DS 2^{nd} homework

58121102 Jiang Yuchu

30 September 2022

P$_{93}$2. If $a = (a_0, \cdots, a_{n-1})$, and $b = (b_0, \cdots, b_{m-1})$ are ordered lists, then $a < b$ if $a_i = b_i$ for $0 \leq i < j$ and $a_j < b_j$, or, if $a_i = b_i$ for $0 \leq i < n$ and $n < m$. Write a function that returns -1, 0 or +1, depending upon whether $a < b$, $a = b$, or $a > b$. Assume that the a_is and b_js are integer.

Answer:

```
template <size_t N, size_t M>
int arrayCompare(int (&a)[N], int (&b)[M]) {
    for (int i = 0;; ++i) {
        if (i == N)
            return i == M ? 0 : -1;
        if (i == M)
            return 1;
        if (a[i] < b[i])
            return -1;
        if (a[i] > b[i])
            return 1;
    }
}
```

P$_{93}$6. Write C++ Functions that evaluates a polynomial at a value x_0 using the representation of this section. Try to minimize the number of operations.

Answer:

```
double evaluate(double x0) const { //in class Polynomial
    double res = 0.0;
    for (int i = 0; i < terms; ++i)
        res += pow(x0, termArray[i].exp) * termArray[i].coef;
    return res;
}
```

In C++20, it can also be completed in a single line.

```
double evaluate(double x0) const {
    return std::accumulate(termArray, termArray + terms, 0.0,
        [=](double sum, const auto& term) {
            return sum + pow(x0, term.exp) * term.coef;
        });
}
```

P$_{94}$9.

Answer:

Compare the representation of this exercise and the representation mentioned before, here are the advantages and drawbacks of former.

Advantages:

- Less memory allocation, which can be more effective in creating large amounts polynomials and storing constant data.

Drawbacks:

- Hard to modify. When the number of terms is increased, it is very expensive to move all the following data forward, so new polynomials can only be added at the end, which will lead to greater memory overhead. When reducing the number of terms of a polynomial, if you adjust a.finish, it will lead to the appearance of memory fragmentation.
- More complex memory management. In order to solve the above-mentioned problems, more sophisticated memory management algorithms are required, which increases the complexity of programming.

P$_{107}$1. How much time does it take to locate an arbitrary element $A[i][j]$ in the representation of this section and to change its value?

Answer:

The code is as follows:

```
int locate(int i, int j) const { //in class SparseMatrix
    double res = 0.0;
    for (int n = 0; i < terms; ++n) {
        if (smArray[n].row == i && smArray[n].col == j)
            return sm[n].value;
    }
    return NaN;
}
```

Let *terms* denote the number of terms. It will take $O(terms)$ to locate an element.

P$_{107}$2. Analyze carefully the computing time and storage requirements of function *Fast-Transpose* (Program 2.11). What can you say about the existence of an even faster algorithm?

Answer:

The fast transpose function achieves its efficiency through the use of two auxiliary arrays both indexed by the number of columns in the original matrix. Array rowSize holds the number of columns in the original matrix; array rowStart holds the starting position of the column, which will become the row, in the new matrix. Using the example from the text, smArray, rowStart, and rowSize will hold the following values:

	row	col	value
smArray[0]	0	0	15
[1]	0	3	22
[2]	0	5	-15
[3]	1	1	11
[4]	2	2	3
[5]	2	3	-6
[6]	2	0	91
[7]	2	2	28

	rowSize	rowStart
[0]	3	0
[1]	2	3
[2]	1	5
[3]	0	6
[4]	1	6
[5]	1	7

Reading down the c column of a verifies that column 2 contains two entries. By accumulating the entries in rowTerms, we obtain the starting position for each row in the transposed matrix.

As the book indicates, the computing time is $O(n+t)$ for sparse matrices and $O(mn)$ for dense ones. Improving the efficiency of transposition requires a data structure that directly accesses the columns of a matrix.

P$_{107}$4. Rewrite function *FastTranspose* (Program 2.11) so that it uses only one array rather than the two arrays required to hold *RowSize* and *RowStart*.

Answer:

```
SparseMatrix SparseMatrix::FastTranspos() {
    SparseMatrix b(cols, rows, terms);
    if (terms > 0) {
        int* rowInfo = new int[cols];
        fill(rowInfo, rowInfo + cols, 0);
        for (int i = 0; i < terms; ++i)
            rowInfo[smArray[i].col]++;
        for (int i = 1; i < cols - 1; ++i)
            rowInfo[i] += rowInfo[i - 1];
        move_backward(rowInfo, rowInfo + col - 1, rowInfo + col);
        rowInfo[0] = 0;
        for (int i = 0; i < terms; ++i) {
            int j = rowInfo[smArray[i].col];
            b.smArray[j].row = smArray[i].col;
            b.smArray[j].col = smArray[i].row;
            b.smArray[j].value = smArray[i].value;
            rowInfo[smArray[i].col]++;
        }
        delete[] rowInfo;
    }
    return b;
}
```

P$_{108}$7.

(a) On a computer with w bits per word, how much storage is needed to represent an $n \times m$ sparse matrix that has t nonzero terms?

Answer:

Assume the size of each value is a words, then this kind of representation will takes $\lceil \frac{n \times m}{w} \rceil + a \cdot t$ words.

(b) Write an algorithm to add two sparse matrices represented as above. How much time does your algorithm take?

Answer:

```
SparseMatrix operator+=(const SparseMatrix& smat) {
    if (rowSize() != smat.rowSize() || colSize() != smat.colSize())
        throw invalid_argument("add operation requires the same size");
    size_t value_index = 0, smat_index = 0;
    for (size_t r = 0; r < smat.rowSize(); ++r) {
        for (size_t c = 0; c < smat.colSize(); ++c) {
            if (smat.mat.at(r, c)) {
                if (mat.at(r, c))
                    values[value_index] += smat.values[smat_index];
                else {
                    values.insert(
```

```
12                          values.begin() + value_index,
13                          smat.values[smat_index]);
14                      mat.at(r, c) = true;
15                  }
16                  smat_index++, value_index++;
17              }
18              else if (mat.at(r, c))
19                  value_index++;
20          }
21      }
22      return *this;
23  }
```

Let *smat* denote matrix to be added to *this. It takes $O(smat.row \times smat.col)$ time to complete add operation.

(c) Discuss the merits of this representation versus the representation of Section 2.4. Consider space and time requirements for such operations as random access, add, multiply, and transpose.

Answer:

I don't want to take the advice given in the title, because I came up with another way to speed up random access. Maintaining the array *ra* will cause more unnecessary troubles, so the following discussion is based on my own design code (read p108-9.cpp for details). Besides, I call the representation given by exercise "bits array way",and call the representation given before "ordinary way"

	ordinary way	bits array
random access	$O(terms)$	$O(row \times col)$($O(col)$if use ra)
add($a+b$)	$O(a.terms \times b.terms)$	$O(b.col \times b.row)$
multiply($a \times b$)	$O(b.cols \times a.terms + a.rows \times b.terms)$	$O(a.row \times a.col \times b.col)$
transpose	$O(terms + col)$	$O(row \times col)$

Time complexity comparison

About space occupation, let a,b denote the size of each value and the size of index, respectively. Then, ordinary way takes $terms \times (2b + a)$, and bits array way representation takes $\lceil \frac{n \times m}{w} \rceil + a \cdot terms$. If maintain array *ra*, it will takes $\lceil \frac{n \times m}{w} \rceil + a \cdot terms + b \cdot row$

	ordinary way	bits array
random access	$O(1)$	$O(1)$($O(row)$if use ra)
add($a+b$)	$O(1)$	$O(1)$
multiply($a \times b$)	$O(row \times col)$	$O(1)$
transpose	$O(col)$	$O1)$

Space complexity comparison

Compare:

- The ordinary way is harder to implement.
- The ordinary way has better time complexity
- The bit array way takes up less space when the data is relatively dense.

P$_{118}$1. Write a function *String::Frequency* that determines the frequency of occurrence of each of the distinct characters in the string. Test your function using suitable data.

Answer:

function:

```
map<char, int> Frequency() {  // in class String
    map<char, int> m;
    for (int i = 0; i < Length(); ++i)
        m[str[i]]++;
    return m;
}
```

test:

```
for(auto& item : Frequency()) {// in main
    cout << item.first << "␣" << item.second << endl;
}
```

P$_{119}$7. Compute the failure function for each of the following patterns:

(a) a a a a b

Failure function: -1 0 1 2 -1

(b) a b a b a a

Failure function: -1 -1 0 1 2 0

(c) a b a a b a a b b

Failure function: -1 -1 0 0 1 2 3 4 -1

P$_{119}$9. The definition of the function may be strengthened to

$$f(j) = \begin{cases} \textbf{largest } i < j \quad \textbf{such that } p_0 \cdots p_i = p_{j-i} \cdots p_j \textbf{ and } p_{i+1} \neq p_{j+1} \\ -1 \qquad\qquad\qquad\quad \textbf{if there is no } i \geq 0 \textbf{ satisfying above} \end{cases}$$

(a) Obtain the new failure function for the pattern *pat* of the text.

Answer:

pat	a	b	c	a	b	c	a	c	a	b
failure function	-1	-1	-1	-1	-1	-1	3	-1	-1	-1

(b) Show that if this definition for f is used, then algorithm *FastFind*(Program 2.16) still works correctly.

Answer:

① if $p_{i+1} \neq p_{j+1}$

This case is same as ordinary KMP algorithm, new definition will do like KMP.

② if $p_{i+1} = p_{j+1}$

In Figure 1, let A, b, c denote the largest common string, the character to be matched in the *pattern* and the character mismatched in the *string*, respectively. According to the new definition, since $p_{i+1} = p_{j+1}$ does not satisfy the strengthened condition, the next match will start from the beginning of the *pattern*. So now our question is, could there be an answer we missed in the skipped part of the *string*?

Use proof by contradiction.

Let's first assume that the above situation really exists. Because the substring in the red box matches the *string*, it is easy to complete some longest common prefixes and suffixes A in the *assuming pattern* and *string*. After completion, it is as shown in the Figure 2, and we call the part between the two A's of the *string* as B.

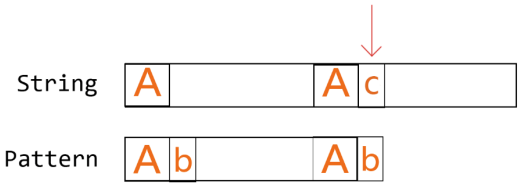


Figure 1

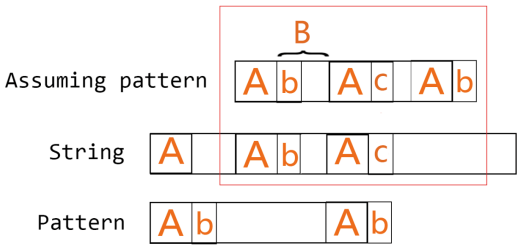


Figure 2

Now, all the strings look as shown in Figure 3(*assuming pattern* and *pattern* are the same). Parts of the *string* can also be inferred since the previous part of the *string* was matched before the *pattern* was moved. Again according to the *string* information, as shown in Figure 4, we get the final *pattern*.

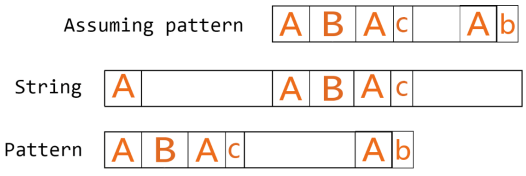


Figure 3

Figure 4

Let's examine the final *pattern*. The substring ABA obviously satisfies $p_0 \cdots p_i = p_{j-i} \cdots p_j$, however, $b \neq c$, which means $p_{i+1} \neq p_{j+1}$. This contradicts the previous assumption.

To sum up, it is impossible to have matched substrings in the newly defined pattern string skipping interval.

(c) Modify algorithm *FailureFunction* (Program 2.17) to compute f under this definition. Show that the computing time is still $O(m)$.

Answer:

```
1  void FailureFunction() {  // in class String
2      f[0  = -1;
3      auto old_f = f;
4      for (int j = 1; j < str.length(); ++j) {
5          int i = old_f[j - 1];
6          while (str[j] != str[i + 1] && i >= 0)
7              i = old_f[i];
8          if (str[j] == str[i + 1]) {
9              old_f[j] = i + 1;
10             f[j] = str[i + 2] != str[j + 1] ? i + 1 : -1;
11         }
12         else
13             old_f[j] = f[j] = -1;
14     }
15 }
```

Compared with the previous algorithm, the new definition only adds few conditional statements, so the same method can also be used to analyze its complexity, which is still $O(m)$.

(d) Are there any patterns for which the observed computing time of *FastFind* is more with the new definition of f than with the old one? Are there any of which it is less? Give examples.

Answer:

The advantage of the new definition is that it is faster to match directly from the beginning when the pattern string repeats are long. Consider an extreme case, for example, the pattern string is *"aaaa...."*. When comparing, the original definition needs to be moved forward one by one, and the new definition can be moved directly to the beginning, which greatly saves time.

Experiment

P_{123}8.

1 Experiment target

1.1 Background

A (drunken) cockroach is placed on a given square in the middle of a tile floor in a rectangular room of size $n \times m$ tiles. The bug wanders randomly from tile to tile throughout the room. Assuming that he may move from his present tile to any of the eight tiles surrounding him (unless he is against a wall) with equal-probability, how long will it take him to touch every tile on the floor at least once?

1.2 Goals

(a) handle all values of n and m,$2 < n \leq 40$,$2 \leq m \leq 20$

(b) perform the experiment for (1) $n = 15, m = 15$,starting point: $(9, 9)$ and (2) $n = 39, m = 19$, starting point: $(0, 0)$

(c) have an iteration limit,that is,a maximum number of squares the bug may enter during the experiment (this avoids getting hung in an infinite loop);a maximum of 50,000 is appropriate for this exercise

(d) for each experiment,print (1)the total number of legal moves that the cockroach makes and (2)the final count array (this will show the density of the walk,that is,the number of times each tile on the floor was touched during the experiment).

2 Basic idea

An $n \times m$ array *count* is used to represent the number of times our cock-roach has reached each tile on the floor.All the cells of this array are initialized to zero.The position of the bug on the floor is represented by the coordinates $(ibug, jbug)$.The eight possible moves of the bug are represented by the tiles located at $(ibug + imove[k], jbug + jmove[k])$, where $0 \leq k \leq 7$ and

$$\begin{aligned}
imove[0] &= -1 & jmove[0] &= 1 \\
imove[1] &= 0 & jmove[1] &= 1 \\
imove[2] &= 1 & jmove[2] &= 1 \\
imove[3] &= 1 & jmove[3] &= 0 \\
imove[4] &= 1 & jmove[4] &= -1 \\
imove[5] &= 0 & jmove[5] &= -1 \\
imove[6] &= -1 & jmove[6] &= -1 \\
imove[7] &= -1 & jmove[7] &= 0
\end{aligned}$$

A random walk to one of the eight given squares is simulated by generating a random value for k lying between 0 and 7. Of course, the bug cannot move outside the room,so coordinates that lead up a wall must be ignored and a new random combination formed.Each time a square is entered,the count for that square is incremented so that a nonzero entry shows the number of times the bug has landed on that square so far.When every square has been entered at least once,the experiment is complete.

3 Experimental procedure

1. Enter n and m, then initialize the square.

```
vector<vector<int>> count;
int m, n;
cout << "input␣m,␣n" << endl;
cin >> m >> n;
count.resize(m);
for_each(count.begin(), count.end(), [&](auto& v) {
    v.resize(n, 0);
});
```

2. Enter starting point (x, y), then start iteration. Meanwhile, check the validation of generating result, and control the number of iteration. In the code snippet `hasTraveledAll(int)` is used to check whether the bug has traveled the whole square.

```
int x, y;
cout << "input␣x,␣y" << endl;
cin >> x >> y;
random_device                rd;
mt19937                      mt(rd());
uniform_int_distribution<> ud(0, 7);

int cnt = 0, i;
for (i = 0; i < 5e4 && !hasTraveledAll(count); i++) {
    int index = ud(mt);
    int tempx = x + imove[index];
    int tempy = y + jmove[index];
    if (tempx >= 0 && tempx < m && tempy >= 0 && tempy < n) {
        count[x = tempx][y = tempy]++;
        cnt++;
    }
}
```

3. Output the result to console and .csv file, then generate 3d graph via python.

4 Results and analysis

In the submitted file, p128-8.cpp shows the C++ code.

1. *data.csv* shows the generated data when m, n = 15, 15 and start from (9, 9).
2. *data2.csv* show the generated data when m, n = 39, 19 and start from (0, 0).

Here are some running results. In the left figure 5, it shows the output of console. In the right figure 6, it shows three columns of data, representing the number of arrivals for the row, column, and corresponding cell.

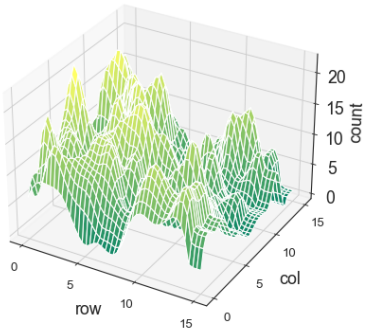
```
input m, n
15 15
input x, y
9 9
number of motion: 3455
  5  7  9  5  4  5 15 18 17 14  5  7 17 13  4
 10 24 20 17  9 17 13 26 17 11 16 15 14 18 14
 21 30 26 23 28 17 23 20 12 14 24 16 18 18  8
 12 28 36 18 26 17 23 15 19 13 18 12 12 15  6
 20 32 34 26 19 28 21 23 18 19 14  9  9 14  3
 20 33 32 25 20  9 18 22 22 19 20 11 12  7  5
 12 15 31 24 15  8 16 23 18 13 15 15 15 13  6
 11 26 20 16 19 16 18 14 14 17 16  9  9  9  2
 12 19 20 18 21 17 10 10 14 12 11 14  9  1  6
 13 26 16 14 16 10 14 14 17 13 11  8  7  7  5
 22 26 27 22 17 16  9 15 13 13  8 10  6  3  4
 20 44 29 20 25 22 21 17 13 14 10  1  4  1  1
 18 36 23 20 20 25 24 15 13 11 13 14  5  3  1
 27 34 25 28 24 20 16 12 10 10  6 13  9 11  5
 19 20 17 19 13  8  8  7  4  3  7  9  6  6  7
```

Figure 5: Console data from case (1)

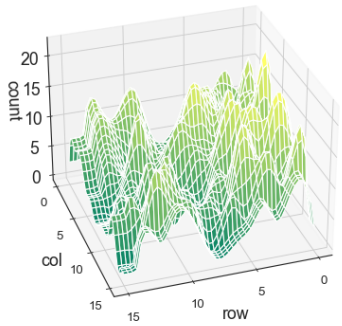
row	col	count
0	0	6
0	1	9
0	2	9
0	3	8
0	4	4
0	5	1
0	6	2
0	7	1
0	8	2
0	9	7
0	10	9
0	11	8
0	12	9
0	13	10
0	14	3
1	0	9

Figure 6: Some data from case (1)

I also drew the graph of each case via python.

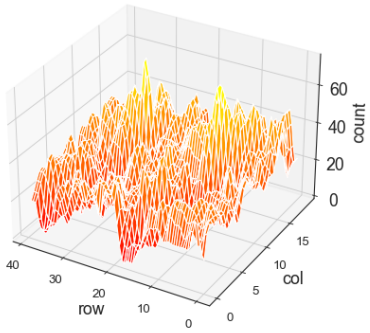


(a)

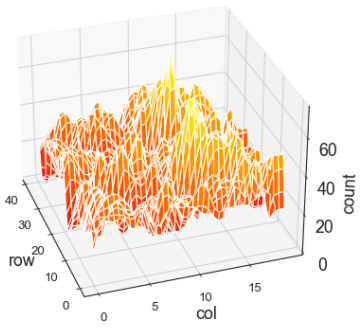


(b)

m, n = 15, 15 and start from (9, 9)



(c)



(d)

m, n = 39, 19 and start from (0, 0)