4. EXAMPLES: SEARCHING AND SORTING

This section of the course is a series of examples to illustrate the ideas and techniques of algorithmic time-complexity analysis. You may or may not have seen these algorithms presented earlier, and if you have they may have been given in a slightly different form.

The emphasis here is on the **analysis techniques, not the algorithms themselves**; in particular the presentations here don’t prioritise elements that would improve performance only by a constant factor (no change under 'O') but give the pseudocode in forms that facilitate a relatively straightforward analysis using the techniques previously described.

However should you need to implement one of these algorithms for a specific purpose, then you *should* consider changes to the (pseudo)code that might improve performance by a constant factor as in reality one is only ever dealing with inputs of finite size, and usually in a known range of sizes, where constant-factor time improvements do make a practical difference.

4.1 SEARCHING ALGORITHMS

The fundamental problem in searching is to retrieve the record associated with a given *search key* in order that the information in the record be made available for processing.

For simplicity during these examples:

- assume that the key is an integer
- the ‘record’ is the position in an array A[0..n-1] where the key is to be found (if anywhere)
- the elementary operation used as a counter will usually be **comparisons of the key with array elements** (‘key=A[i]?’)

The simplest form of search to consider is **sequential search in an unordered array**:

```
ALGORITHM SequentialSearch ( key, A[0..n-1] )
// Sequential search with the search key as a sentinel
// Output is the index of the first element in A[0..n-1] whose
// value is equal to the key or -1 otherwise

A[n] <− key            (to ensure termination if key is
                        nowhere in array positions 0..n-1)
i <− 0
while A[i] ≠ key do
        i <− i+1
if i < n
        return i        (found it at position i)
else
        return -1       (i=n, record not found)
```

SequentialSearch can easily be seen to be **O(n)** in both worst (when the key isn't in the array) and average cases.

In the case of **unsuccessful search** there are **n+1** passes through the loop (key compared to everything i=0..n, test succeeds only in the dummy position n).

In the case of successful search, the key may be found in any one of n positions i=0..n-1 with equal probability. If the key is found in the ith position, i+1 comparisons will have been made. Thus, the **average** number of comparisons for a **successful search** is

$$\frac{1}{n}\sum_{i=0}^{n-1}(i+1) = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{2}(n+1)$$

SequentialSearch can be made more efficient if the array is **sorted** since a search can then be terminated unsuccessfully when an element larger than the search key is found:

```
ALGORITHM SortedSequentialSearch ( key, A[0..n-1] )
// Sequential search of a sorted array
// Output is the index of the first element in A[0..n-1] whose
// value is equal to the key or -1 otherwise

A[n] <− large value     (to ensure termination if key is larger
                         than anything in array positions 0..n-1)
i <− 0
while A[i] < key do
        i <− i + 1
if A[i] = key
        return i        (found it)
else
        return -1       (A[i] is larger, no point in further searching as
                        everything to the right of it will be larger too)
```

For successful search, the situation is as before, requiring ½**(n+1)** passes through the 'while' loop on average.

An unsuccessful search is equally likely to be terminated at the sentinel position n or at positions i=0..n-1. Thus the average number of key comparisons is

$$\frac{1}{n+1}\sum_{i=0}^{n}(i+1) \underset{\substack{\uparrow \\ A[i]=key?}}{+ 1} = \frac{1}{n+1}\sum_{i=1}^{n+1} i + 1 = \frac{1}{2}(n+2) + 1$$

which is just over half the cost of unsuccessful search in an unsorted array.

There are various heuristic techniques which can be used to speed up sequential search algorithms. For example, the most frequently accessed records can be stored towards the beginning of the list, or if information about the relative frequency of access is not available this optimal arrangement can be approximated by moving a record to the beginning of the list each time it is accessed. However none of these techniques will improve the efficiency of the search procedure beyond O(n).

BINARY SEARCH

We assume that the array A is sorted by key into increasing order and at each recursive call look for the key in either the first or the second half of the array. To find out which is the appropriate half we compare the key with the middle element.

```
ALGORITHM BinarySearch ( key, A[b…t] )
// Recursive implementation of binary search,
// called initially (for array A[0..n-1]) with b=0, t=n-1
// Output is the index of the first element in A whose
// value is equal to the key or -1 otherwise

if b = t   (one element left)

        if key = A[b]    (key found at position b)
                return b
        else              (key not found)
                return -1
else
        m <- $\lfloor (b+t)/2 \rfloor$   ('middle' element, using integer division)
        if key $\leq$ A[m] then

                BinarySearch( key, A[b…m] )
                        (look in the first half of the array)
        else
                BinarySearch( key, A[m+1…t] );
                        (look in the second half)
```

(A more efficient implementation would have a three-way branch, checking if key=A[m] before making a recursive call, but it's a bit harder to analyse.)

In order to analyse this algorithm we will assume for simplicity that the array A contains n distinct elements and that the key is indeed among them, with an equal probability of being found at each of the n locations.

Let B(n) be the cost (= number of comparisons) needed to find the key in the n-element array A[0…n-1].

$B(1) = 1$ *(key=A[0]?)*

$$B(n) = \underset{\substack{\uparrow \\ \textit{(key} \le \textit{A[m]?)}}}{1} + \underset{\substack{\nwarrow \\ \text{prob key is} \\ \text{amongst first} \\ \text{m elements}}}{m/n}\, B(m) + \underbrace{(1-m/n)}_{\substack{\text{prob key is amongst} \\ \text{last n-m elements}}} B(n-m)$$

Suppose $n=2^k$, $k=\log_2 n$. Then the array will be divided into two equal halves ($m= 2^{k-1} = n-m$, $m/n=1/2$) so that

$$\begin{aligned} B(2^k) &= 1 + \tfrac{1}{2} B(2^{k-1}) + \tfrac{1}{2} B(2^{k-1}) \\ &= 1 + B(2^{k-1}) \end{aligned}$$

$\rightarrow B_k = B_{k-1} + 1 \qquad B_k \equiv B(2^k)$

$$\begin{array}{lcl} B_k & - & B_{k-1} = 1 \\ B_{k-1} & - & B_{k-2} = 1 \\ \dots & & \dots \\ \underline{B_1} & \underline{-} & \underline{B_0 \;\; = 1} \\ \underline{B_k} & \underline{-} & \underline{B_0 \;\; = k} \end{array}$$

$B(2^k) = B(2^0) + k = B(1) + k$

$\rightarrow B(n) = 1 + \log_2 n$

$\rightarrow B(n) \in O(\log n \mid n \text{ is a power of } 2)$

$\rightarrow$**B(n) ∈ O(log n)** *as log n satisfies 'smoothness' condition etc*

(In fact it can be shown that binary search is O(log n) for both unsuccessful and successful search, for all possible orderings of array elements.)

4.2 SORTING ALGORITHMS

Realistic sorting problems involve files of records containing keys, small parts of the records that are used to control the sort. The objective is to rearrange the records so the keys are ordered according to some well-defined rule, usually alphanumeric, order.

We will just look at the sorting of arrays of integers, corresponding to the keys in a more realistic situation, and for simplicity (so all the array elements are distinct) consider only sorting of **permutations of the integers 1 … n** within an array **A[1..n]**.

(There is no reason n-element arrays always have to be indexed 0..n-1, indexing 1..n will allow the zeroth position to be used for a bit of housekeeping in the cases of InsertionSort and in terms of algebra will in general save us some pain.)

We will measure the cost of the sort in terms of **the number of comparisons of array elements** involved; these will be considered to have unit cost.

InsertionSort

This method considers the elements A[2]..A[n] one at a time, inserting A[i] into its proper place amongst the elements A[1]..A[i-1] whilst keeping these sorted.

While it isn't in general a very efficient method (it is in O(n^2) while competitor algorithms like MergeSort are in O(nlogn)) it can be a good choice if the array is already close-to-sorted and because it is simple to implement also if n is small.

```
ALGORITHM InsertionSort ( A[1…n] )

A[0] <− large negative value    (to ensure termination where an
                                 element at position i is smaller than
                                 everything in positions 1..i-1)
for i <− 2 to n do

        e <− A[i]
        j <− i
        while e < A[j-1] do   <———————   this test is guaranteed to fail
                                          when j=1, so algorithm
                A[j] <− A[j-1]            will always terminate
                j <− j-1

        A[j] <− e
```

Note that if the input is an ordered array the **while** loop is always false at the outset and so only **n-1** comparison operations are performed. InsertionSort can thus be very efficient for close-to-sorted arrays (see later discussion of its use within QuickSort).

Worst case

This corresponds to the input being an array in reverse order. In this case we have to compare e with A[i-1], A[i-2], …, A[0] (for which last value the test will always fail) before leaving the 'while' loop, a total of i comparison operations.

$$I(n) = \sum_{i=2}^{n} i = \sum_{i=1}^{n} i - \sum_{i=1}^{1} i$$

$$= \frac{n}{2}(n+1) - 1$$

So in the worst case **InsertionSort** $\in$ **O(n^2).**

Average case

Suppose (as usual) we are dealing with a permutation of the integers 1…n and that **each initial arrangement is equally likely.**

At the ith iteration of the outer loop element e=A[i] is to be inserted into its correct place among the elements A[1] .. A[i-1]. If e < A[1] (worst case) there are i comparisons carried out, if A[1] < e < A[2] (remember we are dealing with a permutation of 1 .. n, so there are no repeated elements) there are i-1 comparisons, and so on.

Situtation	Probability	# Comparisons
A[i-1]<e	1/i	1
A[i-2]<e<A[i-1]	1/i	2
A[i-3]<e<A[i-2]	1/i	3
…	…	…
A[1] <e<A[2]	1/i	i-1
e<A[1]	1/i	i

The **average number of comparisons for a given value of i** is thus

$$c_{AV}(i) = \frac{1}{i}\sum_{k=1}^{i} k = \frac{1}{i}.\frac{i}{2}(i+1) = \frac{1}{2}(i+1)$$

and therefore

$$\begin{aligned} I_{AV}(n) = \sum_{i=2}^{n} c_{AV}(i) &= \frac{1}{2}\left\{\frac{n}{2}(n+1) - 1 + n - 1\right\} \\ &= \frac{1}{4}(n-1)(n+4) \\ &\in O(n^2) \end{aligned}$$

Since $I_{AV}(n) = \frac{1}{2}I(n) + \frac{1}{2}(n-1)$, InsertionSort -- on average -- is twice as efficient for large arrays than in its worst case.

But we are still $\in O(n^2)$ -- we would like to see a more significant improvement.

MergeSort

The algorithm consists of separating the array A into two parts whose sizes are as nearly equal as possible, sorting these arrays by recursive calls, and then merging the solutions for each part (preserving the sorted order):

```
ALGORITHM MergeSort( A[1..n] )

if n > 1  (something to sort)

    m <− $\lfloor (n/2) \rfloor$
    copy A[1…m] to L        (make a copy of the left half of A)
    copy A[m+1…n] to R    (make a copy of the right half of A)
    Merge( MergeSort(L), MergeSort(R), A )
```

where the procedure Merge merges into a single sorted array of length (p+q) two sub-arrays of length p, q that are already sorted:

```
ALGORITHM Merge( A[1..p], B[1..q], C[1..p+q] )
// Merges two sorted arrays A and B into a single sorted array C

i <− 1; j <− 1; k <− 1
while i ≤ p and j ≤ q do

    if A[i] ≤  B[j]
        C[k] <− A[i]; i <− i + 1
    else
        C[k] <− B[j]; j <− j + 1

    k <− k + 1

if i = p  (possibly some elements left over in array B)
    copy B[j..q] to C[k..p+q]
else    (possibly some elements left over in array A)
    copy A[i..p] to C[k..p+q]
```

(Note that it's possible to implement MergeSort without the need for auxilliary arrays L,R, so saving space -- see eg Sedgwick's book 'Algorithms' for a discussion of this if you are interested.)

Examples with n=8 (original array divided into two (now sorted) arrays of length 4)

A = [1,2,3,4], B=[5,6,7,8]
4 comparisons are carried out:

A[i]] ≤ B[j]?	i	j	test result	C array
1 ≤ 5?	1	1	y	[1]
2 ≤ 5?	2	1	y	[1,2]
3 ≤ 5?	3	1	y	[1,2,3]
4 ≤ 5?	4	1	y	[1,2,3,4]

and copying in the rest (whole) of array B: C=[1,2,3,4,5,6,7,8]

A=[1,3,5,7], B=[2,4,6,8]
7 comparisons are carried out

A[i]] ≤ B[j]?	i	j	test result	C array
1 ≤ 2?	1	1	y	[1]
3 ≤ 2?	2	1	n	[1,2]
3 ≤ 4?	2	2	y	[1,2,3]
5 ≤ 4?	3	2	n	[1,2,3,4]
5 ≤ 6?	3	3	y	[1,2,3,4,5]
7 ≤ 6?	4	3	n	[1,2,3,4,5,6]
7 ≤ 8?	4	4	y	[1,2,3,4,5,6,7]

and copying in the rest of array B: C=[1,2,3,4,5,6,7,8]

The second is an instance of a worst case for Merge, in which the first array to become empty doesn't do so until there is only one element left in the non-empty one.

The worst case for Merge is characterised by **n-1** comparisons, which will be assumed in the following worst case analysis.

Assume also that **the array size n is a power of 2** and let M(n) be the worst case cost (in terms of the number of array element comparisons) to MergeSort an n-element array:

$$M(1) = 0$$
$$M(n) = \underset{\text{2 recursive calls}}{2}M(\underset{\text{size of recursive calls}}{n/2}) + \underset{\text{cost of 'merge' on L and R arrays}}{n - 1}$$

Now we use similar techniques to those used for Strassen's multiplication algorithm and binary search:

Putting $n = 2^k$, $M_k \equiv M(2^k)$:

$$M_k = 2\,M_{k-1} + 2^k - 1$$

and setting

$$M_k = 2^k.\,N_k$$

gives

$$2^k N_k = 2.\,2^{k-1} N_{k-1} + 2^k - 1$$
$$\rightarrow N_k = N_{k-1} + 1 - 1/2^k \quad \textit{(dividing by } 2^k\textit{)}$$

Setting up the usual 'ladder':

$$\begin{array}{rcl} N_k - N_{k-1} &=& 1 - 1/2^k \\ +N_{k-1} - N_{k-2} &=& 1 - 1/2^{k-1} \\ \ldots\quad \ldots && \\ +\underline{N_1 - N_0} &=& \underline{1 - 1/2^1} \\ \underline{N_k - N_0} &=& \underline{k - (1/2^1 + \ldots + 1/2^k)} \end{array}$$

$$\rightarrow N_k = N_0 + k - \sum_{i=1}^{k} \frac{1}{2^i} = N_0 + k - \left(1 - \frac{1}{2^k}\right)$$

x by 2^k:

$$M_k = 2^k M_0 + 2^k k - 2^k + 1$$

=0, as nothing is done for a '1-element array'

$$\rightarrow M(n) = n x M(1) + n \log_2 n - n + 1$$
$$= n \log_2 n - n + 1$$

$$\in \mathbf{O(n \log n \mid n \text{ is a power of } 2)}$$

Since n log n is non-decreasing for n > 1 (there is a turning point between 0 and 1) and

$$(2n) \log (2n) = 2n \log 2 + 2n \log n$$
$$\in O(n \log n)$$

it can be concluded that for all n

$$\mathbf{M(n) \in O(n \log n)}$$

The number of key comparisons done by MergeSort in its worst case, $M(n) \approx n\log_2 n - n$, is very close to the theoretical minimum that must be done in the worst case by any sorting algorithm based on comparison (see later).

However it can nevertheless be bettered -- at least on average -- by another well known divide and conquer algorithm:

QuickSort

In outline:

1. Choose one of the elements in the array to act as the **pivot.**

2. Create a temporary array L to hold all elements **smaller** than the pivot and another array R to hold all those which are **larger**. Ideally the pivot should be chosen so that L and R are as nearly equal in size as possible -- ie the pivot is the **median** -- but in a simple implementation we can just take the pivot to be the first element in the array).

3. **Recursively call QuickSort** on L and R sub-arrays.

4. **Append together** the (sorted) left hand array L, the pivot, and the (sorted) right hand array R.

```
ALGORITHM QuickSort( A[1..n] )

if n > 1  (something to sort)

    l <− 0; r <− 0            (l,r will be the number of entries in
    pivot <− A[1]              the left and right subarrays L,R)
    for i <− 2 to n do

        if A[i] < pivot

            l <− l+1
            L[l] <− A[i]
        else
            r <− r+1
            R[r] <− A[i]

    Append( QuickSort(L), pivot, QuickSort(R), A )
```

Append requires no comparisons:

```
ALGORITHM Append( A[1..p], r, B[1..q], C )
// Copies the p-element array A, element r, and the
// q-element array B into a single array of length p+q+1
for i <− 1 to p do
    C[i] <− A[i]
C[p+1] <− r
for i <− 1 to q do
    C[p+i+1] <− B[i]
```

QuickSort is in some sense the complement of MergeSort.

In MergeSort creating the subinstances is easy, whereas in QuickSort it is more complex ('for' loop indexed by i).

In MergeSort it is the putting together of the subinstances (the Merge procedure) that takes time, whereas the equivalent step in QuickSort is easy, just concatenating the two sorted sub-arrays with the pivot to form one longer sorted array.

Complexity of QuickSort

The complexity of the QuickSort algorithm given above depends very much on the properties of its input. It is inefficient if it happens systematically on most recursive calls that the L and R subinstances are severely imbalanced. In the worst case analysis that follows we will assume l=0 (the left-hand array is empty -- so in fact the array is *already sorted*) on all such calls.

Worst case

Let Q(n) be the worst case cost (number of array element comparisons) of QuickSorting an n-element array:

$Q(0) = 0$ *(no comparisons when the array is defined empty)*

$$Q(n) = \underset{\substack{\uparrow \\ \text{'A[i]>pivot?'} \\ \text{for i=2...n}}}{(n-1)} + \underset{\substack{\nwarrow \quad \uparrow \\ \text{recursive calls} \\ \text{on L,R arrays}}}{Q(l) + Q(r)}$$

In the worst case scenario we are considering

$Q(l) = Q(0) = 0$ *(left sub-array is empty)*

$Q(r) = Q(n-1)$

Then

$$Q(n) = Q(n-1) + n-1$$

So

$$\begin{array}{rll}
Q(n) & - Q(n-1) & = n-1 \\
+Q(n-1) & - Q(n-2) & = n-2 \\
\dots & \dots & \\
+Q(2) & - Q(1) & = 1 \\
+Q(1) & - Q(0) & = 0 \\
\hline
Q(n) & - Q(0) & = \sum_{i=1}^{n}(i-1) \\
\hline
\end{array}$$

$$Q(n) = \sum_{i=1}^{n} i - \sum_{i=1}^{n} 1 = \frac{1}{2}n(n+1) - n = \frac{1}{2}n(n-1)$$

So **in the worst case QuickSort is O(n^2).**

This highlights a **weakness of worst case analysis** since empirically QuickSort turns out to be one of the most efficient sorting algorithms known.

We found that **on average** InsertionSort took about **half** the time that would be expected in its worst case. We need to look at the behaviour of QuickSort on average also.

Average case

As in the case of InsertionSort we will interpret the 'average cost' to mean **the sum of the costs for all possible cases multiplied by the probability of occurrence of each case.**

In this case we have n equiprobable situations defined by the numbers of elements l, r transferred to each of the left, right subarrays:

l	r
0	n-1
1	n-2
…	…
n-2	1
n-1	0

$$Q_{AV}(n) = n-1+\frac{1}{n}\sum_{i=1}^{n}\left(\overset{\overset{l}{\searrow}}{Q_{AV}(i-1)} + \overset{\overset{r}{\downarrow}}{Q_{AV}(n-i)}\right)$$

$$= n-1+\frac{1}{n}\begin{Bmatrix} Q_{AV}(0)+Q_{AV}(1)+\ldots+Q_{AV}(n-1) \\ +Q_{AV}(n-1)+Q_{AV}(n-2)+\ldots+Q_{AV}(0) \end{Bmatrix}$$

$$Q_{AV}(n) = n - 1 + \frac{2}{n}\sum_{i=1}^{n} Q_{AV}(i-1) \qquad (1)$$

Letting n $\rightarrow$ n-1 in (1) gives

$$Q_{AV}(n-1) = n - 2 + \frac{2}{n-1}\sum_{i=1}^{n-1} Q_{AV}(i-1) \qquad (2)$$

n x (1) – (n-1) x (2) $\rightarrow$

$$nQ_{AV}(n) - (n-1)Q_{AV}(n-1) = n(n-1) - (n-1)(n-2) + 2Q_{AV}(n-1)$$

or re-arranging the terms to have all occurrences of $Q_{AV}(n)$ on the left hand side, all occurrences of $Q_{AV}(n-1)$ on the right hand side:

$$nQ_{AV}(n) = (n+1)Q_{AV}(n-1) + 2(n-1)$$

$$Q_{AV}(n) = \underbrace{\frac{n+1}{n}}_{\text{‘}b_n\text{’}} Q_{AV}(n-1) + 2\frac{(n-1)}{n}$$

($\prod_{i=1}^{n} b_i = \frac{2}{1} \times \frac{3}{2} \times \ldots \times \frac{n}{n-1} \times \frac{n+1}{n} = n+1$)

to get rid of multiplying coefficient substitute $Q_{AV}(n)=(n+1)S(n)$

$$\rightarrow (n+1)S(n) = \frac{(n+1)}{\cancel{n}} \times \cancel{n}S(n-1) + 2\frac{(n-1)}{n}$$

Divide by (n+1):

$$\begin{aligned} S(n) &= S(n-1) + 2\frac{n-1}{n(n+1)} \\ &= S(n-1) + \frac{4}{n+1} - \frac{2}{n} \end{aligned}$$

So

$$
\begin{aligned}
& S(n) - S(n-1) = \frac{4}{n+1} - \frac{2}{n} \\
+\ & S(n-1) - S(n-2) = \frac{4}{n} - \frac{2}{n-1} \\
+\ & S(1) - S(0) = \frac{4}{2} - \frac{2}{1}
\end{aligned}
$$

$$S(n) - S(0) = 4\sum_{i=1}^{n}\frac{1}{i+1} - 2\sum_{i=1}^{n}\frac{1}{i}$$

$$\rightarrow S(n) = S(0) + 4\sum_{i=1}^{n}\frac{1}{i+1} - 2\sum_{i=1}^{n}\frac{1}{i}$$

These summations are a problem. There is no explicit formula for sums of these forms, they need to be estimated.

Noting that the function $\frac{1}{x+1}$ is non-increasing and that $-\frac{1}{x}$ is conversely non-decreasing, and using the results on pp.19-20 for approximating a sum by an integral, it can be seen that

$$
\begin{aligned}
S(n) \ \le\ & S(0) + 4\int_0^n \frac{1}{x+1}dx + 2\int_1^{n+1}\left(-\frac{1}{x}\right)dx \\
=\ & S(0) + 4\int_0^n \frac{1}{x+1}dx - 2\int_1^{n+1}\frac{1}{x}dx \\
=\ & S(0) + 4 \ln(n+1) - 2 \ln(n+1)
\end{aligned}
$$

('ln' is $\log_e$, where the number e = 2.718...)

$$= S(0) + 2 \ln(n+1)$$

This is the solution for the new function S(n).

To get the solution for the original function Q(n), the number of comparisons performed by QuickSort on an input of size n, multiply by (n+1) and use the fact that $Q_{AV}(0) = S(0) = 0$ to give the upper-bound result (all we need for a 'O' estimation of time cost):

$$\mathbf{Q_{AV}(n) \leq 2(n+1)ln(n+1)}$$

For large n, for which $n+1 \approx n$ and hence $\ln(n+1) \approx \ln n$

$$\mathbf{Q_{AV}(n \leq 2n\ ln\ n}$$

and so

$$\mathbf{Q_{AV}(n) \in O(n\ log\ n)}$$

The above implementation of QuickSort, which performs very well for randomly ordered files, is a good general-purpose sort. However it is possible to improve the basic algorithm both with regard to speeding up the average case and making it less likely that bad cases occur.

The two main ways in which QuickSort can be speeded up (by a constant factor) are to use a different sorting algorithm for small subfiles and to choose the pivot element more cleverly so that worst cases are less likely to consistently occur.

Small subfiles

The QuickSort algorithm as formulated above just tests whether the array to be sorted has more than one element.

It can be made much more efficient if a simpler sorting algorithm is used on subfiles smaller than a certain size, say M elements.

For example the test could be changed to

```
if (number of elements in A) ≤ M
        InsertionSort(A)
else
        QuickSort(A)
```

where InsertionSort is chosen because it was seen to be close to linear on 'almost sorted' files (the partitioning process in QuickSort necessarily delivers subfiles that are closer to the sorted state than the parent they were derived from).

The optimal value of M depends on the implementation but its choice is not critical; the modified algorithm above works about as fast for M in the range from about 5 to 25. The reduction in running time is on the order of **20%** for most applications.

Choice of pivot element

In the implementation above we arbitrarily chose the pivot element to be A[1], though we noted the pivot should ideally be chosen to be the **median** element in size so that the L and R arrays were as nearly equal in their number of members as possible. In the case that the file to be sorted is in either ascending or descending order the choice of A[1] as the pivot leads to a cost $\in O(n^2)$.

One way of avoiding this worst case would be to choose a **random element** from the array as the pivot, in which case the worst case would happen with very small probability -- this is a simple example of a **probabilistic algorithm** in which randomness is used to achieve good performance almost always, regardless of the arrangement of the input.

A more effective improvement however, and one which brings us closer to the ideal of choosing the pivot to be the median, is to take three elements from the array -- say from the beginning, middle and end -- then use the median of these as the partitioning element. This **median-of-three** method makes the worst case much less likely to occur in any actual sort, since in order for the sort to take time in $O(n^2)$ two out of the three elements examined must be among the largest or smallest elements in the array, and this must happen consistently throughout most of the partitions.

In combination with a cutoff for small subfiles, the median-of-three can improve the performance of QuickSort by about **25%**.