**Summary**

The scheduling problem described in question is equivalent to the [Multiple Knapsack Problem](https://en.wikipedia.org/wiki/Knapsack_problem#Multiple_knapsack_problem) (MKP) which is NP-Hard problem. As such, the problem doesn’t have optimal (polynomial) solution and all the solutions rely on exhaustive search in the solution domain coupled with a use of one or more heuristics to skip subsets of the solution domain to speed up computation. The use of heuristics creates a trade-off between execution time and accuracy (relatively to optimal solution).

I divided the work to several milestones, where each milestone builds upon previous:

1. Implement naïve solution based on branch and bound approach
2. Implement validation routines
3. Establish metrics and implement measurement
4. Implement random test generator and measure metrics across time
5. Add heuristics, compare against previous runs

I choose Python as a programming language for this project because it allows fast modification-experimentation cycles, which is important in projects like this where the solution is likely to change during implementation. For configuration management I chose git because it is free, easy to use and manages the repositories locally. The project was also pushed to GitHub and can be found here:

As an overarching tradeoff I considered whether it is better to focus on speed or correctness and decided to place correctness first (hence the choice of branch and bound algorithm) for two reasons: lots of research is available on improving the existing underlying algorithm, which makes correctness the more difficult part; and for real life applications, specifically in life-critical domain such as driving a car, correctness is undoubtedly a mandatory requirement.

**Solution Details:**

My solution is based on [Bin Completion](https://www.jair.org/media/2106/live-2106-3172-jair.pdf) method without the added heuristics described in the paper to extract as much value as possible with the allocated effort. There are three main stages in the solution:

1. Sort jobs according to Earliest Start Time (EST) using DFS.
2. Schedule all jobs with EST lower or equal to current time using Bin Completion
3. Advance the time and repeat step 2.

Step 2 (Bin Completion) starts with a list of available resources and ready jobs (based on EST), the goal is then to schedule as much jobs as possible using as few resources as possible. The algorithm returns a schedule consisting of a list of resources and jobs that are scheduled to start on each resource at current time. It is possible that not all of the ready jobs can be scheduled on the existing resources in which case the remaining tasks might push back the EST of jobs that depend upon them, which is handled by increasing the EST of all the subtrees dependent on tasks that were not scheduled in time.

The algorithm iterates over all available resources and recursively calculates the maximum amount of work that can be scheduled using subset of resources containing the current resource. The maximum value is then selected and returned to the caller together with the scheduled resources and tasks. The stop condition for the recursion is when the input contains only one resource. This simpler problem is basically a well-known knapsack problem that is solved using dynamic programming in polynomial time.

**Code Structure**

The project contains four python files:

1. main.py – contains scheduling logic functions and input parsing routines.
2. tests.py – contains random test generation, execution and analysis routines.
3. validation.py – contains routines to validate that the solution doesn’t violate constraints
4. visualization.py – contains routines that print visualization of schedules across all cores

**Usage**

There are two options to run the program. First is automated random test generator that goes over all combinations of jobs/resources and summarizes the results. And the second is running individual scheduling task for input data read from files.

**Run tests.py.**

This script will generate 20 random tests for each combination of 1-10 jobs and 1-10 resources, after which it will execute tests and calculate the average time it took to complete the schedule and print the resulting table:

#Jobs\#Resources | 1| | 2| | 3| | 4| | 5| | 6| | 7| | 8| | 9| | 10|

| 1| 0.3 0.0 0.0 0.1 0.1 0.0 0.0 0.2 0.2 0.1

| 2| 0.1 0.1 0.1 0.1 0.1 0.0 0.1 0.2 0.1 0.7

| 3| 0.1 0.3 0.1 0.1 0.2 0.6 0.7 0.9 0.9 1.3

| 4| 0.2 0.3 0.7 0.5 0.6 1.0 2.7 3.7 4.1 5.8

| 5| 0.8 0.4 0.5 0.5 1.0 1.9 4.8 6.1 11.0 26.9

| 6| 0.4 0.4 0.5 1.1 1.9 3.8 9.7 35.6 34.6 48.8

| 7| 0.5 0.5 0.6 1.0 3.3 4.8 28.7 41.3 87.8 137.7

| 8| 0.6 0.9 1.0 2.0 3.1 11.7 23.4 45.5 248.8 825.3

| 9| 1.3 0.8 1.0 1.9 5.8 12.7 38.8 171.0 461.0 2104.3

| 10| 1.0 1.1 1.6 4.4 6.3 24.5 88.0 236.7 585.8 3122.3

Each cell in the table shows execution time averaged over 20 runs of random tests with number of jobs (rows) and number of resources (columns).

This representation is useful to evaluate modifications aimed to improve run time.

**Run main.py.**

This script will read input data from the files jobs.txt and resources.txt and run the scheduling routine. The files must be located in the same directory as the main.py file and conform to the syntax described in the examples given in problem definition.

The output of this script is a schedule using the format described in problem definition:

*Time: 0*

*task1: compute1*

*task2: compute1*

*task4: compute3*

*task6: compute3*

*task9: compute2*

*Time: 100*

*task5: compute3*

*task8: compute2*

*Time: 150*

*task7: compute2*

*Time: 200*

*task3: compute1*

In addition, the output contains visual representation of the schedules across all cores with average core utilization calculated over the batch run:

*compute1:*

*core0: |task1................| |task3......| 42%*

*core1: |task2.......................................|task3......| 71%*

*core2: |task2.......................................|task3......| 71%*

*compute3:*

*core0: |task4................|task5.................| 57%*

*core1: |task4................|task5.................| 57%*

*core2: |task4................|task5.................| 57%*

*core3: |task4................|task5.................| 57%*

*core4: |task6................|task5.................| 57%*

*compute2:*

*core0: |task9................|task8......|task7..............................| 100%*

*core1: |task9................|task8......| 42%*

*core2: |task9................|task8......| 42%*

*core3: |task9................| |task7...................................| 85%*

*0 21 43 65 87 109 131 153 175 196 218 240 262 284 306 328 350*

Finally, a metric called makespan is printed. Makespan is the time that it took to execute all the tasks with the given schedule and is basically the last executed job end time. This metric is very important because in many applications the goal of the algorithm is to minimize the completion time of all the jobs which translates to minimizing the makespan.

*Makespan: 350*

**Validation**

To make sure that the output of the scheduling algorithm doesn’t violate any constraints, after each execution of scheduling routine a validation routine runs on the output and validates it. There are three main constraints that are being validated:

1. Completeness – all tasks are executed
2. Temporal contention – no two tasks are executed on the same core at the same time
3. Dependencies – no task is scheduled to run before all the tasks it depends on have finished

**Metrics:**

The following metrics to evaluate the solution where implemented:

1. Makespan
2. Core utilization
3. Average execution time

One of the challenges evaluating performance of this algorithm is that there is no reference optimal solution for each given input since this is a problem with the same complexity. Nevertheless, this can be avoided using comparisons to previous runs on the same input across all three implemented metrics. Unfortunately comparison of the metrics across different inputs is difficult since the results vary dramatically with the input. Some noise can be cancelled using averaging as was done for the execution time however it raises another problem of computational resources/long execution time.

**Future Improvements**

1. Various heuristics can be used to improve the effectiveness and efficiency of the algorithm. [This](http://shodhganga.inflibnet.ac.in/bitstream/10603/22714/10/10_chapter%202.pdf) document contains overview and comparison of approaches to solving this problem.
2. Data structures and management – there is a lot of potential to remove redundancy from data structures such as indexing instead of using strings, using trees to store data that need to be constantly sorted and eliminating redundant copying.

**Open Issues**

1. Why does the exhaustive search pack algorithm performs significantly better than dynamic programming?
2. Dependency violation issues