Homework #1

(Due: March 18)

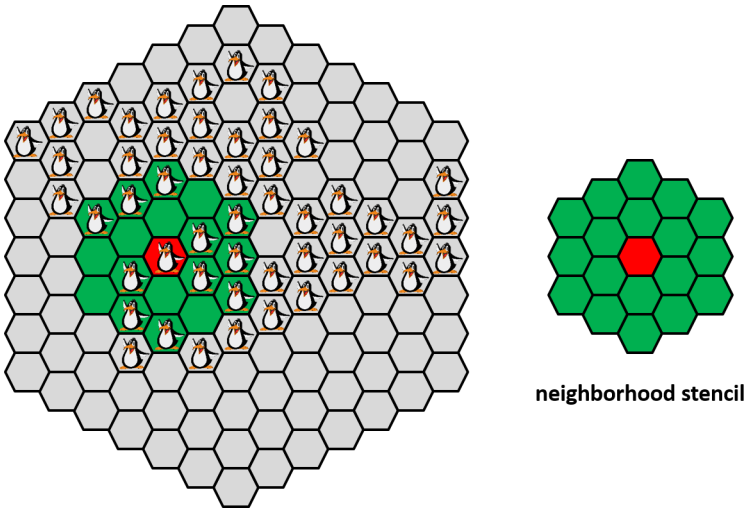

Figure 1: [Task 1] A base-7 hexagonal grid and a neighborhood stencil consisting of all cells (green) within distance 2 of the center cell (red). As an example, the stencil is centered on one of the cells of the grid to count its neighbor coefficient which turns out to be 12.

Task 1. [60 Points] Huddling Penguins[1]

Emperor penguins are known to form dense huddles to survive the brutal cold of Antarctica[2]. It has also been observed that each penguin in a huddle usually positions itself at the center of a regular hexagon in a nearly hexagonal packing of the group[3]. Each hexagonal cell is occupied by at most one penguin.

To study the spatial distribution of penguins in a huddle and how that changes over time it is useful to know the number of penguins in the neighborhood of each cell (including the penguin in that cell, if any) of the hexagonal grid. We will call that number the *neighbor coefficient* of that cell. A *neighborhood stencil* specifies the cells in the neighborhood of every cell.

A base-n ($n \geq 1$) hexagonal grid is composed of regular hexagonal cells such that each of the six sides of the grid contains exactly n hexagonal cells. Every two adjacent cells share an edge. A base-n hexagonal grid has exactly $3n(n-1)+1$ cells and $3(3n-2)(n-1)$ edges. The distance between two cells is the minimum number of edges one must cross in order to go from one of the

[1] Phillip McDowall and Prof. Heather Lynch posed question $1(a)$ to the Algorithms Reading Group in Fall 2017

[2] Watch on Youtube how emperor penguins huddle for warmth: `https://www.youtube.com/watch?v=OL7O5O7U4Gs`

[3] Waters A, Blanchette F, Kim AD (2012) Modeling Huddling Penguins. PLOS ONE 7(11): e50277. https://doi.org/10.1371/journal.pone.0050277

cells to the other. Figure 1 shows a base-7 hexagonal grid and a neighborhood stencil consisting of all cells (green) within distance 2 of the center cell (red). The neighbor coefficient of the red cell in the hexagonal grid is exactly 12 (= 11 penguins in the green cells + 1 penguin in the red cell).

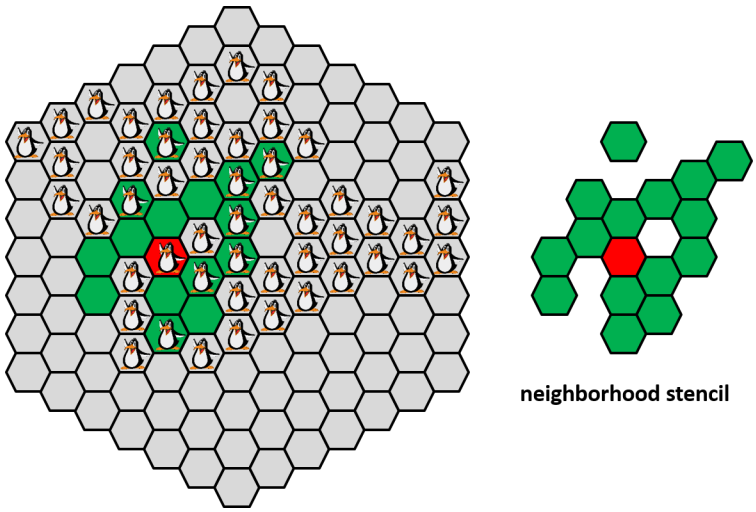

Figure 2: [Task 1] A base-7 hexagonal grid and an irregular shaped neighborhood stencil. Based on this stencil, the red cell of the grid on which the stencil is centered has a neighbor coefficient of 9.

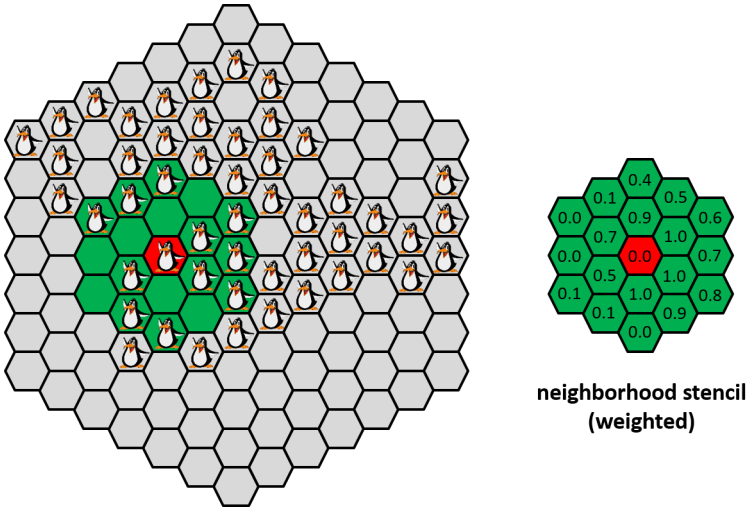

Figure 3: [Task 1] Same as Figure 3 except that each cell of the neighborhood stencil has a real weight. Based on this stencil, the red cell of the grid has a neighbor coefficient of 5.2.

Now answer the following questions.

(*a*) [**20 Points**] Given a base-n hexgonal grid G_n containing at most one penguin in each

cell and a neighborhood stencil consisting of all cells within distance $r \in [0, 2n-2]$ of the center cell (see, e.g., Figure 1), give $\Theta\left(n^2\right)$ algorithm for counting the neighbor coefficient of all $3n(n-1)+1$ cells in G_n. Observe that in this case, the stencil itself is a base-$(r+1)$ hexagonal grid.

(b) [**25 Points**] Repeat part 1(a) but with an irregular shaped stencil which is not necessarily connected. Assume that the stencil is a base-$(r+1)$ hexagonal grid with some cells missing, where $r \in [0, 2n-2]$. Figure 2 shows an example. Running time of your algorithm must be $\mathcal{O}\left(n^2 \log n\right)$.

(c) [**15 Points**] Repeat part 1(a) but assume that each cell of the neighborhood stencil has an arbitrary real-valued weight. So, you compute a weighted count instead of a standard count, that is, if a grid cell correponding to a stencil cell has a penguin in it you add the weight of stencil cell to the neighbor coefficient, otherwise you add nothing. Figure 3 shows an example. Running time of your algorithm must be $\mathcal{O}\left(n^2 \log n\right)$.

Task 2. [30 Points] Deriving Strassen's Algorithm

We studied Strassen's matrix multiplication algorithm in Lecture 3. However, the algorithm we derived in slides 17–18[4] is not exactly the sams as the algorithm we learnt in slides 5–15!

(a) [**10 Points**] Write down the algorithm we derived in in slides 17–18. How is it different from Strassen's algorithm we saw in slides 5–15?

(b) [**20 Points**] Derive the algorithm we discussed in slides 5–15. You must use the approach shown in slides 17–18 of Lecture 3.

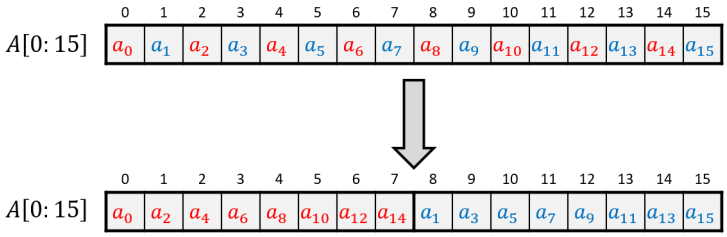

Figure 4: [Task 3] Unshuffling array $A[0:15]$.

Task 3. [50 Points] Unshuffling

Consider the *Rec-FFT* algorithm given on slide 29 of Lecture 4. The input to the algorithm is an array of coefficients $\langle a_0, a_1, a_2, \ldots, a_{n-1} \rangle$, where $n = 2^k$ for some integer $k \geq 0$. When $k > 0$, the algorithm unshuffles the input array into two subarrays – one containing the even-numbered coefficients $\langle a_0, a_2, \ldots, a_{n-2} \rangle$ and the other containing the odd-numbered coefficients

[4]There was a typo on slide 17. Please download the slides again.

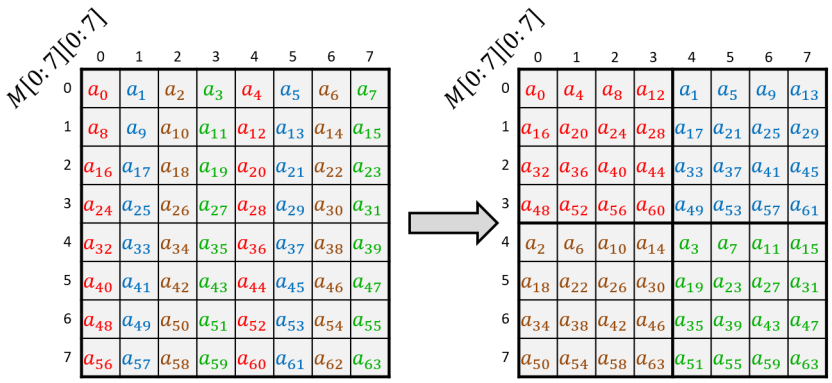

Figure 5: [Task 3] Unshuffling matrix $M[0:7][0:7]$.

$\langle a_1, a_3, \ldots, a_{n-1} \rangle$. Entries in both subarrays appear in exactly the same order as they appeared in the input array. Then in Line 5, *Rec-FFT* is recursively called with the even-numbered coefficients, and in Line 6 with odd-numbered coefficients. The algorithm, however, does not say how the unshuffling was done.

(*a*) [**20 Points**] Given an array $A[0:n-1]$ of coefficients $\langle a_0, a_1, a_2, \ldots, a_{n-1} \rangle$, where $n = 2^k$ for some integer $k > 0$, design an efficient recursive divide-and-conquer algorithm[5] for unshuffling the array using only $\mathcal{O}(1)$ extra space[6]. In the final unshuffled array, a_i must be stored in $A\left[\frac{n}{2}(i \mod 2) + \left\lfloor \frac{i}{2} \right\rfloor\right]$, for $i \in [0, n-1]$. Figure 4 shows an example. Analyze your algorithms running time. What will be *Rec-FFT*'s running time with this unshuffling algorithm?

(*b*) [**30 Points**] Now consider a square matrix $M[0:n-1][0:n-1]$, where $n = 2^k$ for some integer $k > 0$, and for $0 \leq l \leq n^2 - 1$, coefficient a_l is stored in $M\left[\left\lfloor \frac{l}{n} \right\rfloor\right][l \mod n]$. Design an efficient algorithm based on recursive divide and conquer which moves all a_l values with $l \mod 4 = 0$, $l \mod 4 = 1$, $l \mod 4 = 2$, and $l \mod 4 = 3$ to the top-left, top-right, bottom-left, and bottom-right quadrant, respectively. All coefficients in each quadrant must appear in exactly the same order (row by row) as they appeared in the original input matrix. More specifically, a_l must be stored in $M\left[\frac{n}{2}\left((l+1) \mod 2\right) + \left\lfloor \left\lfloor \frac{l}{4} \right\rfloor / \frac{n}{2} \right\rfloor\right]\left[\frac{n}{2}(l \mod 2) + \left(\left\lfloor \frac{l}{4} \right\rfloor \mod \frac{n}{2}\right)\right]$ in the final unshuffled matrix. Figure 5 shows an example. Your algorithm must use $\mathcal{O}(1)$ extra space. Analyze your algorithms running time.

Task 4. [40 Points] Fractals

In this task we will consider recursive divide-and-conquer algorithms for generating recursively self-similar structures.

Consider the algorithm given in Figure 7 which generates the Fractal object known as the *Sierpinski*

[5] We want to design a recursive divide-and-conquer algorithm because we want to avoid/reduce cache misses due to unstructured/random accesses to A.

[6] and $\mathcal{O}(\log n)$ stack space for recursion

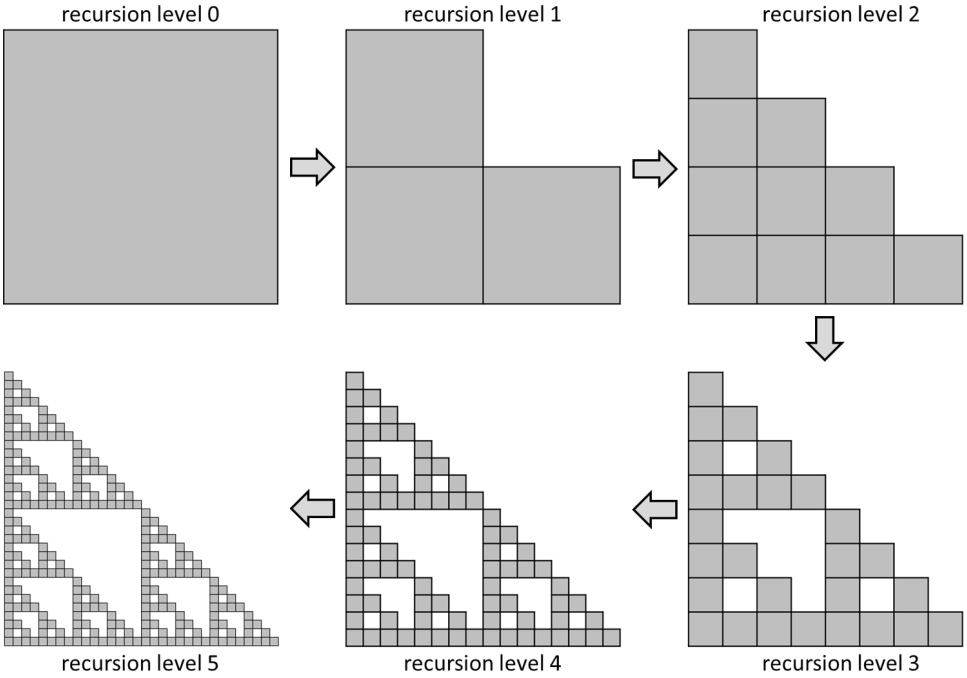

Figure 6: [Task 4] The Sierpinski triangle generated by the algorithm given in Figure 7.

```
SIERPINSKI( A, i, j, n )          {Fill A[i : i + n − 1][j : j + n − 1],
                                   where n is a power of 2}
1. if n = 1 then
2.     color A[i][j] with gray
3. else
4.     SIERPINSKI( A, i, j, n/2 )                    {top-left quadrant}
5.     SIERPINSKI( A, i + n/2, j, n/2 )              {bottom-left quadrant}
6.     SIERPINSKI( A, i + n/2, j + n/2, n/2 )        {top-right quadrant}
7. return
```

Figure 7: [Task 4] Generating the Sierpinski triangle (see Figure 6) in a square array $A[0 : n-1][0 : n-1]$, where $n = 2^k$ for some integer $k \geq 0$. The initial function call is SIERPINSKI($A,\ 0,\ 0,\ n$) on an empty array A.

triangle[7]. Figure 6 shows how the object looks like at various levels of recursion. The running time $T(n)$ of the algorithm for an input of size n, where $n = 2^k$ for some integer $k \geq 0$, is given by the following recurrence.

[7]check https://en.wikipedia.org/wiki/Sierpinski_triangle

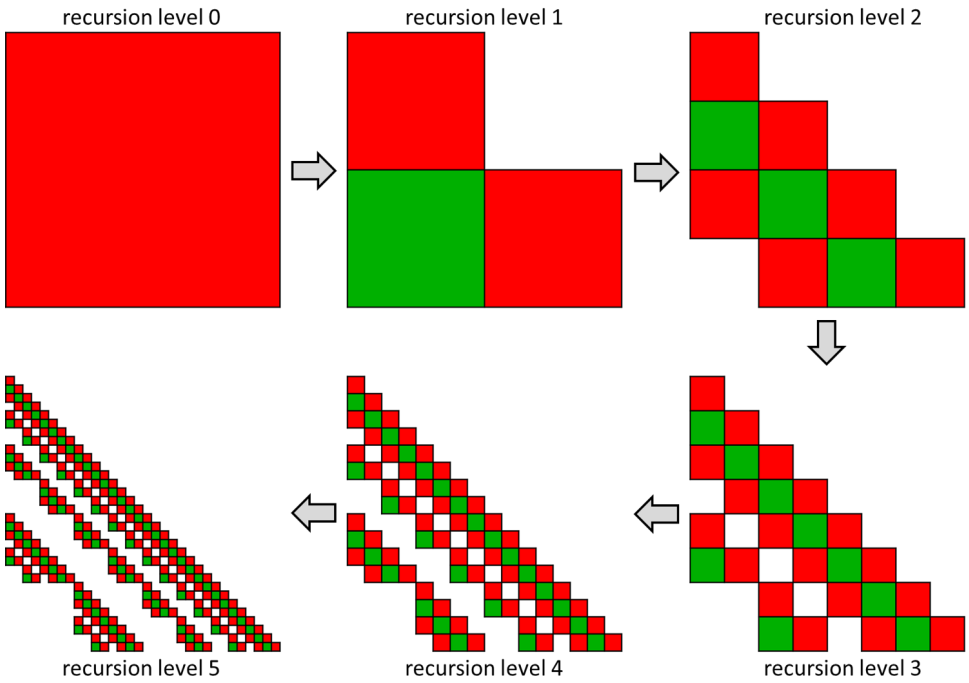

Figure 8: [Task 4] A red-green fractal pattern generated by the algorithm given in Figure 9.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 3T\left(\frac{n}{2}\right) + \mathcal{O}(1) & \text{otherwise.} \end{cases}$$

Using the Master Theorem, $T(n) = \Theta\left(n^{\log_2 3}\right)$.

(*a*) [**10 Points**] Figure 9 shows two mutually recursive functions RED and GREEN that generate the red-green fractal pattern shown in Figure 8 when called as RED(A, 0, 0, n), where n is a non-negative integral power of two. Analyze the running time of this algorithm.

(*b*) [**30 Points**] The following set of recurrences gives the running time of an algorithm for generating a more complicated Fractal structure.

$$T_A(n) = \begin{cases} \Theta(1) & \text{if } n \leq 4, \\ 5T_A\left(\frac{n}{2}\right) + 3T_A\left(\frac{n}{4}\right) + T_B\left(\frac{n}{4}\right) + 2T_C\left(\frac{n}{4}\right) + T_D(n) + \Theta(1) & \text{otherwise.} \end{cases}$$

$$T_B(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T_B\left(\frac{n}{2}\right) + 4T_C\left(\frac{n}{2}\right) + \Theta(1) & \text{otherwise.} \end{cases}$$

$$T_C(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T_B\left(\frac{n}{2}\right) + 3T_C\left(\frac{n}{2}\right) + \Theta\left(n^2\right) & \text{otherwise.} \end{cases}$$

$$T_D(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 3T_D\left(\frac{n}{2}\right) + T_E\left(\frac{n}{2}\right) + \Theta(n) & \text{otherwise.} \end{cases}$$

```
RED( A, i, j, n )                          {Fill A[i : i + n − 1][j : j + n − 1],
                                            where n is a power of 2}
 1. if n = 1 then
 2.     color A[i][j] with red
 3. else
 4.     RED( A, i, j, n/2 )                 {top-left quadrant}
 5.     GREEN( A, i + n/2, j, n/2 )         {bottom-right quadrant}
 6.     RED( A, i + n/2, j + n/2, n/2 )     {bottom-right quadrant}
 7. return
```

```
GREEN( A, i, j, n )                        {Fill A[i : i + n − 1][j : j + n − 1],
                                            where n is a power of 2}
 1. if n = 1 then
 2.     color A[i][j] with green
 3. else
 4.     RED( A, i, j, n/2 )                 {top-left quadrant}
 5.     GREEN( A, i, j + n/2, n/2 )         {top-right quadrant}
 6.     RED( A, i + n/2, j + n/2, n/2 )     {bottom-right quadrant}
 7. return
```

Figure 9: [Task 4] Generating the Red-Green fractal pattern shown in Figure 8 in a square array $A[0 : n-1][0 : n-1]$, where $n = 2^k$ for some integer $k \geq 0$. The initial function call is RED(A, 0, 0, n) on an empty array A.

$$T_E(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 4T_E\left(\frac{n}{2}\right) + \Theta\left(n^2\right) & \text{otherwise.} \end{cases}$$

Solve for $T_A(n)$ assuming $n = 2^k$ for some integer $k > 1$.