Algorithm Theory, Tutorial 2

Johannes Kalmbach

University of Freiburg

johannes.kalmbach@gmail.com

22. November 2018

General Hints

- This is a theory lecture! Do formal proofs!
- Much reasoning and text about correctness but a lot of this not mathematically “bulletproof”
- Recap: Proof structures:
    - Mathematical induction (Recursion: Divide and Conquer, Dynamic Programming)
    - Proof by contradiction
    - Exchange Arguments (Greedy Algorithms!!)

Algorithm Writeups

- Pseudocode, limit to important aspects
- Much better than last week.
- Pictures can also help, especially with geometric structures.

Majority Element

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.

Solution (live)

```
fun maj(A):
    count = 0
    cand = None
    for el in A:
        if count == 0:
            count = 1
            cand = el
        elif cand == el:
            count += 1
        else:
            count -= 1
    return cand
```

?

verify whether
cand is majority el ?

Solution (Master Text)

(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.

Proof

- In the following assume that majority element m exists.

Claim

- When our counter hits 0 at position i then m is also majority element in the remainder $A[i+1:]$ of the array.
- This implies that we can simply restart the algorithm at position $i+1$ without losing information.

Proof, Induction over the number of times the counter has hit 0.
Base case, n=1 (first hit to 0)

- The counter is set to 0 right at the beginning of the algorithm.
- The remainder of the array is the complete array and m is definitely majority element there. (Trivial)

Inductional Step, $n \mapsto n+1$

- Let i_n be the position where the counter hit 0 for the n-th time ✓
- Let k be the number of elements we looked at since the last reset and r the number of elements we still have to look at. ✓
- Let num_m be the number of occurences of m in $A[i_n + 1 :]$ after last (n-th) reset
- m is majority element in $A[i_n + 1 :]$ (Induction hypothesis)$\Rightarrow \boxed{num_m > \frac{k+r}{2}}$
- Since the last reset of the counter we have seen $k/2$ elements equal to our previous candidate c and $k/2$ elements $! = c \Rightarrow$ we have seen $\leq k/2$ occurences of m since the last reset.
- m must be majority element in $A[i_{n+1} :]$ after n+1th reset (remainder of the array) because otherwise $num_m \leq k/2 + r/2 = (k+r)/2$ which would violate our Induction hypothesis.
  reset counter
  m not maj in remainder
- This proofs our claim.

Finishing the proof

- From Claim 1 follows:
- If our counter is 0 at the end of the $\overset{\text{algorithm}}{\text{first}}$ pass, then there exists no majority element
- If our counter is positive at the end with a candidate c then c is a majority element in the subarray since the last reset of the counter (Trivial)
- With our claim follows: All array elements $x \neq c$ are definitely no majority element of the array
- Thus our algorithm correctly identifies a candidate for the majority element.

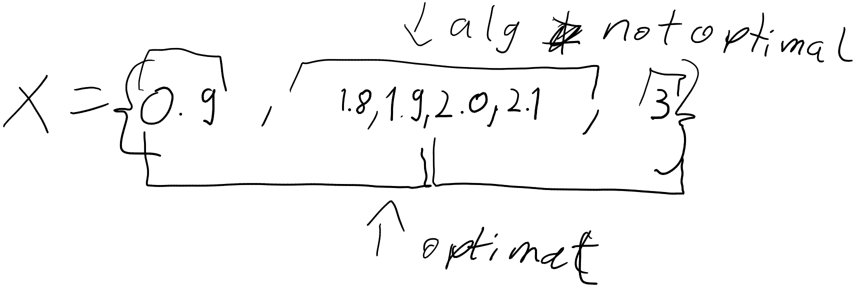
If during the execution, variable `count` never goes to 0, then it is trivial to see that variable `majority` contains the majority element of A. Otherwise, let j be the last index when the algorithm sets `count` to 0. If $j = n$, then A has no majority element. Otherwise, based on Claim 1, $A[j+1, n]$ 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.

Covering Unit Intervals

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.

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 $[s_j, s_j + 1]$.

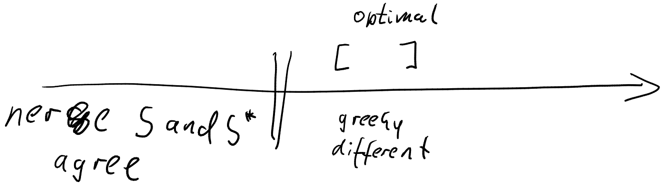


Provide an *efficient* greedy algorithm to solve the problem. Argue the correctness of your answer.

Algorithm 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 $[\delta, \delta + 1]$.

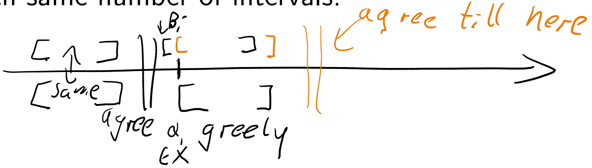
Proof of Correctness

- exchange argument ✓ greedy
- Let $S = \alpha_1 < \alpha_2 < \dots$ 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 $S^* = \beta_1 < \beta_2 < \dots$ ✓ optimal be the ordered sequence of the starting points of the intervals in an arbitrary optimal solution of X.
- Let i be the smallest integer such that $\alpha_i \neq \beta_i$.



Proof of Correctness

- Due to the greedy choices, $\alpha_i \in \overset{\surd}{X}$ and $\alpha_i > \alpha_{i-1} + 1$.
- Since the optimal solution must cover all the elements of X, α_i must be covered by $[\beta_i, \beta_i + 1]$.
- Therefore, $\beta_i < \alpha_i$. ✓
- 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.
- Replace β_i with α_i in optimal solution
- Now solution is still valid and agrees with greedy in one more interval.
- Apply this until optimal solution is transformed to greedy solution with same number of intervals.



Scheduling

- 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 $[15, 16]$.
- 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.

Shortest possible solution (online)

- As soon as the current job is finished or a new job becomes ready:
    - run the job with the minimum remaining time among the jobs that currently are ready

More technical solution

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

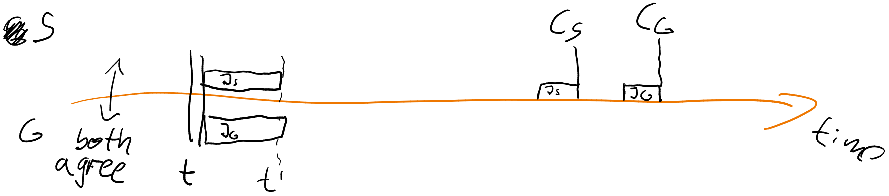
```
Create a sequence $S$ of all jobs sorted by starting time and a priority queue $PQ$
Let $i$ be the minimum starting time of all jobs
while $S \neq \emptyset$ do
    Remove all jobs with starting time $i$ from $S$
    Insert them into PQ with a key of processing time
    $p \leftarrow PQ.min()$
    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
    $i \leftarrow i'$
```

Proof of Optimality

- Exchange argument
- Let G be the solution calculated by the greedy algorithm
- Let S be an arbitrary optimal solution.
- We show that one can stepwise change S until it becomes identical to G.
- After each step the solution is at least as good as before the step

- 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 $t' = \min\{t_g, t_s\}$.
- Let c_g and c_s be the times job J_g and J_s are respectively completed in S.

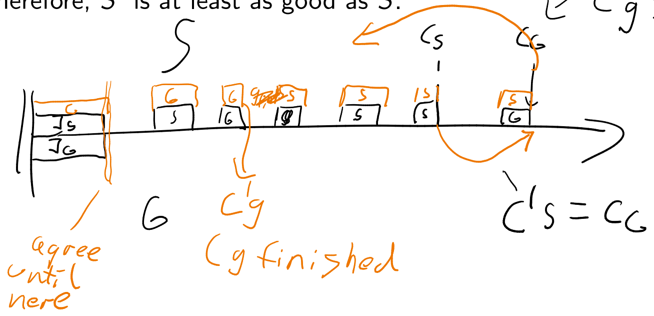
We consider two cases to calculate S'.



(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 $p_g \leq p_s$.
- Therefore, the time at which J_g is completed in S' is at most c_s.
- The time at which J_s is completed in S' is c_g.
- Therefore, S' is at least as good as S.

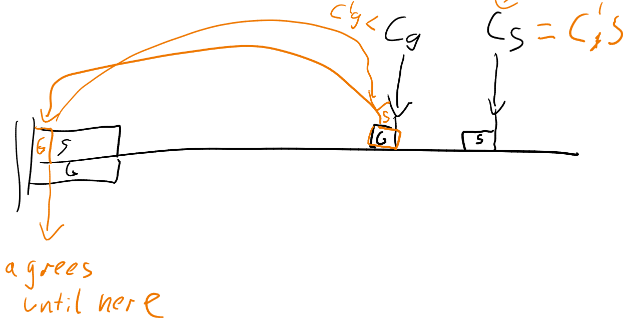
✓ $C'_g \leq C_S$



(2) $[c_g < c_s]$

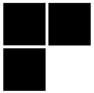
- Let p_g be the remaining processing time of job J_g at time t.
- Due to the fact that J_g is scheduled in G from t to t', $p_g \geq t' - t$.
- First, assign the last $t' - t$ units that are scheduled for J_g in S to J_s.
- Then, assign the time units in $[t, t']$ to J_g.
- Since $c_g < c_s$, the times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

becomes better

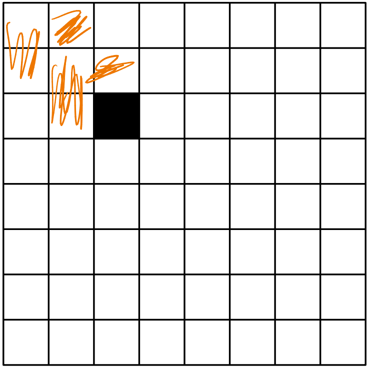


Valid Tiling

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:



4a

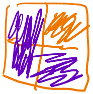
Is there a *valid tiling* for every $2^k \times 2^k$ grid ($k \in \mathbb{N}_{\geq 1}$) that is initially completely empty? Prove or disprove.
No there is not. Assume there were such a valid tiling using t copies of the above tile. Then $3t = 2^{2k}$, i.e., 3 divides a power of 2, a contradiction since 2 and 3 are both prime.

4 is not dividable by 3

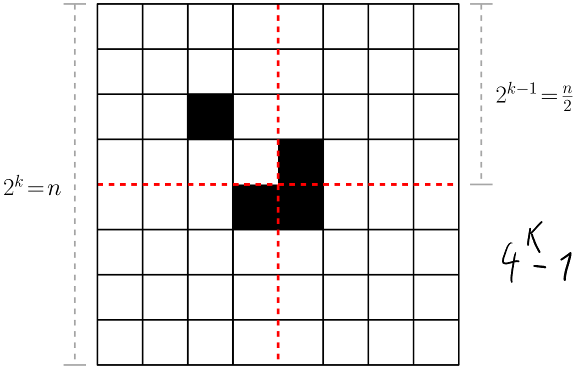
$2^K \cdot 2^K = 2^{2K}$ Never dividable by 3

Divide and Conquer in $O(n^2)$

- Divide a $2^k \times 2^k$ grid into four $2^{k-1} \times 2^{k-1}$ sub grids.
- Maintain the invariant that one cell will always be tiled (using the knowledge that a valid tiling is impossible on a completely empty grid).
- Base Case: If our grid is 2×2 cover it directly


+ rotations

- Divide: Split into four $2^{k-1} \times 2^{k-1}$ sub grids and place a tile s.t. each sub grid has exactly one covered cell.
- Maintains our invariant.



$4^K - 1 \bmod 3 = 0$

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

Runtime

- $T(n) \sim$ time to cover a $n \times n$ grid
- Placing one tile in the middle is $O(1)$
- $T(n) = 4 \cdot T\left(\frac{n}{2}\right) + \Theta(1)$
- Master Theorem gives us $\Theta(n^2)$