Recursion et friends.

Read chapter 1 of Weiss.

Suppose you are given a 'rule' such as

$$a_n = n\, a_{n-1}$$

and also know that

$$a_1 = 1.$$

Then we can use this to build a sequence of numbers . . .

$$1, \; 2, \; 6, \; 24, \; 120, \; 720, \; \ldots$$

which you recognise as factorials. It's easy to see this by building up from the bottom . . .

$$\underset{a_1}{1}, \quad \underset{2\,a_1}{2 \times 1}, \quad \underset{3\,a_2}{3 \times (2 \times 1)}, \quad 4 \times \underbrace{(3 \times 2 \times 1)}_{4\,a_3}, \quad \ldots$$

Although showing that this will really only produce factorials would take an infinite amount of time!!

We could prove this in an intuitively rigourous inductive way by . . .

① Remark that $a_1 = 1 = (1!)$.

② Notice that (if) we were to assume that

$$a_n = n!$$

then

$$a_{n+1} = (n+1) \times a_n \quad \leftarrow \text{by our 'rule'}$$

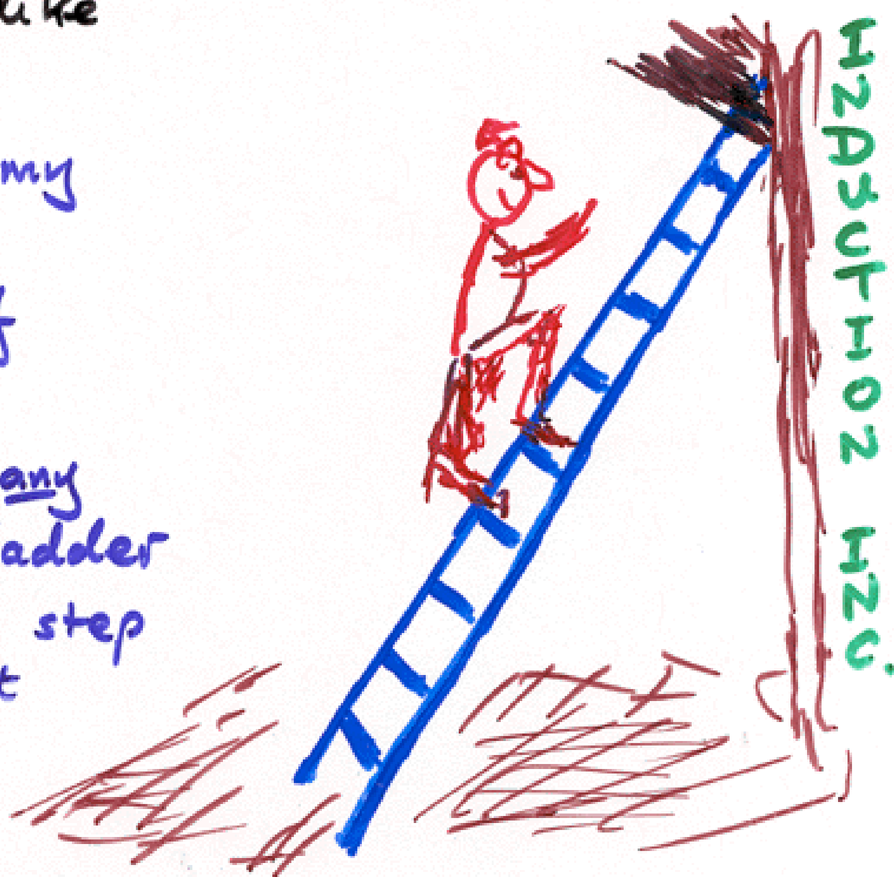
$$= (n+1) \cdot (n!) \quad \leftarrow \text{by our assumption}$$

$$= (n+1)!$$

This is rather like saying . . .

① I <u>can</u> put my foot on the <u>first</u> rung of a ladder.

② IF I'm on <u>any</u> rung of the ladder THEN I can step onto the next rung.

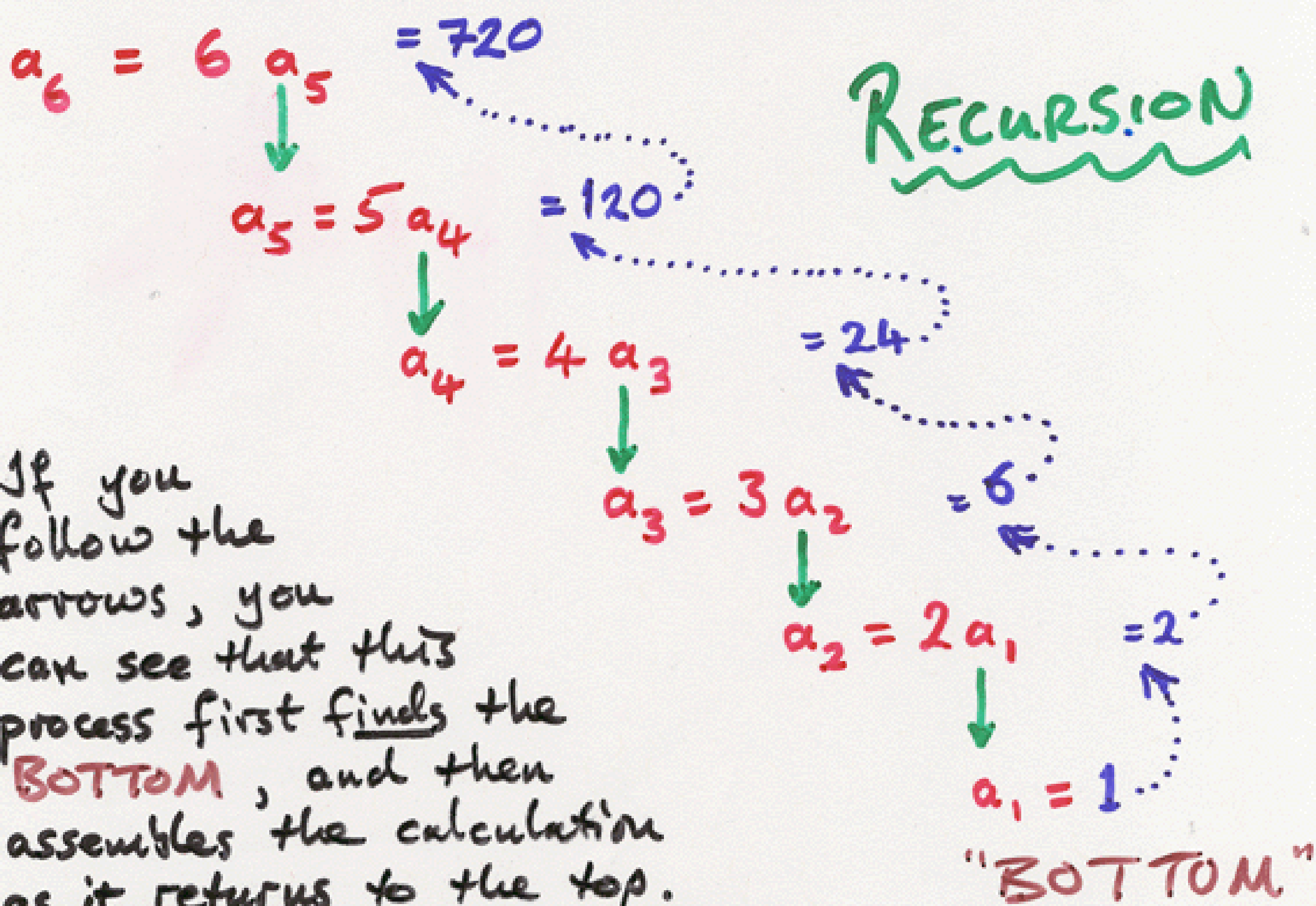


This way of arguing, called **induction**, is very nice because...

a/. We don't have to do infinitely many steps.

b/. It's "jolly obvious" that we've covered every case.

Instead of working from the bottom up, we could work from the top down (provided our 'top' is only 'finitely high')....



If you follow the arrows, you can see that this process first *finds* the BOTTOM, and then assembles the calculation as it returns to the top.

Obviously, if there is no bottom, then we will be waiting a jolly long time for any results!

Let's see what the coded version of this looks like...

```
public static int fact (int n)
      {
          if (n==1)  return 1;
          else         return n*fact(n-1);
      }
```

Then our fact(n) behaves just like our a_n, and it would be invoked by ...

```
int ans = fact(6);
```

for example — producing the same bottom-hungry routine we saw for a_n.

In fact, any sequence defined by a recurrence relation can be converted into recursive code very easily. Without making any comments about efficiency, recursive code is typically very short.

As experiments, first you should run the above code to compute fact(20).

After that, try to find the 100th term in the following Fibonacci-like sequence...

$$a_n = 2a_{n-1} - 3a_{n-2}$$

$$a_1 = 1 \quad , \quad a_2 = 1$$

by running the following code...

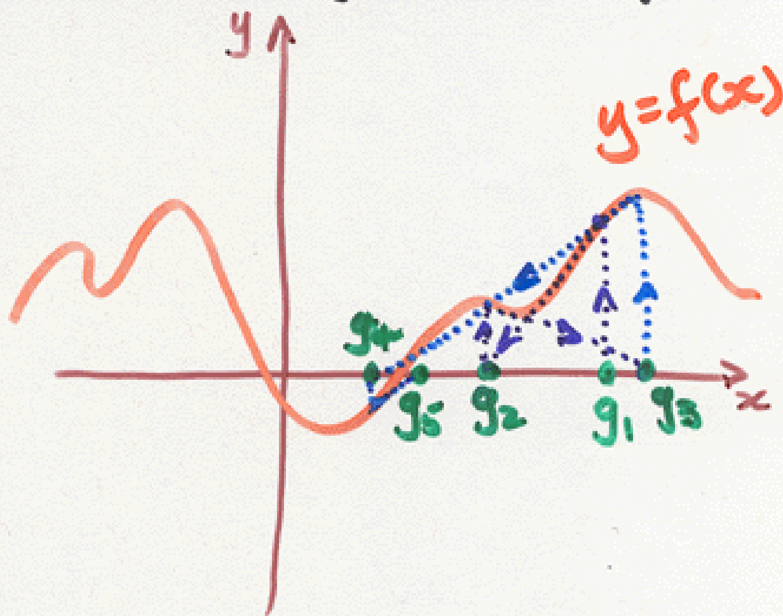
```
fibby (n)
{
    if (n == 1)
            return 1;
    else if (n == 2)
            return 1;
    else
        return 2*fibby(n-1) - 3*fibby(n-2);
}
```

find fibby(100)

Obviously, in both this and the preceding factorial example, you will have to add the appropriate extra stuff to make this into a real Java program — and don't forget to catch the exceptional cases where $n \leq 0$ or is not an integer! (You might find the second example runs a little slowly.)

As you can see, writing this kind of code in these examples is pretty easy.

We can give a non-integer example. In calculus there's a nifty method for finding solutions of equations ascribed to Newton. The calculus flavour of it says that to solve ...



$$f(x) = 0$$

we first make an initial guess g_1. (This guess can be as bad as you like!)

Then the process 'improves' the guess by producing a succession of guesstimates g_n according to ...

$$g_n = g_{n-1} - \frac{f(g_{n-1})}{f'(g_{n-1})}$$

We can use this approach to solve an easy cubic equation; for example ...

$$x^3 - x = 0.$$

Knowing that the answers should be -1, 0 and 1 makes it all the more fun to see which 'guesses' end up at which answers!

Our recursion relation here is . . .

$$g_n = g_{n-1} - \frac{g_{n-1}^3 - g_{n-1}}{3g_{n-1}^2 - 1}$$

g_1 = any actual number you like

which gives the code . . .

```
double acc = 0.00001;   // to stop when accurate enough

double newt (double g)
  {
     double temp = g*g*g - g;

     if (temp < acc && temp > -acc)
        return g;

     else
        return newt (g - (temp/(3*g*g - 1)));
  }
```

which would be run by, for example . . .

```
double ans = newt (3.6);
```

If your initial guess is really close to $\pm\frac{1}{\sqrt{3}}$ or $\pm\frac{1}{\sqrt{5}}$ then funny things happen. Things are also a bit bizarre if your guess g is picked so that

$$-\frac{1}{\sqrt{3}} < g < -\frac{1}{\sqrt{5}} \quad \text{or} \quad \frac{1}{\sqrt{5}} < g < \frac{1}{\sqrt{3}} .$$

We could be more creative in the way we build a recursive program.

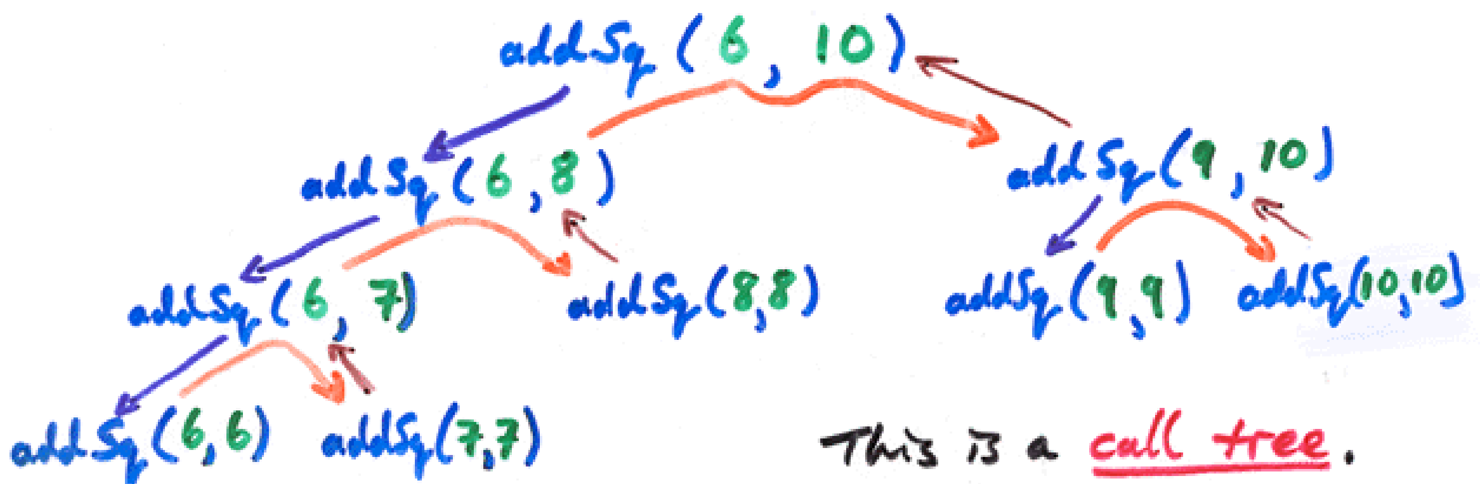
Suppose we wanted to add up the squares of all the integers between n and m (inclusive). Look at the following recursive code ...

```
public static int addSq(int m, int n)   // m ≤ n
  {
    public int temp;   // to find the rough middle

    if (m == n)
        return m * m;
    else
      {
        temp = (m + n) / 2;
        return addSq(m, temp) + addSq(temp+1, n);
      }
  }
```

(repeated halving until only one thing remains)

To understand this, consider $6^2 + 7^2 + 8^2 + 9^2 + 10^2$.



This is a call tree.

We could take a similar sub-dividing approach to write a program to reverse the elements of an array. Consider ...

```
void reverse (int [] A, int m, int n)      // assumes m < n
  {
    if (m < n)
    {
      int temp = A[m];     } swap extremities
      A[m] = A[n];         } of a sub-array
      A[n] = temp;
      reverse (A, m+1, n-1);   <- recurse!
    }
  }
```

Let's see this in action on the following array ...

```
1  4  8  2  3  6  7
7  4  8  2  3  6  1
7  6  8  2  3  4  1
7  6  3  2  8  4  1
```

This might be initiated by mirror(A); where ...

```
void mirror (int [] A)
  {
    reverse (A, 0, A.length - 1);
  }
```

We will see much more of this divide and conquer approach as we progress through the course.

Obviously, the more involved these programs become, the harder it is to be certain that the program really does do what it claims to do. This leads to a whole industry revolving around program verification.

For us, the only easy technique is that of induction, and we'll only consider this in its simplest guises. As a simple illustrative example, we'll prove that ...

$$1^3 + 2^3 + \ldots + n^3 = \frac{1}{4} n^2 (n+1)^2$$

Proof

get on first rung → ① Check the case when $n = 1$ ...

$$1^3 \overset{?}{=} \frac{1}{4} 1^2 (1+1)^2 = \frac{1}{4} 4 = 1 \quad \checkmark$$

show that, if you are on p-th rung, you can climb on to (p+1)-st rung. → ② Assume it's true for $n = p$, then ...

$$\begin{aligned}
1^3 + 2^3 + \ldots + p^3 + (p+1)^3 &= \frac{1}{4} p^2 (p+1)^2 + (p+1)^3 \\
&= (p+1)^2 \left(\frac{1}{4} p^2 + (p+1) \right) \\
&= \frac{1}{4} (p+1)^2 (p^2 + 4p + 4) \\
&= \frac{1}{4} (p+1)^2 (p+2)^2
\end{aligned}$$

hence it's also true for $n = p+1$. //

Read chapter 2 of Weiss

Asymptotic Analysis

Before we look at some nifty applications of recursion, and provoked perhaps by our experiences with the Fibonacci-like sequence, we should look a little more closely at program efficiency.

What concerns us here is how quickly our programs will run. This matters more as the number of computations increases ...

# Steps	Asymptotic behaviour
$150n + 17$	n
$2n^3 + 9000000n^2 - 1$	n^3
$3\log(n^2) - 12$	$\log n$
$5^n + 17n^{84}$	5^n

The important measure is what the LH formula looks like when n is ginormous! Notice that in line 2, the nine million times n^2 is irrelevant when n is "the square of the National Debt"!

Recall (from math) that e^x blows up faster than any polynomial in x, and hence any power (even fractional) of x blows up faster than $\log x$ to any (constant) base.

We use a capital O to mean order, so for example ...

$$2n^3 + 9000000n^2 - 1 = O(n^3)$$

means that the LH side is of order n^3; multiplying factors are ignored — it's growth that counts.

Formal Details

We might think of n as being some parameter, such as the number of elements of an array; and $T(n)$ as some function of n, such as the number of computations in some algorithm. Then ...

① $T(n) = O(f(n))$ if $\exists\, c > 0$ with $T(n) \le c f(n) \;\; \forall n \ge N_0$.

The idea being that we're looking for some function $f(n)$ so that eventually (i.e., as n gets progressively larger) $T(n)$ is bounded above by some multiple of $f(n)$. In other words, the growth of the graph of $f(n)$ describes the shape of the growth of $T(n)$ — or at least provides a worst case scenario. For example ...

$$2n^3 + 9000000n^2 - 1 = O(n^4).$$

There are three other related expressions...

② $T(n) = \Omega(f(n))$ if $\exists\, c > 0$ with $T(n) \geq c\, f(n) \quad \forall n \geqslant N_0$

"bounded below by"

For example...

$$2n^3 + 9\,000\,000\, n^2 - 1 = \Omega(n^2).$$

③ $T(n) = \Theta(f(n))$ if $\exists\, c_1, c_2 > 0$ with $c_1 f(n) \leq T(n) \leq c_2 f(n) \quad \forall n \geqslant N_0$

"asymptotic to"

So $T(n) = \Theta(f(n))$ holds provided both $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$. For example...

$$2n^3 + 9\,000\,000\, n^2 - 1 = \Theta(n^3).$$

We also denote this by $T(n) \sim f(n)$.

"little oh"

④ $T(n) = o(f(n))$ if $\dfrac{T(n)}{f(n)} \longrightarrow 0$ as $n \longrightarrow \infty$.

"negligible compared to"

So $T(n) = o(f(n))$ holds provided both $T(n) = O(f(n))$ and $T(n) \neq \Theta(f(n))$, although really the emphasis is on $T(n)$ being eventually neglible when compared to $f(n)$. For example...

$$2n^3 + 9\,000\,000\, n^2 - 1 = o(n^4).$$

To be honest, we really use these formalisms as shorthand for formal mathematical statements — it's a matter of understanding what a particular function "smells like". It's not hard to see how these expressions combine . . .

$$O(f(n)) + O(g(n)) = O(\text{whichever of } f \text{ or } g \text{ grows } \underline{\text{faster}} \text{ as } n \to \infty)$$

Ditto for $\Omega(f(n)) + \Omega(g(n))$.

$$\Omega(f(n)) + \Omega(g(n)) = \Omega(\text{whichever of } f \text{ or } g \text{ grows } \underline{\text{slower}} \text{ as } n \to \infty)$$

$$O(f(n)) + o(f(n)) = O(f(n))$$

Similar sorts of expressions hold for the various algebraic manipulations.

Applying all this to estimate the running time of code is initially not too hard. As an example, we'll consider the following problem. Suppose we have an array of integers, for example . . .

3 2 -25 14 8 -22 10 1 3 7 -1 3 -12

The exercise is to find the maximum value that can be obtained by adding the terms of any contiguous subsequence.

A rather brain-dead approach, essentially trying everything, would be to consider every starting point, and for each starting point consider every ending point, and for each pair of starting and ending points add up the values of this subsequence, and then compare this with the biggest so far.

Written as code, this gives...

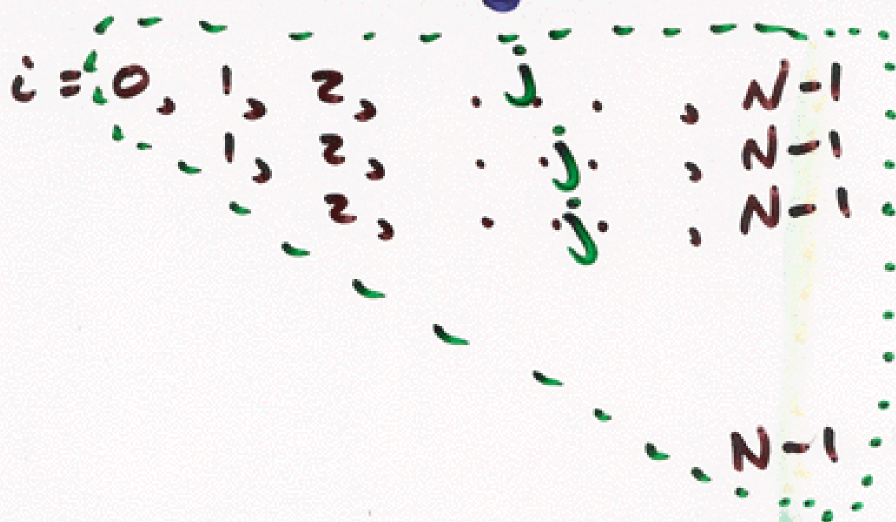
```
public static int maxSSSum (int [] A)
  {
    int maxSum = 0;
→   for (int i = 0 ; i < A.length ; i++)
→     for (int j = i ; j < A.length ; j++)
       {
         int thisSum = 0;
→        for ( int k = i; k <= j ; k++)
             thisSum += A[k];
         if (thisSum > maxSum )
             maxSum = thisSum;
       }
    return maxSum;
  }
```

This set of 3 nested for-loops is easy, but horribly inefficient!

Let's analyse this method to see how many steps are taken as a function of the number of terms in the array.

Analysis...

$a.length = N$, so the 1st **for** loop executes N times. For each run of the 1st loop, the 2nd loop runs $N-i$ times — essentially this is a triangular process



so the 1st two **for** loops run $\frac{1}{2}N(N+1)$ times. Finally the last **for** loop means we're counting the number of triples $0 \le i \le k \le j \le N-1$ — essentially a tetrahedral process giving $\frac{1}{3!}N(N+1)(N+2)$ steps. Ignoring constants, the asymptotic behaviour is hence

$$O(N^3).$$

A simple process, but a horrible number of steps if N is large!!

A slight tidying up of our program can lose the inner k **for** loop, reducing the running time to

$$O(N^2).$$

This is a <u>very</u> valuable improvement.

```
public static int maxSSSum (int [] a)
{
  int maxSum = 0;
  for (int i=0; i<a.length; i++)
  { int thisSum = 0;
    for (int j=i; j<a.length; j++)
    { thisSum += a[j];
      if (thisSum > maxSum)
        maxSum = thisSum;
    }
  }
  return maxSum;
}
```

initialization moved to i-loop

increment thisSum in j-loop since old k-loop gives nothing that's not already here!

Finally we can be a lot sneakier and reduce the running time to

$$O(N).$$

Such success is rare in general ...

```
public static int maxSSSum (int [] a)
{
  int maxSum = 0, thisSum = 0;
  for (int i=0, j=0 ; j<a.length; j++)
  { thisSum += a[j];
    if (thisSum > maxSum)
      maxSum = thisSum;
    else if (thisSum < 0)
    { i = j+1;
      thisSum = 0;
    }
  }
  return maxSum;
}
```

a negative <u>initial</u> portion of a subseq can never give us a <u>new</u> maxSum as i increases from 0!

There's yet another way we could approach this maximum subsequence problem, we can build a recursive attack. Essentially the max subsequence will either

a/. be in the left half of the array

b/. " " " right " " " "

or c/. straddle both the left and right halves.

The corresponding recursive code is . . .

```
public static int maxSS ( int [ ] a )
   { return maxSSR ( a, 0, a.length - 1 ) }

private static int maxSSR ( int [ ] a, int L, int R )
  {              max border sums            current border sums
      int maxL = 0, maxR = 0, Lbord = 0, Rbord = 0;
      int centre = (L + R) / 2 ;  <-- integer arithmetic
base  if ( L == R ) return a[L] > 0 ? a[L] : 0;
case
      int maxLS = maxSSR ( a, L, centre );
      int maxRS = maxSSR ( a, centre + 1 , R );
      for ( int i = centre ; i >= L ; i-- )
        { Lbord += a[i];
          if ( Lbord > maxL ) maxL = Lbord;
        }
      for ( int j = centre + 1 ; j <= R ; j++ )
        { Rbord += a[j];
          if ( Rbord > maxR ) maxR = Rbord;
        }
      return max3 ( maxLS, maxRS, maxL + maxR );
  }        ^-- returns biggest of 3 — easy to write!
```

Before we look at our last example, two principles are worth stating...

Repeated doubling principle: with $x=1$, how often should x be doubled to reach a given N?

$$2^y x = N \implies y = \log_2 N \quad (x=1)$$

Repeated halving principle: with $x=N$, how often should x be halved to reach 1?

$$\left(\tfrac{1}{2}\right)^y N = 1 \implies y = \log_2 N$$

Since

$$\log_2 N = c \log_r N, \text{ for any } r>1 \implies c>0,$$

we're content to say that both of these are $O(\log N)$ principles.

If we need to search a given batch of data, then checking each piece of data term-by-term for a match is clearly $O(N)$. We can improve on sequential search by using the binary search algorithm, which assumes the data to be sorted in advance, and then restricts its attention to the likely half after each unsuccessful match. This repeated halving is of course $O(\log N)$, which is a dramatic improvement....

```
public interface Comparable
  {
      int compares ( Comparable rhs );
      boolean lessThan ( Comparable rhs );
  }
```

This interface would need to be implemented by a class, however, assuming this...

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

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

          if ( a[mid]. compares (x) < 0 )
              low = mid + 1;
          else if ( a[mid]. compares (x) > 0 )
              high = mid - 1;
          else
              return mid;
        }

      throw new ItemNotFound ("Binary Search Failed")
  }
```


We'll look at this example in a lot more detail later on — for now it's enough to notice the repeated halving.

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] ;        <- 'buckets' to show how often a particular number occurs.
      for (int i = 0; i < count.length; i++ )
          count[i] = 0;

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

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

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 ;
nested ---> for (int j = i+1 ; j < A.length ; j++)
               if (A[j] < A[min]) min = j ;
            swap (A , i , min) ;
         }
```

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.

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

No further changes →

happen in either row →

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 the h sequence

sparsely nested

so j can hold a value outside the for-loop

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.

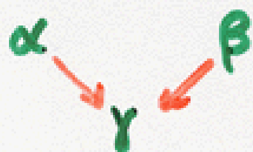
A small industry has grown around improved hopping sequences. For example (Sedgewick) ...

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

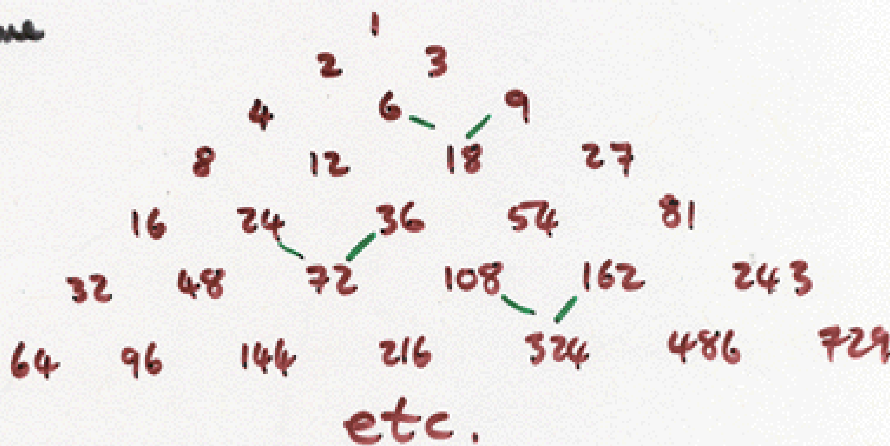
gives a worst case of $O(N^{4/3})$, and we can even get $O(N(\log N)^2)$ by using (Pratt) ...

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

where the numbers come from the triangle

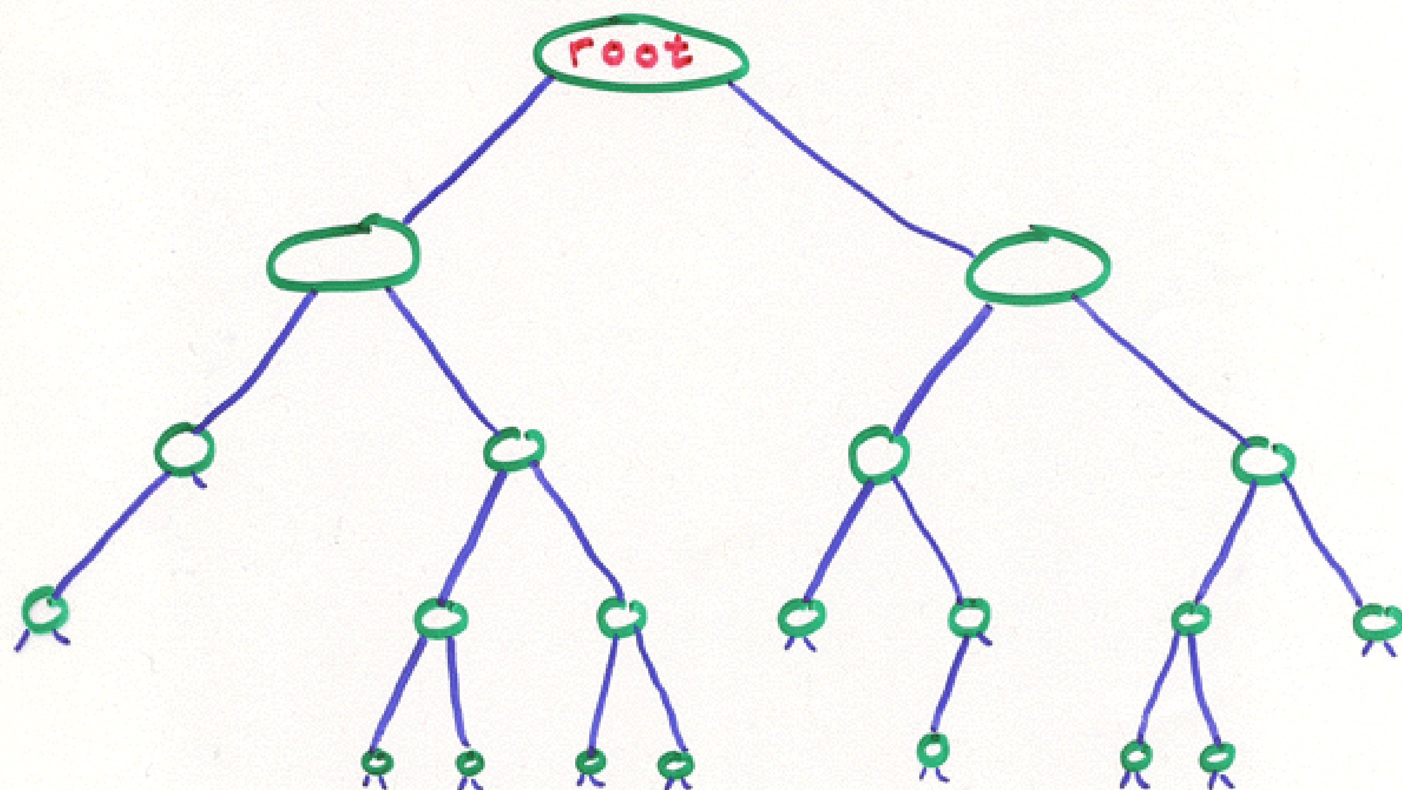


where
$\gamma = 3\alpha = 2\beta$



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.

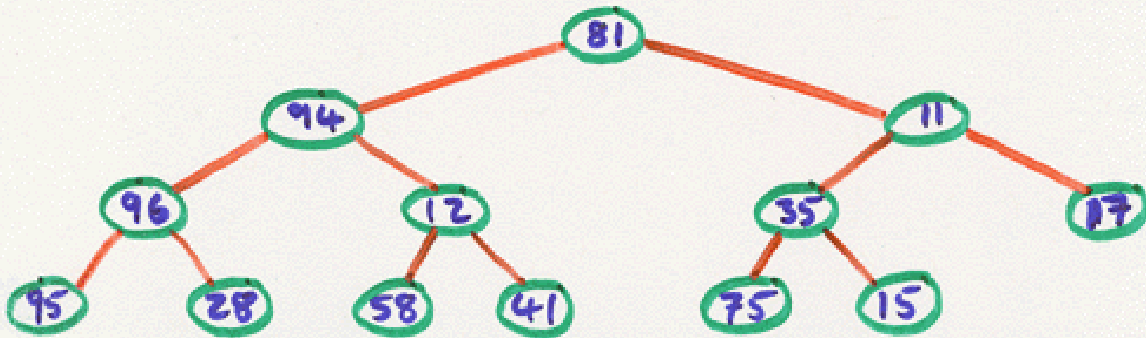
Firstly, some terminology...



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!)

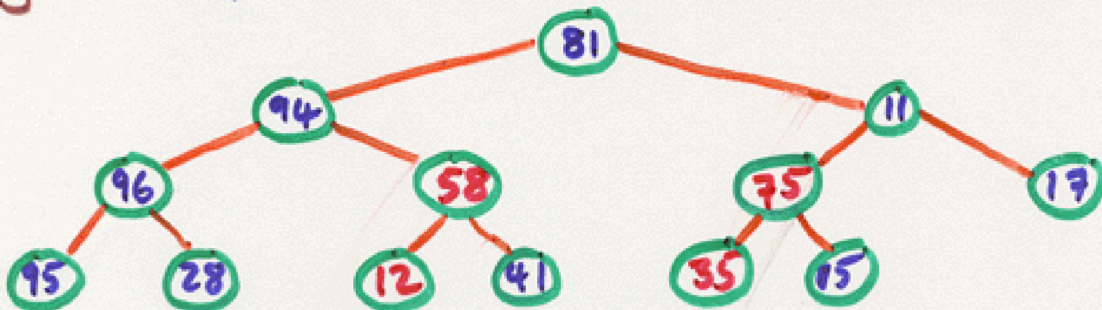
We'll start off with a tree algorithm implemented non-recursively. The worst case scenario for heapsort is O(N log N), see Weiss, section 7.5. The algorithm itself is rather fun. We start by pouring the array into a binary tree ...

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



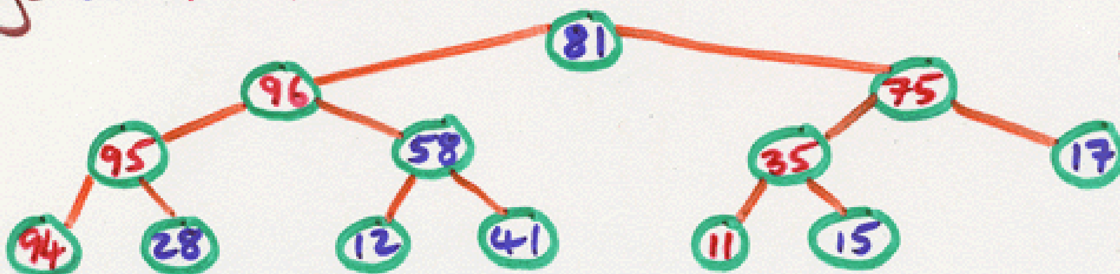
Then we build a heap within this tree by starting at the penultimate layer (working R to L along this layer) and promoting whichever is the larger of the left or right 'child' if this is bigger than the 'parent'...

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



Continuing layer-by-layer ...

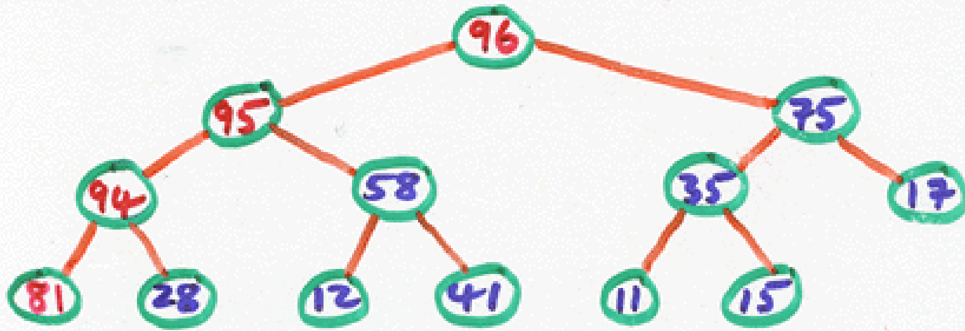
array = 81 96 75 95 58 35 17 94 28 12 41 11 15



Notice how this also reaches down from its layer

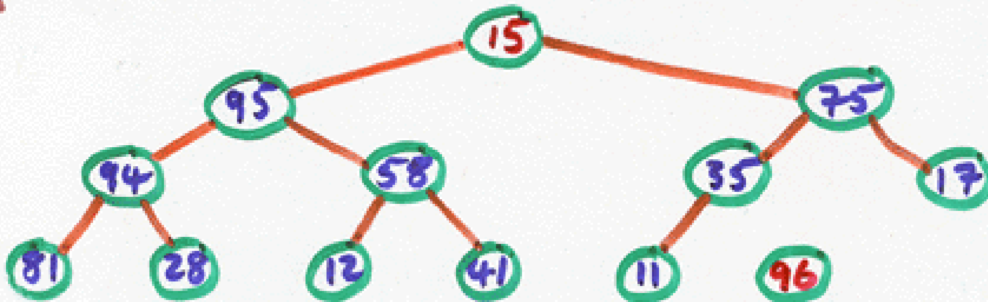
Finally . . .

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



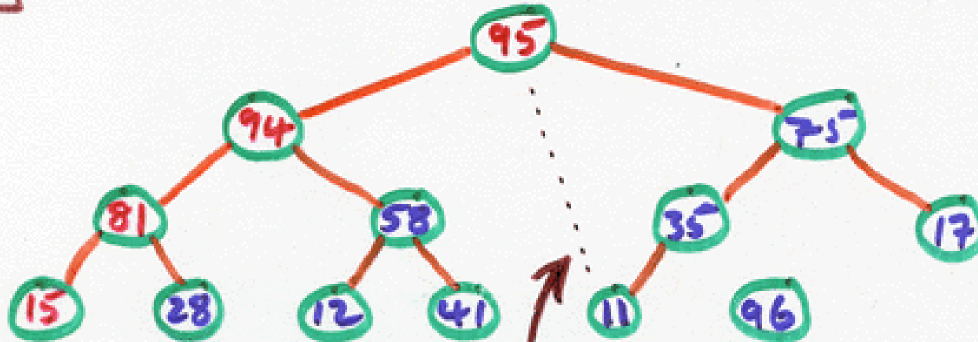
Now that we have a heap, we 'delete' the root value, which is the max value (by swapping it with the far RHS of the array), and successively promote the largest of the left or right 'children'. This process continues until the tree is empty (we slide the effective RH end of the array leftwards by one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

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



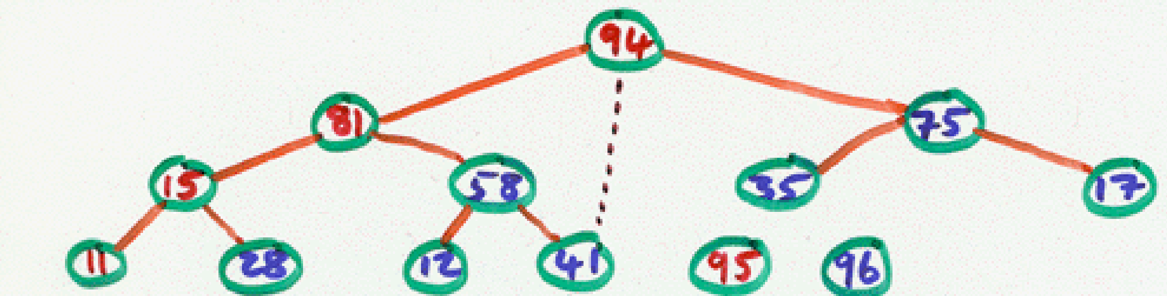
The delete max step via swapping the extremities.

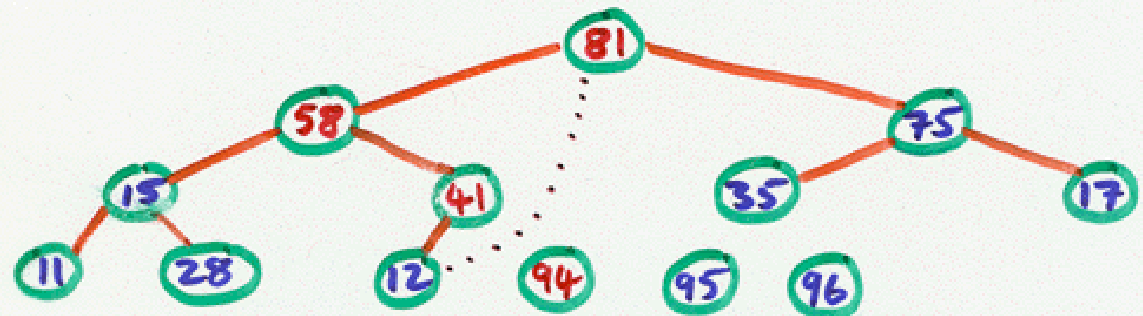
array = 95 94 75 81 58 35 17 15 28 12 41 11 96

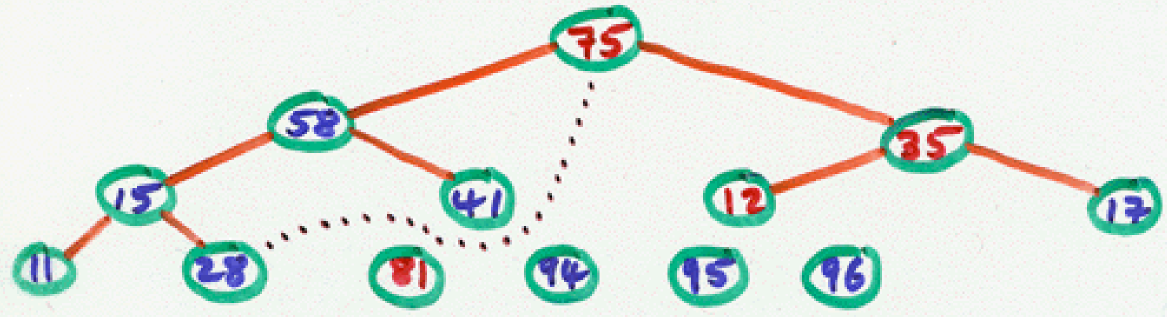


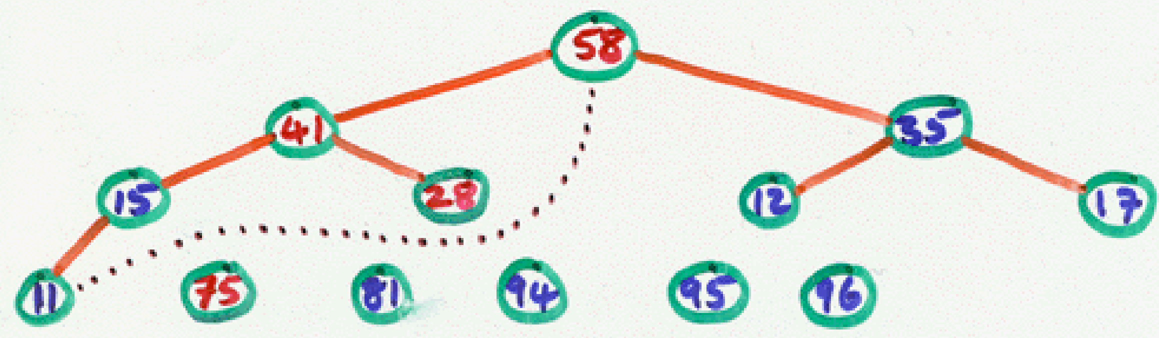
The percdown step to restore the 'heap' nature to this smaller tree.

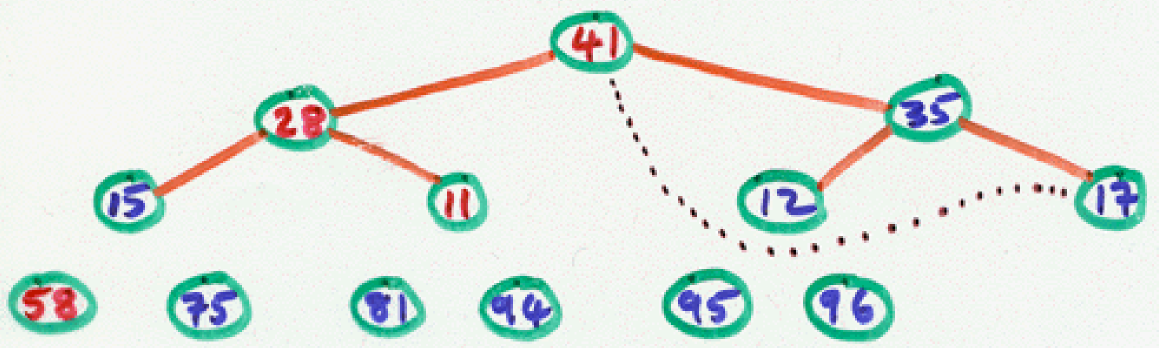
the next delete-swap . . . .

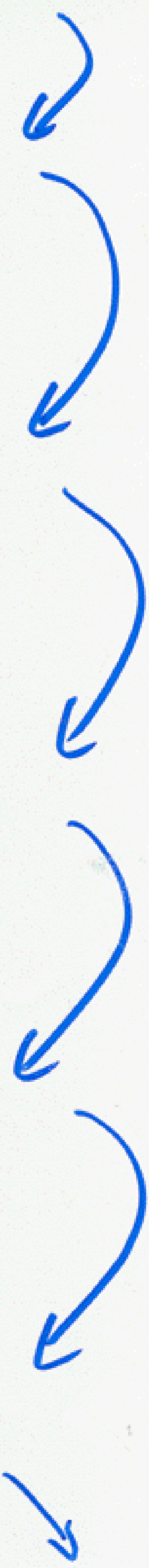

94
81 75
15 58 35 17
11 28 12 41 95 96

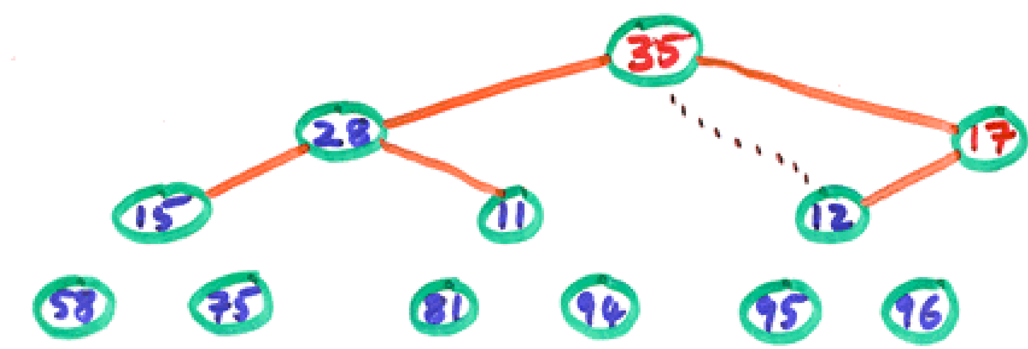

81
58 75
15 41 35 17
11 28 12 94 95 96

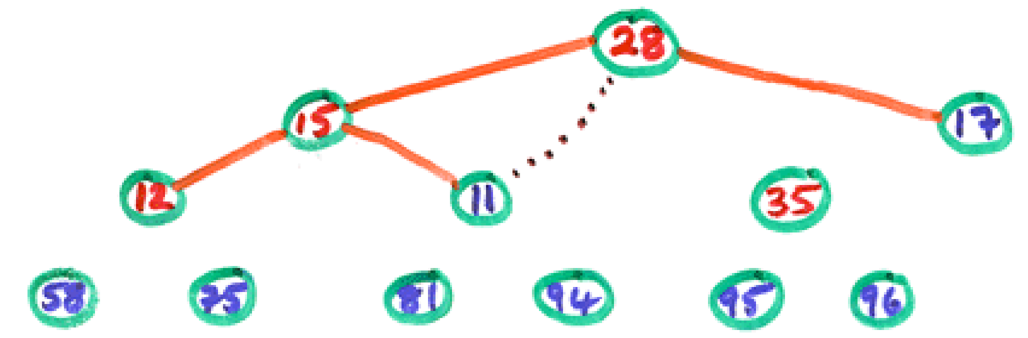

75
58 35
15 41 12 17
11 28 81 94 95 96

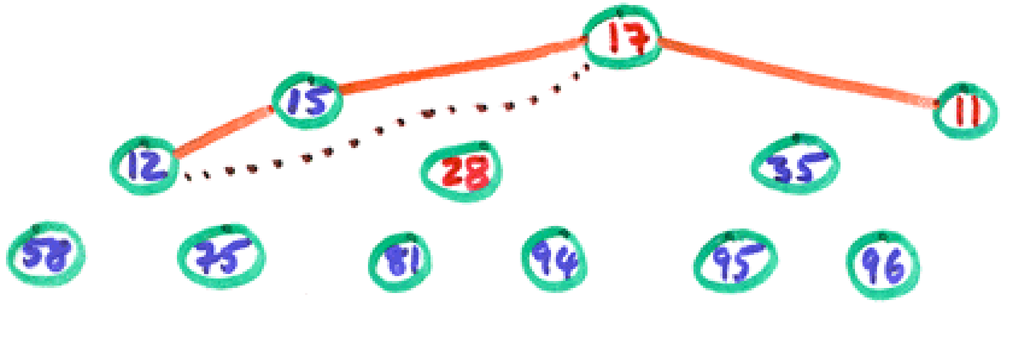

58
41 35
15 28 12 17
11 75 81 94 95 96

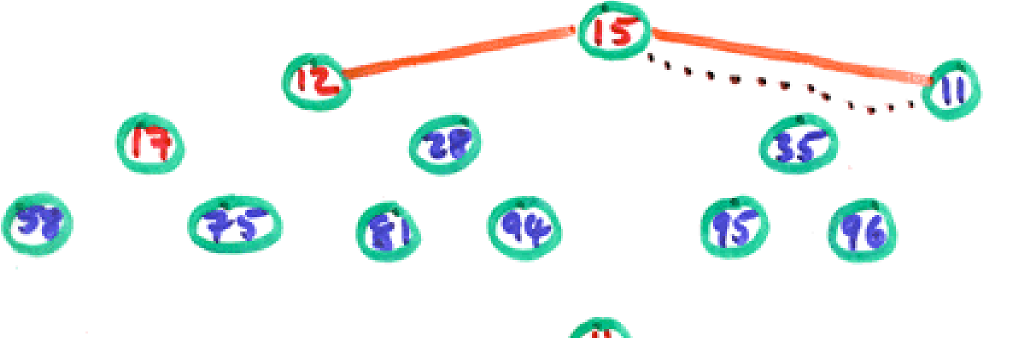

41
28 35
15 11 12 17
58 75 81 94 95 96

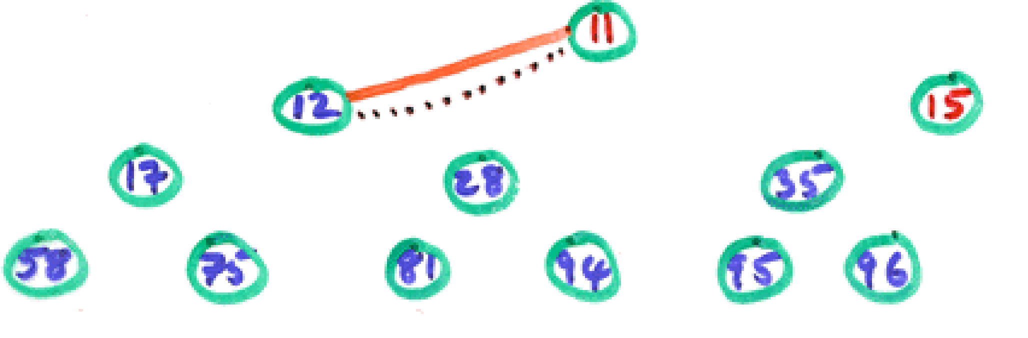


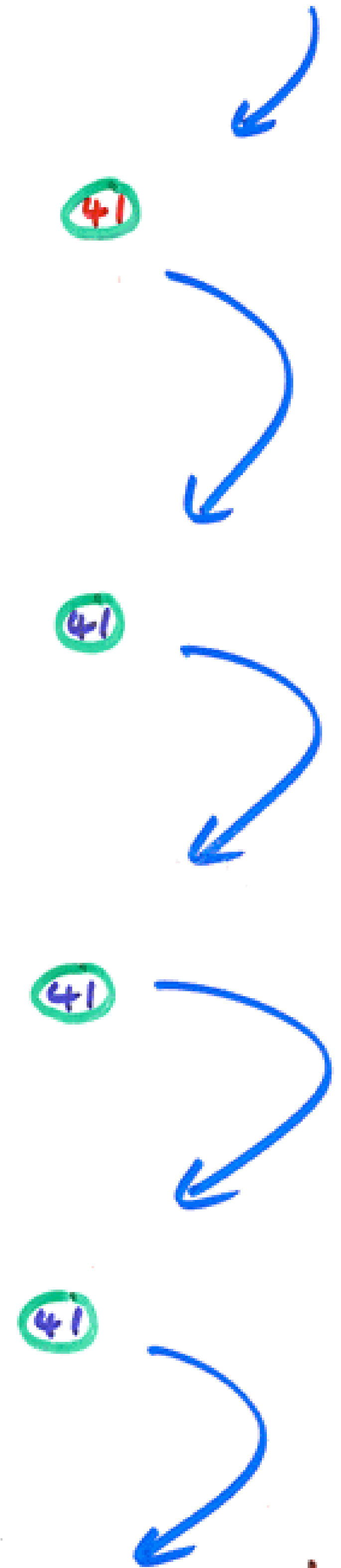












At this step, since $11 < 12$, the 11 and 12 swap back again, and we are finished.

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 );
      }
  }
```

'buildheap', see later code

'deletemax', see later code

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

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

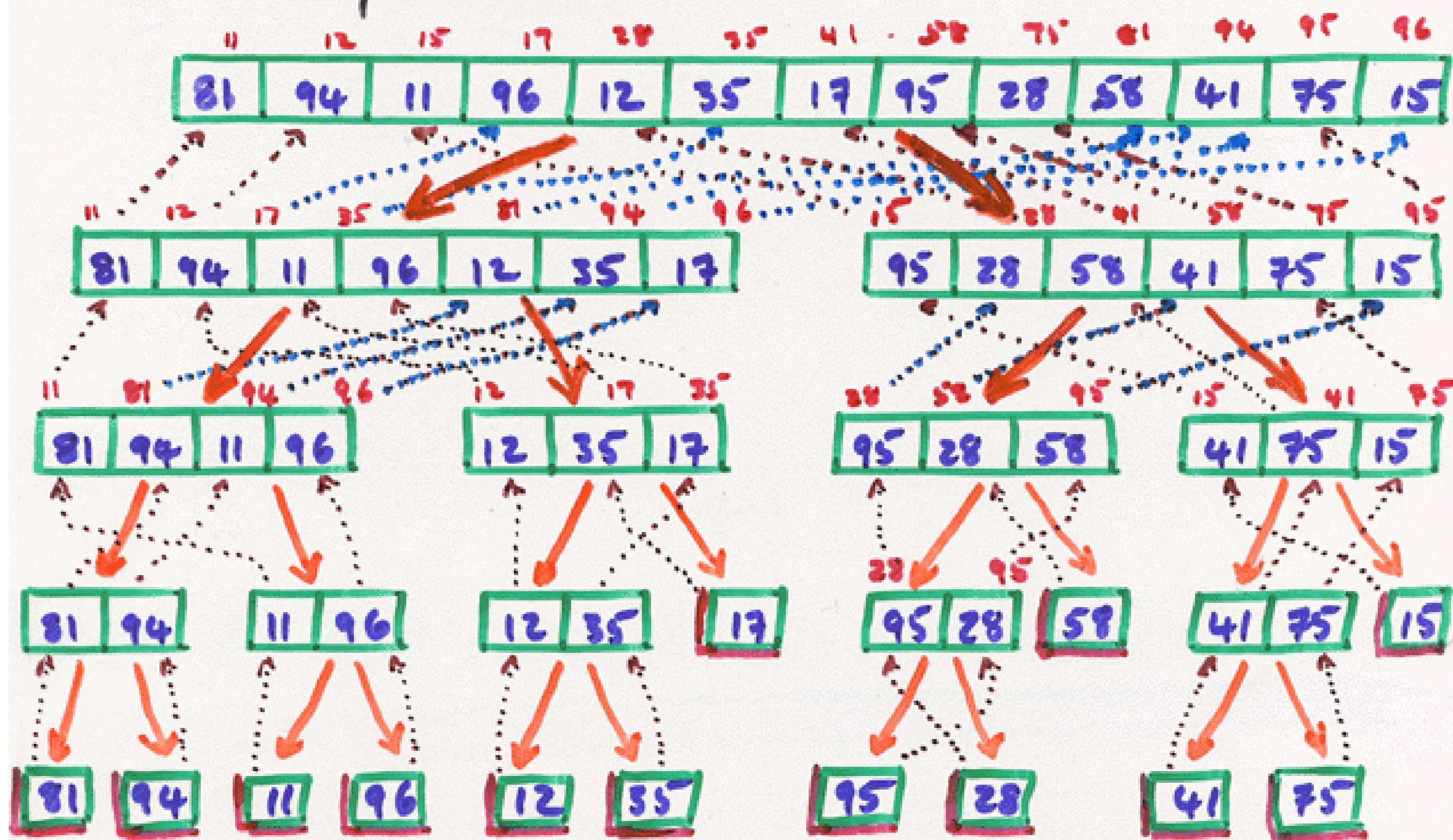
(All this is a lot prettier recursively!!)

```
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!!

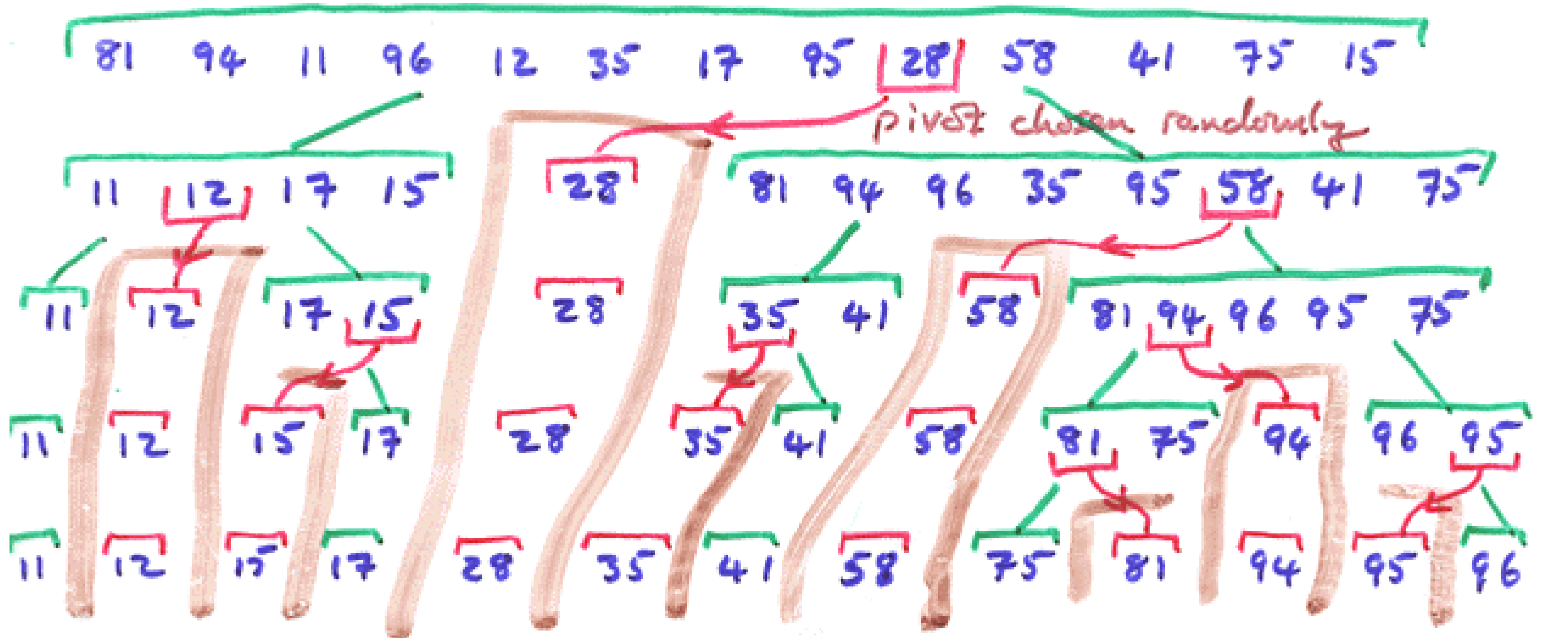
Now merge these two one-element arrays to form a correctly ordered two-element array.

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.

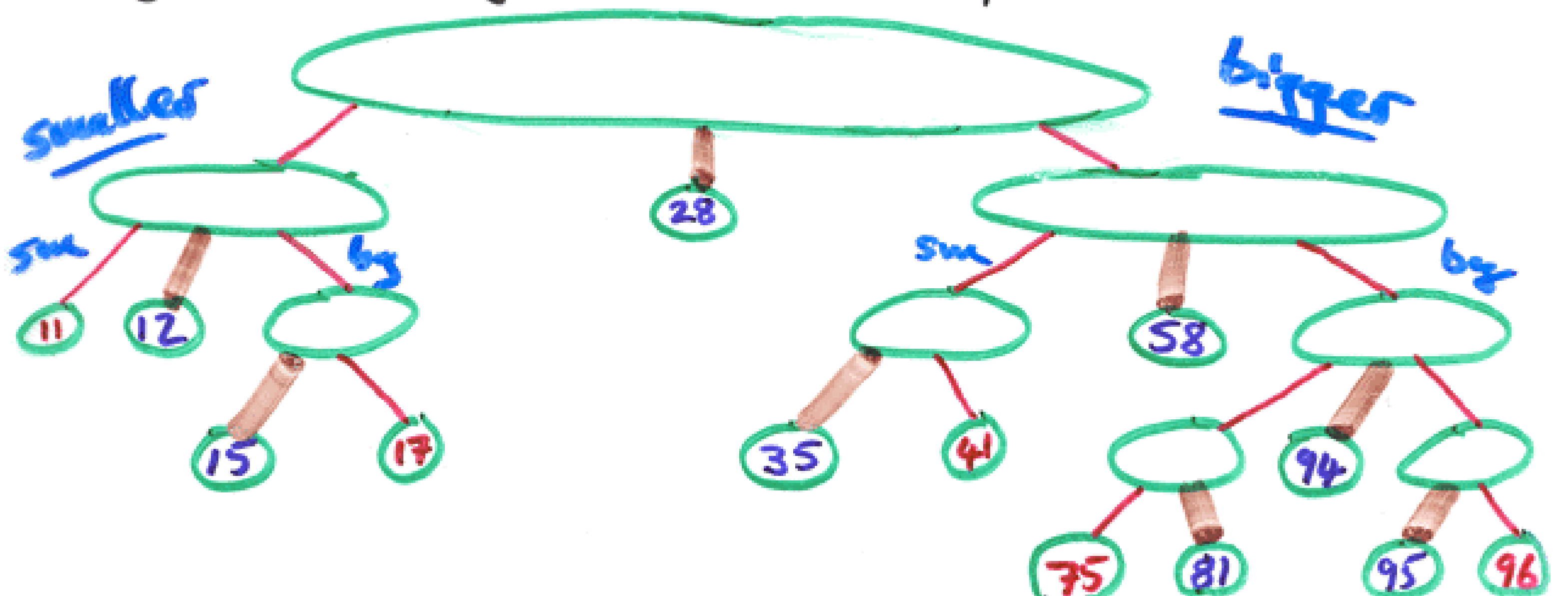
The code for mergesort is now straightforward ...

```
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 was ...

```
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);     <- rearrange A around its pivot value.
     quickSort (A, L, i-1);      <- apply recursively
     quickSort (A, i+1, R);      <- to L & R 'halves'.
   }

private static void partition (Comparable [] A, int L, int i, int R)
   { swap (A, i, R);     <- move the pivot somewhere safe
     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);     <- put the pivot in the correct spot.
   }
```

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

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);
   }
```

Margin annotations:

- G: gauche! D: droit!
- (bracketing the body of `parti`): no change here
- (pointing to `return j;`): basically our previous 'choose' method
- (pointing to `CUTOFF = 11`): see 'hybridSort'
- (bracketing the three `if` lines): median of three pivot choice
- (beside `hybridSort`): 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.)