Problem Statement

National Space Hackathon 2025

Cargo Stowage Management System

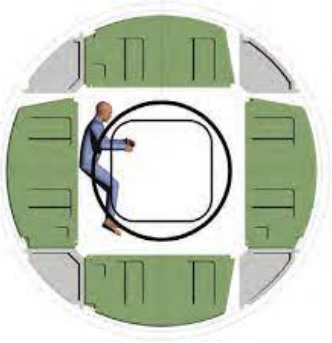
Background

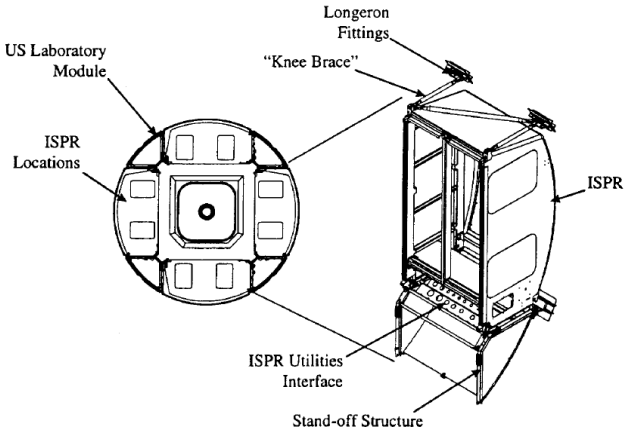
Managing cargo inside a space station like the **International Space Station (ISS)** is an exceptionally complex task. Research states that these stowage operations (the act of storing and retrieving items) on the ISS takes up about 25% of each astronaut's time. Your task is to create a solution to address this problem, by automating this workflow and packaging it into an application.

Every few months, a **resupply mission** arrives, bringing a variety of cargo such as:

- **Food, water, and medicine (consumables)**.
- **Tools, equipment, spare parts, and mission-specific hardware (equipment)**.
- **Scientific instruments or experimental samples (payload)**.

Some examples of storage containers on the ISS:





A poorly managed cargo space can lead to **wasted time, difficulty accessing critical items, and severe inefficiencies** during emergencies or high-priority missions. Astronauts should be able to rely on your **cargo management and optimization software**

Core Objective of Your Software

Your software will act as a **stowage advisor** to the astronauts, ensuring:

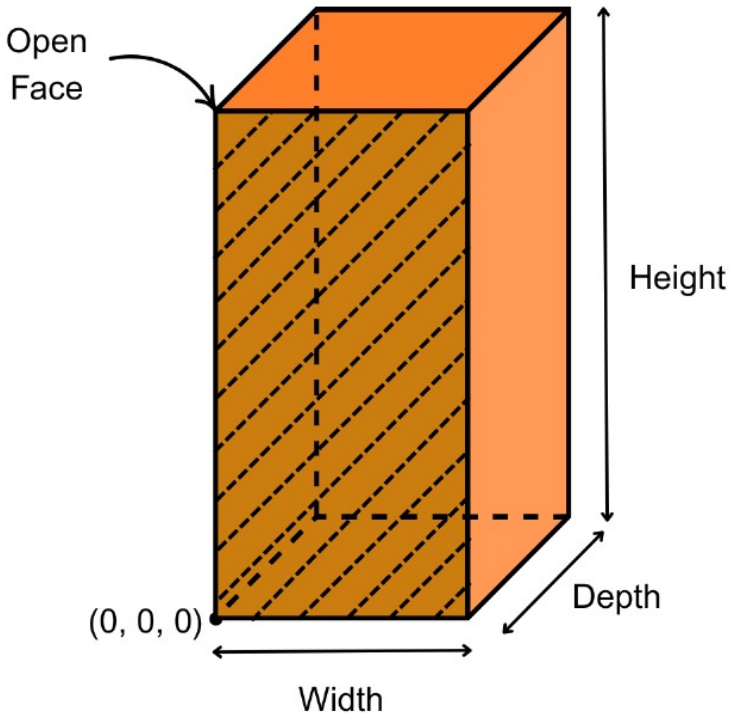
1. **Efficient Placement of Items** – It should suggest where to place new incoming cargo based on **space availability, priority, and accessibility** while handling the space constraint.
2. **Quick Retrieval of Items** – When an astronaut searches for an item, it should suggest the item that can be retrieved the fastest. Details about how data related to retrieval is calculated are given below. You can also add additional logic such as retrieving items that are about to expire, etc.

3. **Rearrangement Optimization** – If a new incoming shipment causes space shortage, it should suggest rearranging certain items to optimize space usage.
4. **Waste Disposal Management** – When items become unusable (expired or fully used), the system should:
    - Automatically categorize them as waste.
    - Suggest which container to move the waste into for undocking.
5. **Cargo Return Planning** – Before a resupply module undocks, the system should provide a clear plan for waste return and space reclamation.
6. **Logging** all the actions performed by the astronauts
7. Make sure the algorithms you use are efficient, **as power and compute is a valuable resource in space**, your scores also depend on this and above mentioned factors. (Rubrics given below)

Important Definitions

1. Storage Containers (Assumed to be cuboids)

- Every container has one of its faces designated as the "open face" (typically one of the larger faces)
- The open face is the only side through which items can be inserted or removed
- The origin of the container is taken as the bottom left corner of this open face
- Coordinates within the container are measured as (Width, Depth, Height) where:
    - Width runs horizontally along the open face
    - Depth runs perpendicular to the open face, into the container
    - Height runs vertically along the open face



2. Retrieval

- An item can only be moved **directly perpendicular** to the open face of the container (i.e., straight out from the opening). Only **visible** items (those directly accessible from the open face) can be moved. The retrieval process requires that no other item obstructs the path of the desired item as it exits the container.
- The **steps needed for retrieval** refer to the number of items that must be temporarily removed (and then placed back) for the desired item to be retrieved. If an item is already visible, its retrieval requires **0 steps**.

Input Format (only for explanation, API format below)

This is an example of some items which can be given in input. You can generate more items as per your need. We will be using a large list of items (thousands) to check your algorithm's efficiency.

Item ID	Name	Width (cm)	Depth (cm)	Height (cm)	Mass (kg)	Priority (1-100)	Expiry Date (ISO Format)	Usage Limit	Preferred Zone
001	Food Packet	10	10	20	5	80	2025-05-20	30 uses	Crew Quarters
002	Oxygen Cylinder	15	15	50	30	95	N/A	100 uses	Airlock
003	First Aid Kit	20	20	10	2	100	2025-07-10	5 uses	Medical Bay
…	…	…	…	…	…	…	…	…	…

Column Details:

- **Item ID:** Unique identifier.
- **Name:** Name of the item.
- **Width, Depth and Height** (You can rotate the items also)
- **Mass:** Mass of the item
- **Priority (0-100):**
    - Higher scores mean higher criticality. They should be retrievable with lesser steps
    - Higher priority items should also be given higher priority for its preferred zones.
- **Expiry Date:** After this date this item is considered as waste.
- **Usage Limit:** Number of times the item can be used before becoming waste. **One use is consumed when an astronaut retrieves the item.**
- **Preferred Zone:** Suggested area for optimal placement.

If an item cannot be placed in its preferred zone, it can be placed in some other zone. However, this will cause points deduction from the priority efficiency category. (Exact policy will be shared soon)

This is an example of some containers which can be given in input. You can generate more as per your need. We will be using a large list of containers (hundreds) to check your algorithm's efficiency.

Zone	Container ID	Width(cm)	Depth(cm)	Height(height)
Crew Quarters	contA	100	85	200
Airlock	contB	50	85	200
Laboratory	contC	200	85	200
…	…	…	…	…

Expected Functionality

Your software must provide the following features:

1. Placement Recommendations (Priority Efficiency)

When new stock arrives:

- Automatically suggest **where to place each item** based on available space and priority.
- If space is insufficient, recommend:
    - **Rearranging existing items** to make space.
    - **Moving lower-priority items** to less accessible zones.
- Ensure **high-priority items** remain easily accessible and in preferred zones.

2. Item Search + Retrieval Optimization

When an astronaut searches for an item:

- Suggest the exact **module and position**.
- Choose the item on the basis of ease of retrieval and closeness to expiry date etc.
- Provide **instructions for retrieval**, minimizing movement of other items.
- Also the astronaut can place it in some other container and enter this in the software
- Log the action in the database (who retrieved it, when, and from where).

3. Rearrangement Recommendations (Space Management)

If a new stock file is uploaded and space is insufficient:

- Automatically suggest which low-priority items can be **relocated**.
- Minimize time spent moving items.
- Show a **step-by-step movement plan** if rearrangement is necessary.

4. Waste Management + Return Planning

- Track items that become waste (expired or finished).
- Automatically mark them for disposal.
- When undocking is triggered:
    - Suggest moving all waste to the undocking module.
    - Make sure a weight limit is followed for the undocking
    - Generate a manifest for cargo return.
- Free up space after undocking.

5. Time Simulation (Fast Forward)

Your UI must have:

- **Next Day Button** – Simulate one day. (Take as input: items to be used in a day)
- **Fast Forward X Days Button** – Simulate multiple days. (Same)
- Items after their expiry date or when there are no uses left are considered as waste.
- This will also help with mission planning

6. Docker-Based Deployment

Note: This is a hard requirement! If the repo provided does not build using Docker and does not use the specified source image, then your submission cannot be auto-tested and will not be considered!

Requirements:

1. There should be a Dockerfile at the root of your GitHub repo that initialises the simulation and exposes the required APIs.
2. The port 8000 should be exported so that the API can be tested.
3. ***The dockerfile should use the `ubuntu:22.04` image! This is to prevent incompatibilities when testing across different environments.***
4. **Here is an example,** valid Dockerfile submission that exposes a Python webserver on port 8000:

```
# Start from Ubuntu:22.04 as the base image
FROM ubuntu:22.04

# Set environment variables to avoid interactive prompts during installation
ENV DEBIAN_FRONTEND=noninteractive

# Update package lists and install Python and pip
RUN apt-get update && \
    apt-get install -y python3 python3-pip && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*

# Set working directory inside the container
WORKDIR /app

# Copy requirements file
COPY requirements.txt .

# Install Python dependencies
RUN pip3 install --no-cache-dir -r requirements.txt

# Copy the rest of the application code
```

```
COPY . .

# Expose port 8000 to the outside world
EXPOSE 8000

# Command to run the Python application
CMD ["python", "main.py"]
```

5. Another valid Dockerfile, this time running a Node.js app:

```
# Start from Ubuntu as the base image
FROM ubuntu:22.04

# Update package lists and install Node.js
RUN apt-get update && \
    apt-get install -y ca-certificates curl gnupg && \
    mkdir -p /etc/apt/keyrings && \
    curl -fsSL https://deb.nodesource.com/gpgkey/nodesource-repo.gpg.key | gpg
--dearmor -o /etc/apt/keyrings/nodesource.gpg && \
    echo "deb [signed-by=/etc/apt/keyrings/nodesource.gpg]
https://deb.nodesource.com/node_$NODE_VERSION nodistro main" | tee
/etc/apt/sources.list.d/nodesource.list && \
    apt-get update && \
    apt-get install -y nodejs && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*

# Set working directory inside the container
WORKDIR /app

# Copy package.json and package-lock.json first
COPY package*.json ./

# Install Node.js dependencies
RUN npm install

# Copy the rest of the application code
COPY . .

# Expose port 8000 to the outside world
EXPOSE 8000

# Command to run the Node.js application
# Assuming your main file is called index.js
CMD ["node", "index.js"]
```

6. Test out your Dockerfile by building and running a Docker image from the root of your repo (use the `--network=host` flag), and trying to hit your API at port 8000:

```
# Build the Docker image
docker build -t my-app .
# Run the container
docker run -p 8000:8000 --network=host my-app
```

7. You can probably find template Dockerfiles for your language online! For more information regarding Dockerfiles, refer to the following websites and material: Docker Documentation; Dockerfile Reference

Common troubleshooting tips:

- Ensure your application is configured to listen on 0.0.0.0 (all interfaces) rather than localhost or 127.0.0.1. For example, when running a Python FastAPI server, you should probably use `--host 0.0.0.0` and not `--host 127.0.0.1.`
- Check that your application is actually listening on port 8000
- Verify that you've used the `--network=host` flag while running
- Verify that all required dependencies are included in your Dockerfile
- If your application requires environment variables, make sure they're set in the Dockerfile or passed at runtime
- Use Docker logs to troubleshoot issues with your running container: `docker logs -f <container_id>`

---

7. API Design

Your program should have these API routes at port 8000 after running the docker image:

1. Placement Recommendations API

Endpoint: `/api/placement`

Method: POST

Request Body:

```
{
    "items": [
        {
            "itemId": "string",
            "name": "string",
            "width": number,
            "depth": number,
            "height": number,
            "priority": number,
            "expiryDate": "string", // ISO format
            "usageLimit": number,
            "preferredZone": "string" // Zone
        }
    ],
    "containers": [
        {
            "containerId": "string",
            "zone": "string",
            "width": number,
            "depth": number,
            "height": number
        }
    ]
}
```

Response:

```
{
    "success": boolean,
    "placements": [
        {
            "itemId": "string",
            "containerId": "string",
            "position": {
                "startCoordinates": {
                    "width": number,
                    "depth": number,
                    "height": number
                },
                "endCoordinates": {
                    "width": number,
                    "depth": number,
                    "height": number
                }
            }
        }
    ],
    "rearrangements": [
        {
            "step": number,
            "action": "string", // "move", "remove", "place"
            "itemId": "string",
            "fromContainer": "string",
            "fromPosition": {
                "startCoordinates": {
                    "width": number,
                    "depth": number,
                    "height": number
                },
                "endCoordinates": {
                    "width": number,
                    "depth": number,
                    "height": number
                }
            },
            "toContainer": "string",
            "toPosition": {
```

```
                "startCoordinates": {
                    "width": number,
                    "depth": number,
                    "height": number
                },
                "endCoordinates": {
                    "width": number,
                    "depth": number,
                    "height": number
                }
            }
        }
    ],
}
```

2. Item Search and Retrieval API

Each retrieval of an object means that the object has been used once.

Endpoint: `/api/search`

Method: GET

Query Parameters:

- `itemId`: **string**
- `itemName`: **string (Either itemName or itemId to be given)**
- `userId`: **string (optional)**

Response:

```
{
    "success": boolean,
    "found": boolean,
    "item": {
        "itemId": "string",
        "name": "string",
        "containerId": "string",
        "zone": "string",
        "position": {
            "startCoordinates": {
```

```
                "width": number,
                "depth": number,
                "height": number
            },
            "endCoordinates": {
                "width": number,
                "depth": number,
                "height": number
            }
        }
    },
    "retrievalSteps": [
        {
            "step": number,
            "action": "string", // "remove", "setAside", "retrieve",
"placeBack"
            "itemId": "string",
            "itemName": "string"
        }
    ]
}
```

Endpoint: `/api/retrieve`

Method: POST

Request Body:

```
{
    "itemId": "string",
    "userId": "string",
    "timestamp": "string" // ISO format
}
```

Response:

```
{
    "success": boolean,
}
```

Endpoint: `/api/place`

Method: POST

Request Body:

```
{
    "itemId": "string",
    "userId": "string",
    "timestamp": "string", // ISO format
    "containerId": "string", // Where the item is kept
    "position": {
        "startCoordinates": {
            "width": number,
            "depth": number,
            "height": number
        },
        "endCoordinates": {
            "width": number,
            "depth": number,
            "height": number
        }
    }
}
```

Response:

```
{
    "success": boolean,
}
```

3. Waste Management API

Endpoint: `/api/waste/identify`

Method: GET

Response:

```
{
    "success": boolean,
    "wasteItems": [
        {
            "itemId": "string",
            "name": "string",
            "reason": "string", // "Expired", "Out of Uses"
            "containerId": "string",
            "position": {
                "startCoordinates": {
                    "width": number,
                    "depth": number,
                    "height": number
                },
                "endCoordinates": {
                    "width": number,
                    "depth": number,
                    "height": number
                }
            }
        }
    ]
}
```

Endpoint: **`/api/waste/return-plan`**

Method: POST

Request Body:

```
{
    "undockingContainerId": "string",
    "undockingDate": "string", // ISO format
    "maxWeight": number
}
```

Response:

```
{
```

```
    "success": boolean,
    "returnPlan": [
        {
            "step": number,
            "itemId": "string",
            "itemName": "string",
            "fromContainer": "string",
            "toContainer": "string"
        }
    ],
    "retrievalSteps": [
        {
            "step": number,
            "action": "string", // "remove", "setAside", "retrieve",
"placeBack"
            "itemId": "string",
            "itemName": "string"
        }
    ],
    "returnManifest": {
        "undockingContainerId": "string",
        "undockingDate": "string",
        "returnItems": [
            {
                "itemId": "string",
                "name": "string",
                "reason": "string"
            }
        ],
        "totalVolume": number,
        "totalWeight": number
    }
}
```

Endpoint: **`/api/waste/complete-undocking`**

Method: POST

Request Body:

```
{
```

```
    "undockingContainerId": "string",
    "timestamp": "string" // ISO format
}
```

Response:

```
{
    "success": boolean,
    "itemsRemoved": number
}
```

4. Time Simulation API

Endpoint: **`/api/simulate/day`**

Method: POST

Request Body:

```
{
    "numOfDays": number, // Either this or toTimestamp
    "toTimestamp": "string", // ISO format
    "itemsToBeUsedPerDay": [
        {
            "itemId": "string",
            "name": "string", // Either of these
        }
    ]
}
```

Response:

```
{
    "success": boolean,
    "newDate": "string", // ISO format
    "changes": {
        "itemsUsed": [
            {
```

```
                "itemId": "string",
                "name": "string",
                "remainingUses": number
            }
        ],
        "itemsExpired": [
            {
                "itemId": "string",
                "name": "string"
            }
        ],
        "itemsDepletedToday": [ // Items which are now out of uses
            {
                "itemId": "string",
                "name": "string"
            }
        ]
    }
}
```

5. Import/Export API

Endpoint: **`/api/import/items`**

Method: POST

Request Body: Form data with CSV file upload

Response:

```
{
    "success": boolean,
    "itemsImported": number,
    "errors": [
        {
            "row": number,
            "message": "string"
        }
    ]
}
```

Endpoint: `/api/import/containers`

Method: POST

Request Body: Form data with CSV file upload

Response:

```
{
    "success": boolean,
    "containersImported": number,
    "errors": [
        {
            "row": number,
            "message": "string"
        }
    ]
}
```

Endpoint: `/api/export/arrangement`

Method: GET

Response: CSV file download with the current arrangement in the format:

Item ID,Container ID,Coordinates (W1,D1,H1),(W2,D2,H2)
001,contA,(0,0,0),(10,10,20) 002,contB,(0,0,0),(15,15,50)

6. Logging API

Endpoint: `/api/logs`

Method: GET

Query Parameters:

- **`startDate`: string (ISO format)**
- **`endDate`: string (ISO format)**
- **`itemId`: string (optional)**
- **`userId`: string (optional)**
- **`actionType`: string (optional) - "placement", "retrieval", "rearrangement", "disposal"**

Response:

```
{
    "logs": [
        {
            "timestamp": "string",
            "userId": "string",
            "actionType": "string",
            "itemId": "string",
            "details": {
                "fromContainer": "string",
                "toContainer": "string",
                "reason": "string"
            }
        }
    ]
}
```

---

8. Functional UI (Frontend)

Your program should have a usable frontend with all the above mentioned features implemented. You may make and use extra API endpoints but you need to create and use the above ones for their designated functions.

Example System Initialization and Flow

- At the start, the system would be initialized with empty containers. The frontend would use the /api/import/containers endpoint to upload a CSV file defining all available storage containers onboard the space station (their IDs, zones, and dimensions).
- When a few days later a resupply mission arrives, the system would use the /api/import/items endpoint to upload a CSV with all incoming items and their properties (dimensions, priority, expiry dates, etc.).
- The system would then call the /api/placement endpoint, passing both the container and item data as JSON. The system would calculate optimal placement for all items and return placement recommendations that prioritize accessibility based on item priority.
- As astronauts retrieve items, they'd use the /api/search endpoint to locate specific items, then the /api/retrieve endpoint to log each use. Each retrieval decrements the item's usage count.
- The Astronaut will place the item anywhere and use the /api/place endpoint to update the location of the item in the system. These two can happen hundreds of times in a day
- Over time, the /api/waste/identify endpoint would automatically flag expired or depleted items as waste. When preparing for a return mission, the team would use /api/waste/return-plan to get instructions for moving waste to the undocking module.
- The /api/simulate/day endpoint lets the team simulate time passing, handling expiration dates and usage counts of items used each day.
- All actions are recorded through the logging system and can be reviewed through the /api/logs endpoint.

Evaluation Criteria

Criteria	Weightage	Description
Priority Efficiency	**15%**	Measures how well higher priority of items is regarded
Retrieval Efficiency	**20%**	Evaluates how quickly items can be retrieved with minimal movement of other items.
Rearrangement Efficiency	**15%**	Assesses how effectively the system proposes reorganization with minimal item movements.
Time taken by your code	**10%**	Your code should maintain speed and efficiency while calculating anything.
Waste Management	**10%**	Measures the system's ability to track, identify, and organize waste items for return.
UI/UX Design	**20%**	Evaluates interface clarity, ease of use, and overall user experience.
Logging Accuracy	**10%**	Assesses how well the system tracks and records all item movements and status changes.

Scoring Methodology:

The hackathon will use a two-phase evaluation process:

1. Objective Assessment: All submissions will be tested against standardized test scenarios using predefined test cases by an automatic checker. Submissions will be judged based on correctness and time taken to run by the algorithm.
2. Relative Ranking: The top submissions based on objective metrics will advance to manual evaluation where judges will assess UI/UX and additional features.

Expected Deliverables

- A docker file containing the following:
    - **An application with Backend+frontend+database(optional)**.
- **A brief report** explaining the algorithms used and the code. (Less than 5mb)
- **A video demo** of the frontend and its functions. (Under 2 minutes, less than 10mb)

The Submission Form will be released later.