ECS510

Algorithms and Data Structures in an Object Oriented Framework

"ADSOOF"

Sorting and Efficiency

Constructive Insertion Sort of ArrayList

```
public static ArrayList<Integer> sort(ArrayList<Integer> a)
{
 ArrayList<Integer> b = new ArrayList<Integer>();
 for(int i=0; i<a.size(); i++)
    insert(a.get(i),b);
 return b;
}

private static void insert(Integer n,ArrayList<Integer> b)
{
 int i;
 for(i=0; i<b.size()&&b.get(i).compareTo(n)<0; i++) {}
 b.add(i,n);
}
```

Same algorithm

```
public static ArrayList<Integer> sort(ArrayList<Integer> a)
{
 ArrayList<Integer> b = new ArrayList<Integer>();
 for(int i=0; i<a.size(); i++)
    {
     int n = a.get(i);
     int j;
     for(j=0; j<b.size()&&b.get(j).compareTo(n)<0; j++) {}
     b.add(j,n);
    }
 return b;
}
```

When asked to write “a method” for a problem, it’s always acceptable and often leads to easier to understand code to write one which uses a separate helper method

Insertion Sort Algorithm

- To sort ArrayList `a`, set up new empty ArrayList `b`
- Then go through each item of `a` and insert into its ordered position in `b`
- To insert `n` into `b`, go through each position in `b`, starting at 0, until you have a position where the item is greater than `n`, or you have reached the end
- Insert `n` into that position (following items will be moved up one place)
- Loop invariant for `sort` is that `b` has the first `i` items of `a` in order

```
a [30,25,47,99,8,12,28,63,16,20]
    ^

i=0

b []
```

```
a [30,25,47,99,8,12,28,63,16,20]
       ^

i=1

b [30]
```

```
a [30,25,47,99,8,12,28,63,16,20]
           ^

i=2

b [25,30]
```

```
a [30,25,47,99,8,12,28,63,16,20]
                ^

i=3

b [25,30,47]
```

a [30,25,47,99,8,12,28,63,16,20]

^

i=4

b [25,30,47,99]

```
a [30,25,47,99,8,12,28,63,16,20]
                        ^

i=5

b [8,25,30,47,99]
```

```
a [30,25,47,99,8,12,28,63,16,20]
                              ^

i=6

b [8,12,25,30,47,99]
```

```
a [30,25,47,99,8,12,28,63,16,20]
                              ^

i=7

b [8,12,25,28,30,47,99]
```

```
a [30,25,47,99,8,12,28,63,16,20]
                                ^

i=8

b [8,12,25,28,30,47,63,99]
```

```
a [30,25,47,99,8,12,28,63,16,20]
                                ^

i=9

b [8,12,16,25,28,30,47,63,99]
```

a [30,25,47,99,8,12,28,63,16,20]

i=10

b [8,12,16,20,25,28,30,47,63,99]

Insertion Sort on ArrayList “In Place”

```
public static void sort(ArrayList<Integer> a)
{
 for(int i=1; i<a.size(); i++)
    insert(a.get(i),a,i);
}

private static void insert(Integer n,ArrayList<Integer> a,int i)
{
 for(; i>0&&a.get(i-1).compareTo(n)>0; i--)
    a.set(i,a.get(i-1));
 a.set(i,n);
}
```

Sorting in place

- There is only one collection
- Sorting is done by moving items in the collection
- Here `insert(a.get(i),a,i);` has the effect "move items from position `i-1` up one place, until the correct position for the item which was at position `i` is found, then put that item at that position"
- As it does not rely on expanding the ArrayList, it works on arrays as well

Insertion sort on arrays in place

```
public static void sort(Integer[] a)
{
 for(int i=1; i<a.length; i++)
    insert(a[i],a,i);
}

private static void insert(Integer n, Integer[] a, int i)
{
 for(; i>0&&a[i-1].compareTo(n)>0; i--)
    a[i]=a[i-1];
 a[i]=n;
}
```

Loop invariant for sort

```
public static void sort(Integer[] a)
{
 for(int i=1; i<a.length; i++)
    insert(a[i],a,i);
}
```

Each time round the loop, the array `a` has the first `i` items from the array as it was initially but rearranged in order

[30|25,47,99,8,12,28,63,16,20]

i=1

[25,30|47,99,8,12,28,63,16,20]

i=2

[25,30,47|99,8,12,28,63,16,20]

i=3

[25,30,47,99|8,12,28,63,16,20]

i=4

[8,25,30,47,99|12,28,63,16,20]

i=5

[8,12,25,30,47,99|28,63,16,20]

i=6

[8,12,25,28,30,47,99|63,16,20]

i=7

[8,12,25,28,30,47,63,99|16,20]

i=8

[8,12,16,25,28,30,47,63,99|20]

i=9

[8,12,16,20,25,28,30,47,63,99]

i=10

Inserting

Environment for `sort`

`i=6`

`a=[8,12,25,30,47,99|28,63,16,20]`

Environment for `insert`

`n=28`

`a=` *alias to above*

`i=6`

```
[8,12,25,30,47,99,28,63,16,20]
                          ^
```

before `a[i]=a[i-1]`

`n=28`

`i=6`

```
[8,12,25,30,47,99,99,63,16,20]
                          ^
```

after `a[i]=a[i-1]`

`n=28`

`i=6`

```
[8,12,25,30,47,99,99,63,16,20]
                    ^
```

before `a[i]=a[i-1]`

`n=28`

`i=5`

```
[8,12,25,30,47,47,99,63,16,20]
                    ^
```

after `a[i]=a[i-1]`

`n=28`

`i=5`

```
[8,12,25,30,47,47,99,63,16,20]
                  ^
```

before `a[i]=a[i-1]`

`n=28`

`i=4`

```
[8,12,25,30,30,47,99,63,16,20]
                ^
```

after `a[i]=a[i-1]`

`n=28`

`i=4`

```
[8,12,25,30,30,47,99,63,16,20]
           ^
```

At end of loop

```
n=28
i=3
```

```
[8,12,25,28,30,47,99,63,16,20]
           ^
```

After `a[i]=n`

`n=28`

`i=3`

Selection Sort Algorithm

Another sort in place algorithm

- Find lowest item, swap with item at position 0
- Find second lowest item, swap with item at position 1
- Find third lowest item, swap with item at position 2

…

- Find $n\text{-}1^{th}$ lowest item, swap with item at position n-2 (n items in total – this deals with last two)

Selection Sort Code

```
public static void sort(Integer[] a)
 {
  for(int i=0; i<a.length-1; i++)
     {
      int pos = findMinPos(a,i);
      Integer temp = a[pos];
      a[pos]=a[i];
      a[i]=temp;
     }
 }

 private static int findMinPos(Integer[] a,int pos)
 {
  for(int i=pos+1; i<a.length; i++)
     if(a[i].compareTo(a[pos])<0)
        pos=i;
  return pos;
 }
```

```
[30,25,47,99,8,12,28,63,16,20]
 ^                 ^

i=0

pos=4
```

```
[8|25,47,99,30,12,28,63,16,20]
 ^                 ^

i=0

pos=4
```

```
[8|25,47,99,30,12,28,63,16,20]
    ^                   ^

i=1

pos=5
```

```
[8,12|47,99,30,25,28,63,16,20]
    ^                   ^

i=1

pos=5
```

```
[8,12|47,99,30,25,28,63,16,20]
        ^                     ^

i=2

pos=8
```

```
[8,12,16|99,30,25,28,63,47,20]
         ^                    ^

i=2

pos=8
```

```
[8,12,16|99,30,25,28,63,47,20]
         ^                    ^

i=3

pos=9
```

```
[8,12,16,20|30,25,28,63,47,99]
           ^                  ^

i=3

pos=9
```

```
[8,12,16,20|30,25,28,63,47,99]
           ^  ^

i=4

pos=5
```

```
[8,12,16,20,25|30,28,63,47,99]
              ^   ^

i=4

pos=5
```

```
[8,12,16,20,25|30,28,63,47,99]
                ^   ^

i=5

pos=6
```

```
[8,12,16,20,25,28|30,63,47,99]
                  ^   ^

i=5

pos=6
```

```
[8,12,16,20,25,28|30,63,47,99]
                  ^^

i=6

pos=6
```

```
[8,12,16,20,25,28,30|63,47,99]
                    ^^

i=6

pos=6
```

```
[8,12,16,20,25,28,30|63,47,99]
                     ^  ^

i=7

pos=8
```

```
[8,12,16,20,25,28,30,47|63,99]
                       ^  ^

i=7

pos=8
```

```
[8,12,16,20,25,28,30,47|63,99]
                        ^^

i=8

pos=8
```

[8,12,16,20,25,28,30,47,63,99]

i=9

Quicksort

- Choose pivot (simple version:first item)
- Divide rest of collection into two parts:
  - Items smaller than pivot
  - Items greater than or equal to pivot
- Sort each part
- Join two parts together with pivot in middle

Constructive Quicksort

```
public static ArrayList<Integer> sort(ArrayList<Integer> a) {
 if(a.size()<=0)
    return a;
 ArrayList<Integer> smaller = new ArrayList<Integer>();
 ArrayList<Integer> greater = new ArrayList<Integer>();
 Integer pivot = a.get(0);
 for(int i=1; i<a.size(); i++) {
     Integer n=a.get(i);
     if(n.compareTo(pivot)<0)
        smaller.add(n);
     else
        greater.add(n);
    }
 smaller=sort(smaller);
 greater=sort(greater);
 smaller.add(pivot);
 smaller.addAll(greater);
 return smaller;
}
```

```
[30,25,47,99,8,12,28,63,16,20]

pivot=30
smaller=[]
greater=[]
```

```
[30,25,47,99,8,12,28,63,16,20]

pivot=30
smaller=[25,8,12,28,16,20]
greater=[47,99,63]
```

```
[30,25,47,99,8,12,28,63,16,20]

pivot=30
smaller=[8,12,16,20,25,28]
greater=[47,99,63]
```

```
[30,25,47,99,8,12,28,63,16,20]

pivot=30
smaller=[8,12,16,20,25,28]
greater=[47,63,99]
```

```
[30,25,47,99,8,12,28,63,16,20]

pivot=30
smaller=[8,12,16,20,25,28,30]
greater=[47,63,99]
```

```
[30,25,47,99,8,12,28,63,16,20]

pivot=30
smaller=
[8,12,16,20,25,28,30,47,63,99]
greater=[47,63,99]
```

```
[30,25,47,99,8,12,28,63,16,20]
```

```
return
[8,12,16,20,25,28,30,47,63,99]
```

An issue with constructive quicksort of arrays

- We do not know the size of the two parts in advance
- With ArrayLists it did not matter, as we created two empty ArrayLists for the two parts and added elements as appropriate to one or the other
- Arrays, however are of fixed size
- So do we have to go through the array checking each element twice the first time to count the number of elements greater or less than the pivot, the second time to actually put the elements into the new arrays created after the counting has been done?

```
…
Integer pivot = a[0];
int count=0;
for(int i=0; i<a.length; i++)
    if(a[i].compareTo(pivot)<0)
       count++;
Integer[] smaller = new Integer[count];
Integer[] greater = new Integer[a.length-count-1];
for(int i=1, j=0, k=0; i<a.length; i++)
   if(a[i].compareTo(pivot)<0)
       smaller[j++]=a[i];
   else
       greater[k++]=a[i];
…
```

- We can avoid doing this, see next slide:

```
…
 Integer pivot = a[0];
 boolean[] greaterthan = new boolean[a.length];
 int count=0;
 for(int i=0; i<a.length; i++)
    if(a[i].compareTo(pivot)<0)
       count++;
    else
       greaterthan[i]=true;
 Integer[] smaller = new Integer[count];
 Integer[] greater = new Integer[a.length-count-1];
 for(int i=1, j=0, k=0; i<a.length; i++)
    if(greaterthan[i])
       greater[j++]=a[i];
    else
       smaller[k++]=a[i];
…
```

What are ArrayLists?

- Remember that ArrayLists are just objects with an array and count inside, that is how the change of size is managed
- Consider from this what is actually happening underneath in the code:

```
ArrayList<Integer> smaller = new ArrayList<Integer>();
ArrayList<Integer> greater = new ArrayList<Integer>();
Integer pivot = a.get(0);
for(int i=1; i<a.size(); i++) {
   Integer n=a.get(i);
   if(n.compareTo(pivot)<0)
      smaller.add(n);
   else
      greater.add(n);
  }
```

- It would be very inefficient if every call of `add(n)` involved creating a new array underneath
- Using an abstract data type hides some of the complexity of code, but when considering efficiency we need to know what happens underneath

Quicksort in place

- Instead of constructing two new arrays for smaller and greater items, swap items around in array until first part is smaller, second part is greater
- Then sort two parts – sort method has parameters saying which part of array is being sorted

Code for Quicksort in place

- The recursive code takes the start and finish positions of the portion of the array each call is sorting as its arguments, along with the array itself.
- So each call has its own `from` and `to` arguments, and a variable which aliases the one array
- The `from` and `to` arguments are initially set to `0` and the length of the array, this must be done in the `public` method which calls the `private` recursive method which takes these extra arguments:

```
public static void sort(Integer[] a)
{
 sort(a,0,a.length);
}
```

```
private static void sort(Integer[] a, int from, int to)
{
 if(to>from+1) {
     Integer pivot = a[from];
     int low=from+1,high=to-1;
     while(low<high) {
         while(low<high&&a[low].compareTo(pivot)<0)
             low++;
         while(pivot.compareTo(a[high])<0)
             high--;
         if(low<high) {
            swap(a,high,low);
            low++;
            high--;
           }
        }
     while(pivot.compareTo(a[high])<0)
        high--;
     swap(a,from,high);
     sort(a,from,high);
     sort(a,high+1,to);
    }
}
```

Helper Methods

```
private static void swap(Integer[] a, int pos1, int pos2)
{
 Integer temp = a[pos1];
 a[pos1]=a[pos2];
 a[pos2]=temp;
}
```

- Always provide only the `public` methods you were asked to provide, with only the parameters that were asked for
- One method call should do all the work asked for
- That method call may make further method calls, to other methods to perform subtasks (such as `swap` here), they are "helper" methods if they are defined just for that purpose
- Always make helper methods `private`
- Making a helper method `public` means any other code could use it, which in some cases could cause problems

```
[30,25,47,99,8,12,28,63,16,20]
```

Sort portion of array
from position 0
to position 9

[30,25,47,99,8,12,28,63,16,20]

pivot=30

```
[30,25,47,99,8,12,28,63,16,20]
          ^                                ^


pivot=30
```

```
[30,25,20,99,8,12,28,63,16,47]
          ^                                ^


pivot=30
```

```
[30,25,20,99,8,12,28,63,16,47]
              ^                 ^

pivot=30
```

```
[30,25,20,16,8,12,28,63,99,47]
              ^                 ^

pivot=30
```

```
[30,25,20,16,8,12,28,63,99,47]
                          ^^

pivot=30
```

```
[28,25,20,16,8,12|30|63,99,47]
                  ^

pivot=30
```

[28,25,20,16,8,12|30|63,99,47]

Sort this part

from position 0

to position 5

[8,12,16,20,25,28|30|63,99,47]

Sorted

[8,12,16,20,25,28|30|63,99,47]

Sort this part
from pos 7
to pos 9

[8,12,16,20,25,28|30|47,63,99]

Sorted

```
[8,12,16,20,25,28,30,47,63,99]
```

Sorted

Recursion expressed using Iteration

- A recursive algorithm can be expressed using iteration if the implicit stack of environments produced by recursive calls is represented by an explicit stack of values
- There is no real reason for doing this, the idea that recursion is always less efficient is false
- Looking at how it works, however, may help you understand how recursion works underneath
- To express quicksort purely iteratively, there would have to be a stack of start and finish positions

```
public static void sort(Integer[] a) {
  ArrayList<Pair> stack = new ArrayList<Pair>();
  stack.add(new Pair(0,a.length));
  while(stack.size()>0)
     {
      Pair p=stack.remove(stack.size()-1);
      int from=p.from;
      int to=p.to;
      if(to>from+1) {
          Integer pivot = a[from];
          int low=from+1,high=to-1;
          while(low<high) {
              while(low<high&&a[low].compareTo(pivot)<0)
                  low++;
              while(pivot.compareTo(a[high])<0)
                  high--;
              if(low<high) {
                 swap(a,high,low);
                 low++;
                 high--;
                }
             }
          while(pivot.compareTo(a[high])<0)
             high--;
          swap(a,from,high);
          stack.add(new Pair(from,high));
          stack.add(new Pair(high+1,to));
         }
     }
 }
```

Nested Classes

- A nested class is the class equivalent of a helper method
- It is a class declared inside another class
- It can be used if you want a collection of objects which just store data values
- The nested class `Pair` just stores two `int` values:

```
private static class Pair
{
 int from, to;

 Pair(int f,int t)
 {
  from=f;
  to=t;
 }
}
```

- Do not use multi-dimensional arrays when a collection of objects like this is what you really need

Merge Sort

- Divide collection into two, arbitrarily
- Sort two parts
- Merge two parts together

[30,25,47,99,8,12,28,63,16,20]

half1=[30,25,47,99,8]

half2=[12,28,63,16,20]

[30,25,47,99,8,12,28,63,16,20]

half1=[8,25,30,47,99]

half2=[12,28,63,16,20]

[30,25,47,99,8,12,28,63,16,20]

half1=[8,25,30,47,99]

half2=[12,16,20,28,63]

```
[_,_,_,_,_,_,_,_,_,_]
i=0
half1=[8,25,30,47,99]
j=0
half2=[12,16,20,28,63]
k=0
```

```
[8|_,_,_,_,_,_,_,_,_]
i=1
half1=[8|25,30,47,99]
j=1
half2=[12,16,20,28,63]
k=0
```

```
[8,12|_,_,_,_,_,_,_,_,_]
i=2
half1=[8|25,30,47,99]
j=1
half2=[12|16,20,28,63]
k=1
```

```
[8,12,16|_,_,_,_,_,_,_,_]
i=3
half1=[8|25,30,47,99]
j=1
half2=[12,16|20,28,63]
k=2
```

```
[8,12,16,20|_,_,_,_,_,_,_]
i=4
half1=[8|25,30,47,99]
j=1
half2=[12,16,20|28,63]
k=3
```

```
[8,12,16,20,25|_,_,_,_,_]
i=5
half1=[8,25|30,47,99]
j=2
half2=[12,16,20|28,63]
k=3
```

```
[8,12,16,20,25,28|_,_,_,_]
i=6
half1=[8,25|30,47,99]
j=2
half2=[12,16,20,28|63]
k=4
```

```
[8,12,16,20,25,28,30|_,_,_]
i=7
half1=[8,25,30|47,99]
j=3
half2=[12,16,20,28|63]
k=4
```

```
[8,12,16,20,25,28,30,47|_,_]
i=8
half1=[8,25,30,47|99]
j=4
half2=[12,16,20,28|63]
k=4
```

```
[8,12,16,20,25,28,30,47,63|_]
i=9
half1=[8,25,30,47|99]
j=4
half2=[12,16,20,28,63]
k=5
```

```
[8,12,16,20,25,28,30,47,63,99]
i=10
half1=[8,25,30,47|99]
j=5
```

```
public static void sort(Integer[] a) {
 if(a.length>1) {
     int i,mid = a.length/2;
     Integer[] half1 = new Integer[mid];
     Integer[] half2 = new Integer[a.length-mid];
     for(i=0; i<mid; i++)
        half1[i]=a[i];
     for(; i<a.length; i++)
        half2[i-mid]=a[i];
     sort(half1);
     sort(half2);
     int j=0, k=0;
     for(i=0; j<half1.length&&k<half2.length; i++)
        if(half1[j].compareTo(half2[k])<0) {
            a[i]=half1[j];
            j++;
           }
        else {
            a[i]=half2[k];
            k++;
           }
     for(; j<half1.length; i++, j++)
        a[i]=half1[j];
     for(; k<half2.length; i++, k++)
        a[i]=half2[k];
    }
}
```

What you need to know

- You don’t need to know the exact code for every algorithm
- You should have a general feel for each algorithm, and be able to describe it in precise English
- Relating the description of the algorithm to the code will improve your programming skills
- The aim is to reach the point where you can think in terms of algorithm, and translate to code as and when needed

Java's Built-in Sorting

- You don't really need to write sorting code, Java has library code to do it for you
- `Arrays.sort(a)` will sort `a` destructively if `a` is of an array type
- `Collections.sort(a)` will sort `a` destructively if `a` is of an ArrayList type

Natural Ordering

- `Arrays.sort(a)` will sort `a` in ascending numerical ordering if its base type is primitive
- Otherwise, it and `Collections.sort(a)` sort in the order given by the `compareTo` operator called on objects of their base type
- This is referred to as the base type's "natural order"

`compareTo`

- With `str1` and `str2` of type `String`, `str1.compareTo(str2)` returns
  - A negative integer if `str1` comes before `str2` alphabetically
  - A positive integer if `str1` comes after `str2` alphabetically
  - 0 if they are equal
- This means the natural order of strings is alphabetical
- You could imagine other orderings, e.g. by length

Generalising sorting

- If you write your own class to describe a type of object, you can write your own `compareTo` method for the class, this means Java's built-in sort code will sort collections of that type using that ordering
- Java's built-in sort code has a way of giving a different ordering, so collections can be sorted in an order different from natural order
- More details on this later

Efficiency

- We have seen several different algorithms for the one problem of sorting
- When we run them and time them:
  - Some run much faster than others, particularly for large amounts of data
  - It's always the same algorithms which run fastest

Efficiency analysis

- Instead of just running experiments with code, try and use reason to see why one algorithm runs faster than another
- Consider the main action of an algorithm and how many times this action takes place
- For example, with sorting, the main action is to compare one item with another

Searching

- Searching is a simpler problem than sorting, and easily demonstrates an algorithm difference when searching an ordered collection e.g.

  `[8,12,16,20,25,28,30,47,63,99]`
- Problem is to find whether a particular item is in a collection
- There are two algorithms:
  - Linear search
  - Binary search

Linear Search

- Look at each item in turn until we have found the one we want, or found one greater than it (as the collection is ordered this means the one we want isn't there)

```
public static boolean search(Integer[] a, Integer n)
{
 int i;
 for(i=0; i<a.length&&a[i].compareTo(n)<0; i++) {}
 return (i<a.length&&a[i].equals(n));
}
```

```
[8|12,16,20,25,28,30,47,63,99]
 ^

n=27

i=0
```

```
[8,12|16,20,25,28,30,47,63,99]
     ^

n=27

i=1
```

```
[8,12,16|20,25,28,30,47,63,99]
         ^

n=27
i=2
```

```
[8,12,16,20|25,28,30,47,63,99]
            ^

n=27
i=3
```

```
[8,12,16,20,25|28,30,47,63,99]
              ^

n=27
i=4
```

```
[8,12,16,20,25,28|30,47,63,99]
                 ^

n=27
i=5
```

Best case, Worst case

- Best case - item being searched for is first item or less than first item, halt after one comparison
- Worst case - item being searched for is last item, or larger than last item, halt after N comparisons for N items
- Average case - depends on distribution, N/2 comparisons if the item occurs and there is an equal chance of any item

Binary Search

- Look at middle item
  - If it is the item we want, halt
  - If it is greater than the item we want, repeat search only on items below it
  - If it is less than the item we want, repeat search only on items above it
  - Halt search with item not found when range being searched in is of size 0

```
public static boolean search(Integer[] a, Integer n)
{
 int from=0, to=a.length;
 while(from!=to)
    {
     int mid = (from+to)/2;
     int res = n.compareTo(a[mid]);
     if(res==0)
        return true;
     else if(res<0)
        to=mid;
     else
        from=mid+1;
    }
 return false;
}
```

```
[|8,12,16,20,25,28,30,47,63,99|]
                      ^

n=27
from=0
to=10
mid=5
```

```
[|8,12,16,20,25|28,30,47,63,99]
          ^

n=27
from=0
to=5
mid=2
```

```
[8,12,16|20,25|28,30,47,63,99]
                ^

n=27
from=3
to=5
mid=4
```

```
[8,12,16,20,25||28,30,47,63,99]

n=27
from=5
to=5
```

Binary search v. Linear search

- Linear search – each step reduces the size of the portion to be looked at by 1
- Binary search – each step reduces the size of the portion to be looked at by half its original size
- Linear search – up to `N` steps for a collection of size `N`
- Binary Search - maximum number of steps is the number of times `N` can be cut in half

How many times can 1000 be cut in half?

- 1000
- 500
- 250
- 125
- 62
- 31
- 15
- 7
- 3
- 1
- 0

Ten times

Logarithms

- $2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 1024$
- $2^{10} = 1024$
- $\log_2 1024 = 10$
- In general if $a^b = c$, we define $\log_a c = b$
- If each step in an algorithm involves one action and cutting a collection of original size `N` into half, it will do no more than $\log_2 N$ actions
- For large values of `N`, $\log_2 N$ is much smaller than `N`

Selection Sort steps

- We make `N-1` comparisons to find the smallest item out of `N` items
- Then `N-2` comparisons to find the smallest item out of the remaining `N-1` items
- Then `N-3` comparisons to find the smallest item out of the remaining `N-2` items

…

- Then `2` comparisons to find the smallest item out of the remaining `3` items
- Finally, `1` comparison to find the smallest item out of the remaining `2` items

Number of comparisons in selection sort

$(N-1)+(N-2)+(N-3)+\ldots+3+2+1$

Number of comparisons in selection sort

$N+(N-2)+(N-3)+\ldots+3+2$

Number of comparisons in selection sort

`N+N+(N-3)+…+3`

Number of comparisons in selection sort

N+N+N+…

Number of comparisons in selection sort

$N+N+N+\ldots$

N added together $(N-1)/2$ times

which is $(N^2-N)/2$

When N is large, this is almost $N^2/2$

Order of Computation

$N+N+N+\ldots$

N added together $(N-1)/2$ times

With $(N^2-N)/2$ steps, when N is large, this is almost $N^2/2$

With a faster or slower computer, the time taken could be doubled or halved, but it is still proportional to N^2

So we say selection sort is "Order N^2" or $O(N^2)$

Big O Notation

- Binary search is `O(log N)`
- Linear search is `O(N)`

In general:

- Find a formula which gives the number of steps
- Consider only the most significant term in this formula
- Ignore constant multiplying factors

Number of Comparisons in Insertion Sort

- 0 to insert first item into empty sorted list
- 1 to insert second item into list of 1 item
- 1-2 to insert third item into list of 2 items
- 1-3 to insert fourth item into list of 3 items

…

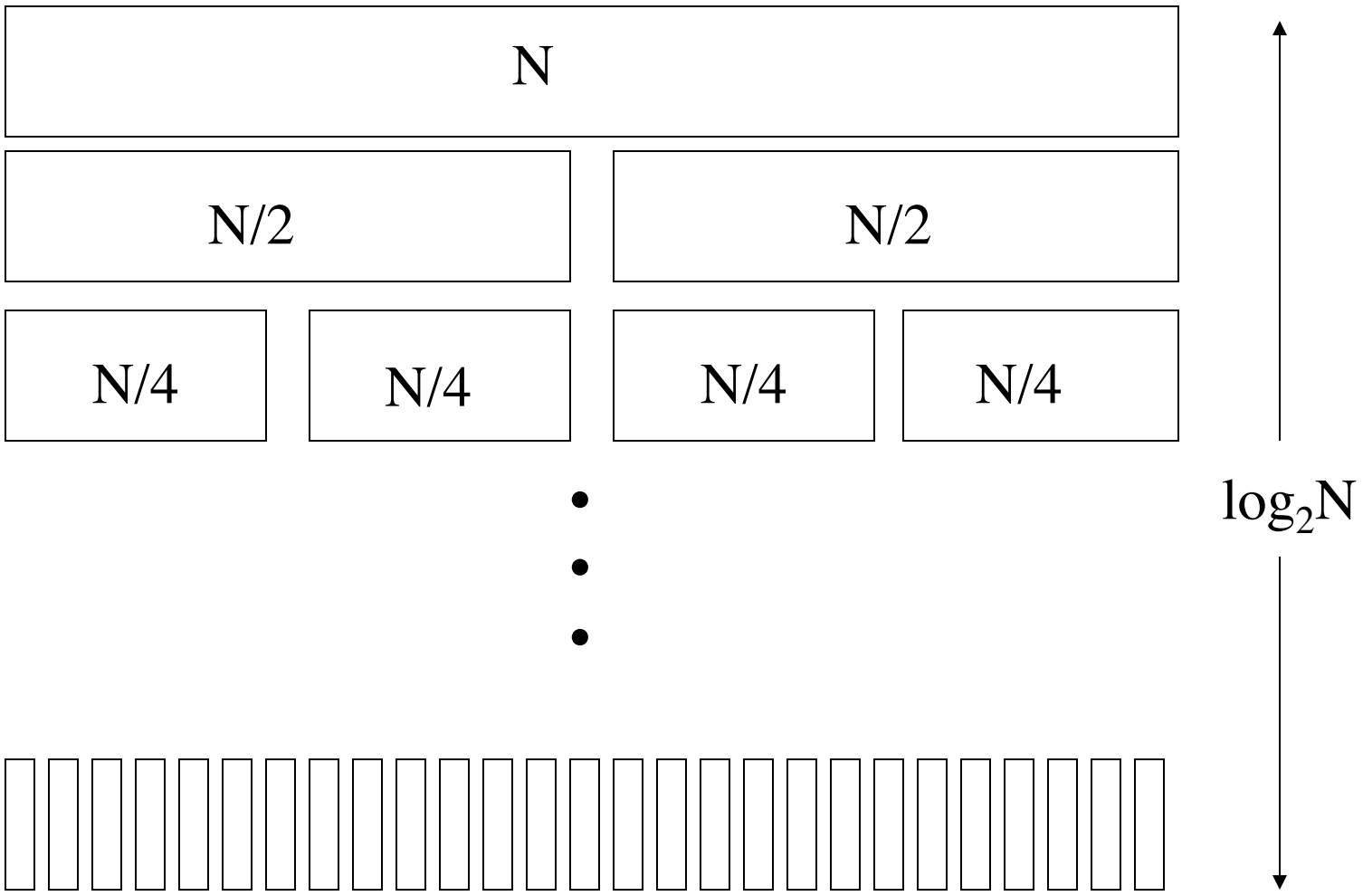
- 1-(N-1) to insert Nth item into list of (N-1) items

- Best case `N-1` comparisons
- Worst case `(N-1)+(N-2)+...+2+1` comparisons
- So insertion sort is $O(N^2)$ as well

Number of Comparisons in Merge Sort

- None to split list of size `N` into two
- Up to `N` to merge the two sorted lists of size `N/2`
- Two lists of size `N/2` each required `N/2` comparisons to merge two sorted lists of size `N/4`
- Four lists of size `N/4` each required `N/4` comparisons to merge two sorted lists of size `N/8`

…


N
N/2
N/2
N/4
N/4
N/4
N/4
log2N

Order of Merge Sort

- Merge sort can be considered as breaking into $\log_2 N$ layers each merging a total of up to N items
- So merge sort is $O(N \log N)$
- When N is large, $N \log N$ is much smaller than N^2
- So merge sort is much more efficient than selection sort and insertion sort

Number of comparisons in Quicksort

- Like merge sort, but the comparisons are made when the list is split into two rather than when the two sorted lists are joined
- If the items are randomly distributed, the two lists after the split will be roughly equal in size, so same reasoning as merge sort can be made
- This makes quicksort `O(N log N)`

Quicksort - worst case

- Suppose the pivot is the first item, and it is always the case that all the items are greater than the pivot (i.e. list is already sorted)
- Then a list of size `N` will be split into an empty list and a list of size `N-1` taking `N-1` comparisons
- Sorting the list of size `N-1` takes `N-2` comparisons to give an empty list and a list of size `N-2`

…

- So worst case for quicksort is $O(N^2)$

Efficiency

- Different algorithms for the same problem can have big differences in efficiency
- For large amounts of data, the algorithm used is more important than any other factor in the speed of finding a solution
- We can use reasoning to categorise algorithms under the “big-O” notation
- We need to be aware that some algorithms have best/worst cases of a different order than their average case