# CSCI 335
# Software Design and Analysis III
# Lecture 17:
# Sorting: Insertion sort, Shellsort

Professor Anita Raja

11-03-22

# Announcement

- Midterm grades to be released
- HW3 due 11/10.

# Agenda

- Leftist heap theorem
- Merge leftist algorithm: recursive, non-recursive
- Skew heap
- Binomial Queue
- Insertion Sort
- Shell Sort
- Midterm discussion

# Chapter 7 Sorting

# Sorting algorithms

- Several easy algorithms to sort in $O(N^2)$
  - Insertion sort
- Sorting algorithm simple to code, $o(N^2)$and efficient
  - Shell sort
- There are slightly more complicated algorithms
  - $O(N\log N)$
- Any general purpose sorting algorithm
  - $\boldsymbol{\Omega}(N\log N)$ comparisons

# Sorting algorithms

- Comparison-based sorting
- Void sort(Iterator begin, Iterator end, Comparator cmp);
- STL versions:
  1. sort(v.begin(), v.end() );
  2. sort(v.begin(), v.begin() + (v.end()-v.begin())/2);
  3. sort(v.begin(),v.end(),greater<int>{ });
- Also: stable_sort()
  - Does not swap the positions of two elements with the same comparison key.

# Insertion Sort

- Pass p=1 through N-1:

  items in positions 0 through p are in sorted order.

| Original | 34 \| | 8 | 64 | 51 | 32 | 21 | Positions Moved |
|---|---|---|---|---|---|---|---|
| After $p=1$ | 8 | 34 \| | 64 | 51 | 32 | 21 | 1 |
| After $p=2$ | 8 | 34 | 64 \| | 51 | 32 | 21 | 0 |
| After $p=3$ | 8 | 34 | 51 | 64 \| | 32 | 21 | 1 |
| After $p=4$ | 8 | 32 | 34 | 51 | 64 \| | 21 | 3 |
| After $p=5$ | 8 | 21 | 32 | 34 | 51 | 64 \| | 4 |

**General Strategy: In pass p, move the element in position p left until its correct place is found among the first p+1 elements**

# Insertion Sort

```
template <typename Comparable>
void InsertionSort(vector<Comparable> & a) {
    for (int p = 1; p < a.size(); ++p) {
        Comparable tmp = std::move(a[p]);
        int j;
        for (j = p; j > 0 && tmp < a[j - 1]; --j)
            a[j] = std::move(a[j - 1]);
        a[j] = std::move(tmp);
    }
}
```

# Stl implementation

```
// Two-parameter version calls three-parameter version.
// Uses C++11 decltype.
template <typename Iterator>
void InsertionSort(const Iterator &begin, const Iterator
&end) {
  InsertionSort(begin, end, less<decltype(*begin)>{});
}
```

# Converting Insert routine to STL Implementation

- Two parameter sort uses pair of iterators (start and endmarker) and assumes items can be ordered.
- 1.Two parameter sort invokes three-parameter sort with less<Object> as third parameter
  - Template type parameters for two-param sort are both Iterator; (generic type but Object is not a generic type parameter
  - C++11 introduces decltype which cleanly expresses the intent.
- 2. Array indexing with use of the iterator and that replaces calls to operator< with calls to the lessThan function object.
- 3. Once the insertSort algorithm is coded,
  - every statement in the original code is replaced with a corresponding statement in new code that makes straightforward use of iterators and the function object.

# Stl implementation

```
// Three-parameter version.
template <typename Iterator, typename Comparator>
void InsertionSort(const Iterator &begin, const Iterator &end,
Comparator
less_than) {
  if (begin == end) return;
  for (Iterator p = begin + 1; p != end; ++p) {
    auto tmp = std::move(*p);
    Iterator j;
    for (j = p; j != begin && less_than(tmp, *(j – 1)); --j)
      *j = std::move(*(j – 1));
    *j = std::move(tmp);
}
---
You can call for instance:
InsertionSort(vec.begin(), vec.end()); // vec is of type
vector<string>
```

# Insertion Sort Analysis

- Worst case: $\Theta(N^2)$
- Best case (presorted input): $O(N)$
  - One comparison per item

# Lower bound for simple sorting algorithms

Inversion in an array **a** is an ordered pair (i, j), 0 <= i, j <**a**.size():

i < j but **a**[i] > **a**[j]

Example array: 34, 8, 64, 51, 32, 21

9 inversions: (34,8), (34,32), (34,21),
(64,51), (64,32), (64,21),
(51,32),(51,21),
(32,21)

--------------------------------------------------------------------

These are exactly the swaps performed by insertion sort!

=> Swapping two adjacent items that are out of place reduces one inversion

=> Sorted array needs no inversions

# Lower bound for simple sorting algorithms

- Average number of inversions in permutation of N integers (assume no duplicates) provides

Average running time for insertion sort and for **all algorithms that are based on swapping adjacent elements !**

# Lower bound for simple sorting algorithms assumptions

- no duplicate elements.
  - can assume that the input is some permutation of the first N integers since only relative ordering is important
- All elements are equally likely.

- Given these assumptions, we have the following theorem:

# Lower bound proof for simple sorting algorithms

- **Theorem:** Average number of inversions in an array of n distinct elements is n(n-1)/4
- **Proof:**

  Consider list L of n integers and its reverse Lr.

  For every pair of indices (i,j), either (i,j) is an inversion in L or Lr.

  Therefore the total number of inversions for both lists is $\binom{n}{2} = \frac{n(n-1)}{2}$

  Average number of inversions across all lists $\frac{n!}{2n!}\binom{n}{2} = \frac{n(n-1)}{4}$

- **Therefore:** Any algorithm that sorts by exchanging adjacent elements is **$\Omega(n^2)$ on average**.

**Result valid for an entire class of algorithms that perform only adjacent exchanges! Insertion, Bubble, Selection**

# Take-away

- **Lower bound shows for a sorting algorithm to run in subquadratic or o(N^2) times:**
  - **it must do comparisons and in particular exchanges between elements that are far apart.**
- **Sorting algorithm makes progress by eliminating inversions**
  - **to run efficiently it must eliminate more than just one inversion per exchange.**

# Shellsort (by Donald Shell)

- Improves average time by exchanging non-adjacent elements
- **Diminishing increment sort**
- Uses increment sequence $h_1=1, h_2, \ldots, h_t$
  - All sequences work, but some perform better!
- Idea:

For k = t down to 1

[all elements spaced $h_k$ apart are sorted]

[ $\mathbf{a}[i] \leq \mathbf{a}[i+h_k]$ for every i ]

[perform insertion sort in $h_k$ subarrays]

--each subarray has size approximately $\mathbf{n/h_k}$

# Shellsort

| Array index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Original | 81 | 94 | 11 | 96 | 12 | 35 | 17 | 95 | 28 | 58 | 41 | 75 | 15 |
| After 5-sort | 35 | 17 | 11 | 28 | 12 | 41 | 75 | 15 | 96 | 58 | 81 | 94 | 95 |
| After 3-sort | 28 | 12 | 11 | 35 | 15 | 41 | 58 | 17 | 94 | 75 | 81 | 96 | 95 |
| After 1-sort | 11 | 12 | 15 | 17 | 28 | 35 | 41 | 58 | 75 | 81 | 94 | 95 | 96 |

At increment $h_k$:

for every position $i=h_k$, $h_k+1$, $h_k+2$, …, N-1

find its right spot among i, $i-h_k$, $i-2h_k$, $i-3h_k$ , …

-----------------------------------------------------------------

Note that at $h_1$ we have regular insertion sort

# Shellsort

| Array index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Original | 81 | 94 | 11 | 96 | 12 | 35 | 17 | 95 | 28 | 58 | 41 | 75 | 15 |
| After 5-sort | 35 | 17 | 11 | 28 | 12 | 41 | 75 | 15 | 96 | 58 | 81 | 94 | 95 |
| After 3-sort | 28 | 12 | 11 | 35 | 15 | 41 | 58 | 17 | 94 | 75 | 81 | 96 | 95 |
| After 1-sort | 11 | 12 | 15 | 17 | 28 | 35 | 41 | 58 | 75 | 81 | 94 | 95 | 96 |

For example for $h_k = 3$:

for i = 3, 3 + 1, 3 + 2, …, **a**.size() -1

find its right spot among i, i - 3, i – 6, i - 9 , … (i>=0)

---

# 5-sort: 5 “subarrays”

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 81 | 94 | 11 | 96 | 12 | 35 | 17 | 95 | 28 | 58 | 41 | 75 | 15 |
| 81 | | | | | 35 | | | | | 41 | | |
| **35** | | | | | **41** | | | | | **81** | | |
| | 94 | | | | | 17 | | | | | 75 | |
| | **17** | | | | | **75** | | | | | **94** | |
| | | 11 | | | | | 95 | | | | | 15 |
| | | **11** | | | | | **15** | | | | | **95** |
| | | | 96 | | | | | 28 | | | | |
| | | | **28** | | | | | **96** | | | | |
| | | | | 12 | | | | | 58 | | | |
| | | | | **12** | | | | | **58** | | | |
| **35** | **17** | **11** | **28** | **12** | **41** | **75** | **15** | **96** | **58** | **81** | **94** | **95** |

# 3-sort: 3 “subarrays”

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 35 | 17 | 11 | 28 | 12 | 41 | 75 | 15 | 96 | 58 | 81 | 94 | 95 |
| 35 | | | 28 | | | 75 | | | 58 | | | 95 |
| **28** | | | **35** | | | **58** | | | **75** | | | **95** |
| | 17 | | | 12 | | | 15 | | | 81 | | |
| | **12** | | | **15** | | | **17** | | | **81** | | |
| | | 11 | | | 41 | | | 96 | | | 94 | |
| | | **11** | | | **41** | | | **94** | | | **96** | |
| **28** | **12** | **11** | **35** | **15** | **41** | **58** | **17** | **94** | **75** | **81** | **96** | **95** |

# 1-sort: 1 "subarray"

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 28 | 12 | 11 | 35 | 15 | 41 | 58 | 17 | 94 | 75 | 81 | 96 | 95 |
| **11** | **12** | **15** | **17** | **28** | **35** | **41** | **58** | **75** | **81** | **94** | **95** | **96** |

- "subarrays" are conceptual (i.e. no need for extra storage)
- Implementation is extremely simple
- Best algorithm for small collections (up to 10000 elements)
- No recursion, good for embedded systems with small stack space.

# Shellsort

- Increment sequence
- $h_k$ sorted
- Although this does not affect the implementation, a careful examination of the action of an $h_k$ sort is to performa an insertion sort on $h_k$ independent subarrays!

- Original increment sequence:
  $h_t = \lfloor N / 2 \rfloor$, $h_k = \lfloor h_{k+1} / 2 \rfloor$

  Not a good selection though…

--Example for an array **a** of size N = 100

$h_6 = 50$, $h_5 = 25$, $h_4 = 12$, $h_3 = 6$, $h_2 = 3$, $h_1 = \mathbf{1}$.

# Shellsort

```
// Shellsort, using Shell's (poor) increments.
template <typename Comparable>
void ShellSort(vector<Comparable> &a) {
  // gap is Shell’s increment.
  for (int gap = a.size() / 2; gap > 0; gap /= 2) {
     for (int i = gap; i < a.size(); ++i) {
         // Insertion sort in subarrays.
         Comparable tmp = std::move(a[i]);
         int j = i;
         for (; j >= gap && tmp < a[j - gap]; j -= gap)
            a[j] = std::move(a[j - gap]);
         a[j] = std::move(tmp);
     }  // End of second for.
  }  // End of first for.
}
```

# Worst Case Analysis

Theorem: Worst case running time using Shell's increments is $\Theta(N^2)$

- **Proof:**

(A) Showing that there exists some input that actually takes $\Omega(N^2)$ to run.

- Provide an example input with behavior $\Omega(N^2)$:
- Consider N is power of 2
- Put the N/2 largest numbers at even positions
- N/2 smallest numbers at odd positions
- <See image of next slide>

(B) Upper bound on the worstcase running time

- Prove that the algorithm is $O(N^2)$

# Worst case analysis

| Positions | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Start | 1 | 9 | 2 | 10 | 3 | 11 | 4 | 12 | 5 | 13 | 6 | 14 | 7 | 15 | 8 | 16 |
| After 8-sort | 1 | 9 | 2 | 10 | 3 | 11 | 4 | 12 | 5 | 13 | 6 | 14 | 7 | 15 | 8 | 16 |
| After 4-sort | 1 | 9 | 2 | 10 | 3 | 11 | 4 | 12 | 5 | 13 | 6 | 14 | 7 | 15 | 8 | 16 |
| After 2-sort | 1 | 9 | 2 | 10 | 3 | 11 | 4 | 12 | 5 | 13 | 6 | 14 | 7 | 15 | 8 | 16 |
| After 1-sort | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

To merely place the n/2 smallest elements in the correct place requires at least $\Omega(N^2)$ ) work.

Input that provides worst behavior. How many inversions after 2 sort in example?

# Worst Case Analysis

- Worst case running time using Shell's increments is $\Theta(N^2)$
- **Proof of (B):** The algorithm is $O(N^2)$:
  - At pass $\mathbf{h_k}$: $h_k$ insertion sorts => (since ins. sort is $O(n^2)$)

    $h_k\,O((N/h_k)^2) = O(h_k\,(N/h_k)^2) = O(N^2 / h_k)$ <u>per pass</u>.
  - Total cost for all passes:

# Worst Case Analysis

- Worst case running time using Shell's increments is $\Theta(N^2)$
- **Proof of (B):** The algorithm is $O(N^2)$:
  - At pass $\mathbf{h_k}$: $h_k$ insertion sorts => (since ins. sort is $O(n^2)$)

    $h_k O((N/h_k)^2) = O(h_k (N/h_k)^2) = O(N^2 / h_k)$ <u>per pass</u>.
  - Total cost for all passes: $O(\sum_{i=1}^{t} N^2 / h_i) = O(N^2 \sum_{i=1}^{t} 1/ h_i)$

$O(N^2 / h_t) + O(N^2 / h_{t-1}) + \ldots + O(N^2 / h_k) + \ldots + O(N^2 / h_1) =$

$O(N^2 / h_t + N^2 / h_{t-1} + \ldots + N^2 / h_k + \ldots + N^2 / h_1) =$

$O(N^2 (1 / h_t + 1 / h_{t-1} + \ldots + 1 / h_k + \ldots + 1 / h_1) =$

$O(N^2)$. $(1 + ½ + ¼ + 1/8 + \ldots = ?$

# Worst Case Analysis of Shell Sort

- Worst case running time using Shell's increments is $\Theta(N^2)$
- **Proof of (B):** Example for N = 64 (power of 2)

  $h_6 = 32$, $h_5 = 16$, $h_4 = 8$, $h_3 = 4$, $h_2 = 2$, $h_1 = \mathbf{1}$.

Total cost for all passes:

$O(N^2 / 32) + O(N^2 / 16) + O(N^2 / 8) + O(N^2 / 4) + O(N^2 / 2) + O(N^2 / 1) =$

$O(N^2 ( 1 / 32 + 1 / 16 + 1 / 8 + 1 / 4 + 1 / 2 + 1) ) = ?$

In general

$O( N^2 ( 1 / h_t + 1 / h_{t-1} + \ldots + 1 / h_k + \ldots + 1 / h_1 ) = ?$

# Problem with Shell's increments

- Pairs of increments are not necessarily prime
- So the smaller increment can have little effect.

# Better increments

Improvements over original insertion sort

- 1, 3, 7, …, $2^k$-1 (Hibbard)
  - Consecutive increments have no common factors
  - **Theorem:** Worst case running time is $\Theta(N^{3/2})$
  - Proving average-case is still open.
- 1, 5, 19, 41, … (Sedgewick)
  - Worst case running time is $O(N^{4/3})$

# Next class

- HeapSort
- MergeSort
- QuickSort