CS170 Discussion Section 1: *1/18*

1. Asymptotic notation

(a)
- $f(n) \in O(g(n))$
- $f(n) \in \Theta(g(n))$
- $f(n) \in O(g(n))$
- $f(n) \in \Omega(g(n))$
- $f(n) \in O(g(n))$

(b)
- $f(n) \in \Theta(1)$
- $f(n) \in \Theta(n^2)$
- $f(n) = \frac{(n+1)n}{2} \in \Theta(n^2)$
- $f(n) \in \Theta(1.01^n)$
- $f(n) \in \Theta(n^{1.1})$

2. Analyze the running time

For each pseudo-code snippet below, give the asymptotic running time in Θ notation. Assume that basic arithmetic operations (+,−,×, and /) are constant time.

(a)
```
for $i := 1$ to $n$ do
    $j := 0$;
    while $j \le i$ do
        $j := j + 2$
```

The inner loop takes time $i/2$, so the running time is

$$\sum_{i=1}^{n} i/2 = \Theta(n^2).$$

(b)
```
$s := 0$;
$i := n$;
while $i \ge 1$ do
    $i := i$ div 2;
    for $j := 1$ to $i$ do
        $s := s + 1$
```

For simplicity, round n up to the nearest power of 2. Remember that $\log n$ is the number of times it is possible to divide a number by 2 before reaching 1, so the outer loop runs $\log n$ times. The inner loop takes i time, and in the kth iteration of the outer loop, $i = \frac{n}{2^k}$. So, the running time is

$$\sum_{k=1}^{\log n} n \cdot 2^{-k} = \Theta(n).$$

Note: Using geometric series, we know

$$\sum_{k=1}^{\log n} 2^{-k} \le \sum_{k=1}^{\infty} 2^{-k} = 1$$

(c)
```
$i := 2$;
while $i \leq n$ do
    $i := i^2$
```

In each loop iteration, i is squared. If we express i in the form 2^m, then squaring it gives $2^{m \cdot 2}$. So, i is of the form $2^{2 \cdot 2 \cdot \dots \cdot 2}$. In fact, after k loop iterations, $i = 2^{2^k}$. The loop stops when $2^{2^k} > n$; that is, when $k > \log\log n$, so the number of loop iterations is $\lceil \log\log n \rceil$. The running time is thus

$$\Theta(\log\log n).$$

(d)
```
for $i := 1$ to $n$ do
    $j := i^2$;
    while $j \leq n$ do
        $j := j + 1$
```

When $i \leq \sqrt{n}$, the inner loop runs for $n - i^2$ steps; otherwise, it stops immediately and takes time $\Theta(1)$. So the running time is

$$\Theta\left((n - \sqrt{n}) + \sum_{i=1}^{\sqrt{n}} (n - i^2) \right) = \Theta(n\sqrt{n}).$$

Note:

$$\sum_{i=1}^{\sqrt{n}} (n - i^2) = \sum_{i=1}^{\sqrt{n}} n - \sum_{i=1}^{\sqrt{n}} i^2 \approx n\sqrt{n} - \frac{1}{3} n\sqrt{n} \in \Theta(n\sqrt{n})$$

This uses the fact that $\sum_{i=1}^{n} i^2 \approx \frac{n^3}{3}$.

3. Four-part Algorithm Practice

Given a sorted array A of n integers, you want to find the index at which a given integer k occurs, i.e. index i for which $A[i] = k$. Design an efficient algorithm to find this i.

Main idea: We find i using binary search, i.e. we compare k with the middle entry to decide which half of the array to recursively search.

Psuedocode:

```
function BinSearch($A[1..n], k$)
    if $length(A) < 1$
        return NOT FOUND
    else if $A[\lceil n/2 \rceil] == k$
        return $\lceil n/2 \rceil$
    else if $k < A[\lceil n/2 \rceil]$
        return BinSearch($A[1..\lceil n/2 - 1 \rceil, k$)
    else
        return $\lceil n/2 \rceil$+BinSearch($A[\lceil n/2 + 1 \rceil..n, k$)
```

Proof of correctness: We will prove by induction that if an array of size n contains k, BinSearch will find the index of k.

Base case: If $n = 1$ (assuming k is present means $A[1] = k$, we will hit the second case, so we will find the correct index.

Inductive hypothesis: BinSearch works for arrays of size $\leq m$ for some m, where "works" means that the correct index is returned if k is present.

Inductive step: Assume the inductive hypothesis. Now consider running BinSearch with an array of size $m+1$. If $\lceil m/2 \rceil$ happens to fall on the desired k, then we output the correct answer immediately. Otherwise, one of the last two cases is hit, so we recurse on one half A. Because A is sorted, our comparison ensures that we recurse on the half of A that contains k. The recursive call will be correct by the inductive hypothesis since one half of the array has size $\leq m$. By induction, this means we will find the correct index for any size array, if k is present.

It remains to show that if k is not present, the algorithm will not return a valid index. This is easy to see, as we only actually return an index if we find k, otherwise "NOT FOUND" is output.

Running time analysis:

$\Theta(\log n)$

In the worst case, we keep hitting one of the last two cases until we have an array of size 0. Since each recursive call throws out at least half of the remaining elements in A, it will take order $\log n$ steps to terminate. Note that the comparisons within each call can be done in constant time. Thus the overall running time is $\Theta(\log n)$.

(Note: This is assuming k and each integer in A are limited to a constant number of bits.)

4. Sorted Array

Given a sorted array A of n (possibly negative) distinct integers, you want to find out whether there is an index i for which $A[i] = i$. Give a divide-and-conquer algorithm that runs in time $O(\log n)$.

Along the same lines as binary search, start by examining $A[\frac{n}{2}]$. Clearly, if $A[\frac{n}{2}]$ is $\frac{n}{2}$ then we have a satisfactory index; if $A[\frac{n}{2}] > \frac{n}{2}$ then no element in the second half of the array can possibly satisfy the condition because each integer is at least one greater than the previous integer, and hence the difference of $A[\frac{n}{2}] - \frac{n}{2}$ can not decrease by continuing through the array; and if $A[\frac{n}{2}] < \frac{n}{2}$ then by the same logic no element in the first half of the array can satisfy the condition. While the algorithm has not terminated or left an empty array, we discard the half of the array that cannot hold an answer and repeat the same check. At each step we do a single comparison and discard at least half of the remaining array (or terminate), so this algorithm takes $O(\log n)$ time.

5. Computing Factorials

Consider the problem of computing $N! = 1 \times 2 \times \cdots \times N$.

(a) If N is an n-bit number, how many bits long is $N!$, approximately (in $\Theta(\cdot)$form)?

When we multiply an m bit number by an n bit number, we get an $(m + n)$ bit number. When computing factorials, we multiply N numbers that are at most n bits long, so the final number has at most Nn bits.

But if you consider the numbers from $\frac{N}{2}$ to N, we multiply at least $\frac{N}{2}$ numbers that are at least $n - 1$ bits long, so the resulting number has at least $\frac{N(n-1)}{2}$ bits.

Thus, the number of bits in $N!$ is in $\Theta(nN)$.

(b) Give an algorithm to compute $N!$ and analyze its running time.

We can compute $N!$ naively as follows:

```
factorial ($N$)
    $f = 1$
    for $i = 2$ to $N$
        $f = f \cdot i$
```

Running time : we have N iterations, each one multiplying an $N \cdot n$-bit number (at most) by an n-bit number. Using the naive multiplication algorithm, each multiplication takes time $O(N \cdot n^2)$. Hence, the running time is $O(N^2 n^2)$.