Assignment 2 – Written

CSC 225

Ryan Woodward

V00857268

1. Case I: $f(N) = N+c$

2. : It is assumed the initial size of the array is 0.

- Therefore the first operation will be a special push operation.

$$\rightarrow f(N) + N + 1 = N + c + N + 1 = 2N + c + 1$$

- Another special operation will not occur until $c-1$ push operations have been completed.

$$\sum_{i=0}^{c-1} (1) = c-1$$

- Therefore a special operation occurs every c^{th} operation.

$$\rightarrow \frac{2N+c+1}{c} = \text{special cost}$$

- And a push operation occurs every $c-1$ operations the total cost is

$$\sum_{i=0}^{n} \frac{2N+c+1}{c} + (c-1) = \sum_{i=0}^{n} \frac{2N+c+1+c^2-c}{c}$$

$$= \sum_{i=0}^{n} \frac{2N+c^2+1}{c}$$

2. Case 2: $f(N) = 2N$

- A special op will occur every $2^{n\text{th}}$ operation.

$$f(N) = \frac{2N}{2^n}$$

· Whereas a push operation would occur (N-1) operations.

· When comparing case (1) and (2) we can see that a special op in (1) will occur much more frequently at $\frac{2N+C+1}{C}$ than in case (2) at $\frac{2N}{2^n}$ since the growth rate of $2^n > c$ is this case.

∴ Therefore for large quantities it is more efficient to use case (2).

2. array A =, n keys, k inversions.

- k occurs when a pair of entries are out of order in the array.

- If there are n keys, and k inversions then insertion sort is

$$k \cdot \sum_{i=1}^{n} (n-1) = k \cdot \frac{(n-1+1)(n-1)}{2} = \frac{kn^2}{2} - \frac{kn}{2}$$

The running time of this is $O(n^2)$. It is assumed that for the worst case scenario there will be k inversions and each key will have to be shifted that many times,

3. Algorithm: inversion Count
input: an array of integers
output: the number of inversions in the array.

① tempArray ← array //copy original array so no changes are made.

② merge (tempArray, int[] left, int[] right) {
int i, j, count ← 0
while (i < left Length or j < right length) {
if (i == left length) {
tempArray [i+j] ← right[j]
j ← j+1
}
else if (j == right length) {
tempArray [i+j] ← left[i]
i ← i+1
}
else if (left[i] ≤ right[j]) {
tempArray [i+j] = left[i]
i ← i+1
}
else {
tempArray [i+j] ← right [j]
count ← count + left Length - i ;
j ← j+1
}
}
return count
}

. This is essentially merge sort without modifying the original array and with a counter.

4. $T(n) = 1$ if $n=1$
$\quad = 4T\left(\frac{n}{2}\right) + n\log n$ if $n \geq 2$.

$$T(n) = \left[4 \cdot \left[4\, T\left(\frac{n}{2^2}\right)\right]\right] + n\log n + n\log n$$

$$= 4^2\, T\left(\frac{n}{2^2}\right) + 2 \cdot n\log n$$

$$T(n) = 4\left[4^2\, T\left(\frac{n}{2^3}\right)\right] + 2n\log n + n\log n$$

$$= 4^3\, T\left(\frac{n}{2^3}\right) + 3n\log n.$$

$$T(n) = 4^i\, T\left(\frac{n}{2^i}\right) + i\, n\log n$$

Let $2^i = n$
$i \log 2 = \log n$
$i = \log n$

$$T(n) = 4^{\log n}(1) + \log n \cdot n \cdot \log n$$

$$\boxed{T(n) = 4^{\log n} + n(\log n)^2}$$

5. sequence S of n elements
- integers in range $[0; n^2-1]$

- Create a separate integer array of n^2-1 elements in length, each element set to zero.

- Go through the original array of n elements,

```
int i = 0
while ( i < n ) {
    newArray[oldArray[i]]++
}
```

- Therefore when each index is switched "on" in the newArray we can simply then use another while loop to reset the old array.

```
int i=0, int j=0
while ( i < n^2-1 ) {
    if (newArray[i] != 0 )
        oldArray[j] = i
        j++
    }
}
```