Data Structures

Sorting

CS284
Objectives

- To learn how to implement the following sorting algorithms:
  - selection sort
  - bubble sort
  - insertion sort
  - shell sort
  - merge sort
  - heapsort
  - quicksort

- To understand the differences in performance of these algorithms, and which to use for small, medium arrays, and large arrays
Shell Sort: A Better Insertion Sort
Shell Sort: A Better Insertion Sort

- Insertion sort takes $O(n^2)$ time
  - In the worst case, needs $O(n^2)$ comparisons/swaps
  - Disadvantage: swap distance can only be 1
  - Can we improve the time complexity if we allow long-distance swaps?

- Shell sort: long distance insertion sort

- History of shellsort:
  - It is named after its discoverer, Donald Shell
  - The time complexity depends on the actual distance being used
  - $O(n^{3/2})$ is a common bound for its time complexity
  - People have improved this bound over the years, by constructing different distance series
Disadvantage of Insertion Sort

1st round:
SORTEXAMPLE

......

6-th round:
EORSTXXAMPLE

......

- Each time can only swap by distance-1

```java
public void insertion_step(E[] a, int this_idx, int stride) {
    E this_val = a[this_idx];
    while (this_idx >= stride && this_val.compareTo(a[this_idx - stride]) < 0) {
        a[this_idx] = a[this_idx - stride];
        this_idx-= stride;
    }
    a[this_idx] = this_val;
}
```

- What if we can swap by longer distance?
Swap distance/stride $h$

<table>
<thead>
<tr>
<th>Stride</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>S O R T E X A M P L E</td>
</tr>
<tr>
<td>3</td>
<td>S O R T E X A M P L E S O R T E X A M P L E</td>
</tr>
</tbody>
</table>

7-sequence: S M O P R L T E E X A

3-sequence: S T A L O E M E R X P

Shell sort Algorithm

```java
public void shell_sort(E[] table) {
    int[] gap_seq = {5, 3, 1};
    for (int h : gap_seq) {
        for (int pos = 1; pos < table.length; pos++) {
            insertion_step(table, pos, h);
        }
    }
}
```

swap count = 17
Shell sort Algorithm

```java
public void shell_sort(E[] table) {
    int[] gap_seq = {5, 3, 1};
    for (int h : gap_seq) {
        for (int pos = 1; pos < table.length; pos++) {
            insertion_step(table, pos, h);
        }
    }
}
```

swap count = 17

```java
public void insertion_sort(E[] table) {
    for (int pos = 1; pos < table.length; pos++) {
        insertion_step(table, pos, 1);
    }
}
```

swap count = 36, how does Shell sort require fewer swaps while having more loops?
Shell sort: execution trace

stride = 7

SORTEXAMPLE
MORTEXASPLE
MORTEXASPLE
MOLTEXASPRE
MOREEXASPLT

stride = 3

3-seq #0 MOREEXASPLT
3-seq #0 EORMEXASPLT
3-seq #1 EORMEXASPLT
3-seq #2 EERMOXASPLT
3-seq #0 EERMOXASPLT
3-seq #0 EERMOXASPLT
3-seq #0 EERMOXASPLT
3-seq #0 (inserting A to E M)
3-seq #0 AEREOXMSPLT
3-seq #1 AEREOXMSPLT
3-seq #2 AEREOXMSPLT
3-seq #2 AEREOXMSPLT
3-seq #2 AEREOXMSPLT
3-seq #2 AEREOXMSPLT
3-seq #2 AEREOXMSPLT
3-seq #2 (inserting P to R X)
3-seq #2 AEPEORMSXLT
3-seq #0 AEPEORMSXLT
3-seq #1 AEPEORLSXMT
Analysis of Shell Sort

- Why is shell sort correct? When gap = 1, reduce to insertion sort
- How does Shell sort reduce # swaps and # comparisons?
  - Answer: the fact that the array is being 5-sorted and 3-sorted makes the algorithm require fewer swaps/comparisons in 1-sorting
- $h$-sort: the process of sorting all the $h$-sequence
**Proposition.** After an array is $h$-sorted then $k$ sorted ($k < h$), the array remains $h$-sorted

**Proof.** We can prove the proposition by contradiction.

Suppose the proposition is false, that means after $k$ sorting, at least one pair of stride-$h$ elements are reversed, i.e., position $i$’s value $> \text{position } i + h$’s value. Suppose $(i, i + h)$ is the first time for this to happen.

*Note & notation: The change happen due to the latest insertion operation in either $x_i$ or $x_{i+h}$’s sequence, but not both. When it happens to one sequence $\cdots, x_l, \cdots$, we use $x_l|$ and $|x'_l$ to denote the before-after values of affected positions $l$. For any position $k$ whose value is unchanged, we use $x_k$ to denote its value.*
Proposition

Before the $k$ sorting, the array was $h$ sorted, and now $(i, i + h)$ values are reversed. This means one of the following two things must have happened during the $k$ sorting: (1) the latest position is at $x_i$’s sequence, and $x_i$ just increased ($|x_i > x_i|$), or (2) the latest position is at $x_{i+h}$’s sequence, and $x_{i+h}$’s just decreased ($|x_{i+h} < x_{i+h}|$).

(1) Suppose it’s the first case. Notice in the process of $k$ insertion sorting, any element can move at most $1 \times k$ position. Most of the time, the value at a position would decrease, the only case of increase is when $x_i$ is the latest position, and it’s replaced by the value before it, e.g., $x_i | = A$ and $|x_i = M$: 
Thus $|x_i = x_{i-k}|$, e.g., $|x_6 = x_3| = M$. Because $(i, i + h)$ is the first time for the reversion to happen, $x_{i-k} < x_{i-k+h}$; meanwhile, $x_{i+h}$ and $x_{i-k+h}$ are in the same $k$ sequence, so when the $k$ sort arrives at position $i + h$ later, $x_{i+h}$ will be replaced by the largest value in this sequence, which $\geq x_{i-k+h} > x_{i-k} = |x_i|$, thus eventually the reversion will not happen, i.e., case (1) is eliminated.
Proposition

(2) Suppose it’s the second case. Due to insertion sort, when $x_{i+h}$’s value is decreased, it must be due to the insertion of the latest visited element $x_{j+h}$ at its sequence, e.g., $x_6 = A$ is inserted upfront which makes the value of $x_0 = E$ and $x_3 = M$ decrease, thus $j > i$, and $x_{j+h} \leq |x_{i+h} < x_i$.

3-seq #0 E E R M O X A S P L T
(inserting A to E M)

3-seq #0 A E R E O X M S P L T
Meanwhile because the value at position $j + h$ has increased, it wouldn’t cause a reversion at position $(j, j + h)$ (unless $x_j$ had increased even more, in which case the violation of $x_j > |x_{j+h} > x_{j+h}|$ means the reversion of case (1) would already happened as early as position $j$, which contradicts with the assumption that $(i, i + h)$ is the first time when the violation happens).

As a result, $x_j < x_{j+h} \leq |x_{i+h} < x_i$, but because $j > i$ and $j$ has already been visited, $x_i$ and $x_j$ should have been sorted, so we have a contradiction, i.e., case (2) is eliminated.
Implication of proposition

- Proposition means, if we first 5 sort the array then 3 sort the array, the array will be both 3-sorted and 5-sorted.
- We can prove that, when an array is both 3 sorted and 5 sorted, #comparison/swap needed by the final 1 sorting is reduced to linear (o/w will be quadratic).
- This property is due to the fact that 3 and 5 are mutually prime numbers.
Complexity of 1-sorting a (3,5)-sorted array

**Theorem.** The \#swaps/comparison of 1 sorting an array that is both 3 sorted and 5 sorted is \(O(N)\).

**Proof.** After the 3 sorting, consider every 3 consecutive values \(x_{3i}, x_{3i+1}, x_{3i+2}\), and how many \#swap/comparison they need in total.

Because the array is 3 sorted, \(x_{3i} > x_{3i-3}, x_{3i-6}, \cdots\); meanwhile, because it is 5 sorted, \(x_{3i} > x_{3i-5}, x_{3i-8}, \cdots\), and \(x_{3i} > x_{3i-10} > x_{3i-13} \cdots\), so the only values that could be smaller than \(x_{3i}\) are: \(x_{3i-1}, x_{3i-2}, x_{3i-4}, x_{3i-7}\). Similarly, we can show there are also at most 4 values that are smaller than \(x_{3i+1}\) and \(x_{3i+2}\), thus the reversed \#pairs are at most \(O(N)\).
Complexity of l-sorting a \((h,k)\)-sorted array

**Theorem (Sedgewick 1996).** The \#swaps/comparison of \(l\)-sorting an array that is both \(h\) sorted and \(k\) sorted is \(O(hkN)\), where \(h\) and \(k\) are mutually prime numbers.

**Proof.** If \(h\) and \(k\) are mutually prime numbers where \(k < h\), we can prove the series of \(h\%k, 2h\%k, \ldots, (k - 1)h\%k\) must be \(k - 1\) unique values (proof: \(h = ak + c, ih\%k = ic\%k\), if \((i - j)c\%k = 0\), it means \(c\) is a factor of \(k\), contradicts with the fact that \(k\) and \(h\) are mutually prime).

So \(x_{ki}\) is only larger than at most \(h/k + 2h/k + \cdots, (k - 1)h/k = (k - 1)h/2\) numbers, thus at most \((k - 1)h/2l\) numbers in each \(l\)-sequence, so the total number of swaps/comparison is of complexity \(O(hkN/l)\).
Estimating the time complexity of Shell sort

Start from two large numbers $h$ and $k$, the complexity of sorting are $(N/h)^2 + (N/k)^2$, followed by a list of linear complexity, e.g., the complexity for $(h, k, 1)$ sort is $O((N/h)^2 + (N/k)^2 + hkN)$, so when $h = k = N^{1/4}$, it will be $O(N^{3/2})$.

**Tight bounds:** the bound depends on the gap sequence. Over the years, people have proved tighter bounds such as $O(N^{4/3})$.
More readings

Sedgewick’s paper: http://thomas.baudel.name/Visualisation/VisuTri/Docs/shellsort.pdf