07 B: Sorting II

CS1102S: Data Structures and Algorithms

Martin Henz

March 5, 2010

Generated on Friday 5th March, 2010, 08:31

1 Recap: Sorting

2 Heapsort

3 Mergesort

Sorting

Input

Unsorted array of elements

Behavior

Rearrange elements of array such that the smallest appears first, followed by the second smallest etc, finally followed by the largest element

Comparison-based Sorting

The only requirement

A comparison function for elements

The only operation

Comparisons are the only operations allowed on elements

Insertion Sort: Idea

Passes

Algorithm proceeds in $N - 1$ *passes*

Invariant

After pass i, the elements in positions 0 to i are sorted.

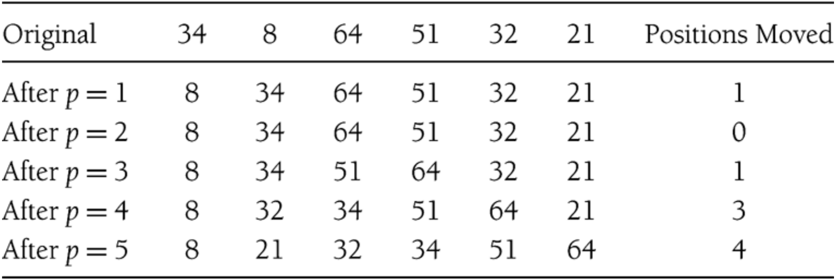
Consequence of Invariant

After $N - 1$ passes, the elements in positions 0 to $N - 1$ are sorted.
That is the whole array!

How to do a pass?

Pass *i*

Move element in position *i* to the left, until it is larger than the element to the left or until it is at the beginning of the array.



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

Worst Case

How many inversions in the worst case?

A list sorted in reverse has the maximal number of inversions

Maximal number of inversions

$$\sum_{i=0}^{N-1} i = N(N-1)/2$$

Average Case

How many inversions in the average case?

Consider the number of inversions in an list L and its reverse L_r.

Consider a pair of elements (x, y)

Either (x, y) is an inversion in L, or in L_r!

Overall

The sum of inversions of L and L_r *together* is $N(N-1)/2$.

Overall average

The overall average of inversions in a given list is $N(N-1)/4$

Runtime of Swapping Sorting Algorithms

Theorem

Any algorithm that sorts its elements by swapping neighboring elements runs in $\Omega(N^2)$.

Theorem

Any algorithm that removes one inversion in each step runs in $\Theta(N^2)$.

Shell Sort: Idea

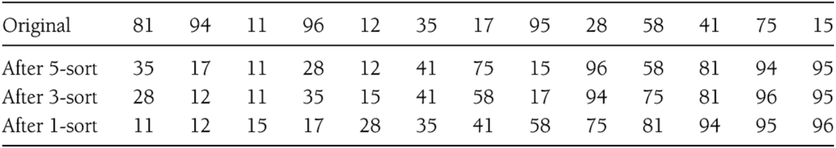
Main idea

Proceed in passes $h_1, h_2, \ldots, h_t$, making sure that after each pass, $a[i] \leq a[i + h_k]$.

Invariant

After pass h_k, elements are still h_{k+1} sorted

Shell Sort: Example using $\{1, 3, 5\}$



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

Analysis

Shell's Increments

The worst-case running time of Shellsort, using Shell's increments $1, 2, 4, \ldots,$, is $\Theta(N^2)$.

Hibbards's Increments

The worst-case running time of Shellsort, using Hibbard's increments $1, 3, 7, \ldots, 2^k - 1$, is $\Theta(N^{3/2})$.

1 Recap: Sorting

2 Heapsort

3 Mergesort

Idea

Use heap to sort

- Build heap from unsorted array (using percolateDown)
- Repeatedly take minimal element (using deleteMin) and place it in sorted array

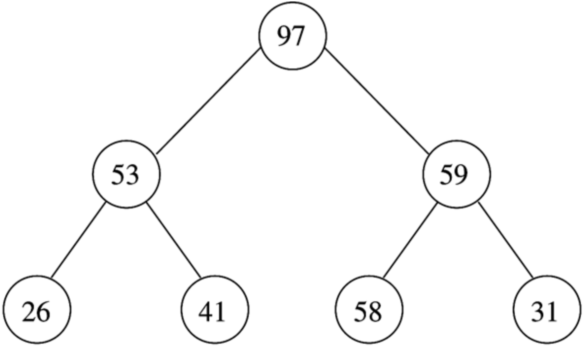
Drawback

Will require extra array

How to avoid this?

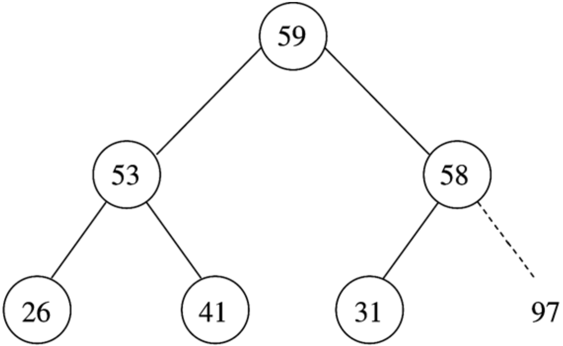
Use free memory at the end of the heap!

Heapsort



	97	53	59	26	41	58	31			
0	1	2	3	4	5	6	7	8	9	10

Heapsort



	59	53	58	26	41	31	97			
0	1	2	3	4	5	6	7	8	9	10

Heapsort

```
private static int leftChild(int i) {
   return 2 * i + 1;
}
```

Heapsort

```
private static <AnyType extends
                    Comparable<? super AnyType>>
void percDown(AnyType [ ] a, int i, int n) {
   int child; AnyType tmp;
   for(tmp = a[ i ]; leftChild(i) < n; i=child) {
      child = leftChild( i );
      if(child != n - 1 &&
         a[child].compareTo(a[child + 1]) < 0)
        child++;
      if(tmp.compareTo(a[child]) < 0)
        a[ i ] = a[ child ];
      else break; }
   a[ i ] = tmp; }
```

Heapsort

```
public static <AnyType extends
                    Comparable<? super AnyType>>
void heapsort(AnyType [ ] a) {
    for(int i = a.length / 2; i >= 0; i--)
        percDown( a, i, a.length );
    for(int i = a.length - 1; i > 0; i--) {
        swapReferences( a, 0, i );
        percDown( a, 0, i );
    }
}
```

1 Recap: Sorting

2 Heapsort

3 Mergesort

- Idea
- Example
- Implementation

Idea: Use recursion!

- Split unsorted arrays into two halves
- Sort the two halves
- Merge the two sorted halves

Merging Two Sorted Arrays

- Use two pointers, one for each sorted array
- Compare values at pointer positions
  - Copy the smaller values into sorted array
  - Advance the pointer that pointed at smaller value

Example

Sort the array

```
26 13 1 14 15 38 2 27
```

Implementation of Mergesort

```
public static <AnyType extends
                    Comparable<? super AnyType>>
void mergeSort( AnyType [ ] a) {
  AnyType [] tmpArray =
    (AnyType[]) new Comparable[ a.length ];
  mergeSort(a, tmpArray, 0, a.length - 1 );
}
```

Implementation of Mergesort

```
private static <AnyType extends
               Comparable<? super AnyType>>
void mergeSort(AnyType [] a,AnyType [] tmpArray,
               int left, int right) {
  if( left < right ) {
    int center = ( left + right ) / 2;
    mergeSort( a, tmpArray, left, center );
    mergeSort( a, tmpArray, center + 1, right );
    merge( a, tmpArray, left, center + 1, right );
  }
}
```

Implementation of Merge Operation

```
private static <AnyType extends
                    Comparable<? super AnyType>>
void merge(AnyType [] a, AnyType [] tmpArray,
                int leftPos, int rightPos, int rightEnd) {
  int leftEnd = rightPos - 1;
  int tmpPos = leftPos;
  int numElements = rightEnd - leftPos + 1;
  while(leftPos <= leftEnd && rightPos <= rightEnd)
    if(a[leftPos].compareTo(a[rightPos]) <= 0 )
      tmpArray[ tmpPos++] = a[leftPos++];
    else
      tmpArray[ tmpPos++] = a[rightPos++];
  ...
```

Implementation of Merge Operation

```
private static <AnyType extends
                    Comparable<? super AnyType>>
void merge(AnyType [] a, AnyType [] tmpArray,
           int leftPos, int rightPos, int rightEnd) {
  ...
  while( leftPos <= leftEnd )
    tmpArray[tmpPos++] = a[leftPos++];
  while( rightPos <= rightEnd )
    tmpArray[tmpPos++] = a[rightPos++];
  for(int i = 0; i < numElements; i++, rightEnd--)
    a[rightEnd] = tmpArray[rightEnd];
}
```

Next Week

- Monday: Sit-in lab
- Wednesday lecture: Sorting III
- Friday: Midterm 2: Trees, Hashing, Priority Queues, Sorting I + II