5008 big O problem

1. O (1) constant time: when the time taken to complete a function does not increase at all as the size of the input increases
2. O (log ^n): binary search: divide and conquer approach,
3. O(n): linear time
4. O(n logn): if the content of an iteration loop is binary search, then the time complexity is O(nlogn)
5. O(n^2): quadratic time
6. O(2^n): recursive algorithms that solves a problem of size N by recursively solving two smaller problems of size N-1
7. O(n!)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Time complexity | | | Space complexity |
|  | Best | Average | Worst | Worst |
| Quicksort | n log(n) | n log(n) | O(n^2) | n log(n) |
| Merge sort | n log(n) | n log(n) | n log(n) | O(n) |
| Heap sort | n log(n) | n log(n) | n log(n) | O(1) |
| Bubble sort | O(n) | O(n^2) | O(n^2) | O(1) |
| Insertion sort | O(n) | O(n^2) | O(n^2) | O(1) |
| Selection sort | O(n^2) | O(n^2) | O(n^2) | O(1) |
| Tree sort | n log(n) | n log(n) | O(n^2) | O(n) |
|  |  |  |  |  |
|  |  |  |  |  |

Table

Description automatically generated

Bubble sort: 反复遍历列表，比较相邻的两个元素，如果顺序不对，则交换他们

|  |
| --- |
| void bubble\_sort(int arr[], int n) {  for (int i = 0; i < n - 1; i++) {  for (int j = 0; j < n - i - 1; j++) {  if (arr[j] > arr[j + 1]) {  int temp = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = temp;  }  }  }  } |

The outer loop runs n times, and inner loop runs n-i times, where I is the current iteration of the outer loop. The total number of iteration is roughly n\*(n-1)/2 , so time complexity is O(n^2).

The best case, when the array is already sorted, bubble sort will have linear complexity O(n)

Selection sort: 将输入列表分为两部分，以排序的子列表在列表的前端，以及带排序的列表。最初以排序的列表是空的，未排序的列表是整个列表。算法通过在待排序子列表中查找最小的元素，将其与左边的带排序元素交换，并将已排序子列表边界向右移动一个元素。

|  |
| --- |
| void selection\_sort(int arr[], int n) {  for (int i = 0; i < n - 1; i++) {  int min\_idx = i;  for (int j = i + 1; j < n; j++) {  if (arr[j] < arr[min\_idx]) {  min\_idx = j;  }  }  int temp = arr[min\_idx];  arr[min\_idx] = arr[i];  arr[i] = temp;  }  } |

Same with bubble sort: Two nested loops, the outer loop runs n times, and inner loop runs (n-i) times, where I is the current iteration of the outer loop. The total number of iteration is roughly n\*(n-1)/2 , so time complexity is O(n^2) in all cases

Insertion sort: 只需要一次遍历， 在大列表中的效率比更高级的算法低。 代码实现简单，对小的数据集更有效。空间复杂度低只有O（1）

|  |
| --- |
| void insertion\_sort(int arr[], int n) {  for (int i = 1; i < n; i++) {  int key = arr[i];  int j = i - 1;  while (j >= 0 && arr[j] > key) {  arr[j + 1] = arr[j];  j--;  }  arr[j + 1] = key;  }  } |

Merge sort:将为排序的列表划分为两个元素数量相同的子数组。 排序这两个子数组， 再将他们合并。

|  |
| --- |
| void merge(int arr[], int left[], int left\_size, int right[], int right\_size) {  int i = 0, j = 0, k = 0;  while (i < left\_size && j < right\_size) {  if (left[i] <= right[j]) {  arr[k++] = left[i++];  } else {  arr[k++] = right[j++];  }  }  while (i < left\_size) {  arr[k++] = left[i++];  }  while (j < right\_size) {  arr[k++] = right[j++];  }  }  void merge\_sort(int arr[], int n) {  if (n < 2) {  return;  }  int mid = n / 2;  int left[mid];  int right[n - mid];  for (int i = 0; i < mid; i++) {  left[i] = arr[i];  }  for (int i = mid; i < n; i++) {  right[i - mid] = arr[i];  }  merge\_sort(left, mid);  merge\_sort(right, n - mid);  merge(arr, left, mid, right, n - mid);  } |

Is a divide and conquer algorithms.

Array is divided into two halves at each level of recursion until the base case (1 element) is reached. The array is merged back together. The depth of the recursion tree is log(n). so at each level n elements are merged. Time complexity is O(n\*log(n)) in all cases.

Quick sort： 从数组中选择一个元素， 称为pivot。 对数组进行排序，使所有小于pivot的元素都位于pivot之前，所有大于pivot的元素都位于pivot之后。递归将上述步骤应用于pivot之前和之后的子数组。 排序不稳定，想等的值在排序前后顺序可能会改变。无法两好应对已经排序好的情况。

|  |
| --- |
| int partition(int arr[], int low, int high) {  int pivot = arr[high];  int i = low - 1;  for (int j = low; j <= high - 1; j++) {  if (arr[j] < pivot) {  i++;  int temp = arr[i];  arr[i] = arr[j];  arr[j] = temp;  }  }  int temp = arr[i + 1];  arr[i + 1] = arr[high];  arr[high] = temp;  return i + 1;  }  void quick\_sort(int arr[], int low, int high) {  if (low < high) {  int pivot\_idx = partition(arr, low, high);  quick\_sort(arr, low, pivot\_idx - 1);  quick\_sort(arr, pivot\_idx + 1, high);  }  } |

Time complexity depends on the choice of the pivot. In the best and average cases, the pivot splits the array into roughly equal parts, leading to a balanced recursion tree with a depth of log(n), time complexity is O(n\*log(n)). Worst case is when the pivot is the smallest or the largest element, the recursion tree becomes unbalanced, and the depth is n, resulting in a time complexity of O(n^2)

Heap sort：就地排序， 但不是稳定排序。利用堆这种数据结构设计的一种排序算法， 堆积是一个近似完全二叉树的结构。子节点的键值总是小于父节点。

|  |
| --- |
| void heapify(int arr[], int n, int i) {  int largest = i;  int left = 2 \* i + 1;  int right = 2 \* i + 2;  if (left < n && arr[left] > arr[largest]) {  largest = left;  }  if (right < n && arr[right] > arr[largest]) {  largest = right;  }  if (largest != i) {  int temp = arr[i];  arr[i] = arr[largest];  arr[largest] = temp;  heapify(arr, n, largest);  }  }  void heap\_sort(int arr[], int n) {  for (int i = n / 2 - 1; i >= 0; i--) {  heapify(arr, n, i);  }  for (int i = n - 1; i >= 0; i--) {  int temp = arr[0];  arr[0] = arr[i];  arr[i] = temp;  heapify(arr, i, 0);  }  } |

heap sort builds a binary heap (max heap) from the input array, which takes O(n) time, then repeatedly extracts the maximum element from the heap and swap it with the last element in the heap. Then restore the heap property by calling heapify on the new root. This process is repeated n times for each element in the heap. Since the time complexity of heapify is O(log(n)), and preform this operation n times, the time complexity is O(n\*log(n)).