

*This homework covers big-*Oh* and analysis of algorithms. It is due in class Wednesday, February 4.*

*Write your solutions carefully — your work should be neat, readable, and organized. Be sure to show your work / provide support for your answers — don't simply state an answer without any indication of where it came from.*

*See the Policies page on the course website for information about revise-and-resubmit, late work, and academic integrity as it applies to homework.*

---

1. Do the homework #1 drill problems on Canvas. (Look for `hw1 drill` in the Quizzes section.) Keep track of your scratch work since #2–4 ask you to write up your solutions for some of the drill problems.
2. For each of the following functions, find a simple function  $g$  such that  $f(n) = \Theta(g(n))$  (or  $T(n) = \Theta(g(n))$ ). Provide support for your answers — show your work, state what rule you are applying or step you are doing, etc. Don't just write down  $g$  by itself!

Keep in mind the various strategies — algebraic simplification (including simplifying logs and exponents), using the sums and recurrence relations tables, and dropping constant multipliers and lower-order terms.

(i) $10n^3 + 2n^5$	(ii) $2^{n-1}$
(iii) $n \log_{10} n^2$	(iv) $n^{1/3} + \log n$
(v) $\sum_{i=1}^n \frac{3n}{i}$	(vi) $\sum_{i=1}^{\frac{n}{2}} (i + \log n)$
(vii) $\log_2 \left( \sum_{i=0}^n 2^i \right)$	(viii) $T(n) = 2T(n/2) + n \log^2 n$
(ix) $T(n) = 2T(n/3) + 1$	(x) $T(n) = 9T(n/4) + n^2$
(xi) $T(n) = T(n - 2) + 1$	

3. For each of the following pairs of functions, indicate which is the faster growing. For functions not in the known ordering of common functions from class, provide support for your answers — indicate how you decided which function is faster growing.

Keep in mind strategies for comparing growth rates to determine  $O$ ,  $\Omega$ ,  $\Theta$  — eliminating common factors, plotting the functions, and identifying  $c$  and  $n_0$  according to the definitions.

(a) $n \log n, n^{1.5}$	(b) $n \log^2 n, n^2 \log n$	(c) $n^{\frac{1}{3}}, \log n$
-------------------------	------------------------------	-------------------------------

4. Give the worst-case running time for the following set of loops. Explicitly utilize that the total time for a loop is the sum of the time taken by each iteration — write the sum(s) and use the sums table to get the big-Oh for each. Assume that the [loop body] steps take  $\Theta(1)$  time.

Hint: if a sum doesn't follow the pattern needed for the sums table (sum range starts at 1,  $f(n)$  has only multiplicative terms), try rewriting the sum — write out the terms of the sum, simplify them as appropriate, and look for a new pattern that lets you write an equivalent but simpler sum.

```
for i = 1 to n do
  for j = 1 to i do
    for k = j to i+j do
      for m = 1 to i+j-k do
        [loop body]
```

5. Bubble sort is a simple sorting algorithm which works by repeatedly comparing adjacent elements and swapping them if they are out of order.

```
algorithm bubbleSort ( A, n ) :
  input: array A storing n items
  output: items in A are sorted in increasing order

  repeat
    swapped ← false
    for ( j ← 0 ; j < n-1 ; j++ ) do
      if A[j] > A[j+1] then // if elements are reversed...
        temp ← A[j] // ...swap them
        A[j] ← A[j+1]
        A[j+1] ← temp
        swapped ← true
    until swapped is false
```

Give the  $\Theta$  running time for bubble sort. Identify the best- and worst-case times separately if there is a difference. Show your work / provide support for your answer(s) — don't just give a  $\Theta$  without explanation.

6. We might try to improve bubble sort by allowing elements that are far out of place to move more than one spot at a time — 'gap' defines this amount, and shrinks as elements (presumably) get closer to their final spots as the algorithm progresses. The idea is to repeatedly "comb" through the array with increasingly finer-toothed combs, swapping elements hit by the comb's teeth if they are out of order.

```

algorithm combSort ( A, n ) :
    input: array A storing n items
    output: items in A are sorted in increasing order

    gap ← n
    repeat
        gap ← gap/s
        swapped ← false
        for ( j ← 0 ; j < n-gap ; j++ ) do
            if A[j] > A[j+gap] then // if elements are reversed...
                temp ← A[j]    // ...swap them
                A[j] ← A[j+gap]
                A[j+gap] ← temp
                swapped ← true
        until gap == 1 and swapped is false

```

Give the  $\Theta$  running time for comb sort. Identify the best- and worst-case times separately if there is a difference. Show your work / provide support for your answer(s) — don't just give a  $\Theta$  without explanation.

7. A *matrix* is a 2D array of numbers. A key operation involving matrices is matrix multiplication.

This problem involves two algorithms for matrix multiplication. Both algorithms involving *partitioning*, in which an  $n \times n$  matrix is split into four  $n/2 \times n/2$  matrices as shown.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

The operations `extractBlock`, `assembleBlocks`, `add`, and `subtract` referenced in the pseudocode below all take  $\Theta(n^2)$  time when applied to  $n \times n$  matrices.

(a) The basic algorithm for `multiply(A,B)` computes  $C$  as follows:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

This can be written in pseudocode as shown:

```

algorithm basicMultiply ( A, B ) :
    input: A, B are nxn arrays
    output: C = AxB

```

```
A11 = extractBlock(A,0,0)
```

```

A12 = extractBlock(A,0,n/2)
A21 = extractBlock(A,n/2,0)
A22 = extractBlock(A,n/2,n/2)

B11 = extractBlock(B,0,0)
B12 = extractBlock(B,0,n/2)
B21 = extractBlock(B,n/2,0)
B22 = extractBlock(B,n/2,n/2)

C11 = add(basicMultiply(A11,B11),basicMultiply(A12,B21))
C12 = add(basicMultiply(A11,B12),basicMultiply(A12,B22))
C21 = add(basicMultiply(A21,B11),basicMultiply(A22,B21))
C22 = add(basicMultiply(A21,B12),basicMultiply(A22,B22))

C = assembleBlocks(C11,C12,C21,C22)

```

Write the recurrence relation for and give the running time of `basicMultiply`.

(b) A more clever approach for `multiply(A,B)` instead computes  $C$  as follows:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix}$$

where the  $M_i$  are  $n/2 \times n/2$  matrices computed as follows:

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22})B_{11} \\ M_3 &= A_{11}(B_{12} - B_{22}) \\ M_4 &= A_{22}(B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12})B_{22} \\ M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

This can be written in pseudocode as shown:

```

algorithm cleverMultiply ( A, B ) :
    input: A, B are nxn arrays
    output: C = AxB

    A11 = extractBlock(A,0,0)
    A12 = extractBlock(A,0,n/2)
    A21 = extractBlock(A,n/2,0)
    A22 = extractBlock(A,n/2,n/2)

    B11 = extractBlock(B,0,0)
    B12 = extractBlock(B,0,n/2)
    B21 = extractBlock(B,n/2,0)

```

```
B22 = extractBlock(B,n/2,n/2)

M1 = cleverMultiply(add(A11,A22),add(B11,B22))
M2 = cleverMultiply(add(A21,A22),B11)
M3 = cleverMultiply(A11,subtract(B12,B22))
M4 = cleverMultiply(A22,subtract(B21,B11))
M5 = cleverMultiply(add(A11,A12),B22)
M6 = cleverMultiply(subtract(A21,A11),add(B11,B12))
M7 = cleverMultiply(subtract(A12,A22),add(B21,B22))

C11 = subtract(add(M1,M4),add(M5,M7))
C12 = add(M3,M5)
C21 = add(M2,M4)
C22 = add(subtract(M1,M2),add(M3,M6))

C = assembleBlocks(C11,C12,C21,C22)
```

Write the recurrence relation for and give the running time of `cleverMultiply`.