Massey University
COLLEGE OF SCIENCES

159201

Week 7

Summer 2014

Te Kunenga
ki Pūrehuroa

Massey University
COLLEGE OF SCIENCES

L 25

Te Kunenga
ki Pūrehuroa

Comparing Algorithms

There are often two (or more) algorithms for the same problem. When we write a program to solve the problem, which algorithm should we use?

It would be good to compare the algorithms before writing the program(s) as writing code requires time and money.

One way (there are others) to compare algorithms is to count operations. An operation is anything that consumes time on a computer, e.g. multiplication, function call, etc.

Comparing Algorithms

Example:

Compare Linear Search and Binary Search for searching through a list of names.

Linear Search : start at the beginning and look at each item until you find the one you want.

Binary Search : look in the middle, then split the list in half, then repeat.

Note that Binary Search requires the data to be ***sorted*** before running the search.

Comparing Algorithms

We will count comparisons as these are suitable operations for a search algorithm.

```
if (data[i] >= x) ...
```

this is an example of a comparison.

We always assume there are N items in the list.

Comparing Algorithms

Linear Search

Minimum number of comparisons = 1

Maximum number of comparisons = n

Binary Search

Each time we make a comparison we split the list into two parts:

Number of comparisons	0	1	2	3	...	k
Length of list	N	N/2	N/4	N/8	...	$N/2^k$

Comparing Algorithms

Binary Search

Number of comparisons	0	1	2	3	...	k
Length of list	N	N/2	N/4	N/8	...	$N/2^k$

$N / 2^k = 1$ in the worst case scenario

Therefore, $N = 2^k$

$\ln(N) = \ln(2^k)$

$\ln(N) = K \ln(2)$

$K = \ln(N)/\ln(2)$

Comparing Algorithms

The problem is to compare worst case scenarios, best case scenarios, run-time, memory occupancy... which criteria should we use?

Answer: algorithm complexity

There is a way to represent what trend **dominates** a certain algorithm.

We cannot be certain about the run-time when we run the algorithm once. But we know that statistically, after lots of runs, it will (asymptotically) tend to a value.

Search example:

Suppose we want to search Auckland's telephone directory: N=450000

The maximum number of comparisons:

Linear search: 450000

Binary search: ln(450000)/ln(2)=18.78

Binary search is a lot faster...in the worst case scenario.

This is a somewhat simplistic analysis, as this assumes that we only find the name last on the list. We also made assumptions (i.e., list is **sorted**)

Big O notation (omicron)

Simplifying a formula, we can tell what portion dominates for large N. We can say that this algorithm is of the order of f(N). We are not so interested in the coefficients of f(N), nor on the complete f(N), only the largest factor. We write that as the *Big O notation*, or O(f(N)).

O() describes the upper bound of the function (worst case scenario). There are other notations such as Θ, Ω, ω, o etc, for other bounds.

Example: suppose an algorithm runtime is written as $N^2+3N+62$.

Massey University
COLLEGE OF SCIENCES

Big O notation

$T=N^2+3N+62$

For N=10, T=192 (or N^2 + 92%)

For N=100, T=10362 (or N^2 + 4%)

For N=1000, T=1003062 (or N^2 + 0.3%)

We can say that this algorithm is of the order of N^2, or that it has $O(N^2)$ ***time complexity***.

We can state that:

Linear Search is O(N)

Binary Search is O(ln(N))

(note the lack of constants such as ln(2))

Te Kunenga ki Pūrehuroa

Big O notation

Common time complexity algorithms:

O(1) constant time

O(N) linear time

$O(N^2)$ quadratic time

$O(N^3)$ cubic time

$O(N^k)$ polynomial time

$O(K^N)$ exponential time

Massey University
COLLEGE OF SCIENCES

L 26

Te Kunenga
ki Pūrehuroa

Sorting

Sorting is a very important area of computing.

Among other things, Sorting is used to assist searching.

There are a number of sorting algorithms and we want to compare them.

We assume an array (or vector) of N integers that we will sort into **ascending** (not **descending**) order.

Let's take a look at **Selection** sort, **Insertion** sort, **Bubble** sort, **Merge** sort, **Quick** sort, **Radix** sort and **Heap** sort.

Selection Sort

Find the smallest number and swap it with the front.

Make the front advance one position and repeat.

Example: suppose we want to sort 12, 4, 3, 9, 1:

					front	minimum
12	4	3	9	**1**	**12**	**1**
1	**4**	3	9	12	**4**	**3**
1	3	**4**	9	12	**4**	**4**
1	3	4	**9**	12	**9**	**9**

Selection Sort

A code snippet:

```
for (pass = 0; pass < n – 1; pass++) {
  min = pass;  // min is an index
  for (i = pass + 1; i < n; i++) {
    if (data[i] < data[min]) { min = i; } // a comparison here
  }
  swap(data[min], data[pass]);
}
```


Selection Sort

Note that to find the minimum we are re-examining the whole sequence after the front.

We can also say that $O(N^2)$ is the time complexity.

Usually for{for{}} implies in $O(N^2)$, but *not always.*

Advantages: very simple, works on arrays and linked-lists

Disadvantages: slow (there are better options

Careful to implement swap(), *by reference* not by value.

Massey University
COLLEGE OF SCIENCES

Insertion Sort

Read one number at a time. Copy the sequence using the right place, inserting one by one:

12	4	3	9	1

12

4	12

3	4	12

3	4	**9**	12

1	3	4	9	12

Te Kunenga ki Pūrehuroa

Insertion Sort

We can also use a single array and keep track of the sorted list and the unsorted list:

```
for (pass=0; pass < n – 1; pass++) {
  temp=data[pass+1];//note comparison in next line
  for (i=pass + 1; i > 0 && data[i-1] > temp; i--)
  { // not i++
    data[i] = data[i-1];  // shuffling
  }
  data[i] = temp;
}
```


Massey University
COLLEGE OF SCIENCES

Insertion Sort

Lets follow the effects of the previous snippet:

					temp
12	4	3	9	1	**4**
12	**12**	3	9	1	4
4	12	3	9	1	4

4	12	3	9	1	**3**
4	12	**12**	9	1	3
4	**4**	12	9	1	3
3	4	12	9	1	3

TIME

Te Kunenga ki Pūrehuroa

Massey University
COLLEGE OF SCIENCES

Insertion Sort

					temp
3	4	12	9	1	**9**
3	4	12	**12**	1	9
3	4	**9**	12	1	9
3	4	9	12	1	**1**
3	4	9	12	**12**	1
3	4	9	**9**	12	1
3	4	**4**	9	12	1
3	**3**	4	9	12	1
1	3	4	9	12	1

TIME

Te Kunenga ki Pūrehuroa

Insertion Sort

Note: the inner loop shuffles portions of the array that are larger than temp, to make space for it. We then copy temp to the space left (where there is a repetition of the value).

Advantages: works really well for linked-lists (no shuffle is required, we can insert between nodes)

Simple

Disadvantages: also slow $O(N^2)$

Here it is an interesting fact: although both have the same big O complexity, runtimes are actually different!

Challenge

1) Implement Insertion sort using linked-lists.

- Should you use AddNodeFront() or AddNodeRear()?
- Or other type of AddNode()?

2) What extra pointers could make it go faster? Does that change the complexity of the Insertion Sort itself?

Massey University
COLLEGE OF SCIENCES

L 27

Te Kunenga
ki Pūrehuroa

Bubble Sort

Move along the array examining pairs of elements, swapping items that are in the wrong order.

12	4	3	9	1

4	12	3	9	1

4	3	12	9	1

4	3	9	12	1

4	3	9	1	12

Bubble Sort

A code snippet is:

```
swapping = true;
while (swapping) {
   swapping = false;
    for (i = 0; i < n-1; i++) {//don't look at the last one
      if (data[i] > data[i+1]) {//comparison
         swap(data[i], data[i + 1]);
         swapping = true;
      }
   }
}
```


Bubble Sort

This represents one pass required, the array is still not sorted. The element 12 was dragged along to its position. How many passes will be required? How many swap() calls?

12	4	3	9	1
4	12	3	9	1
4	3	12	9	1
4	3	9	12	1
4	3	9	1	12

(swapping == true) for red

Bubble Sort

Continue to sort...so far we only managed to get 12 to the last position, go to the second pass.

4	3	9	1	12
3	4	9	1	12
3	4	9	1	12
3	4	1	9	12

3	4	1	9	12

Bubble Sort

Pass 3...

3	4	1	9	12

3	4	1	9	12

3	1	4	9	12

3	1	4	9	12

3	1	4	9	12

Bubble Sort

Pass 4... it is sorted already, but we have another pass.

3	1	4	9	12
1	3	4	9	12
1	3	4	9	12
1	3	4	9	12

1	3	4	9	12

Bubble Sort

Pass 5...we make a final pass, so `swapping==false`

1	3	4	9	12

1	3	4	9	12

1	3	4	9	12

1	3	4	9	12

1	3	4	9	12

Notes on Bubble Sort

The outer loop performs as many passes as required (+1), in the worst case scenario N times.

The inner loop runs N-1 times.

So we still have $O(N^2)$ complexity...

Advantages: very simple, stops when finished (so the outer loop may run less than N times)

Disadvantage: as slow as the previous sorting algorithms

Massey University
COLLEGE OF SCIENCES

L 28

Te Kunenga
ki Pūrehuroa

Merge Sort

Merging: process of making a single sorted list C out of two smaller sorted lists A and B.

1) Look at the first item in A and the first item in B
2) Choose the smallest and add to C
3) Move to the next item on the list where the smallest was chosen
4) repeat

Merge Sort

Example:

List A: {3,4,12}

List B: {1,10,23}

Merging the lists into C:

List C: {1,3,4,10,12,23}

We assumed that the lists A and B are sorted already, let's look at the Merge function...

Merge(int,int)

```
void Merge(int first, int last) {
   int mid = (first + last)/2;
   int i1 = first – 1;
   int i2 = first;  int i3 = mid + 1;
   while ((i2 <= mid) && (i3 <= last)) {
     if (data[i2] < data[i3]) {
        i1++;  temp[i1] = data[i2];  i2++;
     } else {
        i1++;  temp[i1] = data[i3];  i3++;
     }
   }   // may still be items in the first section
   while (i2 <= mid) {
     i1++;  temp[i1] = data[i2];  i2++;
   }   // may still be items in the second section
   while (i3 <= last) {
     i1++;  temp[i1] = data[i3];  i3++;
   }   // copy from temp back to data
   for (i = first; i <= last; i++) { data[i] = temp[i]; }
}
```


Recursive Merge Sort

We still didn't sort the lists, we can use merge() in the following sequence:

1) Split the array into two sections
2) Sort the first section
3) Sort the second section
4) Merge the two sections into a sorted array

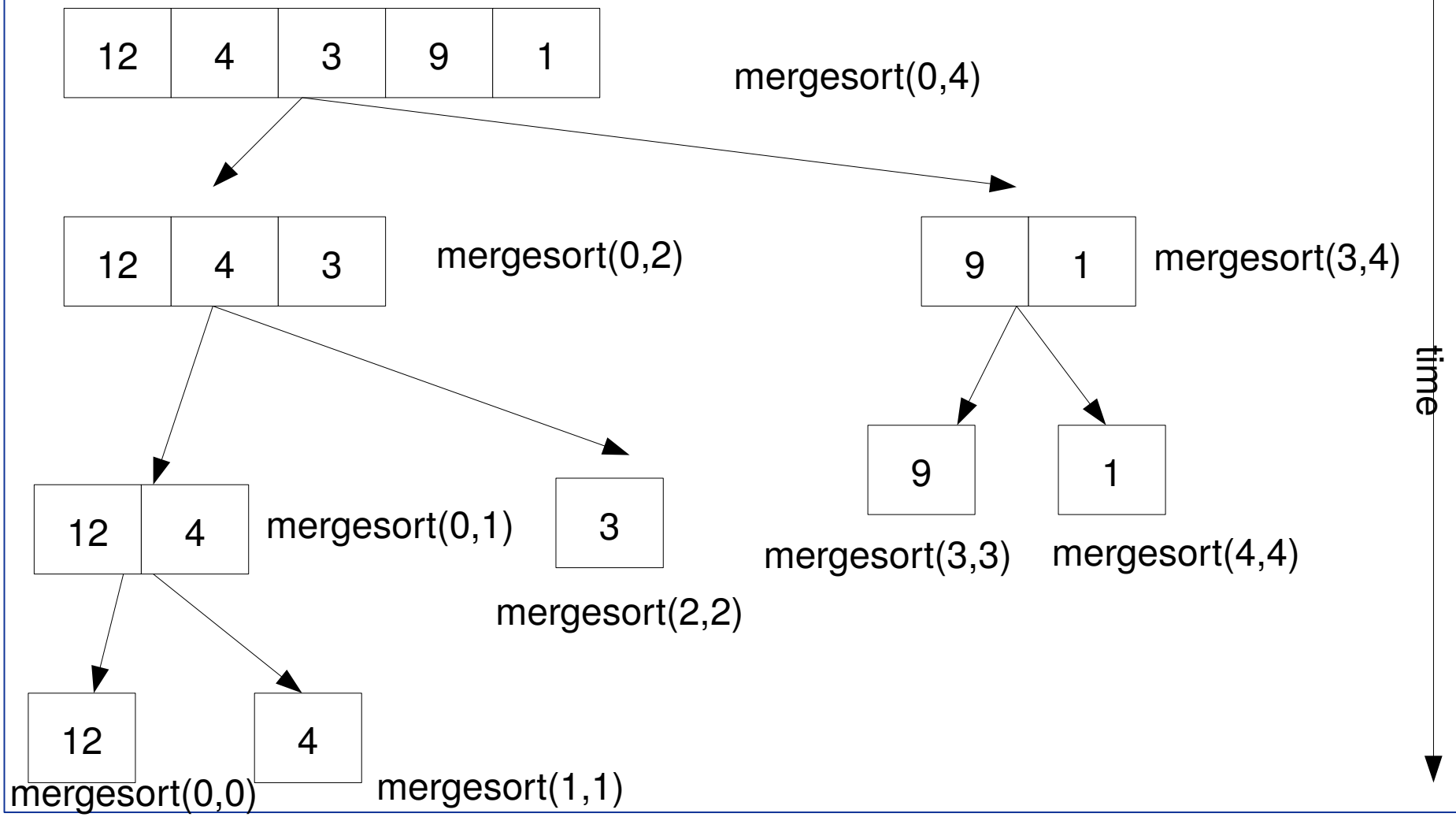
The process requires that we split the list into smaller chunks until it is small enough to sort by swapping, then we merge the lists back until we get all elements into a single list.

MergeSort(int,int)

```
void MergeSort(int first, int last) {
    int mid = (first + last)/2;
    if (first < last) {
        MergeSort(first, mid);
        MergeSort(mid + 1, last);
        Merge(first, last);
    }
}
```

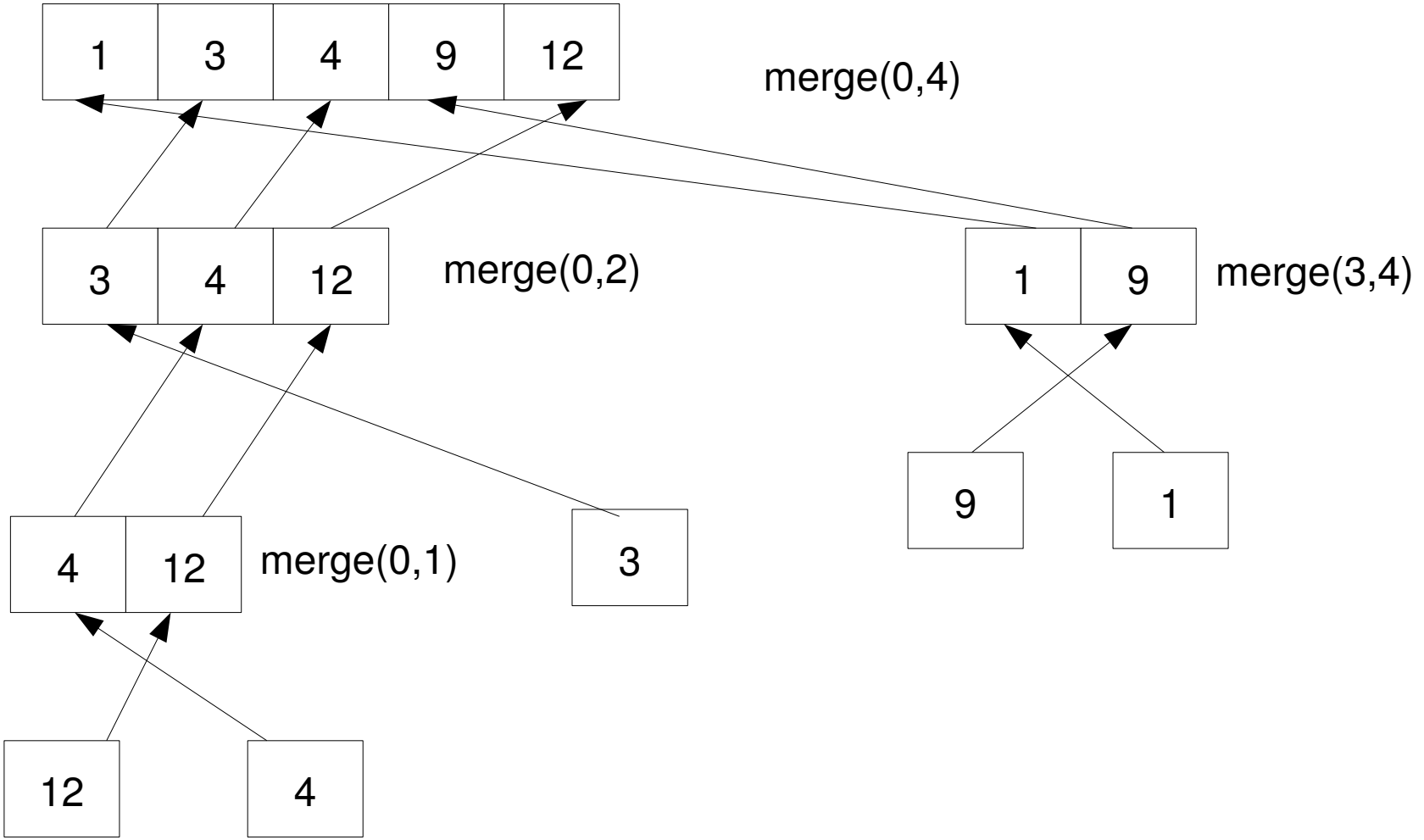

Merge Sort

Example: suppose we want to sort 12, 4, 3, 9, 1 again



Merge Sort

Example: merging back...



Complexity O() for merge sort

Each Merge() is of order O(N).

Each MergeSort() is of order O(ln(N)).

The whole operation is **O(N ln(N))**. This is slightly better than $O(N^2)$, so merge sort can solve much larger problems than the previous sort algorithms.

The disadvantage of merge sort is that we need extra space for the recursive calls. Also there are extra copies to be made, so extra time for the copying operations.

Quicksort

Quicksort (one word):

Select one of the values, which we call the *pivot*.

Divide the list into two, such that the first part contains all values less than the pivot, and the second part contains all values greater than the pivot.

Sort the two parts separately and join the parts.

No merging is required! But what is the time complexity?

Quicksort function (simplified)

```
void Quicksort(int first, int last) {
 int i = first+1, j = last, pivot = data[first];
 while (i < j) {
   while ( (data[i] < pivot) && (i<last) ) { i++;}
   while ( (data[j] > pivot) ){ j--; }
   if (i < j) {swap(&data[i], &data[j]); }
 }
 if(pivot>data[j]){swap(&data[first], &data[j]);}
 if(first < j-1) { Quicksort(first, j-1); }
 if(j+1 < last) { Quicksort(j+1, last); }
}
```


Quicksort

Note that we also call Quicksort recursively.

There are other ways to write Quicksort... in the web there are lots of examples.

Some implementations use more memory and run (on average) a bit faster.

Let's see how the previous code works...

Quicksort

Example: choose a pivot, say, 26 (the first element).

26	5	37	1	61	11	59	48	19

Find the first number larger than the pivot, 37

26	5	**37**	1	61	11	59	48	19

Search up

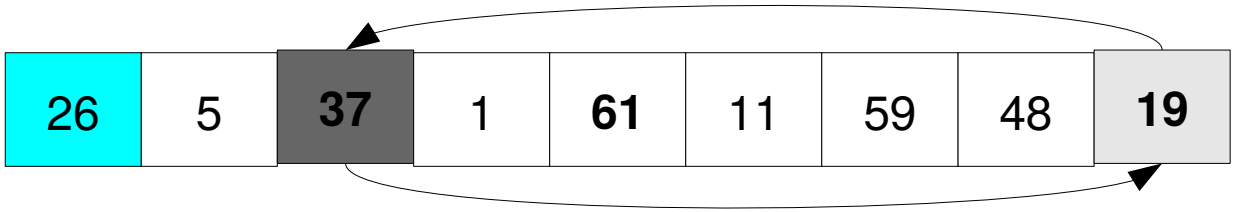
Find the first number smallest than the pivot, 19

26	5	37	1	61	11	59	48	**19**

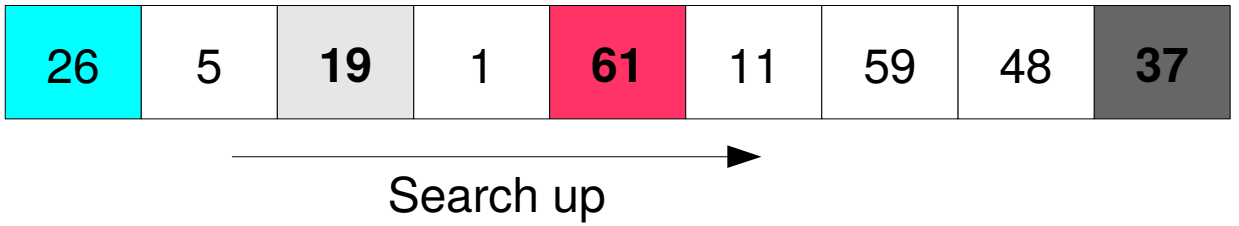
Search down

Quicksort

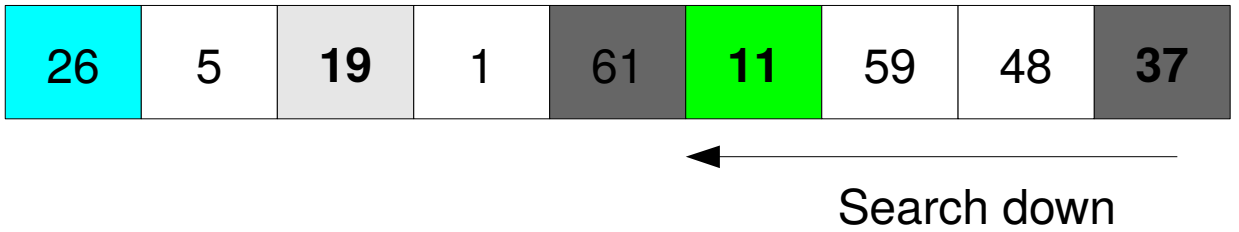
Swap 37 <-> 19 and continue to search from both sides until we meet at the middle



Find the next number larger than the pivot, 61

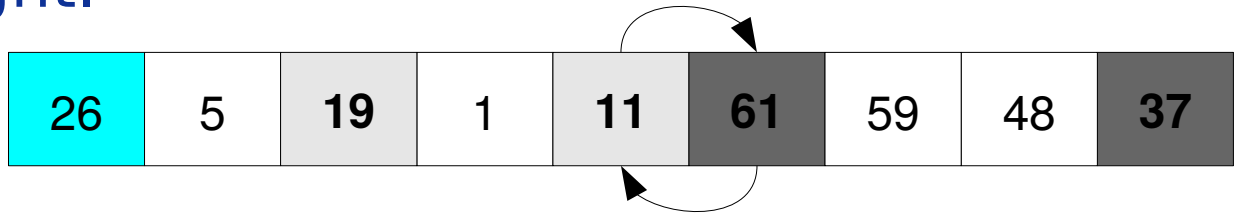


Find the first number smallest than the pivot, 11

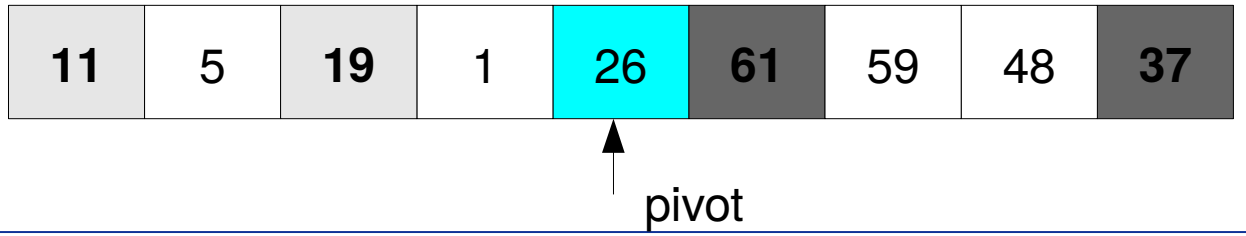


Quicksort

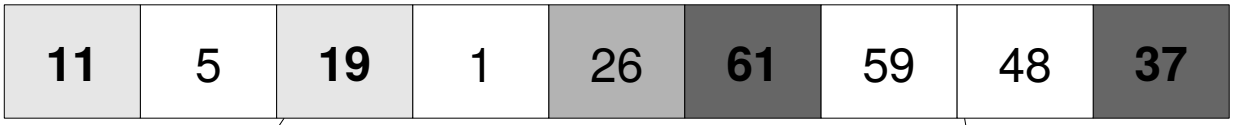
Swap 61 <-> 11. We now have two halfs of the array after 26, four elements <26 to the left and four >26 to the right.



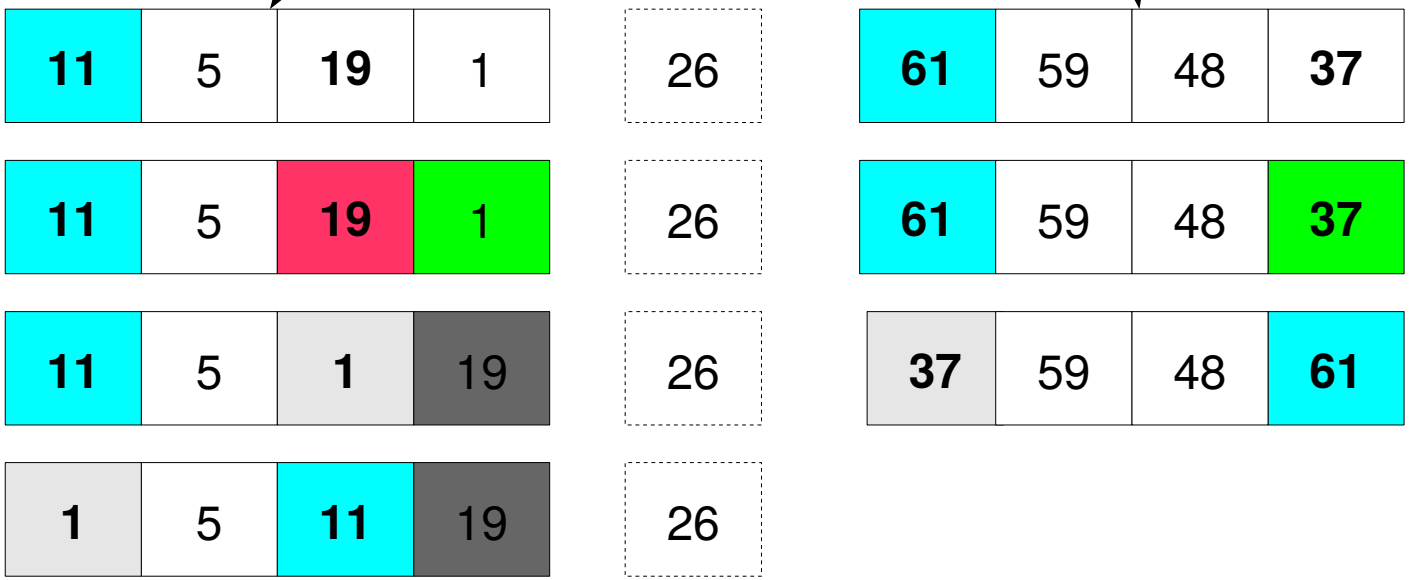
We move the pivot in between the two sub-arrays, i.e., swap 11<->26:



Quicksort

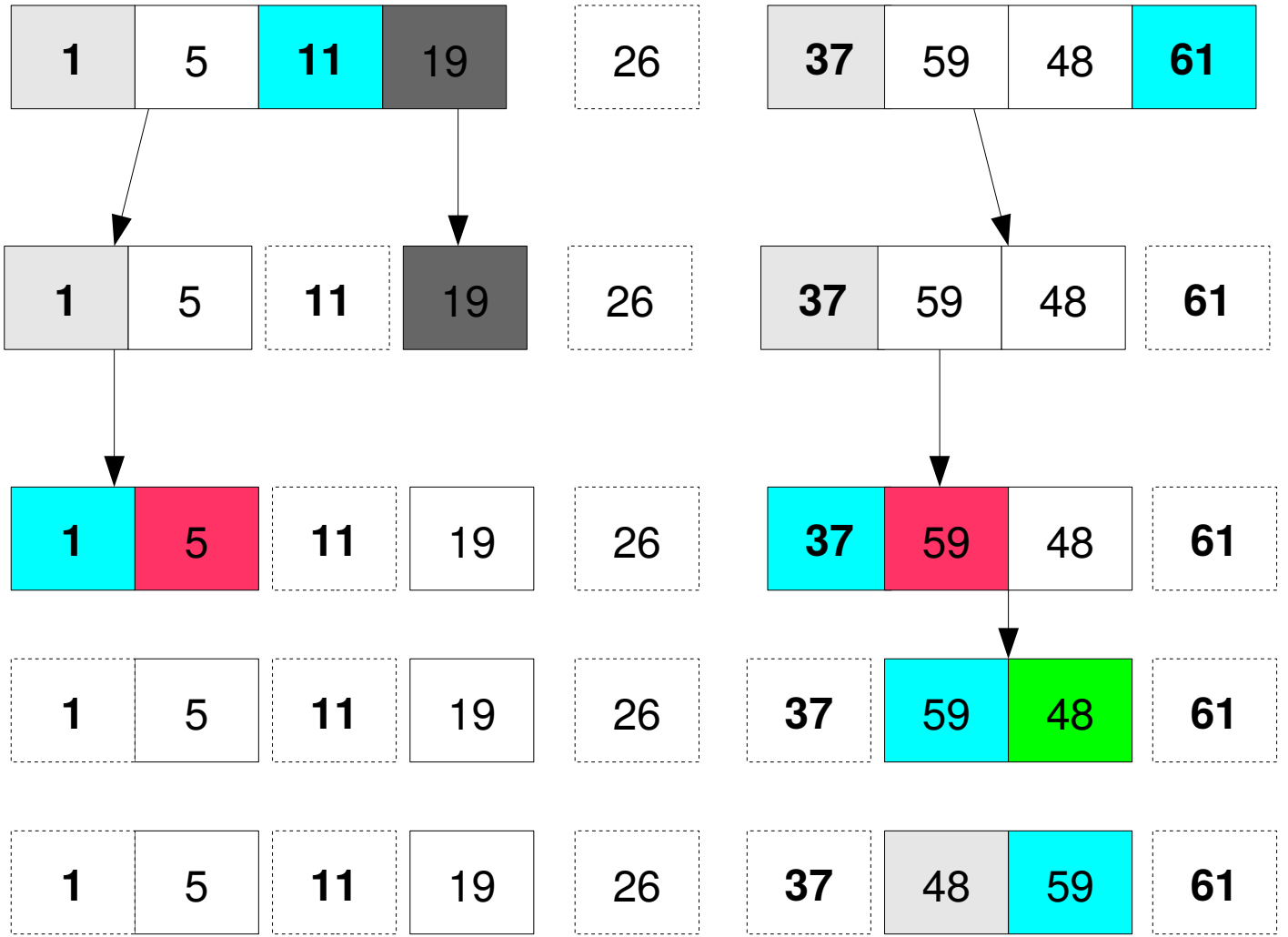


We sort the parts separately, repeating the process



Massey University
COLLEGE OF SCIENCES

Quicksort



Te Kunenga ki Pūrehuroa

Quicksort

One way to describe Quicksort is based on the ***pivot***.

If we always choose the first element as the pivot, it will move to its correct position in the array.

So, after recursive calls, the array is sorted.

How many comparisons?

Unfortunately the answer depends on the choice of the pivot... and some other subtle details.

On average, O(N log(N)).

Worst case (rare), $O(N^2)$.

In practise, Quicksort is faster than any other O(N log(N)) algorithms ***on average.***

Quicksort

How many comparisons in our code?

Lets assume that the pivot is always the *median* value.

We do N-1 comparisons, divide the problem into two N/2

The number of comparisons would approximate to:

$N+2N/2+4N/4+8N/8+...+NN/N=N(\log(N)/\log(2)+1)$

Which is O(N log(N))

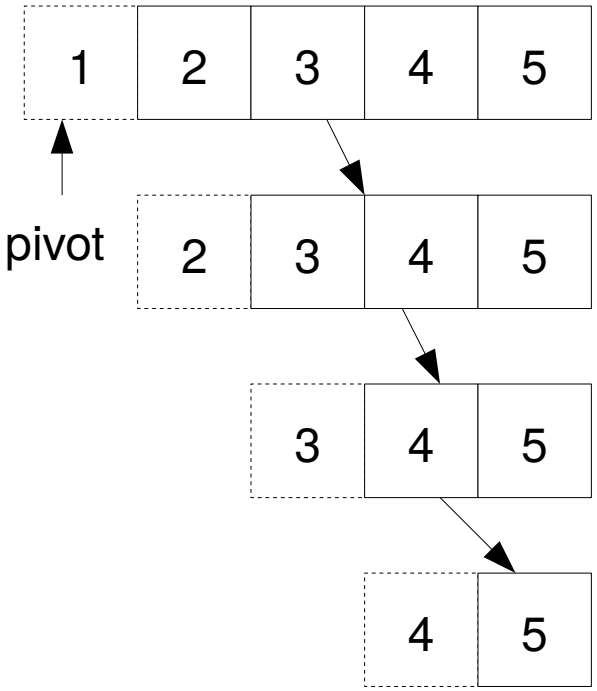
Or

$N-1+2(N-3)/2+4(N-5)/4+8(N-9)/8...+N(N-N+1)/N=N(\log(N))$

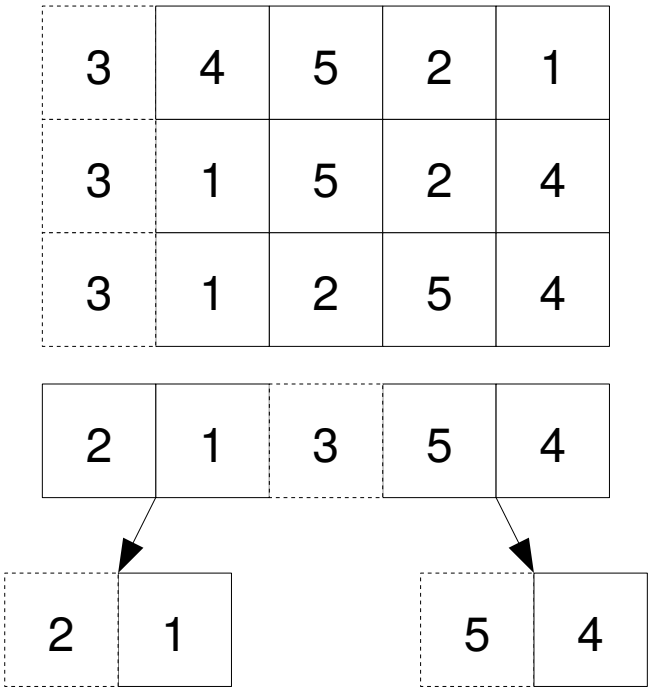
Quicksort

How many recursive calls?

Depends on the data's original order (assume pivot=first)



Case 1



Case 2

Quicksort

Advantages: pretty fast (hence Quicksort...),
O(N log(N))

Disadvantages:

Very sensitive code

For example, a bad choice of pivot can cause long runtime

Difficult to estimate complexity

Discussion

Why *MergeSort* and *Quicksort* are faster than the other algorithms?

Divide-and-conquer approach.

Both algorithms divide the problem into smaller problems until they are small enough to solve, then we join the solutions. Usually this approach leads to a logarithmic function.

There is also time cost N associated with the join operation, so O(N log(N)).

Challenge

1) Implement a program that sorts an array containing a number of integers. Two sorting algorithms should be used, Insertion and Bubble.

2) Add a counter (what C++ type would be adequate?) to count the *number of cycles* (closely related to the number of comparisons)

3) Compare the number of cycles when using three different arrays with 1000 numbers:

   a) In order array: 1,2,3...1000

   b) reversed order array: 1000, 999, 998... 1

   c) random order array

Analyse the different counts for each run.