Backtracking Scheduling:

- Input:
  - A list of tasks, sorted by priorities, durations, time slots, and possibly other attributes
  - A list of satellites, each with available time slots, and possibly other attributes
- Algorithm flow:
  1. Initialization:
     Initialize the best solution as None.
     Initialize the current solution as an empty dictionary.
     Define a backtracking function.
  2. Backtracking Function:
     The function takes an index representing the current task to be assigned.
     - Base Case:
       If all tasks are assigned, update the best solution if the current solution is better.
     - Recursive Case:
       - Iterate through satellites.
       - For each satellite, check if the task can be assigned based on constraints (e.g., time slots).
       - If valid, assign the task to the satellite, and recursively call the function for the next task.
       - Backtrack by undoing the assignment before exploring other possibilities.
  3. Main Algorithm:
     Call the backtracking function with the initial task index.
- Output:
  The best solution, a dictionary mapping satellites to assigned tasks.
- Pros:
  - Can potentially find the optimal solution by exploring all possibilities
  - Can handle various constraints and requirements
  - Easily adaptable to different task assignment scenarios
- Cons:
  - The time complexity can be high, especially for a large number of tasks
  - Resource intensive
- Time complexity:
  The time complexity of the backtracking scheduling algorithm can be expressed as O(N^M), where N is the number of satellites and M is the number of tasks.

Earliest Deadline First (EDF) incorporated with Priority Scheduling:

- Input: a list of tasks with priority, duration, designated execution time frame specified (time frame within which the task should be executed)
- Algorithm flow:

    - Schedule the tasks with highest priority using EDF first, and then do the same for medium priority on the existing schedule, finally for low priority.
    - When a new task A comes in, depends on A's priority, keep the schedule of tasks with higher priority than A, and redo EDF on the group of tasks with the same priority as A, and finally redo EDF on tasks with lower priority than A.
- Output: schedule
- Pros:
    - Satisfy the requirement of high priority gets completed first
    - EDF can deal with dynamic task arrivals
    - EDF ensures tasks with tight deadlines are completed first, which in a way encourages more tasks to be fitted (...?)
- Cons:
    - Starvation of low priority task: but this should not be a big deal in our case since first, we don't have a huge number of tasks to be scheduled at a time, so tasks with low priority should, at some point, be considered. And second, we do need to consider tasks with high priority first, so even if in the worst case, high priority tasks occupy all the time slots in a schedule, leaving no room for low priority tasks, it's fine in our context
    - Not able to deal with designated starting time (need modification on EDF): our tasks have a designated execution time frame, meaning the start time and end time are specified (ex. Take an image between 9am and 11am). EDF is able to take care of the end time, but probably not the start time. It would be problematic if it scheduled a task between its designated start time. We might need to modify the algorithm for this.
- Time complexity: n = #tasks
    - Divide tasks list into sub lists with different priority: O(n)
    - EDF on each sub list: O(n*logn) * constant
    - When new task comes in (trigger EDF): O(n*logn) * constant
    - OVERALL: O(n*logn)

Knapsack_scheduling:

- Input:
    - A list of tasks, each with associated durations and scores.
    - A list of satellites, each with available time slots.
    - The available time of the satellites.
- Algorithm flow:
    1. Initialization:
       - Initialize a 2D array dp for dynamic programming.
       - Iterate through the tasks and available time slots.

    2. Dynamic Programming:

Fill in the dynamic programming table dp to find the optimal combination of tasks that maximizes the total score within the knapsack capacity.

3. Task Reconstruction:
   Identify and list the specific tasks that have been selected as part of the optimal solution

- Output:
  The tasks selected to maximize the total score within the given time constraints.
- Pros:
  - The algorithm aims to find the optimal solution by maximizing the total score.
  - Can handle various scoring criteria and constraints.
  - Utilizes dynamic programming for efficiency in finding optimal solutions.
- Cons:
  - Resource intensive, Dynamic programming requires additional memory, and the algorithm may not scale well for a large number of tasks.
- Time complexity:
  The time complexity of the dynamic programming-based knapsack algorithm is O(n * W), where n is the number of tasks and W is the knapsack capacity (available time of the satellites).

Genetic Algorithm (GA):

- Input:
  Image Orders: Specifications about the location, time constraints, and type of image requested.
  Satellite Maintenance Activities: Details about onboard actions preventing imaging, with specified execution windows.
  Constraints: Including Ground Station Constraints (locations, uplink/downlink rates), Satellite Constraints (storage, power, orbit), Order Constraints (imaging location, type, deadlines, revisit frequencies), and Maintenance Constraints (execution windows and duration).
- Algorithm flow:
  1. **Initialization**: Generate an initial population of possible schedules (chromosomes) randomly or based on heuristics.
  2. **Selection**: Evaluate the fitness of each schedule using a function that considers all constraints, image orders, and maintenance activities.
  3. **Crossover**: Pair up schedules and combine their features to produce new offspring schedules.
  4. **Mutation**: Randomly modify some schedules to introduce variability.
  5. **Replacement**: Create a new population using the offspring and potentially some of the best schedules from the current population.
  6. **Termination**: Repeat steps 2-5 for a predetermined number of iterations or until a certain performance criterion is met.

If a new task (image order or maintenance activity) is added after the GA has started:

1. The current population can be re-evaluated with the updated fitness function that considers the new task.
2. Alternatively, the GA can be restarted with the new input included.

- Output:
    1. **Satellite Schedule**: A list of actions for each satellite, indicating when imaging or maintenance tasks will be performed.
    2. **Station Schedule**: A timetable indicating when each ground station will be used for uplinking or downlinking, and with which satellite.
- Pros:
  A naturally adapts and evolves solutions over time.Less likely to get stuck in local optima.Evaluates multiple solutions simultaneously, making it suitable for parallel processing.
- Cons:
  Finds approximate solutions, not always the global best.Performance can be sensitive to settings like mutation rate, crossover rate, etc.Depending on the population size and number of generations, it can require significant computation.
- Time complexity:
  The time complexity of a GA is generally O(P * G * F)
  P:size of the population
  G: number of generations.
  F:time taken to evaluate the fitness function for one individual.

[Shortest Task First]:

- Input:
  Image Orders: Specifications about the location, time constraints, and type of image requested.
  Satellite Maintenance Activities: Details about onboard actions preventing imaging, with specified execution windows.
  Constraints: Including Ground Station Constraints (locations, uplink/downlink rates), Satellite Constraints (storage, power, orbit), Order Constraints (imaging location, type, deadlines, revisit frequencies), and Maintenance Constraints (execution windows and duration).
- Algorithm flow:
    1. **Sort Tasks**: Arrange all tasks, which include image orders and satellite maintenance activities, based on their duration in ascending order.
    2. **Allocate**: Starting with the shortest task, allocate each task to its appropriate time slot, ensuring no overlap with previously allocated tasks and considering all constraints.
    3. Continue: Proceed with the next shortest task until all tasks are allocated or it's determined that some tasks cannot be scheduled.

If a new task (image order or maintenance activity) is added after the GA has started:

The new task is added to the list of tasks.The algorithm needs to be re-run to reallocate tasks, especially if the new task affects the positioning of previously allocated tasks.

- Output:
  Satellite Schedule: A list of actions for each satellite, indicating when imaging or maintenance tasks will be performed.
  Station Schedule: A timetable indicating when each ground station will be used for uplinking or downlinking, and with which satellite.
- Pros:
  Simple, predictable, efficient
- Cons:
  No priority handling, starvation of the long task, not always optimal
- Time complexity:
  Allocation: O(n^2)
  Sorting task: O(n log n)

Some thoughts (if time efficiency is important):
Try a non-time-consuming (but not necessary the optimal) algorithm first
Calculate how many tasks are successfully scheduled = # tasks got scheduled
If (# tasks got scheduled) != (# total tasks):

Try a time-consuming algorithm (like brute force) that promises to yield the optimal solution