University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn
M. Ahmadi, P. Schneider

Algorithms Theory
Sample Solution Exercise Sheet 2

Due: Monday, 19th of November, 2018, 14:15 pm

Exercise 1: Majority and Frequent Elements *(4+6 Points)*

Let us assume that A is an arbitrary array of size n that contains n integers. An element in A is called a *majority element* if the number of occurrences of the element in A is strictly greater than $n/2$.

(a) Provide an algorithm that returns the majority element of A if there is one. The algorithm must have a time complexity of $O(n)$ and constant auxiliary space.

(b) Argue the correctness of your answer.

Note: Auxiliary space is the space the algorithm needs in addition to the input (i.e., array A) throughout the algorithm execution.

Sample Solution

(a) The algorithm considers two integer variables `count` and `majority`, where `count` is initially set to 0 and `majority` is initially set to the first element of the array. The algorithm consists of two phases.

In the first phase, the algorithm scans the whole array. Let a be the element currently being read. If a is the same element as `majority` then increment `count`, else, decrement `count`. As soon as `count` becomes 0, the current element a becomes the new `majority` and `count` is incremented.

In the second phase, the algorithm checks whether the content of variable `majority` at the end of the first phase is actually a majority element or not. To do so, it scans the whole array once again and counts the number of occurrences of the content of variable `majority` in A.

(b) Considering array $A[1..n]$, let us define the majority element of a subarray $A[i..j]$ of A as the element whose number of occurrences in $A[i..j]$ is more than $(j-i+1)/2$. Then, consider the following claim:

Claim 1. Let $A'[1..n']$ be an arbitrary array containing n' integers, and let x be its majority element. For an integer $i < n'$, if subarray $A'[1..i]$ of A has no majority element, then x is the majority element of $A'[i+1..n']$

Proof. We first show that $A'[i+1..n']$ has a majority element. Afterwards, we will show that it must be x. For the sake of contradiction, let us assume that $A'[i+1..n']$ has no majority element. Fix any element y in A'. Then, since $A'[1..i]$ or $A'[i+1..n']$ have no majority elements, the number of occurrences of y in $A'[1..i]$ is at most $i/2$ and in $A'[i+1..n']$ is at most $(n'-i)/2$. Therefore, the number of occurrences of y in A' is at most $i/2 + (n'-i)/2 = n'/2$. It implies that y cannot be a majority element in A', which contradicts our assumption. Hence, $A'[i+1..n']$ has a majority element.

It is not difficult to see that the number of occurrences of any element except x in $A'[1..i]$ and $A'[i+1..n']$ in total is at most $n'/2$. Hence, the majority element of $A'[i+1..n']$ must be x. ■

If during the execution, variable `count` never goes to 0, then it is trivial to see that variable `majority` contains the majority element of A. Otherwise, let j be the last index when the algorithm sets `count` to 0. If $j = n$, then A has no majority element. Otherwise, based on Claim 1, $A[j+1, n]$ must have a majority element, if A has a majority element. Therefore, due to the choice of j, and the fact that the content of variable `majority` is the majority element of $A[j+1, n]$, the only element that can be the majority element of A is the element in variable `majority` at the end. Hence, the second phase of the algorithm correctly determines whether there is a majority element or not.

Exercise 2: Covering Unit Intervals *(3+6 Points)*

We are given a set X of real numbers. The problem is to find the minimum cardinality set of unit intervals $S = \{[s_1, s_1 + 1], [s_2, s_2 + 1], \ldots, [s_k, s_k + 1]\}$, where $s_1, s_2, \ldots, s_k \in \mathbb{R}$, such that every real number in X belongs to at least one interval in S.

(a) Consider the following greedy algorithm $\mathcal{A}$, determine whether it solves the problem or not and explain why.

$\mathcal{A}$ proceeds in steps until X becomes empty. In the j^{th} step, $\mathcal{A}$ determines $s_j \in \mathbb{R}$ such that the unit interval $[s_j, s_j + 1]$ contains the maximum number of elements left in X. Then, to conclude the j^{th} step, $\mathcal{A}$ removes all the real numbers from X that are contained in $[s_j, s_j + 1]$.

(b) Provide an *efficient* greedy algorithm to solve the problem. Argue the correctness of your answer.

Sample Solution

(a) The given greedy algorithm does not solve the problem. Here is a counter example showing that the algorithm does not return an optimal solution. Consider a small constant, say $\epsilon = \frac{1}{100}$. Then, let $X := \{-\epsilon,\ 1-3\epsilon,\ 1-2\epsilon,\ 1-\epsilon,\ 1+\epsilon,\ 1+2\epsilon,\ 1+3\epsilon,\ 2+\epsilon\}$. The first chosen interval by the greedy algorithm must contain $\{1-3\epsilon,\ 1-2\epsilon,\ 1-\epsilon,\ 1+\epsilon,\ 1+2\epsilon,\ 1+3\epsilon\}$. Then, the algorithm needs to pick two extra intervals to cover the two remaining uncovered elements of X. However, an optimal solution is $\{[-\epsilon, 1-\epsilon], [1+\epsilon, 2+\epsilon]\}$, which has cardinality of 2.

(b) The algorithm first sorts the elements in X. Then, it proceeds in steps until X becomes empty. In each step, it picks the minimum element δ left in X. Then, it adds the unit interval $[\delta, \delta + 1]$ to the set of chosen intervals, and concludes the step by removing all the elements from X that are in interval $[\delta, \delta + 1]$.

Here, we present an exchange argument to show that the greedy solution is an optimal solution. Let $S = \alpha_1 < \alpha_2 < \ldots$ be the ordered sequence of the starting points of the intervals calculated by the above greedy algorithm on the arbitrary given input X. Moreover, let $S^* = \beta_1 < \beta_2 < \ldots$ be the ordered sequence of the starting points of the intervals in an arbitrary optimal solution of X. Let $i \leq y$ be the smallest integer such that $\alpha_i \neq \beta_i$. Due to the greedy choices, $\alpha_i \in X$ and $\alpha_i > \alpha_{i-1} + 1$. Therefore, since the optimal solution must cover all the elements of X, α_i must be covered by $[\beta_i, \beta_i + 1]$. Therefore, $\beta_i < \alpha_i$.

Note that all the elements in X that are smaller than α_i are already covered by the intervals of S^* with starting point less than α_i. Hence, one can modify S^* by replacing β_i with α_i. This change does not increase the number of intervals in S^*, and still all the numbers in X are covered with respect to S^*. One can apply this exchange argument finite number of times to stepwise change S^* to S without increasing the number of intervals or violating the solution's correctness.

Exercise 3: Scheduling *(4+7 Points)*

We are given n jobs $J_1 = (s_1, p_1), \ldots, J_n = (s_n, p_n)$, where s_i is the earliest start time and p_i is the processing time of job J_i. The goal is to schedule these jobs on a single processor, whereas each job

can be suspended and resumed again as many times as needed. For example a job $J_k = (4, 5)$ could be scheduled to be processed in intervals $[4, 6], [10, 12]$, and $[15, 16]$.

(a) Provide an efficient greedy algorithm to compute a scheduling of the n jobs on a single processor while minimizing parameter $C = \sum_{i=1}^{n} c_i$, where c_i is the time when job J_i is completed.

(b) Argue the correctness of your answer.

Sample Solution

(a) Consider the following algorithm.

Algorithm 1 `Schedule`$(\langle J_1, J_2, \ldots, J_n \rangle)$

Create a sequence S of all jobs sorted by starting time and a priority queue PQ
Let i be the minimum starting time of all jobs
while $S \neq \emptyset$ and $PQ \neq \emptyset$ **do**
  Remove all jobs with starting time i from S
  Insert them into PQ with a key of processing time
  $p \leftarrow PQ.min()$
  Let J be a job in PQ with key p
  Let t be the minimum starting time of all jobs in S
  Schedule job J from time i to $i' := \min\{t, p\}$
  Decrease the key of job J by $i' - i$
  Remove J from PQ if its key is 0
  $i \leftarrow i'$

(b) Here, we present an exchange argument to show that the above greedy algorithm provides an optimal solution. Let G be the solution calculated by the above greedy algorithm. Moreover, let S be an arbitrary optimal solution. We show that one can stepwise change S until it becomes identical to G. Furthermore we show that in each step, the solution after the change is at least as good as the solution before the change.

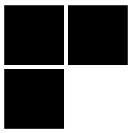
Let t be the largest real number such that G and S are identical until time t (always schedule the same jobs or are idle at the same time). Let job J_g be the job that is scheduled from time t to t_g in G, and job J_s be job that is scheduled from time t to t_s in S (S must schedule a job J_s at time t otherwise it would not be optimal). Let $t' = \min\{t_g, t_s\}$. We show that one can change S to S' such that S' is identical to G until time t' and still as good as S. Let c_g and c_s be the times job J_g and J_s are respectively completed in S. We consider two cases to calculate S'.

(1) $[c_s < c_g]$ Let p_g and p_s respectively be the remaining processing times of jobs J_g and J_s at time t. To calculate S' from S, consider all the time units that are assigned to jobs J_g or J_s. Then, we change S by assigning the first p_g of these time units to job J_g, and the rest to J_s. Due to the greedy choice, we have $p_g < p_s$. Therefore, the time at which J_g is completed in S' is at most c_s. Moreover, it is not difficult to see that the time at which J_s is completed in S' is c_g. Therefore, S' is at least as good as S.

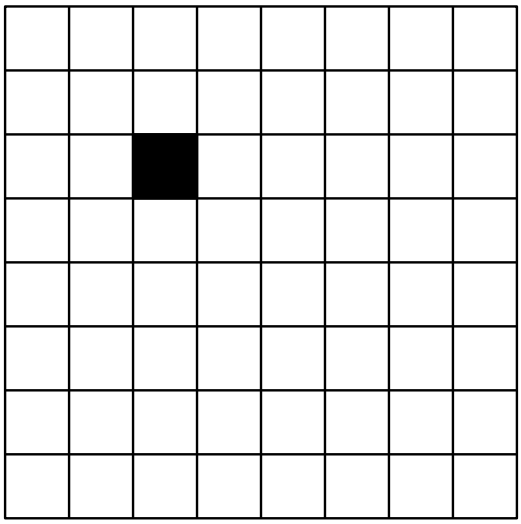
(2) $[c_g < c_s]$ Let p_g be the remaining processing time of job J_g at time t. Due to the fact that J_g is scheduled in G from t to t', $p_g \geq t' - t$. To construct S', we apply the following two steps to transform S to S'. First, assign the last $t' - t$ units that are scheduled for J_g in S to J_s. Then, assign the time units in $[t, t']$ to J_g. Since $c_g < c_s$, the times at which J_g and J_s are completed do not change. Therefore, S' is at least as good as S.

Exercise 4: Divide and Conquer *(2+6+2 Points)*

Consider a $n \times n$ *square* grid with $n = 2^k$ for a $k \in \mathbb{N}_{\geq 1}$. We have an unlimited supply of a specifically shaped tile, which covers exactly 3 cells of the grid as follows:



The goal is to cover the whole grid with these tiles (which can also be turned by 90, 180 and 270 degrees). We call an arrangement of tiles on grid cells a *valid tiling*, if all cells of the $n \times n$ grid can be covered with the tile above *without any overlaps* of tiles and *without going over the edges* of the grid. Assume that the input grid has an arbitrary *single* cell that is initially tiled (before the start of the algorithm). E.g. for $n = 8$ the input grid may look like this:



(a) Is there a *valid tiling* for every $2^k \times 2^k$ grid ($k \in \mathbb{N}_{\geq 1}$) that is initially completely empty? Prove or disprove.

(b) Describe a *divide and conquer* algorithm that computes a *valid tiling* on a $n \times n$ grid in $O(n^2)$ (with $n = 2^k, k \in \mathbb{N}_{\geq 1}$) that has one cell that is initially tiled. Assume that placing a tile is in $O(1)$.

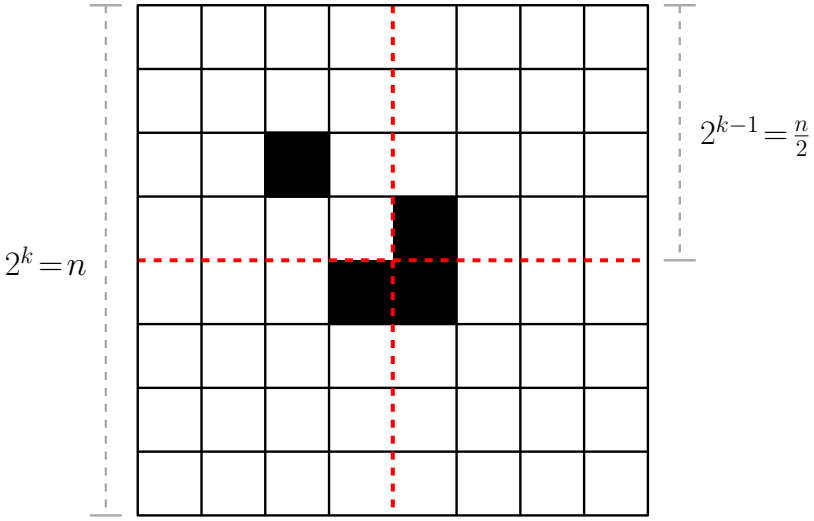
(c) Show the running time of $O(n^2)$.

Sample Solution

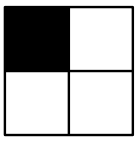
(a) No there is not. Assume there were such a valid tiling using t copies of the above tile. Then $3t = 2^{2k}$, i.e., 3 divides a power of 2, a contradiction since 2 and 3 are both prime.

(b) Our strategy is to divide a $2^k \times 2^k$ grid into four $2^{k-1} \times 2^{k-1}$ sub grids and maintain the invariant that one cell will always be tiled (using the knowledge that a valid tiling is impossible on a completely empty grid). This invariant immediately gives us a valid tiling for the base case of a 2×2 grid.

Divide: We have a $2^k \times 2^k$ grid with $k > 1$ and one cell already tiled. We divide it into four $2^{k-1} \times 2^{k-1}$ sub-grids.We make sure that each sub grid has one cell that is already tiled. We do this by placing the tile such that it covers exactly one cell of each sub-grid that does not have a tiled cell yet, as exemplified by the following figure.



Base Case: We have a 2×2 grid with one cell already tiled which looks like the example below (except for the orientation of the tiled cell). This can obviously be covered using one of our specifically shaped tiles and we obtain a valid tiling.



Conquer: By producing valid tilings for all 2×2 sub grids we obtain a valid tiling for the whole grid.

(c) The run time is $T(n) = 4T(n/2) + \mathrm{O}(1)$ which solves to $\mathrm{O}(n^2)$ using the Master Theorem.