CmS 311
Ben Holmes

btholmes-hw2.pdf

Problem 1

a) $$r = \sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=1}^{j-i} 1 = \sum_{i=1}^{n} \sum_{j=i}^{n} \left[(j-i) \right]$$

$$= \sum_{i=1}^{n} \left[\sum_{j=i}^{n} j - \sum_{j=i}^{n} i \right] = \sum_{i=1}^{n} \left[\sum_{j=i}^{n} j - i(n-i+1) \right]$$

$$= \sum_{i=1}^{n} \left[\frac{(i-n-1)(i+n)}{2} - in + i^2 - i \right]$$

$$= \sum_{i=1}^{n} \left[\frac{i^2 - n^2 - i - n}{2} - in + i^2 - i \right]$$

$$= \sum_{i=1}^{n} \left[\frac{n^2 + n + (1+i)}{2} - in - i + i^2 \right]$$

$$= \sum_{i=1}^{n} \left[\frac{n^2 + n - 2in - i - 1}{2} \right] = \sum_{i=1}^{n} \frac{(i-n-1)(i-n)}{2}$$

$$= \frac{1}{6} n (n^2 - 1) = \frac{1}{6} n^3 - \frac{1}{6} n = \underline{\Theta(n^3)}$$

(b) Consider the inner loop,
it is a simple for loop, from 1 to i, where
i is originally set to n.
On the next iteration of the outer loop, the inner loop runs $\frac{n}{2}$ times, and so on.
So the sum of inner loops executions can be found through the following geometric sequence.

$$n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} \ldots \log(n-1) + \log(n)$$

Using the formula for the geometric series:

$$S_n = \frac{a(1-r^n)}{1-r} \quad \text{where } r = \frac{1}{2}, \; a = n, \; n = \log(n)$$

Thus.

$$S_{\log(n)} = \frac{n\left(1-\left(\frac{1}{2}\right)^{\log(n)}\right)}{1-\left(\frac{1}{2}\right)} = 2n\left(1-\left(\frac{1}{2}\right)^{\log(n)}\right)$$

$$= 2n\left(1-2^{-\log(n)}\right) = \underline{O(n)}$$

COMS 311 HW 2
Ben Holmes

Problem 2:

2) Overview:
The algorithm starts at 1, and sets a pointer there called j. It then loops through an array of size n, from 1 to n-1. At each iteration, it compares index j with index (j-1).
If a[j-1] < a[j], it increments j by 1, but if a[j-1] > a[j], it swaps(a[j], a[j-1]), decrements j by 1, and then continues comparing each a[j] with a[j-1] until j == 0, or a[j-1] < a[j].

a) Worst Case : Reverse Sorted runs in O(n^2)
In this case, at the start of each iteration i from 1 to n-1, there will be i elements to the left, which are greater than the element at a[i]. So, there will be i additional executions
of the while loop while each element slowly makes its way back to the front of the array. Since each execution in the while loop only takes constant time, we have :
c*1 + c*2 + c*3 .... c(n-1) This sum is an arithmetic series that goes up to (n-1).
Using the formula for the arithmetic series, we get:
c*(n-1+1)((n-1)/2) = cn^2/2 - cn/2

$$\therefore \frac{c(n-1)n}{2} = \frac{cn^2}{2} - \frac{cn}{2}$$

Best Case : Sorted runs in O(n)
In this scenario, the inner while loop will never be executed, because there will never be a time when a[j-1] < a[j]. Therefore the total time is just the constant time it takes to check the
statment, a[j-1] > a[j], which is the summation from 1 to (n-1) of c, which is c(n-1):
c(n-1) which is O(n).

b) Total time for SORTED selection of 3000 is : 20
Total time for REVERSED selection of 3000 is : 20
Total time for RANDOM selection of 3000 is : 7

Total time for SORTED bubble of 3000 is : 19
Total time for REVERSED bubble of 3000 is : 14
Total time for RANDOM bubble of 3000 is : 2

Total time for SORTED insertion of 3000 is : 16
Total time for REVERSED insertion of 3000 is : 27
Total time for RANDOM insertion of 3000 is : 5

Total time for SORTED selection of 30000 is : 1268
Total time for REVERSED selection of 30000 is : 1246
Total time for RANDOM selection of 30000 is : 1271

Total time for SORTED bubble of 30000 is : 214

Total time for REVERSED bubble of 30000 is : 197
Total time for RANDOM bubble of 30000 is : 198

Total time for SORTED insertion of 30000 is : 168
Total time for REVERSED insertion of 30000 is : 176
Total time for RANDOM insertion of 30000 is : 161

Total time for SORTED selection of 300000 is : 128856
Total time for REVERSED selection of 300000 is : 129915
Total time for RANDOM selection of 300000 is : 129130

Total time for SORTED bubble of 300000 is : 20875
Total time for REVERSED bubble of 300000 is : 20906
Total time for RANDOM bubble of 300000 is : 20792

Total time for SORTED insertion of 300000 is : 19859
Total time for REVERSED insertion of 300000 is : 18763
Total time for RANDOM insertion of 300000 is : 18794

Com S 311
Hw 2

Problem 3

a) Is $48n^4 - 46n^2 + 25n + 31 \in O(n^4)$?

$$48n^4 - 46n^2 + 25n + 31 \leq 48n^4 + 46n^4 + 25n^4 + 31n^4$$
$$= 150n^4$$

Thus take $c = 150$
$\therefore 48n^4 - 46n^2 + 25n + 31 \leq 150n^4$

b) Is $n^{\log(n)} \in O(2^{\sqrt{n}})$
There is a constant c such that

$$n^{\log(n)} \leq c(2^{\sqrt{n}})$$

$\Rightarrow \log^2(n) \leq \log(c) + \sqrt{n}$ [Take log of both sides]

$\Rightarrow \log^2(n) - \sqrt{n} \leq \log(c)$

No: $\log^2(n) - \sqrt{n}$ is a monotonically increasing function, and can't be bound by a constant.

c) Is $2^{2^{n+1}} \in O(2^{2^n})$
Suppose there is a constant c such that

$$2^{2^{n+1}} \leq c \cdot 2^{2^n}$$
$$2^{n+1} \leq 2^n + \log(c) \quad \text{[Take lg of sides]}$$
$\Rightarrow 2^{n+1} - 2^n \leq \log(c)$
$\Rightarrow 2^n \leq \log(c)$

No, 2^n is a monotonically incr. function, and can't be bound by a constant.

d) Is $n^3(5+\sqrt{n}) \in O(n^3)$?

Suppose there is a constant c such that

$$n^3(5+\sqrt{n}) \leq cn^3$$

$$5+\sqrt{n} \leq c \quad [\text{divide by } n^3]$$

No, $5+\sqrt{n}$ is monotonically increasing, and can't be bound by a constant.

4a) Algorithm for k^m, runs in $\theta(\log n)$ time.

```
function power(k, m) {
    if m == 0
        return 1;
    else if (m % 2 == 0) {
        result = power(k, m/2)
        return result * result;
    }
    else {
        result = power(k, m/2)
        return result * result * k;
    }
}
```

```
function decimalNum(k, array-int n) {
    result = 0
    for(i = 0; i < n.length; i++) {
        result += n[n.length - i] * power(k, i)
    }
    return result;
}
```

Run Time: For loop runs n.length times, or just n times for clarity. Then each execution of the for loop runs the power function, which finishes in $\theta(\log(n))$ time.

Thus, Total runtime = $\underline{\theta(n * \log(n))}$

4b) function convertFromDecimal(K, m) {

// If we convert this ~~case~~ K base # to an array of bits, we can use concepts of modular arithmetic to find its value.
Lets say we need n bits to represent this K-base #.
We create an array of n bits for the num.

```
array[n];
int i = n;
while(m ≥ 0) {
    remainder = m % K;
    array[i] = remainder; // put remainder on end
    i--;
    m = m/K;
}
return array;
}
```

Time Complexity:

A decimal m can be represented in K-base format using $\log_K m$ bits.

Complexity = $\underline{\theta(\log_K m)}$