Dynamic Programming

Joy Mukherjee

Edit Distance Problem

- Source string (src)
- Target string (tgt)
- Assumption: Each character is an English Lowercase alphabet.
- Objective: To convert src to tgt with minimum cost

Operations	Cost
Insert	1
Delete	1
Replace	2

Example

- src = “tea” tgt = “eat”
- Delete ‘t’ at src[0]
- Insert ‘t’ at src[2]

Algorithm

- Cost[i][j] = Min Cost to convert src[0..i-1] to tgt[0..j-1]
- Length of source string is n
- Length of target string is m

- Src = “tea” Tgt = “”…. Deletion of n characters
- Src = “” Tgt = “eat” ….. Insertion of m characters

Algorithm

- Cost[i][j] = Min Cost to convert src[0..i-1] to tgt[0..j-1]
- Length of source string is n
- Length of target string is m
- Cost[i][0] = i
- Cost[0][j] = j

Algorithm

- Cost[i][j] = Min Cost to convert src[0..i-1] to tgt[0..j-1]
- To populate Cost[i][j], we have to investigate the relation of src[i-1] and tgt[j-1]
- If (src[i-1] == tgt[j-1]) Cost[i][j] = Cost[i-1][j-1], where Cost[i-1][j-1] = Min Cost to convert src[0..i-2] to tgt[0..j-2]
- Else {

  Cost[i][j] = minimum cost of three operations,

  1. Replace src[i-1] with tgt[j-1] = 2 + Cost[i-1[j-1]
  2. Insert tgt[j-1] = 1 + Cost[i][j-1]
  3. Delete src[i-1] = 1 + Cost[i-1][j]

  }

Base Cases

2-D array Cost[1+n][1+m], where n = strlen(src), m = strlen(tgt)

Cost	0	E (1)	A (2)	T (3)
0	0	1	2	3
T (1)	1			
E (2)	2			
A (3)	3			

Execution

Cost	0	E (1)	A (2)	T (3)
0	0	1	2	3
T (1)	1	2	3	2
E (2)	2	1	2	3
A (3)	3	2	1	2

Cost	0	E (1)	A (2)	T (3)
0	0	1	2	3
T (1)	1	2	3	2
E (2)	2	1	2	3
A (3)	3	2	1	2

Insertion of T into Tgt at 3rd position
No Operation
No Operation
Deletion of T from Src at 1st position

Egg Dropping Puzzle

- K-storied building
- N eggs
- Objective: To find out the critical floor with minimum no of attempts
- Critical floor: The highest floor from which an egg can be dropped without breaking
- Assumptions:
- If an egg is not broken for i-th floor, then it won’t be broken while being dropped from 1 to (i-1)-th floor.
- If an egg is broken for i-th floor, then it will be broken while being dropped from (i+1)-th to K-th floor.

Egg Dropping Puzzle

Floor
7
6
5
4
3
2
1

If N = 1, then No of attempts = K (No of floors in the building)

Drop an egg from x-th floor, where $1 \leq x \leq K$

Case 1: The egg is broken

1 + Opt(N-1, x-1) where $1 \leq x \leq K$

Case 2: The egg is not broken

1 + Opt(N, K-x) where $1 \leq x \leq K$

Maximum (1 + Opt(N-1, x-1), 1 + Opt(N, K-x)) where $1 \leq x \leq K$

Opt(N, K) = $\text{Minimum}_{1 \leq x \leq K}$[Maximum (1 + Opt(N-1, x-1), 1 + Opt(N, K-x))]

Base Cases

	0	1	2	3	4	5	6
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	**0**	**1**	**2**	**3**	**4**	**5**	**6**
2	**0**						

Execution

Min	0	1	2	3	4	5	6
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	**0**	**1**	**2**	**3**	**4**	**5**	**6**
2	**0**	**1**	**2**				

Drop 1st egg from 1st floor = Max{1 + 0, 1 + 1) = 2
Drop 1st egg from 2nd floor = Max{1 + 1, 1 + 0) = 2

Execution

Min	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6
2	0	1	2	2			

Drop 1st egg from 1st floor = Max{1 + 0, 1 + 2) = 3
Drop 1st egg from 2nd floor = Max{1 + 1, 1 + 1) = 2
Drop 1st egg from 3rd floor = Max{1 + 2, 1 + 0) = 3

Execution

Min	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6
2	0	1	2	2	3		

Drop 1st egg from 1st floor = Max{1 + 0, 1 + 2) = 3
Drop 1st egg from 2nd floor = Max{1 + 1, 1 + 2) = 3
Drop 1st egg from 3rd floor = Max{1 + 2, 1 + 1) = 3
Drop 1st egg from 4th floor = Max{1 + 3, 1 + 0) = 4

Execution

Min	0	1	2	3	4	5	6
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	**0**	**1**	**2**	**3**	**4**	**5**	**6**
2	**0**	**1**	**2**	**2**	**3**	**3**	

Drop 1st egg from 1st floor = Max{1 + 0, 1 + 3) = 4
Drop 1st egg from 2nd floor = Max{1 + 1, 1 + 2) = 3
Drop 1st egg from 3rd floor = Max{1 + 2, 1 + 2) = 3
Drop 1st egg from 4th floor = Max{1 + 3, 1 + 1) = 4
Drop 1st egg from 5th floor = Max{1 + 4, 1 + 0) = 5

Execution

Min	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6
2	0	1	2	2	3	3	3

Drop 1^{st} egg from 1^{st} floor = Max{1 + 0, 1 + 3) = 4
Drop 1^{st} egg from 2^{nd} floor = Max{1 + 1, 1 + 3) = 4
Drop 1^{st} egg from 3^{rd} floor = Max{1 + 2, 1 + 2) = 3
Drop 1^{st} egg from 4^{th} floor = Max{1 + 3, 1 + 2) = 4
Drop 1^{st} egg from 5^{th} floor = Max{1 + 4, 1 + 1) = 5
Drop 1^{st} egg from 6^{th} floor = Max{1 + 5, 1 + 0) = 6

Time Complexity = $O(NK^2)$

An Interview Question

- Given a 100-storied building and 2 eggs, what is the minimum number of attempts to find out the highest floor from which dropping an egg doesn’t cost its damage.
- Solution: Try from 10, 20, 30,…100-th floor
- Worst case: If the egg doesn’t break at 10, 20,…,90-th floor, but breaks while dropping from the 100-th floor, we have to try with the second egg by dropping it from 91 to 99-th floor to find out the critical floor.
- It results in 19 attempts.

Can we do better?

- How to choose the first attempt/to select the correct floor?
- Say it is x-th floor.
- Case 1: Egg breaks at x-th floor
- No of attempts = 1 + (x-1) = x attempts
- Case 2:
- Egg doesn’t break in the 1^{st} attempt, what will be my second floor?
- X + (x-1) = 2x -1-th floor
- Egg doesn’t break in the 2^{nd} attempt, what will be my thirdd floor?
- X + (x-1) + (x-2)= 3x - 3-th floor

Solution

- Total no of attempts
- $x + (x-1) + (x-2) + \ldots. + 1 >= 100$
- $x(x+1)/2 >= 100$
- $x^2 + x - 200 >= 0$
- $x >= 14$
- Ans: 14 attempts in the worst case

- 1^{st} attempt 14^{th} floor
- 2^{nd} attempt 27^{th} floor
- 3^{rd} attempt 39^{th} floor
- 4^{th} attempt 50^{th} floor
- 60, 69, 77, 84, 90, 95, 99-th floor
- If it breaks at 99-th floor, then try out 96, 97, and 98-th floor to find out the critical floor.

Weighted Job Scheduling

- A uniprocessor (execute one job at a time)
- N different jobs <start time, end time, profit>
- Objective: To maximize the total profit by executing a subset of non-overlapping jobs
- Interval is modelled as [start time, end time)

Weighted Job Scheduling

1. Sort the jobs in the non-decreasing order of end time.

Job	Start Time	End Time	Profit
1	1	3	5
2	2	5	6
3	4	6	5
4	6	7	4
5	5	8	11
6	7	9	2

Weighted Job Scheduling

1. Sort the jobs in the non-decreasing order of end time.
2. Result[i] = Maximum total profit gained, where job i is the last executed job
3. Find the maximum in the result array

Job	Start Time	End Time	Profit
1	1	3	5
2	2	5	6
3	4	6	5
4	6	7	4
5	5	8	11
6	7	9	2

	1	2	3	4	5	6
Result	5	6	10	14	17	16

Weighted Job Scheduling

1. Sort the jobs in the non-decreasing order of end time.
2. Create an array Result[n], where Result[i] represents the maximum total profit gained, where job i is the last executed job
3. Result[0] = job[0].profit
4. For(i = 1; i < N; i++) {

```
        For(j = 0; j < i; j++) {
                If(job[j].end <= job[i].start) //non-overlapping {
                        Result[i] = max(Result[i], Result[j] + job[i].profit);
                }
        }
```

5. }
6. Find the maximum in the result array

Time Complexity = O(n^2)

Weighted Job Scheduling: A better algorithm

1. Sort the jobs in the non-decreasing order of end time.

Job	Start Time	End Time	Profit
1	1	3	5
2	2	5	6
3	4	6	5
4	6	7	4
5	5	8	11
6	7	9	2

Weighted Job Scheduling

1. Sort the jobs in the non-decreasing order of end time.
2. Result[i] = Maximum total profit gained for the first i jobs
3. Result[i] = max{Include i-th job, Exclude i-th job}
4. Exclude i-th job = Result[i-1]
5. Include i-th job = Profit[i] + Result[index of the latest non-overlapping job]
6. Return Result[n-1]

Job	Start Time	End Time	Profit
1	1	3	5
2	2	5	6
3	4	6	5
4	6	7	4
5	5	8	11
6	7	9	2

Weighted Job Scheduling

1. Sort the jobs in the non-decreasing order of end time.
2. Result[i] = Maximum total profit gained for the first i jobs
3. Result[i] = max{Include i-th job, Exclude i-th job}
4. Return Result[n-1]

Job	Start Time	End Time	Profit
1	1	3	5
2	2	5	6
3	4	6	5
4	6	7	4
5	5	8	11
6	7	9	2

	1	2	3	4	5	6
Result	5	6	10	14	17	Max (14+2,17) = 17

Weighted Job Scheduling

1. Sort the jobs in the non-decreasing order of end time.
2. Create an array Result[n], where Result[i] represents the maximum total profit gained for the first i-jobs
3. Result[0] = job[0].profit
4. For(i = 1; i < N; i++) {

   Result[i] = max(Result[i-1], Result[index of the latest non-overlapping job] + job[i].profit)

   }

}

5. Return Result[n-1]

Time Complexity = O(n log n)