CMSC 651, Spring 2018, University of Maryland
HW3, due as PDF to the address *cmsc651.umd@gmail.com* by 11:59PM on March 13, 2018

Notes: (i) Please work on this with your group (one writeup per group). Consulting other sources (including the Web) is not allowed. (ii) Write your solutions neatly and *include your names*; if you are able to make partial progress by making some additional assumptions, then **state these assumptions clearly and submit your partial solution**. (iii) **Please make the subject line of your email** "*651 HW3*" followed by your full name, and email a PDF to *cmsc651.umd@gmail.com*: PDF generated from Word or LaTeX strongly encouraged.

Note. As before, when we refer to the Williamson-Shmoys book below, please use the *version of the book posted under "Resources".*

1. Exercise 2.1 from Williamson-Shmoys. **(10 points)**

2. Exercise 2.2 from Williamson-Shmoys. **(10 points)**

Suppose we have n machines and k jobs, with the jobs' requirements labeled $p_1, p_2, \ldots, p_k$. Since the requirement of each job is strictly greater than one-third of the total job time, each machine can have no more than two jobs. It follows that $k \leq 2n$. WLOG, we arrange them so that $p_1 \geq p_2 \geq \ldots \geq p_k$. And WLOG, let's pad our list of jobs with jobs of requirement zero until we have exactly $2n$ jobs, so that $p_1 \geq p_2 \geq \ldots \geq p_{2n}$ where $p_i = 0$ for $i > k$.

The LPT algorithm will assign jobs to the machines as follows: $(p_1, p_{2n}), (p_2, p_{2n-1}), \ldots, (p_n, p_{n+1})$. Basically the largest job gets paired with the smallest job, the second-largest job gets paired with the second-smallest job, etc. We call this the "LPT assignment" and we need to show that this particular way of assigning jobs produces an optimal solution.

Suppose we have an optimal solution, with the slowest machine having jobs i and j.

We introduce terminology: a slow job is one whose index lies between 1 and n (its time requirement is high), and a fast job is one whose index lies between $n+1$ and $2n$ (its time requirement is low).

Claim: one of i and j is a fast job and the other is a slow job. Suppose not and they are both slow. Then, by pigeonhole principle, there must be a machine with two jobs that are fast. It follows that if we did a job swap between a job from the first machine and a job from the second machine, each of the resulting machines would have run-time quicker than the machine that previously had the two slow jobs. Since the machine that previously had the two slow jobs was the slowest machine of the schedule, we have hence improved the schedule, which is a contradiction of our original arrangement being optimal. Similarly, if we suppose that i and j both are fast, we also get a contradiction because we can find a machine with two slower jobs, which contradicts the assumption that i and j belong to the longest-running machine.

Claim: we can make every other machine in our solution also have one slow job and one fast job, without disturbing the optimality condition. This is because, for each machine with two fast jobs, we can find another machine with two slow jobs, and apply the "swapping" argument from above.

So far, we have shown that any optimal solution can be expressed with each machine having one slow job and one fast job.

Claim: if, in a schedule, each machine has one slow job and one fast job, the schedule will either have the same run-time as the "LPT assignment," or we can improve it by making it look like the "LPT assignment." First take the machines and order them by the slow jobs they were given, so we have machines ordered $1 \ldots n$. We argue that if their paired "fast" jobs are not $2n \ldots n+1$, then we can improve the schedule time. Take the machine with the longest run-time, call the jobs i and j, where i is the slow job and j is the fast job. If i and j correspond to one of the pairings of the LPT assignment, then we know that this arrangement has the run-time of the LPT assignment and we stop there. If i and j do not come from one of the pairs of the LPT assignment, then we know that the ordering of fast jobs is not $2n \ldots n+1$. Depending on how this ordering

has been permuted, we can find a slow job $i*$ which is slower than i, but which is paired with a fast job $j*$ that is nevertheless slower than j; or, we can find a slow job i_* which is faster than i, but which is paired with a fast job j_* that is nevertheless faster than j. Whichever the case may be (and it's possible to find both), we can do a job swap and improve the run-time of the resulting machines, and hence of the schedule.

Hence, to wrap things up, any optimal solution can be expressed with each machines having one slow job and one fast job. Any solution with one slow job and one fast job will have the same run-time as the "LPT assignment" or we can improve it by making it look exactly like the "LPT assignment." Hence the LPT assignment is indeed optimal.

3. Exercise 2.3 from Williamson-Shmoys. **(10 points)**

Let's call the jobs $x_1, \ldots, x_n$. The way the algorithm will work is that it will start packing in jobs "tightly," i.e., in the usual greedy fashion with no gaps, until it comes across a "dominant" job which all remaining jobs depend on - the algorithm cannot proceed with the scheduling until this job has finished. If x_{*i} is a dominant job, that means for all remaining jobs x, $x_{*i} \prec x$. The algorithm will then place x_{*i} down according to the next available machine, and then there will be a gap across all machines until this dominant job is finished. The max gap length in this situation is the time requirement of x_{*i}.

Let's say the schedule that the algorithm produces is a bunch of tightly packed areas interspersed with a string of gaps determined by dominant jobs labeled $x_{*1}, \ldots, x_{*p}$. First of all,the tightly packed areas, since they are filled with jobs while entertaining no gaps, cannot be longer than a full schedule of this problem, and hence cannot be longer than the optimal schedule. Second of all, the gaps also cannot be longer than the optimal schedule. This is because the sum of the gap lengths is at most the sum of the time requirements of $x_{*1}, \ldots, x_{*p}$. However, note that $x_{*1} \prec \ldots \prec x_{*p}$, and so therefore this non-overlapping sequence of jobs must appear in any feasible schedule of the problem, and hence in the optimal schedule. So the optimal schedule must also be longer than the sum of the time requirements of $x_{*1}, \ldots, x_{*p}$. So finally, both parts of the schedule produced by the algorithm can be no longer than the optimal schedule, and therefore the algorithm is a 2-approximation for the problem. Note: the algorithm is clearly polynomial.

4. Exercise 3.1 from Williamson-Shmoys. **(10 points)**

5. Exercise 3.2 from Williamson-Shmoys. **(10 points)**

6. Exercise 3.3 from Williamson-Shmoys. **(10 points)**