第 2 章 算法基础

2.1 插入排序

INSERTION-SORT(A)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1..j-1].
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i+1] = A[i]
7          i = i - 1
8      A[i+1] = key
```

$T(n) = \Theta(n^2)$

Ex2.1-2 重写过程 INSERTION-SORT, 使之按非升序排序.

INSERTION-SORT(A)

```
1  for j = 2 to A.length
2      key = A[j] A[1..j-1].
3      i = j - 1
4      while i > 0 and A[i] < key
5          A[i+1] = A[i]
6          i = i - 1
7      A[i+1] = key
```

Ex2.1-3 线性查找问题.

LINEAR-FIND(A, v)

```
1  for j = 1 to A.length
2      if A[j] == v
3          return j
4  return NIL
```

Ex2.1-4 二进制数加法

BINNARY-NUMBER-ADDITION(A, B, n)

```
1  Let C[1..n+1] be a new array.
2  r = 0
3  for i = n downto 1
4        C[i] = (A[i] + B[i] + r)%2
5        r = (A[i] + B[i] + r)/2
6  C[1] = r
7  return C
```

2.2 分析算法

Ex2.2-2: 选择排序 首先找到 A 中最小元素并与 A[1] 进行交换, 然后找出次最小元素并与 A[2] 进行交换...

SELECTION-SORT(A)

```
for i = 1 to A.length - 1
      min = i
      for  j = i + 1 to A.length
            if A[j] < A[min]
                  min = j
      Exchange A[min] and A[i]
```

2.3 设计算法

MERGE(A, p, q, r)

```
 1  n_1 = q - p + 1
 2  n_2 = r - q
 3  let L[1..n_1 + 1] and R[1..n_2 + 1] be bew arrays
 4  for i = 1 to n_1
 5      L[i] = A[p + i - 1]
 6  for j = 1 to n_2
 7      R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else
17          A[k] = R[j]
18          j = j + 1
```

MERGE-SORT(A, p, r)

```
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

利用递归树, 可以得知 MERGE-SORT 的算法复杂度为 $\Theta(nlgn)$

Ex2.3-2 重写过程 MERGE-SORT, 使之不使用哨兵, 而是一旦数组 L 或 R 的所有元素均被复制回 A 就立刻停止, 然后把另一个数组的剩余部分复制回 A.

MERGE-SORT-WITHOUT-GUARD(A, p, q, r)

```
1  n_1 = q - p + 1
2  n_2 = r - q
3  Let L[1..n_1] and R[1..n_2] be new arrays.
4  for i = 1 to n_1
5      L[i] = A[p + i - 1]
6  for j = 1 to n_2
7      R[j] = A[q + j]
8  i = 1
9  j = 1
10 k = p
11 while i ≤ n_1 and j ≤ n_2
12     if L[i] ≤ R[j]
13         A[k] = L[i]
14         i = i + 1
15     else
16         A[k] = R[j]
17         j = j + 1
18     k = k + 1
19 while i ≤ n_1
20     A[k] = L[i]
21     i = i + 1
22 while j ≤ n_2
23     A[k] = R[j]
24     j = j + 1
```

Ex2.3-4 我们可以把插入排序表示为如下的一个递归过程。为了排序 $A[1..n]$, 我们递归地排序 $A[1..n-1]$, 然后把 $A[n]$ 插入已排序的数组 $A[i..n-1]$。为插入排序的这个版本的最坏情况运行时间写一个递归式。

INSERTION(A, p)

```
1  key = A[p]
2  i = p - 1
3  while i > 0 and A[i] < key
4      A[i + 1] = A[i]
5      i = i - 1
6  A[i + 1] = key
```

INSERTION-SORT-RECURSIVE(A, p)

```
1  if p > 1
2      INSERTION-SORT-RECURSIVE(A, p − 1)
3      INSERTION(A, p)
```

其递归式如下：

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(n-1) + \Theta(n) & \text{if } n = 2 \end{cases}$$

Ex2.3-5 假设 A 已排好序，为二分查找写出迭代式或递归的伪代码。证明二分查找的最坏情况运行时间为 $\Theta(lgn)$

BINARY-SEARCH-RECURSIVE$(A, key, low, high)$

```
1  if low ≤ high
2      mid = ⌊(low + high)/2⌋
3      if key < A[mid]
4          return BINARY-SEARCH-RECURSIVE(A, key, low, mid − 1)
5      elseif key > A[mid]
6          return BINARY-SEARCH-RECURSIVE(A, key, mid + 1, high)
7      else
8          return mid
9  else
10     return NIL
```

该算法的递归式为 $T(n) = T(n/2) + \Theta(1)$，由递归树可知，其算法复杂度为：$\Theta(lg(n))$

Ex2.3-6 注意到过程 INSERTION-SORT 的第 5-7 行的 while 循环采用一种线性查找来 (反向) 扫描已排好序的子数组 $A[1..j-1]$。我们可以使用二分查找来把插入排序的最坏情况总运行时间改进到 $\Theta(nlgn)$ 吗？

先将上面的 BINARY-SEARCH-RECURSIVE 算法修改为如下形式:

BINARY-SEARCH-RECURSIVE$(A, key, low, high)$

```
1  if low ≤ high
2      mid = ⌊(low + high)/2⌋
3      if key < A[mid]
4          return BINARY-SEARCH-RECURSIVE(A, key, low, mid − 1)
5      elseif key > A[mid]
6          return BINARY-SEARCH-RECURSIVE(A, key, mid + 1, high)
7      else
8          return mid
9  else
10     return low
```

INSERTION-SORT-USING-BINARY-SEARCH(A)

```
1  for j = 2 to A.length
2      key = A[j]
3      i = j − 1
4      index = BINARY-SEARCH-RECURSIVE(A, key, 1, i)
5      if A[index] < key
6          index = index + 1
7      while i ≥ index
8          A[i + 1] = A[i]
9          i = i − 1
10     A[i + 1] = key
```

可见，在最坏情况下，就算使用了 BINARY-SEARCH-RECURSIVE 算法，也不能将其运行时间降低到 $\Theta(nlgn)$。

2.3-7 描述一个运行时间为 $/Theta(nlgn)$ 的算法，给定 n 个整数的集合 S 和另一个整数 x，该算法能确定 S 中是否存在两个其和刚好为 x 的元素。

先用归并排序对 S 进行排序，其运行时间为 $\Theta(nlgn)$，再使用二分查找算法查找 $x-e$，其运行时间为 $\Theta(nlgn)$，总运行时间为 $\Theta(nlgn)$。

思考题

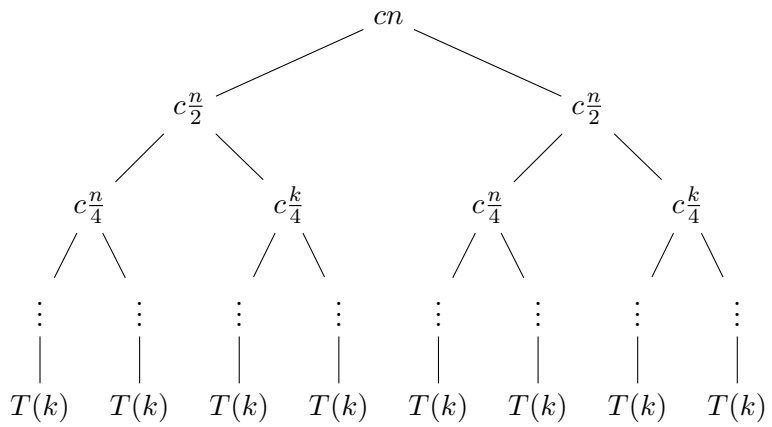
2-1 在归并排序中对小数组采用插入排序，虽然归并排序在最坏情况运行下时间为 $\Theta(nlgn)$，而插入排序的最坏情况运行时间为 $\Theta(n^2)$，但是在插入排序中的常量银子可能使得它在 n 较小时，在许多机器上实际运行得更快。因此，在归并排序中当 n 较小时，采用插入排序来使递归的叶变粗是有意义的。考虑对归并排序的一种修改，其中使用插入排序来排序长度为 k 的 n/k 个子表，然后使用标准的合并机制来合并这些子序列，这里 k 是一个待定的值。

a. 证明：插入排序最坏情况可以在 $\Theta(nk)$ 时间内排序每个长度为 k 的 n/k 个子表。

根据插入排序的特点，排序长度为 k 的序列在最坏情况下的运行时间为 $\Theta(k^2)$，那么排序 n/k 个这样的子序列则需要 $\frac{n}{k}\Theta(k^2) = \Theta(nk)$。

b. 表明在最坏情况下如何在 $\Theta(nlg(\frac{n}{k}))$ 时间内合并这些子表。

首先画出其递归树:



可知，该递归树总共有 $lg(\frac{n}{k}) + 1$ 层，除了最后一层的代价为 $\Theta(nk)$ 外，其他各层的代价都是 cn，从最后一层向上都是在合并，所以总代价为:$cnlg(\frac{n}{k}) = \Theta(nlg(\frac{n}{k}))$