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

Algorithm Theory
Sample Solution Exercise Sheet 4

Due: Tuesday, 16th of November, 2021, 4 pm

Exercise 1: Bagging Marbles *(8 Points)*

We have n marbles and an array $A[1..n]$, where entry $A[i]$ equals the price of a bag with *exactly* i marbles. We want to distribute the n marbles into bags such that the profit, which is the total value of all bags (with at least one marble), is maximized.

(a) Give an efficient algorithm that uses the principle of dynamic programming to compute the maximum profit one can make. *(4 Points)*

(b) Argue why your algorithm is correct. Give a tight (asymptotic) upper bound for the running time of your algorithm and prove that it is an upper bound for your solution. *(4 Points)*

Sample Solution

(a) **Idea:** (and argument of correctness for part (b)) Let $p(n)$ be the maximum profit we can achieve when we have n marbles left for bagging. Obviously, we have n choices how many marbles (say i) to put into the next bag. We choose the number i which optimizes the profit for the current bag plus the maximum profit we can achieve with the remaining marbles. We obtain the following recursion:

$$p(n) = \max_{i\in[1..n]} \big(A[i] + p(n-i)\big), \quad p(0) = 0$$

Based on that, we propose the following algorithm:

Algorithm 1 `profit(`n`)` ▷ assume we have a global dictionary `memo` initialized with `Null`

```
if n = 0 then return 0                                   ▷ base case
if memo[n] ≠ Null then return memo[n]                    ▷ profit was computed before
memo[n] ← max_{i∈[1..n]}(A[i] + profit(n−i))             ▷ Memoization
return memo[n]
```

(b) Due to the memoization we compute each value `profit(`n`)` for $i \in [1..n]$ at most once. Each computation of $p(i)$ costs at most $\mathcal{O}(n)$ in the current step (determining the maximum of at most n numbers) not counting the cost of recursive calls. The total cost is therefore $\mathcal{O}(n^2)$.

Remark: The upper bound for this algorithm is tight because we must compute each value `profit(`i`)` *for* $i \in [1..n]$ *with a cost of* $\Omega(i)$ *in the current recursion.*

Exercise 2: Parenthesization *(12 Points)*

Consider a string B of $n \geq 3$ symbols over $\{0, 1, \wedge, \vee, \oplus\}$ which correspond to boolean true and false values and the logical operators and, or, xor, where B starts and ends with 0 or 1, and a 0 or 1 is always followed by one of the operators $\wedge, \vee, \oplus$ (unless it is the last symbol).

The goal is to count the number of possibilities you can put substrings of B *reasonably* into brackets, such that it evaluates to true (i.e., 1). Roughly speaking, by reasonable we mean that it is clear in which order to evaluate the operators of B and there are no unnecessary brackets.

Formally we define a *reasonable* parenthesization of such a string B (that does not have any brackets yet) recursively as follows. In the base case, if B has just one operator then there is no need for brackets, i.e., the only parenthesization of B that is reasonable is if there are *no* parenthesis.

If B has more than two operators we pick an operator o in B and define the substring either to o's left or to its right as B'. If $|B'| \geq 3$, we put the substring B' into brackets and recursively pick a reasonable parenthesization of B'. Then we pick a reasonable parenthesization of B where the substring (B') is replaced with a single symbol x (which we now consider as a boolean value 0,1). All parenthesizations of B that can be obtained this way are reasonable.[1]

(a) Let $B = 0 \oplus 1 \vee 0 \wedge 0 \vee 1$. Give all reasonable parenthesizations of B (as defined above) so that the resulting expression evaluates to true (it is sufficient to give their total number if you feel confident that it is correct). *(2 Points)*

(b) Give an efficient algorithm that uses the principle of dynamic programming to compute the *number* of reasonable parenthesizations of B that evaluate B to true. *(6 Points)*

(c) Argue why your algorithm is correct. Give a tight (asymptotic) upper bound for the running time of your algorithm and prove that it is an upper bound for your solution. *(4 Points)*

Sample Solution

(a) We realize that no matter which operation we evaluate last, the result is always true. The reasonable parenthesizations of B that evaluate B to true are all of them:

- $0 \oplus (1 \vee (0 \wedge (0 \vee 1)))$
- $0 \oplus (1 \vee ((0 \wedge 0) \vee 1))$
- $0 \oplus ((1 \vee (0 \wedge 0)) \vee 1)$
- $0 \oplus (((1 \vee 0) \wedge 0) \vee 1)$
- $0 \oplus ((1 \vee 0) \wedge (0 \vee 1))$
- $(0 \oplus 1) \vee (0 \wedge (0 \vee 1))$
- $(0 \oplus 1) \vee ((0 \wedge 0) \vee 1)$
- $(0 \oplus (1 \vee 0)) \wedge (0 \vee 1)$
- $((0 \oplus 1) \vee 0) \wedge (0 \vee 1)$
- $(0 \oplus (1 \vee (0 \wedge 0))) \vee 1$
- $(0 \oplus ((1 \vee 0) \wedge 0)) \vee 1$
- $((0 \oplus 1) \vee (0 \wedge 0)) \vee 1$
- $((0 \oplus (1 \vee 0)) \wedge 0) \vee 1$
- $(((0 \oplus 1) \vee 0) \wedge 0) \vee 1$

Remark: To count the reasonable parenthesizations, we can use the recursive definition. Let $P(k)$ be that number for a string B with k operators. We have $P(k) = \sum_{i=0}^{k-1} P(i) \cdot P(k-1-i)$, [2] *as we can split at each operator and then combine all possible parenthesizations to the left and right. In the base case we have $P(0) = 1$, as there is only a single reasonable parenthesization of a boolean value (no brackets). Recursively we get $P(1) = 1$, $P(2) = 2$, $P(3) = 5$, $P(4) = 14$.*

(b) **Idea:** The definition of reasonable parenthesizations gives a strong hint on how to define a recursive procedure to compute the number of those that evaluate to true. The goal is to split B at the operators and compute for all obtained substrings B' the corresponding number of parenthesizations that evaluate B' to true and false, respectively.

Given these results for all those substrings of B we can compute the same numbers for B, which then takes only linear time. The important observation is that, even though there are exponentially

[1] Alternatively, we obtain the reasonable parenthesizations by splitting string B (with $|B| > 3$, otherwise use the base case) at some operator o into the substrings B_1, B_2, left and right of o, respectively. Then put each substring B_1, B_2 with at least 3 symbols into brackets and recursively determine reasonable parenthesizations.

[2] Interestingly this corresponds to the k^{th} Catalan number, as someone pointed out in class.

many parenthezitaions of B, there is only a quadratic number of substrings and by memorizing the results for those we can avoid unnecessary re-computations.

The following function `count(`B`,`b`)` counts the number of reasonable parenthesizations of B that evaluate to b (where $b \in \{0,1\}$). Further, we assume that we are given an `eval(`B`)` function that evaluates a string B as defined above with at most one operator, i.e., that returns the boolean value B if $|B| = 1$, else if $|B| = 3$ it returns the result when we evaluate that operator with its two boolean arguments.

Algorithm 2 `count(`B`,`b`)` ▷ $b \in \{0,1\}$ *is a boolean value,* `memo` *is a global dictionary*

```
k ← number of operators in B
if k ≤ 1 then
    if eval(B) = b then return 1 else return 0
if memo[B, b] ≠ Null then return memo[B, b]          ▷ result was computed before
c ← 0
for each operator o in B do                          ▷ count parenthesizations left and right of o
    B1, B2 ← substrings of B left and right of o respectively
    for each pair (b1, b2) ∈ {0,1}^2 do              ▷ iterate over possible results for B1, B2
        if eval(b1 o b2) = b then                    ▷ b1 o b2 must evaluate to b in order to count
            c ← c + count(B1,b1) · count(B2,b2)      ▷ combined parenthesizations of B1, B2
memo[B, b] ← c                                       ▷ memoization
return memo[B, b]
```

(c) **Running time:** If we ignore the recursive sub-calls in `count(`B`,`b`)`, then the running time of `count(`B`,`b`)` is linear in the length of B, i.e., $\mathcal{O}(n)$, since we loop over each operator in B. Note that the inner loop has at most four iterations.

To account for the recursive subcalls, we count how many of them can occur before the result is already stored in the dictionary `memo`. The number of entries in `memo` corresponds to the number of cohesive substrings of B. Each cohesive substring of B has a dedicated start and end symbol in B and there are at most n^2 distinct choices of such start and end symbols.

Therefore `memo` can contain at most $\mathcal{O}(n^2)$ entries. Note that the second argument of `memo[`B`,`b`]` can only take two different values, which increases the size of `memo` only by a factor of 2.

This means $\mathcal{O}(n^2)$ calls of `count(`B`,`b`)` can occur before all results are precomputed (and can be looked up in $O(1)$ time). Since each call of `count(`B`,`b`)` costs $\mathcal{O}(n)$ (excluding recursive calls of which there can be only $\mathcal{O}(n^2)$ many), the total running time is $\mathcal{O}(n^3)$.

Remark: Our analysis so far might be considered an overexploitation of the REAL RAM model, which assumes that basic operations are in $\mathcal{O}(1)$ (which is reasonable as long as numbers do not get too large). In fact the numbers which we deal with here can be huge, as large as the n^{th} Catalan number (which we called $P(n)$ further above). This is particularly problematic when doing multiplications (as in the third last line).

However, we know that $P(n) \in \mathcal{O}(4^n)$ thus the numbers we deal with can be expressed with $\mathcal{O}(\log(4^n)) = \mathcal{O}(n)$ bits, i.e., we deal with numbers of length linear in the input size. We have seen earlier in the lecture that we can multiply those in $\mathcal{O}(n \log n)$ time. We can adjust our analysis accordingly by multiplying with a factor $\mathcal{O}(n \log n)$ for the (most costly) multiplication operation, so we obtain $\mathcal{O}(n^4 \log n)$ (which is still polynomial in n).

Correctness: For brevity, we call a reasonable parenthesization that evaluates B to b *satisfying* (with respect to b). We argue that our combinatoric calculations in `memo[`B`,`b`]` to count the number of satisfying parenthesizations, are correct. We make an inductive argument over the number of operators k of B. In the base case $k = 1$ we return 1 if `eval(`B`)` $= b$, since by our definition having no brackets is considered a reasonable parenthesization. Else, we return 0 as there is no way to add brackets to B that would facilitate `eval(`B`)` $= b$.

Let B be a string (as defined in the exercise) with $k \geq 2$ operators. Our hypothesis is that we count the number of satisfying parenthesizations for substrings of B with *less* than k operators

correctly. In the algorithm we split B into B_1, B_2 at an operator o and then independently count the satisfying parenthesizations when we (implicitly) put brackets around B_1 and B_2.[3] Then we sum up these counts up for each operator. Since this split follows the definition of reasonable parenthesizations, we will eventually count all satisfying parenthesizations, provided that we count them correctly for each such split.

We need to argue that we do not count the same parenthesization more than once in different splits. Let o and o' be two different operators of B with corresponding substrings B_1, B_2 and B'_1, B'_2 respectively. W.l.o.g., assume B_1 is a substring of B'_1. Assume that we have a parenthesization of B that puts a pair of brackets both around B'_1 (which has at least one operator) *and* and around B_2 (also has at least one operator) at the same time. Then the closing bracket of the former would precede the opening bracket of the latter, which is not a reasonable parenthesization.[4] Therefore the parenthesizations for each "split" at some operator must be distinct.

It remains to show that for a given split at operator o into substrings B_1, B_2 we do in fact count all satisfying parenthesizations (where no pair of brackets spans over o) correctly. Consider all pairs $(b_1, b_2) \in \{0,1\}^2$ such that $\texttt{eval}(b_1\, o\, b_2) = b$. We fix one such pair (b_1, b_2). By the induction hypothesis we obtain the correct numbers $\texttt{count}(B_1, b_1)$ and $\texttt{count}(B_2, b_2)$. Every satisfying parenthesization of B_1 (w.r.t. b_1) can be combined with any such of B_2 (w.r.t. b_2), thus multiplication is correct.

Finally, in the algorithm we repeat this for all $(b_1, b_2) \in \{0,1\}^2$ with $\texttt{eval}(b_1\, o\, b_2) = b$, and sum up the according counts. Here we do not have double counts for different pairs $(b_1, b_2) \in \{0,1\}^2$ since a parenthesization of B_1, B_2 can only evaluate to *either* value 0 *or* 1.

[3] Strictly speaking, in case either the left or right is just a single boolean value we do not put brackets around it. But this does not change the calculation, as we count 1 (satisfying parenthesization) if that value equals b else 0.

[4] This could still result in a valid bracket term, but we would not consider it reasonable by our definition as we would either have enclosed the complete term B inside a pair of brackets or enclosed a substring of B in 2 pairs of brackets.