Solution of the test problem by Kirill Atapin

> Task description:
> * Requires a task queue with priorities and resource limits.
> * Each task has a priority and the required amount of resources to process it.
> * Publishers create tasks with specified resource limits, and put them in a task queue.
> * Consumer receives the highest priority task that satisfies available resources.
> * The queue is expected to contain thousands of tasks.
> * Write a unit test to demonstrate the operation of the queue.

Comments:
The python documentation has discussed a similar problem (https://docs.python.org/3/library/heapq.html#priority-queue-implementation-notes). However, this recipe of using heaps cannot be directly applied here because we additionally have to check extra parameters (required resources) before sending the task to a consumer. On the other hand, an amount of "a few thousands" awaiting tasks in the queue seems to be not too big to cause performance problems with any implementation. A "first thought" approach is to store the data in a single python list, and it may be sufficient for the posed problem. More complicated solutions are also considered. Below we assume that higher task priorities correspond to lower values of the priority key.

I. Using a single list as the storage (the TaskQueue_list class in the source code in the attached arhive).
The advantage of this approach is the very fast add_task() method. We can just put a new task to the end of the list using append() which has a complexity of O(1). When get_task() is being called, we use filter() to filters away the tasks that don't satisfy the resource requirements, sort the resultant list and return its first element. Filter() costs O(N). Also, we have to remove the selected task from the storage, this (unfortunately) costs O(N) as well.
The algorithm:
1) storage = list()

2) The add_task() method puts a task to the end of the list
storage.append(new_task) -> O(1)

3) The get_task() method at the first stage the storage and produces a new list filtered_tasks
filtered_tasks = list(filter(condition, storage)) ->O(N)

4) Then we use sort() to sort the tasks by their priorities. Note that it guaranties stability, so the tasks with the same priority will save their order (https://docs.python.org/3/howto/sorting.html#sort-stability-and-complex-sorts)
filtered_tasks.sort(key=attrgetter('priority')) ->O (K log K), where K is the length of the filtered list.

5) Get the first item to send it to the consumer
task_to_return = selected_tasks(0) -> O(1)

6) Remove the task from the storage
storage.remove(task_to_return) -> O(N)

II. The sorted storage
We can make get_task() faster keeping the storage always sorted. Instead of calling sort() every add_task() call , we can use the 'bisect' module (https://docs.python.org/3/library/bisect.html) to determine the position of a new element in the existing list. It costs O(log N) (vs O(Nlog N) for sort()) but now we have to put an element into the middle of the list (O(N)). So add_task() would have a complexity O(N). The get_task() begins a for loop starting from the head of the sorted list to select the task satisfying the requirements. In the most cases (random resource values) the number of attempts will be much less than N. However we still need to remove the selected task from the middle of the storage. So get_task() is also O(N), the sorted list is a bad idea.

Since the priority is an integer (and probability has a small number of grades), instead of sorting, we can divide all the tasks into groups depending on their priority key and store these groups in a dictionary (TaskQueue_dict_of_lists). The expensive removals from the lists can also be avoided if each group is also a dictionary (TaskQueue_dict_of_dicts). The python dicts preserve their keys order since version 3.7.
The algorithm:
1) The storage is a dict of dicts

2) The add_task() method puts a new task into the internal dictionary using the priority key and the task id key.
storage[task.priority][task.id] = task -> O(1)

3) The get_task() begins a double loop over priorities and ids to check the requirements (O(N) in the worst case).
```
for priority in sorted(storage.keys()):
    for id in storage[priority]:
        …
```

4) Get one element from the dict by its key to return it and then delete it from the storage
task_to_return = self.storage[priority][id] -> O(1)
del self.storage[priority][id] -> O(1)

III. Further improvements
If the consumer have strong limitations of its resources, it has to pass through whole the queue to find an appropriate task. Therefore, it could be reasonable to skip groups that don't have any appropriate task at all. It can be implemented by storing the minimal requirements for each group.

Result of the performance testing:
Two cases have been tested: 1) pure random amount of resources in each task and in each consumer, the priority is also random 2) all tasks except one exceed consumer's resource limit, the only appropriated task is located at the end of the queue (the worst case). The number of the tasks were either 10 000 or 5 000 000. First, the queue was filled with tasks and then one get_task() call was made. Such an experiment was repeated 10 times for each test case/implementation to obtain the average time.

Results for 10 000 tasks:

```
Timing for TaskQueue_list:
  random: 0.003083 +\- 0.000486 sec (per call)
  worst:  0.006754 +\- 0.000353 sec (per call)
Timing for TaskQueue_dict_of_lists:
  random: 0.000027 +\- 0.000005 sec (per call)
  worst:  0.006010 +\- 0.000196 sec (per call)
Timing for TaskQueue_dict_of_dicts:
  random: 0.000049 +\- 0.000012 sec (per call)
  worst:  0.001947 +\- 0.000058 sec (per call)
```

Results for 5 000 000 tasks:

```
Timing for TaskQueue_list:
  random: 1.264388 +\- 0.072581 sec (per call)
  worst:  2.736242 +\- 0.046360 sec (per call)
Timing for TaskQueue_dict_of_lists:
  random: 0.000803 +\- 0.000099 sec (per call)
  worst:  2.536824 +\- 0.035148 sec (per call)
Timing for TaskQueue_dict_of_dicts:
  random: 0.000049 +\- 0.000010 sec (per call)
  worst:  0.742462 +\- 0.002045 sec (per call)
```

It is seen that the single-list solution processes a get_task() call in less than 0.01 sec for 10 000 tasks in the queue but the time increases with the number of tasks in the queue. The 'dict of dicts' solution is 100 times faster, and the execution time does not increase (if the properties of the tasks are random).