

More Powerful Sorting Methods

Simplify our discussion

- Limit topics to *distinguished* integer sorting
- to sorting in *distinguished* order
- to *internal* sorting, i.e. the entire sorting is performed in the main memory;

Motivation

- Sorting is used *often* in practice.
- Sorting problems are good examples to show how to take many different points of view towards the same problem.
- Sorting is one of few classes of problems for which we can easily derive good lower bounds for the worst and average cases.
- Sorting problems have been well studied

Insertion sort

Example 22 Sort a list: 34 8 64 51 32 21

Sorted part	Unsorted part	Current integer	Integer(s) moved
-----	-----	-----	-----
	34 8 64 51 32 21		
34	8 64 51 32 21	34	-
8 34	64 51 32 21	8	34
8 34 64	51 32 21	64	-
8 34 51 64	32 21	51	64
8 32 34 51 64	21	32	34 51 64
8 21 32 34 51 64		21	32 34 51 64

Algorithm

```

for p=2 to N do
  begin
    x=A[p];
    find location j,
      where for j=p-1, p-2, ..., 1, such that x<A[j];
    move A[j]..A[p-1] one location to the right (cell);
    place x in A[j];
  end
end {for}

```

Selection sort

For each element A[i],

1. find the MinKey (the minimum key) and its location m;
2. swap MinKey and A[i].

Selection sort

Example 23 Sort the integers (34, 8, 64, 51, 32, 21) into ascending order.

Sorted part	Unsorted part	MinKey so far	Integers to swap
-----	-----	-----	-----
	34 8 64 51 32 21	8	8 34
8	34 64 51 32 21	21	21 34
8 21	64 51 32 34	32	32 64
8 21 32	34 64 51	34	-
8 21 32 34	64 51	51	51 64
8 21 32 34 51	64	64	-
8 21 32 34 51 64			

Algorithm (1). find the MinKey

MinKey=first element

for each element A[j] where j>=2

if A[j]<MinKey then

begin

MinKey=A[j];

m=j

end;

2. swap MinKey and A[i]

A[m]=A[i]

A[i]=MinKey

Segment: 5, 3, 1.

h_k	Original list	Result in each pass
	81 94 11 96 12 35 17 95 28 58 41 75 15	
1 : 5	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	
2 : 3	35 .. 28 .. 75 .. 58 .. 95	28 .. 35 .. 58 ..
	17 .. 12 .. 15 .. 81	12 .. 15 .. 17 ..
	11 .. 41 .. 96 .. 94	11 .. 41 .. 94 ..
	28 12 11 35 15 41 58 17 94 75 81 96 95	
3 : 1	11 12 15 17 28 35 41 58 75 81 94 95 96	

The Shell sort algorithm

Instead of comparing only adjacent keys, Shellsort uses the *increment sequence*, $h_1, h_2, \dots, h_t$, where $h_t = 1$ and check the items in h_k distance at pass k , $k = 1 \dots t$.

many choices for the increment sequence $h_1, h_2, \dots, h_t$ as integers.

A simple choice:

$$h_k = h_{k-1} \text{ div } 3 + 1$$

Example 24 Sort the integers (81, 94, 11, 96, 12, 35, 17, 95, 28, 58, 41, 75, 15) into increasing order.

Algorithm (Shell Sort)

```

1. choose the initial increment;
2. repeat
    update increment;
    {something simple,
      e.g. increment=increment div 3+1}
    for start=1 to increment do
        InsertSort(start, increment, L);
        {where L is the list of items to sort}
    until increment=1.
```

0.1 Algorithm analysis

Difficult! Because it depends on the choice of increment sequence. Average case is still an *open problem*.

The Heap sort algorithm

Algorithm (Heap sort)

```
const
    max=100;
type
    index=1..max;
    item=integer;
    heapArray=array[1..max] of item;

procedure HeapSort(var A:heapArray);
var
    tmp:item;
    k:index;
```

Algorithm (Basic Heap operations)

```
procedure AddRoot(i,n:index);
var
    j:index;
    tmp:item;
begin
    j=2*i;
    if j<=n then
        begin
            if (j<n) and (A[j]<A[j+1]) then j=j+1;
            if A[i]<A[j] then
                begin
                    tmp=A[i]; A[i]=A[j];
                    A[j]=tmp; AddRoot(j,n)
                end
            end
        end
    end;

procedure BuildHeap;
begin
    for k=(Length(A) div 2) downto 1 do
        AddRoot(k,Length(A));
    end;
```

```
begin
    BuildHeap; {Construct a heap}
    for k=Length(A) downto 2 do
        begin
            tmp=A[k];
            A[k]=A[1];
            A[1]=tmp; {swap A[1] with A[k]}
            AddRoot(1,k-1);
            {Restore the heap properties
             for the shortened array}
        end
    end;
end;
```

Example 25 Trace the execution of the algorithm `HeapSort(L)` on a list of integers $L = (1, 6, 5, 14, 8, 10, 12, 15)$. Suppose the list is stored in array A .

1. Call `BuildHeap`: Let ix be index.

	ix	1	2	3	4	5	6	7	8	k	i	j
					k							
A[ix]	1	6	5	14	8	10	12	15				
				i		j				4	4	8
A[ix]	1	6	5	15	8	10	12	14		8	16	>n=8
				k								
A[ix]	1	6	5	15	8	10	12	14				
				i		j				3	3	6
						j						7
A[ix]	1	6	12	15	8	10	5	14		7	14	>n
				k								
A[ix]	1	6	12	15	8	10	5	14				
				i		j				2	2	4

```

A[ix]  1  15 12 6  8  10 5  14
        i      j      4  8
A[ix]  1  15 12 14 8  10 5  6      8  16 >n

        k
A[ix]  1  15 12 14 8  10 5  6
        i  j      1  1  2
A[ix]  15 1  12 14 8  10 5  6
        i      j      2  4
A[ix]  15 14 12 1  8  10 5  6
        i      j      4  8
A[ix]  15 14 12 6  8  10 5  1      8  16 >n

```

2. Execute the remain algorithm:

```

ix      1  2  3  4  5  6  7  8      k  i  j
        k
A[ix]  15 14 12 6  8  10 5  1      8
A[ix]  1  14 12 6  8  10 5  15
        i  j      1  2
A[ix]  14 1  12 6  8  10 5  15
        i      j      2  4
        j      2  5
A[ix]  14 8  12 6  1  10 5  15      5  10 >k-1

```

```

        k
A[ix]  8  6  5  1  10 12 14 15      4
A[ix]  1  6  5  8  10 12 14 15
        i  j      1  2
A[ix]  6  1  5  8  10 12 14 15      2  4 >k-1

        k
A[ix]  6  1  5  8  10 12 14 15      3
A[ix]  5  1  6  8  10 12 14 15
        i  j      1  2
A[ix]  5  1  6  8  10 12 14 15      2  4 >k-1

        k
A[ix]  1  5  6  8  10 12 14 15      2  1  2 >k-1

```

```

        k
A[ix]  14 8  12 6  1  10 5  15      7
A[ix]  5  8  12 6  1  10 14 15
        i  j      1  2
        j      3
A[ix]  12 8  5  6  1  10 14 15
        i      j      3  6
A[ix]  12 8  10 6  1  5  14 15      6  12 >k-1

        k
A[ix]  12 8  10 6  1  5  14 15      6
A[ix]  5  8  10 6  1  12 14 15
        i  j      1  2
        j      3
A[ix]  10 8  5  6  1  12 14 15      3  6 >k-1

        k
A[ix]  10 8  5  6  1  12 14 15      5
A[ix]  1  8  5  6  10 12 14 15
        i  j      1  2
A[ix]  8  1  5  6  10 12 14 15
        i      j      2  4
A[ix]  8  6  5  1  10 12 14 15      4  8 >k-1

```

The Merge sort algorithm

Algorithm (Merge sort)

```

MergeSort(List)
begin
  if the List has length>1 then
    begin
      partition the List into
        two lists Llist and Rlist;
      sort(Llist);      (recursively)
      sort(Rlist);      (recursively)
      combine(Llist, Rlist)
    end
  end;
end;

```

Example 26 *Sort the list 26 33 35 29 19 12 22.*

Llist	Rlist	Result on each recursive level
26 33 35 29	19 12 22	
26 33	35 29	
26	33	26 33
26 33		
35	29	29 35
	29 35	
26 33	29 35	26 29 33 35
26 29 33 35		
19 12	22	
19	12	12 19
12 19	22	12 19 22
	12 19 22	
26 29 33 35	12 19 22	12 19 22 26 29 33 35

```

procedure Split(first,last:index,
                var splitPoint:index);

var x:key;
    unknown:index;
begin
    x=A[first];
    splitPoint=first;
    for unknown=first+1 to last do
        begin
            if A[unknown]<x then
                begin
                    splitPoint=splitPoint+1;
                    Exchange(A[splitPoint], A[unknown])
                end
            end;
        end;
    Exchange(A[first], A[splitPoint])
end;

```

The Quick sort algorithm

Algorithm (Quick sort)

```

procedure Quicksort(var first,last:index);
var
    pivotAddr:index;
begin
    if first<last then
        begin
            Split(first, last, pivotAddr);
            QuickSort(first, pivotAddr-1);
            QuickSort(pivotAddr+1, last)
        end
    end;
end;

```

```

procedure Exchange(var a,b: key);
var
    tmp:key;

begin
    tmp=a;
    a=b;
    b=tmp
end;

```

Example 27 Sort the list 33 26 35 29 19 12 22.

Pivots	Llist	Rlist	Result on each recursive level
26	19 12 22	33 35 29	
19	12	22	12 19 22
	12 19 22		
33	29	35	29 33 35
		29 33 35	
	12 19 22	29 33 35	12 19 22 26 29 33 35

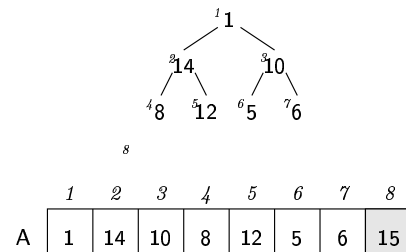
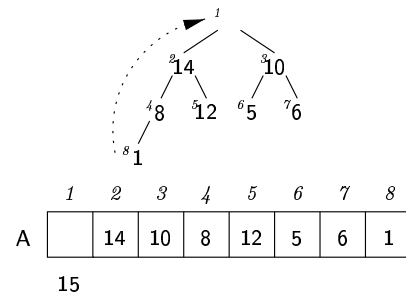


Figure 22: AddRoot-1

Heap Sort

Heap sort is only one application of *heaps*.

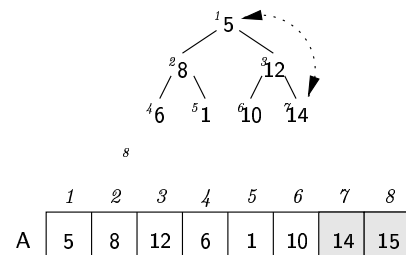
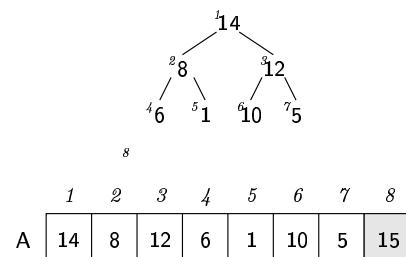


Figure 23: AddRoot-1

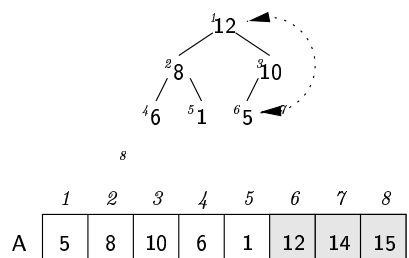
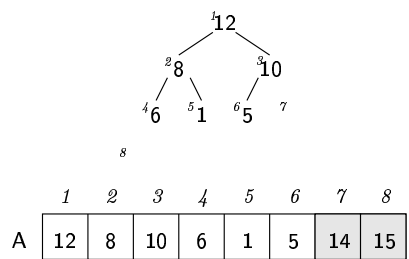


Figure 24: AddRoot-1

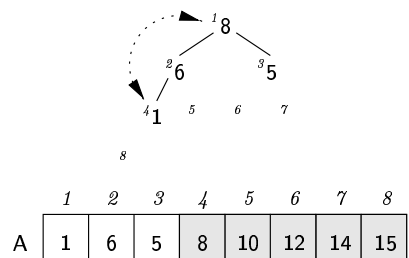
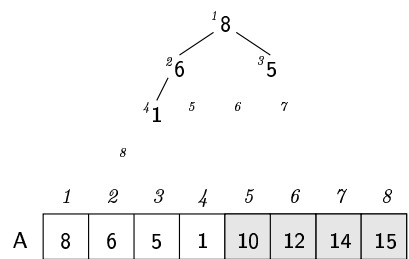


Figure 26: AddRoot-1

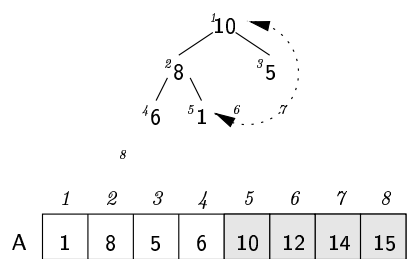
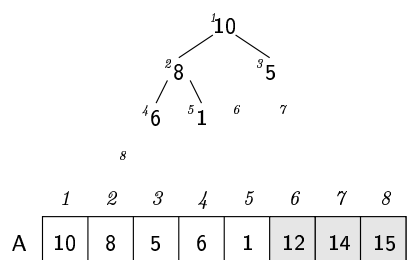


Figure 25: AddRoot-1

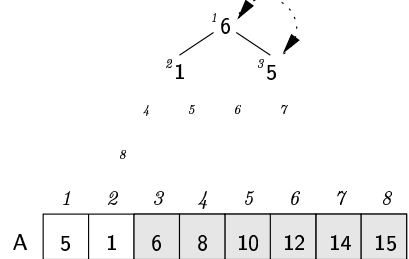
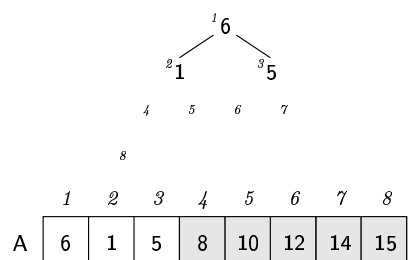


Figure 27: AddRoot-1

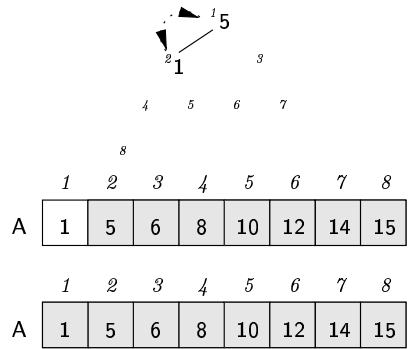
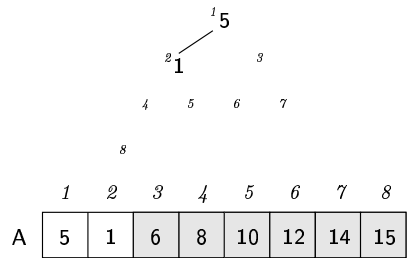


Figure 28: `AddRoot-1`

Complexity Analysis

Two optimizations in this implementation:

1. Build the heap from a list consisting of n elements in arbitrary order.
2. No extra space required.