Problem 1 (10 points). Solve each of the following recursions.

1. $T(n) = 2 \cdot T(n/2) + n \cdot \log n$
2. $T(n) = 4 \cdot T(n/2) + n \cdot (\log n)^2$
3. $T(m,n) = 4 \cdot T(m,n/2) + m \cdot n^2$

Solution.

1. We can use the *recursion tree* approach to solve it:
$T(n) = \sum_{k=0}^{\log n} 2^k \cdot (n/2^k) \cdot \log(n/2^k) = \sum_{k=0}^{\log n} n \cdot (\log n - k) = n \cdot \log n \cdot (\log n + 1) - n \cdot \sum_{k=0}^{\log n} k = n \cdot \log n \cdot (\log n + 1) - n \cdot (\log n + 1) \cdot \log n/2 = \Theta(n \cdot (\log n)^2)$
2. We can use *lower and upper bounds* to solve it. Let $f(n) = n \cdot (\log n)^2$, $g_1(n) = n$, and $g_2(n) = n^{1.01}$. We have $f = O(g_2)$ and $f = \Omega(g_1)$. Let $T_1(n) = 4 \cdot T(n/2) + g_1(n)$ and $T_2(n) = 4 \cdot T(n/2) + g_2(n)$. We have $T_1(n) \leq T(n) \leq T_2(n)$. Solving $T_1(n)$ and $T_2(n)$ with master's theorem gives $T_1(n) = \Theta(n^2)$ and $T_2(n) = \Theta(n^2)$. Therefore, we have $T(n) = \Theta(n^2)$.
3. Notice that in subproblems $T(m,n/2)$ the first parameter m is the same with that in $T(m,n)$. Hence we can regard m as a "constant" in solving this recursion. Let $T'(n) = T(m,n)$. We have $T'(n) = 4 \cdot T(n/2) + m \cdot n^2$. By applying master's theorem, we have $T'(n) = \Theta(m \cdot n^2 \cdot \log n) = T(m,n)$.

NOTE: Following is the more general form of the master's theorem, which can be used to directly solve the above first two recursions. Let $T(n) = a \cdot T(n/b) + f(n)$ be the recursion we want to solve. Let constant $c = \log_b a$. Let $\varepsilon > 0$ be any positive constant. We have

$$T(n) = \begin{cases} \Theta(n^c) & \text{if} \quad f(n) = O(n^{c-\varepsilon}) \\ \Theta(f(n) \cdot \log n) & \text{if} \quad f(n) = \Theta(n^c \cdot (\log n)^k) \\ \Theta(f(n)) & \text{if} \quad f(n) = \Omega(n^{c+\varepsilon}) \end{cases}$$

Problem 2 (10 points). You are given a polygon with n vertices, represented as the coordinates of its n vertices $((x_1,y_1),(x_2,y_2),\cdots,(x_n,y_n))$ along the polygon in counter-clockwise order. Design a linear-time algorithm to decide whether this polygon is convex.

Solution. The algorithm is to check, for any continuous 3 vertices, (p_k, p_{k+1}, p_{k+2}), along the polygon, their orientation must be either "turn left", or colinear (then p_{k+2} must be on the extension of $p_k p_{k+1}$).

```
function decide-convex-polygon (P[1···n])
    append P[1] and P[2] to P;
    for k = 1 to n
        let vector a = P[k+1] − P[k] := (a1, a2);
        let vector b = P[k+2] − P[k] := (b1, b2);
        let x = a1 · b2 − a2 · b1;
        if x > 0: continue;
        if x < 0: return false;
        if ||a||2 ≤ ||b||2: continue; /* when a and b are colinear, compute and compare their norms */
        else return false;
    end for;
    return true;
end function
```

Problem 3 (20 points). You are given a sorted array $S[1 \cdots n]$ with n distinct integers, i.e., $S[i] < S[i+1]$, for all $1 \le i < n$. Design a divide-and-conquer algorithm to decide whether there exists an index k such that $S[k] = k$. Your algorithm should run in $O(\log n)$ time.

Solution. The algorithm is to do a *binary search*. We compare $S[n/2]$ with $n/2$. If $S[n/2] = n/2$ then we find such index. If $S[n/2] > n/2$, then we can claim that for any $k > n/2$ we must have $S[k] > k$. This is because all integers in S are distinct and sorted in ascending order. Therefore for any $k > n/2$ we have $S[k] \ge S[n/2] + (k - n/2) > n/2 + k - n/2 = k$. In other words, when we have $S[n/2] > n/2$ when we can discard the second half of the array, and only search such index in the first half. With the same reasoning, when we have $S[n/2] < n/2$ when we can discard the first half of the array, and only search such index in the second half.

We define the recursive function decide-index (S, a, b) with parameters array $S[1 \cdots n]$, two indices a and b satisfying $1 \le a \le b \le n$, returns an index k if $S[k] = k$ and $a \le k \le b$, and returns false if no such k can be found.

---

function decide-index ($S[1 \cdots n]$, a, b)
   if $a = b$ and $S[a] = a$: return a;
   if $a = b$ and $S[a] \neq a$: return false;
   let $m = (a+b)/2$;
   if $S[m] = m$: return m;
   if $S[m] > m$: return decide-index (S, a, $m-1$);
   if $S[m] < m$: return decide-index (S, $m+1$, b);
end function

With the above recursive function, the call of decide-index (S, 1, n) will give us such k if $S[k] = k$ in the entire array, and return false if no such index can be found in the entire array.

Notice how to analyze the running time of this algorithm. First, the array S never changes during the recursive call. So we can always pass a *pointer* of S, which takes constant space, to every call of decide-index. This means that n is *not* the input size of the recursive function. In fact, the input size of decide-index should be defined as $b - a$, as it will be reduced by half in every iteration.

Let $T(k)$ be the running time of decide-index (S, a, b) when $k = b - a$ (regardless the size of S). We then have $T(k) = T(k/2) + O(1)$. Therefore, we have $T(k) = O(\log k)$. We eventually need to run decide-index (S, 1, n), which therefore takes $O(\log n)$ time.

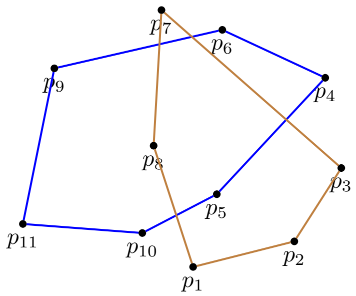
Problem 4 (20 points). Given the following two convex polygons $C_1 = (p_1, p_2, p_3, p_7, p_8)$ and $C_2 = (p_5, p_4, p_6, p_9, p_{11}, p_{10})$, compute the convex hull of $C_1 \cup C_2$ using the linear time algorithm described within the divide-and-conquer algorithm for convex hull:

1. Partition C_2 into two sorted list, C_2^{UP} and C_2^{LOW}, such that the points in each list are sorted w.r.t. the anchor point p_1 in counter-clockwise order.

2. Give the merged list of C_2^{UP} and C_2^{LOW}, denoted as C_2', such that all points in C_2' are sorted w.r.t. the anchor point p_1 in counter-clockwise order.

3. Give the merged list C_2' and C_1, denoted as C, such that all points in C are sorted w.r.t. the anchor point p_1 in counter-clockwise order.

4. Run the Graham-Scan-Core algorithm with C as input: give the status of the stack as each point in C gets processed.



Solution.

1. $C_2^{UP} = (p_4, p_6, p_9)$ and $C_2^{LOW} = (p_{11}, p_{10}, p_5)$.
2. $C_2' = (p_4, p_5, p_6, p_9, p_{10}, p_{11})$.
3. $C = (p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11})$.
4. The status of stack is as follows:
   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: $[p_1, p_2, p_3, p_4, p_6$
   processing p_7: $[p_1, p_2, p_3, p_4, p_6, p_7$
   processing p_8: $[p_1, p_2, p_3, p_4, p_6, p_7, p_8$
   processing p_9: $[p_1, p_2, p_3, p_4, p_6, p_7$
   processing p_9: $[p_1, p_2, p_3, p_4, p_6, p_7, p_9$
   processing p_{10}: $[p_1, p_2, p_3, p_4, p_6, p_7, p_9, p_{10}$
   processing p_{11}: $[p_1, p_2, p_3, p_4, p_6, p_7, p_9$
   processing p_{11}: $[p_1, p_2, p_3, p_4, p_6, p_7, p_9, p_{11}$

Problem 5 (10 points). Analysis the expected running time of the following randomized algorithm for sorting. You may assume that all elelments in A are distinct.

---

function combined-sort (array $A[1 \cdots n]$)
 if $n = 1$, then return A;
 select k from $\{1, 2, \cdots, n\}$ uniformly at random;
 compute A_L as the list of elements in A that are smaller than $A[k]$;
 compute A_R as the list of elements in A that are larger than $A[k]$;
 $X_L =$ merge-sort (A_L);
 $X_R =$ merge-sort (A_R);
 return $(X_L, A[k], X_R)$.
end function

Solution. Let $T(n)$ be the running time of combined-sort (A) when array A contains n elements. Notice that $T(n)$ is a random variable as there is randomization within the algorithm. Notice also that this algorithm is not a recurive algorithm. Let $M(m)$ be the running time of merge-sort when its input size is m. We know that $M(m) = O(m \cdot \log m)$. We have $T(n) = O(n) + M(|A_L|) + M(|A_R|)$. Notice that $|A_L|$ is also a random variable with distribution of $\Pr(|A_L| = i) = 1/n$, for any $0 \le i \le n-1$. Similarly,

$\Pr(|A_R| = i) = 1/n$, for any $0 \le i \le n-1$. Take expection w.r.t. the randomness of $|A_L|$ and $|A_R|$ on both sides. We have $T(n) = O(n) + 2 \cdot \sum_{i=0}^{n-1} 1/n \cdot M(i) = O(n) + 2/n \cdot \sum_{i=0}^{n-1} O(i \cdot \log i) = O(n \cdot \log n)$.

Problem 6 (30 points). The square of a matrix A is its product with itself, i.e., AA.

1. Show that 5 multiplications are sufficient to compute the square of a 2×2 matrix.
2. What is wrong with the following algorithm for computing the square of an $n \times n$ matrix?
   "Use a divide-and-conquer algorithm as in Strassen's algorithm, except that instead of getting 7 subproblems of size $n/2$, we now get 5 subproblems of size $n/2$ thanks to part (1). Using the same analysis as in Strassen's algorithm, we can conclude that the algorithm runs in time $O(n^{\log_2 5})$"
3. Show that if $n \times n$ matrices can be squared in $O(n^c)$ time for certain constant c, then any two $n \times n$ matrices can be multipled in $O(n^c)$.

Solution.

1. We have
   $$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} a^2 + bc & ab + bd \\ ac + cd & bc + d^2 \end{bmatrix}$$
   It suffice to compute a^2, bc, $b(a+d)$, $c(a+d)$, and d^2.
2. The problem with above reasoning is that, as the recursion goes, the subproblems are not "square of a matrix" any more.
3. Given any two $n \times n$ matrices A and B, build a new matrix of size $2n \times 2n$:
   $$X = \begin{bmatrix} A & B \\ 0 & 0 \end{bmatrix}$$
   We can then compute the square of X using the given algorithm. The running time would be $O((2n)^c) = O(n^c)$, as c is a constant. Notice that
   $$X \times X = \begin{bmatrix} A & B \\ 0 & 0 \end{bmatrix} \times \begin{bmatrix} A & B \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} A^2 & AB \\ 0 & 0 \end{bmatrix}$$
   which gives the product of A and B. Therefore, we have an algorithm that can compute AB in $O(n^c)$ time.