NTIN066 - solutions 1

David Mareček

February 16, 2022

1. What is the difference between $O(g(n))$, $\Omega(g(n))$, and $\Theta(g(n))$?

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm. A function $f(n)$ belongs to the set $O(g(n))$ if there exists a positive constant c such that it lies between 0 and $cg(n)$, for sufficiently large n.

$$\exists c > 0 \; \exists n_0 \; \forall n \geq n_0 : f(n) \leq cg(n)$$

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best-case complexity of an algorithm. A function $f(n)$ belongs to the set $\Omega(g(n))$ if there exists a positive constant c such that it lies above $cg(n)$, for sufficiently large n.

$$\exists c > 0 \; \exists n_0 \; \forall n \geq n_0 : f(n) \geq cg(n)$$

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm. A function $f(n)$ belongs to the set $\Theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be sandwiched between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large n.

$$\exists c_1 > 0 \; \exists c_2 > 0 \; \exists n_0 \; \forall n \geq n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n)$$

2. Make a queue from two stacks

You have two stacks, supporting only the POP and PUSH operations. Propose an algorithm, that would simulate a Queue with operations ENQUEUE and DEQUEUE. Besides the two stacks, you have only a constant amount of memory. Show that the queue operations have a constant amortized time complexity.

One stack (let's denote it as A) is for ENQUEUE operations, the other one (B) is for DEQUEUE operations. If a DEQUEUE is needed and B is empty, copy all the items in reverse order from A to B. This will take n POP and n PUSH operations. Each item is copied at most once, so for each item, we can spend at most 4 operations: 1 for PUSH into A, two for copying (POP from A and PUSH to B), and one for POP from B. Therefore, the amortized time complexity for EN(DE)QUEUEing an item is constant.

3. Flexible arrays with $C' = 3C$
Compute the amortized time complexity of a single append, if the newly allocated memory is always three-times bigger.

After adding n elements, all reallocations together take $1+3+9+27+81+\ldots+3^{k-1}$, *where* $3^{k-1} \leq n < 3^k$. *Then,* $n < 3^k \leq 3n$. *The sum of first n members of a geometric progression is* $s_n = a \cdot (r^n - 1)/(r-1)$. *Applied to our case, we have* $s(n) = 1 + 3 + 9 + 27 + 81 + \ldots + 3^{k-1} = 1 \cdot (3^k - 1)/(3-1) = (3^k - 1)/2$. *Because* $3^k \leq 3n$, *the sum* $s(n) < 3n$, *so we found the needed constant* $c = 3$ *and therefore* $s(n) \in O(n)$. *And because* $3^k > n$, *the sum* $s(n) > n/2$, *so we have the needed constant* $c = 1/2$ *and therefore* $s(n) \in \Omega(n)$.

4. Flexible arrays with $C' = C + D$
Compute the amortized time complexity of a single append, if the newly allocated memory is always bigger by a constant D.

After adding n elements, all reallocations together take: $1 + 1 + D + 1 + 2D + 1 + 3D + 1 + 4D + \ldots + 1 + (k-1)D$, *where* $1 + (k-1)D \leq n < 1 + kD$. *Then,* $n < 1 + kD \leq n + D$. *The sum of first n members is:* $s(n) = 1 + 1 + D + 1 + 2D + \ldots + 1 + (k-1)D = k + Dk(k-1)/2 = k(1 + D(k-1)/2)$. *This is always lower than* n^2, *so we proved that the complexity is* $O(n^2)$ *by setting* $c_1 = 1$. *To find a lower bound, we take the inequality* $n < 1 + kD$, *from which we derive* $(n-1)/D < k$ *and* $(n + 1 - D)/2 < 1 + (k-1)D/2$. *Then,* $s(n) = k(1 + D(k-1)/2 > (n-1)/D \cdot (n + 1 - D)/2 = (n^2 - nD + D - 1)/2D > (n^2 - nD)/2D = (Dn^2 - n)/2D^2 > n^2/2D^2$. *So we found the constant* $c_2 = 1/2D^2$ *and proved that the complexity is* $\Omega(n^2)$. *So the amortized complexity of the whole sequence is* $\Theta(n^2)$ *which is* $\Theta(n)$ *per one operation.*

5. Add to front
So far, new elements could be added only to the end of array. Is it possible to modify the array so that we can also add elements to the front? And what about deleting elements?

We can have rolling array and keep two pointers to the first (f) and to the last (l) item in our data. When we want to add an item and $f = (l+1)$ *mod* C, *we need to reallocate and expand the array.*