-1

تجزیه بر اساس داده ورودی: تعداد داده‌های ورودی را تقسیم بر تعداد نخ‌ها می‌کنیم و هر batch از داده را به یک نخ assign می‌کنیم. در نهایت با join کردن نخ‌ها و جمع کردن داده‌ی خروجی هر کدام از آن‌ها هیستوگرام نهایی را حساب می‌کنیم. نیازی به Syncing و Balancing نداریم.

```
Float[] data, Float[][] limits, int threadCount, Float[] count;
func processData(Float[] data, int start, int end) {
int limitCounter = 0;
  for (i =start; i < end; i++)
      if (data[i] > limits[limitCounter][0) && data[i] < limits[limitCounter][1])
            count[threadNumber]++;
      else if (data[i] > limits[limitCounter][1])
            limitCounter++;
}
Int batchSize = data.Size / threadCount;
for (i =0; I < data.Size; i += batchSize, threadCount++)
        new Thread(processData, data, i, i + batchSize);
```

تجزیه بر اساس داده خروجی: به تعداد رنج‌ها نخ تولید می‌کنیم، همه‌ی داده‌ها را به همه‌ی نخ‌ها می‌دهیم و به هر نخ یک رنج assign می‌کنیم. همه‌ی نخ‌ها همه‌ی دیتا را بررسی می‌کنند و یک خانه از آرایه‌ی شمارنده‌ها را آپدیت می‌کنند.

```
Float[] data, Float[][] limits, int threadCount, Float[] count;
func processData(Float[] data, int start, int end, int threadNumber) {
  for (i =start; i < end; i++)
      if (data[i] > limits[threadNumber][0) && data[i] < limits[threadNumber][1])
            count[threadNumber]++;
}
Int batchSize = data.Size / threadCount; threadCount = 0;
for (i =0; I < data.Size; i += batchSize, threadCount++)
        new Thread(processData, data, i, i + batchSize, threadCount);
```

نیازی به Syncing و Balancing نداریم.

-2

فرض می‌کنیم از یک ند ریشه شروع می‌کنیم که تعداد فرزند دارد، به صورت bfs درخت را پیمایش می‌کنیم و به ازای هر ند جدید پیدا شده یک نخ تولید می‌کنیم تا آن ند را بررسی کند. (در انتهای هر bfs interation نخ فعلی خود مسئول آخرین ند پیدا شده می شود تا نخ بیکار نداشته باشیم) هر نخ در ابتدا بررسی می‌کند که ند فعلی آن برگ است یا خیر، در صورت برگ بودن چک می‌کند آیا جواب پازل است یا خیر. اگر جواب بود سیگنالی به سیستم می فرستد تا بقیه‌ی نخ‌ها را از پیدا شدن جواب مطلع کند تا فعالیت آن‌ها متوقف شود.

```
Func processNode(node* current) {
  foreach (child in current.children) {
       if (child.children.length == 0) {
          if (child.value == puzzleAnswer) {
                signal (“Answer found at node” + current)
                return;
```

```
        }
        else {
           new Thread (processNode, child);
        }

    }
new Thread(root);
```

نیاز به Sync نداریم ولی Balancing چرا.

-3

روش سطری: هر سطر از ماتریس اول و ستون مرتبط از ماتریس دوم به یک نخ داده می‌شود تا با ضرب اعداد آن و جمع کردن نتیجه، درایه‌ی مربوط در ماتریس پاسخ را حساب کند.
نیازی به Syncing و Balancing نداریم.

```
Float elementResult(Float[] row1, Float[] column1) {
  float ans = 0;
  for (i=0; i<row1.length; i++) { ans +=  row1[i] * column1[i] };
  return ans;
for (i =0; i<matrix1.rows.length; i++)
     for (int j=0; i<matrix2.columns.length; j++)
       answerMatrix[i][j] = new Thread(elementResult, matrix1.rows[i], matrix2.columns[j]);
```

روش دو بعدی:
ماتریس را به ماتریس‌های مربعی کوچک‌تر می‌شکنیم و هر نخ ماتریس‌های مربعی متناظر هر دو ماتریس را گرفته و ضرب‌های آن‌ها را محاسبه می‌کند، سپس این نخ‌ها با ارسال پیام از یک‌دیگر درخواست حاصل‌ضرب‌های ناموجود ماتریس متناظرشان را از ماتریس‌های دیگر می‌کنند و نهایتا جمع‌های لازمه را انجام می‌دهند.
نیاز به Syncing داریم ولی Balancing خیر.

-4

چون می‌دانیم فرمول کلی دنباله فیبوناچی به صورت

$$F_n = \frac{1}{\sqrt{5}} \cdot \left(\frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \cdot \left(\frac{1-\sqrt{5}}{2} \right)^n$$

.

است می‌توان با assign کردن nهای متفاوت به نخ‌ها به صورت موازی این دنباله را محاسبه کرد.
نیازی به Syncing و Balancing نداریم.

```
Float[] results;
func computeNthFibo(int n) {…}
for (i=0; i<results.length; i++) { results[i] = new Thread(computeNthFibo, i);
```

-5

وظیفه‌ی محاسبه‌ی مقدار هر نقطه را به یک نخ می‌دهیم و برای همه‌ی نقاط یک مقدار اولیه در نظر می‌گیریم، سپس به همه‌ی نخ‌ها دستور می‌دهیم تا با ارسال پیام به نخ‌های مجاور و دریافت مقدار نقاط اطراف، مقدار جدید نقطه‌ی خودشان را حساب کنند.
نیاز به Syncing داریم، نیاز به Balancing نداریم.