A&D

Exercise Session 13

Nil Ozer

A&D Overview

A&D

Math Basics
- Asymp. Notation
- Induction
- Loop-Counting

Max-Subarray-Sum
- naive, divide-conquer, inductive
- complexity

Searchs
- Linear Search
- Binary Search
- Lower-Bound

Sorts
- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quicksort
- Heapsort
- Lower-Bound

Data Structures
- Array
- List (Linked, Doubly)
- Trees (BST, Heap Tree, AVL Tree)

DP
- Fibonacci
- Max-Subarray-Sum
- Jump-Game
- Longest-Common-Subseq.
- Edit-Distance
- Subset Sum
- Knapsack
- Longest Ascending Subseq.

Graph Definitions
- Definitions
- Eulerian Tours
- Topological Sorting

Graph Searchs
- DFS
- BFS

Shortest Paths

one-to-all
- BFS usage
- Dijkstra
- Bellman-Ford
- negative-closed w. detection

all-to-all
- One-to-all usage
- Floyd-Warshall
- Johnson
- #walks using A_G

- overview

MST
- Prim
- Boruvka
- Kruskal

Outline

- Shortest Paths - all to all
- DP Recap
- Exam preparation session organization

Shortest Paths

→ Shortest Paths

one-to-all

- ↳ BFS usage
- ↳ Dijkstra
- ↳ Bellman-Ford
- ↳ negative-closed w. detection

all-to-all

- ↳ one-to-all usage
- ↳ Floyd-Warshall
- ↳ Johnson
- ↳ #walks using A_G

↳ overview

Recap

Shortest Paths (one - to - all)

G (directed/undirected)	Algorithm	Runtime
unweighted , all edges with the same positive weight	BFS usage	O (|V| + |E|)
weighted , nonnegative edge weights $c(e) \geq 0$	Dijsktra	O ((|V| + |E|) * log n)
weighted, positive and (possibly) negative edge weights $c(e) \in \mathbb{R}$	Belmann-Ford	O (|V| * |E|)
G has no cycles	topological sorting + DP	O (|V| + |E|)

Shortest Paths

All-to-all

G (directed/undirected)	Algorithm	Runtime	
unweighted , all edges with the same positive weight	n x BFS	$O(\|V\| * (\|V\| + \|E\|))$	
weighted , nonnegative edge weights $c(e) \geq 0$	n x Dijsktra	$O(\|V\| * (\|V\| + \|E\|) * \log(\|V\|))$ $O(\|V\|*\|E\| + \|V\|^2 \log(\|V\|))$	with Fibonacci-Heap
weighted, positive and (possibly) negative edge weights $c(e) \in \mathbb{R}$	n x Belmann-Ford	$O(\|V\| * \|V\| * \|E\|)$	
	Floyd - Warshall	$O(\|V\|^3)$	
weighted, positive and (possibly) negative edge weights $c(e) \in \mathbb{R}$, no negative cycles	Johnson	$O(\|V\| * (\|V\| + \|E\|) * \log n)$ $O(\|V\|*\|E\| + \|V\|^2 \log(\|V\|))$	with Fibonacci-Heap

All-to-all Shortest Paths

Floyd-Warshall

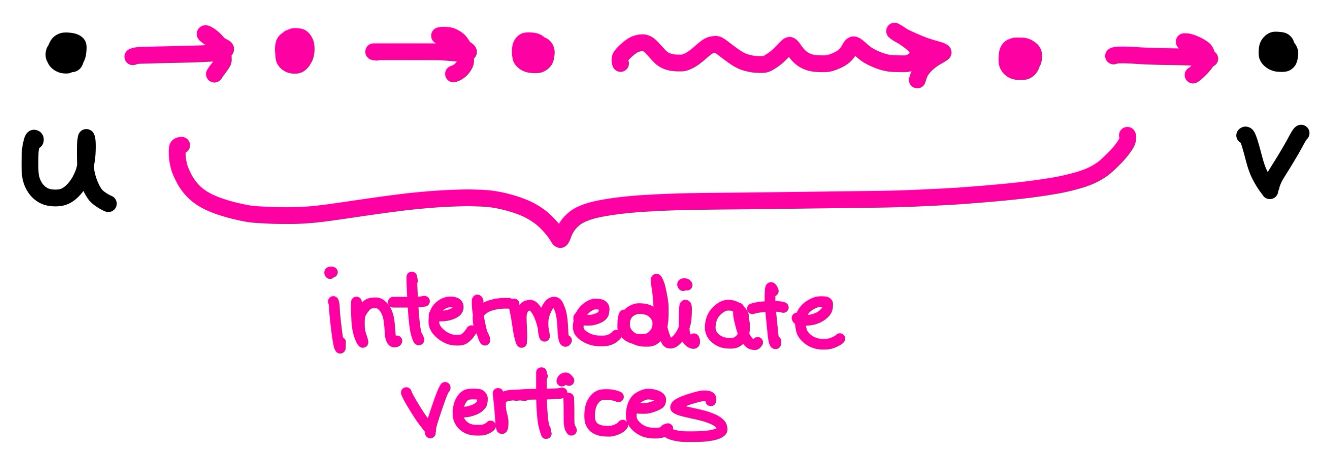
Idea : Two things can happen about a vertex i, considering a walk from vertex u to v

- i does not get used in walk u-v
- i gets used in walk u-v

Definition of the DP table :

DP[i][u][v] = "The length of the shortest u-v walk that only uses the intermediate vertices from {1...i}"

intermediate vertices :



All-to-all Shortest Paths

Floyd-Warshall

DP[i][u][v] = "The length of the shortest u-v walk that only uses the intermediate vertices from {1...i}"

$\textsc{Floyd-Warshall}(G = (V, E), c)$

$DP[0][u][v] \leftarrow \begin{cases} 0 & \text{falls } u = v \text{ ist,} \\ c(u,v) & \text{falls } (u,v) \in E \text{ ist,} \\ \infty & \text{sonst} \end{cases}$

no need to walk, we're already there

just walk that edge, you don't use any intermediate vertices

you can't reach v without using intermediate vertices

for $i \leftarrow 1, \dots, n$ **do**

  for $u \leftarrow 1, \dots, n$ **do**

    for $v \leftarrow 1, \dots, n$ **do**

      $DP[i][u][v] \leftarrow \min(\underbrace{DP[i-1][u][v]}_{\text{don't use vertex i}}, \underbrace{DP[i-1][u][i] + DP[i-1][i][v]}_{\text{use vertex i}})$

return DP

All-to-all Shortest Paths

Floyd-Warshall

DP[i][u][v] = "The length of the shortest u-v walk that only uses the intermediate vertices from {1...i}"

FLOYD-WARSHALL$(G = (V, E), c)$

$DP[0][u][v] \leftarrow \begin{cases} 0 & \text{falls } u = v \text{ ist,} \\ c(u,v) & \text{falls } (u,v) \in E \text{ ist,} \\ \infty & \text{sonst} \end{cases}$

no need to walk, we're already there

just walk that edge, you don't use any intermediate vertices

you can't reach v without using intermediate vertices

for $i \leftarrow 1, \ldots, n$ **do**

  for $u \leftarrow 1, \ldots, n$ **do**

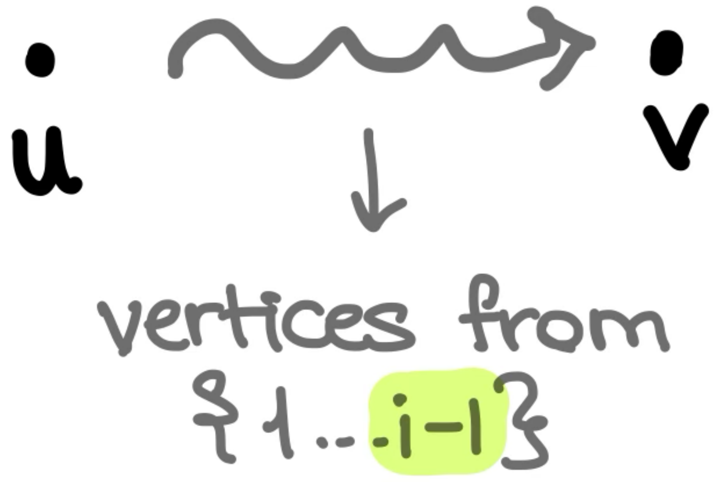
    for $v \leftarrow 1, \ldots, n$ **do**

      $DP[i][u][v] \leftarrow \min(\underbrace{DP[i-1][u][v]}_{\text{don't use vertex i}}, \underbrace{DP[i-1][u][i] + DP[i-1][i][v]}_{\text{use vertex i}})$

return DP

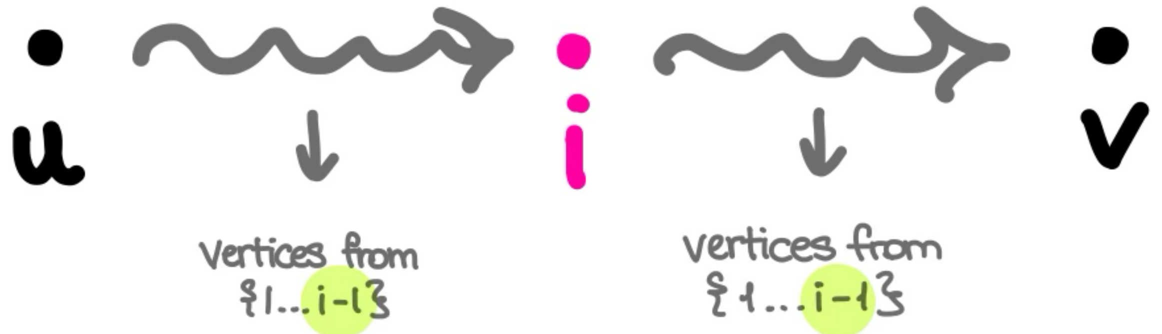
DP [i-1][u][v]

shortest walk from u to v using {1...i-1}



DP [i-1][u][i] + DP[i-1][i][v]

shortest walk from u to i using {1...i-1} + shortest walk from i to v using {1...i-1}



All-to-all Shortest Paths

Floyd-Warshall

DP[i][u][v] = "The length of the shortest u-v walk that only uses the intermediate vertices from {1...i}"

Floyd-Warshall$(G=(V,E),c)$

$$DP[0][u][v] \leftarrow \begin{cases} 0 & \text{falls } u = v \text{ ist,} \\ c(u,v) & \text{falls } (u,v) \in E \text{ ist,} \\ \infty & \text{sonst} \end{cases}$$

no need to walk, we're already there

just walk that edge, you don't use any intermediate vertices

you can't reach v without using intermediate vertices

for $i \leftarrow 1, \ldots, n$ **do**

  for $u \leftarrow 1, \ldots, n$ **do**

    for $v \leftarrow 1, \ldots, n$ **do**

$$DP[i][u][v] \leftarrow \min(\underbrace{DP[i-1][u][v]}_{\text{don't use vertex i}}, \underbrace{DP[i-1][u][i] + DP[i-1][i][v]}_{\text{use vertex i}})$$

return DP

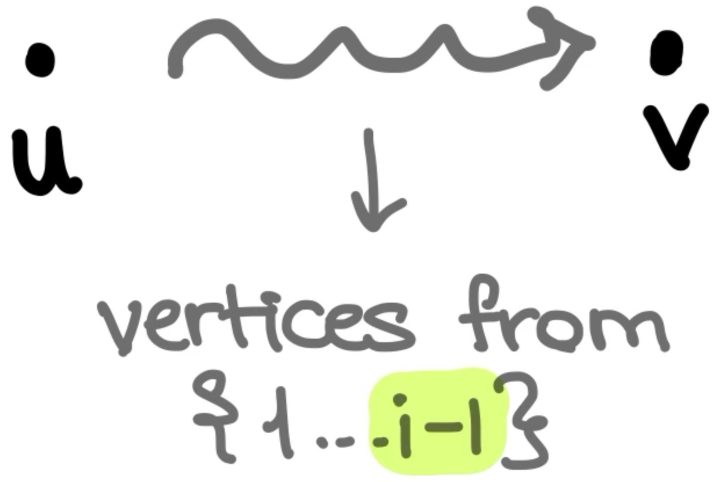
Runtime: $O(n^3)$

Solution at: DP[n][][]

"using all vertices"

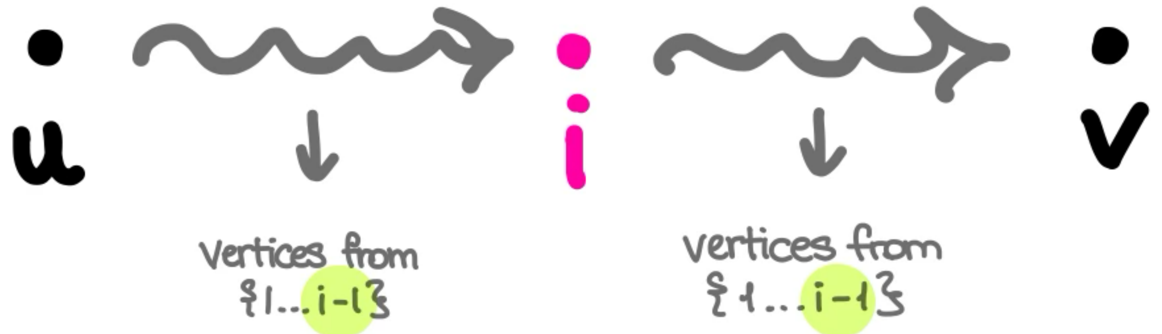
DP [i-1][u][v]

shortest walk from u to v using {1...i-1}



DP [i-1][u][i] + DP[i-1][i][v]

shortest walk from u to i using {1...i-1} + shortest walk from i to v using {1...i-1}



All-to-all Shortest Paths

Floyd-Warshall , Negative Closed Walk Detection

∃ a negative closed walk $\iff$ ∃ v with DP[n][v][v] < 0

DP[n][v][v] : The shortest walk from v to v using {1...n}

All-to-all Shortest Paths
Johnson

Problem is the negative edges ! (we can’t use dijkstra)

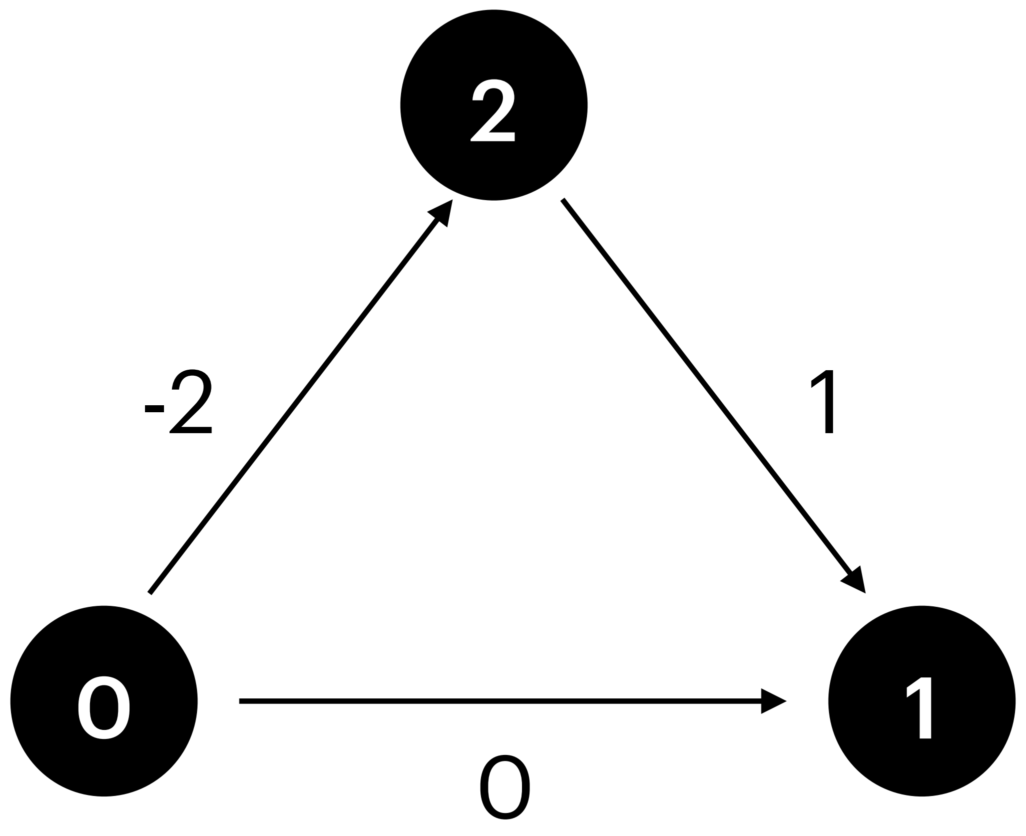
Idea :
- Make all edge weights ≥ 0
- n x Dijkstra

DOES NOT WORK WITH NEGATIVE CYCLES !

We know Dijkstra, how can we make all edge weights ≥ 0 ?

All-to-all Shortest Paths

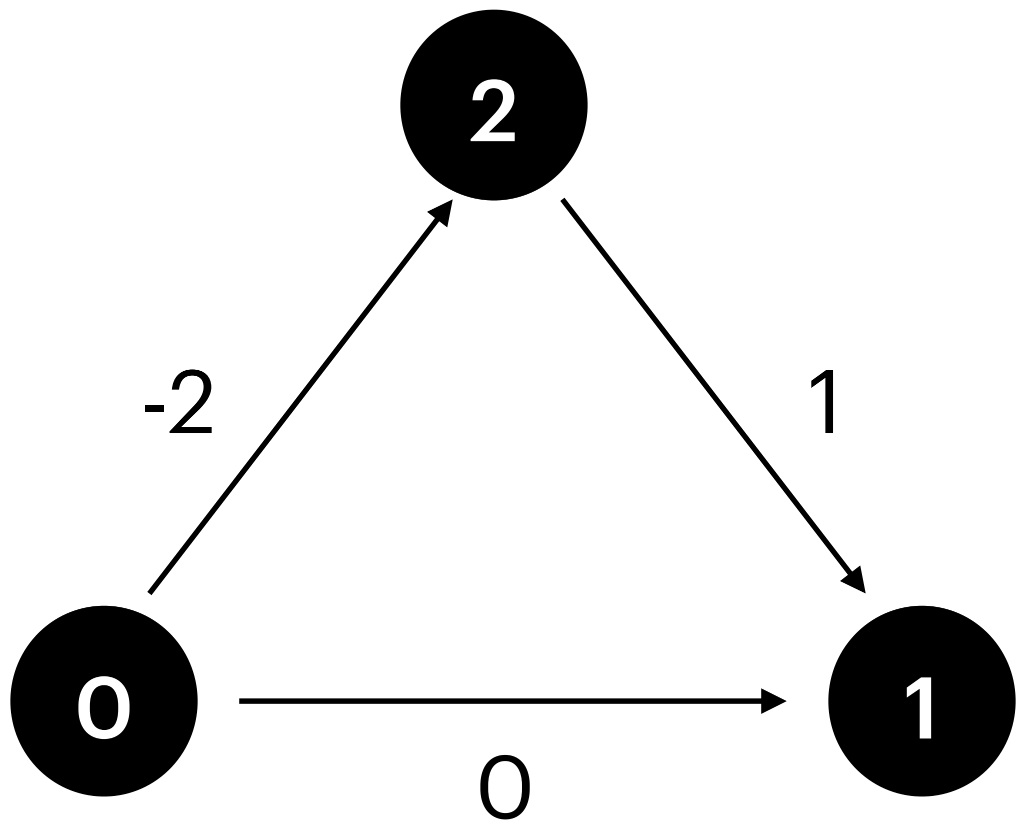
Johnson - Making all edge weights ≥ 0



All-to-all Shortest Paths

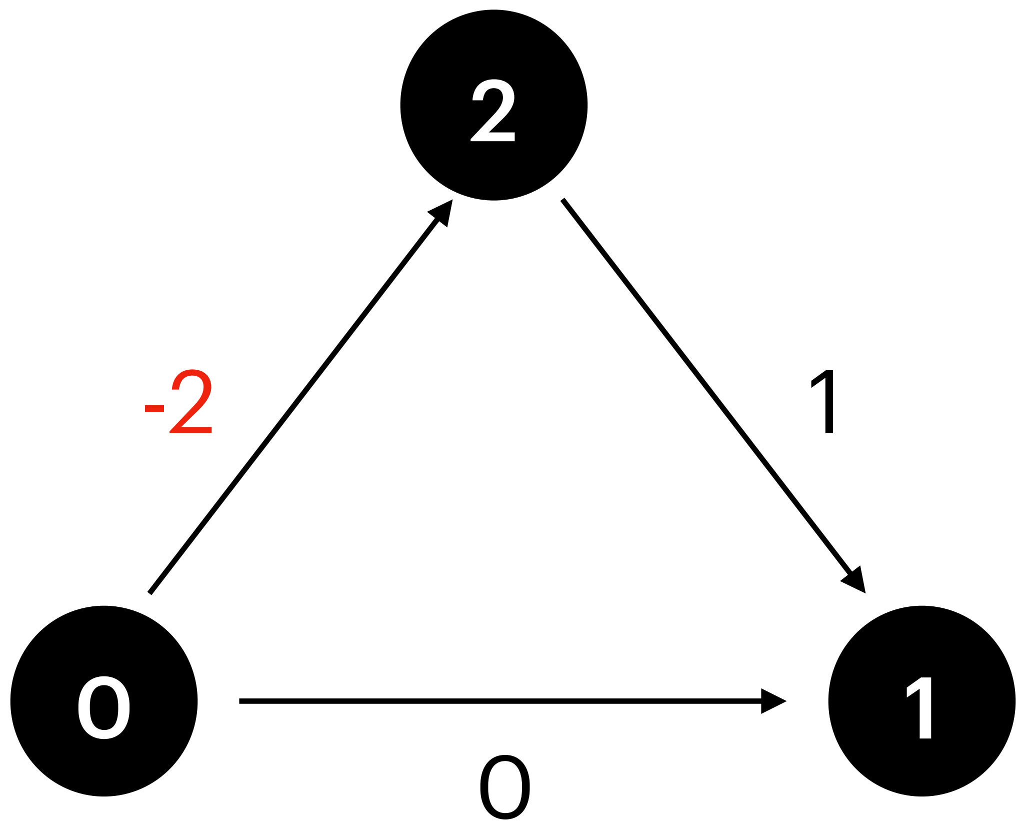
Johnson - Making all edge weights ≥ 0

$c(0,2) = -2$ $c(2,1) = 1$ $c(0,1) = 0$



All-to-all Shortest Paths

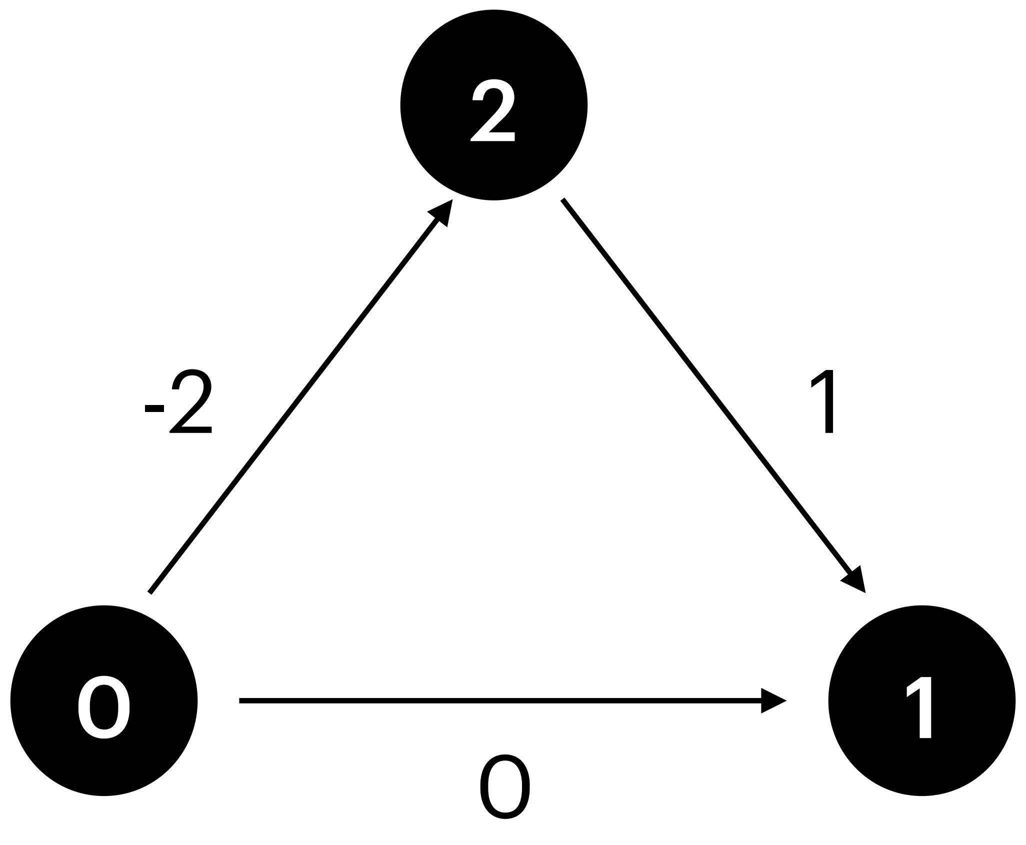
Johnson - Making all edge weights ≥ 0



All-to-all Shortest Paths

Johnson - Making all edge weights ≥ 0

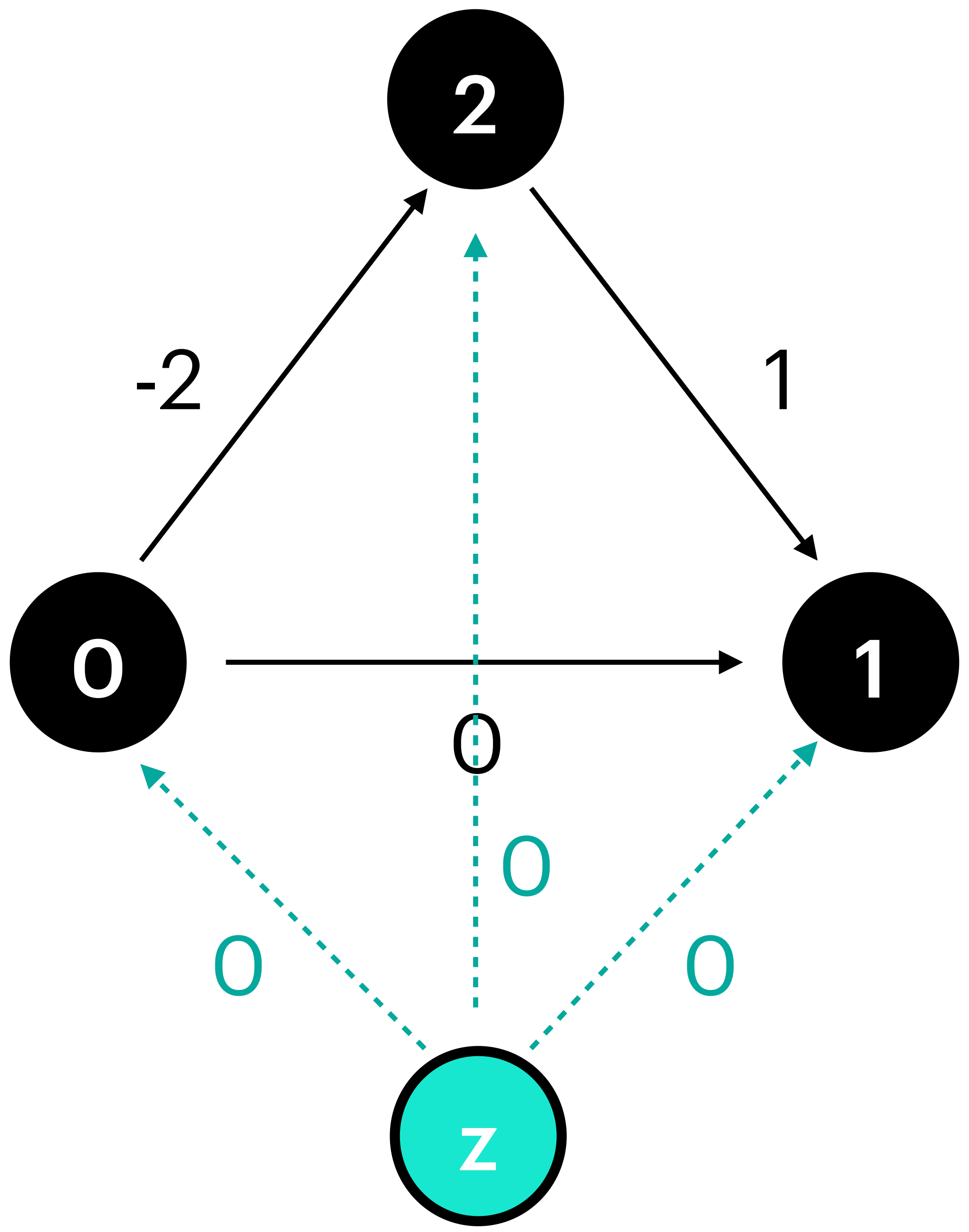
- Add a new vertex z , and connect it to every vertex in the original G with and edge with cost 0



All-to-all Shortest Paths

Johnson - Making all edge weights ≥ 0

- Add a new vertex z , and connect it to every vertex in the original G with and edge with cost 0



All-to-all Shortest Paths

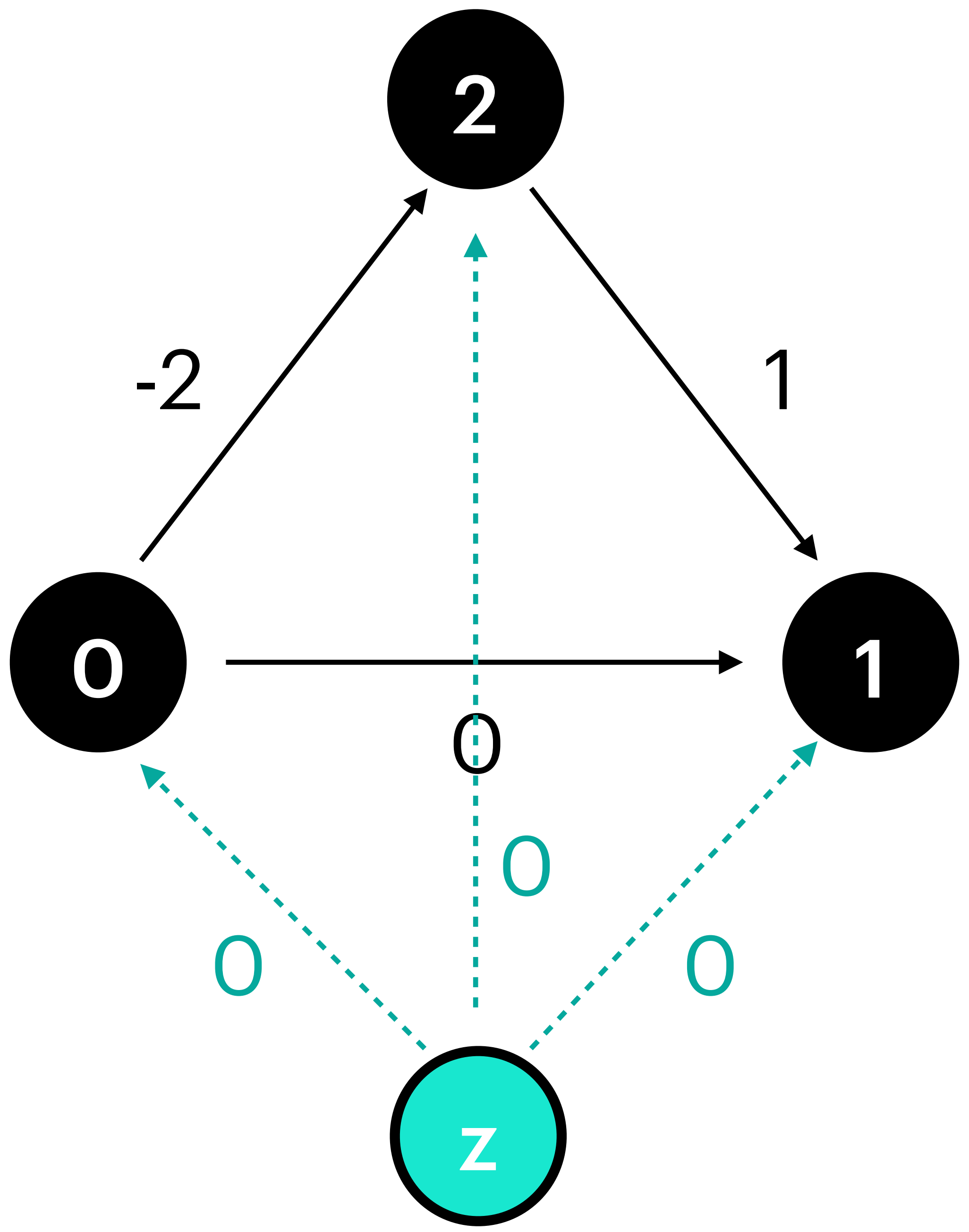
Johnson - Making all edge weights ≥ 0

- Add a new vertex z , and connect it to every vertex in the original G with and edge with cost 0

- Find h(u) for every u

  h(u) := length of the shortest path from z to u

with Bellman-Ford x1 from z



All-to-all Shortest Paths

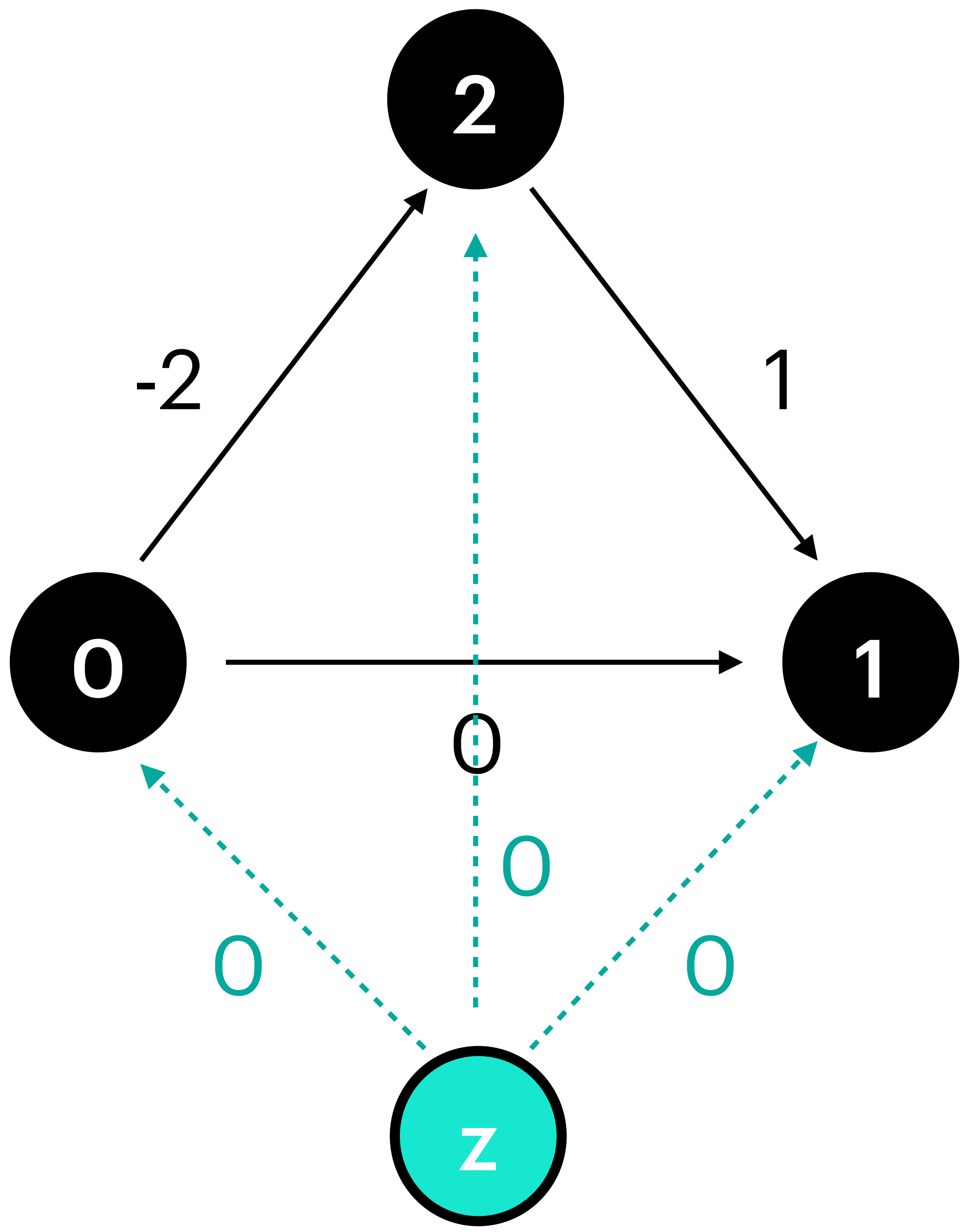
Johnson - Making all edge weights ≥ 0

- Add a new vertex z , and connect it to every vertex in the original G with and edge with cost 0

- Find h(u) for every u

with Bellman-Ford x1 from z

h(u) := length of the shortest path from z to u



All-to-all Shortest Paths

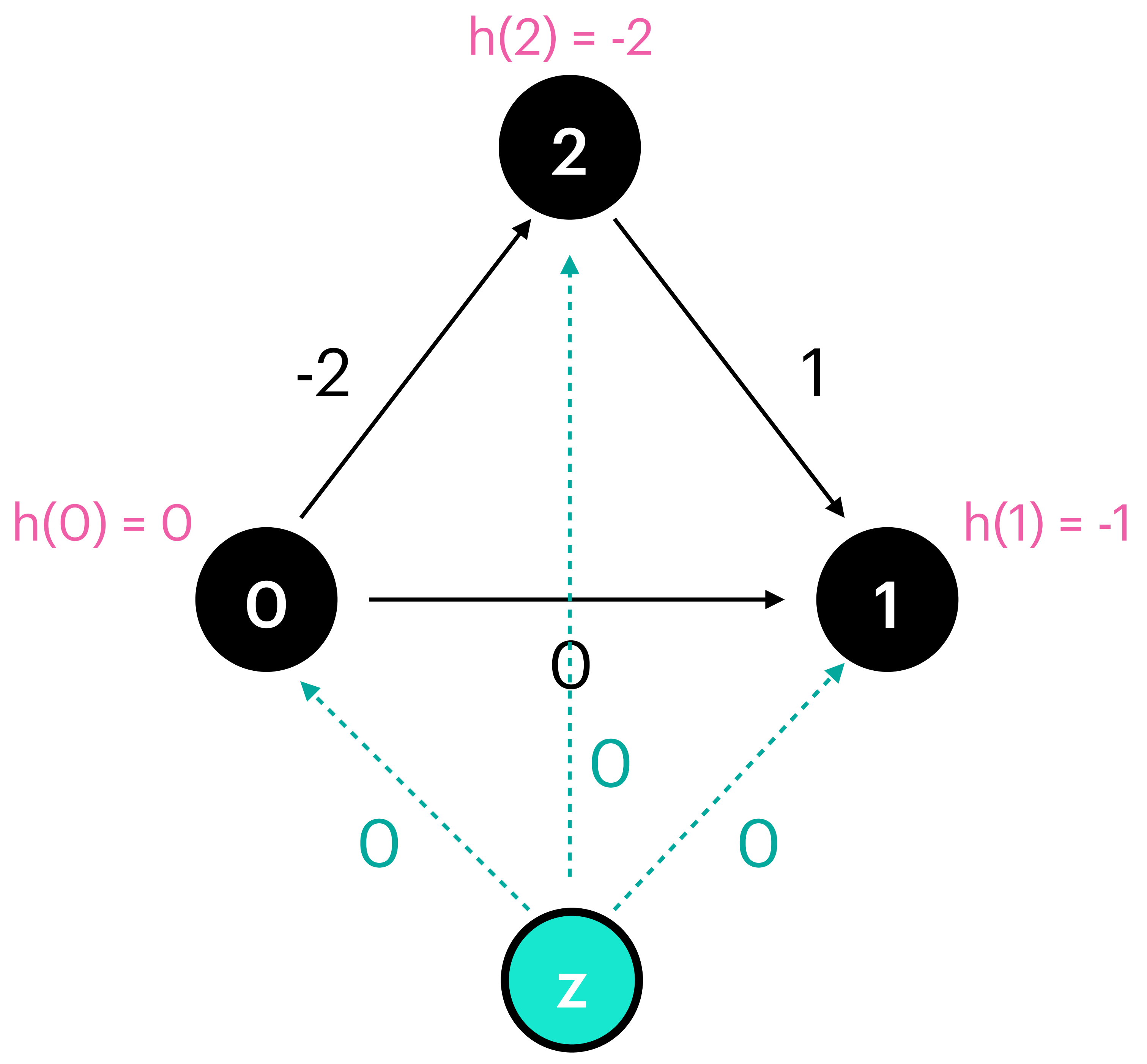
Johnson - Making all edge weights ≥ 0

- Add a new vertex z , and connect it to every vertex in the original G with and edge with cost 0

- Find h(u) for every u

  h(u) := length of the shortest path from z to u

with Bellman-Ford x1 from z



All-to-all Shortest Paths

Johnson - Making all edge weights ≥ 0

- Add a new vertex z , and connect it to every vertex in the original G with and edge with cost 0

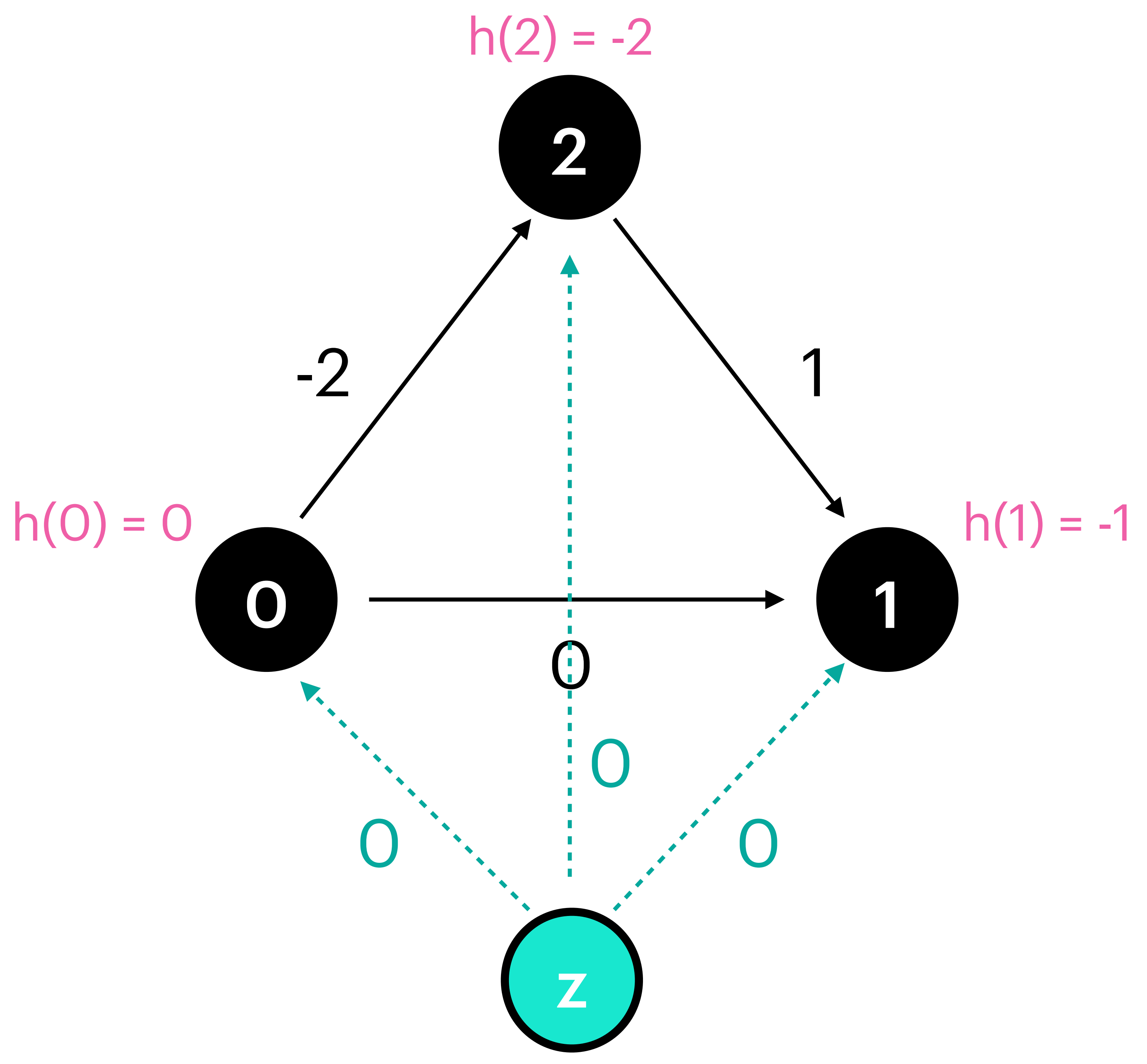
- Find h(u) for every u

  with Bellman-Ford x1 from z

  h(u) := length of the shortest path from z to u

- Calculate c′(u,v) for every edge

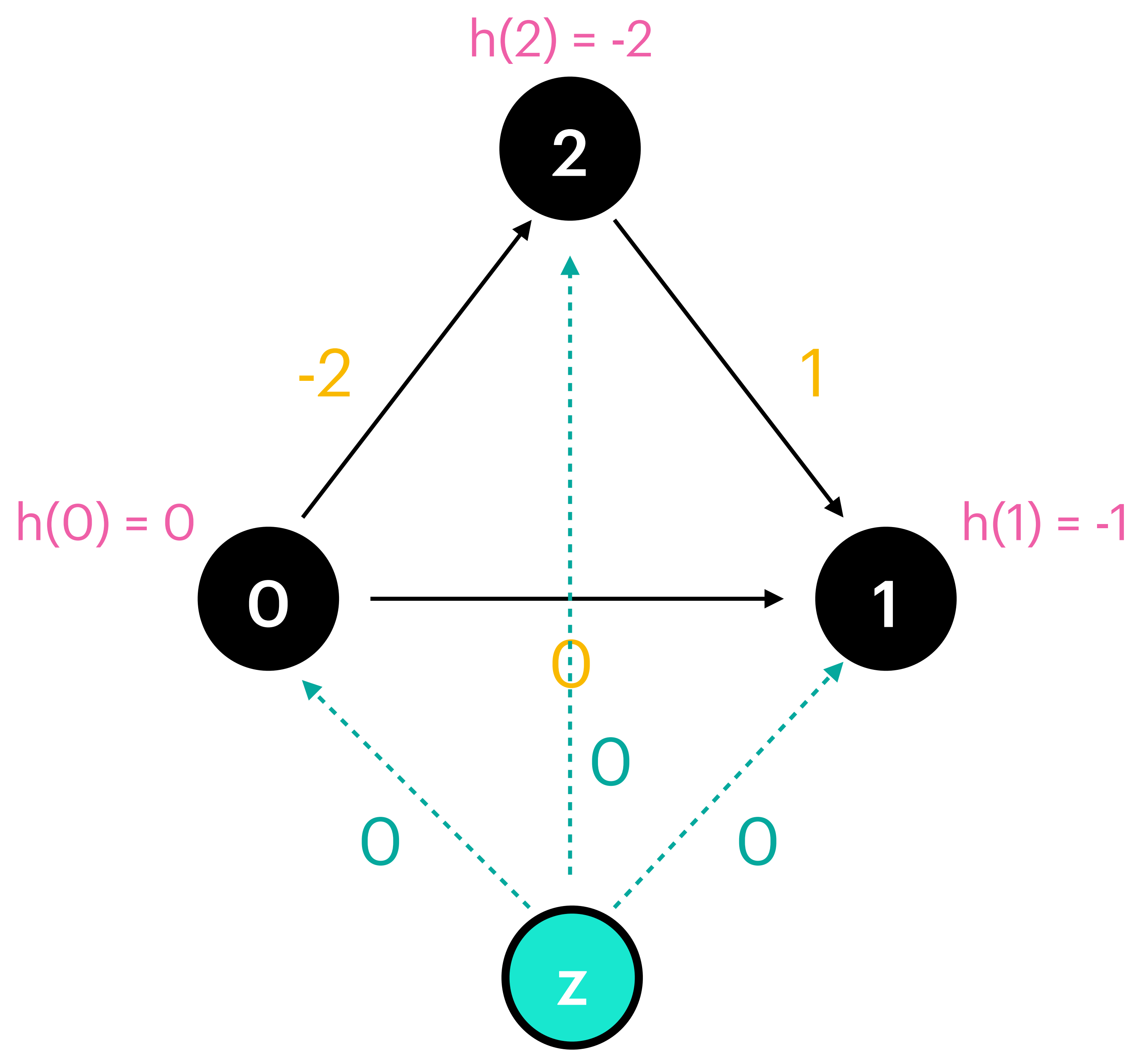
  c′(u,v) := c(u,v) + h(u) - h(v)



All-to-all Shortest Paths

Johnson - Making all edge weights ≥ 0

$c(0,2) = -2$ $c(2,1) = 1$ $c(0,1) = 0$



- Add a new vertex z , and connect it to every vertex in the original G with and edge with cost 0

- Find $h(u)$ for every u

  $h(u)$:= length of the shortest path from z to u

  with Bellman-Ford x1 from z

- Calculate $c'(u,v)$ for every edge

  $c'(u,v) := c(u,v) + h(u) - h(v)$

$c'(0,2) = c(0,2) + h(0) - h(2) = -2 + 0 - (-2) = 0$

$c'(0,1) = c(0,1) + h(0) - h(1) = 0 + 0 - (-1) = 1$

$c'(2,1) = c(2,1) + h(2) - h(1) = 1 + (-2) - (-1) = 0$

All-to-all Shortest Paths

Johnson - Making all edge weights ≥ 0

- Add a new vertex z , and connect it to every vertex in the original G with and edge with cost 0

- Find h(u) for every u

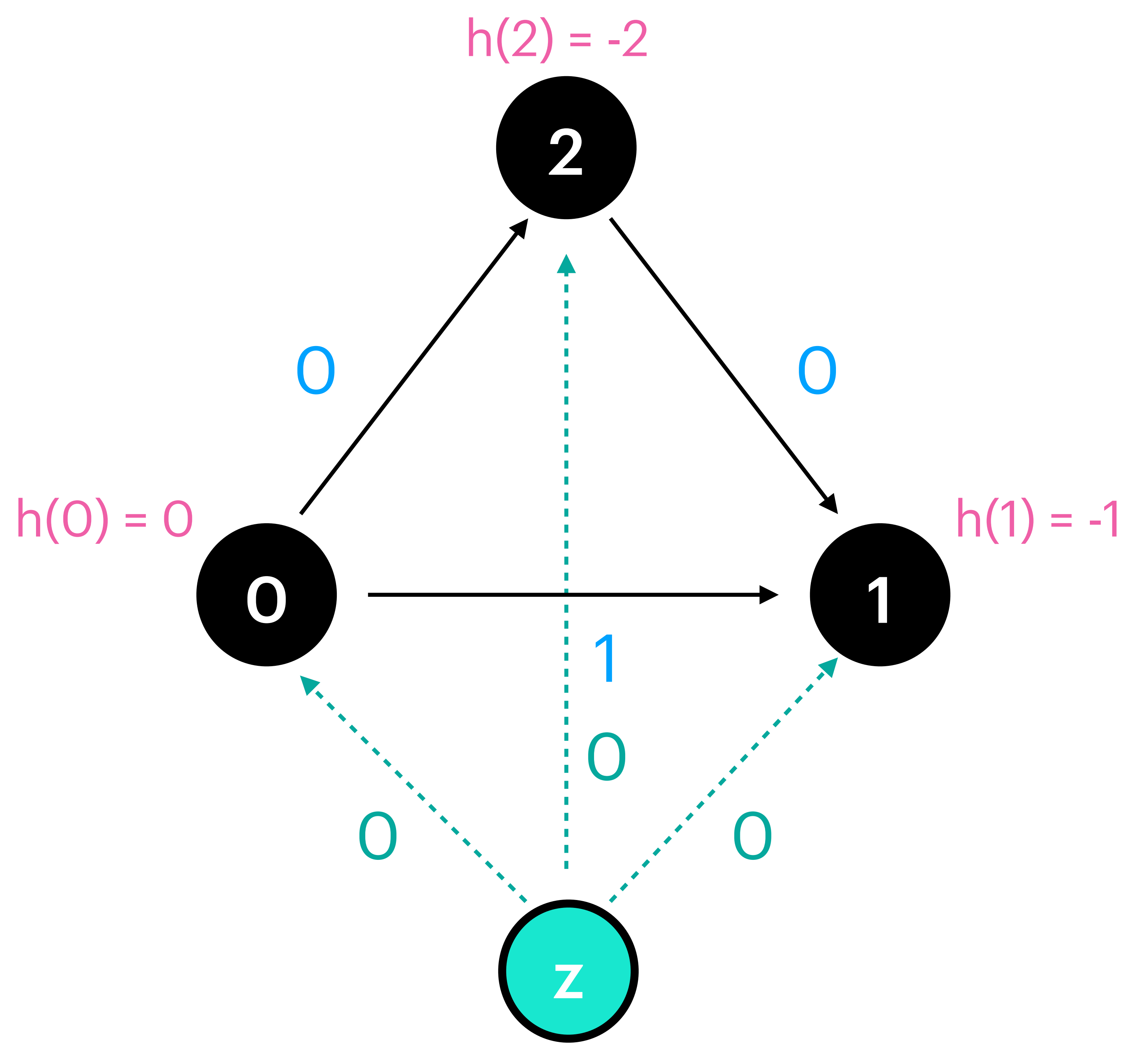
  with Bellman-Ford x1 from z

  h(u) := length of the shortest path from z to u

- Calculate c′(u,v) for every edge

  c′(u,v) := c(u,v) + h(u) - h(v)

c(0,2) = -2 c(2,1) = 1 c(0,1) = 0



c′(0,2) = c(0,2) + h(0) - h(2) = -2 + 0 - (-2) = 0

c′(0,1) = c(0,1) + h(0) - h(1) = 0 + 0 - (-1) = 1

c′(2,1) = c(2,1) + h(2) - h(1) = 1 + (-2) - (-1) = 0

All-to-all Shortest Paths

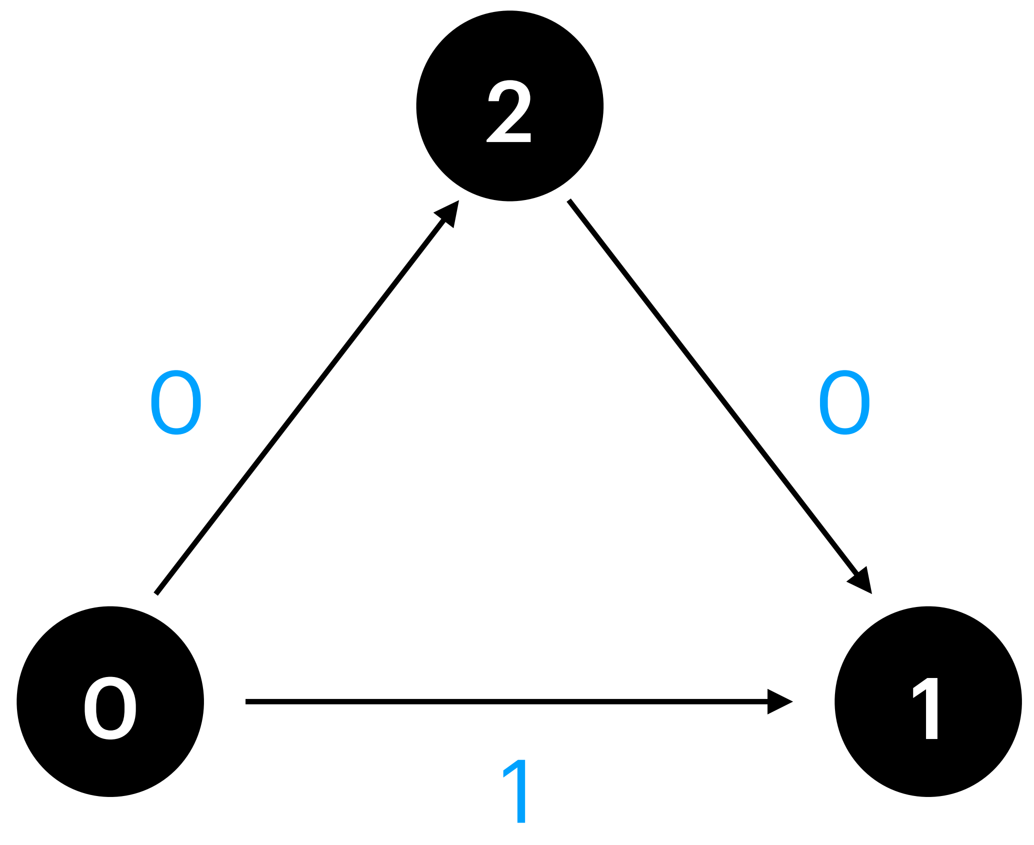
Johnson - Making all edge weights ≥ 0

- Add a new vertex z , and connect it to every vertex in the original G with and edge with cost 0
- Find h(u) for every u

  with Bellman-Ford x1 from z

  h(u) := length of the shortest path from z to u
- Calculate c′(u,v) for every edge

  c′(u,v) := c(u,v) + h(u) - h(v)



All-to-all Shortest Paths
Johnson

Problem is the negative edges ! (we can’t use dijkstra)

Idea :

- Make all edge weights ≥ 0
- n x Dijkstra

DOES NOT WORK WITH NEGATIVE CYCLES !

Shortest Paths

Overview

one-to-all

G (directed/undirected)	Algorithm	Runtime
unweighted , all edges with the same positive weight	BFS usage	O (|V| + |E|)
weighted , nonnegative edge weights c(e) ≥ 0	Dijsktra	O ((|V| + |E|) * log n)
weighted, positive and (possibly) negative edge weights c(e) ∊ ℝ	Belmann-Ford*	O (|V| * |E|)
G has no cycles	topological sorting + DP	O (|V| + |E|)

all-to-all

G (directed/undirected)	Algorithm	Runtime
unweighted , all edges with the same positive weight	n x BFS	O (|V| * (|V| + |E|))
weighted , nonnegative edge weights c(e) ≥ 0	n x Dijsktra	O (|V| * (|V| + |E|) * log(|V|))
weighted, positive and (possibly) negative edge weights c(e) ∊ ℝ	n x Belmann-Ford	O (|V| * |V| * |E|)
	Floyd - Warshall*	O($\|V\|^3$)
weighted, positive and (possibly) negative edge weights c(e) ∊ ℝ , no negative cycles	Johnson	O (|V| * (|V| + |E|) * log n)

*negative closed walk detection

Shortest Paths

Exam Tipps

- Know every detail of the "overview"
- Graph Modelling
  - Model the problem correctly, define G,V,E,w
  - Know which algorithm to apply for that particular graph problem
    - BFS,DFS
    - Shortest Paths
    - MST

- Practice, practice, practice !!!

Recap

DP

Recap

Theory Task T3. / 9 P

Let $A = [a_1, a_2, \ldots, a_n]$ be an array of non-negative integers. Given A and a non-negative integers $S \geq 0$, we want to determine whether S can be written as a (non-repeating) sum of elements of A, where we are allowed to take the square of elements. Formally, we want to determine if there exist $I, J \subseteq \{1, 2, \ldots, n\}$ with $I \cap J = \emptyset$, $I \cup J = \{1, 2, \ldots, n\}$ such that:

$$S = \sum_{i \in I} a_i + \sum_{j \in J} a_j^2.$$

Provide a *dynamic programming* algorithm that outputs `True` if this is possible, and `False` otherwise. For example,

- The inputs $A = [2, 4, 4]$, $S = 34$ should result in `True`, since $34 = 2 + 4^2 + 4^2$.
- The inputs $A = [2, 4, 4]$, $S = 35$ should result in `False`.
- The inputs $A = [2, 4, 3, 22]$, $S = 21$ should result in `False`.

In order to obtain full points, your algorithm should run in time $O(n^2 \cdot S)$. Address the following aspects of your solution:

1) *Definition of the DP table*: What are the dimensions
of each entry?
2) *Computation of an entry*: How can an entry be co
Specify the base cases, i.e., the entries that do not
3) *Calculation order*: In which order can entries be c
entry have been determined in previous steps?
4) *Extracting the solution*: How can the final solution b
5) *Running time*: What is the running time of your alg
of n and S, and justify your answer.

Theory Task T3. / 9 P

You are given an array of n natural numbers $a_1, \ldots, a_n \in \mathbb{N}$, and two natural numbers $A, B \in \mathbb{N}$. You want to determine whether there is a subset $I \subseteq \{1, \ldots, n\}$ satisfying

$$\sum_{i \in I} a_i = A \quad \text{and} \quad \sum_{i \in I} a_i^2 = B.$$

For example,

- The answer for the input $(a_i)_{i \leq n} = [2, 4, 8, 1, 4, 5, 3]$, $A = 8$ and $B = 30$ is *yes* because the set of indices $I = \{1, 4, 6\}$, which corresponds to $(a_i)_{i \leq I} = [2, 1, 5]$, yields the *sum* $2 + 1 + 5 = 8$ and the *sum-of-squares* $2^2 + 1^2 + 5^2 = 30$.
- The answer for the input $(a_i)_{i \leq n} = [2, 4, 8, 1]$, $A = 6$ and $B = 15$ is *no.*

Provide a *dynamic programming* algorithm that determines whether such a subset I exists. In order to get full points, your algorithm should have an $O(n \cdot A \cdot B)$ runtime. Address the following aspects in your solution:

1) *Definition of the DP table*: What are the dimensions of the table $DP[\ldots]$? What is the meaning of each entry ?

be com
lo not d
es be co
s ?
lution be
our algor

heory Task T3. / 9 P

ou are given an array of n natural numbers $a_1, \ldots, a_n \in \mathbb{N}$ summing to $A := \sum_{i=1}^n a_i$, which is a
ultiple of 3. You want to determine whether it is possible to partition $\{1, \ldots, n\}$ into three disjoint
ıbsets I, J, K such that the corresponding elements of the array yield the same sum, i.e.

$$\sum_{i \in I} a_i = \sum_{j \in J} a_j = \sum_{k \in K} a_k = \frac{A}{3}.$$

ote that I, J, K form a partition of $\{1, \ldots, n\}$ if and only if $I \cap J = I \cap K = J \cap K = \emptyset$ and
$\cup J \cup K = \{1, \ldots, n\}$.

or example, the answer for the input $[2, 4, 8, 1, 4, 5, 3]$ is *yes*, because there is the partition $\{3, 4\}$,
$2, 6\}$, $\{1, 5, 7\}$ (corresponding to the subarrays $[8, 1]$, $[4, 5]$, $[2, 4, 3]$, which are all summing to 9).
n the other hand, the answer for the input $[3, 2, 5, 2]$ is *no*.

rovide a *dynamic programming* algorithm that determines whether such a partition exists. Your
lgorithm should have an $\mathcal{O}(nA^2)$ runtime to get full points. Address the following aspects in your
olution:

1) *Definition of the DP table*: What are the dimensions of the table $DP[\ldots]$? What is the meaning of each entry?

n an entry be computed from the values of other entries?
ries that do not depend on others.
can entries be computed so that values needed for each
vious steps?
e final solution be extracted once the table has been filled?
time of your algorithm? Provide it in Θ-notation in terms
r.

Theory Task T3. / 8 P

An array of non-negative integers $A = [a_1, \ldots, a_n]$ is called *summy* if and only if, for all $i \in \{2, \ldots, n\}$, there exists a (possibly empty) set $I \subseteq \{1, \ldots, i-1\}$ such that $a_i = \sum_{j \in I} a_j$. In other terms, every integer in the array except the first one must be the sum of (distinct) integers that precede it in the array.

For example,

- The array $[2, 2, 4, 6, 0, 12]$ is summy, because $2 = 2$, $4 = 2 + 2$, $6 = 2 + 4$, $12 = 2 + 4 + 6$.
- The array $[2, 2, 4, 6, 0, 13]$ is not summy, since 13 can not be written as a sum of integers from $\{2, 2, 4, 6, 0\}$.

Provide a *dynamic programming* algorithm that, given an array A of length n, returns `True` if the array is summy, and `False` otherwise. In order to obtain full points, your algorithm should have an $O(n \cdot \max A)$ runtime (where $\max A$ means the maximum value of entries in A). Address the following aspects in your solution:

1) *Definition of the DP table*: What are the dimensions of the table $DP[\ldots]$? What is the meaning of each entry ?
2) *Computation of an entry*: How can an entry be computed from the values of other entries ? Specify the base cases, i.e., the entries that do not depend on others.
3) *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps ?
4) *Extracting the solution*: How can the final solution be extracted once the table has been filled ?
5) *Running time*: What is the running time of your algorithm ? Provide it in Θ-notation in terms of n and $\max A$, and justify your answer.

Theory Task T3. / 9 P

Let m, r be two integers satisfying $m \geq 2$ and $0 \leq r < m$. We say that a finite set $A \subset \mathbb{N}$ of natural numbers is (m, r)-aligned if

$$\left(\sum_{x \in A} x\right) \bmod m = r\,.$$

Note that for $A = \varnothing$, we adopt the convention that $\sum_{x \in A} x = 0$. Hence, the empty set is $(m, 0)$-aligned for every $m > 0$.

Given three integers m, r, n such that $0 \leq r < m < n$ and $m \geq 2$, we would like to determine the number of subsets of $\{1, 2, \ldots, n\}$ which are (m, r)-aligned.

For example,

- If $r = 1$, $m = 2$ and $n = 3$, the subsets of $\{1, 2, 3\}$ that are $(3, 1)$-aligned are $\{1\}$, $\{3\}$, $\{1, 2\}$ and $\{2, 3\}$. Hence, the answer is 4.

Provide a *dynamic programming* algorithm that solves the problem. In order to get full points, your algorithm should have an $O(n \cdot m)$ runtime. Address the following aspects in your solution:

1) *Definition of the DP table*: What are the dimensions of the table $DP[\ldots]$? What is the meaning of each entry ?
2) *Computation of an entry*: How can an entry be computed from the values of other entries ? Specify the base cases, i.e., the entries that do not depend on others.
3) *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps ?
4) *Extracting the solution*: How can the final solution be extracted once the table has been filled ?
5) *Running time*: What is the running time of your algorithm ? Provide it in Θ-notation in terms of n, m and r, and justify your answer.

DP
Recap

Theory Task T3. / 9 P

Let $A = [a_1, a_2, \dots, a_n]$ be an array of non-negative integers. Given A and a non-negative integers $S \geq 0$, we want to determine whether S can be written as a (non-repeating) sum of elements of A, where we are allowed to take the square of elements. Formally, we want to determine if there exist $I, J \subseteq \{1, 2, \dots, n\}$ with $I \cap J = \emptyset$, $I \cup J = \{1, 2, \dots, n\}$ such that:

$$\boxed{S = \sum_{i \in I} a_i + \sum_{j \in J} a_j^2.}$$

Provide a *dynamic programming* algorithm that outputs `True` if this is possible, and `False` otherwise. For example,

- The inputs $A = [2, 4, 4]$, $S = 34$ should result in `True`, since $34 = 2 + 4^2 + 4^2$.
- The inputs $A = [2, 4, 4]$, $S = 35$ should result in `False`.
- The inputs $A = [2, 4, 3, 22]$, $S = 21$ should result in `False`.

In order to obtain full points, your algorithm should run in time $O(n^2 \cdot S)$. Address the following aspects of your solution:

1) *Definition of the DP table*: What are the dimensions
of each entry?
2) *Computation of an entry*: How can an entry be co
Specify the base cases, i.e., the entries that do not
3) *Calculation order*: In which order can entries be c
entry have been determined in previous steps?
4) *Extracting the solution*: How can the final solution b
5) *Running time*: What is the running time of your alg
of n and S, and justify your answer.

Theory Task T3. / 9 P

You are given an array of n natural numbers $a_1, \dots, a_n \in \mathbb{N}$, and two natural numbers $A, B \in \mathbb{N}$. You want to determine whether there is a subset $I \subseteq \{1, \dots, n\}$ satisfying

$$\boxed{\sum_{i \in I} a_i = A \quad \text{and} \quad \sum_{i \in I} a_i^2 = B.}$$

For example,

- The answer for the input $(a_i)_{i \leq n} = [2, 4, 8, 1, 4, 5, 3]$, $A = 8$ and $B = 30$ is *yes* because the set of indices $I = \{1, 4, 6\}$, which corresponds to $(a_i)_{i \leq I} = [2, 1, 5]$, yields the *sum* $2 + 1 + 5 = 8$ and the *sum-of-squares* $2^2 + 1^2 + 5^2 = 30$.
- The answer for the input $(a_i)_{i \leq n} = [2, 4, 8, 1]$, $A = 6$ and $B = 15$ is *no*.

Provide a *dynamic programming* algorithm that determines whether such a subset I exists. In order to get full points, your algorithm should have an $O(n \cdot A \cdot B)$ runtime. Address the following aspects in your solution:

1) *Definition of the DP table*: What are the dimensions of the table $DP[\dots]$? What is the meaning of each entry ?

be com
lo not d
es be co
s ?
lution be
our algo

Theory Task T3. / 9 P

ou are given an array of n natural numbers $a_1, \dots, a_n \in \mathbb{N}$ summing to $A := \sum_{i=1}^{n} a_i$, which is a nultiple of 3. You want to determine whether it is possible to partition $\{1, \dots, n\}$ into three disjoint ubsets I, J, K such that the corresponding elements of the array yield the same sum, i.e.

$$\boxed{\sum_{i \in I} a_i = \sum_{j \in J} a_j = \sum_{k \in K} a_k = \frac{A}{3}.}$$

ote that I, J, K form a partition of $\{1, \dots, n\}$ if and only if $I \cap J = I \cap K = J \cap K = \emptyset$ and $\cup J \cup K = \{1, \dots, n\}$.

or example, the answer for the input $[2, 4, 8, 1, 4, 5, 3]$ is *yes*, because there is the partition $\{3, 4\}$, $2, 6\}$, $\{1, 5, 7\}$ (corresponding to the subarrays $[8, 1]$, $[4, 5]$, $[2, 4, 3]$, which are all summing to 9). n the other hand, the answer for the input $[3, 2, 5, 2]$ is *no*.

rovide a *dynamic programming* algorithm that determines whether such a partition exists. Your lgorithm should have an $\mathcal{O}(nA^2)$ runtime to get full points. Address the following aspects in your olution:

1) *Definition of the DP table*: What are the dimensions of the table $DP[\dots]$? What is the meaning of each entry?

n an entry be computed from the values of other entries?
ries that do not depend on others.
can entries be computed so that values needed for each
vious steps?
e final solution be extracted once the table has been filled?
time of your algorithm? Provide it in Θ-notation in terms
r.

Theory Task T3. / 8 P

An array of non-negative integers $A = [a_1, \dots, a_n]$ is called *summy* if and $\boxed{\text{only if, for}}$ all $i \in \{2, \dots, n\}$, there exists a (possibly empty) set $I \subseteq \{1, \dots, i-1\}$ such that $a_i \boxed{= \sum_{j \in I} a_j}$. In other terms, every integer in the array except the first one must be the sum of $\boxed{\text{(distinct) integers}}$ that precede it in the array.

For example,

- The array $[2, 2, 4, 6, 0, 12]$ is summy, because $2 = 2$, $4 = 2 + 2$, $6 = 2 + 4$, $12 = 2 + 4 + 6$.
- The array $[2, 2, 4, 6, 0, 13]$ is not summy, since 13 can not be written as a sum of integers from $\{2, 2, 4, 6, 0\}$.

Provide a *dynamic programming* algorithm that, given an array A of length n, returns `True` if the array is summy, and `False` otherwise. In order to obtain full points, your algorithm should have an $O(n \cdot \max A)$ runtime (where $\max A$ means the maximum value of entries in A). Address the following aspects in your solution:

1) *Definition of the DP table*: What are the dimensions of the table $DP[\dots]$? What is the meaning of each entry ?
2) *Computation of an entry*: How can an entry be computed from the values of other entries ? Specify the base cases, i.e., the entries that do not depend on others.
3) *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps ?
4) *Extracting the solution*: How can the final solution be extracted once the table has been filled ?
5) *Running time*: What is the running time of your algorithm ? Provide it in Θ-notation in terms of n and $\max A$, and justify your answer.

Theory Task T3. / 9 P

Let m, r be two integers satisfying $m \geq 2$ and $0 \leq r < m$. We say that a finite set $A \subset \mathbb{N}$ of natural numbers is (m, r)-aligned if

$$\boxed{\left(\sum_{x \in A} x\right) \bmod m = r\,.}$$

Note that for $A = \emptyset$, we adopt the convention that $\sum_{x \in A} x = 0$. Hence, the empty set is $(m, 0)$-aligned for every $m > 0$.

Given three integers m, r, n such that $0 \leq r < m < n$ and $m \geq 2$, we would like to determine the number of subsets of $\{1, 2, \dots, n\}$ which are (m, r)-aligned.

For example,

- If $r = 1$, $m = 2$ and $n = 3$, the subsets of $\{1, 2, 3\}$ that are $(3, 1)$-aligned are $\{1\}$, $\{3\}$, $\{1, 2\}$ and $\{2, 3\}$. Hence, the answer is 4.

Provide a *dynamic programming* algorithm that solves the problem. In order to get full points, your algorithm should have an $O(n \cdot m)$ runtime. Address the following aspects in your solution:

1) *Definition of the DP table*: What are the dimensions of the table $DP[\dots]$? What is the meaning of each entry ?
2) *Computation of an entry*: How can an entry be computed from the values of other entries ? Specify the base cases, i.e., the entries that do not depend on others.
3) *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps ?
4) *Extracting the solution*: How can the final solution be extracted once the table has been filled ?
5) *Running time*: What is the running time of your algorithm ? Provide it in Θ-notation in terms of n, m and r, and justify your answer.

DP

Σ

$$\sum_{i \in I} a_i = A$$

a_1	a_2	a_3	a_4	a_5	a_6	a_7
1	2	0	1	0	0	0

DP

Σ

elements a_i according to indexes i

$I \subseteq \{1, \ldots, n\}$

$$\sum_{i \in I} a_i = A$$

a_1	a_2	a_3	a_4	a_5	a_6	a_7
1	2	0	1	0	0	0

sum of the elements a_i

indexes i from Index-set I

DP

Σ

elements a_i according to indexes i

$I \subseteq \{1, \ldots, n\}$

$$\sum_{\underline{i}\in\underline{I}} \underline{a_i} = \underline{A}$$

sum of the elements a_i

indexes i from Index-set I

example:

$$\sum_{\underline{i}\in\underline{I}} \underline{a_i} = \underline{A}$$

$A = 3$

$I = \{1,2\}$

a_1	a_2	a_3	a_4	a_5	a_6	a_7
1	2	0	1	0	0	0

DP

Σ

elements a_i according to indexes i

$I \subseteq \{1, \ldots, n\}$

$$\sum_{i \in I} a_i = A$$

sum of the elements a_i

indexes i from Index-set I

example:

$$\sum_{i \in I} a_i = A$$

A = 3

I = {1,2}

a_1	a_2	a_3	a_4	a_5	a_6	a_7
1	2	0	1	0	0	0

$a_1 + a_2 = 1 + 2 = 3 = A$

DP

Σ

elements a_i according to indexes i

$$I \subseteq \{1, \ldots, n\} \qquad \sum_{i \in I} a_i = A$$

sum of the elements a_i

indexes i from Index-set I

$$S = \sum_{i \in I} a_i + \sum_{j \in J} a_j^2.$$

$$\sum_{i \in I} a_i = A \quad \text{and} \quad \sum_{i \in I} a_i^2 = B.$$

$$\sum_{i \in I} a_i = \sum_{j \in J} a_j = \sum_{k \in K} a_k = \frac{A}{3}.$$

$$\left(\sum_{x \in A} x\right) \bmod m = r\,.$$

DP

Practice

Theory Task T3. / 9 P

Let $A = [a_1, a_2, \dots, a_n]$ be an array of non-negative integers. Given A and a non-negative integers $S \geq 0$, we want to determine whether S can be written as a (non-repeating) sum of elements of A, where we are allowed to take the square of elements. Formally, we want to determine if there exist $I, J \subseteq \{1, 2, \dots, n\}$ with $I \cap J = \emptyset$, $I \cup J = \{1, 2, \dots, n\}$ such that:

$$S = \sum_{i \in I} a_i + \sum_{j \in J} a_j^2.$$

Provide a *dynamic programming* algorithm that outputs `True` if this is possible, and `False` otherwise. For example,

- The inputs $A = [2, 4, 4]$, $S = 34$ should result in `True`, since $34 = 2 + 4^2 + 4^2$.
- The inputs $A = [2, 4, 4]$, $S = 35$ should result in `False`.
- The inputs $A = [2, 4, 3, 22]$, $S = 21$ should result in `False`.

In order to obtain full points, your algorithm should run in time $O(n^2 \cdot S)$. Address the following aspects of your solution:

1) *Definition of the DP table*: What are the dimensions
of each entry?
2) *Computation of an entry*: How can an entry be co
Specify the base cases, i.e., the entries that do not
3) *Calculation order*: In which order can entries be c
entry have been determined in previous steps?
4) *Extracting the solution*: How can the final solution b
5) *Running time*: What is the running time of your alg
of n and S, and justify your answer.

Theory Task T3. / 9 P

You are given an array of n natural numbers $a_1, \dots, a_n \in \mathbb{N}$, and two natural numbers $A, B \in \mathbb{N}$. You want to determine whether there is a subset $I \subseteq \{1, \dots, n\}$ satisfying

$$\sum_{i \in I} a_i = A \quad \text{and} \quad \sum_{i \in I} a_i^2 = B.$$

For example,

- The answer for the input $(a_i)_{i \leq n} = [2, 4, 8, 1, 4, 5, 3]$, $A = 8$ and $B = 30$ is *yes* because the set of indices $I = \{1, 4, 6\}$, which corresponds to $(a_i)_{i \leq I} = [2, 1, 5]$, yields the *sum* $2 + 1 + 5 = 8$ and the *sum-of-squares* $2^2 + 1^2 + 5^2 = 30$.
- The answer for the input $(a_i)_{i \leq n} = [2, 4, 8, 1]$, $A = 6$ and $B = 15$ is *no*.

Provide a *dynamic programming* algorithm that determines whether such a subset I exists. In order to get full points, your algorithm should have an $O(n \cdot A \cdot B)$ runtime. Address the following aspects in your solution:

1) *Definition of the DP table*: What are the dimensions of the table $DP[\dots]$? What is the meaning of each entry ?

be com
lo not de
es be co
s ?
lution be
our algori

Theory Task T3. / 9 P

ou are given an array of n natural numbers $a_1, \dots, a_n \in \mathbb{N}$ summing to $A := \sum_{i=1}^{n} a_i$, which is a
nultiple of 3. You want to determine whether it is possible to partition $\{1, \dots, n\}$ into three disjoint
ibsets I, J, K such that the corresponding elements of the array yield the same sum, i.e.

$$\sum_{i \in I} a_i = \sum_{j \in J} a_j = \sum_{k \in K} a_k = \frac{A}{3}.$$

Iote that I, J, K form a partition of $\{1, \dots, n\}$ if and only if $I \cap J = I \cap K = J \cap K = \emptyset$ and
$\cup J \cup K = \{1, \dots, n\}$.

or example, the answer for the input $[2, 4, 8, 1, 4, 5, 3]$ is *yes*, because there is the partition $\{3, 4\}$,
$2, 6\}$, $\{1, 5, 7\}$ (corresponding to the subarrays $[8, 1]$, $[4, 5]$, $[2, 4, 3]$, which are all summing to 9).
n the other hand, the answer for the input $[3, 2, 5, 2]$ is *no*.

rovide a *dynamic programming* algorithm that determines whether such a partition exists. Your
lgorithm should have an $\mathcal{O}(nA^2)$ runtime to get full points. Address the following aspects in your
olution:

1) *Definition of the DP table*: What are the dimensions of the table $DP[\dots]$? What is the
meaning of each entry?

n an entry be computed from the values of other entries?
ries that do not depend on others.
can entries be computed so that values needed for each
vious steps?
e final solution be extracted once the table has been filled?
time of your algorithm? Provide it in Θ-notation in terms
r.

Theory Task T3. / 8 P

An array of non-negative integers $A = [a_1, \dots, a_n]$ is called *summy* if and only if, for all $i \in \{2, \dots, n\}$, there exists a (possibly empty) set $I \subseteq \{1, \dots, i-1\}$ such that $a_i = \sum_{j \in I} a_j$. In other terms, every integer in the array except the first one must be the sum of (distinct) integers that precede it in the array.

For example,

- The array $[2, 2, 4, 6, 0, 12]$ is summy, because $2 = 2$, $4 = 2 + 2$, $6 = 2 + 4$, $12 = 2 + 4 + 6$.
- The array $[2, 2, 4, 6, 0, 13]$ is not summy, since 13 can not be written as a sum of integers from $\{2, 2, 4, 6, 0\}$.

Provide a *dynamic programming* algorithm that, given an array A of length n, returns `True` if the array is summy, and `False` otherwise. In order to obtain full points, your algorithm should have an $O(n \cdot \max A)$ runtime (where $\max A$ means the maximum value of entries in A). Address the following aspects in your solution:

1) *Definition of the DP table*: What are the dimensions of the table $DP[\dots]$? What is the meaning of each entry ?
2) *Computation of an entry*: How can an entry be computed from the values of other entries ? Specify the base cases, i.e., the entries that do not depend on others.
3) *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps ?
4) *Extracting the solution*: How can the final solution be extracted once the table has been filled ?
5) *Running time*: What is the running time of your algorithm ? Provide it in Θ-notation in terms of n and $\max A$, and justify your answer.

Theory Task T3. / 9 P

Let m, r be two integers satisfying $m \geq 2$ and $0 \leq r < m$. We say that a finite set $A \subset \mathbb{N}$ of natural numbers is (m, r)-aligned if

$$\left(\sum_{x \in A} x \right) \bmod m = r .$$

Note that for $A = \varnothing$, we adopt the convention that $\sum_{x \in A} x = 0$. Hence, the empty set is $(m, 0)$-aligned for every $m > 0$.

Given three integers m, r, n such that $0 \leq r < m < n$ and $m \geq 2$, we would like to determine the number of subsets of $\{1, 2, \dots, n\}$ which are (m, r)-aligned.

For example,

- If $r = 1$, $m = 2$ and $n = 3$, the subsets of $\{1, 2, 3\}$ that are $(3, 1)$-aligned are $\{1\}$, $\{3\}$, $\{1, 2\}$ and $\{2, 3\}$. Hence, the answer is 4.

Provide a *dynamic programming* algorithm that solves the problem. In order to get full points, your algorithm should have an $O(n \cdot m)$ runtime. Address the following aspects in your solution:

1) *Definition of the DP table*: What are the dimensions of the table $DP[\dots]$? What is the meaning of each entry ?
2) *Computation of an entry*: How can an entry be computed from the values of other entries ? Specify the base cases, i.e., the entries that do not depend on others.
3) *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps ?
4) *Extracting the solution*: How can the final solution be extracted once the table has been filled ?
5) *Running time*: What is the running time of your algorithm ? Provide it in Θ-notation in terms of n, m and r, and justify your answer.

Last Session
Organization

- Last session on monday 16 Dec
  - Exam Preparation Session
    - Exam tipps, lernphase tipps, mock exam
  - Recap topics
    - The rest will be covered during mock exam
  - Semester-end celebration !!!

Questions

Feedbacks , Recommendations

Nil Ozer