please grade

CSE 548 – Analysis of Algorithms, Spring 2013

Assignment #2

Duo Xu (#108662210)

partner: Yu-Yao Lin (#109090793)

Problem 6

(1) The standard merge sort uses $O(\frac{N}{B}\log\frac{N}{M})$ memory transfers:

First, the N numbers are split to N/M small chunks, each with size M (except the last chunk).

Then, we do merge sort. Every time we merge two chunks, loading a block of each chunk, so in one interation we need to load all the blocks once, which is N/B memory transfers.

We do this kind of iteration again and again, the chunks becoming bigger and bigger, until the N numbers are sorted in one chunk. The total number of iternation is $\log\frac{N}{M}$.

Thus, the standard merge sort uses $O(\frac{N}{B}\log\frac{N}{M})$ memory transfers.

The only differences from the standard merge sort are:

(1) The N numbers are split to N/B small chunks, each with size B (except the last chunk).

(2) Every time we merge M/B blocks instead of two. When one of the blocks has been fully merged (name it as A), we write the first merged block into disk, and then load the following block which is in the same chunk with A into memory.

In this way, we can sort the numbers in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ memory transfers.

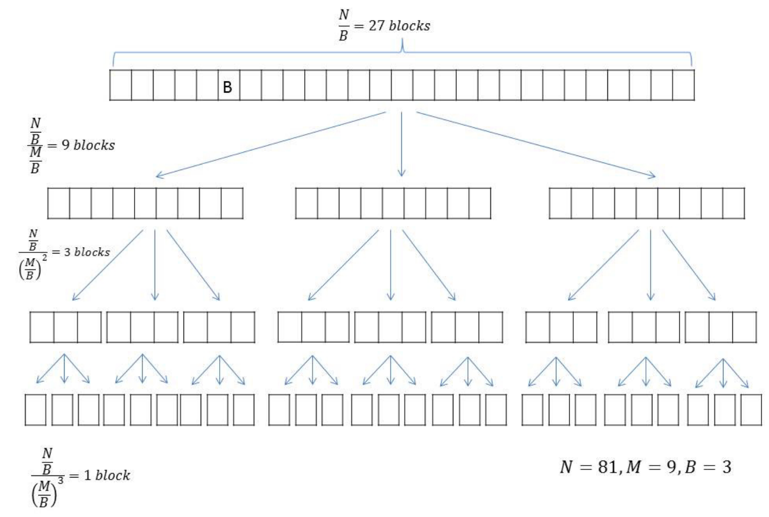

$\frac{N}{B} = 27$ blocks
B
$\frac{N/B}{M/B} = 9$ blocks
$\frac{N/B}{(M/B)^2} = 3$ blocks
$\frac{N/B}{(M/B)^3} = 1$ block
$N = 81, M = 9, B = 3$

Figure 1: Example of dividing and merging the N numbers by Yu-Yao Lin

(2) $T(n) = kT(n/k) + f(n)$

$T(n) = O(1)$ when $n = 1$

n is the number of small chunks: $n = N/B$

k is the number of chunks to merge every time: $k = M/B$

f(n) is the number of memory transfers in every iteration: $f(n) = O(N/B)$

Problem 7

From the problem description, we know that there is only one number in A appearing twice in B.

(1) in $O(n \log n)$ time

We can use quick sort to sort B, which is in $O(n \log n)$ time. Then scan the array to find which number appears twice, which is in $O(n)$ time.

So in total, this algorithm is in $O(n \log n)$ time, and in $O(1)$ space.

Pseudo code:

```
void function findDuplicate
sort(B);
/*assume index starts from 1*/
for (i=1; i<n; i++) {
    if $i_{th}$ value of B == $(i+1)_{th}$ value of B
        print "$B[i]$ appears twice";
        break;
}
```

(2) in $O(n)$ time

We can use hashmap to count the frequencies of each number in B by scanning B once. During the traverse, if we find any number's frequency is 2, we find it.

This algorithm is in $O(n)$ time, and in $O(n)$ space.

Pseudo code:

```
void function findDuplicate
initial a hashmap;
/*assume index starts from 1*/
for (i=1; i<=n; i++) {
    if $i_{th}$ value of B already exists in the hashmap
        print "$B[i]$ appears twice";
        break;
    else
        add $i_{th}$ value of B into hashmap;
}
```

Problem 8

(1) We can simply solve it by using brute force, similar to bubble sort. For every nut, scan all the bolts to find a matching pair.

Pseudo code:

```
void function matchBolt
/*assume index starts from 1*/
for (i=1; i<=n; i++) {
    for (j=1; j<=n; j++) {
        if the size of $i_{th}$ nut == the size of $j_{th}$ bolt
            print “nut i and bolt j are a matching pair”;
            break;
    }
}
```

(2) We randomly pick a nut and a bolt to test,

if size of nut > size of bolt, we remove nut from collection, as it will not be the smallest. At the same time, if previously we marked any bolt as an candidate, we remove it, too. Then pick another nut to test.

if size of nut < size of bolt, we remove bolt from collection, and pick another bolt to test.

if size of nut == size of bolt, we keep the bolt as an candidate, and pick another bolt to test.

Not stop until only one bolt or nut exists.

Thus, we can see on average, every comparison will result in removing one nut or bolt from the collection. So the worst case is to remove $2n-2$ bolts and nuts from the collection, then only one nut and one bolt exist, which will cost $2n-2$ comparisons.

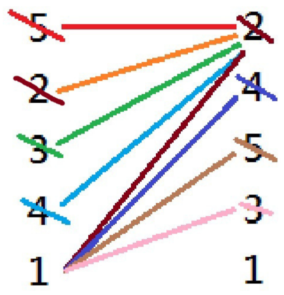


Figure 2: Find the smallest nut by Yu-Yao Lin

Problem 9

We need $O(n \log n)$ memory transfers to perform the matrix transpose.

The main idea is divide and conquer. Every time we divide this problem into 4 smaller sub matrix, each of size $n/4$, until the matrix becomes something like the below:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \mathit{or} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Then we conquer each sub problem by swaping blocks on the antidiagonal and leave blocks on the main diagonal unchanged. We do it like merging in merge sort until the whole matrix has been transposed. In each iteration, when we swap 2 numbers, we need to load the related two pages into memory, thus it is in O(n) transfers.

The recurrence relation is $T(n) = 4T(n/4) + O(n)$.

So in total, we need $O(n \log n)$ memory transfers.

Below is an example of matrix transpose from Yu-Yao Lin.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \Rightarrow \begin{bmatrix} a_{11} & a_{21} & a_{13} & a_{23} \\ a_{12} & a_{22} & a_{14} & a_{24} \\ a_{31} & a_{41} & a_{33} & a_{43} \\ a_{32} & a_{42} & a_{34} & a_{44} \end{bmatrix} \Rightarrow \begin{bmatrix} a_{11} & a_{21} & a_{31} & a_{41} \\ a_{12} & a_{22} & a_{14} & a_{24} \\ a_{13} & a_{23} & a_{33} & a_{43} \\ a_{32} & a_{42} & a_{34} & a_{44} \end{bmatrix} \Rightarrow \begin{bmatrix} a_{11} & a_{21} & a_{31} & a_{41} \\ a_{12} & a_{22} & a_{32} & a_{42} \\ a_{13} & a_{23} & a_{33} & a_{43} \\ a_{14} & a_{24} & a_{34} & a_{44} \end{bmatrix}$$

Problem 10

(1) We can set two pointers. At beginning, they both points to the root of the linked list. Then one runs faster than the other one (e.g., one goes forward 2 nodes every time while the other goes forward only 1 node). Both of two nodes stop when they get to the end of the linked list.

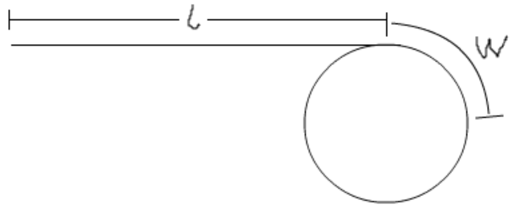
So if they meet before they both reach the end (except the root node), it means there exists a loop in the linked list. Otherwise, it is a proper list.

This algorithm uses $O(n)$ time and $O(1)$ space.

(2) We can set two pointers. One's speed is 2 nodes every time while the other is 1 node every time. Because there exists a loop, the two pointers will meet at some point inside the loop. Then we reset one pointer to the head and the two pointers goes at the same speed (e.g., 1 node every time). Finally they will meet at one node. That is the place where the loop starts.

This algorithm is in $O(n)$ time and $O(1)$ space.

Take the following picture to make a brief proof.



l is the length from the head node to the entrance node of the loop.

w is the length from the entrance node to the node where two pointers first meet.

Assume the length of the loop is c, the slow pointer's speed is v, and the fast pointer's speed is $2v$.

Then we get equations:

$2v = l + kc + w$ where k means the fast pointer has run k circles before it meets the slow pointer.

$v = l + w$

Thus, we get $2l + 2w = l + kc + w$, which is $l + w = kc$.

So after the two pointers meet at the first time, we set one to the beginning of the linked list, and let them go with the same speed. When the pointer which starts at the head of the linked list reaches the entrance node of the loop, it runs length l.

The second pointer which starts at the node where they first met also runs length l. We see that $l + w = kc$, which means the second pointer has just run k circles and now it stops at the entrance node of the loop.

The two pointers meet again and they are all at the entrance node of the loop!

Problem 11

I think it is just right. I spend a week on assignment 1 and 2.

Previously I did not use latex, so it took me some time to learn it. Now I love it!