Advanced Algorithms 2014. Exam Answers

1. Already the case $m = 2$ and $p = 0$ is NP-complete: We can reduce the original Load Balancing problem to Load Balancing with Preemptions. Namely, a Load Balancing instance $t_1, \ldots, t_n$ with $m = 2$ allows a makespan $\sum_{j=1}^{n} t_j/2$ if and only if the same sequence allows a solution to Load Balancing with Preemptions where $p = 0$.
(The "overall" NP-completeness does not exclude polynomial-time solutions for other special cases, achieved in various ways.)

2. Idea: One has to compare the greedy and optimal solution. An immediate observation is that the greedy algorithm can, in each step, add at least the size of some set from the optimal solution, minus the already covered elements. Full elaboration (not expected) could work as follows:
Consider any optimal solution $P \cup Q$ that covers $c := |P \cup Q|$ elements. Clearly, $|P| \geq c/2$ or $|Q| \geq c/2$. Since X is the largest set, it follows instantly $|X| \geq c/2$. We cannot know in general where P, Q, X are located. So let us define $s := |(P \cup Q) \cap X|$. Then $c - s$ elements of $P \cup Q$ are not in X. Similarly as above, P or Q has at least $(c-s)/2$ of these elements outside X. Since Y has the largest difference to X, the second step adds at least $(c-s)/2$ elements to the solution. In total we get $|X \cup Y| \geq \max(s, c/2) + (c-s)/2 = \max((c+s)/2, c - s/2)$. The worst case is $(c+s)/2 = c - s/2$, thus $s = c/2$, and $|X \cup Y| \geq 3c/4$.

3. The probability not to hit S_j with one random element is $(1 - k/n)$. Due to independence, the probability to miss S_j all r times is $(1 - k/n)^r \approx \exp(-kr/n)$. We set $\exp(-kr/n) = \epsilon$. Simple algebra gives $r = \ln(1/\epsilon) \cdot n/k$. (This simplification is only a proposal. The result can also be expressed more precisely.)
Since we cannot know how many sets S_j have large intersections with S, we assume the worst case m. By the union bound, the failure probability is now at most $m \exp(-kr/n) = \epsilon$. The same calculation as before yields $r = \ln(m/\epsilon) \cdot n/k$.

4. Let X denote the number of votes for A. By linearity of expectation we have $\mu := E[X] = k(1-p) + (n-k)p < n/2$. The wrong candidate wins if $X > n/2$. So the question is basically: How likely is a large deviation from the expected value? This immediately suggests the use of Chernoff bounds. In fact, they are applicable, since X is a sum of independent $0, 1$-valued

random variables. (So, yes, independence is essential here). The Chernoff bound decreases exponentially in n.
In more technical detail (not expected):
Using $a = k/n$ we can also write $\mu = (a(1-p) + (1-a)p) \cdot n$. Our δ is $n/(2\mu) - 1$. The Chernoff bound from class has the form b^μ, where the base $b < 1$ only depends on δ which, in turn, only depends on a and p. Thus, for fixed a and p, the base is constant, and μ is linear in n. Hence the failure probability decreases exponentially in n.

5. The scheme is very similar to the "fast" FPT algorithm for Vertex Cover, only some problem-specific details are different. In one branch we put c (thus, 1 element) in the solution. If we don't, we must put one of a, b and one of d, e in the solution. These are 4 options, each selecting 2 elements. This yields the claimed recurrence. If no branching is possible, we are in some polynomial-time case (as claimed in the exercise). For solving the recurrence we set $T(k) = x^k$ and divide the equation by x^{k-2}. The quadratic equation $x^2 = x + 4$ yields $x < 2.57$, thus a time bound of $O^*(2.57^k)$.

6. A greedy approach: Make the tree directed, by declaring an arbitrary node the root. Now consider an optimal solution I, Let v be any leaf. If neither v nor its siblings are in I, we go upwards until we meet the first node $u \in I$. Replacing u with v in I does not increase $|I|$. Also the distance from v to other nodes in I remains larger than 2. Thus it is safe to put v in I. This reasoning shows the correctness of the following algorithm: Put a leaf in I, remove this leaf, all its siblings, its parent and grandparent. Iterate until the tree is empty.
(A dynamic programming alternative is not shown here.)