序列和集合的算法II

Instructor: Shizhe Zhou
Course Code: 00125401
排序算法

桶排序
插入排序
选择排序
归并排序
quickSort
pivot
partition
排序算法

桶排序

插入排序

选择插入排序

归并排序

NOT In-place

quickSort

In-place

pivot

partition

Comparison-based sort
桶排序和基数排序

Bucket and Radix Sort

算法 `Straight_Radix (X, n, k)`

输入：`X`（元素下标从 1 至 `n` 的整数数组，每个元素有 `k` 位）
输出：`X`（排序后的数组）

begin

  We assume that all elements are initially in a global queue `GQ`;
  \{为简单起见，这里使用 `GQ`；当然也可以通过 `X` 本身来实现\}
  for `i := 1` to `d` do
  {`d` 是可能的数制，比如对于十进制，`d = 10`}
    `Initialize queue Q[I] to be empty;`
    for `i := k` downto `1` do
      while `GQ` is not empty do
        pop `x` from `GQ`;
        `d := the ith digit of x`;
        insert `x` into `Q[d]`;
        for `t := 1` to `d` do
          insert `Q[t]` into `GQ`;
        for `i := 1` to `n` do
          pop `X[i]` from `GQ`
  end

end

图 6.6 算法 `Straight_Radix`
对最高位相同的任意两个元素，在最后一步之前，他们已经按正确顺序排列了。
从高到低，如果直接按上一页 slides 的方法做，结果肯定是错的！必须要按递归做，见下一页，
从高到低，必须按递归式，结果才正确。
Analysis

• 重收集桶的循环执行 K 次，K 是最大数的位数 (e.g.10 进制下)
• 需要全局队列 GQ 和 分桶队列 Q[0]...Q[K-1].
• \( O((n+n)k) = O(nk) \);
插入排序

insertion sort

• example

Pseudocode

for i ← 1 to length(A)
    j ← i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j ← j - 1
The running time of Shellsort is heavily dependent on the gap sequence it uses. *Best interval? open questions!*

Shellsort is **unstable** *(changing relative order of equal elements)*. It is an **adaptive sorting algorithm** in that it executes faster when the input is partially sorted.

The subarrays that Shellsort operates on are initially **short**; later they are **longer but almost ordered**: In BOTH cases insertion sort works efficiently.
选择排序

selection sort

• example

基本思想:
每次都从右边剩下的 n-k 个数中选最小 (大) 的插入到第 k 个位置处.

输入 list 被分为两部分，前一部分总是排好序的 sublist.

将 64 换成 24, 则最后一步也发生交换

```c
/* a[0] to a[n-1] is the array to sort */
int i, j;
int iMin;

/* advance the position through the entire array */
/* (could do j < n-1 because single element is also min element) */
for (j = 0; j < n-1; j++) {
    /* find the min element in the unsorted a[j .. n-1] */
    /* assume the min is the first element */
    iMin = j;
    /* test against elements after j to find the smallest */
    for (i = j+1; i < n; i++) {
        /* if this element is less, then it is the new minimum */
        if (a[i] < a[iMin]) {
            /* found new minimum; remember its index */
            iMin = i;
        }
    }

    /* iMin is the index of the minimum element. Swap it with the current position */
    if (iMin != j) {
        swap(a[j], a[iMin]);
    }
}
```
选择排序

- 如果输入数据结构是 (linked) list, 性能更好。

![选择排序示意图](image)
Finally, selection sort is greatly outperformed on larger arrays by \( \Theta(n \log n) \) divide-and-conquer algorithms such as mergesort.

However, insertion sort or selection sort are both typically faster for small arrays (i.e. fewer than 10–20 elements).

A useful optimization in practice for the recursive algorithms is to switch to insertion sort or selection sort for "small enough" sublists.
top-down merge sort

```c
int TopDownMergeSort(int A[], int B[], int n)
{
    TopDownSplitMerge(A, 0, n, B);
}

void TopDownSplitMerge(int A[], int iBegin, int iEnd, int B[]) {
    int iMiddle = (iEnd + iBegin) / 2; // iMiddle = mid point
    TopDownSplitMerge(A, iBegin, iMiddle, B); // split / merge left half
    TopDownSplitMerge(A, iMiddle, iEnd, B); // split / merge right half
    TopDownMerge(A, iBegin, iMiddle, iEnd, B); // merge the two half runs
    CopyArray(B, iBegin, iEnd, A); // copy the merged runs back to A
}

void TopDownMerge(int A[], int iBegin, int iMiddle, int iEnd, int B[]) {
    int i0 = iBegin, i1 = iMiddle;
    int i;

    // While there are elements in the left or right runs
    for (; i0 < iMiddle || i1 < iEnd; i++) {
        // If left run head exists and is <= existing right run head.
        if (i0 < iMiddle && (i1 >= iEnd || A[i0] <= A[i1])) {
            B[i] = A[i0];
            i0 = i0 + 1;
            i1 = i1 + 1;
        } else {
            B[i] = A[i1];
            i1 = i1 + 1;
        }
    }
}
```

Major drawback:
Need a B for the merged list!
Merge sort

- **Bottom-Top**

**Bottom up merge sort algorithm:**
1. Treats the list as an array of $n$ sublists (called *runs* in this example) of size 1, 2.
2. Iteratively merges sub-lists back and forth between two buffers.
Merge Sort 的一个改进

- Natural merge sort

| Start  | 3--4--2--1--7--5--8--9--0--6 |
| Select runs | 3--4 2 1--7 5--8--9 0--6 |
| Merge  | 2--3--4 1--5--7--8--9 0--6 |
| Merge  | 1--2--3--4--5--7--8--9 0--6 |
| Merge  | 0--1--2--3--4--5--6--7--8--9 |

Any naturally occurring runs (sorted sequences) in the input are exploited!

Adavantage: 不需要像标准 merge sort 那样非得 pass lbn 次.
## MergeSort Performance

<table>
<thead>
<tr>
<th>Class</th>
<th>Sorting algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data structure</td>
<td>Array</td>
</tr>
<tr>
<td><strong>Worst case performance</strong></td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td><strong>Best case performance</strong></td>
<td>$O(n \log n)$ typical,</td>
</tr>
<tr>
<td></td>
<td>$O(n)$ natural variant</td>
</tr>
<tr>
<td><strong>Average case performance</strong></td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td><strong>Worst case space complexity</strong></td>
<td>$O(n)$ auxiliary</td>
</tr>
</tbody>
</table>
Quick Sort

• Best sorting algo ! Proposed by C.A.R Hoare in 1962
• Divide-&-Conquer algo
• In place (as insertion sort), comparison based
• Very practical (need a bit tuning though)
Partition (相向而行版本)

**Algorithm Partition** (X, Left, Right);

**Input**: X (an array), Left (the left boundary of the array), and Right (the right boundary).

**Output**: X and Middle such that X[i] ≤ X[Middle] for all i ≤ Middle and X[j] > X[Middle] for all j > Middle.

begin

pivot := X[Left] ;
L := Left ; R := Right ;
while L < R do

while X[L] ≤ pivot and L ≤ Right do L := L + 1 ;
while X[R] > pivot and R ≥ Left do R := R - 1 ;
if L < R then

exchange X[L] with X[R] ;
Middle := R ;
exchange X[Left] with X[Middle]

end

**Figure 6.9** Algorithm Partition.
Partition Example

Pivot = 6

Left = $L$

R = Right

If $L < R$ then
exchange $X[L]$ with $X[R]$.

Left

Middle = R

exchange $X[Left]$ with $X[Middle]$.
Analysis

• Worst Case – input sorted (reversed sorted) – $O(n^2)$
• Best Case: uniform split 1:1 – $O(n \log n)$
• 2nd-Worst Case: 1:9 split, using recursion tree – $O(n \log n)$
• Lucky-Unlucky switch case: $O(n \log n)$
• Average case (Randomized QuickSort): Substitution method – $O(n \log n)$.
To make things more practical

• Randomized QuickSort
  – Rearranging the element randomly
  – Selecting the pivot randomly
  – $\geq 3$ times faster than mergeSort!
  – Works well with cache and virtual mem.
Analysis of Randomized QuickSort

• We don’t know which exactly split is made. Each recursive step uses random pivot!

• So, we compute the Expectation of the $T(n)$ in terms of a random variable $X_k$.

\[ X_k = \begin{cases} 
1 & \text{if PARTITION generates a } k : n-k-1 \text{ split,} \\
0 & \text{otherwise.} 
\end{cases} \]

\[ E[X_k] = \Pr\{X_k = 1\} = 1/n, \text{ since all splits are equally likely, assuming elements are distinct.} \]
Expectation of $T(n)$

$$T(n) = \begin{cases} 
T(0) + T(n-1) + \Theta(n) & \text{if 0 : } n-1 \text{ split,} \\
T(1) + T(n-2) + \Theta(n) & \text{if 1 : } n-2 \text{ split,} \\
\vdots \\
T(n-1) + T(0) + \Theta(n) & \text{if } n-1 : 0 \text{ split,} 
\end{cases}$$

$$= \sum_{k=0}^{n-1} X_k \left( T(k) + T(n-k-1) + \Theta(n) \right)$$

$$E[T(n)] = E \left[ \sum_{k=0}^{n-1} X_k \left( T(k) + T(n-k-1) + \Theta(n) \right) \right]$$

Linearity of expectation:

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]$$

$$= \frac{1}{n} \sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n} \sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n} \sum_{k=0}^{n-1} \Theta(n) = \frac{2}{n} \sum_{k=1}^{n-1} E[T(k)] + \Theta(n)$$

Independence of $X_k$ from other random choices.
Prove by substitution method

\[ E[T(n)] = 2 \sum_{k=2}^{n-1} E[T(k)] + \Theta(n) \]

(The \( k = 0, 1 \) terms can be absorbed in the \( \Theta(n) \).)

**Prove:** \( E[T(n)] \leq an \log n \) for constant \( a > 0 \).
- Choose \( a \) large enough so that \( an \log n \) dominates \( E[T(n)] \) for sufficiently small \( n \geq 2 \).

**Use fact:** \[ \sum_{k=2}^{n-1} k \log k \leq \frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \] (exercise).
Prove by substitution method

\[
E[T(n)] \leq 2 \sum_{k=2}^{n-1} ak \lg k + \Theta(n)
\]

\[
\leq \frac{2a}{n} \left( \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \right) + \Theta(n)
\]

\[
= an \lg n - \left( \frac{an}{4} - \Theta(n) \right)
\]

Express as \textit{desired} – \textit{residual}.