Midterm 2 Solutions

1 True Xor False [2 points each]

Write either True or False to the left of the number.

1. The UC Berkeley Blue team, which includes CS 170 GSI Lewin Gan, beat the Stanford Cardinal team in the regional ACM International Collegiate Programming Contest. (Hint: we are at UC Berkeley.)

   True, of course.

2. Integer factorization is the problem of determining whether a number is *not* prime. This problem is in NP.

   True, given a non-prime n and a proper divisor p of n, it is easy to verify in polynomial time that n is not prime. Hence the above problem is in NP.

3. The Ford-Fulkerson algorithm can be used to find the maximum flow through a graph with cycles.

   True, Ford-Fulkerson works for any graph.

4. All Huffman encoding trees are full binary trees. (That is, each node has zero or two children.)

   True, by definition of Huffman encoding trees.

5. If a linear program has an unbounded value, then its dual is also unbounded.

   False, if an LP has unbounded value, then its dual must be *infeasible*.

6. If one uses the disjoint sets data structure on n elements with path compression, where one does some unions and then does a sequence of n finds (without any intevening unions), then the subsequent n finds have $O(1)$ amortized complexity.

   True, this was covered in class. The reason is that we can give dollar to each of n nodes, and this node can use this node to pay for the change in its parent pointer during path compression. This there are no-unions, this happens at most once to any node and hence the amortized cost is $O(1)$.

7. If a dynamic programming algorithm has n subproblems, then its running time is $O(n)$.

   False, each subproblem can still take more than $O(1)$ time, e.g., in the Longest Increasing Subsequence problem.

8. Each vertex of a multi-variable linear program's feasible region occurs at the intersection of at least two tight constraint inequalities. (A tight constraint inequality is an inequality, or even equality, constraint that holds with equality. A vertex is defined as unique point in which some subset of hyperplanes meet.)

   True, if there are $n \geq 2$ variables, a vertex is at the intersection of at least n tight constraint inequalities.

9. In an n-dimensional space, it is impossible to have a vertex that is represented by fewer than n tight constraint inequalities.

   True, same reasoning as last question.

10. Kruskal's algorithm requires at least linear time to test whether adding an edge will create a cycle.

    False, we saw several disjoint set data structures which can do it much faster than that.

11. Given any LP, all feasible points that have an optimal value are vertices.

    FALSE, while at least one optimum will be a vertex, non-vertices can also achieve the optimal value.

12. Given a spanning tree and a cut, you can exchange the lightest edge across the cut with any of the tree edges across the cut to obtain a lighter spanning tree.

    FALSE, doing so for a general cut can produce a cycle if there are multiple tree edges crossing the cut. However, it is guaranteed to produce a lighter spanning *tree* if there is exactly one tree edge crossing the cut.

13. In the DP algorithm for Knapsack (without repetition), since for each item i, the recurrence considers both cases of adding item i or not, the algorithm inevitably calculates the cost of all possible subsets of all n items.

    FALSE, if this were true, the algorithm would take time at least $\Omega(2^n)$, since it will have to compute a number for each of the 2^n sets.

14. In zero-sum games, once we fix the strategy (distribution over moves) of the row player, the column player doesn't need to randomize his/her moves to achieve a best payoff.

    TRUE, once the row player announces her strategy, the column player simply needs to pick the column achieving the best expected payoff with respect to this strategy.

15. Suppose we have a DAG and u comes before v in a topological ordering. Then, the length of the longest path ending at u is less than or equal to the the length of the longest path ending at v.

    FALSE, u can come before v in a topological ordering *without* there being a path from u to v, in which case it is easy to construct examples where the longest path ending at u is shorter than the longest path ending at v.

16. We have n operations, each of which takes amortized $O(1)$. Then, the worst case running time for any single operation can be as bad as $\Theta(n^2)$.

    FALSE, since the n operations have amortized running time $O(1)$, their total running time is at most $O(n)$.

17. For a graph with nonnegative edge weights, when a node u is removed from the priority queue in Dijkstra's algorithm its distance label is correct (i.e., equal to the length of the shortest $s - u$ path.)

    TRUE, this is exactly the invariant maintained by Dijkstra's algorithm.

18. Recall that a pass of the Bellman-Ford algorithm is to update all edges. After k passes of Bellman-Ford, at least k nodes have the correct distance label in a graph with no negative cycles.

    TRUE, after $k \leq n$ passes, Bellman-Ford would have found all shortest paths that use k or fewer edges, since there are no negative weight cycles. This implies that at least k vertices have already received their shortest path labels.

19. It is possible for a dynamic programming algorithm to have exponential running time.

    TRUE, we saw such an algorithm for TSP.

20. A linear program with n variables and $m < n$ constraints will have an unbounded feasible region.

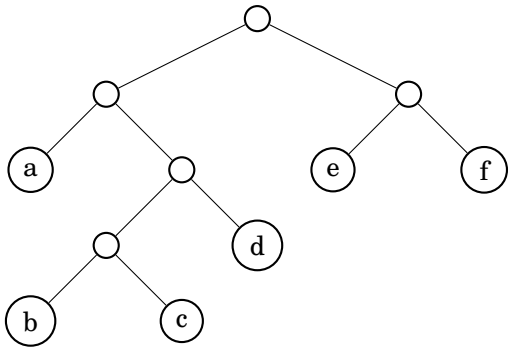
    FALSE, we can, for example, construct an infeasible program with $n \geq 4$ variables and just three constraints.

2 Give me an example, I'll move the world. [20 points]

Short answer. No need to justify, though you may have a brief justification if it is helpful.

1. Huffman Encoding.

   (a) Give an example of frequencies $f_a, f_b, f_c, f_d, f_e, f_f$ such that when Huffman encoding algorithm is run, the following tree is constructed.



   Solution: Many possible solutions (e.g. $f_a = 0.25, f_b = 0.0625, f_c = 0.0625, f_d = 0.125, f_e = 0.25, f_f = 0.25$). Answer did not have to be normalized.

   (b) What is the highest that f_e could be for this tree to be returned by Huffman encoding?

   Solution: 0.4. 1) proof from discussion notes 7 proves upper bound and 2) the following example shows feasibility: $f_a = 0.2, f_b = 0.05, f_c = 0.05, f_d = 0.1, f_e = 0.4, f_f = 0.2$. This can also be proved using linear programming.

2. Given a graph G with nodes s, a, b, t, and edges (s, b), (a, t) and (a, b) with capacity 1, and edges (s, a) and (b, t) with capacity 3.

   (a) What is the maximum flow function?

   Solution: $f(E(s, b)) = 1, f(E(a, t)) = 1, f(E(a, b)) = 1, f(E(s, a)) = 2, f(E(b, t)) = 2.$

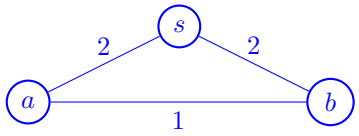
   (b) What is the minimum $s - t$ cut?

   Solution: $S = \{s, a\}, T = \{b, t\}$.

3. Give an example of a graph with edge weights where the minimum spanning tree is different from the shortest path tree from some vertex. (A graph with more than five nodes will receive no credit!)

Solution:

The question is asking for a graph where the tree of shortest paths from some vertex to the other vertices in the graph cannot be a minimum spanning tree. In the following graph, all MSTs will contain the edge from a to b, but the shortest paths tree starting at s will have only the edges from s to a and s to b.



Comments: Partial credit was given to solutions where one of the shortest path trees was an MST (there can be several because of ties) and solutions where there was no SPT because of negative edges.

4. We are given the following linear program:

$$\begin{aligned} \min\, & 3x_1 + x_2 \\ & 2x_1 - x_2 \leq 5 \\ & x_1 + 3x_2 \leq 7 \\ & x_1 \geq 0 \end{aligned}$$

We want to convert this linear program into the following form:

$$\begin{aligned} \max\, & c^T x \\ & Ax = b \\ & x \geq 0 \end{aligned}$$

What is the new linear program? Hint: You may introduce new variables. (You do not need to put it in matrix form, just a set of equality constraints with variables appropriately identified with the variables in the original formulation.)

Solution:

$$\begin{aligned} \max\, & -3x_1 - x_2^+ + x_2^- \\ & 2x_1 - x_2^+ + x_2^- + s_1 = 5 \\ & x_1 + 3x_2^+ - 3x_2^- + s_2 = 7 \\ & x_1, x_2^+, x_2^-, s_1, s_2 \geq 0 \end{aligned}$$

3 How low can you go? [12 points]

1. Given a graph, G with edges capacities, c_e and a source s and sink t, and a maximum flow solution which assigns a flow, f_e, to each edge e, give a linear time algorithm for finding a minimum $s-t$ cut. (The algorithm should output the partition of nodes (S,T). You may use the residual graph without explaining what it is.)

   Solution: Delete any edges with residual capacity 0. S is the set of all nodes that are reachable from s (which can be found by a DFS from s), and $T = V - S$.

   Comments: This was covered in lecture and in the book (page 215).

2. You are given an undirected graph $G = (V, E)$ and a penalty function $P(x)$. Each edge $(u, v) \in E$ has a positive weight $w_{u,v}$. Give an algorithm that returns a set of edges $E' \subseteq E$ that minimizes $P(k) + \sum_{(u,v)\in E'} w_{u,v}$ where k is the number of connected components of $G' = (V, E')$.

   Solution: Sort the edges from smallest to largest w_e. Start with $E_{|V|} = \{\}$. Let $k = |V|$. For each edge e in sorted order, if e does not create a cycle in E_k, let $E_{k-1} = E_k \cup \{e\}$.

   Output E_k that minimizes $P(k) + \sum_{e\in E_k} w_e$.

   Comments: We did not require justification, but a key point is the penalty only depends on the number of components, thus we only need to evaluate the expression for a minimum cost set of edges which have k components for each k.

   In the procedure above, which is essentially Kruskal's algorithm, E_k is a minimum weight set with respect to having k components as can be seen as follows. Consider the first point where this greedy algorithm chooses an edge, e, that is not in any minimum weight solution for the minimal k component problem. Consider a minimum weight solution, F, to the k component problem. If adding e to F could connect two components in F one could replace the heaviest edge in F with e and produce a k-component set of edges. Edge e' must be at least as heavy as e as the algorithm could have chosen it, and $F + e - e'$ must be at least as good a solution to the k-component problem. If e creates a cycle in F, as with MSTs, there must be an edge, e' in F which is at least as heavy as e on the cycle and again $F + e - e'$ is at least as good a solution to the minimum k-component problem. Thus, the greedy algorithm can't disagree with all optimal solutions to the k-component problem.

4 Don't be static! [18 points]

No need to justify, though you may have a brief justification if it is helpful.

1. Alice and Bob put n pebbles on a table, and share a sorted list of numbers $[a_1, \ldots, a_k]$. They then take turns removing pebbles. In each turn, they are allowed to remove a_i pebbles (for any a_i) or all of them if there are fewer than a_1 pebbles left. The player who removes the last pebble loses. Alice always goes first because her name comes first in the alphabetical ordering.

   Alice and Bob are CS170 students, and hence will play optimally. We wish to find an algorithm that, given n and $[a_1, \ldots, a_k]$, determines who will win the game. Let $A(n)$ be true if Alice wins for n pebbles.

   (a) What is $A(i)$ if $i \leq a_1$?

   Solution: Since Alice will have to remove all the pebbles in this case and hence lose, we have $A[i] = \textsc{False}$ for all $1 \leq i \leq a_1$.

   Comments: Several answers misinterpreted the game and set $A[i] = \textsc{True}$. Note that the player to move the last pebble **loses**.

   (b) Give a recurrence for $A(n)$.

   Solution: Note that if it is Alice's turn, she can win if and only if she can make a move to a position where the first player to win (which would now be Bob) *loses*. Since $A[i]$ is supposed to be $\textsc{True}$ if and only if the first player can move when there are i pebbles, we see that this translates to the recurrence

$$A[n] = \bigvee_{i:a_i<n} (\neg A[n - a_i]) \text{, for } n > a_1.$$

   The base cases are already handled in part (a).

   Comments: Several answers lost point for not complementing the value of the $A[i]$ used in the recurrence. We did not take away points even if the condition $a_i < n$ was not emphasized, or if a less efficient "two-level" recurrence was used (which considers each choice of Alice and then looks at Bob's best possible move in that scenario).

2. Consider a knapsack problem with n items with weight w_i and value v_i where repetition is allowed but one wishes to have at least K different items. Give an algorithm for finding the maximum value knapsack of weight W that has at least K different items. Let $K(w, k, i)$ be the maximum value of a knapsack of weight w that uses only items $1, \ldots, i$ that uses k different items. Give a recurrence for computing $K(w, k, i)$ as well as base cases.

   Solution:

   Recurrence: Consider $K(w, k, i)$, the optimal value for using at least k different items among $1, \ldots, i$ with total weight no more than w, there are three cases regarding item i

   (a) The optimal solution doesn't use i, then the optimal value is the same as $K(w, k, i-1)$

   (b) The optimal solution uses i once, then the optimal value is the same as $K(w - w_i, k - 1, i - 1) + v_i$

   (c) The optimal solution uses i more than once. In this case we just take one copy away, and the value is $K(w - w_i, k, i) + v_i$, notice we don't decrease i, since we will (potentially) use i again, and we don't decrease k here, since we want to decrease it only once for all copies of item i, and we wait until the last time we use i to decrease k.

   Thus we have the recurrence $K(w, k, i) = \max\{K(w, k, i-1), K(w - w_i, k - 1, i - 1) + v_i, K(w - w_i, k, i) + v_i\}$

Base case: For the above recurrence, we need to deal with three base cases

(a) $w < 0$: this case violates the weight constraint, we need $K(w, k, i) = -\infty$

(b) $w \geq 0, i = 0, k = 0$: We have no more items to choose, so $K(w, k, i) = 0$

(c) $w \geq 0, i = 0, k > 0$: this case violates the number of different items constraint, we need $K(w, k, i) = -\infty$.

Notice in the base cases, if you have 0 instead of $-\infty$ for those cases, you may end up with a solution not satisfying the constraints.

Alternatively, you can have the recurrence $K(w, k, i) = \max\{\max_{1 \leq a \leq \lfloor w/w_i \rfloor}\{K(w - aw_i, k - 1, i - 1) + av_i\}, K(w, k, i - 1)\}$ if you want to make the decision of how many copies of item i to use all at once. You won't need the $w < 0$ base case then, since your recurrence will never incur $K(w, k, i)$ for $w < 0$.

3. A zig-zagging sequence is a sequence of numbers, $a_1, \ldots, a_k$ where $a_1 > a_2 < a_3 \cdots$ or $a_1 < a_2 > a_3 \cdots$: for example, $[3, 2, 4, 3, 5]$ is a zig-zagging sequence.

   Give a subproblem definition, recurrence, and base cases for your subproblem for a dynamic programming algorithm for finding the length of the longest zig-zagging subsequence in a sequence. Also, state how you would output a final answer.

   (Note that $[1, 9, 7, 10, 4]$ is a zig-zagging subsequence of $[1, 5, 9, 8, 7, 10, 4]$.)

   The running time is not needed, but the associated dynamic programming algorithm should be polynomial.

   Solution: Let $Z(i, r)$ denote the longest zig-zagging subsequence using the prefix $a_1, \ldots, a_i$ which uses a_i. The second argument r denotes the last inequality, which is 1 if the last inequailty was $>$, and 0 if the last ineuqliaty was $<$.

   Comments: Splitting up into two cases was the key thing here. Solutions that didn't do this were very likely to be incorrect
   We define our base case to be $Z(1, 0) = Z(1, 1) = 1$, and we define our recurrence to be

$$Z(i, r) = \max_{k<i} Z(k, 1 - r) + 1$$

   where if $r = 1$, only $a_k < a_i$ are considered and if $r = 0$, only $a_k > a_i$ are considered (if no such k exists, we just set $Z(i, r) = 1$)

   Comments: This is like mutual recursion, since both recurrences interact with each other
   The problem is only asking for the longest length, so we can take

$$\max_i(\max(Z(i, 0), Z(i, 1)))$$

   as our solution

   Comments: Many people only took $\max(Z(k, 0), Z(k, 1))$, which although is valid did not completely follow from the defintion of the subproblems. Full credit was awarded for solutions that said only check the last entries, if they proved that the last entry was always part of some longest zigzagging sequence

   Comments: There was a greedy solution that I may have overlooked while grading. I was a bit more strict with these solutions, since the problem did ask explicitly ask for a dynamic programming solution, but if the greedy solution was correct AND had a correct justification, I awarded full credit.

   Comments: An extention to think about: how would you do longest common zig-zagging subsequence? (given two sequences, find the longest zig-zagging sequence that is a subsequence of both) If both sequences have length n, this is still possible in $O(n^2)$

5 War and Peace [12 points]

You may use facts from class about two person games and linear programs.

1. In a battle for China, Mulan is once again defending against the Huns. She models her choices as the following two person game:

	Attack	Attack Harder
Rocket	1	-1
Dragon	-1	2

Recall, the row player (Mulan) wishes to minimize the expected payoff. Mulan plays Rocket with probability .6 and Dragon with probability .4.

(a) What is the best (maximum) payoff the Huns can get?

Solution: If the Huns choose Attack, then the payoff is $(1 \times 0.6) + (-1 \times 0.4) = 0.2$. If the Huns choose Attack Harder, then the payoff is $(-1 \times 0.6) + (2 \times 0.8) = 0.2$. That is, regardless of what the Huns choose to do, Mulan's strategy of playing Rocket with probability .6 and Dragon with probability .4 forces the expected payoff to be 0.2.

Comments: We meant to ask for the best expected payoff the Huns could achieve. Students who put down 2 (which corresponds with Dragon and Attack Harder) received partial credit.

(b) Either provide a strategy that is better for Mulan or give a short proof that her strategy is optimal. (Hint: for the latter, give a strategy for the Huns which ensure that Mulan can get no less.)

Solution: The payoff matrix is symmetrical. Thus, if the Huns choose Attack with probability 0.6 and Attack Harder with probability 0.4, then the Huns can also force the expected payoff to be 0.2. Since the Huns can guarantee that the expected payoff is at least 0.2, Mulan's strategy of forcing the payoff to be 0.2 is optimal.

2. It's peacetime. Mulan wishes to manufacture rocket fireworks, or snake fireworks. She needs 1 unit of black gunpowder and 2 units white gunpowder for a rocket firework and 2 units of black gunpowder and 2 unit of white gunpowder for a snake firework. She makes $\$ 3$ for each rocket and $\$ 4$ for each snake. And has 150 units of black gunpowder and 200 units of white gunpowder. She determines that she should make 50 rocket fireworks, and 50 snake fireworks. Either find a better way for Mulan to choose what to manufacture or prove that she has done the best possible thing. (Hint: she is making 350 dollars.)

   Comments: The hint is referring to the fact that by making 50 rocket fireworks and 50 snake fireworks, Mulan will make $350. It is not stating that the optimal amount of money Mulan can make is $350.

   Solution: Define the following:

   - r = number of rocket fireworks Mulan makes
   - s = number of snake fireworks Mulan makes

   The linear program for this problem is:

$$\begin{aligned} \max\ & 3r + 4s \\ r + 2s &\leq 150 \quad (1) \\ 2r + 2s &\leq 200 \quad (2) \\ r, s &\geq 0 \end{aligned}$$

   Constraint (1) corresponds to Mulan's limited supply of black gunpowder and constraint (2) corresponds to her limited supply of white gunpowder.

   After this, there were many ways to proceed:

   - We can add constraints (1) and (2) to get the inequality $3r + 4s \leq 350$. This tells us that Mulan's plan to make 50 rocket fireworks and 50 snake fireworks (which would make Mulan $350) is optimal.
   - Notice that no matter what type of fireworks Mulan makes, she gets at most $1 per unit of gunpowder. Since she has 350 units of gunpowder, she can get at most $350, and so her strategy is optimal.

     Comments: You did not even need to write the linear program for this solution. This is basically the same as adding the constraints (1) and (2) in the linear program above.
   - Solve the primal graphically or by enumerating vertices. The vertices are $(0, 0)$ with profit 0, $(100, 0)$ with profit 300, $(0, 75)$ with profit 300, and $(50, 50)$ with profit 350. Since the best vertex must be optimal, $350 is indeed optimal.
   - Write down the dual:

$$\begin{aligned} \max\ & 150y_1 + 200y_2 \\ y_1 + 2y_2 &\geq 3 \\ 2y_1 + 2y_2 &\geq 4 \\ y_1, y_2 &\geq 0 \end{aligned}$$

     $y_1, y_2 = 1$ is a feasible point with value 350. Since both the primal and the dual can achieve 350, it must be optimal.
   - Run simplex and show that it ends at the vertex $(50, 50)$.
   - Related to the above - if you already expected that $(50, 50)$ was optimal, you could do a coordinate transformation to make $(50, 50)$ the origin:

$$\begin{aligned} y_1 &= 150 - r - 2s \\ y_2 &= 200 - 2r - 2s \\ r &= y_1 - y_2 + 50 \\ s &= 50 - y_1 + \frac{y_2}{2} \end{aligned}$$

     The objective then becomes $3(y_1 - y_2 + 50) + 4(50 - y_1 + \frac{y_2}{2}) = 350 - y_1 - y_2$.
     Since the coefficients are negative, this must be optimal.

6 All work and no play.... [18 points]

In this problem, please justify the correctness of your algorithm.

Assume we have N workers. Each worker is assigned to work at one of M factories. For each of the M factories, they will produce a different profit depending on how many workers are assigned to that factory. We will denote the profits of factory i with j workers by P_i^j.

1. How would you find the assignment of workers that produces the most profit? What is the running time?

   Solution:

   We will use dynamic programming where $B(m, j)$ denotes the best cost of the best assignment of j workers to the first m factories.

   An optimal solution of j workers to the first m factories assigns some number of workers, k, to factory m, and must assign $j - k$ workers with maximal profit to the first $m - 1$ factories, which, by induction, has cost $B(m - 1, j - k)$. Thus, we have

   $$B(m, j) = \max_{0 \le k \le j} P_i^k + B(m - 1, j - k).$$

   Furthermore, the base case is $B(0, 0) = 0$; the profit of assigning no worker to no factory is 0.

   Comments: Here, we were a bit picky about proofs as this was the only problem that require justification. We looked for the notion that one need only consider an optimal solution to the smaller subproblem. Here a word or two could make a difference, I am afraid, some correctness arguments used the phrasing search over solutions which we viewed as incorrect, others used search over the number of workers assigned to m, which is correct.

   The time to compute each value $B(m, j)$ is $O(M)$ and since there are MN subproblems, the total runtime is $O(M^2N)$.

   To recover the assignment, one backtracks from $m = M$ and $j = N$ and assigns k workers to machine m if $B(m, j) = B(m - 1, j - k) + P_i^k$ and continues to backtrack from $m - 1$, $j - k$.

2. How can this algorithm be improved if we enforce the law of diminishing returns? Namely that for each factory each additional worker (past the first worker) cannot result in a larger increase in profits than the previous worker. (For example, $P_1^1 = 5$ and $P_1^2 = 7$ obeys diminishing returns as the first worker adds profit 5 and the second adds profit 2.)

Solution:

The solution is to assign the N workers greedily according to the largest available increase in profit: that is, for a factory i with j workers the increase in profit is $P_i^{j+1} - P_i^j$. (We assume $P_i^0 = 0$.)

Comments: Several solutions, sorted by P_i^j/j which does not work. Consider $N = 2$, and $P_1^1 = 4$, $P_1^2 = 5$, and $P_2^1 = 2$. One obtains a solution of value 5 instead of 6.

Consider the first point where an optimal solution, S, differs from this algorithm's solution A. That is, A assigns a worker to position j for factory i and S assigns no worker here. S must assign some worker to a position j' in factory i' where A assigns no worker. The profit increase for factory i' at position j' is less than profit increase for i' for the current number of workers assigned to i' by the diminishing return condition, which, in turn, is at most the profit increase for worker j on factory i since the algorithm producing A has both choices available to it and it chooses the largest. Thus, we can move a worker assigned to factory i' in S to factory i and produce an improved solution that is consistent with greedy for at least one more step. By induction, we can conclude that repeatedly choosing the largest increase in profit produces an optimal solution.

Comments: Few actually did an exchange argument. Many suggested that since greedy chooses the maximal profit that it must be optimal. The problem with this argument is that it does not use the diminishing returns condition explicitly. One needs to use diminishing returns to argue that *any* future worker will have a profit increase that is at most the current best option. Without diminishing returns, one could find that by scheduling a worker who adds a very low profit increase may provide a later opportunity to assign a very high profit worker. Indeed, this is why the greedy algorithm fails for part (a).

An efficient implementation of this algorithm uses a max-heap, with an entry for each factory i with initial key P_i^1. Then we repeatedly assign each worker to the factory i with the maximum key. When a is assigned to factory i, we re-insert factory i into the queue with key $P_i^{j+1} - P_i^j$ where j is the number of workers currently assigned to factory i. The time for each extract-max and insert is $O(\log M)$, thus the total time to implement the algorithm is $O((N + M) \log M)$ as there are at most $(N + M)$ inserts and N extract-max's.

Comments: Some failed to use a max heap here.