Motivation of choice

In this exercise we decided to use an AdaptableHeapPriorityQueue because it was the only data structure that allowed us to have a maximum logarithm complexity for the most expensive operations, i.e. remove_min, add and update, thus allowing us to achieve the best performance.

Computational Complexity

For the computation complexity calculation we will consider only the portion of code inside the while(true).

In the first part of the function the only operations that have a time complexity grater than O(1) are the add() and the remove_min() method of the AdaptableHeapPriorityQueue, that have a complexity of O(log(n)).
In the second part (when we update the priority of the jobs) there is an for-loop on the n elements of the queue. In this loop there is also the update method of the priority queue that has an computation complexity of O(log(n)). So, this loop has a complexity of O(n*log(n)).

Therefore, the whole algorithm has an computational complexity equal to:

0(log(n)) + 0(n*log(n)) = I(n*log(n))

If we also consider the cycles of the CPU, the computational complexity of the algorithm will be multiplied by the sum of the length of all the jobs. As a result we will have:

O(n*log(n))*$\sum_{i=1}^{n} apq.\_data[i].\_value.length$