Read chapter 7 of Weiss.

Sorting

Now that we've established a moderate comfort level with recursion and algorithm analysis, we'll look at some straightforward applications to sorting.

Let's start by supposing that we must sort (into ascending order) an array A containing integers ranging between 0 and some positive number M. We can develop a linear time algorithm, actually O(M + N) where N is the size of the array, as follows...

```
static void bucket (int [] A, int M)
  {
    int [] count = new int [M + 1];
    for (int i = 0; i < count.length; i++ )
        count[i] = 0;

    for (int j = 0; j < A.length ; j++)
        count [A[j]] ++;

    for (int i=0, int j=0; i<count.length && j<A.length ; i++)
        {
          if (count[i] != 0)
            { int k = 0;
              for ( ; k < count[i] ; k++)
                  A[j+k] = i;
              j+= k;
}}}
```

'buckets' to show how often a particular number occurs.

fill buckets

refill A from buckets

This bucket sort technique only works because we know a priori the size of the largest integer in the given array. (There's a short discussion of this in section 7.9 of Weiss.) From now on we'll focus on sorting arrays where we have no such a priori information.

We will suppose the existence of a swap method ...

```
static void swap ( Comparable [ ] A , int i , int j )
   {
     Comparable temp = A[i];
     A[i] = A[j];
     A[j] = temp;
   }
```

and consider first the standard easy sorting methods. Firstly, selection sort scans the array A from left to right, at each point looking along the rest of the array to find the (first) least term, and then swapping A[i] with this least term ...

```
static void selection ( Comparable [ ] A )
   {
   for ( int i = 0 ; i < A.length - 1 ; i++ )
     { int min = i ;
       for ( int j = i+1 ; j < A.length ; j++ )
          if ( A[j] < A[min] ) min = j ;
       swap ( A , i , min );
     }
```

nested

Fairly evidently, this is an $O(N^2)$ algorithm.

Another simple sorting method often seen is bubble sort, which runs through the array from left to right, but at each step it zips along the rest of the array from right to left swapping adjacent terms if they are the wrong way round ...

```
static void bubble (Comparable [] A)
  {
    for (int i = 0 ; i < A.length - 1 ; i++)
      for (int j = A.length-1 ; j > i ; j--)
        if (A[j-1] > A[j])
          swap (A , j-1, j);
  }
```

nested

Again, clearly an $O(N^2)$ algorithm. Finally, amongst simple sorting methods, insertion sort moves from left+1 to right, at each stage sliding the terms to the left of the current position one step to the right to create 'space' for the current term to go into the correct spot ...

```
static void insertion (Comparable [] A)
  {
    int j;
    for (int p = 1 ; p < A.length ; p++)
      { Comparable temp = A[p];
        for (j=p; j>0 && temp.lessThan(A[j-1]) ; j--)
          A[j] = A[j-1];
        A[j] = temp;
  } }
```

nested

Again, this is an $O(N^2)$ algorithm.

To clarify these algorithms, let's see their effects on an explicit example...

Selection sort

At each step, look to the right for the smallest to swap with.

	34	8	64	51	32	21
	8	34	64	51	32	21
	8	21	64	51	32	34
	8	21	32	51	64	34
	8	21	32	34	64	51
	8	21	32	34	51	64

Bubble sort

At each step, bubble the smaller terms from the RHS.

No further changes happen in either loop →

	34	8	64	51	32	21
			21	64	51	32
	8	34	21	64	51	32
	8	21	34	32	64	51
	8	21	32	34	51	64
	8	21	32	34	51	64

Insertion sort

At each step, slide terms rightwards by one slot if they're bigger than current term

	34	8	64	51	32	21
	8	34	64	51	32	21
	8	34	64	51	32	21
	8	34	51	64	32	21
	8	32	34	51	64	21
	8	21	32	34	51	64

It's worth noting that empirically, insertion and selection sort are about twice as fast as bubble sort for small arrays. Furthermore, if the act of making the comparisons is slow, then insertion sort wins over the other two methods; and if the act of swapping is slow, then selection sort is best.

array size =	64	256	1024
Bubble sort	2.76	46.42	766.22
Selection sort	1.40	22.18	354.48
Insertion sort	1.12	17.58	280.27

times are in 1/60 secs

data from T. A. Standish "Data Structures in Java", 1998, Addison-Wesley

In fact, it's not hard to show that the best that can be achieved by a sorting algorithm which relies on swapping adjacent terms (explicitly or implicitly) is $O(N^2)$ — see Weiss, section 7.3.

One algorithm which was introduced to beat the adjacent swap limit, originally due to Donald Shell, is **shellsort**. It starts by comparing distant terms, and progresses towards adjacent comparisons by the end. Depending on how the progression is directed, the worst case scenario varies from $O(N^2)$ to $O(N^{3/2})$ or better.

For h some positive integer, we say that an array is h-sorted if it comprises h interleaved sorted arrays. So if we start with a large value of h and progress eventually to $h=1$, we end up with the array being sorted, and get the advantage of being able to move distant terms by large jumps, thus avoiding the $O(N^2)$ bound associated with adjacent swaps.

The real trick is choosing the right sequence of h's. For example

$h = 1, 2, 4, 8, 16, 32, 64, 128, \ldots$

could be coded as ...

```
static void shell (Comparable [] A)
{
    int h;
    for (h=1 ; h < A.length/2 ; h = 2*h);   *
    for ( ; h>0 ; h/=2)                     *
        for (int i = h ; i < A.length ; i++)
        {
            Comparable temp = A[i];
            int j = i;
            for ( ; j>=h && temp.lessThan(A[j-h]); j-=h)
                A[j] = A[j-h];
            A[j] = temp;
        }
}
```

build enough of this h sequence

sparsely nested

insertion sort-like slide skipping h terms each time

insertion sort-like insert

To illustrate, let's consider an example ...

81 94 11 96 12 35 17 95 28 58 41 75 15

A.length = 13, h = 1, 2, 4, 8

i = 8, temp = 28

28 94 11 96 12 35 17 95 81 58 41 75 15

i = 9, temp = 58

28 58 11 96 12 35 17 95 81 94 41 75 15

i = 10, temp = 41 no change

i = 11, temp = 75

28 58 11 75 12 35 17 95 81 94 41 96 15

i = 12, temp = 15 no change

h = 4, i = 4, temp = 12

12 58 11 75 28 35 17 95 81 94 41 96 15

i = 5, temp = 35

12 35 11 75 28 58 17 95 81 94 41 96 15

i = 6, i = 7, i = 8, i = 9, i = 10, i = 11 all no change

i = 12, temp = 15

12 35 11 75 15 58 17 95 28 94 41 96 81

h = 2, changes only for i = 2, 5, 9 as shown below ...

11 35 12 75 15 58 17 95 28 94 41 96 81

11 35 12 58 15 75 17 95 28 94 41 96 81

11 35 12 58 15 75 17 94 28 95 41 96 81

h = 1, changes for i = 2, 4, 6, 8, 10, 12 as shown below ...

11 12 35 58 15 75 17 94 28 95 41 96 81

11 12 15 35 58 75 17 94 28 95 41 96 81

11 12 15 17 35 58 75 94 28 95 41 96 81

11 12 15 17 28 35 58 75 94 95 41 96 81

11 12 15 17 28 35 41 58 75 94 95 96 81

11 12 15 17 28 35 41 58 75 81 94 95 96

Although choosing powers of 2 gave the original 'hopping' sequence for h, this is not the best sequence — its worst case running time is still $O(N^2)$ — see section 7.4.1 of Weiss. This can be improved to $O(N^{3/2})$ by replacing the code marked ⊛ with ...

```
for(h=1; h<=A.length/9; h=3*h+1);
for( ; h>0; h/=3)
```

which gives Donald Knuth's hopping sequence ...

$h = 1, 4, 13, 40, 121, 364, 1093, \ldots$

Actually $O(N^{3/2})$ is the worst case for any hopping sequence which is relatively prime and grows exponentially.

A small industry has grown around improved hopping sequences. For example (Sedgewick) ...

$h = 1, 8, 23, 77, 281, 1073, \ldots, 4^n + 3.2^n + 1, \ldots$

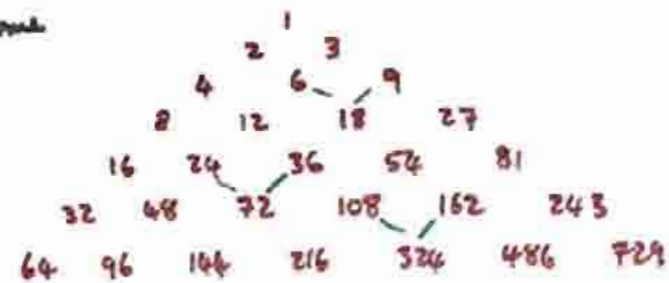
gives a worst case of $O(N^{4/3})$, and we can even get $O(N(\log N)^2)$ by using (Pratt) ...

$h = 1, 2, 3, 4, 6, 9, 8, 12, 18, 27, 16, 24, \ldots$

Where the numbers come from the triangle

Where

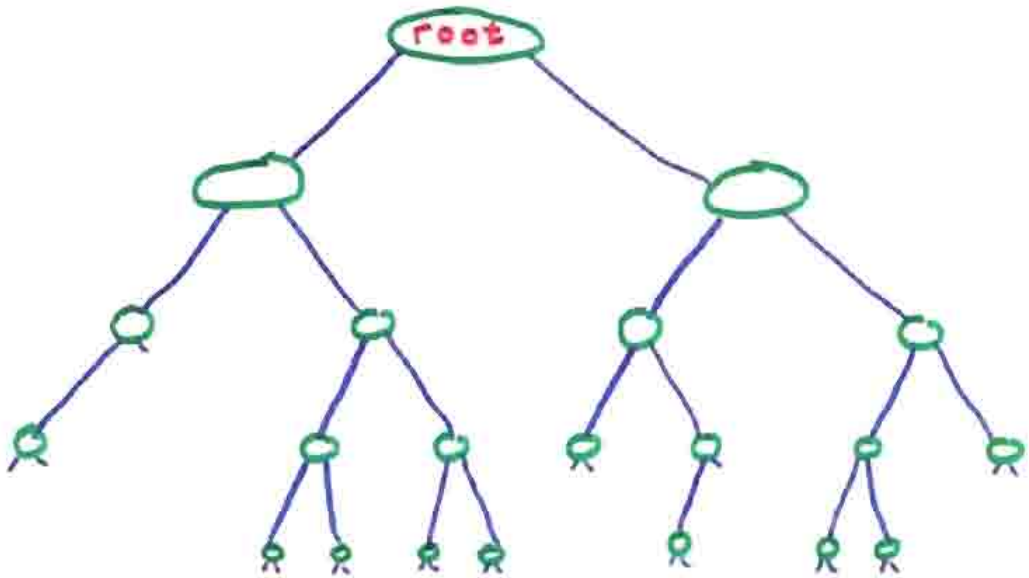
$\gamma = 3\alpha = 2\beta$



etc.

Although this can be useful, the analyses can become quite ugly! So we'll change direction here, and look at some fun, graphically inspired, sorting algorithms which use recursion.

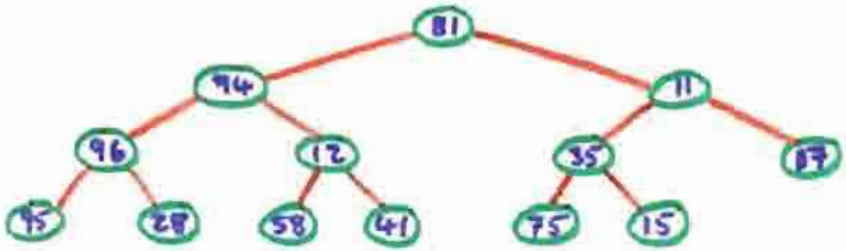
Firstly, some terminology...



A **tree** is a collection of **nodes** and **edges** such that there is exactly one path of edges leading from any node to any other node. Let's assume that things flow from the top to the bottom. The **root node** is chosen to be at the top. Nodes which are at the ends of 'branches' are called **leaves**. If each node has at most two nodes coming 'down' from it, then the tree is a **binary tree**. (We'll be more precise later on!)

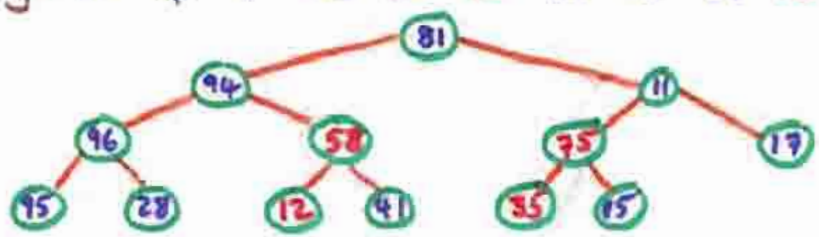
We'll start off with a tree algorithm implemented non-recursively. The worst case scenario for heapsort is O(N log N), see Weiss, section 7.5. The algorithm itself is rather fun. We start by pouring the array into a binary tree ...

81 94 11 96 12 35 17 95 28 58 41 75 15



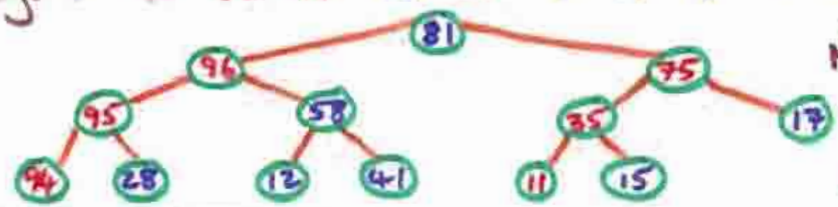
Then we build a heap within this tree by starting at the penultimate layer (working R to L along this layer) and promoting whichever is the larger of the left or right 'child' if this is bigger than the 'parent' ...

array= 81 94 11 96 58 75 17 95 28 12 41 35 15



Continuing layer-by-layer ...

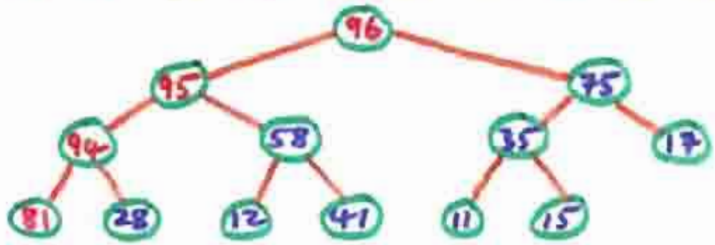
array= 81 96 75 95 58 35 17 94 28 12 41 11 15



Notice how this also reaches down from its layer

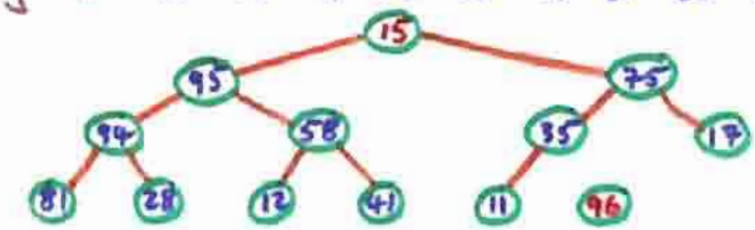
Finally...

array = 96 95 75 94 58 35 17 81 28 12 41 11 15



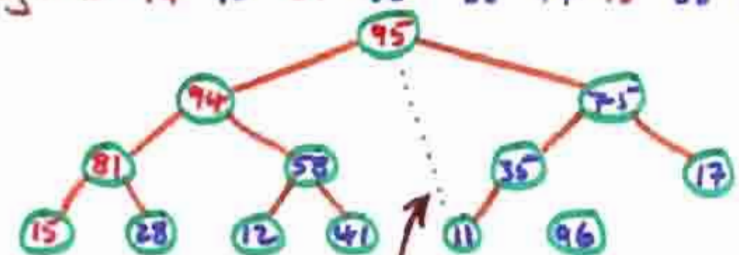
Now that we have a heap, we 'delete' the root value, which is the max value (by swapping it with the far RHS of the array), and successively promote the largest of the left or right 'children'. This process continues until the tree is empty (we slide the effective RH end of the array leftwards by one slot each time)...

array = 15 95 75 94 58 35 17 81 28 12 41 11 96



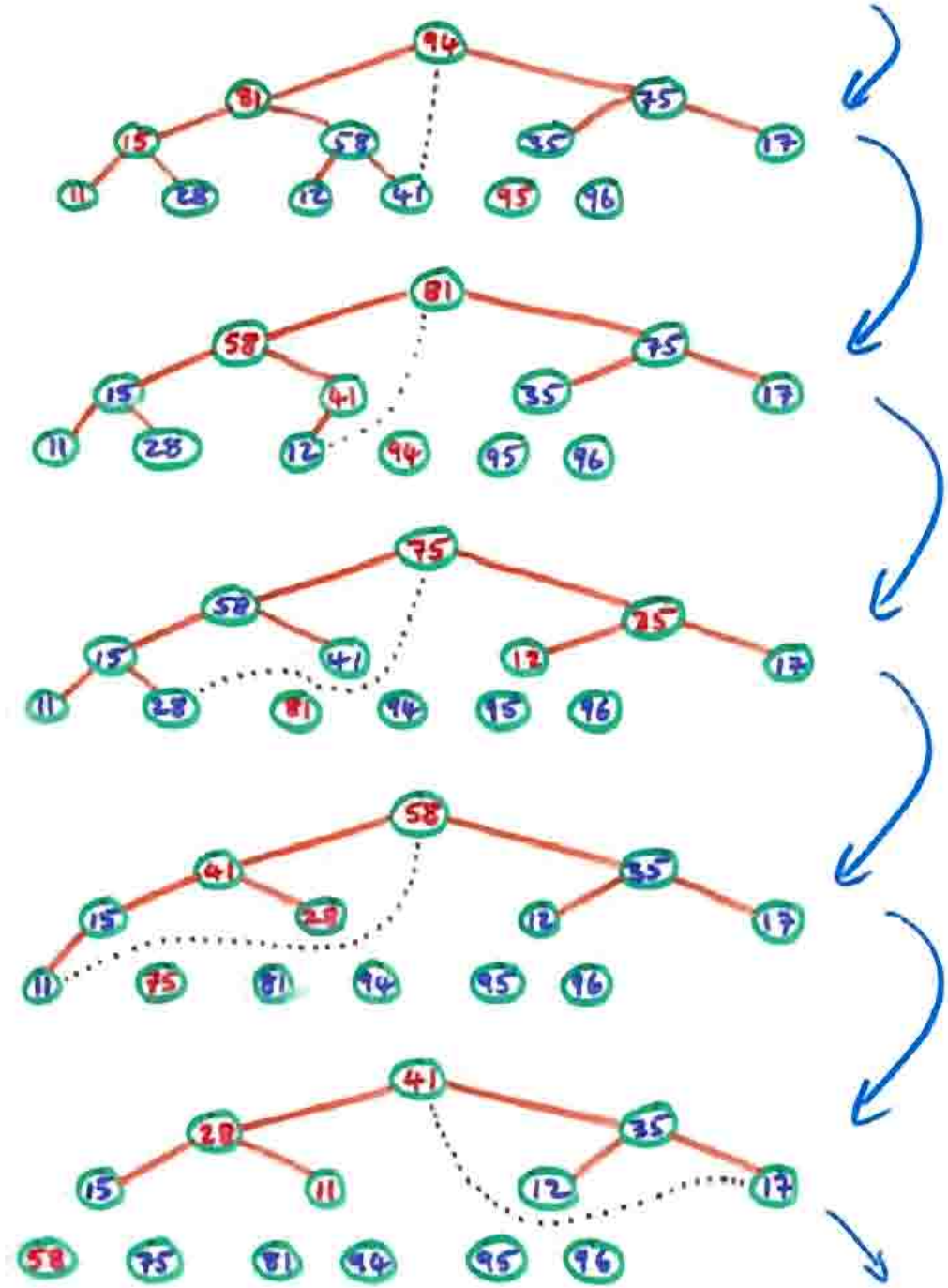
The delete max step via swapping the extremities.

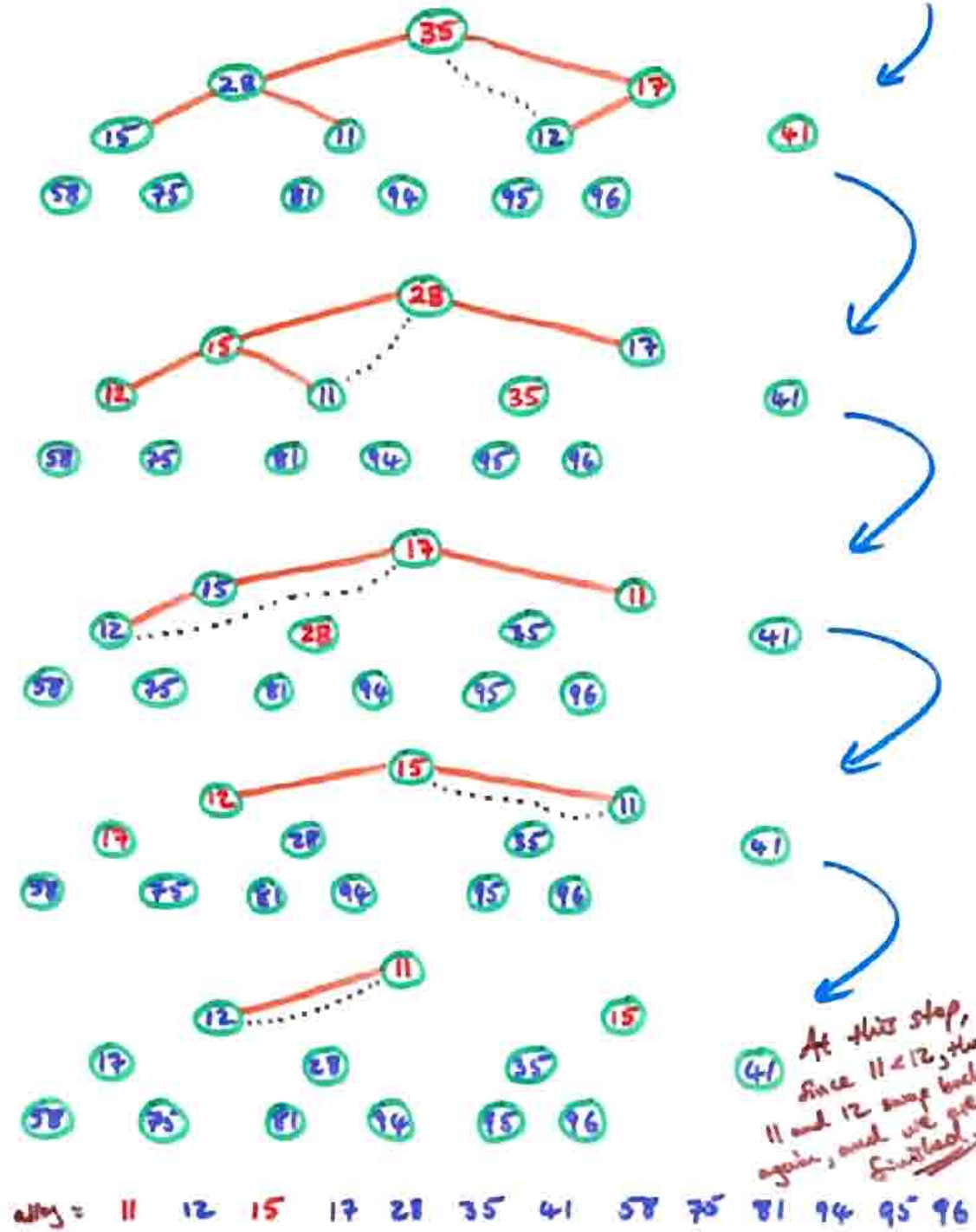
array = 95 94 75 81 58 35 17 15 28 12 41 11 96



the next delete-swap....

The percolate down step to restore the 'heap' nature to this smaller tree.


94
81
75
15
58
35
17
11
28
12
41
95
96
81
58
75
15
41
35
17
11
28
12
94
95
96
75
58
35
15
41
12
17
11
28
81
94
95
96
58
41
35
15
28
12
17
11
75
81
94
95
96
41
28
35
15
11
12
17
58
75
81
94
95
96


35 28 17 15 11 12 41 58 75 81 94 95 96
28 15 17 12 11 35 41 58 75 81 94 95 96
17 15 11 12 28 35 41 58 75 81 94 95 96
15 12 11 17 28 35 41 58 75 81 94 95 96
11 12 15 17 28 35 41 58 75 81 94 95 96
At this step, since 11 < 12, the 11 and 12 swap back again, and we get finished...
array = 11 12 15 17 28 35 41 58 75 81 94 95 96

The code for all this is thankfully quite simple ...

```
public static void heapsort ( Comparable [] A)
  {
    for (int i = A.length/2 ; i>=0 ; i--)
      percDown (A , i , A.length);

    for (int i = A.length-1 ; i>0 ; i--)
      { swap (A , 0 , i);
        percDown (A, 0, i);
      }
  }

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

'buildheap', see later code

'deletemax', see later code

messy method to hop along the array to hit the left child of A[i]

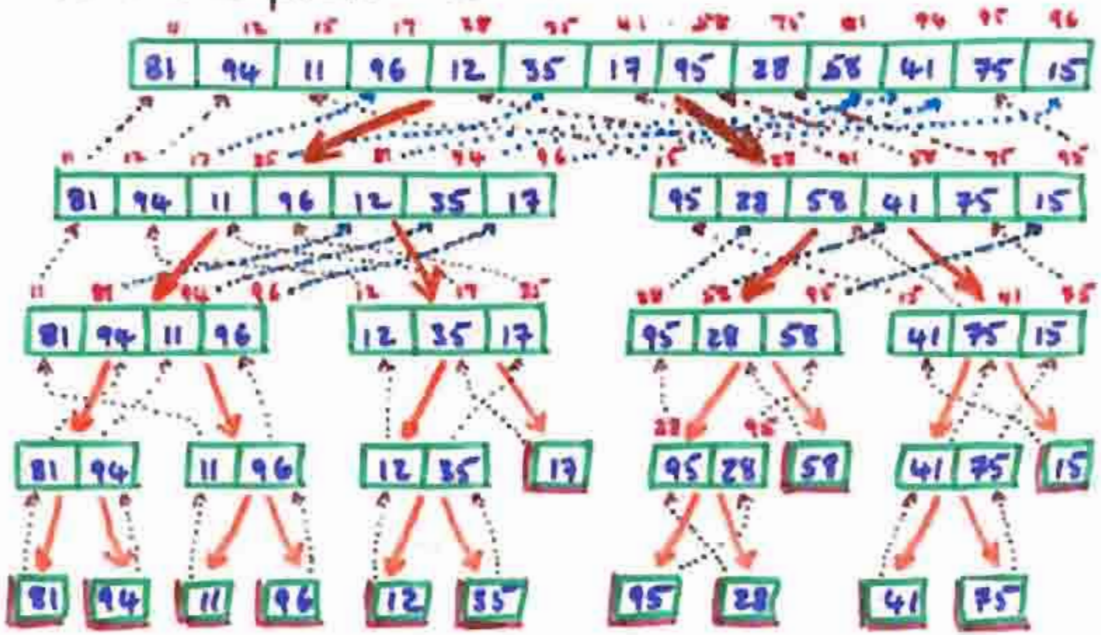
(All this is a bit prettier recursively!?)

```
private static void percDown (Comparable [] A, int i, int n)
  {
    int child;
    Comparable temp;
    for (temp = A[i] ; leftChild(i) < n ; i = child)
      { child = leftChild(i);
        if (child != n-1 && A[child].lessThan(A[child+1]))
          child++;
        if (temp.lessThan(A[child]))
          A[i] = A[child];
        else break;
      }
    A[i] = temp;
  }
```

which child is bigger?

promote

The first of the two recursive tree-based sorting algorithms we're going to look at is **mergesort**. It gets its name from the fact that it involves merging pairs of sorted data, and it does this by using recursion to find the 'bottom', and then working its way back 'up' to the 'top'. An example will make the process clear ...



Mergesort starts by recursively splitting up the array into successive 'halves' until reaching the 'bottom' level comprising arrays holding only a single element. It's easy to sort a single-element array!!

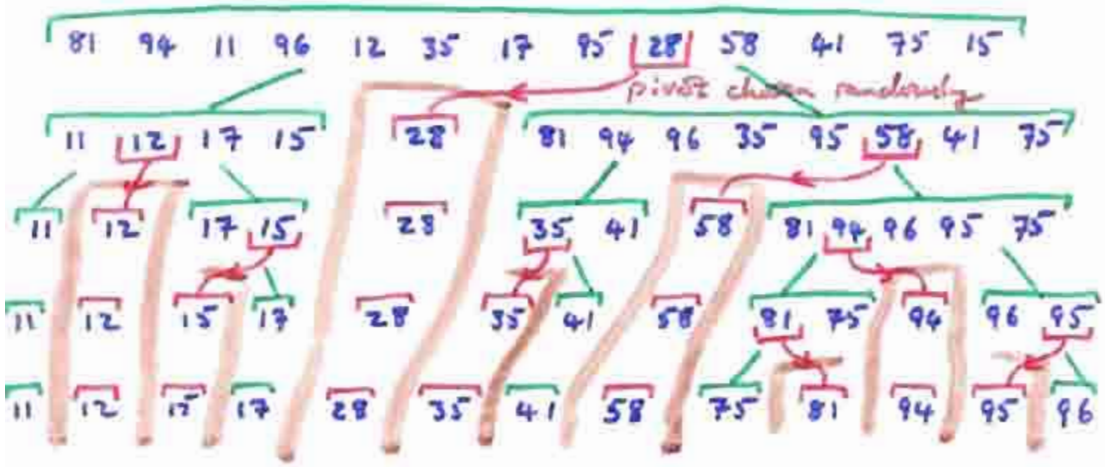
Now **merge** these two one-element arrays to form a correctly ordered two-element array.

Now, as we proceed back 'up' the tree, we repeat the process of merging pairs of sorted sub-arrays into larger arrays (as illustrated by the dotted lines). This repeated halving of array size gives an $O(\log N)$ component, and then the flowing together (merging) is an $O(N)$ process, so the worst case scenario for mergesort is $O(N \log N)$ — see Weiss, section 7.6.

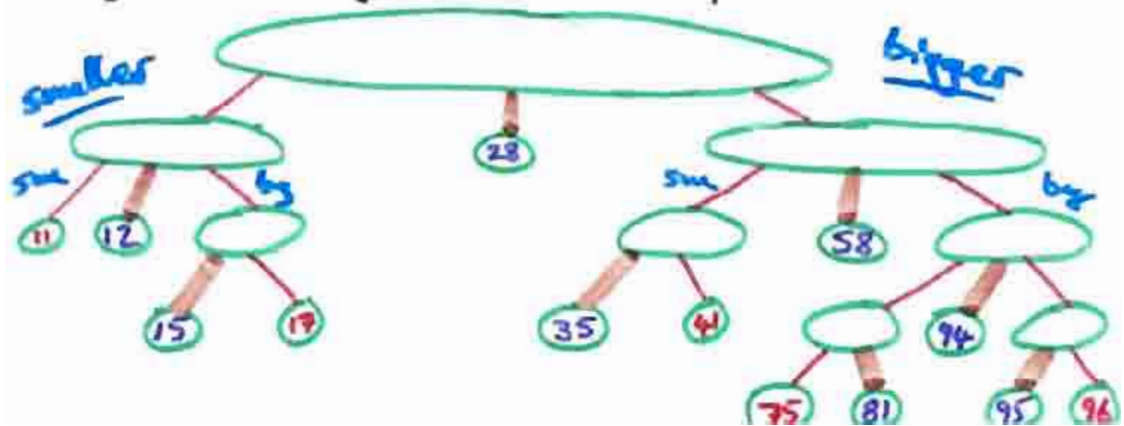
The code for mergesort is now straightforward ...

```
public static void mergeSort ( Comparable [] A )
  { Comparable [] temp = new Comparable [A.length];
    mergeSort ( A , temp , 0 , A.length -1 );
  }
private static void mergeSort (Comparable [] A, Comparable [] T, int L, int R)
  { if (L < R)
      { int centre = (L + R) / 2;
        mergeSort ( A , T , L , centre );
        mergeSort ( A , T , centre +1 , R );
        merge ( A , T , L , centre +1 , R );
  } }
private static void merge ( Comparable [] A, Comparable [] T, int L, int P, int R)
  { int LEnd = P-1 , pos = L , size = R - P + 1 ;
    while ( L <= LEnd && P <= R )
      if ( A[L]. lessEq (A[P-1])) temp[pos++] = A[L++];
      else                        temp[pos++] = A[P++];
    while ( L <= LEnd )  temp[pos++] = A[L++];
    while ( P <= R )     temp[pos++] = A[P++];
    for ( int i=0 ; i < size ; i++, R--)
      A[R] = temp [R];
  }
```

Finally, in our look at recursive tree-based sorts, we'll play with **quicksort**, which is typically $O(N \log N)$, although it has $O(N^2)$ worst-case scenario. We start by illustrating the process graphically . . .



Notice that we now have an ordered set of 1-element arrays. Schematically, the idea is to pick a **pivot** at random, break the array into 3 arrays — numbers less than the pivot, equal to the pivot, and greater than the pivot — then repeat until dealing with 1-element arrays. Pictorially, the above example becomes . . .



The question arises whether random selection of a pivot is the best approach — certainly on average it's not a bad choice, but there are plenty of quicksort analysts who prefer other methods. Suppose our favourite approach uses ...

```
private static int choose ( Comparable [] A, int L, int R) {...}
```

to find the array location between L and R of our chosen pivot.

```
public static void quickSort ( Comparable [] A)
  {  quickSort ( A , 0 , A.length - 1 ); }
```

calls ...

```
private static void quickSort (Comparable [] A, int L , int R)
  {  if ( R <= L ) return;
     int i = choose (A, L, R);
     partition ( A, L , i , R );
     quickSort ( A, L , i -1 );
     quickSort ( A , i+1 , R );
  }
```

arrange A around its pivot value.

apply recursively to L & R 'halves'.

```
private static void partition ( Comparable [] A , int L, int i, int R)
  {  swap ( A , i , R );
     int j = L-1 , k = R;
     for ( ; ; )
        {  while ( A[++j]. lessThan (A[R]) ) ;
           while ( A[R]. lessThan (A[--k])
              if ( k == L ) break;
           if ( j >= k ) break;
           swap ( A , j , k );
        }
     swap (A , j , R );
  }
```

move pivot somewhere safe

move from the outside in, comparing with the pivot value, swapping when necessary.

put the pivot in the correct spot.

Although that code is relatively easy to follow, it lacks efficiency by splitting off the pivot choice from the actual partitioning. Combining these, with an eye towards another trick ...

```
private static int parti( Comparable [] A, int G, int D)
  { int j = G-1, k = D;
    for ( ; ; )
      { while (A[++j].lessThan(A[D]));
        while (A[D].lessThan(A[--k])) if (k == G) break;
        if (j >= k) break;
        swap (A, j, k);
      }
    swap (A, j, D);
    return j;
  }
static final int CUTOFF = 11;
private void quickSort( Comparable [] A, int L, int R)
  { if (R-L < CUTOFF) return;
    swap (A, (L+R)/2, R-1);
    if (A[R-1].lessThan(A[L])) swap(A, L, R-1);
    if (A[R].lessThan(A[L])) swap(A, L, R);
    if (A[R].lessThan(A[R-1])) swap(A, R-1, R);

    int j = parti(A, L+1, R-1);
    quickSort (A, L, j-1);
    quickSort (A, j+1, R);
  }
private static void hybridSort(Comparable [] A, int L, int R)
  { quickSort (A, L, R);
    insertionSort (A, L, R);
  }
```

guards? don't?

no change here

basically our previous 'choose' method

see 'hybridSort'

median of three pivot choice

This is because insertion sort is faster on small arrays than quicksort.

In the previous code, insertionSort is simply the usual insertion sort method applied to the subarray of A from location L to location R, and would be called by the obvious public static void hybridSort(Comparable [] A){...}.

The 'trick' referred to as median of three gives a partial nod towards what would appear to be the optimal choice of pivot; namely the median of the whole subarray. Choosing the median as pivot would lead to a 'balanced' tree of minimal depth, but is unfortunately rather expensive to compute. Instead we merely pick the median of three effectively almost randomly chosen array elements — the first, the last, and the middle one. These three terms are then put in the right order, but put into the first spot and the last two spots. The penultimate term is then the pivot for subsequent partitions.

Before we leave the topic of sorting, it's worth seeing some relative timings of these three tree-based sorting algorithms on large arrays of integers...

array size	Quicksort	Mergesort	Heapsort
25 000	7	11	8
100 000	27	52	42
400 000	122	238	232

(data from Sedgewick, "Algorithms in C++", 1998, p395.)

So now let's assume that we've been given an array of Comparable things, and that this array has been sorted so that the least is now in position 0. We now want to search for a particular compable thing, i.e., we want to if it's in our array, and if so, return where it is.

Simply checking every term from position 0 onwards is very inefficient — on average it is as likely to be beyond ½ way as before it, so is an $O(N)$ algorithm. Much faster is to look at the ½ way point, then since the array is sorted decide to go left or right, pick the ½ way point of the LH half or RH half, decide again, and repeat until either the thing is found, or found to be absent. This is an $O(\log N)$ algorithm — a great improvement if the array is large.

We'll write this code twice — once using loops, and the other recursively.

```
public static int loopSearch (Comparable[] a,
                              Comparable x)
{
    int low = 0, high = a.length - 1, mid;

    while (low <= high)
    {
        mid = (low + high) / 2;

        if (x.lessThan(a[mid])) high = mid - 1;
        else
            if (x.greaterThan(a[mid])) low = mid + 1;
            else
                if (x.equals(a[mid]))
                {
                    System.out.println("Found it!");
                    return mid;
                }
                else
                {
                    System.out.println("Not there!");
                    return -1;
                }
    }
}
```

both low and high can change here!!

Go left!

Go right!

Found it, so return with location

Not found, so return with 'bad' value. It would be better to define our own exception here.

This continual re-setting of low &/or high to more or less the middle results in a repeated halving of the size of the problem.

The recursive code for this is very similar.

```
public static int curveSearch ( Comparable [] a,
                                Comparable xc )
  {
    return BS ( a, xc, 0, a.length -1 );
  }

private static int BS ( Comparable [] a
                        Comparable xc, int L, int R)
  {
    int mid = (L + R) / 2;

    if ( xc.lessThan (a[mid]))
                        return BS(a, xc, L, mid-1);
    else
      if (xc.greaterThan (a[mid]))
                        return BS (a, xc, mid+1, R);
      else
        if (xc.equals (a[mid])
          {
            System.out.println ("Found it!");
            return mid;
          }
        else
          {
            System.out.println ("Not there!");
            return -1;
          }
  }
```

look left

look right

Again, it would be better to throw a custom built exception