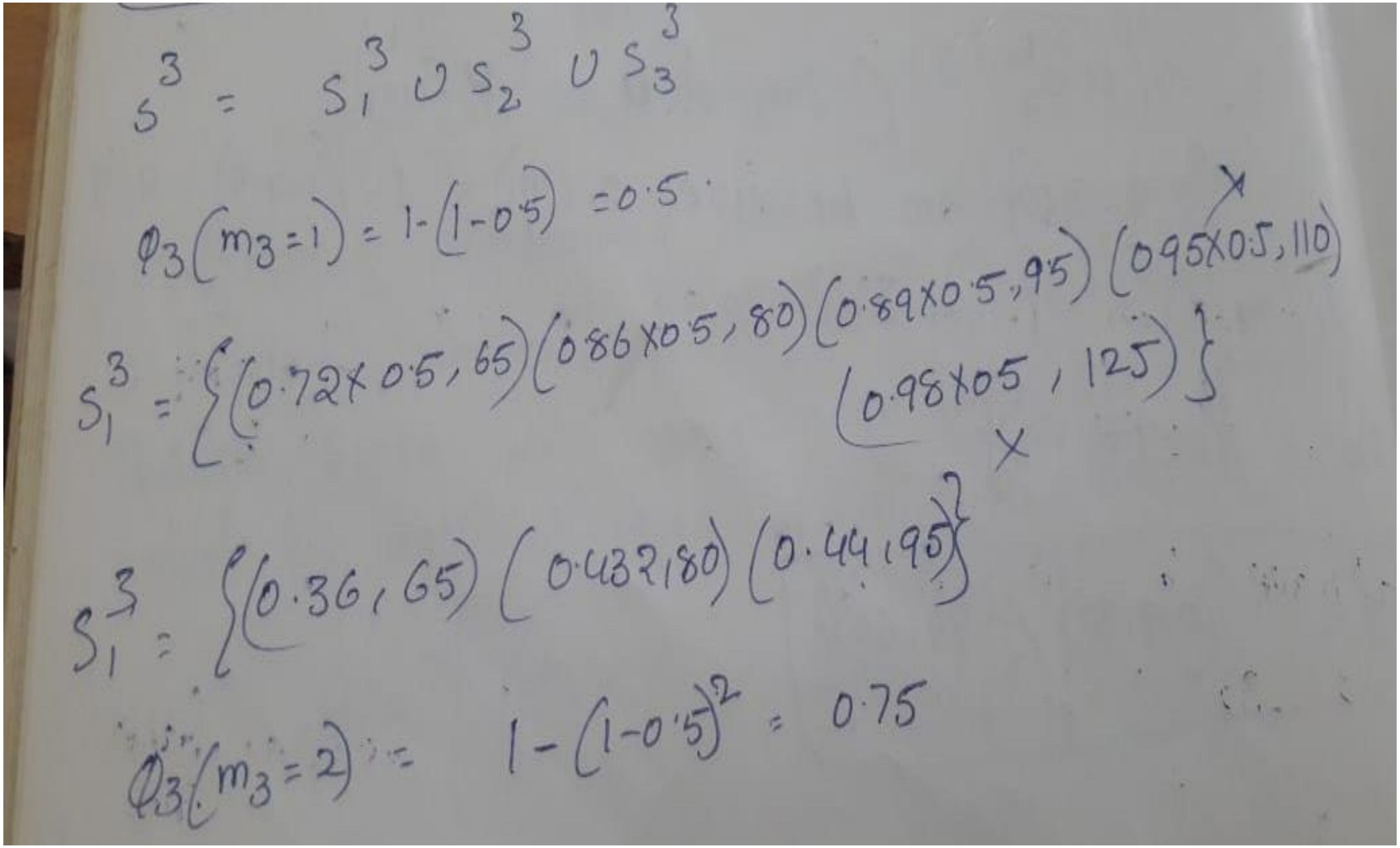
$S_3^2 = \{(0.9\times 0.992, 75), (0.99\times 0.992, 105)\}$

$= \{(0.8928, 75), (0.98208, 105)\}$

$S^2 = \{(0.72,45)(0.864,60)\cancel{(0.792,75)}(0.8928,75)(0.9504,90)(0.98208,105)\}$

$$\boxed{S^2 = \{(0.72,45)(0.864,60)(0.8928,75)(0.9504,90)(0.98208,105)\}}$$

Example for Reliability Design Problem



$S^3 = S_1^3 \cup S_2^3 \cup S_3^3$

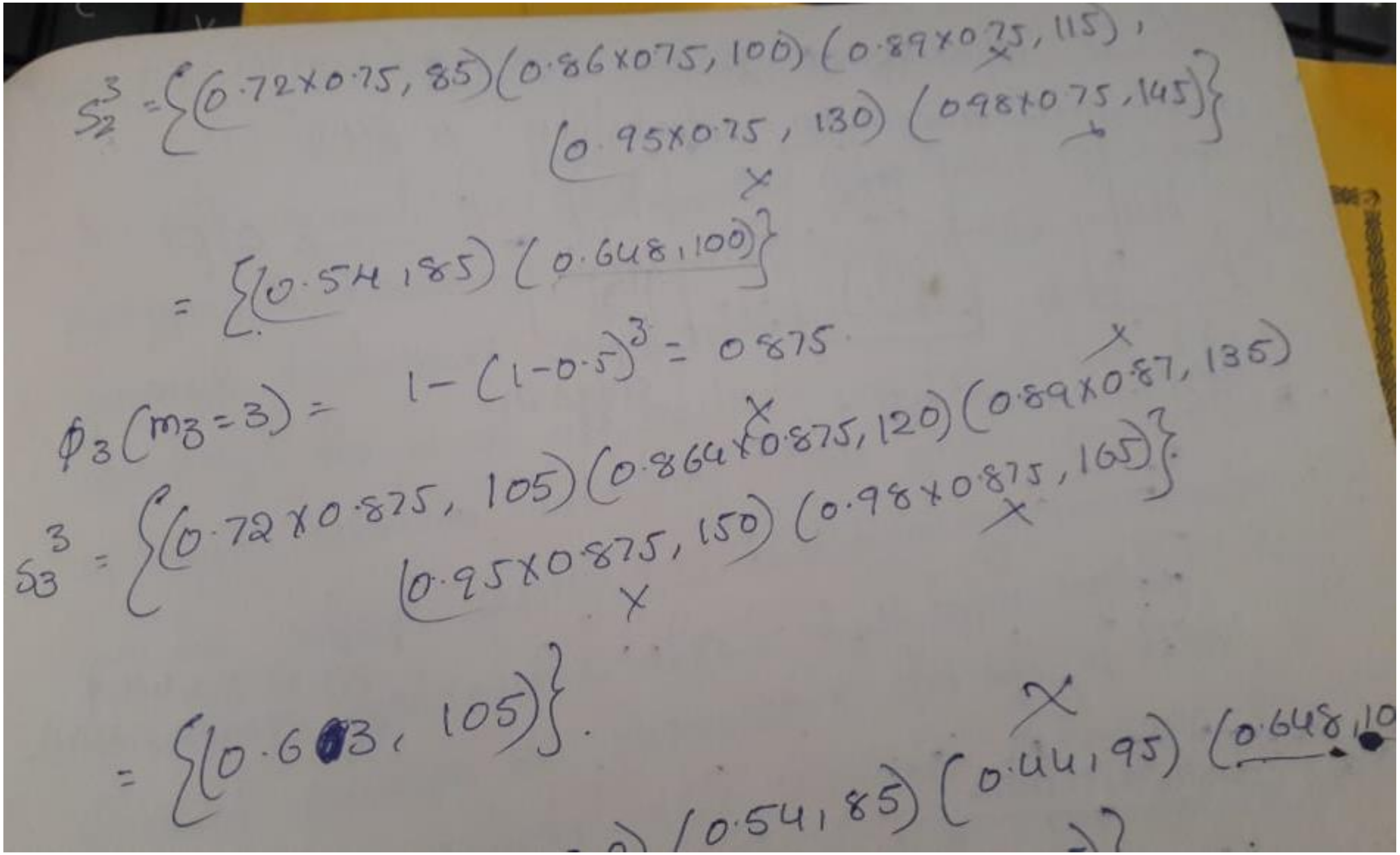
$\phi_3(m_3=1) = 1-(1-0.5) = 0.5$

$S_1^3 = \{(0.72 \times 0.5, 65)(0.86 \times 0.5, 80)(0.89 \times 0.5, 95)\cancel{(0.95 \times 0.5, 110)}$
$\cancel{(0.98 \times 05, 125)}\}$

$S_1^3 = \{(0.36, 65)(0.432, 80)(0.44, 95)\}$

$\phi_3(m_3=2) = 1-(1-0.5)^2 = 0.75$

Example for Reliability Design Problem



$S_2^3 = \{(0.72\times0.75, 85)(0.86\times0.75, 100)(0.89\times0.75, 115),$
$(0.95\times0.75, 130)(0.98\times0.75, 145)\}$

$= \{(0.54, 85)(0.648, 100)\}$

$\phi_3(m_3=3) = 1-(1-0.5)^3 = 0.875$

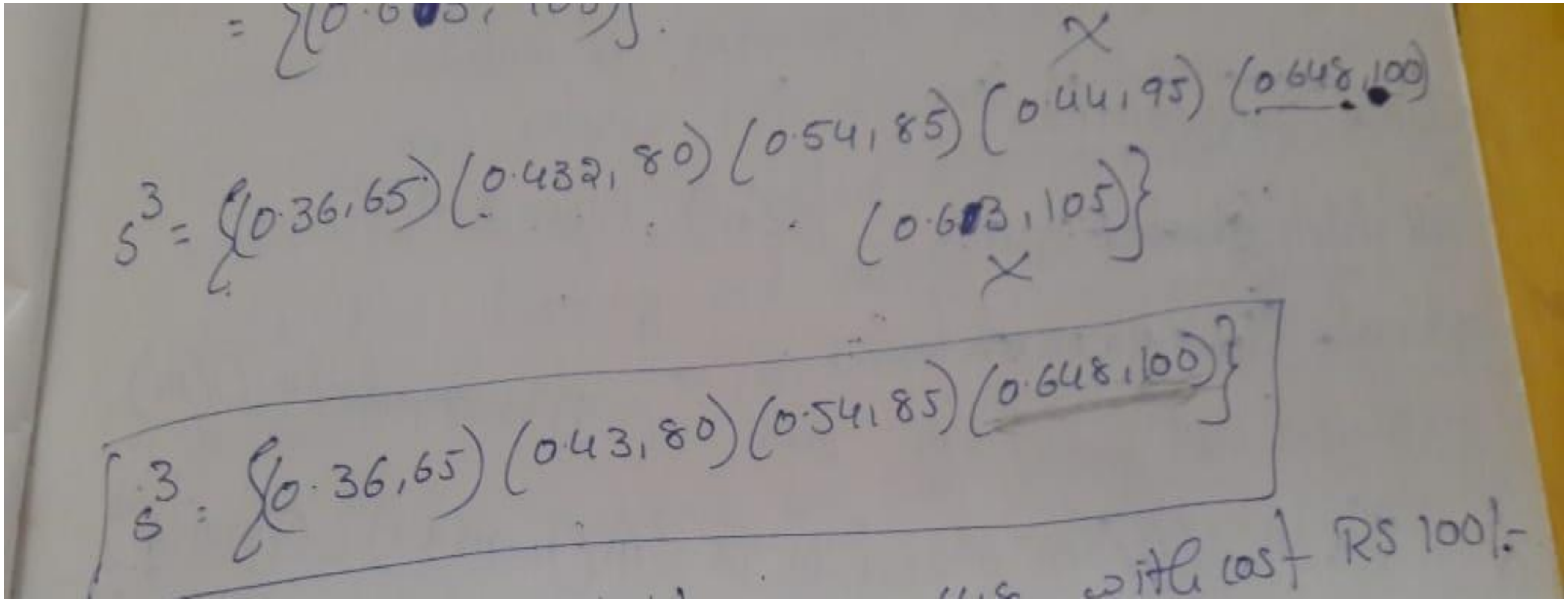
$S_3^3 = \{(0.72\times0.875, 105)(0.864\times0.875, 120)(0.89\times0.87, 135)(0.98\times0.875, 165)\}.$
$(0.95\times0.875, 150)$

$= \{(0.6$[illegible]$3, 105)\}.$

$(0.44, 95)\ (0.648, 10$

$)\ (0.54, 85)$

Example for Reliability Design Problem



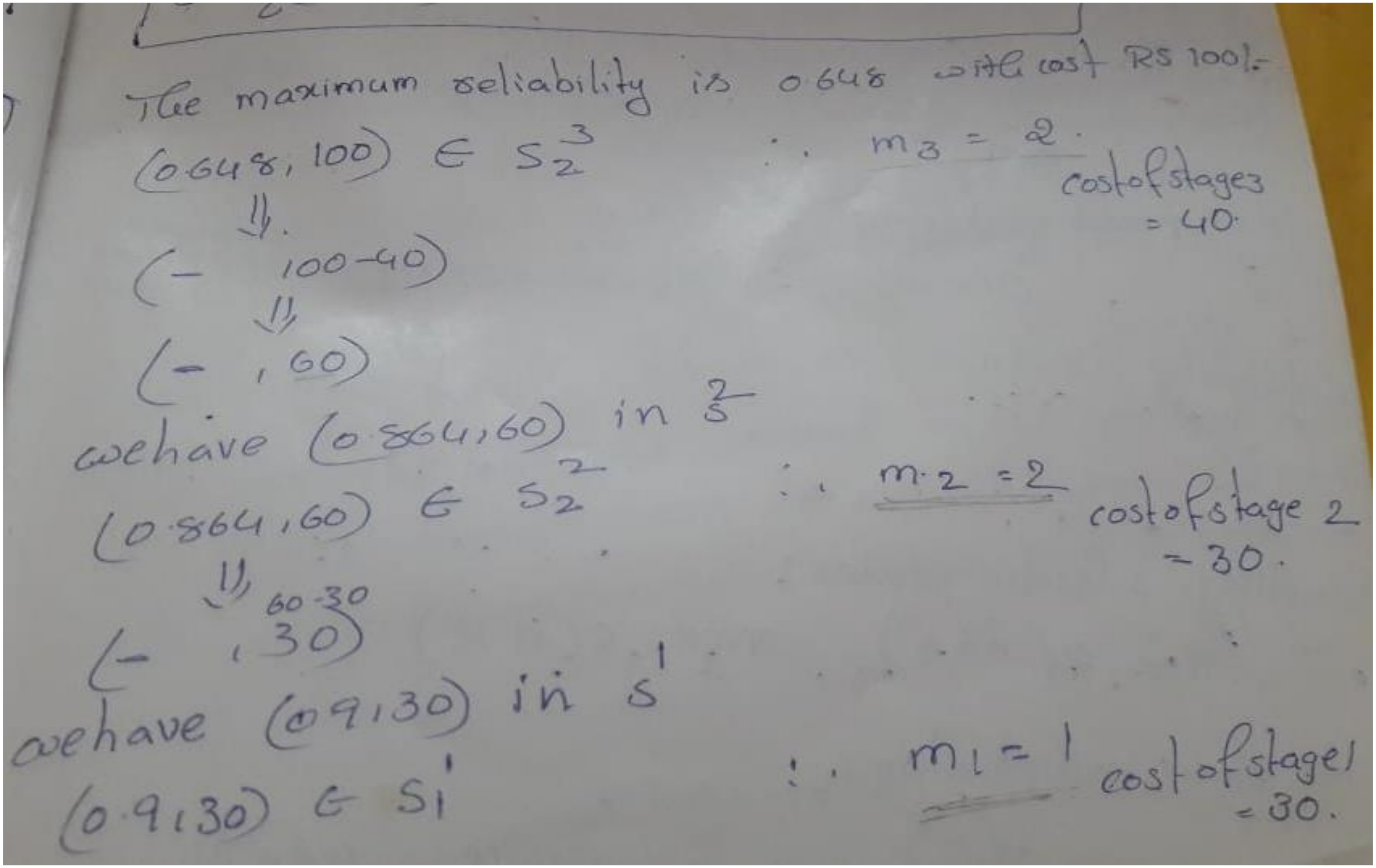
$= \{(0.6[illegible]3, 100)\}.$

$S^3 = \{(0.36, 65)(0.432, 80)(0.54, 85)\cancel{(0.44, 95)}(0.648, 100)$
$\cancel{(0.6[illegible]3, 105)}\}$

$$\boxed{S^3 : \{(0.36, 65)(0.43, 80)(0.54, 85)(0.648, 100)\}}$$

with cost RS 100/-

Example for Reliability Design Problem



The maximum reliability is 0.648 with cost RS 100/-

$(0.648, 100) \in S_2^3$ $\therefore$ $m_3 = 2$. cost of stage 3 = 40.

$\Downarrow$

$(-\ \ 100-40)$

$\Downarrow$

$(-\ , 60)$

we have $(0.864, 60)$ in S^2

$(0.864, 60) \in S_2^2$ $\therefore$ $m_2 = 2$ cost of stage 2 = 30.

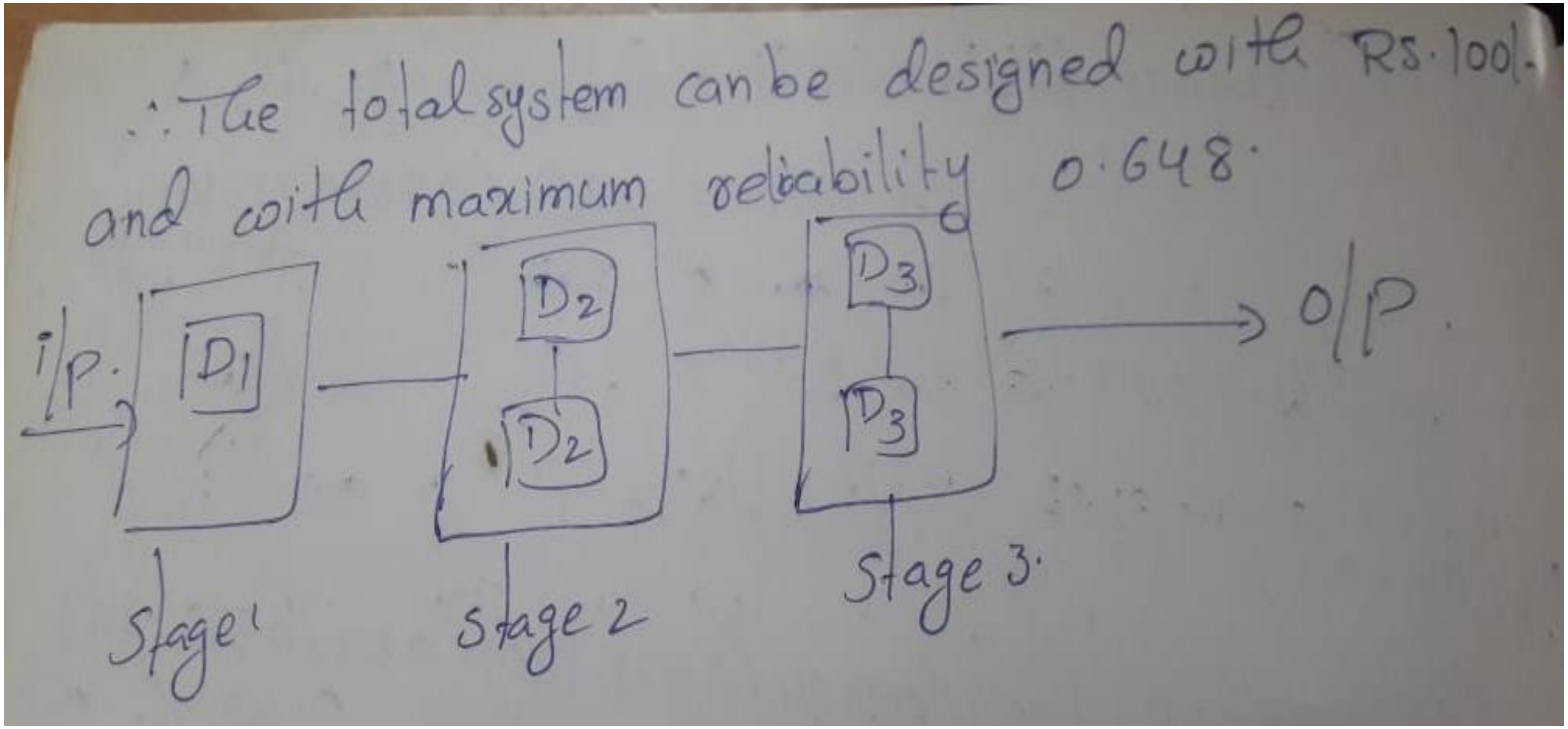
$\Downarrow$

$(-\ , \overset{60-30}{30})$

we have $(0.9, 30)$ in S^1

$(0.9, 30) \in S_1^1$ $\therefore$ $m_1 = 1$ cost of stage 1 = 30.

Example for Reliability



∴ The total system can be designed with Rs. 100/- and with maximum reliability 0.648.

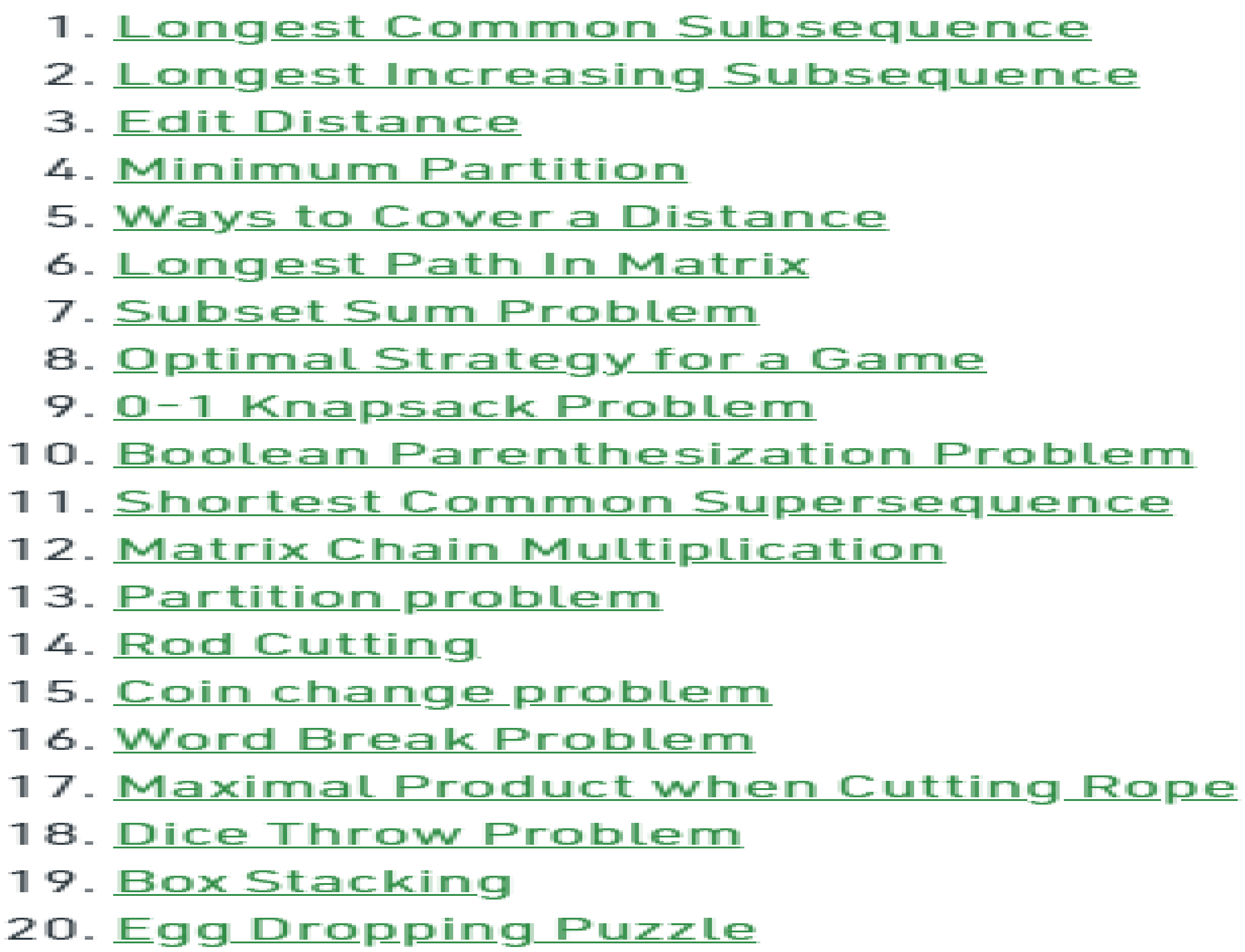
Example for Practice

- Design a system with maximum reliability with the following information.

  c1=50, c2=60, c3=30 and C=210

  r1=0.7, r2=0.3, r3=0.9.

Frequently asked interview questions on Dynamic programming



1. Longest Common Subsequence
2. Longest Increasing Subsequence
3. Edit Distance
4. Minimum Partition
5. Ways to Cover a Distance
6. Longest Path In Matrix
7. Subset Sum Problem
8. Optimal Strategy for a Game
9. 0-1 Knapsack Problem
10. Boolean Parenthesization Problem
11. Shortest Common Supersequence
12. Matrix Chain Multiplication
13. Partition problem
14. Rod Cutting
15. Coin change problem
16. Word Break Problem
17. Maximal Product when Cutting Rope
18. Dice Throw Problem
19. Box Stacking
20. Egg Dropping Puzzle

Topics

- General method
- Travelling sales person problem
- All pairs shortest Path problem
- 0/1 knapsack problem
- Reliability Design Problem