# (2-4)

1. (1, 5)、(2, 5)、(3, 4)、(3, 5)、(4, 5)
2. 当元素从大到小排序时，逆序对最多，数量为**n\*(n-1)/2**
3. **成线性关系**。由插入排序过程知，设当前待插入元素为a[k],a[0]到a[k-1]有序，则k与前k-1个元素构成m个逆序对，则对应m次移动操作（将这些元素右移，留出空间给a[k]插入）

插入排序的每次插入操作对应于消除一个逆序对，且每个逆序对消除所需的时间是常数，因此插入排序的运行时间正比于逆序对数目。

1. 采用**分治策略**

**1.算法思想:**合并两个有序子数组时，如果从左侧子数组取出的元素大于从右侧子数组取出的元素，那么左侧子数组中该元素之后的所有元素都会与右侧子数组中的这个元素形成逆序对。因此，我们可以在合并的过程中统计这些逆序对。

**2.算法流程**

(1)初始化计数器：定义一个全局变量 count 来记录逆序对的数量，初始值为0。

(2)归并排序：对数组进行归并排序，但在合并的过程中加入逆序对的统计。

(3)合并过程：

创建两个指针，分别指向两个子数组的开始位置。

创建一个临时数组用于存放合并后的结果。

比较两个子数组的指针所指向的元素，将较小的元素添加到临时数组中，并移动对应的指针。

如果左侧子数组的元素大于右侧子数组的元素，那么左侧子数组中的这个元素与右侧子数组中所有尚未被合并的元素都会形成逆序对，因此将 count 加上右侧子数组中剩余的元素数量。

(4)递归合并：递归地对两个子数组进行上述合并过程，直到整个数组被排序。

(5)返回结果：归并排序完成后，count 中存储的就是整个数组的逆序对数量。

1. **代码**
2. #include <iostream>
3. #include <vector>
4. long long mergeAndCount(std::vector<int>& arr, std::vector<int>& temp, int left, int mid, int right) {
5. int i = left;    *// 左子数组起点*
6. int j = mid + 1; *// 右子数组起点*
7. int k = left;    *// 临时数组起点*
8. long long inv\_count = 0;
9. *// 合并两个已排序的子数组，并计数逆序对*
10. while (i <= mid && j <= right) {
11. if (arr[i] <= arr[j]) {
12. temp[k++] = arr[i++];
13. }
14. else {
15. temp[k++] = arr[j++];
16. inv\_count += (mid - i + 1); *// 统计逆序对数量*
17. }
18. }
19. *// 将剩余元素复制到临时数组*
20. while (i <= mid)
21. temp[k++] = arr[i++];
22. while (j <= right)
23. temp[k++] = arr[j++];
24. *// 将临时数组复制回原数组*
25. for (i = left; i <= right; i++)
26. arr[i] = temp[i];
27. return inv\_count;
28. }
29. long long mergeSortAndCount(std::vector<int>& arr, std::vector<int>& temp, int left, int right) {
30. long long inv\_count = 0;
31. if (left < right) {
32. int mid = left + (right - left) / 2;
33. *// 递归分治统计逆序对*
34. inv\_count += mergeSortAndCount(arr, temp, left, mid);
35. inv\_count += mergeSortAndCount(arr, temp, mid + 1, right);
36. *// 合并两个已排序部分，并统计跨越两个子数组的逆序对*
37. inv\_count += mergeAndCount(arr, temp, left, mid, right);
38. }
39. return inv\_count;
40. }
41. long long countInversions(std::vector<int>& arr) {
42. std::vector<int> temp(arr.size());
43. return mergeSortAndCount(arr, temp, 0, arr.size() - 1);
44. }
45. int main() {
46. std::vector<int> arr = { 2,3,8,6,1 };
47. std::cout << "Number of inversions: " << countInversions(arr) << std::endl;
48. return 0;
49. }

# (4.1-5)

1. MAX-SUBARRAY-LINEAR(A)
2. n = A.length
3. max-sum = -∞
4. ending-here-sum = -∞
5. for j = 1 to n
6. ending-here-high = j
7. if ending-here-sum > 0
8. ending-here-sum = ending-here-sum + A[j]
9. else ending-here-low = j
10. ending-here-sum = A[j]
11. if ending-here-sum > max-sum
12. max-sum = ending-here-sum
13. low = ending-here-low
14. high = ending-here-high
15. return (low, high, max-sum)

该算法是Kadane 算法的一种实现方式，通过动态更新当前和/最大和，在线性时间复杂度内实现求解

算法的核心思想是使用low 和 high来跟踪当前子数组的起始和结束位置，同时使用 ending-here-sum 来跟踪以当前元素结尾的最大子数组的和。如果 ending-here-sum 变为负数，则从下一个元素开始重新计算子数组的和，这是因为当ending-here-sum为负数时会使总和减小，显然不如直接从A[j]重新开始为好。

最终，返回子数组的起始和结束索引 low、high 以及最大和 max-sum

# (4.3-2)

替代法，假设有



又 

带入 

要使得下式成立



只需取c>=2,当n足够大时 可忽略不计、

递归树法：,故时间复杂度为0(lgn)

# (4.3-9)

代数变换，令



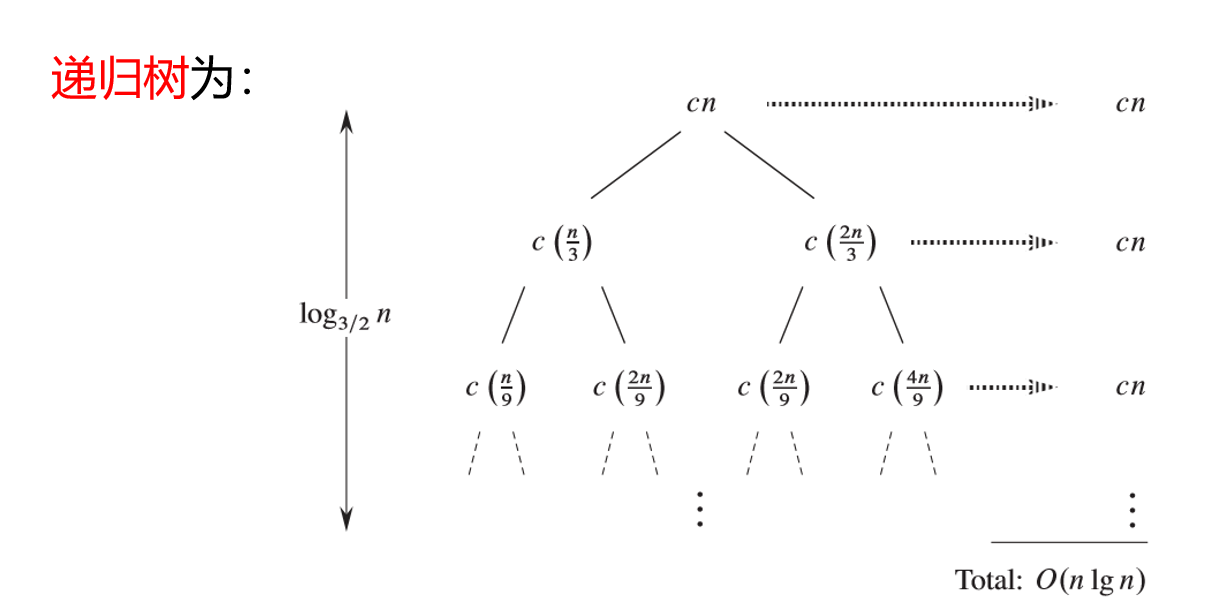
再令 

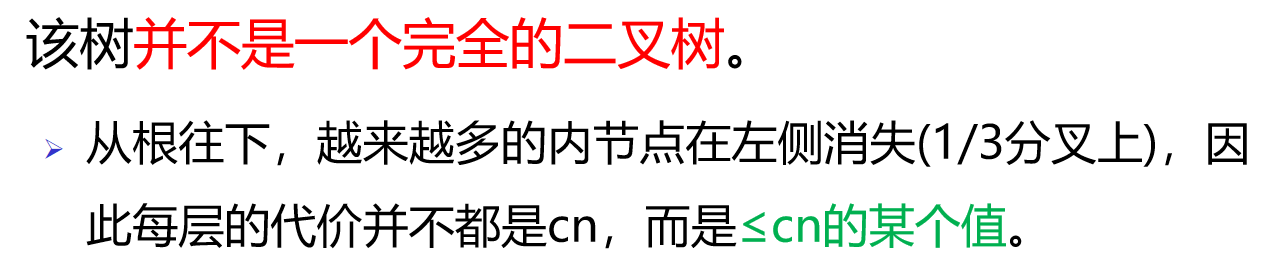
得

由主定理知解为

带回n得解为

# (4.4-6)





最左叶子节点得深度为,在此之前每层的代价都为cn,

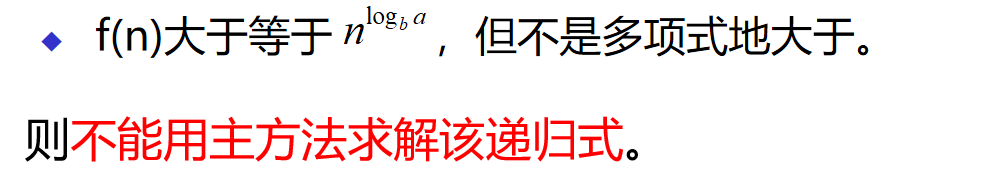
因此

即下界为nlgn

# (4.5-1)

1. ,高于f(n),渐进紧确界为
2. 同上，与f(n)同阶，需加上对数因子渐进紧确界为
3. 同上，阶数低于f(n),,渐进紧确界为
4. 同上，阶数低于f(n)，,渐进紧确界为

# (4.5-4)



两者相差lgn,不能表达成n的多项式形式，故不符合主定理条件。

将递归式展开并求和后，可得渐进上界为