Algorithms and Programation

Megi Dervishi

October 23, 2019

Homework 1 (09/10/2019)

Exercise 1

Solution

1. Let n be the size of the array. We will first perform insertion until $\frac{n}{2}$. Then everytime we add a new element the array would expand and if we delete an element the size would cut in half. For example we can have the following sequence of operations which would result in expansion one time and shrinking two times i.e. a very big amortized cost ($\mathcal{O}(n)$)

   Add , Del , Del, Add, Add, Del, Del ...

2. Instead of shrinking by 1/2 we can shrink by 1/4 of the size of the array. We can still expand twice the size of the array when inserting an element. This way the deletion cost is "covered" by the insertion.

Notations:

1. Φ_i := Potential after the operation i
2. c_i := Capacity(size) of the array after the operation i
3. n_i := Number of elements in the array after the operation i
4. α_i := The Load factor which is the faction of the number of elements stored in the array over the capacity(size) after operation i

Definitions:

We define the potential as:

$$\Phi_i := \begin{cases} 2 \cdot n_i - c_i & \text{if } \alpha_i \geq \frac{1}{2} \\ \frac{c_i}{2} - n_i & \text{if } \alpha_i < \frac{1}{2} \end{cases}$$

where $\Phi_0 = 0$, $\Phi_i = 0$ when $\alpha_i = \frac{n_i}{c_i} = \frac{1}{2}$, $\Phi_i = n_i$ when $\alpha_i = \frac{1}{4}$ and $\Phi_i \geq 0, \forall i \geq 0$.

Proof

There are two operations which have their own specific cases depending on the load factor at position i and $i-1$: Insertion and Deletion of an element.

Insertion

Insertion has the following definition $n_i = n_{i-1} + 1 \qquad (\star)$

- case 1: $\alpha_{i-1} \geq \frac{1}{2}$
  (proof from lecture, constant amortized cost $\mathcal{O}(1)$)
- case 2: $\alpha_i < \frac{1}{2} \wedge \alpha_{i-1} < \frac{1}{2}$
  Since load factor is smaller than half then there is no expansion i.e. $c_i = c_{i-1}$ Using $(\star)$ we obtain:

$$\hat{t_i} = t_i + \Phi_i - \Phi_{i-1} = 1 + (\frac{c_i}{2} - n_i) - (\frac{c_{i-1}}{2} - n_{i-1})$$

$$= 1 + (\frac{c_i}{2} - n_i) - (\frac{c_i}{2} - (n_i - 1)) = 0 \leq \mathcal{O}(1)$$

- case 3: $\alpha_i \geq \frac{1}{2} \wedge \alpha_{i-1} < \frac{1}{2}$
  Expansions may not occur here either and $c_i/2 = n_i$ so we have:

$$\hat{t}_i = t_i + \Phi_i - \Phi_{i-1} = 1 + (\frac{c_i}{2} - n_i) - (2n_{i-1} - c_{i-1})$$

$$= 3 + 3\frac{c_i}{2} - 3n_i = 3 + 3\frac{c_i}{2} - 3\frac{c_i}{2} = 3 \leq \mathcal{O}(1)$$

  So for all cases the amortized cost is constant.

Deletion

Deletion has the following definition $n_i = n_{i-1} - 1$ (♠)

- case 1 : $\alpha_i \geq \frac{1}{2} \wedge \alpha_{i-1} > \frac{1}{2}$
  Here shrinking may not happen as the load factor is bigger than half i.e. $c_i = c_{i-1}$. Using (♠) we get:

$$\hat{t}_i = t_i + \Phi_i - \Phi_{i-1} = 1 + (2n_i - c_i) - (2n_{i-1} - c_{i-1}) = -1 = \mathcal{O}(1)$$

- case 2: $\alpha_i < \frac{1}{2} \wedge \alpha_{i-1} = \frac{1}{2}$ Shrinking does not occur.

$$\hat{t}_i = t_i + \Phi_i - \Phi_{i-1} = 1 + (\frac{c_i}{2} - n_i) - (2\frac{1}{2} - 1)$$

$$= 1 + (\frac{c_i}{2} - n_i) = 1 + (n_i + 1 - n_i) = 2 = \mathcal{O}(1)$$

- case 3: $\alpha_{i-1} = \frac{1}{4}$
  Shrinking happens so the capacity and the number of elements halfs and also we observe that $t_i = n_i + 1$

$$\hat{t}_i = t_i + \Phi_i - \Phi_{i-1} = n_i + 1 + (\frac{c_i}{2} - n_i) - (\frac{c_{i-1}}{2} - n_{i-1})$$

$$= n_i + 1 - n_i = 1 = \mathcal{O}(1)$$

- case 4: $\frac{1}{4} \leq \alpha_i < \frac{1}{2}$

$$\hat{t}_i = t_i + \Phi_i - \Phi_{i-1} = 1 + (\frac{c_i}{2} - n_i) - (\frac{c_{i-1}}{2} - n_{i-1}) = 2 = \mathcal{O}(1)$$

In conclusion we have shown that the amortized cost for this potential (and method) is constant.

Exercise 2

Solution

(1)

Assuming that we have f such that it computes any value in F in constant time then we would have the following:

Algorithm 1 Distance between two Strings

```
1:  procedure EDIT_DISTANCE (S,T)
2:      n ← length of string S
3:      n ← length of string T
4:      Tab ← Table of size n × n
5:      for all i ← 1 to n by jumping t do
6:          for all j ← 1 to l to n by jumping t do
7:              B ← Tab[i][j]
8:              A = Tab[i : i + t][j]
9:              C = Tab[i][j : j + t]
10:             D = S[i : i + t]
11:             E = T[j : j + t]
12:             for k → 1 to t do
13:                 for l → 1 to t do
14:                     Tab[i + k][j + l] ← f(A, B, C, D, E)
```

Time Complexity: $\mathcal{O}((\frac{n}{t})^2 \cdot t^2) = \mathcal{O}(n^2)$

(2)

Take (i, j) then it is trivial to see that $|Tab(i-1, j) - Tab(i, j)| \leq 1$ since moving left (resp. up, right and down) corresponds simply to a single string operation i.e. to go left, right, up or down one requires only one deletion or one insertion. Moving in diagonal corresponds to a replacement, which has a cost of 0 if the two characters of the alphabet are identical and 1 otherwise. So no matter the direction: left, up, right, down, superior left diagonal ... each movement corresponds only to 1 string operation of cost lesser or equal to 1. So every cell has at most 1 of difference from its neighbors.

(3)

To compute the value of a cell in F one does $\min(a, b, c)$ where a, b, c are values in A, B, C plus some constant. Then:

$$\min(a - B, b - B, c - B) = \min(a, b, c) - B$$

Therefore

$$f(A - B, 0, C - B, D, E) = f(A, B, C, D, E) - B$$
$$f(A' - B', 0, C' - B', D', E') = f(A', B', C', D', E') - B'$$

Since $A - B = A' - B'$, $C - B = C' - B'$, $E = E'$, $D = D'$ then we have:

$$f(A, B, C, D, E) = f(A', B', C', D', E') - (B' - B)$$

(4)

We saw in the previous question that we can pre-compute f using $B = 0$ and calculate values for $B \neq 0$ by simply adding a constant. From question 2 we know that the distance in between terms is of at most 1. Then at each term of the list we have $A_{i+1} = A_i + \begin{vmatrix} 1 \\ 0 \\ -1 \end{vmatrix}$. Therefore for each term we have 3 choices, and we have a total of t terms, so from combinations we know that we have 3^t possibilities.This means that we have 3^t possible lists for A (and C), so a total of 3^{2t} possible inputs for A, C. Now since we assumed the alphabet has a finite constant size we know that by the same reasoning we have σ^{2t} possibilities for D, E. Then for any given input we need to compute every term of F i.e. $(t-1)^2$ terms. So in total to compute one cell our algorithm would be $\mathcal{O}(3^{2t}\sigma^{2t}(t-1)^2) = \mathcal{O}(3^{2t}\sigma^{2t}t^2)$.

(5)

We have to find a way to store the values of $f(A, B, C, D, E)$, since we have $(3\sigma)^{2t}$ possible inputs. An easy way to store them would be in a tree of depth $4t$ where the root is B, the first t layers each node would have 3 branches representing the term fact that $A_{i+1} = A_i + \begin{vmatrix} 1 \\ 0 \\ -1 \end{vmatrix}$. The layers t to $2t$ would be structured the same and represent the list C. Finally in the layers $2t$ to $3t$ nodes would have σ branches to construct D and identically layers $3t$ to $4t$ would construct E. Since our tree has a depth of $4t$ a lookup in it is of order $\mathcal{O}(t)$. Then we have (we store directly the value from which the root has been removed):

Algorithm 2 Optimized

```
1:  procedure EDIT_DISTANCE (S,T)
2:      n ← length of string S
3:      n ← length of string T
4:      Tab ← Table of size n × n
5:      for all i ← 1 to n by jumping t do
6:          for all j ← 1 to l to n by jumping t do
7:              B ← Tab[i, j]
8:              A ← Tab[i, j : j + t] − B
9:              C ← Tab[i : i + t, j] − B
10:             D ← S[i : i + t]
11:             E ← T[j : j + t]
12:             Tab[i + 1 : i + t, j + 1 : j + t] ← Lookup_f(A, 0, C, D, E) + B
```

The two for loops give complexity $\mathcal{O}(\frac{n^2}{t^2}k)$ where $\mathcal{O}(k)$ is the complexity of the computations inside. Now looking at those more in detail all the setting and manipulations from line 8 to 11 would be linear in t and the

complexity of the lookup is also linear in t (as discussed previously), so overall the complexity of the whole algorithm is: $\mathcal{O}(\frac{n^2}{t})$. Since we are free on the value of t we can choose $t = \log n$ and we get an algorithm that runs in $\mathcal{O}(\frac{n^2}{\log n})$. While in memory it costs $\mathcal{O}((3\sigma)^{2\log n})$ so in total it would take $\mathcal{O}(n^2 + (3\sigma)^{2\log n}) = \mathcal{O}(n^2 + \exp(2\log(3\sigma)\log n)) = \mathcal{O}(n^2 + n^{2\log(3\sigma)})$.

Exercise 3

Solution

```python
def subarraySum(k, A):
    # it's a bit ugly with all the cases ...
    sums, n = 0 , len(A)
    if len(A) <= 1 : return False

    if k != 0:
        remainder = [0] * k
        remainder[1] =1
        for i in range(n): #O(n)
            sums = (sums + A[i]) % k
            #case e.g: [0,0,_,.._,k,k,_.._,0,0,_..,k,k]
            if A[i] == k or A[i] == 0:
                if (i!=0 and i!=n-1) and (A[i-1] == A[i] or A[i+1] == A[i]):
                    return True
                elif (i==0 and A[0] == A[1]): return True
                elif (i==n-1 and A[-1] == A[-2]): return True
                else: continue
            else:
                remainder[(sums+1)%k] += 1
            if remainder[(sums + 1)%k] >= 2: return True
        return False

    else:
        for i in range(n): #O(n)
            if A[i] == k:
                if (i!=0 and i!=n-1) and (A[i-1] == A[i] or A[i+1] == A[i]):
                    return True
                elif (i==0 and A[0] == A[1]): return True
                elif (i==n-1 and A[-1] == A[-2]): return True
                else: continue
        return False
```

Exercise 4

Solution

```
def majority_element(T):
    #Your code goes here
    if len(T) == 0: return False
    res = majority_aux(T)
    if res[0] <= len(T)/2: return False
    return res

def majority_aux(T):
    if len(T) <= 1: return (1,T[0])
    k = len(T)//2
    left = majority_aux(T[0:k]) #split T
    right = majority_aux(T[k:])

    if left[1] == right[1] : return (2, left[1])
    else:
        count_l = 0 #increment for the left split
        count_r = 0 #increment for the right split
        for i in range(len(T)):
            if T[i] == left[1]: count_l+=1
            elif T[i] == right[1]: count_r += 1
            else: continue
        if count_l > count_r: return (count_l, left[1])
        return (count_r, right[1])

```

Exercise 5

Solution

```
def median(nums1, nums2):
    #Your code goes here
    m , n = len(nums1),len(nums2)
    sums = m+n

    if sums == 0: return 0

    if sums % 2 == 0: #even
        res = (findk(nums1,m,nums2,n, sums//2)
        + findk(nums1,m,nums2,n, sums//2 +1))/2
        return float(res)
    res2 = findk(nums1,m,nums2,n, sums//2 + 1)
    return float(res2)

def findk(M,m,N,n,k):
    #small cases
    if m > n:
        M,m,N,n,k = N,n,M,m,k
    if m == 0:
        return N[k-1]
    if k == 1:
        return min(M[0],N[0])

    index1 = min(int(k//2), m) #pointer for M
    index2 = k-index1; #pointer for N

    if M[index1-1] <= N[index2-1]: #compare the medians

        return findk(M[index1:], m-index1, N, n, k-index1)
    else:
        return findk(M, m, N[index2:], n-index2, k-index2)
```

Appendix A (Questions)

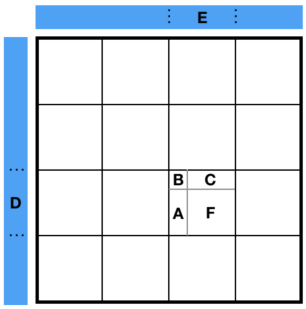
Exercise 1.1

We have seen how dynamic arrays enable arrays to grow while still achieving constant-time amortized performance. This problem concerns extending dynamic arrays to let them both grow and shrink on demand.

1. Consider an underflow strategy that cuts the array size in half whenever the array falls below half full. Give an example sequence of insertions and deletions where this strategy gives a bad amortized cost.
2. Then, give a better underflow strategy than that suggested above, one that achieves constant amortized cost per deletion. Prove that the amortized cost is indeed constant.

Exercise 1.2

The edit distance between two strings S, T is defined as the minimum number of edit operations that are required in order to transform S to T. The edit operations are: substitution (replace one letter of S by another letter), deletion (delete one letter of S), or insertion (insert one letter). Your task is to develop a dynamic programming algorithm that computes the edit distance between strings S and T, both of length n, in $O(n^2/\log n)$ time. You can assume that the size of the alphabet of S and T, σ is constant.



Recall the $O(n^2)$-time dynamic programming algorithm for computing the edit distance. Let D be the dynamic programming table. Divide this table into non-overlapping square blocks of size $t \times t$ (see the figure below). Note that the block's values F depend only on the first column A, the first row C, the corner value B, and the substrings D, E of S, T, resp.

Suppose that you have a function f that takes A, B, C, D, E, preprocesses them, and after that can compute any value in F in constant time.

1. Modify the dynamic programming algorithm to use the function f.
2. Show that the values in any two neighbouring cells (by side or by corner) differ by at most one.
3. Consider two different blocks corresponding to values A, B,C,D,E and A', B', C', D', E'. Suppose that after subtracting B from every value in A and C and B' from every value in A' and C', we have $A = A', B = B', C = C', D = D', E = E'$. What can you say about the values of the function f for these blocks?
4. Show that you can precompute the values of the function f in $O(3^{2t}\sigma^{2t}t^2)$ time for all possible A, B, C, D, E simultaneously.
5. Derive an $O(n^2/\log n)$-time algorithm for computing the edit distance.

Exercise 1.3

Given a list of non-negative numbers and a target non-negative integer k, write a function subarraySum(k, A) to check if the array has a continuous subarray of size at least 2 that sums up to a multiple of k, that is, sums up to $p \cdot k$ where p is also an integer. The function must output True if there is such a subarray and False otherwise.

The running time of your algorithm must be $O(n)$. Don't use any data structures excepts for lists.

For example:

Test	Result
print(subarraySum(2, [1, 2, 4, 3, 5]))	True

Exercise 1.4

The majority element of an array is the element that appears more than n/2 times (and thus there is at most one such element). Given a list of integers T, develop a recursive algorithm that returns False if T does not contain a majority element, and otherwise returns a tuple of two integers nb, elt where elt is the majority element of T and nb the number of times it appears in T.

For example:

Test	Result
print(majority_element([1, 1, 100, 7, 100, 100, 7, 100, 100]))	(5, 100)
print(majority_element([1,2,1,2]))	False

Exercise 1.5

Let T be a sorted array. The median of T is its $(k+1)$-th element if T is of length $2k+1$, and the average of its k-th and $(k+1)$-th elements if it is of length $2k$. Let T_1 and T_2 be two sorted arrays. The median of T_1 and T_2 is the median of the array obtained by merging T_1 and T_2.

Let $nums1$ and $nums2$ be two sorted array of size m and n respectively. Write a function $median(nums1, nums2)$ that finds the median of $nums1$ an $nums2$. The algorithm must run in time complexity $O(\log(m+n))$ (in particular you have no time to merge the two arrays).

You may assume $nums1$ and $nums2$ cannot be both empty.

DO NOT CHANGE THE NAME OF THE FUNCTION.

For example:

Test	Result
print(median([2,5,9,14],[3,7,8,12]))	7.5