The exam will be in class (one hour) and closed book. You may not use any additional resources other than the sums and recurrence relations tables and logarithm/exponent rules from class. (These will be provided on the exam.)

It is strongly recommended that you treat this as a practice exam and work on it in a similar environment.

---

1. Showing your work / providing some evidence of where your answers come from can help you earn partial credit if you make a mistake.

   (a) For each of the functions below, find a simple function g such that $f(n) = \Theta(g(n)$ (or $T(n) = \Theta(g(n))$).

   (i) $10 \log^2 n = \Theta(\log^2 n)$
   Drop multiplicative constants.

   (ii) $5n \log n + 2n^3 = \Theta(n^3)$
   Drop lower-order terms ($n \log n$ grows slower than n^3 — divide out the common factor n from both sides, then the comparison is between the well-known functions $\log n$ and n^2) and multiplicative constants.

   (iii) $10000 = \Theta(1)$

   (iv) $\sum_{i=1}^{n} \left(\frac{2}{3}\right)^i i^2 = \Theta(1)$
   $b^a = \frac{2}{3} < 1$ so the pattern is $\Theta(1)$.

   (v) $\sum_{i=1}^{n^2} n \log i = \Theta(n^3 \log n)$

   First factor out the constant terms: $n \sum_{i=1}^{n^2} \log i$. For the sum, $b^a = 1$, $d = 0$, $e = 1$ so the pattern is $\Theta(nf(n))$ where n is the upper limit of the sum — thus $\Theta(n^2 f(n^2))$ where $n^2 f(n^2) = n^2 \log n^2 = 2n^2 \log n = \Theta(n^2 \log n)$. The final result is n times the sum, or $n\Theta(n^2 \log n) = \Theta(n^3 \log n)$.

   (vi) $\sum_{i=1}^{\log n} i = \Theta(\log^2 n)$
   $b^a = 1$, $d = 1$, $e = 0$ so the pattern is $\Theta(nf(n))$ where n is the upper limit of the sum — thus $\Theta((\log n) f(\log n))$ where $(\log n) f(\log n) = (\log n)(\log n) = \log^2 n$.

   (vii) $T(n) = 3T(n/3) + 3n = \Theta(nlogn)$
   This is the $aT(n/b)+\Theta(n^c \log^d n)$ pattern — $a = 3$, $b = 3$ so $(\log a)/(\log b) = 1$. $c = 1$ so $(\log a)/(\log b) = c$; $d = 0$ so the pattern is $T(n) = \Theta(f(n) \log n) = \Theta(n \log n)$.

(viii) $T(n) = 2T(n-2) + 4 = \Theta(2^{n/2})$
This is the $aT(n-b) + \Theta(n^c \log^d n)$ pattern — $a = 2$ so the pattern is $T(n) = \Theta(a^{n/b}) = \Theta(2^{n/2})$.

(ix) $T(n) = T(n/2) + n^2 = \Theta(n^2)$
This is the $aT(n/b)+\Theta(n^c \log^d n)$ pattern — $a = 1, b = 2$ so $(\log a)/(\log b) = (\log 1)/(\log 2)$. $c = 2$ so $(\log a)/(\log b) < c$ and the pattern is $T(n) = \Theta(f(n)) = \Theta(n^2)$.

(b) List the functions in order according to growth rate, from slowest growing to fastest growing. If several functions have the same growth rate, indicate that and list them together.

The big-Ohs in order: 1, $log^2 n$, $n \log n$, n^2, n^3, $n^3 \log n$, $2^{n/2}$. (Note that the question is actually asking for a list of the functions from (a), not just the big-Ohs.)

2. For each of the following, if there is a difference between best-case and worst-case behavior, give running times for both. Show your work / provide some evidence of how you arrived at your answers, such as by writing the applicable sums or recurrence relations.

   (a) Give the running time of `alg` for arrays A and B of length n.

```
algorithm alg ( A, B ) :
  input: arrays A, B of length n

 for ( i ← 0 ; i < n ; i += 2 ) do
   B[i] ← A[i]
   B[i+1] ← i

 for ( j ← n-1 ; j ≥ 0 ; j-- ) do
   for ( k ← 1 ; k < j ; k *= 2 ) do
     B[j] ← B[j] + A[k]
```

   `alg` consists of one loop after another, so add the work of the `i` and `j` loops. The `i` loop repeats $n/2$ times with $O(1)$ work each repetition, so it is $O(n)$. For the nested loops, write the sums. For the `k` loop, the value of k doubles each time (1, 2, 4, 8, 16, 32, ...). We need an incrementing-by-1 value for a sum variable, so observe that this pattern is powers of two i.e. at iteration $k' \geq 0$, $k = 2^{k'}$. The loop ends when $k = 2^{k'} \geq j$ i.e. $k' \geq \log j$. Thus the running time for the `k` loop can be written as $\sum_{k'=0}^{\log j} \Theta(1)$ and the overall running time for the nested loops is $\sum_{j=0}^{n-1} \sum_{k'=0}^{\log j} \Theta(1)$.

   Solve this from the inside out. $\Theta(1)$ is a constant, so $\sum_{k'=0}^{\log j} \Theta(1)$ is just the number of repetitions times the work per repetition — $\Theta(\log j)$.

   The total running time for nested loops is thus $\sum_{j=0}^{n-1} \Theta(\log j)$. $b^a = 1$, $d = 0$, $e = 1$ so the pattern for this sum is $\Theta(nf(n)) = \Theta(n \log n)$.

   The total time is the sum of the time for the `i` and `j` loops. Since $n = O(n \log n)$, $\Theta(n) + \Theta(n \log n)$ simplifies to $\Theta(n \log n)$.

(b) Give the running time of `alg` for arrays A and B with lengths m and n, respectively.

```
algorithm alg ( A, m, B, n ) :
  input: array A of length m, array B of length n; m ≤ n

  for ( b ← 0 ; b ≤ n-m ; b++ ) do
    if helper(A,B,b) then
      return b
  return -1


algorithm helper ( A, B, i ) :
  input: array A of length m, array B of length n,
         0 ≤ i ≤ n − m

  for ( a ← 0 ; a < m ; a++ ) do
    if A[a] != B[i+a] then
      return false
  return true
```

Consider `alg`: the best case for the loop is if `helper` returns true on the first call, while the worst case for the loop is if `helper` always returns false.

In order for `helper` to return `true`, its loop repeats m times — $\Theta(1)$ work per repetition means $\Theta(m)$ work in this case. So the best case for `alg`'s loop means $\Theta(m)$ work total.

If `helper` returns `false`, the best case is if $A[0] \neq B[i]$ and it returns on the first loop iteration — $\Theta(1)$ work. The worst case is if the loop repeats until the next-to-last iteration — $A[a] = B[i + a]$ until $a = m - 1$. This is $\Theta(m)$ work.

So, the worst case for `alg` is for its loop to repeat all $n - m + 1$ times with `helper` taking $\Theta(m)$ each time for a total of $\Theta(m(n-m))$ work. This could be rounded up to $O(nm)$ but since the time is really proportional to the difference in length between A and B, it is more meaningful to leave it as is.

(c) Give the running time of `alg` for arrays A, B of length n. You can assume n is a power of 2.

```
algorithm alg ( A, B, n ) :
  input: arrays A, B of length n
  output: an array of length 2n

  if n = 1 then
    return { A[0]*B[0] }
  if n = 2 then
    return { 0, A[1]*B[1], A[1]*B[0]+A[0]*B[1], A[1]*B[1] }

  for ( i ← 0 ; i < n/2 ; i++ ) do
    A'[i] ← A[i]+A[n/2+i]
    B'[i] ← B[i]+B[n/2+i]

  R1 ← alg(helper(A,0,n/2),helper(B,0,n/2),n/2)
  R2 ← alg(helper(A,n/2,n),helper(B,n/2,n),n/2)
  R3 ← alg(A',B',n/2)

  for ( i ← 0 ; i < n ; i++ ) do
    R4 ← R3[i] - R1[i] - R2[i]

  for ( i ← 0 ; i < 2n ; i++ ) do
    R[i] ← 0
    if 0 ≤ i < n then
      R[i] ← R[i] + R1[i]
    if 0 ≤ i-n/2 < n then
      R[i] ← R[i] + R4[i-n/2]
    if 0 ≤ i-n < n then
      R[i] ← R[i] + R2[i-n]
  return R


algorithm helper ( A, i, j ) :
  input: array A, values i < j
  output: an array of length j − i containing A[i..j − 1]

  for ( k ← i ; k < j ; k++ ) do
    B[k-i] ← A[k]
  return B
```

`alg` is recursive, so write a recurrence relation to get the running time. For `alg(n)`, there are three calls to `alg(n/2)` plus a bunch of additonal work: $T(n) = 3T(n/2)+f(n)$. The additional work involves three loops (with $n/2$, n, and $2n$ iterations respectively and $\Theta(1)$ work per iteration), four calls to `helper` where $j - i = n/2$, and $\Theta(1)$ additional steps. Consider each piece:

- The loops each have $\Theta(n)$ iterations with $\Theta(1)$ work per iteration, for a total of $\Theta(n)$ work.
- `helper` has a loop with $j - i$ iterations and $\Theta(1)$ work per iteration. Four calls where $j - i = n/2$ means a total of $\Theta(n)$ work.
- A fixed number of $\Theta(1)$ additional steps is a total of $\Theta(1)$ additional work.

Thus there is a total of $\Theta(n)$ additional work and the recurrence relation is $T(n) = 3T(n/2) + \Theta(n)$. This is the $aT(n/b) + \Theta(n^c \log^d n)$ pattern — $a = 3$, $b = 2$ so $(\log a)/(\log b) = (\log 3)/(\log 2)$. $c = 1$ so $(\log a)/(\log b) > c$ and the pattern is $T(n) = \Theta(n^{(\log a)/(\log b)}) = \Theta(n^{(\log 3)/(\log 2)}) = \Theta(n^{1.58})$.

3. In addition to answering the questions (yes or no), briefly explain your answers — why or why not?

   (a) If $f(n) = O(n^2)$, can it also be true that $f(n) = \Theta(n \log n)$?

   Yes — O defines an upper bound, and $f(n)$ can grow slower.

   (b) If $f(n) = \Omega(n^2)$, can it also be true that $f(n) = O(n \log n)$?

   No, Ω defines a lower bound so $f(n)$ grows at least that fast. $n^2 = \Omega(n \log n)$ so $f(n)$ can't grow at least as fast as n^2 but also no faster than the slower-growing $n \log n$.

   (c) If an algorithm is described as taking $\Theta(n^2)$ time, is it possible for the best case to be $O(n)$?

   No, because describing the algorithm as $\Theta(n^2)$ means that the running time of both best and worst cases grows like n^2, not n.