INSTRUCTIONS:

1. Please clearly write your name and your PSU Access User ID (i.e., xyz1234) in the box on top of **every page**; otherwise that page will be taken as "scratch page" and everything you write on that page will be ignored.
2. For each problem please try to finish your answer within the page and the page next. If not enough please use the last 4 pages (pages 13–16) and clearly mark which problem your answer is for.
3. Unless it is explicitly specified, by default you do not need to prove the correctness nor prove the running time of your algorithm.

Problem 1 (24 points). For each pairs of functions below, indicate one of the three: $f = O(g)$, $f = \Omega(g)$, or $f = \Theta(g)$. You do not need to illustrate the middle steps leading to your answer.

1. $f(n) = 100 \cdot \log(n^{1.01})$, $g(n) = \log(100 \cdot n)$
   Solution: $f = \Theta(g)$
2. $f(n) = n^3 \cdot 2^n$, $g(n) = n^2 \cdot 3^n$
   Solution: $f = O(g)$
3. $f(n) = 2^{n \cdot \log n}$, $g(n) = 3^n$
   Solution: $f = \Omega(g)$
4. $f(n) = (\log n)^{\log \log n}$, $g(n) = (\log \log n)^{\log n}$
   Solution: $f = O(g)$
5. $f(n) = n!$, $g(n) = n^{\log n}$
   Solution: $f = \Omega(g)$
6. $f(n) = (\log n)^2$, $g(n) = \sum_{k=1}^{n}(1/k)$
   Solution: $f = \Omega(g)$

Problem 1 (24 points). For each pairs of functions below, indicate one of the three: $f = O(g)$, $f = \Omega(g)$, or $f = \Theta(g)$. You do not need to illustrate the middle steps leading to your answer.

I. $f(n) = n^2 \cdot 3^n$, $g(n) = n^3 \cdot 2^n$
   Solution: $f = \Omega(g)$

II. $f(n) = 10 \cdot \log(n^{1.01})$, $g(n) = \log(10 \cdot n)$
   Solution: $f = \Theta(g)$

III. $f(n) = n^{\log n}$, $g(n) = n!$
   Solution: $f = O(g)$

IV. $f(n) = (\log n)^2$, $g(n) = \sum_{k=1}^{n}(1/k)$
   Solution: $f = \Omega(g)$

V. $f(n) = 3^n$, $g(n) = 2^{n \cdot \log n}$
   Solution: $f = O(g)$

VI. $f(n) = (\log\log n)^{\log n}$, $g(n) = (\log n)^{\log\log n}$

Solution: $f = \Omega(g)$

Problem 2 (16 points). Solve each of the following recursions. You do not need to illustrate the middle steps leading to your answer.

1. $T(n) = 3 \cdot T(n/\sqrt{3}) + n^2$.
   Solution: $T(n) = \Theta(n^2 \cdot \log n)$
2. $T(n) = 2 \cdot T(n/4) + n^{0.5}$.
   Solution: $T(n) = \Theta(n^{0.5} \cdot \log n)$
3. $T(n) = 4 \cdot T(n/5) + n$.
   Solution: $T(n) = \Theta(n)$
4. $T(n) = 3 \cdot T(n/2) + n \cdot \log n$.
   Solution: $T(n) = \Theta(n^{\log_2 3})$

Problem 2 (16 points). Solve each of the following recursions. You do not need to illustrate the middle steps leading to your answer.

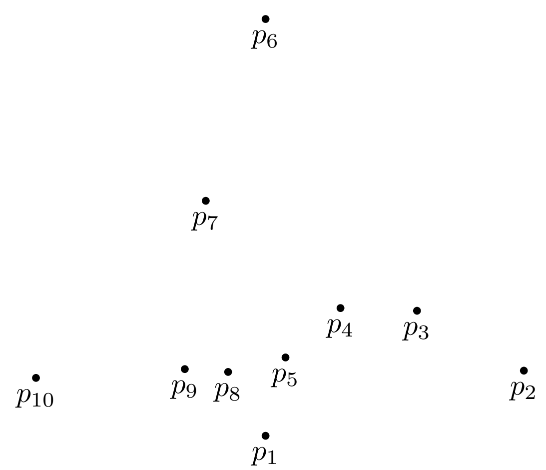
I. $T(n) = 3 \cdot T(n/4) + n$.
Solution: $T(n) = O(n)$

II. $T(n) = 2 \cdot T(n/4) + n^{0.5}$.
Solution: $T(n) = O(n \cdot \log n)$

III. $T(n) = 4 \cdot T(n/\sqrt{2}) + n^3$.
Solution: $T(n) = O(n^4)$

IV. $T(n) = 4 \cdot T(n/3) + n \cdot \log n$.
Solution: $T(n) = O(n^{\log_3 4})$

Problem 3 (10 points). Run the Graham-Scan algorithm on the following instance: draw the status of the stack as you process each point.



Solution: Through running the Graham-Scan algorithm, the status of the stack will be (the left side shows the bottom of the stack, and the right side shows the top of the stack):
processing p_1: $[p_1$
processing p_2: $[p_1, p_2$
processing p_3: $[p_1, p_2, p_3$
processing p_4: $[p_1, p_2, p_3, p_4$
processing p_5: $[p_1, p_2, p_3, p_4, p_5$
processing p_6: $[p_1, p_2, p_3, p_4$
processing p_6: $[p_1, p_2, p_3$
processing p_6: $[p_1, p_2$
processing p_6: $[p_1, p_2, p_6$
processing p_7: $[p_1, p_2, p_6, p_7$
processing p_8: $[p_1, p_2, p_6, p_7, p_8$
processing p_9: $[p_1, p_2, p_6, p_7$
processing p_9: $[p_1, p_2, p_6, p_7, p_9$
processing p_{10}: $[p_1, p_2, p_6, p_7$
processing p_{10}: $[p_1, p_2, p_6$
processing p_{10}: $[p_1, p_2, p_6, p_{10}$

Problem 4 (15 points). Let $S[1 \cdots n]$ be an array with *n distinct* positive integers. We say two indices (i, j) form a *big-inversion* of S if we have $i < j$ and $S[i] \geq 2 \cdot S[j]$. Design a divide-and-conquer algorithm to count the number of big-inversions in S. Your algorithm should run in $O(n \cdot \log n)$ time.

Solution:

Define recursive function count-big-inversions (S) returns (S', N), where S' is the sorted list of S, and N is the number of big-inversions in array S. The pseudo-code for this function is below.

---

function count-big-inversions (S)
    if ($|S| = 1$): return $(S, 0)$;
    let $n = |S|$;
    $(S_1, N_1) =$ count-big-inversion ($S[1 \cdots n/2]$);
    $(S_2, N_2) =$ count-big-inversion ($S[n/2+1 \cdots n]$);
    $N_3 =$ count-big-inversions-between-two-sorted-arrays (S_1, S_2);
    $S_3 =$ merge-two-sorted-arrays (S_1, S_2); /* used in merge-sort */
    return $(S_3, N_1 + N_2 + N_3)$;
end function

The function count-big-inversions-between-two-sorted-arrays (S_1, S_2) used above takes two sorted list S_1 and S_2 as input, and returns N_3, where N_3 is the number of big-inversions across S_1 and S_2, i.e., number of pair (i, j) such that $S_1[i] > 2 \cdot S_2[j]$. The pseudo-code for this function is below.

---

function count-big-inversions-between-two-sorted-lists (S_1, S_2)
    let $k_1 = 1, k_2 = 1, N = 0$;
    while ($k_1 \leq |S_1|$ and $k_2 \leq |S_2|$)
        if ($S_1[k_1] < 2 \cdot S_2[k_2]$)
            $k_1 = k_1 + 1$;
        else
            $N = N + (|S_1| - k_1) + 1$;
            $k_2 = k_2 + 1$;
        end if
    end while
    return N;
end function

The function of count-big-inversions-between-two-sorted-arrays takes linear time. The function of merge-two-sorted-arrays also takes linear time. Hence, the recursion for the entire algorithm is $T(n) = 2 \cdot T(n/2) + O(n)$. Therefore the running time is $O(n \cdot \log n)$.

Problem 5 (15 points). You are given an array $S[1 \cdots n]$ with n distinct positive integers. Array S satisfies a property: there exists an unknown *summit-index* k such that $S[i] < S[i+1]$ for any $1 \leq i < k$ and that $S[i] > S[i+1]$ for any $k \leq i < n$. Design an algorithm to find the summit-index of such an array S. Your algorithm should run in $O(\log n)$ time.

Solution:

Define recursive function find-summit-index (S, a, b) returns the summit-index of $S[a \cdots b]$, i.e., returns the index $k \in [a, b]$ such that $S[k] > S[i]$ for any $a \leq i \leq b$ and $i \neq k$.

---

function find-summit-index (S, a, b)
    if $a = b$: return a;
    let $m = (a+b)/2$;
    if $S[m] < S[m+1]$: return find-summit-index (S, $m+1$, b);
    else if $S[m] < S[m-1]$: return find-summit-index (S, a, $m-1$);
    else: return m
end function

Call find-summit-index (S, 1, n) will give the summit-index of S. The recursion for running time is $T(n) = O(n) + T(n/2)$. Therefore, the running time of this algorithm is $O(\log n)$.

Problem 6 (20 points). Let $S[1 \cdots n]$ be an array with n *distinct* positive integers. Let $N = \sum_{k=1}^{n} S[k]$ be the sum of all numbers in S. For each $1 \leq k \leq n$, define $X_k = \sum_{i:S[i]<S[k]} S[i]$, i.e., X_k is the sum of all numbers in S that are less than $S[k]$. We say $S[k]$ is the *sum-median* of S if we have $X_k < N/2$ and $X_k + S[k] \geq N/2$.

1. **(5 points).** Design an algorithm to compute the sum-median of S. Your algorithm should run in $O(n \cdot \log n)$ time.

   Solution: The following procedure will return the sum-median of S.

```
function find-sum-median (S)
    S' = merge-sort (S); /* S' is the sorted list of S; any O(n · log n) sorting algorithm works */
    let N = 0;
    for k from 1 to n;
        N = N + S[k];
    end for;
    let s = 0;
    for k from 1 to n;
        if s + S[k] ≥ N/2: return k;
        s = s + S[k];
    end for;
end function
```

The sorting step dominates the running time. Therefore, this algorithm runs in $O(n \log n)$ time.

2. **(15 points).** Design a randomized algorithm to compute the sum-median of S. The expected running time of your algorithm should be $O(n)$ and you need to prove this.
Hint: use the randomized selection algorithm in your algorithm.

Solution 1:

Define recursive function randomized-find-bound (S, M) returns $S[k]$ such that $X_k < M$ and $X_k + S[k] \geq M$.

```
function randomized-find-sum-median (S, M)
    let m = |S|/2
    let x = selection (S, m); /* use the randomized selection algorithm to get median of S */
    build S_L and S_R, where S_L = {S[i] | S[i] < x}, S_R = {S[i] | S[i] > x};
    compute N_L = ∑_{s∈S_L} s;
    if N_L ≥ M: return randomized-find-sum-median (S_L, M);
    else if N_L + x ≥ M: return x;
    else: return randomized-find-sum-median (S_R, M − N_L − x);
end function
```

Call randomized-find-sum-median (S, $N/2$) will give the sum-median of S.

Let $T(n)$ be the running time of randomized-find-sum-median when $|S| = n$. then we have $T(n) = S(n) + O(n) + T(n/2)$, where we use $S(n)$ to denote the running time of the randomized selection algorithm, $O(n)$ represents the running time of building S_L and S_R, and $T(n/2)$ represents the running time of the recursive call of randomized-find-sum-median as the size of S_L or S_R is $n/2$ (because x is the median of S). Take expectation on both side, we have $T(n) = O(n) + O(n) + T(n/2) = O(n) + T(n/2)$. Therefore, the expected running time of the above algorithm is $O(n)$.

Solution 2:

Define recursive function randomized-find-bound (S, M) returns $S[k]$ such that $X_k < M$ and $X_k + S[k] \geq M$.

---

function randomized-find-sum-median (S, M)

  randomly select k from 1 to n

  let $x = S[k]$;

  partition S into x, S_L and S_R, where $S_L = \{S[i] \mid S[i] < x\}$, $S_R = \{S[i] \mid S[i] > x\}$;

  compute $N_L = \sum_{s \in S_L} s$;

  if $N_L \geq M$: return randomized-find-sum-median (S_L, M);

  else if $N_L + x \geq M$: return x;

  else: return randomized-find-sum-median (S_R, $M - N_L - x$);

end function

Call randomized-find-sum-median (S, $N/2$) will give the sum-median of S.

Let $T(n)$ be the running time of randomized-find-sum-median when $|S| = n$. Then we have $T(n) = O(n) + T(X)$, where $O(n)$ represents the running time of partition, X is the input size of next recursion, and $T(X)$ represents the running time of the next recursive call of randomized-find-sum-median. Take expectation on both side, we have $T(n) = O(n) + \sum_k \Pr(X = k) \cdot T(k) = O(n) + \sum_{k \leq 3n/4} \Pr(X = k) \cdot T(k) + \sum_{k > 3n/4} \Pr(X = k) \cdot T(k)$. Then $T(n) \leq O(n) + 1/2 \cdot T(3n/4) + 1/2 \cdot T(n)$ (see lecture notes of Feb 05 for more details). Thus $T(n) \leq O(n) + T(3n/4)$ and $T(n) = O(n)$. Therefore, the expected running time of the above algorithm is $O(n)$.