COMP1917: Computing 1

19. Sorting and Efficiency

Reading: Moffat, Section 12.1,12.6

Overview

- Efficiency
- Sorting
- SelectionSort
- MergeSort
- Analysis

Efficiency

As well as asking whether our programs are effective, we also need to consider whether they are efficient.

When you click a button on a Web page, you are much happier if your request is processed in two seconds as opposed to, say, two minutes.

When we write a program, it is legitimate to ask:

- can this program be made to run more efficiently?
- can a new program be written which achieves the same result more efficiently? (i.e. using less time, or less memory)

Response Delayed is Response Denied

It is possible to write a program which would, in theory, choose a perfect chess move by exhaustively searching all possible future move sequences, or break the security of an Internet financial transaction by exhaustively searching all possible cryptographic “keys”.

However, such a program would take an exponentially long time to run (perhaps longer than the age of the universe).

We consider these transactions to be secure not because it is absolutely impossible to break them, but because it is practically impossible in the sense that it would take an extremely long time to do so.

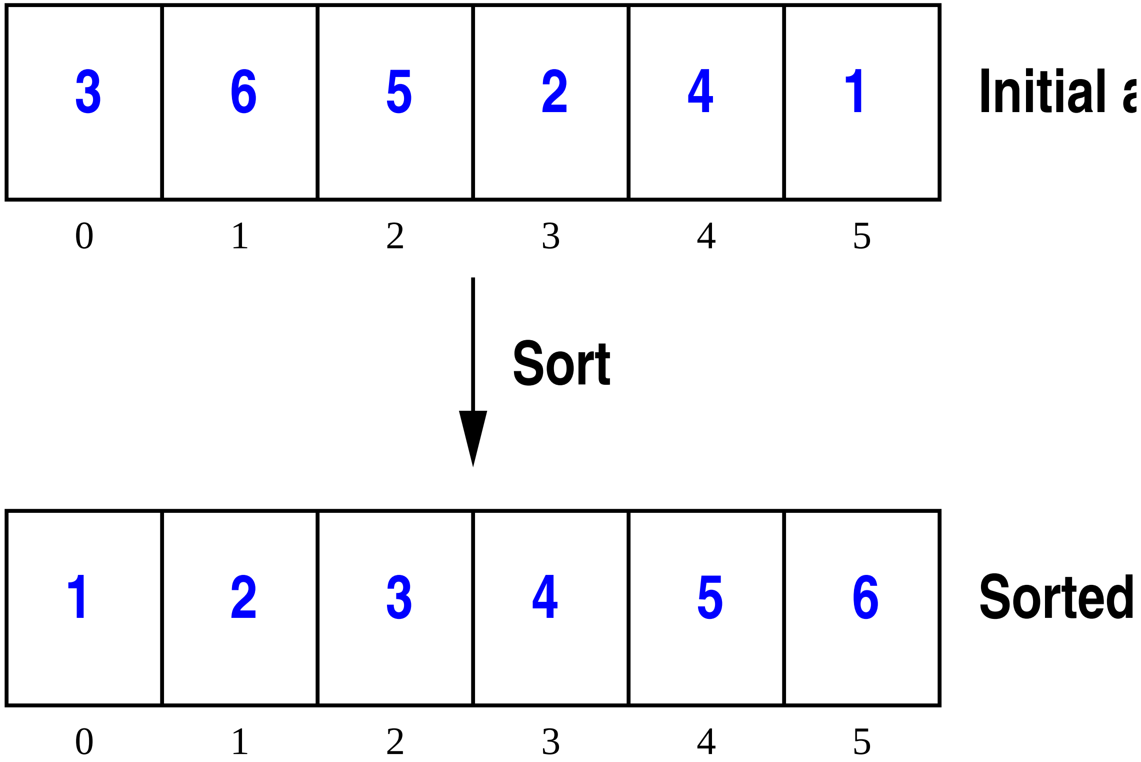
Sorting

- Aim: rearrange a sequence of items so they are organized in non-decreasing order by key
- Advantages
  - sorted sequence can be searched efficiently
  - items with equal keys are located together
- The problem of sorting
  - naive approaches lead to very slow algorithms
  - careful design can lead to efficient solutions

Sorting — Nature of the Problem

- Input: (unsorted) sequence of items stored in a data structure (e.g., array, linked list, etc.)
- Output: sequence of items sorted in non-decreasing order
- We shall consider the input to be an array consisting of n unsorted items (integers) in cells $0 \ldots n-1$ and the output to be in the same array cells $0 \ldots n-1$ but in sorted order
- By modifying the way in which we compare items it is quite straightforward to extend these algorithms to work with other types of items: floating-point numbers, strings, structs, etc.

Sorting — Illustration



Sorting Algorithms

- Slow sorting algorithms
  - SelectionSort
  - InsertionSort
  - BubbleSort
- Fast sorting algorithms
  - MergeSort
  - HeapSort
  - QuickSort

We will only discuss SelectionSort and MergeSort in detail.

SelectionSort

- First scan the array to find the minimum item, and move it to the front by swapping it with whatever item was previously there.
- Next, find the minimum of the remaining $n - 1$ items and "swap" it into the 2nd position of the array.
- Continue in this manner – finding the 3rd, 4th, 5th, etc. item and "swapping" each item into its correct position when it is found.
- Repeat this for all items, until the entire array is sorted.

SelectionSort code

```
void selectionSort(int a[], int n ) {
    int i, j, min, tmp;

    for( i=0; i < n; i++ ) {
        min = i; // initial minimum is first unsorted item

        // find index of minimum item
        for( j = i+1; j < n; j++ )
            if( a[j] < a[min] )
                min = j;

        // swap minimum item into place
        tmp = a[i];
        a[i] = a[min];
        a[min] = tmp;
    }
}
```

SelectionSort

3	6	5	2	4	1
1	6	5	2	4	3
1	2	5	6	4	3
1	2	3	6	4	5
1	2	3	4	6	5
1	2	3	4	5	6

Analysis of Algorithms

How can we find out whether this program is efficient or not?

- empirical approach - write the program, run it several times with different input data, and measure the time taken (we will look at this later).
- theoretical approach - try to count the number of "primitive operations" performed by the algorithm and assume that each primitive operation takes about the same amount of time.
- $T(n)$ = running time of algorithm on input of size n

SelectionSort – Analysis

- Consider sorting a sequence of n items:
  - 1st time we execute outer loop, need to compare n items
  - 2nd time we execute outer loop, need to compare $n-1$ items
  - ...
  - nth time we execute outer loop, need to compare only one item
- Summing this sequence:

$$T(n) \approx n + (n-1) + (n-2) + \cdots + 1 = \sum_{k=1}^{n} k = \frac{n(n+1)}{2} \simeq (\frac{n^2}{2})$$

Key Idea: Merging Sorted Sequences

We will now discuss a more efficient sorting algorithm known as MergeSort.

The key idea is that two already sorted sequences of length r and s can be merged into a single sorted sequence of length $r+s$ in time proportional to $r+s$.

Merging Sorted Sequences

```
/* merge two sorted arrays a[] and b[] of length r and s
        into a single sorted array c[] of length r+s     */
void  merge( int a[], int r, int b[], int s, int c[] )
{
   int i=0, j=0, k=0;

   while(( i < r )&&( j < s )) {
       if( a[i] < b[j] ) // transfer whichever item is smaller
           c[k++] = a[i++];
       else
           c[k++] = b[j++];
   }
   while( i < r )
       c[k++] = a[i++];  // copy any remaining items from a[]
   while( j < s )
       c[k++] = b[j++];  // copy any remaining items from b[]
}
```


Analysis

Question:

How do we know that this code will run in time proportional to $r+s$?

Analysis

Answer:

- there are three while loops
- whenever a statment inside one of the while loops is executed, either i or j will be incremented.
- in the beginning, i and j are both equal to zero
- In the end, i is equal to r and j is equal to s
- therefore, a total of $r+s$ "statements" have been executed

Copying Back

The `merge()` function transfers the items from the original arrays `a[]` and `b[]` into a new array `c[]`. We will need another function to copy the items from `c[]` back to the original array `a[]`:

```
/*  copy m items from array c[] to array a[]  */
void  copy( int a[], int m, int c[] )
{
    int k;
    for( k=0; k < m; k++ ) {
        a[k] = c[k];
    }
}
```

Clearly, the time taken for the copying is proportional to the number of items copied.

MergeSort

The strategy for MergeSort is this:

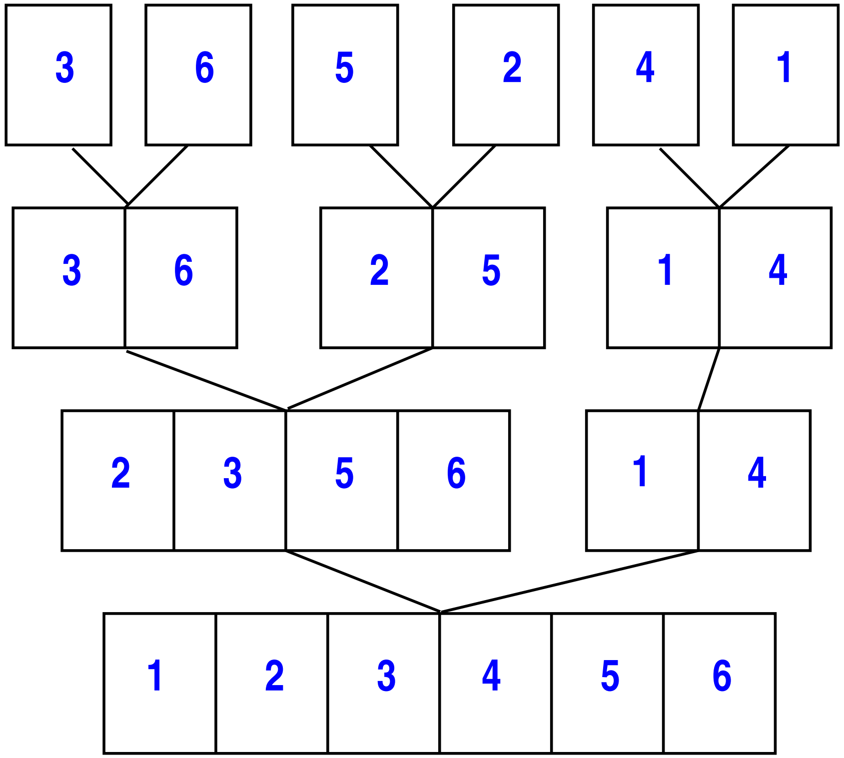
- first make sure that, for each successive “pair” of items, the first item of the pair is smaller than the second item.
- next, merge each “pair” of sorted pairs into a sorted sequence of 4 items
- continue in this way, merging sorted sequences of 4 items into sorted sequences of 8, 16, 32, 64, etc. items
- keep going until the “sorted sequence size” is large enough to fill the entire array.

MergeSort code

```
void MergeSort( int a[], int n )
{
    int *c = (int *)malloc( n * sizeof( int ));
    int k,r;

    for( r = 1; r < n; r = 2*r ) {
        // merge blocks of length r into blocks of length 2*r
        for( k = 0; k + 2*r < n; k = k + 2*r ) {
            merge( &a[k], r, &a[k+r], r, c );
            copy(  &a[k], 2*r, c );
        }
        if( k+r < n ) { // merge final blocks of length r, n-(k+r)
            merge( &a[k], r, &a[k+r], n-(k+r), c );
            copy(  &a[k], n-k, c );
        }
    }
}
```


MergeSort



Analysis of MergeSort

- In each iteration of the outer loop, several pairs of blocks are merged which disjointly cover the entire sequence (i.e. their total length is n).
- Each pair of blocks is merged in time proportional to the sum of their lengths.
- Therefore, the entire loop is executed in time proportional to n.

Question:

How many times is the outer loop executed?

Logarithms

Here is a table showing the value of r at each iteration of the loop:

iteration	0	1	2	3	4	5	6	7	8	...	i
block size	1	2	4	8	16	32	64	128	256	...	$r = 2^i$

Iteration continues until $n = r$, so $n = 2^i$ or $i = \log_2(n)$

Therefore, the entire MergeSort algorithm runs in time proportional to $n \log_2(n)$

Empirical Studies

We can run the two programs using a Unix utility called "`time`":

```
% time ./ssort < r6.in > tmp

real     23m30.284s
user     23m30.036s
sys      0m0.092s
%
```

The "user" component is the best estimate of how much CPU time the program has used for the actual computation.

Comparison

n	10^4	10^5	10^6	$\dots$	10^9
$n^2/2$	5×10^7	5×10^9	5×10^{11}	$\dots$	5×10^{17}
SelectionSort	0.14 sec	14 sec	23.5 min	$\dots$	45 years
$n \log_2(n)$	1.3×10^4	1.7×10^5	2×10^7	$\dots$	3×10^{10}
MergeSort	0.01 sec	0.08 sec	0.84 sec	$\dots$	21 min

These values have all been obtained using the `time` utility, except for those in the last column (which are estimates).

Top-Down MergeSort

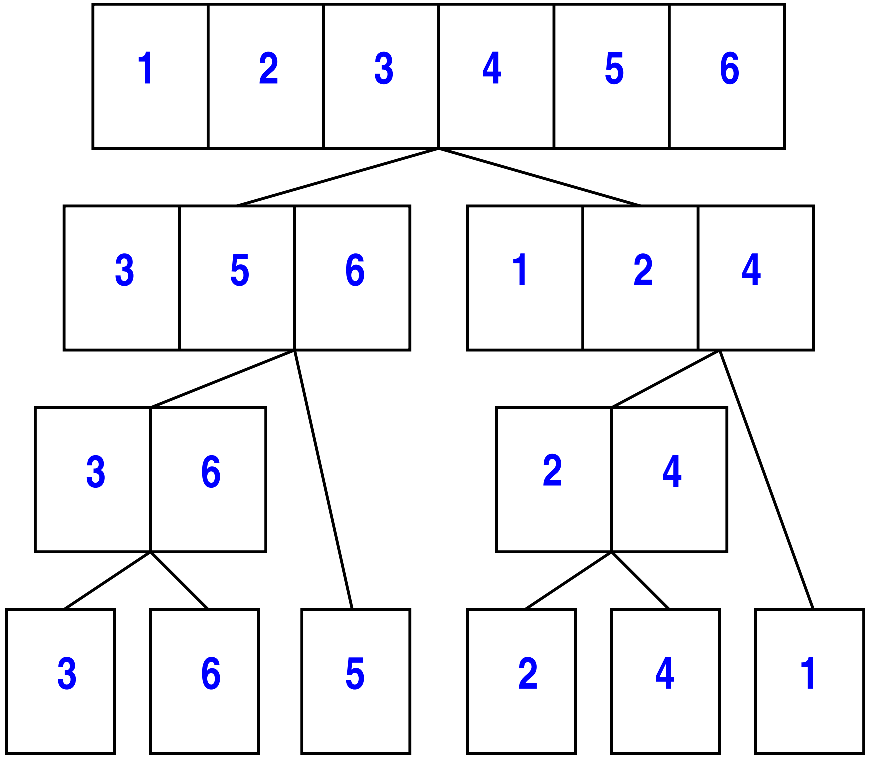
- What we have described is sometimes called Bottum-up MergeSort.
- There is also a Top-down version of MergeSort, using a divide and conquer strategy:
  - “split” the unsorted sequence into two halves
  - sort each half
  - merge the two halves
- it employs a recursive algorithm as well as the merge function to accomplish the sorting

Top-Down MergeSort

```
void MergeSort( int a[], int n, int c[] )
{
   if( n > 1 ) {
        int m = n/2;
        MergeSort(  a,     m,    c );    // sort 1st half
        MergeSort( &a[m], n-m, c );    // sort 2nd half

        merge( a, m, &a[m], n-m, c );
        copy(  a, n, c );
   }
}
```

Top-Down MergeSort



Top-Down MergeSort – Analysis

- analysing the time complexity of recursive algorithms is more difficult than for iterative algorithms
- it usually involves solving recurrence equations
- using this technique it can be shown that top-down MergeSort also runs in time proportional to $n\log_2(n)$

Summary

- Sorting data is a commonly performed task particularly when dealing with large amounts of data
- Carefully designed algorithms can greatly improve efficiency
- Efficiency can often be estimated by theoretical analysis before the algorithm has even been implemented.

