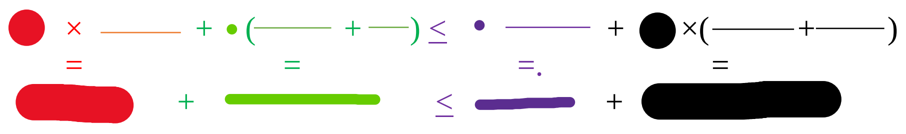
Adam Frenkel

My Algorithm works by assigning a value to job by diving w_i/d_i. I then sort the jobs by these values (highest value first). Then I simply add all the W_i's of each job. I accomplish by going through the sorted jobs, calculating each one's W_i and then adding them all together.

Intuitively this algorithm makes sense, because in order to minimize the total weighted wait time, there are two different factors you have to balance. You want to do jobs with the smallest duration first, and jobs with the largest weight weight. By dividing w_i/d_i, you perfectly balance the two of these. By increasing the weight and decreasing the duration of a job you get a higher values, thereby giving the job a higher priority to be done.

- Proof:
- Lemma: By adding the next job based on the highest value job, determined by w_i/d_i, you will end up minimizing the total weighted wait time of the problem than if you added another job.
- Direct Proof: This job will have the largest weight relative to its duration, compared to the remaining jobs.
- If you add this job (job_a) later than the current job (job_b) being added, then job_b will have a smaller weight relative to the duration being added than what would have been added if job_a had been added, by the properties of division.
- That means that when you add job_a later, you will be multiplying a larger weight times a larger duration (relative to if you added job_a first).
- Meaning, had you added job_a first, you would have multiplied a larger w_i times a smaller W_i, then later you'd multiply a smaller w_j times a larger W_j. (All sizes are relative, on the last line of the lemma this will be clear.)
- Adding job_a first will be better than multiplying a smaller w_i times a smaller W_i, then later multiplying a larger w_j times a larger W_j.
- We know this is true, because by the properties of division if $w_i/d_i \geq w_j/d_j$. Then, $w_i \times W_i + w_j \times (W_i + d_i) \leq w_j \times W_j + w_i \times (W_j + d_j)$ QED
- Here is an illustration of this lemma: (All sizes are relatively true because: $w_i/d_i \geq w_j/d_j$)

$$w_i \times W_i \quad + w_j \times (W_i + d_i) \quad \leq \quad w_j \times W_j \quad + \quad w_i \times (W_j + d_j)$$



- Theorem: My algorithm is optimal. (M (my algorithm) = O (the optimal Algorithm)).
- Base Case: W_i will be 0 in all algorithms, so this must be the same. ($M_1 = O_1$)
- Inductive step: Let's assume after W_j has been added to the total in M, there exists an optimal feasible series of jobs such that M [0..j] = O[0..j].
- Now I'll prove that after W_{j+1} has been added to M, there still exists an optimal ordering of jobs O_{new} such that $O_{new}[0..j + 1] = M[0..j + 1]$/
- Suppose we add job_a as M[j+1] (that's the decision made by the greedy algorithm at the jth + 1 step (we'll call this k)).
- By the lemma, We know that M's $W_k \leq$ O's W_k relative to the duration being added.
- Therefore, if algorithm O* (O with an exchange) would take job_a, instead of the job it's taking, job_b. Then O* could still take job_b at some later point and the total (after summing together all the W's) would be $\leq$ O, meaning O* $\leq$ O.
- This can be said about any step after this as well. QED

-My algorithm takes *O*(nlog(n)) time. (n being the length of the duration/weight array.
-This is because my algorithm goes though the duration and weight arrays one, which takes O(n).

-Then my algorithm performs a sort on a new array it created which is also size n. This takes O(nlog(n)) time.
-Finally, my algorithm goes thought this newly created array which takes O(n).
- O(n) + O(nlog(n)) + O(n) = O(nlog(n))