University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn
Marc Fuchs

Algorithm Theory
Sample Solution Exercise Sheet 5

Due: Friday, 21st of November 2025, 10:00 am

Exercise 1: The Fair Load Balancing Problem (8 Points)

You are a project manager leading a team of two developers, Alice and Bob. You have a list of n tasks that must be completed. Each task i has an estimated effort/workload $w_i > 0$ (a positive integer representing, for example, hours). You must assign every task to exactly one of the two developers. Your goal is to distribute the tasks as fairly as possible, so the total workload between Alice and Bob is as balanced as possible. More precise, let W_A be the overall workload of Alice and W_B be the overall workload ob Bob, we want $|W_A - W_B|$ to be as small as possible.

Your algorithm should be based on **dynamic programming** and run in time $O(n \cdot W)$, where n is the number of tasks and W is the overall workload, i.e., $W := \sum_{i=1}^{n} w_i = W_A + W_B$.

Hint: Let $w_1, w_2, \ldots, w_n$ be a fixed ordering of the tasks, then you may want to compute $OPT(i, x)$ (for all possible input values) that is a function evaluating to `true` iff one can reach workload exactly $0 \le x \le W$ by only considering the first i tasks $w_1, w_2, \ldots, w_i$.

Sample Solution

We will first solve the problem described in the hint. Let first consider some edge cases. We say $OPT(0, 0) = 1$, as even without elements we can reach a total sum of 0. However, for all $x > 0$, we say $OPT(0, x) = 0$. Also, for all $i > 0$ we can say that at total sum of 0 is reachable, by simply not taking any of the i items into account, i.e., $OPT(i, 0) = 1$. Not lets see when $OPT(i, x) = 1$ in general. First, it is not hard to see that if $OPT(i - 1, x) = 1$, then $OPT(i, x)$ as well, as we do not need to use task i. Second, if $OPT(i, x - w_i) = 1$ we also clearly have $OPT(i, x) = 1$ by considering task i in addition to the solution that achieved a weight of $x - w_i$. We can now summarize this observation:

$$OPT(i, x) = \begin{cases} 1 & \textit{if } x = 0 \\ 0 & \textit{if } i = 0 \textit{ and } x > 0 \\ 1 & \textit{if } OPT(i - 1, x) = 1 \\ 1 & \textit{if } OPT(i, x - w_i) = 1 \\ 0 & \textit{else} \end{cases}$$

Similarly to the Knapsack problem from the lecture, we can solve this problem efficiently by creating a 2D-matrix with $(n + 1) \times (W + 1)$ entries. Note that every entry only depends on the previous entries in the same row and column and thus we can compute $OPT(i, x)$ in constant time assuming that $OPT(i - 1, x)$ and $OPT(i, x - w_i)$ is already computed. This implies that we can fill the whole matrix in time $O(n \cdot W)$ (and hence compute all possible OPT values).

We will now solve the load balancing problem assuming that we have already computed all the OPT values. First note that if there exist a perfectly balanced solution i.e., we can split the jobs in two parts with $W_A = W_B = W/2$, the value of $OPT(n, W/2)$ must be 1. In that case we could directly stop. However, in general we can just do the following: We iterate through all possible values for W_A,

that are all integer values in $[0, W]$. For each of these values we check if $OPT(n, W_A) = 1$. If so, we know that there is an assignment of jobs for Alice with workload W_A. We then compute the respective workload of Bob in constant time: $W_B = W - W_A$. Then we compute the difference $d := |W_A - W_B|$ and store this d value. After we iterated over all possible W_A, we just take the difference d that was the smallest among all those we computed. The W_A and W_B that belong to this smallest d, define the optimal balanced workloads for Alice and Bob. Note that this whole iteration takes time $O(W)$, as we bruteforce all possible assignments (with n jobs). Thus, the overall runtime is dominated by filling out all entries of the matrix.

Remark: There is an alternative - easier - solution to this problem if we use the Knapsack Algorithm from the lecture in black-box fashion. We leave the details of how to set the parameters of the Knapsack problem (i.e., weights, values and capacity) and how to do the post-processing as an exercise to the reader.

Exercise 2: Amortized Analysis (12 Points)

Your plan to implement a `Stack` with the classical operations `push`, `pop` and `peek`. As underlying data structure you use a dynamic array that will grow its size whenever 'many' elements are stored and on the other hand also shrinks its size when only a view elements remain in the array. In the following let n_i be the number of elements stored in the array and let s_i be the size of the array after the i-th operation.

- Before you **push** a new element x to the array, you check if $n_{i-1} + 1 < 80\% \cdot s_{i-1}$. If this is the case then you simply add x. We say for simplicity, that this can be done in 1 time unit. If on the other hand $n_{i-1} + 1 \geq 80\% \cdot s_{i-1}$, you set up a new (empty) array of size $s_i := 2s_{i-1}$ and copy all elements (and x) into the new one. We assume this can be done in s_{i-1} time units[1].

- To **pop** an element from the array, you first check if $n_{i-1} - 1 > 20\% \cdot s_{i-1}$. If this is the case then pop x within 1 time unit. If the table size is small, say $s_{i-1} \leq 8$, you also just pop x. But, if $n_{i-1} - 1 \leq 20\% \cdot s_{i-1}$ and $s_{i-1} > 8$, create a new (empty) array of size $s_i := s_{i-1}/2$ and copy all values except x into this new array. By assumption, this step takes s_i time units.

- The **peek** operation returns the last inserted element in 1 time unit. Note that state of the array does not change, i.e., $n_{i-1} = n_i$ and $s_{i-1} = s_i$.

Initially, the array is of size $s_0 = 8$. Assume that this initial step can also be done in 1 time unit. Note that by this initial size and the definition of the pop method we have $s_i \geq 8$ for all $i \geq 0$. Also note that after every operation that resized the array at least one element can be pushed or popped until a further resize is required.

(a) Let i be a push operation that resized the array. Show that the following holds. *(2 Points)*

$$0.4 \cdot s_i \leq n_i < 0.55 \cdot s_i$$

Further, show that if i is a pop operation that resized the array, the following holds. *(2 Points)*

$$0.25 \cdot s_i < n_i \leq 0.4 \cdot s_i$$

(b) Use the `Accounting Method` from the lecture to show that the **amortized running times** of push, pop and peek are $O(1)$, i.e., state by how much you additionally charge these three operation and show that the costs you spare on 'the bank' are enough to pay for the costly operations. *(4 Points)*
Hint: Use the previous subtask, even if you didn't manage to show them.

---

[1] For a simpler calculation we use normalized time units, such that all the operations that would take $O(1)$ time will take at most 1 time unit and operations that would take $O(s_{i-1})$ time will take at most s_i time units.

(c) Show the same statement as in the previous task, but use the `Potential Function Method` this time, i.e., find a potential function $\phi(n_i, s_i)$ and show that this function is sufficient to achieve constant amortized time for the supported operations. *(4 Points)*

Hint: There is not just one but infinitely many potential functions that work here. However, you may want to use a function of the form $c_0 \cdot |n_i - c_1 \cdot s_i|$ for some properly chosen constants $c_0 > 0$ and $c_1 > 0$.

Sample Solution

(a) **Push:** It is clear that for the previous state $n_{i-1} < 0.8 s_{i-1}$ is true (otherwise there would have been a resize before) and by definition also $n_{i-1} + 1 \geq 0.8 s_{i-1}$ holds. Since $n_i = n_{i-1} + 1$ and $s_i = 2s_{i-1}$ we directly get the following:

- $0.4 \cdot s_i = 0.8 \cdot s_{i-1} \leq n_{i-1} + 1 = n_i$ and
- $n_i = n_{i-1} + 1 < 0.8 s_{i-1} + 1 = 0.4 s_i + 1 \leq 0.4 s_i + s_i/8 < 0.55 \cdot s_i$

Note that in the second line we used that in all cases $s_i \geq 8$ and hence $1 \leq s_i/8$. These two lines imply the statement.

Pop: For similar reasons as before, we have $0.2 \cdot s_{i-1} < n_{i-1} \leq 0.2 \cdot s_{i-1} + 1$. Substituting $n_i = n_{i-1} - 1$ and $s_i = s_{i-1}/2$ we get $0.4 \cdot s_i < n_i + 1 \leq 0.4 \cdot s_i + 1$. By subtracting all sides by 1 and use that $1 \leq s_i/8$ we get $0.275 \cdot s_i < n_i \leq 0.4 \cdot s_i$. This implies the statement.

(b) We charge all 3 operations by 25 'dollars'. Every push, pop or peek operation costs one actual dollar (not counting resizing) and puts the remaining 24 in the bank to pay for resizing. Now, let us estimate how much money is **at least** in the bank before the next resizing takes place. Observe that we can ignore peek operations in the following, since they just increase our bank account (by 24 per operation) and do not change the state of the array. Let us for now assume that the last operation (say operation i) did a resizing and currently there is no money at our account. If this last resize operation was 'push', then we have $0.4 \cdot s_i \leq n_i < 0.55 \cdot s_i$. Thus, the next costly operation can not happen before $(0.8 - 0.55)s_i = 0.25 s_i$ push or $(0.4 - 0.2) \cdot s_i = 0.2 s_i$ pop operations. Before we proceed, let us see how it looks if operation i was a pop operation. By the statement of the previous task we have the next costly operation not before $(0.8 - 0.4) \cdot s_i = 0.4 s_i$ push or $(0.25 - 0.2) \cdot s_i = 0.05 \cdot s_i$ pop operations. By this analysis, the *worst-case* (i.e., the shortest chain of operation until the next resize) is a costly pop operation that is followed by more than $0.05 \cdot s_i$ additional deletions. Since we charge both operation with the same amortized cost, we clearly have more 'money' on our account in the other cases. Also observe that if we alternate between pushing and deleting over and over, the hash table is never resized, so we save up a lot of money in the bank. For that, assume operation i is a pop operation and from here on at least $0.05 \cdot s_i$ further pop operations follow until the next resize comes. The money on the bank after these many operations is at least $(25 - 1) \cdot 0.05 \cdot s_i = 1.2 \cdot s_i$. Because the costly operation costs $1 \cdot s_i$, we can afford it and thus, the bank account never drops below zero. Therefore, the amortized cost of pop is $O(1)$. This, by above's analysis, also implies amortized costs of $O(1)$ for push (as well as for peek).

(c) We define our potential function by $\phi(n_i, s_i) := c_0 \cdot |n_i - c_1 \cdot s_i|$ and start by guessing some constant $c_1 \in [0, 1]$. For some intuition: We want our potential to be large before a resizing operation and small (close to zero) after a resizing operation. To make sure that the potential before resizing is not 0, we have to choose $c_1 \neq 0.8$ in the push case and $c_1 \neq 0.2$ in the pop case. So let us 'guess' $c_1 = 0.4$ and show later that it works. Now we go through all operations and proof that by choosing a large enough $c_0 > 0$, all amortized costs are in $O(1)$. Note that we are going into 5 cases now, that we call `peek`, `cheap push` (push without resizing), `costly push` (includes resizing), `cheap pop` (deleting without resizing) and `costly pop` (includes resizing). Like in the lecture, we notate the actual cost of operation i by t_i and its amortized costs by $a_i := t_i + \phi(n_i, s_i) - \phi(n_{i-1}, s_{i-1})$.

Peek   Here we have $s_i = s_{i-1}$ and $n_i = n_{i-1}$ and thus,

$$
\begin{aligned}
a_i &= 1 + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_{i-1} - 0.4s_{i-1}| \\
&= 1 + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_i - 0.4s_i| \\
&= 1
\end{aligned}
$$

Cheap Push   Here we have $n_i = 1 + n_{i-1}$ and $s_i = s_{i-1}$

$$
\begin{aligned}
a_i &= 1 + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_{i-1} - 0.4s_{i-1}| \\
&= 1 + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_i - 1 - 0.4s_i| \\
&\leq 1 + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot (|n_i - 0.4s_i| - 1) \\
&\leq 1 + c_0
\end{aligned}
$$

Note that this implies that $a_i \in O(1)$ if c_0 is a fixed constant.

Costly Push   Here we have $t_i = s_{i-1}$, $s_i = 2s_{i-1}$, $n_i = n_{i-1} + 1$ and by the first subtask $0.55s_i > n_i \geq 0.4s_i$

$$
\begin{aligned}
a_i &= s_{i-1} + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_{i-1} - 0.4s_{i-1}| \\
&= \frac{s_i}{2} + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot (|n_i - 0.2s_i| - 1) \\
&\leq \frac{s_i}{2} + c_0 \cdot (n_i - 0.4s_i) - c_0 \cdot (n_i - 0.2s_i) + c_0 \\
&\leq \frac{s_i}{2} - \frac{1}{5} \cdot c_0 \cdot s_i + c_0 \\
&\leq c_0
\end{aligned}
$$

That the last step only follows if we choose $c_0 \geq \frac{5}{2}$.

Cheap Pop   Here we have $n_i = -1 + n_{i-1}$ and $s_i = s_{i-1}$

$$
\begin{aligned}
a_i &= 1 + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_{i-1} - 0.4s_{i-1}| \\
&= 1 + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_i + 1 - 0.4s_i| \\
&\leq 1 + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_i - 0.4s_i| + c_0 \\
&\leq 1 + c_0
\end{aligned}
$$

Costly Pop   Here we have $t_i = s_{i-1}$, $s_i = s_{i-1}/2$, $n_i = n_{i-1} - 1$ and by the first subtask $n_i > 0.25s_i$ and $n_i \leq 0.4s_i$.

$$
\begin{aligned}
a_i &= s_{i-1} + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_{i-1} - 0.4s_{i-1}| \\
&= 2s_i + c_0 \cdot |n_i - 0.4s_i| - c_0 \cdot |n_i + 1 - 0.8s_i| \\
&\leq 2s_i + c_0 \cdot (0.4s_i - n_i) - c_0 \cdot (0.8s_i - n_i) + c_0 \\
&= 2s_i - \frac{2}{5} \cdot c_0 \cdot s_i + c_0 \\
&\leq c_0
\end{aligned}
$$

The last step only follows if we choose $c_0 \geq 5$.

Final Statement   It is clear by the previous calculations that if we choose $c_0 := 5$ the amortized costs for all 3 operations are at most $a_i \leq 1 + c_0 = 6$ and therefore constant. The potential function used is

$$\phi(n_i, s_i) := 5 \cdot \left| n_i - \frac{2}{5} \cdot s_i \right| = |2 \cdot s_i - 5 \cdot n_i| \geq 0$$

Remark: We have that $\phi(n_0, s_0) = 5 \cdot |0 - \frac{2}{5} \cdot 8| = 16$ and hence $\sum_i t_i \leq 16 + \sum_i a_i$. This does not completely match with the definition of amortized costs, however we can fix this problem by choosing the potential function $\phi'(n_i, s_i) := 5 \cdot |n_i - \frac{2}{5} \cdot s_i - \frac{16}{5}|$. Here we have $\phi'(n_0, s_0) = 0$. Further we have that for all i, $|\phi'(n_i, s_i) - \phi(n_i, s_i)| \leq 16$ holds, and thus we can simply adjust the 5 cases and show that for each operation it follows $a_i \leq t_i + \phi(n_i, s_i) - \phi(n_{i-1}, s_{i-1}) + 2 \cdot 16 \leq 6 + 32 = 38$.