Chapter 1

排序算法

1.1 Quick Sort

1.1.1 性能

时间复杂度

Average	Worst
$O(n \cdot \log n)$	$O(n^2)$

Worst case: In the most unbalanced case, a single Quicksort call involves $O(n)$ work plus two recursive calls on lists of size 0 and $n-1$, so the recurrence relation is:

$$\begin{aligned} T(n) &= O(n) + T(0) + T(n-1) \\ &= O(n) + T(n-1) \\ &= O(n^2) \end{aligned} \tag{1.1}$$

Average: In the most balanced case, a single quicksort call involves $O(n)$ work plus two recursive calls on lists of size $n/2$, so the recurrence relation is:

$$\begin{aligned} T(n) &= O(n) + 2T(\frac{n}{2}) \\ &= O(n \log n) \end{aligned} \tag{1.2}$$

空间复杂度

@TODO

1.1.2 实现

```
int partition(vector<int>& data, int low, int high)
{
  int pivot = data[high];
  int small = low - 1;
  for (int i = low; i < high; ++i)
  {
    if (data[i] < data[high])
    {
      small++;
      if (small != i)
        swap(data[i], data[small]);
    }
  }
  ++small;
  swap(data[high], data[small]);
  return small;
}

void quicksort(vector<int>& data, int low, int high)
{
  if (low < high)
  {
    int k = partition(data, low, high);
    quicksort(data, low, k-1);
    quicksort(data, k+1, high);
  }
}
```

Listing 1.1: 算法导论中的实现

1.2 Merge Sort

1.2.1 实现

1.3 Heap Sort

1.3.1 性能

时间复杂度

Average	Worst
$O(n \cdot \log n)$	$O(n \cdot \log n)$

1.3.2 实现

下面的代码是采用"sift down" 来做heapify的实现。heapify的过程是一个自底向上(bottom-up)过程：从最后一个父节点(start节点)开始，用"sift down"保持从这个节点开始后面的所有节点是个heap，然后向上移动这个(start)节点，也就是start减1, 直到0。

```
void sift_down(vector<int>& data, int start, int end)
{
  int root = start;
  while (root * 2 + 1 <= end)
  {
    int child = root * 2 + 1;
    int s = root;
    if (data[s] < data[child])
      s = child;
    if (child + 1 <= end && data[s] < data[child+1])
      s = child + 1;
    if (s != root)
    {
      swap(data[root], data[s]);
      root = s;
    }
    else
```

```
      return;
    }
}

void heapify(vector<int>& data)
{
  int n = data.size();
  int start = (n - 2) / 2;  // last parent node
  while (start >= 0)
  {
    sift_down(data, start, n - 1);
    start--;
  }
}

void heap_sort(vector<int>& data)
{
  heapify(data);
  int n = data.size();
  int end = n - 1;
  while (end >= 0)
  {
    swap(data[end], data[0]);
    end--;
    sift_down(data, 0, end);
  }
}
```

Listing 1.2: Heap Sort

下面的代码是采用"sift up" 来做heapify的实现。heapify2的过程是一个自顶向下(top-down)过程：从第一个子节点(start节点)开始，用"sift up"保持从这个节点开始前面的所有节点是个heap，然后向下移动这个(start)节点，也就是start加1, 直到count-1。

```
void sift_up(vector<int>& data, int start, int end)
{
  int child = end;
  int parent = -1;
  while (child > start)
```

```
  {
    parent = (child - 1) / 2;
    if (data[parent] < data[child])
    {
      swap(data[parent], data[child]);
      child = parent;
    }
    else
      return;
  }
}

void heapify2(vector<int>& data)
{
  int n = data.size();
  int end = 1;  // first left child
  while (end < n);
  {
    sift_up(data, 0, end);
    end++;
  }
}
```

Listing 1.3: Heap Sort

”sift down”版本的heapify的时间复杂度是$O(n)$, 而”sift up”版本的heapify2的时间复杂度是$O(n \log n)$。从直觉上说这个差别来自于heapify2 是自顶向下，节点数增加深度递增，”sift up” 中swap的操作就增加。而第一个heapify是自底向上，深度递减，swap的操作也递减。

Chapter 2

搜索算法

2.1 Binary Search

2.1.1 实现

```
int bsearch(const vector<int>& data, int key)
{
  int low = 0;
  int high = data.size() - 1;
  while (low <= high)
  {
    int mid = low + ((high - low) >> 1);
    if (key < data[mid])
      high = mid - 1;
    else if (key > data[mid])
      low = mid + 1;
    else
      return mid;
  }
  return -1;
}
```

Listing 2.1: Iterative Implementation

```
int bsearch(vector<int>& data, int key, int low, int high)
{
```

```
  if (low > high)
    return -1;

  int mid = low + ((high - low) >> 1);
  if (key < data[mid])
    binary_search(data, key, low, mid-1);
  else if (key > data[mid])
    binary_search(data, key, mid+1, high);
  else
    return mid;
}
```

Listing 2.2: Recursive Implementation

Chapter 3

数组相关问题

Chapter 4

链表相关问题

Chapter 5

二叉树相关问题