CSC3100 Assignment 1 Report

Shuqi Ke

November 3, 2020

Contents

Chapter 1

Problem 1: Merge Sort

1.1 Possible Solutions

1.1.1 Problem analysis

The problem is to sort some numbers. The input scale is up to 500000 numbers.

For typical sorting problem, we can apply different kinds of algorithms:

- $O(n^2)$ algorithms: selection sort, insertion sort. They require much more time and are not efficient enough for this problem.
- $O(n \log n)$ algorithms: merge sort
- Quick sort (Average performance $O(nlogn)$)
- Bucket sort, counting sort: the data is in long long type of c++. Since the data is big, bucket sort and counting sort are not suitable for this problem.

However, the problem requires us to sort the numbers by implementing merge sort. Therefore, the proper solution is merge sort.

1.1.2 Description of possible solution

Merge sort is a divide-and-conquer algorithm. It generally works as follows:

1. For the current array, equally divide it into two subarrays by its midpoint.
2. Recursively sort these two subarrays.
3. Merge the two sorted subarrays.

Merge sort has some advantages. It is stable and works in $\Theta(nlogn)$ time.
There are four possible implementations for merge sort. If we consider the algorithmic implementation, we can implement merge sort either **Top-down** or **Buttom-up**. If we consider the data structure implementation, we can implement merge sort with either **array** or **linked list**

1.2 My Solution

I use buttom-up implementation with three optimizations:

- stable $\Theta(n)$ working memory
- insertion sort in small cases
- faster input/output implementation

1.2.1 Buttom-up implementation

In divide and conquer, instead of calling recursive function every time, we enumerate the length of subarrays from length 1 to A.length. Merge from small subarrays to bigger subarrays.

1.2.2 Stable $\Theta(n)$ working memory

In the above pseudo-code, I use array B[1,...,n] as a temporary memory for buttom-up merge. Using only A[1,...,n] and B[1,...,n] throughout the whole algorithm, my solution has stable $\Theta(n)$ working memory.

Algorithm 1 Bottum Up Merge Sort

```
1: function ButtomUpMergeSort(l,r,A[ ],B[ ])          ▷ A, B are arrays. Their index starts from 1
2:     width ← 1
3:     while width <= A.length do                      ▷ Loop until the maximum length
4:         i ← 0
5:         while i <= A.length do
6:             BottumUpMerge(i,min{i + width, A.length + 1},min{i + 2 * width, A.length + 1},A,B)
7:         end while
8:         CopyArray(B, A)
9:     end while
10: end function
11: function ButtomUpMerge(l,mid,r,A[ ],B[ ])
12:     Let B be a new array
13:     i ← l, j ← mid, k ← l
14:     while k < r do
15:         if i < mid and (j >= r || A[i] <= A[j]) then
16:             B[k] = A[i];
17:             i ← i + 1
18:         else
19:             B[k] = A[j];
20:             j ← j + 1
21:         end if
22:     end while
23: end function
24: function CopyArray(B[ ],A[ ])
25:     A ← B
26: end function
```

1.2.3 Faster input/output implementation

This optimization is essential for this problem because the data amount is large. As described in the problem, the total number $N \leq 500000$. Inputing and outputing data take $\Theta(n)$ time. For Java implementation, I will use bufferedreader instead of Scanner. For c++ implementation, I will use scanf or getchar, printf or putchar instead of cin and cout.

1.2.4 Source Code

Listing 1.1: Problem 1 c++ source code

```
1  #include <cstdio>
2  #include <cstring>
3  #include <iostream>
4
5  long long arr[2][500011];
6  int n, cur = 0;
7
8  void merge(int left, int mid, int right) {
9      int i = left, j = mid, ind = left - 1;
10     while (i < mid && j < right) {
11         if (arr[cur][i] <= arr[cur][j])
12             arr[cur^1][++ind] = arr[cur][i++];
13         else
14             arr[cur^1][++ind] = arr[cur][j++];
15     }
16     while (i < mid)
17         arr[cur^1][++ind] = arr[cur][i++];
```

```
18      while (j < right)
19          arr[cur^1][++ind] = arr[cur][j++];
20  }
21
22  void buttom_up_merge_sort() {
23      for (int k = 1; k <= n; k <<= 1) {
24          for (int i = 1; i <= n; i += (k << 1))
25              merge(i, std::min(i + k, n + 1), std::min(i + (k << 1), n + 1));
26          cur ^= 1;
27      }
28  }
29
30  int main() {
31      scanf("%d", &n);
32      for (int i = 1; i <= n; ++i)
33          scanf("%lld", &arr[cur][i]);
34      buttom_up_merge_sort();
35      for (int i = 1; i <= n; ++i)
36          printf("%lld\n", arr[cur][i]);
37      return 0;
38  }
```

1.3 Comparison with other methods

1.3.1 Advantages of buttom-up implementation

Let's compare buttom-up implementation with top-down implementation. Recursive function calls take extra memory in system's stack. When a program calls a function, that function goes on top of the call stack. When the amount of input data increases, the stack usage will also increase. Therefore, buttom-up loop implementation is more efficient than top-down recursive implementation in practice.

Moreover, buttom-up loop implementation is precise and elegant in implementation. The code length can even be shorter than the recursive implementation. Hence, buttom-up loop implementation is better than top-down recursive implementation.

1.3.2 Advantages of stable $\Theta(n)$ working memory

We can analyze my solution's space complexity with other implementations. Some merge sort implementations create two arrays for temporary storage in the Merge function. If the program does not free this extra memory immediately, it may result in redundant memory use.

We can also compare the time complexity in merge step. Some merge sort implementations copy the data from A array to two new arrays in merge step. After during merge step, they copy the data from these two arrays back to the original array. We can see this in the following implementation pseudo-code

Such copy processes waste much time. In my solution, the copy process only proceed when all the subarrays (of same length) are sorted. Therefore, my solution is more efficient in merge step.

1.3.3 Advantages of faster input/output implementation

The input is considerably big. With faster input/output(IO) implementation, the time cost in IO is significantly reduced. If we do not use faster IO implementation, in some large test point on the Online Judge, the program will exceed the time limit.

Algorithm 2 Redundant Merge Implementation (bad solution)

```
1: function MERGE(A[ ],p,q,r)
2:     n1 = q-p+1,n2 = r-q Let L[1,...,n1+1] and R[1,...,n2+1] be new arrays
3:     for i=1 to n1 do
4:         L[i]=A[p+i-1]
5:     end for
6:     for j=1 to n2 do
7:         R[i]=A[q+j]
8:     end for
9:     L[n1+1]=∞,R[n2+1]=∞,i = 1,j = 1
10:    for k=p to r do
11:        if L[i]≤R[j] then
12:            A[k]=L[i]
13:            i=i+1
14:        else
15:            A[k]=R[j]
16:            j=j+1
17:        end if
18:    end for
19: end function
```

1.4 Test of my solution

I submitted several solutions to the Online Platform.

On average the top-down implementation takes 3.41s time and with 67.1 MB memory use.

On average the buttom-up implementation takes 1.20s time and with 10.2 MB memory use.

Hence, comparing the time and memory cost performance, I decide to use buttom-up implementation.

1.5 Further Improvements

1.5.1 Merge sort with linked list

We can use linked list in merge sort so that we do not have to use temporary arrays in merge step. If we use linked list we can directly change the order of the two sorted subarrays. And it will be much efficient from theoretic perspective.

1.5.2 Wiki sort

We can also use Block Sort (also known as wiki sort). This algorithm takes both the advantages of insertion sort and merge sort. It is non-recursive and does not require the use of dynamic allocations. In other words, this algorithm does not take extra memory while maintaining to be stable.

Algorithm 3 Block Sort

```
1: function BlockSort(A[ ])
2:     powerTwo=2^floor(log(A.length))
3:     scale=A.length/powerTwo
4:     for m=0 to powerTwo in step of 16 do
5:         InsertionSort(A,m*scale,m*scale+16*scale)
6:     end for
7:     for len=16 to powerTwo, len=len+len do
8:         for m=0 to powerTwo, m=m+len*2 do
9:             start = m * scale
10:            mid = (m + len) * scale
11:            end = (m + len * 2) * scale
12:            if ( thenarray[end − 1] < array[start])
13:                Rotate(A, mid − start, [start, end))
14:            else
15:                if ( thenarray[mid − 1] > array[mid])
16:                    Merge(A, L = [start, mid), R = [mid, end))
17:                end if
18:            end if
19:        end for
20:    end for
21: end function
```

Chapter 2

Problem 2: Prefix Expression

2.1 Possible Solutions

2.1.1 Problem analysis

This problem requires to evaluate a preffix expression.

2.1.2 Description of possible solutions

There are generally two kinds of implementations for this problem: **recursion** and **loop and stack method**.

2.2 My Solution

Use a recursive algorithm for this problem. Every operator is related to later two operands. Therefore, we can make use of this property to design our recursion.

2.2.1 Source Code

Listing 2.1: Problem 2 c++ source code

```
1    #include <cstdio>
```

```cpp
#include <cstdlib>
#include <cstring>
#include <iostream>

using namespace std;

const long long mo = 1e9 + 7;
int n;

inline long long gg() {
    if (!n) {
        puts("Invalid");
        exit (0);
    }
    for(n--;;) {
        char c;
        scanf("%c", &c);
        if (c >= '0' && c <= '9') {
            long long ans = 0;
            for (;c >= '0' && c <= '9'; scanf("%c", &c))
                ans = ((ans << 3) + (ans << 1) + c - '0') % mo;
            return ans % mo;
        }
        if (c == '+' || c == '-' || c == '*') {
            long long lans = gg() % mo, rans = gg() % mo;
            return c == '+'? lans + rans : (c == '-'? lans - rans : lans * rans);
```

```
28              }
29          }
30      }
31
32      int main(int argc, char const *argv[])
33      {
34          scanf("%d", &n);
35          long long ans = gg();
36          if (n)
37              puts("Invalid");
38          else
39              printf("%lld", (ans + mo) % mo);
40          return 0;
41      }
```

2.3 Comparison with other methods

Let's first look at a naive stack implementation source code. It does not use recursive method.

Listing 2.2: Problem 2 bad solution

```
1   #include <cstdio>
2   #include <cstdlib>
3   #include <cstring>
4   #include <iostream>
5
6   using namespace std;
```

```
7
8       const long long mo = 1e9 + 7;
9       char ch[200];
10      int n, top;
11      bool isNum[2000010];
12      long long m[2000010], stack[2000010];
13
14      long long toLongLong(char *p) {
15        long long ans = 0;
16        for (int i = 0; i < strlen(p); ++i) {
17          ans = ans * 10 + p[i] - '0';
18          if (ans > mo) {
19            ans %= mo;
20          }
21        }
22        return ans;
23      }
24
25      int main(int argc, char const *argv[]) {
26        scanf("%d", &n);
27        for (int i = 1; i <= n; ++i) {
28          scanf("%s", ch);
29          switch (ch[0]) {
30          case '+':
31            m[i] = 0;
32            isNum[i] = false;
```

```
33            break;
34          case '-':
35            m[i] = 1;
36            isNum[i] = false;
37            break;
38          case '*':
39            m[i] = 2;
40            isNum[i] = false;
41            break;
42          default:
43            m[i] = toLongLong(ch);
44            isNum[i] = true;
45          }
46        }
47        bool flag = false;
48        for (int i = n; i >= 1; --i) {
49          if (isNum[i]) {
50            stack[++top] = m[i];
51          } else {
52            if (top < 2) {
53              flag = true;
54              break;
55            }
56            long long x = stack[top--];
57            long long y = stack[top--];
58            switch (int(m[i])) {
```

```
59              case 0:
60                stack[++top] = x + y;
61                break;
62              case 1:
63                stack[++top] = x - y;
64                break;
65              case 2:
66                stack[++top] = x * y;
67              }
68              if (stack[top] > mo) {
69                stack[top] %= mo;
70              }
71              if (stack[top] < -mo) {
72                stack[top] = (stack[top] + mo) % mo;
73              }
74            }
75        }
76        if (flag) {
77          puts("Invalid");
78        } else {
79          cout << (stack[top] + mo) % mo << endl;
80        }
81        return 0;
82      }
```

This recursive method compute the result during the input process, while the stack method requires to

input the whole data first and then traverse through the inputed data again. Also the stack method requires the programmer to implement a stack, which result in unprecise and longer code. Therefore, the recursive method is relatively more efficient than the naive stack method.

2.4 Test of my solution

I submitted several solutions to the Online Platform.

On average the stack implementation takes 1.138s time with 20.23 MB memory use.

On average the recursive implementation takes 0.9s time and with 13.3 MB memory use.

Hence, comparing the time and memory cost performance, I decide to use recursion implementation.

2.5 Further Improvements

2.5.1 Faster input/output(IO)

In c++ implementation, we can use fread function. It processes char input much faster.

Listing 2.3: Problem 2 Faster IO

```
1   namespace IO {
2       const int MAXBUF = 1 << 22;
3       inline char flow(){
4           static char B[MAXBUF],*S = B,*T = B;
5           if(S == T){
6               T = (S = B) + fread(B, 1, MAXBUF, stdin);
7               if(S == T)
8                   return 0;
9           }
10          return *S++;
```

```
11          }
12          // Faster reader for int-type
13          inline void Readin(int &x){
14              x = 0;
15              static char c;
16              for(; !isdigit(c = flow()););
17              for(x=c-48;isdigit(c=flow());x=(x<<1)+(x<<3)+c-48);
18          }
19          // Faster reader for long-long-type
20          inline void Readin(long long &x){
21              x = 0;
22              static char c;
23              for(; !isdigit(c = flow()););
24              for(x = c - 48; isdigit(c = flow()); x = (x << 1) + (x << 3) + c - 48);
25          }
26          // Faster writer for long-long-type
27          inline void Printout(long long x){
28              static char pb[101];
29              int top=0;
30              while(x) {
31                  pb[++top] = (x % 10LL) + '0';
32                  x /= 10LL;
33              }
34              while(top)
35                  putchar(pb[top--]);
36              putchar('\n');
```

```
37              }
38          }
```

References

- Wikipedia: Block sort
- ”Introduction to algorithms” by Thomas H.Cormen, Charles E.