CS 170 HW 1

Due on 2018-09-02, at 11:59 pm

1 (★) Study Group

List the names and SIDs of the members in your study group.

2 (★★★) Analyze the running time

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

(a)
```
for i := 1 to n do
    j := 0;
    while j ≤ i do
        j := j + 2
    end
end
```

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

(c)
```
i := 2;
while i ≤ n do
    i := i^2
end
```

(d)
```
for i := 1 to n do
    j := i^2;
    while j ≤ n do
        j := j + 1
    end
end
```

Solution:

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

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

(b) 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

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

(c) 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 $2^{2 \cdot 2 \cdot \cdots \cdot 2}$. 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) 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 (★★★) Asymptotic Complexity Comparisons

(a) Order the following functions so that $f_i = O(f_j) \iff i \leq j$. Do not justify your answers.

(i) $f_1(n) = 3^n$
(ii) $f_2(n) = n^{\frac{1}{3}}$
(iii) $f_3(n) = 12$
(iv) $f_4(n) = 2^{\log_2 n}$
(v) $f_5(n) = \sqrt{n}$
(vi) $f_6(n) = 2^n$
(vii) $f_7(n) = \log_2 n$
(viii) $f_8(n) = 2^{\sqrt{n}}$
(ix) $f_9(n) = n^3$

Solution: $f_3, f_7, f_2, f_5, f_4, f_9, f_8, f_6, f_1$

(b) In each of the following, indicate whether $f = O(g)$, $f = \Omega(g)$, or both (in which case $f = \Theta(g)$). ***Briefly*** justify each of your answers. Recall that in terms of asymptotic growth rate, logarithmic < linear < polynomial < exponential.

	$f(n)$	$g(n)$
(i)	$\log_3 n$	$\log_4 n$
(ii)	$n \log(n^4)$	$n^2 \log(n^3)$
(iii)	$\sqrt{n}$	$(\log n)^3$
(iv)	2^n	2^{n+1}
(v)	n	$(\log n)^{\log \log n}$
(vi)	$n + \log n$	$n + (\log n)^2$
(vii)	$\log(n!)$	$n \log n$

Solution:

(i) $f = \Theta(g)$; using the log change of base formula, $\frac{\log n}{\log 3}$ and $\frac{\log n}{\log 4}$ differ only by a constant factor.

(ii) $f = O(g)$; $f(n) = 4n \log(n)$ and $g(n) = 3n^2 \log(n)$, and the polynomial in g has the higher degree.

(iii) $f = \Omega((\log n)^3)$; any polynomial dominates a product of logs.

(iv) $f = \Theta(g)$; $g(n) = 2f(n)$ so they are constant factors of each other.

(v) $f = \Omega(g)$; $n = 2^{\log n}$ and $(\log n)^{\log \log n} = 2^{(\log \log n)^2}$, so f grows faster than g since $\log n$ grows faster than $(\log \log n)^2$.

(vi) $f = \Theta(g)$; Both f and g grow as $\Theta(n)$ because the linear term dominates the other.

(vii) $f = \Theta(g)$;
Observe that

$$n! = 1 * 2 * 3 \cdots * n \leq n * n * n \cdots * n \leq n^n$$

and assuming n is even (without loss of generality)

$$n! = 1 * 2 * 3 \cdots * n \geq n * (n-1) * (n-2) \cdots * (n - n/2) \geq \left(\frac{n}{2}\right)^{\frac{n}{2}+1}.$$

Hence $\left(\frac{n}{2}\right)^{\frac{n}{2}} \leq n! \leq n^n$. Then,

$$\frac{n}{2} \log\left(\frac{n}{2}\right) \leq \log(n!) \leq n \log n.$$

and we conclude that the functions grow at the same asymptotic rate.

4 (★★) Bit Counter

Consider an n-bit counter that counts from 0 to 2^n.

When $n = 5$, the counter has the following values:

Step	Value	# Bit-Flips
0	00000	–
1	00001	1
2	00010	2
3	00011	1
4	00100	3
⋮	⋮	
31	11111	1
31	00000	5

For example, the last two bits flip when the counter goes from 1 to 2. Using $\Theta(\cdot)$ notation, find the growth of the *total* number of bit flips (the sum of all the numbers in the "# Bit-Flips" column) as a function of n.

Solution:

The number of times the (i)-th bit from the left is flipped is (2^i). For example, the first bit is flipped once, and the last is flipped every time, 2^n times.

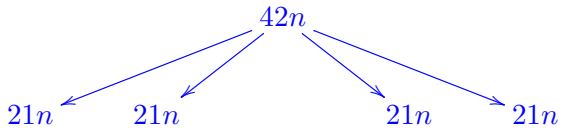
There are n bits, so the total number of bit flips is

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

5 (★★) Recurrence Relations

(a) $T(n) = 4T(n/2) + 42n$

Solution: Use the master theorem. Or:



⋮

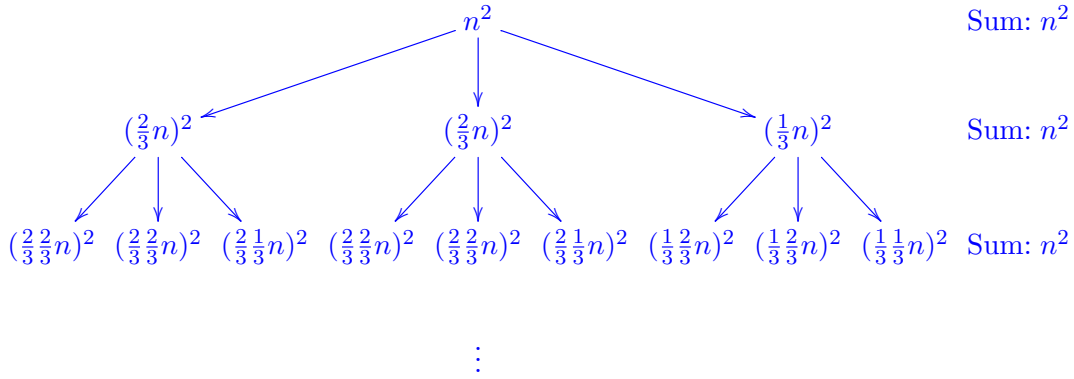
The first level sums to $42n$, the second sums to $84n$, etc. The last row dominates, and we have $\log n$ rows, so we have $42 \cdot 2^{\log n} \cdot n = \Theta(n^2)$.

(b) $T(n) = 4T(n/3) + n^2$

Solution: Use the master theorem (the case $d > \log_b a = \log_3 4$), or expand like the previous question. The answer is $\Theta(n^2)$.

(c) $T(n) = 2T(2n/3) + T(n/3) + n^2$

Solution: This one is a bit tricky:



So the answer is: $\Theta(n^2 \log n)$.

(d) $T(n) = 3T(n/4) + n \log n$

Solution: We end up with $\sum_{i=0}^{\log_4 n} (3/4)^i n \log(n/4^i)$. We can lower-bound this by $n \log n$ by taking the first term, and upper-bound it by $n \log n$ be replacing $\log(n/4^i)$ by $\log n$, so this is $\Theta(n \log n)$.

6 (★★) 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)?

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

Solution:

(a) 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) We can compute $N!$ naively as follows:

```
factorial (N)
    f = 1
    for i = 2 to N
        f = f · 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 $O(N^2 n^2)$.

7 (★★★) Four-subpart 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:

Psuedocode:

Proof of correctness:

Running time analysis:

Solution:

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:

```
procedure BinSearch(A[1..n], k)
    if length(A) < 1
        return NOT FOUND
    else if A[⌈n/2⌉] == k
        return ⌈n/2⌉
    else if k < A[⌈n/2⌉]
        return BinSearch(A[1..⌈n/2 − 1⌉, k)
    else
        return ⌈n/2⌉+BinSearch(A[⌈n/2 + 1⌉..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.)

8 (★★★) Hadamard matrices

The Hadamard matrices $H_0, H_1, H_2, \ldots$ are defined as follows:

- H_0 is the 1×1 matrix $[1]$
- For $k > 0$, H_k is the $2^k \times 2^k$ matrix

$$H_k = \left[\begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array}\right]$$

(a) Write down the Hadamard matrices H_0, H_1, and H_2.

Solution: $H_0 = 1$

$$H_1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$H_2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

(b) Compute the matrix-vector product $H_2 v$, where H_2 is the Hadamard matrix you found above, and $v = \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix}$ is a column vector. Note that since H_2 is a 4×4 matrix, and v is a vector of length 4, the result will be a vector of length 4.

Solution:

$$H_2 v = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 4 \end{bmatrix}$$

(c) Now, we will compute another quantity. Take v_1 and v_2 to be the top and bottom halves of v respectively. Therefore, we have that $v_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$, $v_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$, and $v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$. Compute $u_1 = H_1(v_1 + v_2)$ and $u_2 = H_1(v_1 - v_2)$ to get two vectors of length 2. Stack u_1 above u_2 to get a vector u of length 4. What do you notice about u?

Solution:

$$H_1(v_1 + v_2) = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$H_1(v_1 - v_2) = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ -2 \end{bmatrix} = \begin{bmatrix} 0 \\ 4 \end{bmatrix}$$

We notice that $u = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 4 \end{bmatrix} = H_2 v$

(d) Suppose that

$$v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

is a column vector of length $n = 2^k$. v_1 and v_2 are the top and bottom half of the vector, respectively. Therefore, they are each vectors of length $\frac{n}{2} = 2^{k-1}$. Write the matrix-vector product $H_k v$ in terms of H_{k-1}, v_1, and v_2 (note that H_{k-1} is a matrix of dimension $\frac{n}{2} \times \frac{n}{2}$, or $2^{k-1} \times 2^{k-1}$). Since H_k is a $n \times n$ matrix, and v is a vector of length n, the result will be a vector of length n.

Solution: $H_k v = \begin{bmatrix} H_{k-1} v_1 + H_{k-1} v_2 \\ H_{k-1} v_1 - H_{k-1} v_2 \end{bmatrix} = \begin{bmatrix} H_{k-1}(v_1 + v_2) \\ H_{k-1}(v_1 - v_2) \end{bmatrix}$

(e) Use your results from (c) to come up with a divide-and-conquer algorithm to calculate the matrix-vector product $H_k v$, and show that it can be calculated using $O(n \log n)$ operations. Assume that all the numbers involved are small enough that basic arithmetic operations like addition and multiplication take unit time.

Solution: Compute the 2 subproblems described in part (d), and combine them as described in part (c). Let $T(n)$ represent the time taken to find $H_k v$. We need to find the vectors $v_1 + v_2$ and $v_1 - v_2$, which takes $O(n)$ time. And we need to find the matrix-vector products $H_{k-1}(v_1 + v_2)$ and $H_{k-1}(v_1 - v_2)$, which take $T(\frac{n}{2})$ time. So, the recurrence relation for the runtime is:

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

Using the master theorem, this give us $T(n) = O(n \log n)$.