Homework 2

BY SILAO_XU

Problem 1.

1. How much time does it take to read in the *same* order?

   Let l be the *cache lines size*, c be the *total cache size*, t be the time used for copying an aligned block of memory of size l containing the requested location to the cache.

   Assume l divides n and the cache is empty originally. We will read the $n \times n$ array in the *same* order as it is stored in.

   Because l divides n, $\frac{n^2}{l}$ is the total number of blocks of memory we need to copy to the cache. Even though *least recently used* cache lines would be evicted from the cache when the cache is full, it's irrelevant to the latest iteration. So, we only need to do $\frac{n^2}{l}$ times of copying without any need to fetching backward. So the total time would be

   $$\frac{n^2}{l}t \tag{1}$$

2. How much time does it take to read in the *other* order?

   Let l be the *cache lines size*, c be the *total cache size*, t be the time used for copying an aligned block of memory of size l containing the requested location to the cache.

   Assume l divides n and the cache is empty originally. We will read the $n \times n$ array in the *other* order from which it is stored in.

   (Case 1) If $c \geqslant n \times l$, which means the cache is big enough to store an entire row. Even though the array are read in the *other* order from which it is stored in, every time it wants to fetch elements from next row, there is no need to fetch that again since it has been already stored in the cache. So in this case the total time would be

   $$\frac{n^2}{l}t \tag{2}$$

   (Case 2) If $c < n \times l$, which means the cache is not big enough to store the entire row. When the cache is full, the *least recently used* (in this case it's the oldest cache line) column would be evicted out and for next row iteration the same column which is in size l would be fetched again. So there would be totally n^2 times of copying. The time measured for this case would be

   $$n^2 t \tag{3}$$

3. Plot the answer.

4. Describe qualitatively how, in practice, you might notice that your code is running into cache issues.

   Let l be the *cache lines size*, c be the *total cache size*, t be the time used for copying an aligned block of memory of size l containing the requested location to the cache.

   Citing from *Qustion 1* and *Question 2*, from (1) and (2), we know that given a $n \times n$ array, where l divides n, when $c \geqslant n \times l$, no matter in what order we are reading the array, we will have the same running time, that is $\frac{n^2}{l}t$. However, when $c < n \times l$, from (1) and (3), we know that reading in the *same order* would be qualitatively l times faster than reading in the *other order*.

   So, in practice if we don't run into cache issues, there would be no difference between traversing column first (let i be the index for traversing columns) and traversing row first (let j be the index for traversing rows). If we get significant different running time we could reason that our array length n has exceeded the $\frac{c}{l}$ limit. Then we can judge which iterating strategy (iterating i first or iterating j first) is in the *same order* with caching.

Problem 2.

1. How many operations can a single assignment require?

   Assume the time complexity for allocating and freeing memory are both $O(\log(n))$, let it be $a_{\text{alloc}} \log(n) + c_{\text{alloc}}$ and $a_{\text{free}} \log(n) + c_{\text{free}}$, where a_{alloc}, a_{free}, c_{alloc}, c_{free} are constant values. Let e be the exceeding length.

   The time for allocating new memory would be $a_{\text{alloc}} \log(n + e) + c_{\text{alloc}}$. The time for assigning the new value is constant, name it c_{assign}. Time for copying the original array, which is proportional to the size n, is $n \times c_{\text{assign}}$. The time for freeing the original array memory is $a_{\text{free}} \log(n) + c_{\text{free}}$. So, the time combined together is:

   $$T_1 = a_{\text{alloc}} \log(n+e) + a_{\text{free}} \log(n) + (n+1)c_{\text{assign}} + c_{\text{alloc}} + c_{\text{free}}$$

   In sum, the total time is dominated by the assigning part, that is $O(n)$ in total.

2. Given an algorithm that involves at most n assignments to positions $\leqslant n$ in an array, how many operations might Matlab require to execute the code?

   In the worst case, when every time the position we need to assign exceeds the original array length, we would need to do $\sum_{i=1}^{N} i$ times allocations for new array; we need to do $\sum_{i=1}^{N-1} i$ number of scale for elements shifting from original array to the new array; and we also need to do $\sum_{i=1}^{N-1}$ number of scale for freeing the original array. For simplicity, we can ignore the coefficient a_{alloc}, a_{free}, a_{assign} and constant values c_{alloc}, c_{free}, c_{assign} because they would be compressed in big-O notation. So the time complexity for Matlab's algorithm would be calculated as

   $$\begin{aligned} &\underbrace{\log(1) + \log(2) + \ldots + \log(N)}_{\text{operations for allocation}} \; + \\ &\underbrace{\log(1) + \log(2) + \ldots + \log(N-1)}_{\text{operations for freeing}} \; + \\ &\underbrace{1 + 2 + \ldots + N - 1}_{\text{operations for copying}} \; + \\ &\underbrace{N}_{\text{ops for assigning}} \end{aligned}$$

   $\Rightarrow$

   $$\underbrace{\log(N!)}_{\text{operations for allocation}} + \underbrace{\log((N-1)!)}_{\text{operations for freeing}} + \underbrace{(N-1)(N-2)/2}_{\text{ops for copying}} + \underbrace{N}_{\text{ops for assigning}}$$

   $$\leqslant 2N \log(N) + \frac{N(N-1)}{2}$$

   which would finally dominated by the copying operations part, $O(N^2)$ in total.

3. What is the competitive ratio of Matlab's algorithm to the optimal (with *foresight*) algorithm?

   Assume the size of allocating with *foresight* is N, where $N = n + e$, e is the predicted extension and n is the array size originally demanded, $e \geqslant 1$ and $n \geqslant 1$. We also assume that in total it involves at most N assignment to positions $\leqslant N$.

   a) Competitive Ratio of Time

For the original Matlab's algorithm, in the worst case, when every time the position we need to assign exceeds the original array length, we would need to do $\sum_{i=1}^{N} i$ times allocations for new array; we need to do $\sum_{i=1}^{N-1} i$ number of scale for elements shifting from original array to the new array; and we also need to do $\sum_{i=1}^{N-1}$ number of scale for freeing the original array. For simplicity, we can ignore the coefficient $a_{\text{alloc}}, a_{\text{free}}, a_{\text{assign}}$ and constant values $c_{\text{alloc}}, c_{\text{free}}, c_{\text{assign}}$ because they would be compressed in big-O notation. So the time complexity for Matlab's algorithm would be calculated as

$$\begin{aligned} &\underbrace{\log(1)+\log(2)+...+\log(N)}_{\text{operations for allocation}} \ + \\ &\underbrace{\log(1)+\log(2)+...+\log(N-1)}_{\text{operations for freeing}} \ + \\ &\underbrace{1+2+...+N-1}_{\text{ops for copying}} \ + \\ &\underbrace{N}_{\text{ops for assigning}} \end{aligned}$$

$\Rightarrow$

$$\underbrace{\log(N!)}_{\text{operations for allocation}} + \underbrace{\log((N-1)!)}_{\text{operations for freeing}} + \underbrace{(N-1)(N-2)/2}_{\text{ops for copying}} + \underbrace{N}_{\text{ops for assigning}}$$

$$\leqslant 2N\log(N) + \frac{N(N-1)}{2}$$

which would finally dominated by the copying operations part, $O(N^2)$ in total.

For the optimal algorithm, we only need to allocate once, the time complexity for allocating would be $O(\log(N))$ and for assigning would be $O(N)$ and no need for shifting. The overall complexity would be dominated by the assigning part, that is, $O(N)$ in total.

So the competitive ratio between the Matlab algorithm and the optimal algorithm would be

$$\frac{O(N^2)}{O(N)}$$

b) Competitive Ratio of Memory Usage

For the Matlab algorithm, in the worst case, we would allocate new array, which is 1 bigger in size than the original one when the index exceeds the array boundary. It would take at most $N + N - 1$ number of memory at once, where N is for the newest array, $N - 1$ is for the original array, waiting for shifting. $N - 1$ size of memory would be freed in the end. So the memory usage would be $O(N)$ overall.

For the optimal algorithm, we allocate only once and it is the final size. So it would be $O(N)$ overall as well.

The competitive ratio between the Matlab algorithm and the optimal algorithm would be

$$\frac{O(N)}{O(N)}$$

4. Improvement. Every time we are forced to expand an array, instead of expanding it only as much as is necessary, we now round up to the nearest power of 2 size. Write pseudocode for the new strategy.

```
assign(array, value, i, n):
```

```
if i < n then:
    array[i] ← value
else:
    newSize ← 2^⌈log2(i)⌉
    memPtr ← allocate(newSize)
    free(array, n)
    array ← memPtr
    array[i] ← value
```

5. Analyze your pseudocode: What is its competitive ratio with OPT? How does its memory use compare with OPT? How does its memory use compare with the original Matlab implementation?

   a) Competitive Ratio of Time

   We are doing assignment with array index i, where i is from 1 to N. For convenience of computation, we let $N = 2^k$ $(i.e, k = \log(N)), k \geqslant 0$. We also assume that in total it involves at most N assignment to positions $\leqslant N$.

   For the algorithm given above, in the worst case when every time for assignment the index exceed only 1 of the original size, the overall space scale we need to allocate is $\sum_{i=1}^{k} 2^i$ for the reason that we round up the nearest power of 2 size everytime and given any consecutive index access between $[2^{i-1}, 2^i)$ there is no need to assign again.

   For simplicity, we can ignore the coefficient $a_{\text{alloc}}, a_{\text{free}}, a_{\text{assign}}$ and constant values $c_{\text{alloc}}, c_{\text{free}}, c_{\text{assign}}$ because they would be compressed in big-O notation. So the time complexity for this algorithm would be calculated as

$$\begin{aligned} &\underbrace{\log(2^1) + \log(2^2) + ... + \log(2^k)}_{\text{operations for allocation}} \; + \\ &\underbrace{\log(2^1) + \log(2^2) + ... + \log(2^{k-1})}_{\text{operations for freeing}} \; + \\ &\underbrace{2^1 + 2^2 + ... + 2^{k-1}}_{\text{ops for copying}} \; + \\ &\underbrace{N}_{\text{ops for assigning}} \end{aligned}$$

   $\Rightarrow$

$$\underbrace{\log(2^{1+2+3+...+k})}_{\text{operations for allocation}} + \underbrace{\log(2^{1+2+3+...+k-1})}_{\text{operations for freeing}} + \underbrace{2^k - 2}_{\text{ops for copying}} + \underbrace{N}_{\text{ops for assigning}}$$

$$\leqslant 2 \times \frac{k(k-1)}{2} + 2^k - 2 + 2^k$$

   So, the overall complexity would be dominated by the copying and assigning part, namely $O(2^k)$. Substituting k with $\log(N)$, the running time complexity of this algorithm would be $O(N)$. The competitive ratio between the OPT algorithm and this algorithm would be

$$\frac{O(N)}{O(N)}$$

   b) Competitive Ratio of Memory Usage

   For this algorithm, we keep assigning from 1 to N, where $N = 2^k$, $k \geqslant 0$. We will keep reallocating at an interval of $2^i - 2^{i-1}$, i is from 1 to k. It would take at most $2^k + 2^{k-1}$ number of memory at once, where 2^k is for the newest array, 2^{k-1} is for the original array, waiting to be shifting. 2^{k-1} size of memory would be freed in the end. Substituting k with $\log(N)$, we will get $O(N)$ space complexity.

The competitive ratio between the OPT algorithm and the algorithm here we are trying to optimize would be

$$\frac{O(N)}{O(N)}$$

6. Instead of rounding up to the nearest power of 2, repeat the calculations assuming we round up to the nearest power of c, for some constant $c > 1$. Reason about the c you pick.

Problem 3.

1. The amount of memory used by the standard dynamic programming approach to edit distance is $O(n^2)$, which is the same order as the amount of time the algorithm takes; while we might be happy waiting for several billion CPU cycles, our code might crash on current hardware if it also demands several billion memory locations. Fortunately, we do not need to store the whole two-dimensional table: it is enough to store just the current row being computed, and the previous row that we have just finished computing. Write pseudocode for this strategy.

```
edit_distance(s1, s1_len, s2, s2_len):
    create 2D array M[2][s2_len + 1]
    for j ← 1 to s1_len do:
        M[0][j] ← j
    for i ← 1 to s2_len do:
        M[i % 2][0] ← i
        for j ← 1 to s1_len do:
            if s1[i - 1] = s2[j - 1] then:
                M[i % 2][j] ← M[(i - 1) % 2][j - 1]
            else:
                iPrev ← M[(i - 1) % 2][j]
                jPrev ← M[i % 2][j - 1]
                ijPrev ← M[(i - 1) % 2][j - 1]
                M[i % 2][j] ← min(iPrev, jPrev, ijPrev) + 1
    return M[(s1_len - 1) % 2][s2_len]
```

2. Describe, in a few sentences, and using concepts from lecture and/or your knowledge of systems why and how this approach is an improvement.

   In this strategy, we only need to use $2n$ memory and optimally the entire table could be loaded in cache and new value could be renewed just in cache.

3. Find a way to lay out the entries of the table in memory so that when the two letters are the same, instead of doing a memory copy, the algorithm *does nothing*. Write pseudocode for the edit distance problem where the *same* memory address is used for location $(i-1, j-1)$ and for location (i, j).

```
edit_dist_relayout(s1, s1_len, s2, s2_len):
    create array M[s2_len + 1]
    for j ← 0 to s1_len do:
        M[j] ← j
    for i ← 1 to s2_len do:
        if s1[0] ≠ s2[i - 1] do:
            M[0] ← min(M[0], M[1], i) + 1
        for j ← 2 to s1_len do:
            if s1[j - 1] ≠ s2[i - 1]:
                M[j - 1] ← min(M[j - 2], M[j - 1], M[j]) + 1
    return M[s2_len]
```

4. Describe in a few sentences how your algorithm works, and why it is correct.

a	b
c	d

Table 1. $a = M[i-1][j-1], b = M[i-1][j], c = M[i][j-1], d = M[i][j]$

For the algorithm in *Question 1*, for every current iteration of i and j, we access 4 unit in the table. Let them to be $a = M[i-1][j-1], b = M[i-1][j], c = M[i][j-1], d = M[i][j]$. We'll no longer need c and a after the current iteration (color in RED) and will yank one new table unit d for next iteration (color in YELLOW). So the intuition is that if we could replace a or c with new value d, the benefit would be that we don't need to copy a and generate new value d if $s_1[i] = s_2[j]$ since the result of previous calculation has already been stored in a.

a	c	b

Table 2. $a = M[i-1][j-1], b = M[i-1][j], c = M[i][j-1]$

After re-layouting the table, for next iteration, we will need entry c, b again, where c might be renewed and b is the current diagonal value.

5.

6. In all of the above variants of the edit distance algorithm, we are computing the same entries of the same table, in the same order—just storing them differently. However, there is a different order of computing these entries that may have some advantages: diagonal. We start at (0, 0), and in round r, for $r \leqslant n$, compute the entries along the diagonal going from (r, 0) to (0, r); after n rounds of this, we next compute the entries in the diagonal from (n, 1) to (1, n), and so on. Write pseudocode for this and explain why it works.

```
edit_dist_diagonal(s1, s1_len, s2, s2_len):
    create 2D-array M[s1_len + 1][s2_len + 1]
    for j ← 0 to s2_len do:
        M[0][j] ← j
    for i ← 1 to s1_len do:
        M[i][0] ← i
    for i ← 1 to s1_len do:
        for j ← 1 to i do:
            if s1[i - j] = s2[j] do:
```

```
                M[i - j][j] ← M[i - j - 1][j - 1]
            else:
                M[i - j][j] ← min(M[i - j - 1][j],
                                  M[i - j - 1][j - 1],
                                  M[i - j][j - 1]) + 1
    for j ← 1 to s2_len do:
        for i ← j to 1 do:
            if s1[i - j] = s2[j] do:
                M[s1_len - (i - j)][j] ← M[s1_len - (i - j) - 1][j - 1]
            else:
                M[s1_len - (i - j)][j] ← min(M[s1_len - (i - j) - 1][j],
                                             M[s1_len - (i - j) - 1][j - 1],
                                             M[s1_len - (i - j)][j - 1]) + 1

    return M[s1_len][s2_len]
```

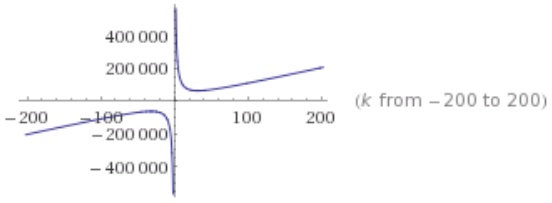
Since we are doing diagonal computation for $m \times n$ table, everytime $M[i][j]$ depends on the values $M[i-1][j-1], M[i][j-1], M[i-1][j]$ and they are all have been computed by prior diagonal computation.

7. Plot the rate of getting work done as a function of k, for a workload of 1,000,000 units. What is the best number of processors to use?

   Let the rate of getting work done to be R.

$$R = \frac{1,000,000}{k} + 1,000k$$

   Plot R:



   We know that given any real number a and b

$$(a+b)^2 \geqslant 0 \;\Rightarrow\; a^2+b^2+2ab \geqslant 0 \;\Rightarrow\; a^2+b^2 \geqslant 2ab$$

   R could be represented with the above form

$$\begin{aligned} R &= (\frac{1,000}{\sqrt{k}})^2 + (10\sqrt{10k})^2 \\ &\geqslant 20,000\sqrt{10} \end{aligned}$$

   The minimum rate should be $20,000\sqrt{10}$ and therefore the best number of processors to be use is $\lceil 10\sqrt{10} \rceil = 32$.

8. What is the general formula for the best number of processors to use?

   Let the rate of getting work done to be R.

$$\begin{aligned} R &= \frac{w}{k} + kp \\ &= (\sqrt{\frac{w}{k}})^2 + (\sqrt{kp})^2 \\ &\geqslant 2\sqrt{wp} \end{aligned}$$

The minimum rate should be $2\sqrt{wp}$ and if we substitute $2\sqrt{wp}$ into the equation $R = \frac{w}{k} + kp$ we could get the best number of processors to use is $\sqrt{\frac{w}{p}}$.

Problem 4.

1. Write pseudocode to look up location of a key in a k-ary search tree.

```
lookup(currRoot, key):
    if currRoot not exists:
        return NOT_FOUND
    for i ← 1 to k do:
        if currRoot.keys[i - 1] > key then:
            lookup(currRoot.children[i - 1], key)
        else if currRoot.keys[i - 1] = key then:
            return currRoot.children[i - 1]
    if i = k then:
        lookup(currRoot.children[i], key)
```

2. Given a search tree with n nodes, what is its depth?

$$\begin{aligned} k^0 + k^1 + k^2 + ... + k^{d+1} &= n \\ \frac{(1 - k^{d+1})}{1 - k} &= n \\ d &= \log_k(nk - n + 1) - 1 \end{aligned}$$

3. How long does a linear search for a random element in a list of length $k - 1$ take, under these rules?

$$\frac{1}{k-1} \sum_{j=1}^{k-1} (\underbrace{\frac{j}{c} \times t}_{\text{loading cache}} + \underbrace{j}_{\text{iterating the key list}} + 1)$$

4. Taking the product of your results in the last two parts of this problem should give you an estimate for how long a simple lookup will take in your k-ary tree. Now we must decide: what value of k is best? Plot the time you have computed as a function of k.

5. How to traverse?