Algorithm Design and Analysis: Assignment 1

钟赟 202028013229148

2020 年 10 月 28 日

Problem 1

Suppose an array sorted in ascending order is rotated at some pivot unknown to you be-forehand. (i.e., [0, 1, 2, 4, 5, 6, 7] is an ascending array, then it might be rotated and become [4, 5, 6, 7, 0, 1, 2].) How to find the minimum of a rotated sorted array? (Hint: All elements in the array are distinct.)

For example, the minimum of rotated sorted array [4, 5, 6, 7, 0, 1, 2] is 0.

Please give an algorithm with $\mathcal{O}(logn)$ complexity, prove the correctness and analyze the complexity.

Answer 1

1.1 Algorithm Description

We consider using binary search method to solve the problem. For array [0, 1, 2, 3, 4, 5, 6, 7], there are total 7 rotated forms as follows (the red number is the median) :

[0, 1, 2, 4, 5, 6, 7]
[7, 0, 1, 2, 4, 5, 6]
[6, 7, 0, 1, 2, 4, 5]
[5, 6, 7, 0, 1, 2, 4]
[4, 5, 6, 7, 0, 1, 2]
[2, 4, 5, 6, 7, 0, 1]
[1, 2, 4, 5, 6, 7, 0]

After selecting the median as a pivot, the key problem is to decide which part to search next step. We can see in the example that if the median is less than the element rightmost, the right part is sorted and the left part contains the minimum, the left part is sorted and the right part contains the minimum otherwise.

Suppose the length of array A is n. $A[i]$ is the minimum if and only if $A[i] == A[(i+n-1)\%n]$ – the element before the minimum is greater than it.

Here is my algorithm. For an array A with length n:

1. Select the median $A[i]$ of array A (or sub-array) as an pivot, $i = \frac{l+r}{2}$;

2. If $A[i] == A[(+n-1)\%n]$, $A[i]$ is the minimum. Otherwise goto step 3. ;
3. If $A[i] < A[rightmost]$, substitute A with A_{LEFT} . Goto step 1. ;
4. If $A[i] > A[rightmost]$, substitute A with A_{RIGHT} . Goto step 1.

Pseudo-code : see **Algorithm 1**.

Algorithm 1 MIN_ROTATED_ARRAY algorithm

```
1: function MIN_ROTATED_ARRAY(A, n, l, r)
2:     l = 0; r = n − 1;
3:     while l ≤ r do
4:         pivot = (l+r)/2;
5:         if A[pivot] < A[(pivot + n − 1)%n] then
6:             return A[pivot];
7:         end if
8:         if A[pivot] < A[r] then
9:             r = pivot;
10:        else
11:            l = pivot;
12:        end if
13:    end while
14:    return 0 ;
15: end function
```

1.2 Subproblem Reduction Graph

Take array [6,7,0,1,2,4,5] as an example, **Graph 1** shows the procedure of finding its minimum.

1.3 Correctness Proof

(1) The algorithm is clearly correct where $n = 1$ and $n = 2$.

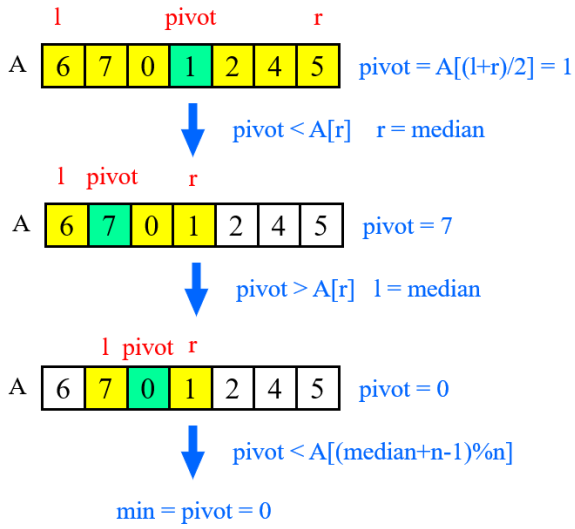
(2) Assuming that the algorithm is correct in the case of $n \leq k, k \geq 2$, let's think about cases when $n = k + 1$.

If $A[\lfloor \frac{n}{2} \rfloor] < A[n]$ and the index of the minimum element exists in $(\lfloor \frac{n}{2} \rfloor, n)$, then we have $minimum \leq A[n-1] < A[\lfloor \frac{n}{2} \rfloor]$ according to the ascending order before rotated, which results in contradiction. Threfore the minimum element is between $A[0]$ and $A[\lfloor \frac{n}{2} \rfloor]$. After one iteration, $A' = A[0, \lfloor \frac{n}{2} \rfloor]$. The size of $\mid A' \mid \leq [\frac{n}{2}] + 1 < k$, then we can always find the minimum by the assumption.

If $A[\lfloor \frac{n}{2} \rfloor] > A[n]$ and the index of the minimum element is in $[0, \lfloor \frac{n}{2} \rfloor]$, then we have $minimum \leq A[\lfloor \frac{n}{2} \rfloor] < A[n]$ according to the ascending order before rotated, which results in contradiction. Threfore the minimum element is between $A[\lfloor \frac{n}{2} \rfloor]$ and $A[n-1]$. After one iteration, $A' = A[\lfloor \frac{n}{2} \rfloor, n-1]$. The size of $\mid A' \mid \leq [\frac{n}{2}] + 1 < k$, then we can always find the minimum by the assumption.

(3) To sum up, the algorithm is correct for any array $A[n]$ with $n \geq 1$.

Graph 1



1.4 Complexity

The time complexity is $\mathcal{O}(logn)$ as binary search once costs time of $(logn)$.
The space complexity is $\mathcal{O}(1)$.

Problem 2

Consider an n-node complete binary tree T , where $n = 2^d - 1$ for some d. Each node v of T is labeled with a real number x_v. You may assume that the real numbers labeling the nodes are all distinct. A node v of T is a local minimum if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge. You are given such a complete binary tree T , but the labeling is only specifed in the following: implicit way: for each node v, you can determine the value x_v by probing the node v. Show how to find a local minimum of T using only $\mathcal{O}(logn)$ probes to the nodes of T .

Answer 2

2.1 Algorithm Description

In order to find the local minimum, we need to compare the label x_v of node v and the labels for nodes joined to v. Let's devide the complete binary tree T into 2 subtrees when searching from the root, and the size of the subproblems decreases exponentially. For node v, if it has no children, then $returnv$; or if x_v is more than the labels of v's children, then $returnv$; else if x_v is less than one of v's child's label, search the subtree rooted by the child with larger label recursively.

Pseudo-code : see **Algorithm 2**.

Algorithm 2 LOCAL_MIN_PROBE algorithm

```
1: function LOCAL_MIN_PROBE(T, v)
2:     if v has no child then
3:         return v;
4:     end if
5:     l = left child of v;
6:     r = right child of v;
7:     if x_v > x_l and x_v > x_r then return v ;
8:     else
9:         u = min (l, r);                    ▷ min(l, r) returns the node with less label
10:        return LOCAL_MIN_PROBE (T, u);
11:    end if
12: end function
```

2.2 Subproblem Reduction Graph

See **Graph 2**.

2.3 Correctness Proof

We already know that the number of tree T is $n = 2^d - 1$:

(1) When $d = 0, n = 1$, the root is the local minimum. The algorithm is clearly correct.

(2) Assuming that the algorithm is correct in the case of $d \leq k, k \geq 0$, let's think about cases when $d = k + 1$, that is the depth of the tree increase by 1.

If the label of root node is less than its children's, the root node returned by the algorithm is the local minimum. Otherwise if its child u has the smallest label, the algorithm would call LOCAL_MIN_PROBE(T, u). The subtree u has a depth $< d$ and a number of nodes $\leq 2^{d-1} - 1 = 2^k - 1$, thus the algorithm should return the local minimum of tree u, say w, by the assumption. If $u = w$, w is clearly the local minimum of tree T. If $u \neq w$, w is also the local minimum of tree T as the label of w is less than the labels for all nodes joined to w by an edge.

(3) To sum up, the algorithm is correct for given $d \geq 0$.

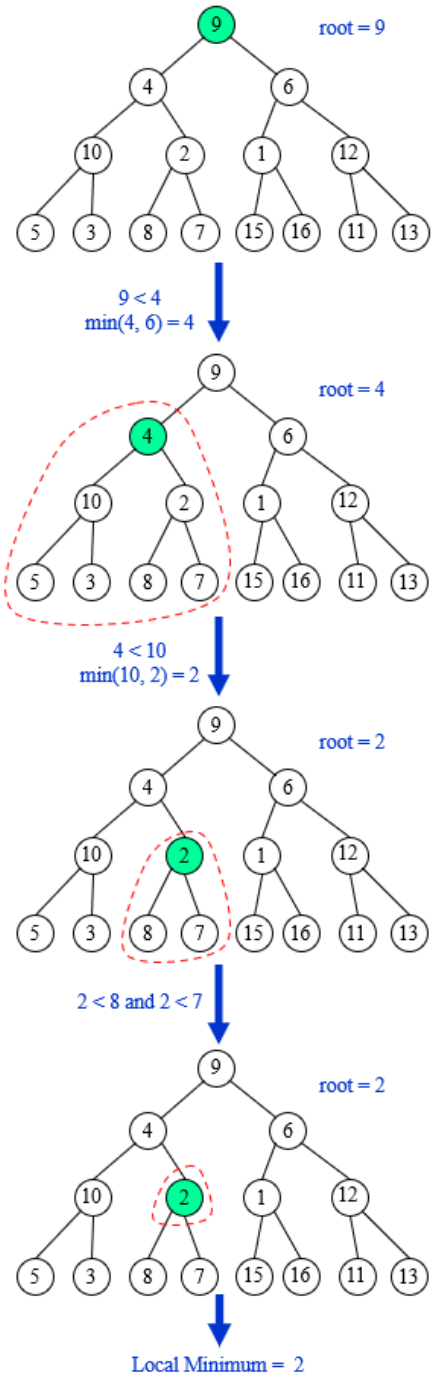
2.4 Complexity

According to the Master's Theorem, $T(n) = T(\frac{n}{2}) + 1 = \mathcal{O}(logn)$.

Problem 3

Given an integer array, one or more consecutive integers in the array form a sub-array. Find the maximum value of the sum of all subarrays.
Please give an algorithm with $\mathcal{O}(nlogn)$ complexity.

Graph 2


root = 9
9 < 4
min(4, 6) = 4
root = 4
4 < 10
min(10, 2) = 2
root = 2
2 < 8 and 2 < 7
root = 2
Local Minimum = 2

Answer 3

3.1 Algorithm Description

Similar to the algorithm used in CLOSEST_PAIRS problem, I decide to apply DEVIDE-CONQUER-COMBINE method to this problem.

DEVIDE: dividing the array into 2 roughly equal subarrays.

CONQUER: finding the maximum value of the sum of subarrays in each half.

COMBINE: considering subarrays consisting of elements from both left half and right half, and finding the real maximum sum.

We define a function MAX_SUM_SUBARRAY(A, l, r) to identify finding the the maximum sum of subarray $A[l, r]$. Each time we devide interval $[l, r]$ into 2 halves by $m = \lfloor \frac{l+r}{2} \rfloor$, and implement CONQUER in $A[l, m]$ and $A[m + 1, r]$, respectively. When the length of intervals are reduced to 1, we start to COMBINE information of $A[l, m]$ and $A[m + 1, r]$.

Some extra variants are needed to store information of subarrays:

$imax$: sum of all elements in $A[l, r]$;

$lmax$: the maximum sum of subarrays of $A[l, r]$ with $A[l]$ as the left endpoint;

$rmax$: the maximum sum of subarrays of $A[l, r]$ with $A[r]$ as the right endpoint;

$mmax$: the maximum sum of subarrays of $A[l, r]$.

While combining information, the variants iterate like this:

$imax_{[l,r]} = imax_{[l,m]} + imax_{[m+1,r]}$;

$lmax_{[l,r]}$ =MAX$(lmax_{[l,r]}, imax_{[l,m]} + lmax_{[m+1,r]})$;

$rmax_{[l,r]}$ =MAX$(rmax_{[l,r]}, rmax_{[l,m]} + imax_{[m+1,r]})$;

$mmax_{[l,r]}$ =MAX(MAX$(mmax_{[l,m]}, mmax_{[m+1,r]}), rmax_{[l,m]} + lmax_{[m+1,r]})$;

After combining all subarrays into the whole array, the $mmax_{[0,n-1]}$ returns the maximum sum of subarrays of A.

Pseudo-code : see **Agorithm 3**.

3.2 Subproblem reduction graph

Take array [-2, 1, -3, 4, -1, 2, 1] as an example, the subproblem reduction graph is seen in **Graph 3**.

3.3 Correctness Proof

(1) Suppose array A has length of n. The algorithm returns $A[0]$ where $n = 1$, which is clearly correct.

(2) We assume that the algorithm is correct in case of $n \leq k$, $k \geq 1$, and then consider cases of $n = k + 1$.

Algorithm 3 MAX_SUM_SUBARRAYS algorithm

```
1: function MAX_SUM_SUBARRAY(A, l, r)
2:     while l < r do
3:         m = ⌊(l+r)/2⌋;                                          ▷ DIVIDE procedure
4:         {imax_l, lmax_l, rmax_l, mmax_l} =MAX_SUM_SUBARRAY(A, l, m);      ▷ CONQUER
5:         {imax_r, lmax_r, rmax_r, mmax_r} =MAX_SUM_SUBARRAY(A, m + 1, r);
6:         imax = imax_l + imax_r;
7:         lmax =MAX(lmax_l, imax_l + lmax_r);                     ▷ COMBINE procedure
8:         rmax =MAX(rmax_r, rmax_l + imax_r);   ▷ MAX(a, b) returns the larger number of a and b
9:         mmax =MAX(MAX(mmax_l, rmax_r), rmax_l + lmax_r);
10:        return {imax, lmax, rmax, mmax} ;
11:    end while
12: end function
13: function MAX_SUM_SUBARRAYS(A)
14:    return MAX_SUM_SUBARRAY(A, 0, n);
15: end function
```

The algorithm firstly devide A into 2 subarrays – $A[0, \lfloor \frac{n}{2} \rfloor]$ and $A[\lfloor \frac{n}{2} \rfloor + 1, n-1]$, and MAX_SUM_SUBARRAY$(A, 0, \lfloor$ and MAX_SUM_SUBARRAY$(A, \lfloor \frac{n}{2} \rfloor, n-1)$ returns the maximum sum of their subarrays respectively by the assumption, denoted as $lmax_l, rmax_l, mmax_l$ and $lmax_r, rmax_r, mmax_r$. Then we have $mmax_{total}$ = MAX (MAX$(mmax_l, rmax_r), rmax_l + lmax_r)$. The subarray of A with maximum sum lies in either one of the subarrays or both halves, therefore the algorithm guarantees $mmax_{total}$ to be the maximum sum of A's subarrays.

(3) To sum up, the algorithm is correct.

3.4 Complexity

Time complexity: $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(1) = \mathcal{O}(n)$

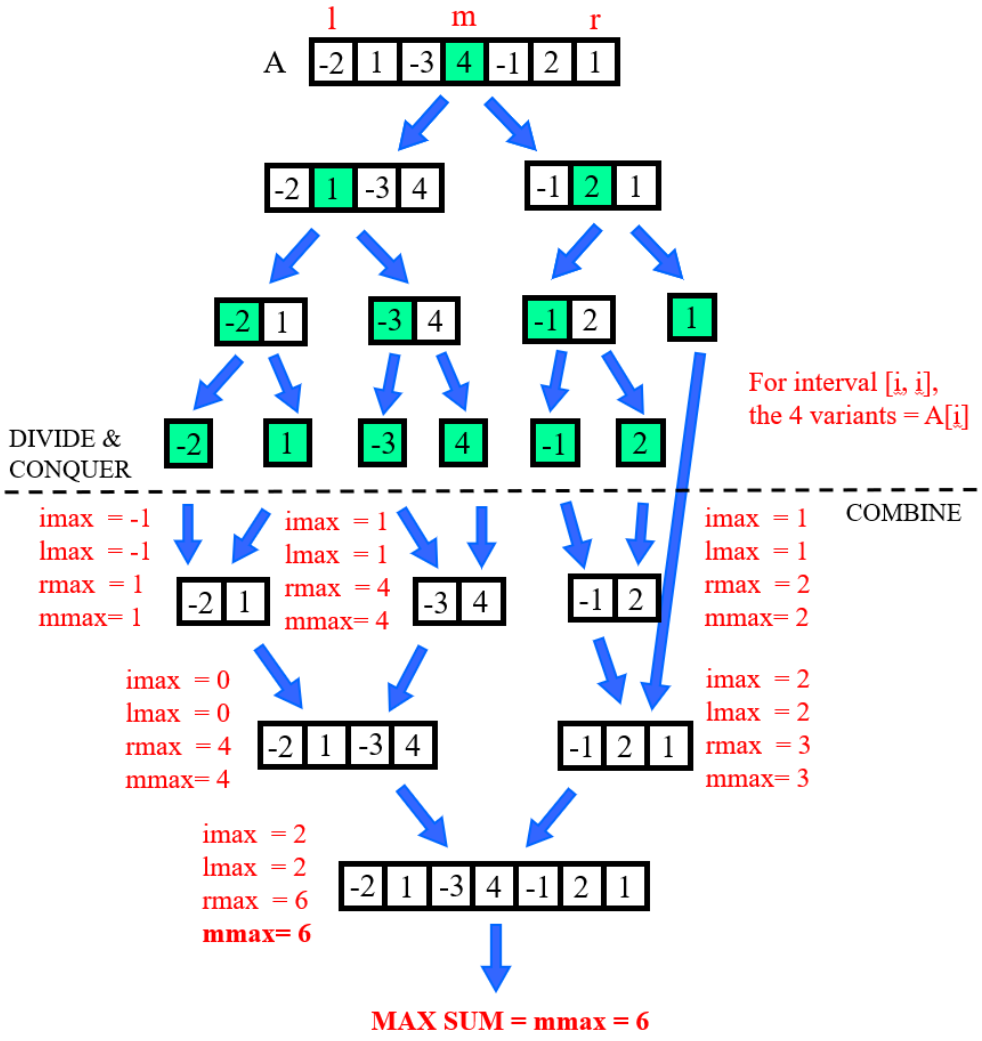
Space complexity: $\mathcal{O}(logn)$.

Problem 4

Given an array of integers nums sorted in ascending order, find the starting and ending position of a given target value. If the target is not found in the array, return [-1, -1]. For example, if the array is [5, 7, 7, 8, 8, 10] and the target is 8, then the output should be [3, 4].

Your algorithm's runtime complexity must be in the order of $\mathcal{O}(logn)$, prove the correctness and analyze the complexity.

Graph 3



Answer 4

4.1 Algorithm Description

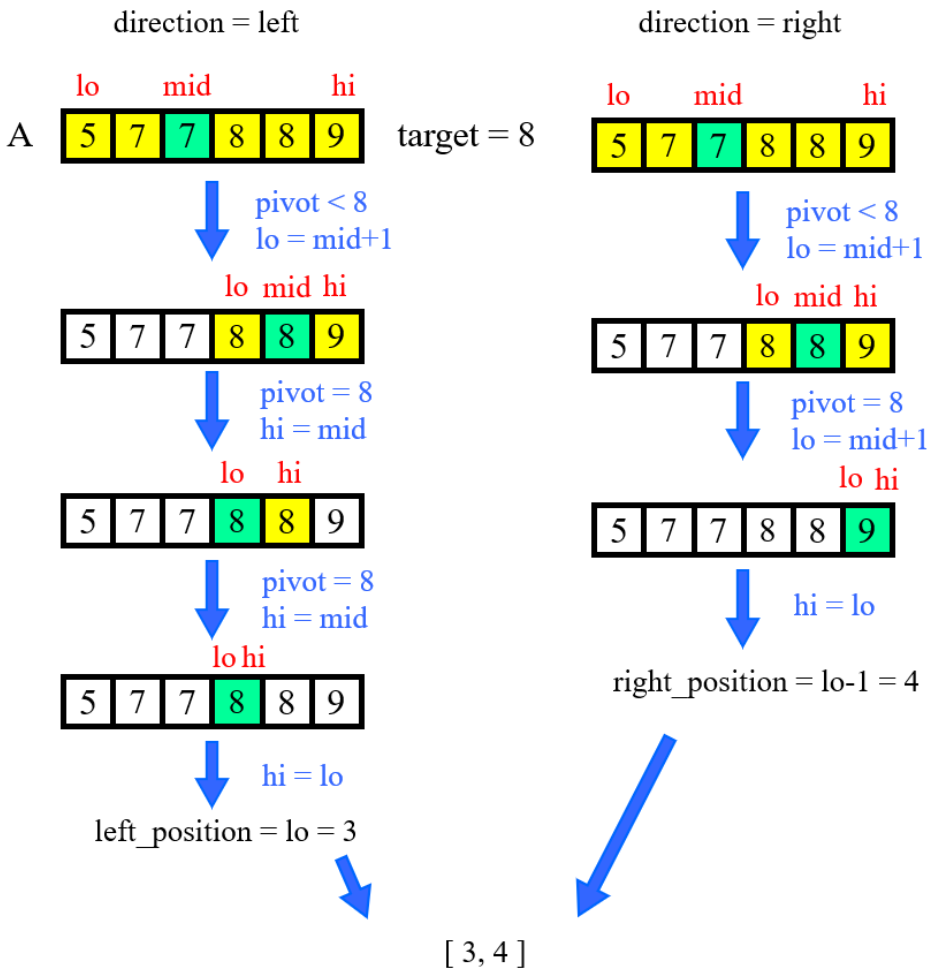
We consider finding the target value via binary search since the array is sorted. Now we have to deal with the cases of $pivot = target$, when the starting position and ending position of the target exist in either side of the pivot. Therefore, we have to search left and right , respectively. If we aim to get the starting position, searching in the left subarray; searching in the right subarray for ending position otherwise.

Pseudo-code : see **Algorithm 4**.

4.2 Subproblem Reduction Graph

See **Graph 4**.

Graph 4



4.3 Correctness Proof

(1) Suppose array A has length of n. The algorithm returns $[0, 0]$, where $n = 1$, which is clearly correct.

(2) We assume that the algorithm is correct in case of $n \leq k$, $k \geq 1$, and then consider cases of $n = k + 1$.

If $A[\lfloor \frac{n}{2} \rfloor] \geq target$, the starting position of target, if exists, must be in $A[0, \lfloor \frac{n}{2} \rfloor]$. The algorithm would return the starting position of target by calling SEARCH_FOR_RANGE $(A', target, "left")$, where $A' = A[0, \lfloor \frac{n}{2} \rfloor]$ with length $\lfloor \frac{n}{2} \rfloor + 1 < k$. Then we get the starting index of the target or -1.

In the same way, if $A[\lfloor \frac{n}{2} \rfloor] < target$, the starting position of target, if exists, must be in $A[\lfloor \frac{n}{2} \rfloor + 1, n-1]$. The algorithm would return the starting position of target by calling SEARCH_FOR_RANGE

$(A', target, "right")$, where $A' = A[\lfloor \frac{n}{2} \rfloor + 1, n-1]$ with length $n - \lfloor \frac{n}{2} \rfloor - 2 < k$. Then we get the starting index of the target or -1.

Therefore, the algorithm returns the right results when $n = k + 1$.

(3) To sum up, the algorithm is correct.

4.4 Complexity

Since binary search once costs time of $\mathcal{O}(logn)$ and the algorithm calls binary search twice, the total time complexity is $\mathcal{O}(logn)$.

Space complexity is $\mathcal{O}(1)$.

Problem 5

Given a convex polygon with n vertices, we can divide it into several separated pieces, such that every piece is a triangle. When $n = 4$, there are two different ways to divide the polygon; When $n = 5$, there are five different ways.

Give an algorithm that decides how many ways we can divide a convex polygon with n vertices into triangles.

Answer 5

5.1 Algorithm Description

The simplest case: a 3-vertice convex polygon has 1 different way to be divided. For each n-vertice convex polygon P, we consider deviding it into two sub-convex polygon, and recursively dividing until into the simplest cases. Let $d(n)$ indicate the number of ways that an n-vertice convex polygon has. Suppose polygon P has vertices $v_1, v_2, \ldots, v_n$ and edges $(v_i, v_j), i, j = 1, 2, \ldots, n; i \neq j$. Select a vertice v_k randomly, $k \in [2, n-1]$, add an edge (v_1, v_k) and (v_n, v_k) that divide P into a k-vertice convex polygon and a $n - k + 1$-vertice convex polygon. For each k, P has $d(k) * d(n - k + 1)$ different ways. So P has totally $\sum_{k=2}^{n-1} d(k) * d(n - k + 1)$ different ways.

Pseudo-code : see **Algorithm 5**. We only display one iteration of the loop of a pentagon.

5.2 Subproblem Reduction Graph

See **Graph 5**.

5.3 Correctness Proof

(1) The algorithm is clearly correct when $n = 3$;

(2) We assume that the algorithm is correct in case of $n \leq k$, $k \geq 1$, and then consider cases of $n = k + 1$.

Supposing that the convex polygon has vertices $v_1, v_2, \ldots, v_{k+1}$. Choose a vertice v_s randomly, $s \in [2, k]$. The triangle with vertices v_1, v_{k+1} and v_s devides the convex polygon into two parts: a polygon containing edge (v_1, v_s) and a polygon containing edge (v_{k+1}, v_s), thus $d(k+1) = \sum_{i=2}^{k} d(i) * d(k-i+2)$. Since $i \leq k$ and $k-i+2 \leq k$ for $i = 2, \ldots, k$, $d[i]$ and $d[k-i+2]$ are right computed by the assumption. Therefore the algorithm returns the correct result when $n = k+1$.

(3) To sum up, the algorithm is correct for $n \geq 3$.

5.4 Complexity

The algorithm checks all possible edges of the convex polygon, therefore its time complexity is $\mathcal{O}(n^2)$.

Problem 6

Recall the problem of nding the number of inversions. As in the course, we are given a sequence of n numbers $a_1, \ldots, a_n$, which we assume are all distinct, and we define an inversion to be a pair $i < j$ such that $a_i < a_j$.

We motivated the problem of counting inversions as a good measure of how different two orderings are. However, one might feel that this measure is too sensitive. Let's call a pair a significant inversion if $i < j$ and $a_i > 3a_j$. Given an $O(nlogn)$ algorithm to count the number of significant inversions between two orderings.

Answer 6

6.1 Algorithm Description

Since a significant inversion requires $i < j$ and $a_i > 3a_j$, we cannot count the number of significant inversions while $MERGE - SORT$. Based on the $INVERSION - COUNTING$ algorithm in the course, we only need an extra counting procedure for 2 sorted arrays in $\mathcal{O}(n)$ time.

The pseudo-code is shown in **Algorithm 6**.

6.2 Subproblem Reduction Graph

See **Graph 6**.

6.3 Correctness Proof

The algorithm, based on MERGE-SORT algorithm in the course, adds an extra function COUNT-SIGNIFICANT-INVERSION to count number of significant inversions. For any two sorted arrays, itetating through the arrays and doing comparisons do not change the structure of the array, therefore it does not change the correctness of the algorithm.

6.4 Complexity

According to the Master's Theorem, $T(n) = 2T(\frac{n}{2}) + cn = \mathcal{O}(nlogn)$.

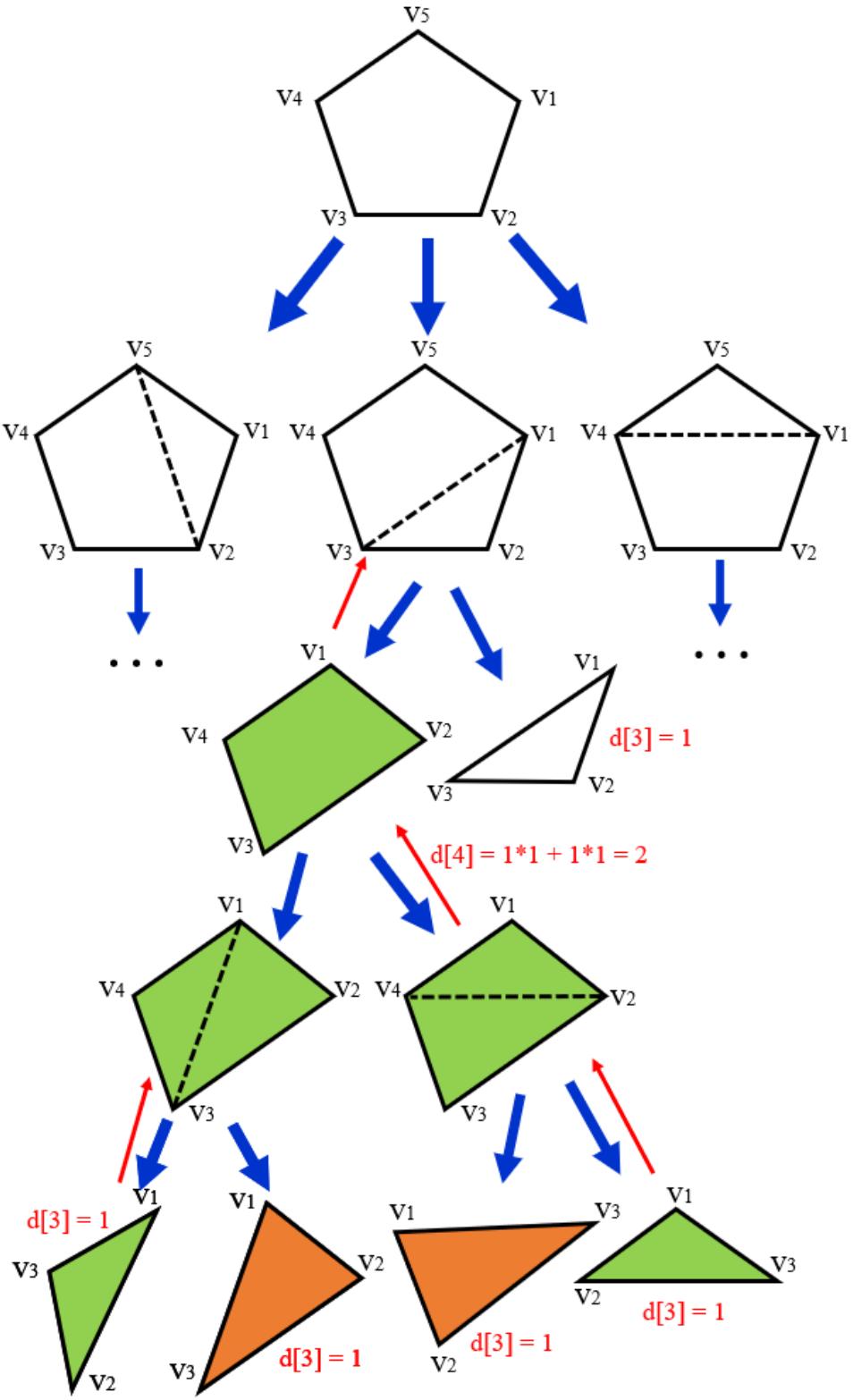
Algorithm 4 SEARCH_FOR_RANGE algorithm

```
1: function SEARCH_FOR_INDEX((A, target, direction))
2:     lo = 0;    hi = A.length;
3:     while lo < hi do
4:         mid = ⌊(lo+hi)/2⌋;
5:         if A[mid] > target then
6:             hi = mid;                                   ▷ Binary search
7:         else
8:             lo = mid + 1;
9:         end if
10:        if A[mid] == target then
11:            if direction == "left" then
12:                hi = mid;                               ▷ Search for the starting position
13:            else
14:                lo = mid + 1;                           ▷ Search for the ending position
15:            end if
16:        end if
17:    end while
18:    return lo ;
19: end function
20: function SEARCH_FOR_RANGE((A, target))
21:    starting_position = SEARCH_FOR_RANGE(A, target, "left");
22:    ending_position = SEARCH_FOR_RANGE(A, target, "right") − 1;
23:    if A[starting_position] ≠ target then
24:        return [−1, −1];
25:    else
26:        return [starting_position, ending_position];
27:    end if
28: end function
```

Algorithm 5 CONVEX-POLYGON-DIVIDING algorithm

```
1: function CONVEX-POLYGON-DIVIDING(P)
2:     P has n vertice;
3:     d[2] = 1; d[3] = 1;
4:     for k = 2 to n − 1 do
5:         d[n]+ = d[k] ∗ d[n − k + 1];
6:     end for
7:     return d[n];
8: end function
```

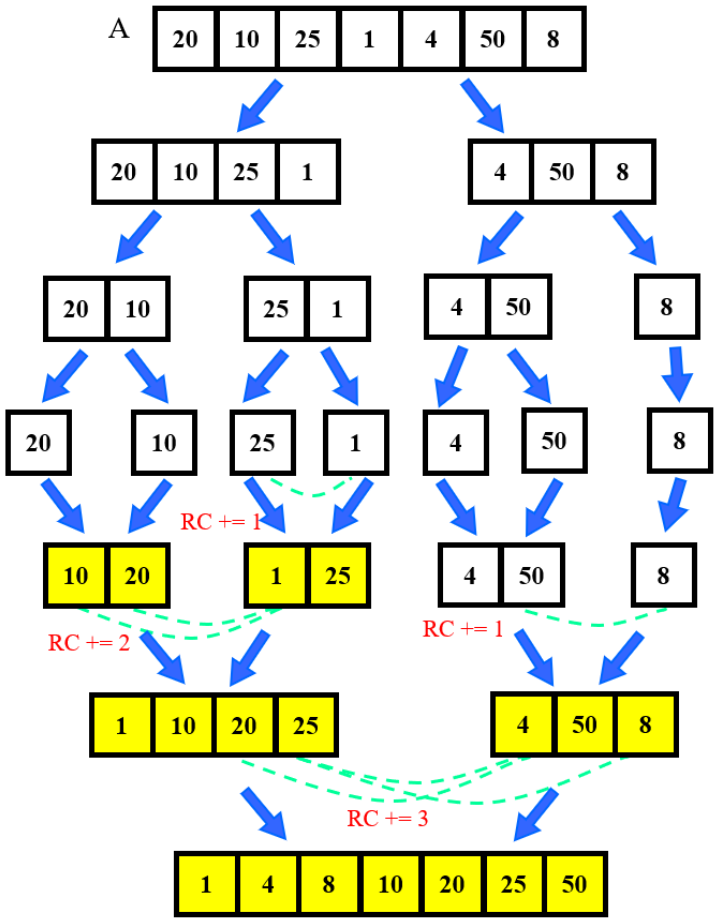
Graph 5



Algorithm 6 SIGNIFICANT-INVERSION-COUNTING algorithm

```
1: function SORT-AND-COUNT-INVERSION(A)
2:     if ‖ A ‖ == 1 then
3:         return ;
4:     end if
5:     Divide A into 2 sub-sequences L and R;
6:     (RC_L, L) = SORT-AND-COUNT-INVERSION(L);
7:     (RC_R, R) = SORT-AND-COUNT-INVERSION(R);
8:     (C, A) = MERGE(L, R);
9:     return (RC = RC_L + RC_R + C, A);
10: end function
11: function MERGE(L, R)
12:     i = 0; j = 0;
13:     RC = COUNT-SIGNIFICANT-INVERSION(L, R, A);
14:     for k = 0 to ‖ L ‖ + ‖ R ‖ −1 do
15:         if L[i] > R[j] then
16:             A[k] = R[j];
17:             j + +;
18:         else
19:             A[k] = L[i];
20:             i + +;
21:         end if
22:     end for
23:     return (RC, A);
24: end function
25: function COUNT-SIGNIFICANT-INVERSION(L, R)
26:     RC = 0;i = 0; j = 0;
27:     while i <‖ L ‖ and j <‖ R ‖ do
28:         if L[i] > 3 ∗ R[j] then
29:             RC+ = (‖ L ‖ −i);
30:             j + +;
31:         else
32:             j + +;
33:         end if
34:     end while
35: end function
```

Graph 6


A
20 10 25 1 4 50 8
20 10 25 1
4 50 8
20 10
25 1
4 50
8
20
10
25
1
4
50
8
RC += 1
10 20
1 25
4 50
8
RC += 2
RC += 1
1 10 20 25
4 50 8
RC += 3
1 4 8 10 20 25 50