CPSC 221
Basic Algorithms and Data Structures

May 27, 2015

Administrative stuff

Lab 5 is posted.

Programming assignment 1 is posted. Due no later than Friday, June 5, 11:59pm.

Final exam is Wednesday, June 24 at 8:30am.

Administrative stuff

Turning in Theory Assignment 1

- Use online handin
- The course is ‘cs221’
- The name is ‘ta1’
- Handin is up and running
  - (some submissions already in)

Deadline is midnight tonight!

Administrative stuff

Midterm exam is 1 week from today

There will be no labs next week.

But the TAs will be in our lab room (X350) holding office hours during the Tuesday lab time and the Wednesday lab time that’s right before class.

Administrative stuff

Midterm exam is 1 week from today

- 90 minutes
- You may bring three (3) 8.5 x 11 inch sheets of paper with your notes, double-sided, any format
- You may not bring magnifying glasses, microscopes, weird goggles, Google glass, etc.
- Exam will start shortly after beginning of class
- Exam will take place in SWNG 121 and SWNG 122

Administrative stuff

Midterm exam is 1 week from today

Exam protocol:

No electronics, no books

Pen or pencil

Make sure you sit with an empty chair or aisle on either side of you. There should be plenty of space with two Swing Space rooms. We'll tell you in advance which of the two rooms you should go to.

Administrative stuff

Midterm exam is 1 week from today

Exam protocol:

You can’t leave the room during the first 30 minutes of the exam.

After that, washroom breaks are escorted by TA and only one at a time.

If you arrive more than 30 minutes after the start of the exam, you will not be permitted to write the exam.

Administrative stuff

Midterm exam is 1 week from today

Exam protocol:

If you are sick on exam day, don’t write the exam. Bring us documentation from your physician. We’ll count your final exam for both midterm and final exam marks.

If you write the exam, you own the mark, regardless of whether you were sick. You don’t get to write the exam and then tell us it shouldn’t count because you were ill.

Administrative stuff

Midterm exam is 1 week from today

Sample midterm questions and solutions are posted. Note that this is a sample from a different offering of the course, and the ordering of topics was different. We haven’t seen some of that stuff yet (e.g., heaps), and we probably won’t see that stuff (e.g. heaps) before the exam.

Note the domain/range question: that’s in your assigned reading from the discrete math book, and is stuff you should know from CPSC 121. This question is a reminder that while we don’t cover it again in class, you should be able to deal with questions about basic discrete math.

Questions?

Big-O

The formal mathematical definition for Big-O:

T(n) is O(f(n)) if there are two constants, n_0 and c, both > 0, and a function f(n) such that cf(n) >= T(n) for all $n > n_0$

Binary search with recursion

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    int mid = (left + right) / 2;
    if (array[mid] == target)
        return mid;
    else if (target < array[mid])
        return binSearch(array, target, left, mid - 1);
    else
        return binSearch(array, target, mid + 1, right);
}
```

Binary search with recursion

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑														↑

```
int binSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    int mid = (left + right) / 2;
    if (array[mid] == target)
        return mid;
    else if (target < array[mid])
        return binSearch(array, target, left, mid - 1);
    else
        return binSearch(array, target, mid + 1, right);
}

cout << binSearch(array, 55, 0, 14) << endl;

binSearch(array, 55, 0, 14)
```

Binary search with recursion

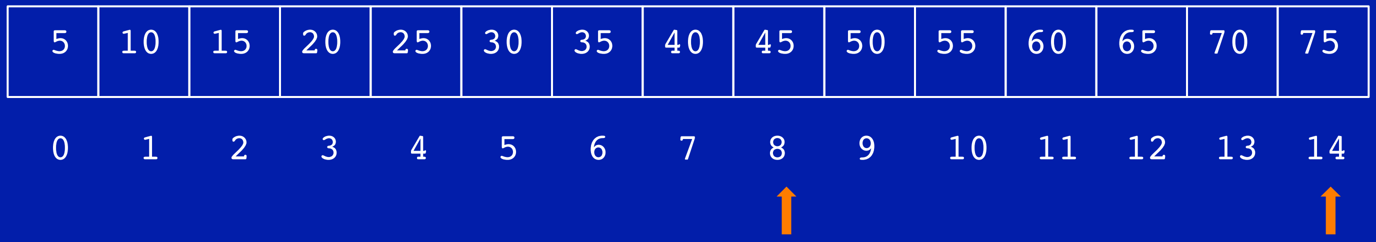
5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑							↑							↑

```
int binSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    int mid = (left + right) / 2;
    if (array[mid] == target)
        return mid;
    else if (target < array[mid])
        return binSearch(array, target, left, mid - 1);
    else
        return binSearch(array, target, mid + 1, right);
}

cout << binSearch(array, 55, 0, 14) << endl;

binSearch(array, 55, 0, 14)
```

Binary search with recursion



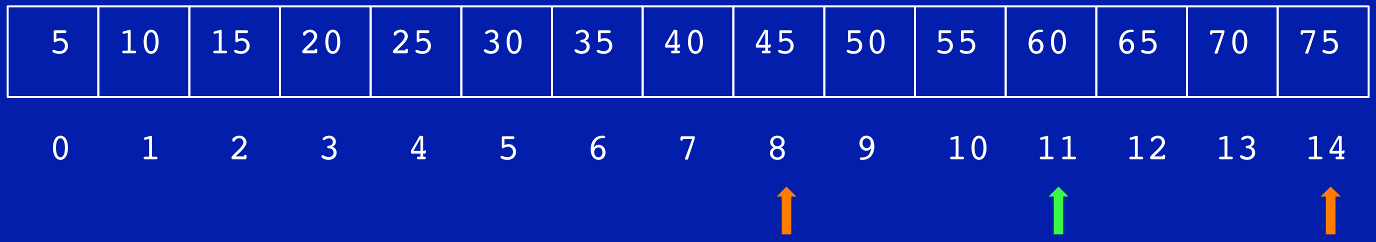
5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    int mid = (left + right) / 2;
    if (array[mid] == target)
        return mid;
    else if (target < array[mid])
        return binSearch(array, target, left, mid - 1);
    else
        return binSearch(array, target, mid + 1, right);
}

cout << binSearch(array, 55, 0, 14) << endl;

binSearch(array, 55, 0, 14)
  binSearch(array, 55, 8, 14)
```

Binary search with recursion



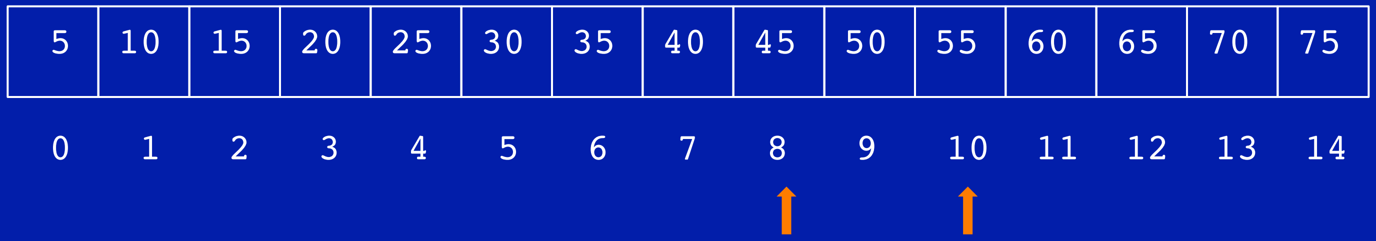
5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    int mid = (left + right) / 2;
    if (array[mid] == target)
        return mid;
    else if (target < array[mid])
        return binSearch(array, target, left, mid - 1);
    else
        return binSearch(array, target, mid + 1, right);
}

cout << binSearch(array, 55, 0, 14) << endl;

binSearch(array, 55, 0, 14)
  binSearch(array, 55, 8, 14)
```

Binary search with recursion



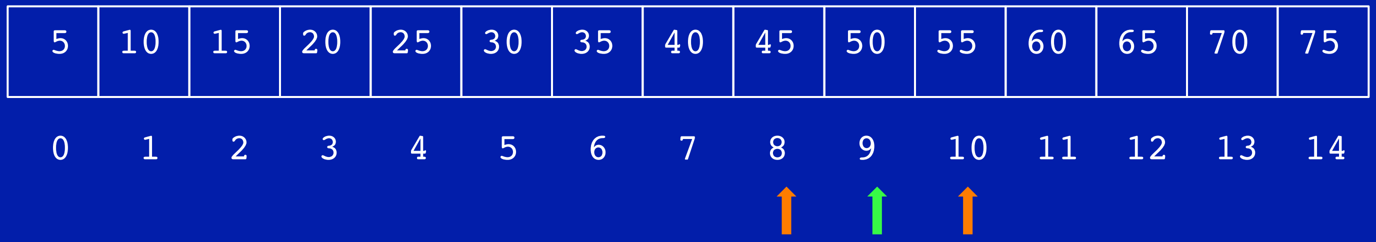
5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    int mid = (left + right) / 2;
    if (array[mid] == target)
        return mid;
    else if (target < array[mid])
        return binSearch(array, target, left, mid - 1);
    else
        return binSearch(array, target, mid + 1, right);
}

cout << binSearch(array, 55, 0, 14) << endl;

binSearch(array, 55, 0, 14)
  binSearch(array, 55, 8, 14)
    binSearch(array, 55, 8, 10)
```

Binary search with recursion



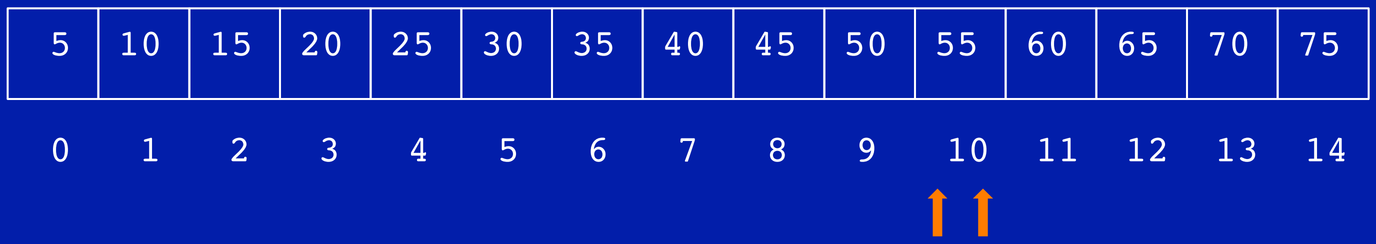
5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    int mid = (left + right) / 2;
    if (array[mid] == target)
        return mid;
    else if (target < array[mid])
        return binSearch(array, target, left, mid - 1);
    else
        return binSearch(array, target, mid + 1, right);
}

cout << binSearch(array, 55, 0, 14) << endl;

binSearch(array, 55, 0, 14)
  binSearch(array, 55, 8, 14)
    binSearch(array, 55, 8, 10)
```

Binary search with recursion



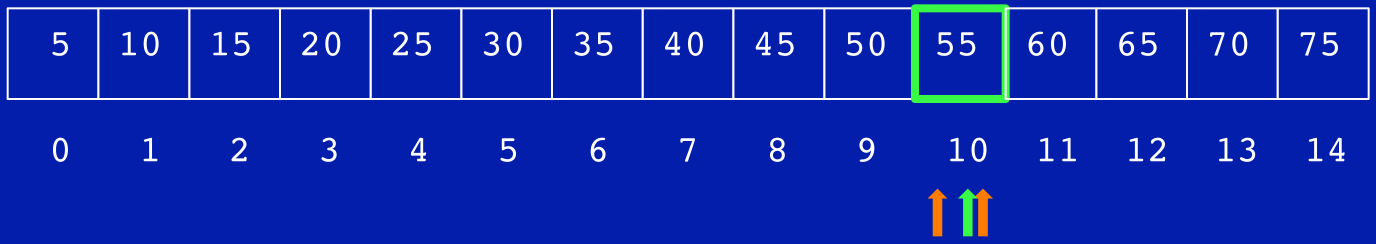
5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    int mid = (left + right) / 2;
    if (array[mid] == target)
        return mid;
    else if (target < array[mid])
        return binSearch(array, target, left, mid - 1);
    else
        return binSearch(array, target, mid + 1, right);
}

cout << binSearch(array, 55, 0, 14) << endl;

binSearch(array, 55, 0, 14)
  binSearch(array, 55, 8, 14)
    binSearch(array, 55, 8, 10)
      binSearch(array, 55, 10, 10)
```

Binary search with recursion

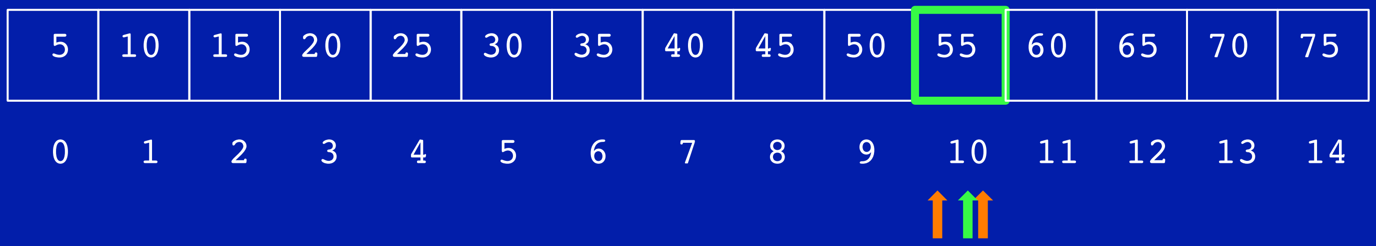


```
int binSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    int mid = (left + right) / 2;
    if (array[mid] == target)
        return mid;
    else if (target < array[mid])
        return binSearch(array, target, left, mid - 1);
    else
        return binSearch(array, target, mid + 1, right);
}

cout << binSearch(array, 55, 0, 14) << endl;

binSearch(array, 55, 0, 14)
  binSearch(array, 55, 8, 14)
    binSearch(array, 55, 8, 10)
      binSearch(array, 55, 10, 10)
```

Binary search with recursion



```
int binSearch(int array[], int target, int left, int right)
{
    if (right < left) return -1;
    int mid = (left + right) / 2;
    if (array[mid] == target)
        return mid;
    else if (target < array[mid])
        return binSearch(array, target, left, mid - 1);
    else
        return binSearch(array, target, mid + 1, right);
}

cout << binSearch(array, 55, 0, 14) << endl;

binSearch(array, 55, 0, 14)
  binSearch(array, 55, 8, 14)
    binSearch(array, 55, 8, 10)
      binSearch(array, 55, 10, 10)
10 // the returned value
```

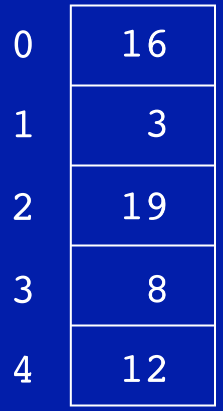
Searching and sorting

Computers are essential for keeping track of large quantities of data and finding little pieces of that data on demand. Searching for and finding data when necessary is made much easier when the data is sorted in some way, so computer people think a lot about how to sort things. Finding medical records, banking information, income tax returns, driver's licence information...even names in a phone book...all depend on the information being sorted.

Searching and sorting

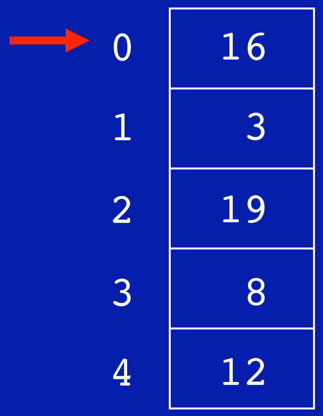
There’s lots of information out there to be searched. So computer folks have spent a lot of time thinking about how to sort efficiently. Does the efficiency of one approach to sorting when compared to another really make a difference? It could. Here's an example of a well-known, simple, but not very efficient sorting algorithm that's easily implemented using an array.

Selection sort



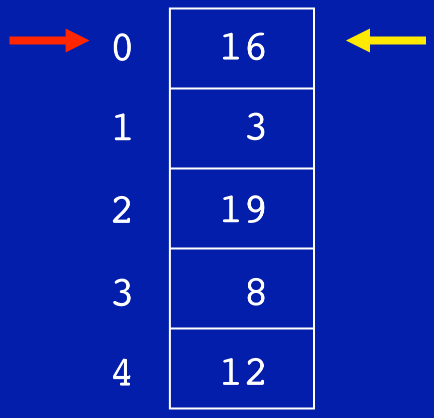
Let's say we want to sort the values in the array at the left in increasing order. One way to approach the problem would be to use an algorithm called selection sort.

Selection sort



Let's say we want to sort the values in the array at the left in increasing order. One way to approach the problem would be to use an algorithm called selection sort. We start by setting a pointer to the first element in the array; this is where the smallest value in the array will be placed.

Selection sort

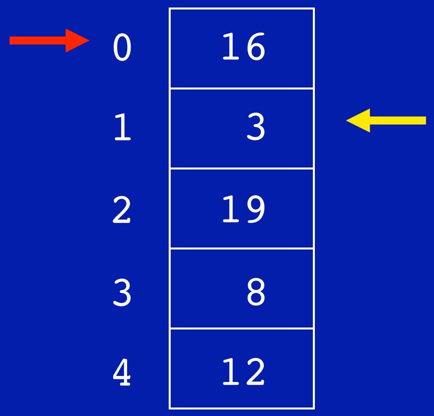


Let's say we want to sort the values in the array at the left in increasing order. One way to approach the problem would be to use an algorithm called selection sort. We start by setting a pointer to the first element in the array; this is where the smallest value in the array will be placed. Then we'll look at every value in this unsorted array and find the minimum value.

The smallest value so far is 16

Its index is 0

Selection sort

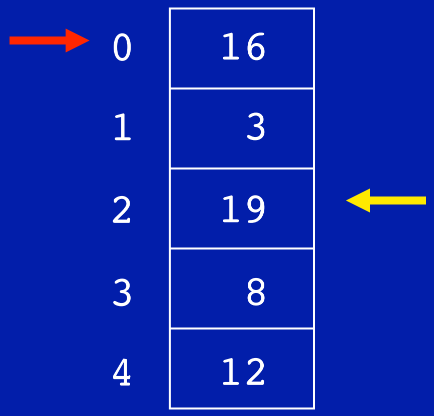


Let's say we want to sort the values in the array at the left in increasing order. One way to approach the problem would be to use an algorithm called selection sort. We start by setting a pointer to the first element in the array; this is where the smallest value in the array will be placed. Then we'll look at every value in this unsorted array and find the minimum value.

The smallest value so far is 3

Its index is 1

Selection sort

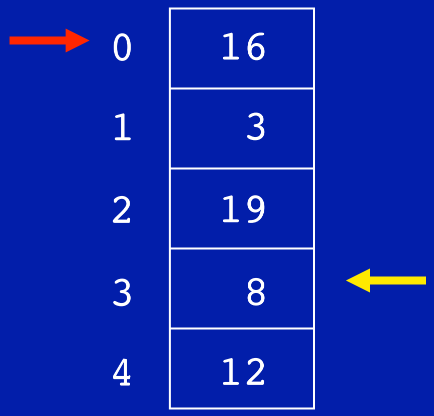


Let's say we want to sort the values in the array at the left in increasing order. One way to approach the problem would be to use an algorithm called selection sort. We start by setting a pointer to the first element in the array; this is where the smallest value in the array will be placed. Then we'll look at every value in this unsorted array and find the minimum value.

The smallest value so far is 3

Its index is 1

Selection sort

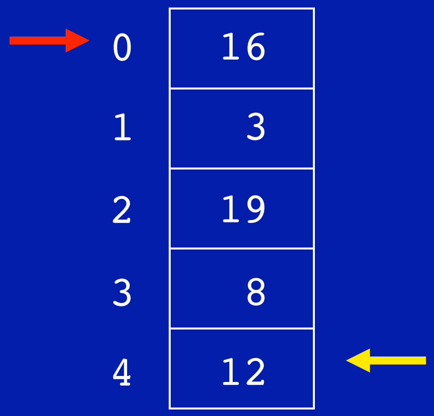


Let's say we want to sort the values in the array at the left in increasing order. One way to approach the problem would be to use an algorithm called selection sort. We start by setting a pointer to the first element in the array; this is where the smallest value in the array will be placed. Then we'll look at every value in this unsorted array and find the minimum value.

The smallest value so far is 3

Its index is 1

Selection sort

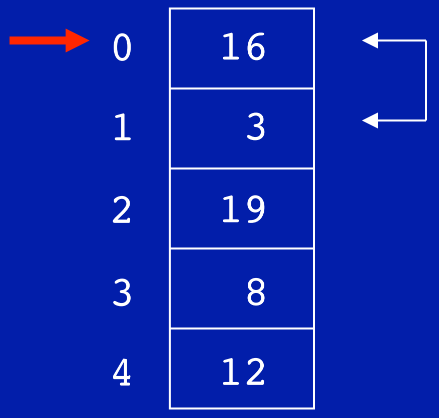


Let's say we want to sort the values in the array at the left in increasing order. One way to approach the problem would be to use an algorithm called selection sort. We start by setting a pointer to the first element in the array; this is where the smallest value in the array will be placed. Then we'll look at every value in this unsorted array and find the minimum value.

The smallest value so far is 3

Its index is 1

Selection sort

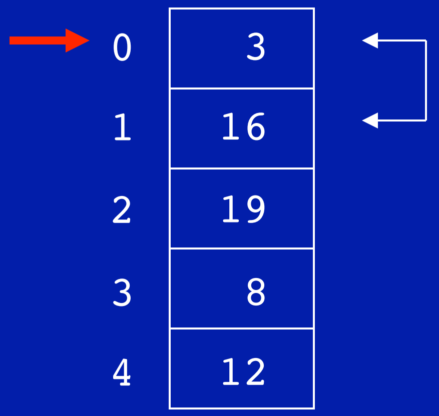


The smallest value so far is 3

Its index is 1

Let's say we want to sort the values in the array at the left in increasing order. One way to approach the problem would be to use an algorithm called selection sort. We start by setting a pointer to the first element in the array; this is where the smallest value in the array will be placed. Then we'll look at every value in this unsorted array and find the minimum value. Once we've found the minimum value, we swap that value with the one we selected at the beginning.

Selection sort

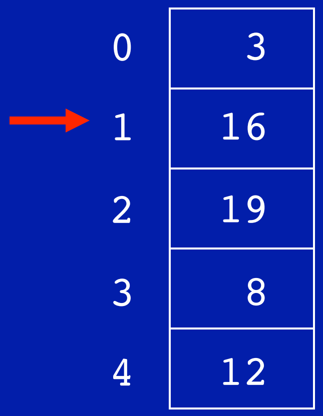


The smallest value so far is 3

Its index is 1

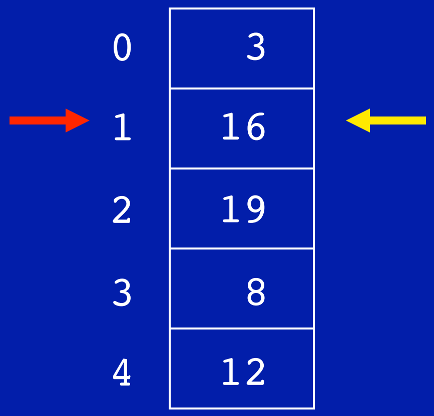
Let's say we want to sort the values in the array at the left in increasing order. One way to approach the problem would be to use an algorithm called selection sort. We start by setting a pointer to the first element in the array; this is where the smallest value in the array will be placed. Then we'll look at every value in this unsorted array and find the minimum value. Once we've found the minimum value, we swap that value with the one we selected at the beginning.

Selection sort



At this point we know that the smallest number in the array is in the first (index 0) element in the array. That is, the first element is sorted, and the rest of the array remains unsorted. So now we can select the second element of the array to be the location which will hold the next smallest value in the array.

Selection sort



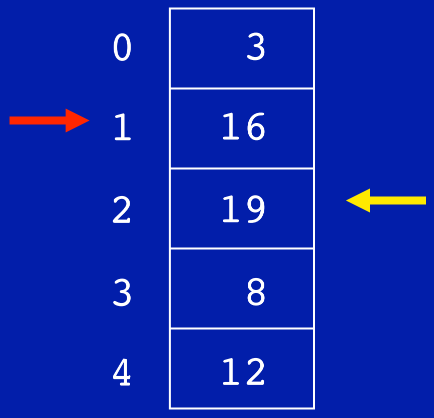
At this point we know that the smallest number in the array is in the first (index 0) element in the array. That is, the first element is sorted, and the rest of the array remains unsorted. So now we can select the second element of the array to be the location which will hold the next smallest value in the array.

In other words, we'll do all that stuff we just did, only we'll do it only to the unsorted part of the array -- in this case, all but the first element.

The smallest value so far is 16

Its index is 1

Selection sort



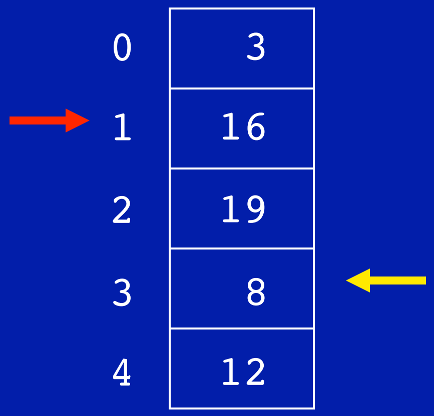
The smallest value so far is 16

Its index is 1

At this point we know that the smallest number in the array is in the first (index 0) element in the array. That is, the first element is sorted, and the rest of the array remains unsorted. So now we can select the second element of the array to be the location which will hold the next smallest value in the array.

In other words, we'll do all that stuff we just did, only we'll do it only to the unsorted part of the array -- in this case, all but the first element.

Selection sort



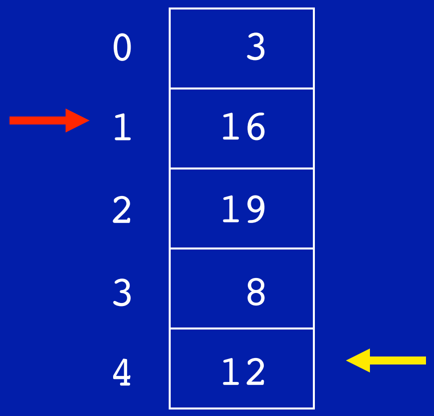
The smallest value so far is 8

Its index is 3

At this point we know that the smallest number in the array is in the first (index 0) element in the array. That is, the first element is sorted, and the rest of the array remains unsorted. So now we can select the second element of the array to be the location which will hold the next smallest value in the array.

In other words, we'll do all that stuff we just did, only we'll do it only to the unsorted part of the array -- in this case, all but the first element.

Selection sort



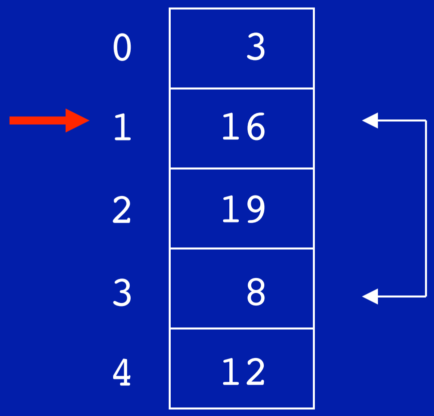
At this point we know that the smallest number in the array is in the first (index 0) element in the array. That is, the first element is sorted, and the rest of the array remains unsorted. So now we can select the second element of the array to be the location which will hold the next smallest value in the array.

In other words, we'll do all that stuff we just did, only we'll do it only to the unsorted part of the array -- in this case, all but the first element.

The smallest value so far is 8

Its index is 3

Selection sort



The smallest value so far is 8

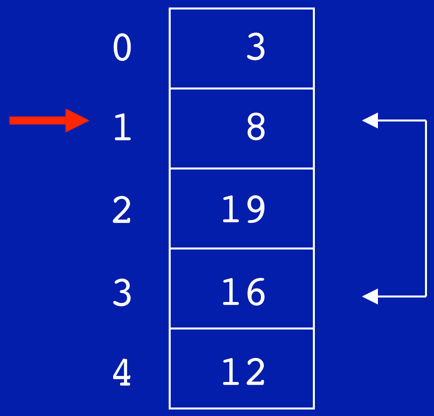
Its index is 3

At this point we know that the smallest number in the array is in the first (index 0) element in the array. That is, the first element is sorted, and the rest of the array remains unsorted. So now we can select the second element of the array to be the location which will hold the next smallest value in the array.

In other words, we'll do all that stuff we just did, only we'll do it only to the unsorted part of the array -- in this case, all but the first element.

Now we swap the minimum value with the selected array value -- in this case, the second element.

Selection sort



The smallest value so far is 8

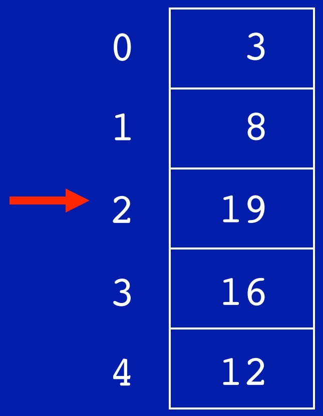
Its index is 3

At this point we know that the smallest number in the array is in the first (index 0) element in the array. That is, the first element is sorted, and the rest of the array remains unsorted. So now we can select the second element of the array to be the location which will hold the next smallest value in the array.

In other words, we'll do all that stuff we just did, only we'll do it only to the unsorted part of the array -- in this case, all but the first element.

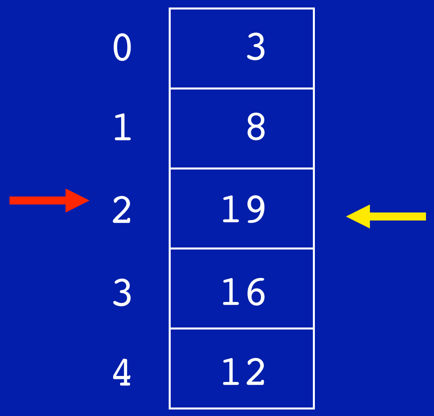
Now we swap the minimum value with the selected array value -- in this case, the second element.

Selection sort



Now the first two elements of the array are sorted. We select the third element of the array (index 2) as the eventual location of the next smallest value, and search the unsorted portion of the array for that value, just like before.

Selection sort

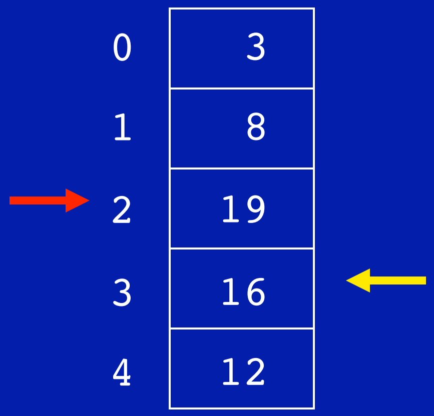


Now the first two elements of the array are sorted. We select the third element of the array (index 2) as the eventual location of the next smallest value, and search the unsorted portion of the array for that value, just like before.

The smallest value so far is 19

Its index is 2

Selection sort

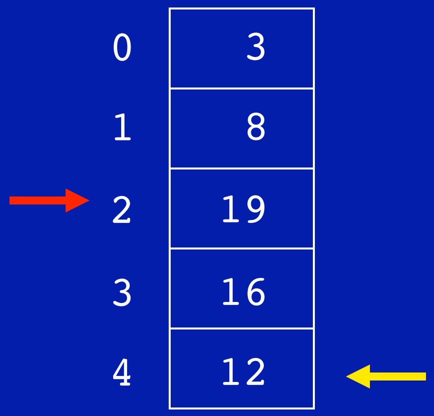


Now the first two elements of the array are sorted. We select the third element of the array (index 2) as the eventual location of the next smallest value, and search the unsorted portion of the array for that value, just like before.

The smallest value so far is 16

Its index is 3

Selection sort

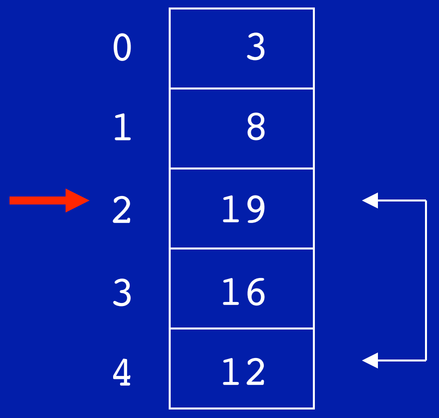


Now the first two elements of the array are sorted. We select the third element of the array (index 2) as the eventual location of the next smallest value, and search the unsorted portion of the array for that value, just like before.

The smallest value so far is 12

Its index is 4

Selection sort



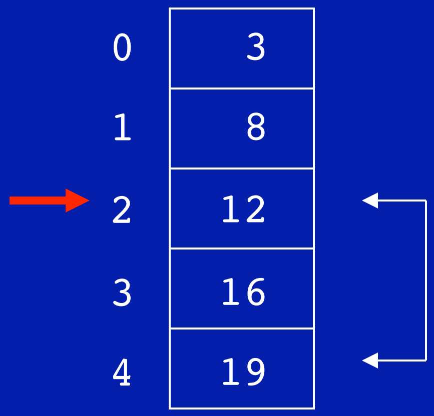
Now the first two elements of the array are sorted. We select the third element of the array (index 2) as the eventual location of the next smallest value, and search the unsorted portion of the array for that value, just like before.

Again, we swap the values...

The smallest value so far is 12

Its index is 4

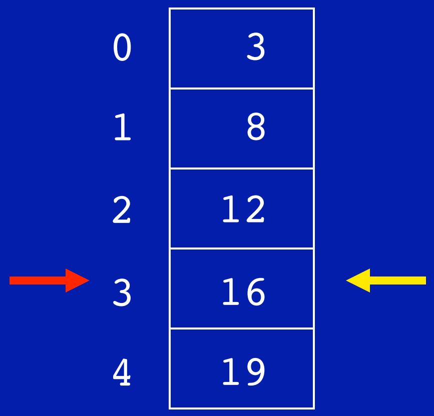
Selection sort



Now the first two elements of the array are sorted. We select the third element of the array (index 2) as the eventual location of the next smallest value, and search the unsorted portion of the array for that value, just like before.

Again, we swap the values...

Selection sort



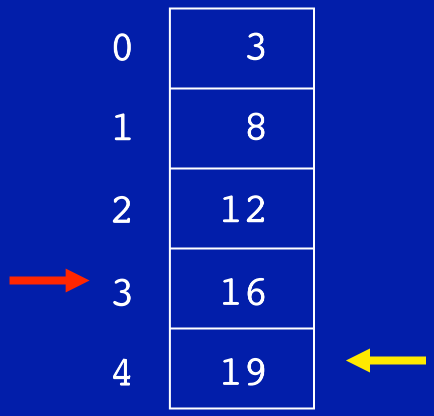
Now the first two elements of the array are sorted. We select the third element of the array (index 2) as the eventual location of the next smallest value, and search the unsorted portion of the array for that value, just like before.

Again, we swap the values...and do the whole thing again.

The smallest value so far is 16

Its index is 3

Selection sort



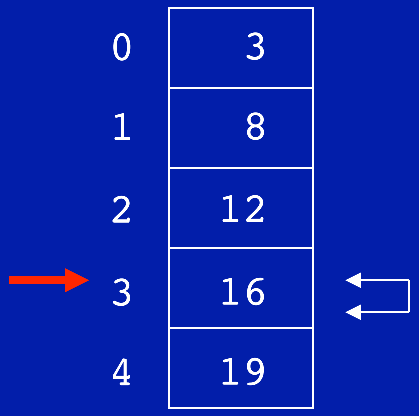
Now the first two elements of the array are sorted. We select the third element of the array (index 2) as the eventual location of the next smallest value, and search the unsorted portion of the array for that value, just like before.

Again, we swap the values...and do the whole thing again.

The smallest value so far is 16

Its index is 3

Selection sort



The smallest value so far is 16

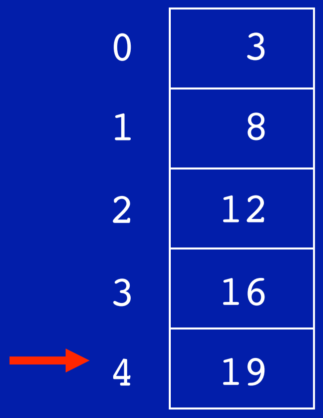
Its index is 3

Now the first two elements of the array are sorted. We select the third element of the array (index 2) as the eventual location of the next smallest value, and search the unsorted portion of the array for that value, just like before.

Again, we swap the values...and do the whole thing again.

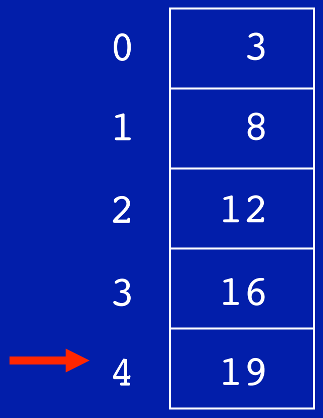
And we swap again...it's not actually necessary in this particular case, but it's what the algorithm says to do, so we do it.

Selection sort



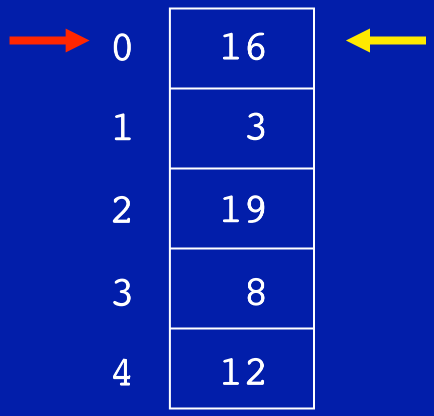
Finally, we could select the last element of the array (index 4). But since we know all of the array except for this element is already sorted, we can quickly conclude that this element is the largest value in the array, so it's already where we want it. In other words, the array is sorted, and we're done. We don't need to select the last element.

Selection sort



There's a pseudocode version of this algorithm in your textbook. You should take a look at it and compare to what we've just done.

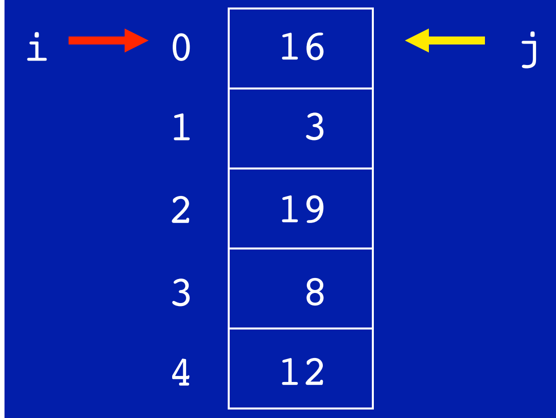
Selection sort



As the example proceeded, you watched the red arrow on the left and the yellow arrow on the right move down the array. The arrows represent two different variables, each one containing an index into the array. A Java program that performs a selection sort on an array of numbers is shown on the next slide; when you look at it, think of the outer loop a controlling the movement of the red arrow, and the inner loop as controlling the movement of the yellow arrow.

You should be able to translate the Java to C++ in your sleep.

Selection sort



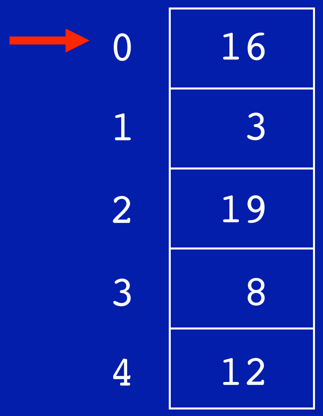
```
// selection sort
public class SortTest1
{
  public static void main(String[] args)
  {
    int[] numbers = {16,3,19,8,12};
    int min, temp;
    //select location of next sorted value
    for (int i = 0; i < numbers.length-1; i++)
    {
      min = i;
      //find the smallest value in the remainder of
      //the array to be sorted
      for (int j = i+1; j < numbers.length; j++)
      {
        if (numbers[j] < numbers[min])
        {
          min = j;
        }
      }
      //swap two values in the array
      temp = numbers[i];
      numbers[i] = numbers[min];
      numbers[min] = temp;
    }

    System.out.println("Printing sorted result");
    for (int i = 0; i < numbers.length; i++)
    {
      System.out.println(numbers[i]);
    }
  }
}
```

Now for the good stuff

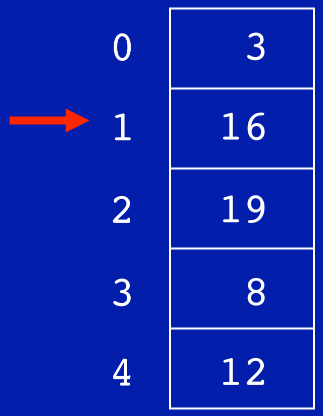
Selection sort is a comparison-based sort, so to get some sense of the time requirements, we count the comparisons that must be made to complete the sorting.

Estimating time required to sort



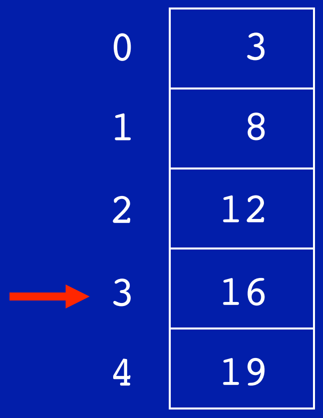
We can go back to the selection sort example and count the comparisons. The first pass through the array of 5 elements started with 16 being compared to 3, then 3 was compared to 19, 8, and 12. There were 4 comparisons. The value 3 was moved into the location at index 0.

Estimating time required to sort



We can go back to the selection sort example and count the comparisons. The first pass through the array of 5 elements started with 16 being compared to 3, then 3 was compared to 19, 8, and 12. There were 4 comparisons. The value 3 was moved into the location at index 0. Then the second pass through the array began, starting with index 1. 16 was compared to 19, then 16 was compared to 8, which became the new minimum and was compared to 12. So among 4 elements in the array, there were 3 comparisons.

Estimating time required to sort



It takes 4 passes through the array to get it completely sorted. There are 4 comparisons on the first pass, 3 comparisons on the second pass, 2 comparisons on the third pass, and 1 comparison on the last pass. That is, it takes 4 + 3 + 2 + 1 = 10 comparisons to sort an array of five values.

If you do this same computation on an array with six values, you'll find it takes 5 + 4 + 3 + 2 + 1 = 15 comparisons to sort the array. Do you see a familiar pattern?

The number of comparisons required to perform selection sort on an array of N values is given by the expression: N*(N-1)/2. The time complexity here is O(N^2).

Estimating time required to sort

Either way, it should be easy to see that as N, the number of values in the array gets very big, the number of comparisons needed to sort the array grows in proportion to N^2, with the other terms becoming insignificant by comparison.

So sorting an array of 1,000 values would require approximately 1,000,000 comparisons. Similarly, sorting an array of 1,000,000 values would take approximately 1,000,000,000,000 comparisons.

As the number of values to be sorted grows, the number of comparisons required to sort them grows much faster. Fortunately, there are other sorting algorithms that are much less time-consuming, and we'll talk about them real soon. In the meantime, here are some real numbers to help you think about just how long it might take to sort some really big arrays...

Estimating time required to sort

Let's assume that your computer could make 1 billion (1,000,000,000) comparisons per second. That's a lot of comparisons in a second. And let's say your computer was using selection sort to sort the names of the people in the following hypothetical telephone books. Here's some mathematical food for thought.

phone book	number of people (N)	N^2	number of seconds needed to sort

Estimating time required to sort

Let's assume that your computer could make 1 billion (1,000,000,000) comparisons per second. That's a lot of comparisons in a second. And let's say your computer was using selection sort to sort the names of the people in the following hypothetical telephone books. Here's some mathematical food for thought.

phone book	number of people (N)	N^2	number of seconds needed to sort	
Vancouver	544,320	296,284,262,400	296	or 5 minutes

Estimating time required to sort

Let's assume that your computer could make 1 billion (1,000,000,000) comparisons per second. That's a lot of comparisons in a second. And let's say your computer was using selection sort to sort the names of the people in the following hypothetical telephone books. Here's some mathematical food for thought.

phone book	number of people (N)	N^2	number of seconds needed to sort	
Vancouver	544,320	296,284,262,400	296	or 5 minutes
Canada	30,000,000	900,000,000,000,000	900,000	or 10.4 days

Estimating time required to sort

Let's assume that your computer could make 1 billion (1,000,000,000) comparisons per second. That's a lot of comparisons in a second. And let's say your computer was using selection sort to sort the names of the people in the following hypothetical telephone books. Here's some mathematical food for thought.

phone book	number of people (N)	N^2	number of seconds needed to sort	
Vancouver	544,320	296,284,262,400	296	or 5 minutes
Canada	30,000,000	900,000,000,000,000	900,000	or 10.4 days
People's Republic of China	1,000,000,000	1,000,000,000,000,000,000	1,000,000,000	or 31.7 years

Estimating time required to sort

Let's assume that your computer could make 1 billion (1,000,000,000) comparisons per second. That's a lot of comparisons in a second. And let's say your computer was using selection sort to sort the names of the people in the following hypothetical telephone books. Here's some mathematical food for thought.

phone book	number of people (N)	N^2	number of seconds needed to sort	
Vancouver	544,320	296,284,262,400	296	or 5 minutes
Canada	30,000,000	900,000,000,000,000	900,000	or 10.4 days
People's Republic of China	1,000,000,000	1,000,000,000,000,000,000	1,000,000,000	or 31.7 years
World	7,000,000,000	49,000,000,000,000,000,000	49,000,000,000	or 1554 years

Estimating time required to sort

But note that the best sorting algorithms can run in N $\log_2$ N time instead of N^2 time. If N is 7,000,000,000, then $\log_2$ N is just a little less than 33. So if we round up to 33, N log N would be 231,000,000,000 comparisons. If our computer can perform 1,000,000,000 comparisons per second, then sorting all the names in the whole world phone book now takes 231 seconds instead of 1554 years. And that's why it's important to think about the efficiency of algorithms, especially as the size of your data set gets really really big.

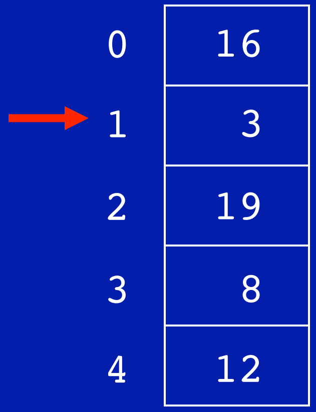
Questions?

Insertion sort

0	16
1	3
2	19
3	8
4	12

Insertion sort takes a slightly different approach. Let’s assume that when we begin to sort the elements of an array, the very first element represents a sorted list. Everything after that remains to be sorted.

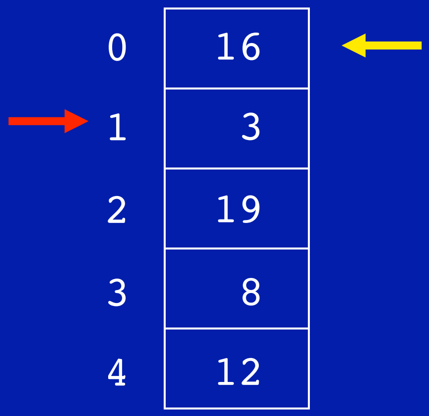
Insertion sort



Insertion sort takes a slightly different approach. Let's assume that when we begin to sort the elements of an array, the very first element represents a sorted list. Everything after that remains to be sorted. So we store the first value in the unsorted portion (indicated by the red arrow) as the item to be inserted into the sorted portion.

The value to be inserted is 3.

Insertion sort

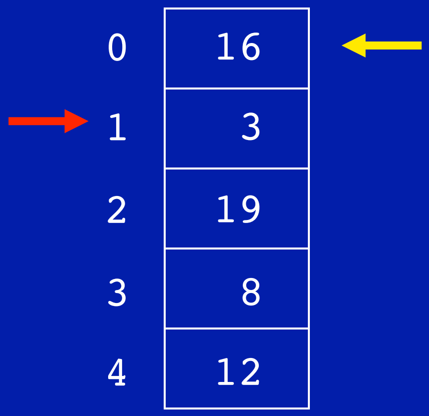


The value to be inserted is 3.

Insertion sort takes a slightly different approach. Let's assume that when we begin to sort the elements of an array, the very first element represents a sorted list. Everything after that remains to be sorted. So we store the first value in the unsorted portion (indicated by the red arrow) as the item to be inserted into the sorted portion.

Now we start comparing the value to be inserted to the sorted items above it, from bottom to top, to determine where the new item belongs among the already sorted items. We'll indicate the bottom of the sorted portion with a yellow arrow.

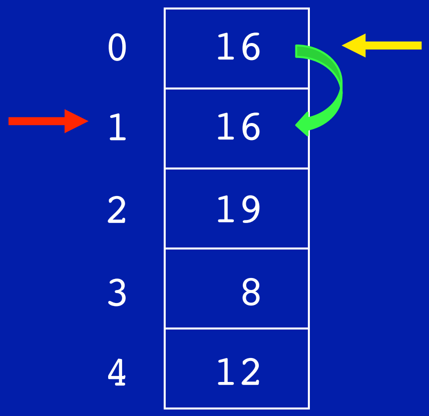
Insertion sort



We ask “Is 3 less than 16, the value at the bottom of the sorted portion?”

The value to be inserted is 3.

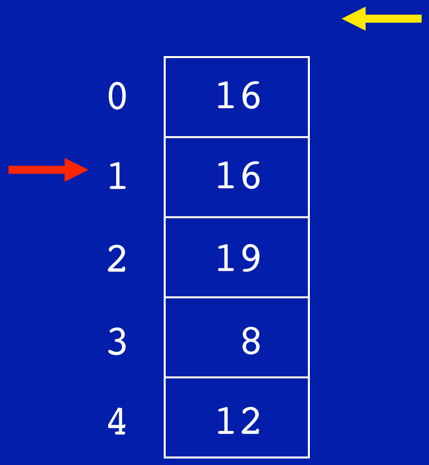
Insertion sort



We ask "Is 3 less than 16, the value at the bottom of the sorted portion?" If so, 3 needs to be above 16 in the sorted portion. We have to make space for 3, so we move 16 down one space.

The value to be inserted is 3.

Insertion sort

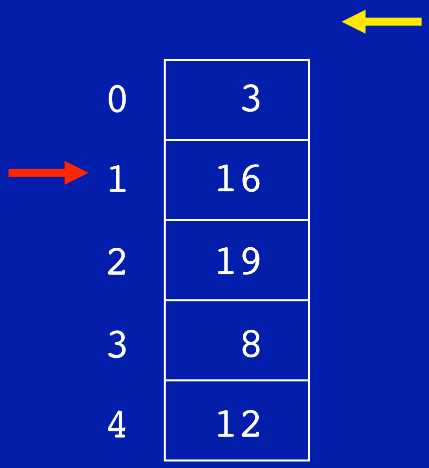


The value to be inserted is 3.

We ask “Is 3 less than 16, the value at the bottom of the sorted portion?” If so, 3 needs to be above 16 in the sorted portion. We have to make space for 3, so we move 16 down one space.

We want to look at the next to last item in the sorted portion, but there is nothing to look at. We’ve run out of array, so we copy the item to be inserted into the top or first place in the array.

Insertion sort

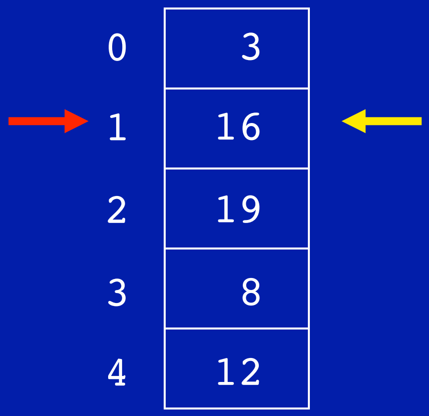


The value to be inserted is 3.

We ask "Is 3 less than 16, the value at the bottom of the sorted portion?" If so, 3 needs to be above 16 in the sorted portion. We have to make space for 3, so we move 16 down one space.

We want to look at the next to last item in the sorted portion, but there is nothing to look at. We've run out of array, so we copy the item to be inserted into the top or first place in the array.

Insertion sort



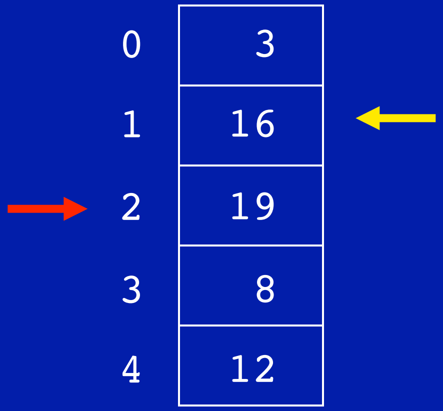
The value to be inserted is 3.

We ask “Is 3 less than 16, the value at the bottom of the sorted portion?” If so, 3 needs to be above 16 in the sorted portion. We have to make space for 3, so we move 16 down one space.

We want to look at the next to last item in the sorted portion, but there is nothing to look at. We’ve run out of array, so we copy the item to be inserted into the top or first place in the array.

We reset the pointer to the bottom of the sorted portion,

Insertion sort



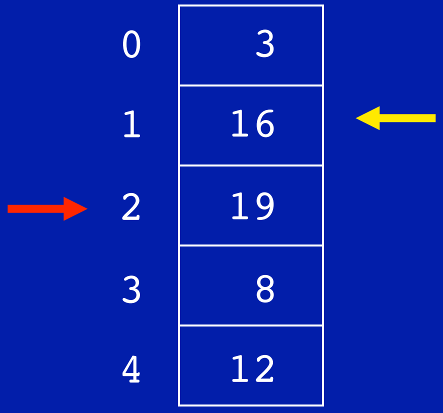
The value to be inserted is 19.

We ask "Is 3 less than 16, the value at the bottom of the sorted portion?" If so, 3 needs to be above 16 in the sorted portion. We have to make space for 3, so we move 16 down one space.

We want to look at the next to last item in the sorted portion, but there is nothing to look at. We've run out of array, so we copy the item to be inserted into the top or first place in the array.

We reset the pointer to the bottom of the sorted portion, and we now want to get the next value to be inserted.

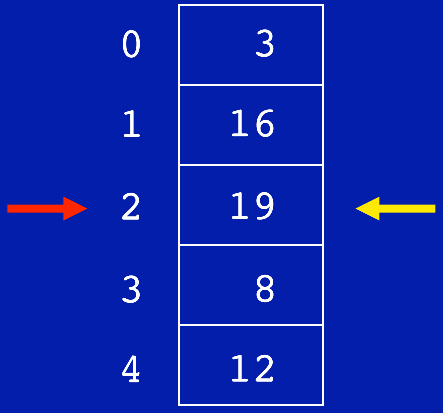
Insertion sort



Is 19 less than 16? No. So we don’t need to look any further; 19 is where it should be.

The value to be inserted is 19.

Insertion sort

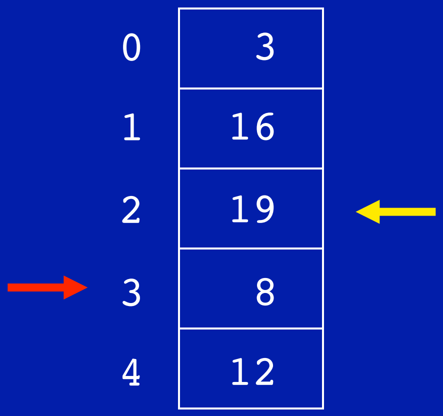


Is 19 less than 16? No. So we don’t need to look any further; 19 is where it should be.

Now the top three elements are the sorted portion,

The value to be inserted is 19.

Insertion sort

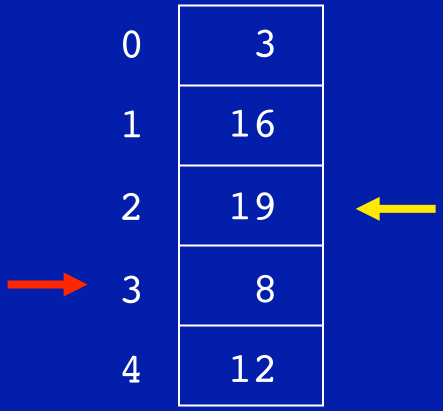


Is 19 less than 16? No. So we don’t need to look any further; 19 is where it should be.

Now the top three elements are the sorted portion, and we store the next element to be inserted into the sorted portion.

The value to be inserted is 8.

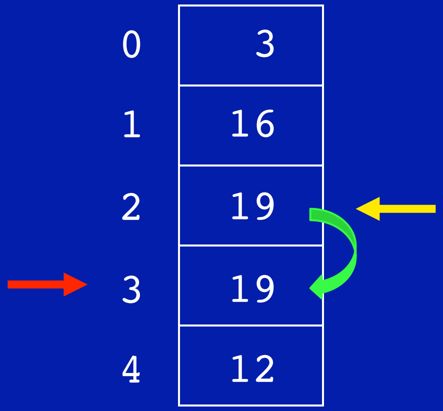
Insertion sort



Is 8 less than 19?

The value to be inserted is 8.

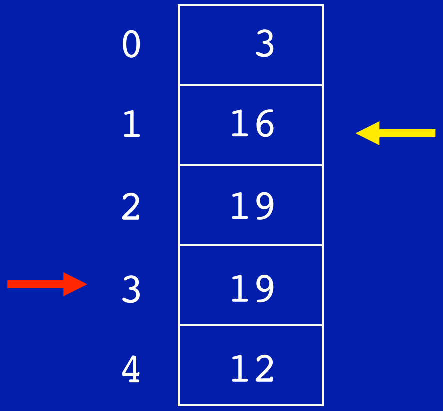
Insertion sort



Is 8 less than 19? Yes, so 19 moves down.

The value to be inserted is 8.

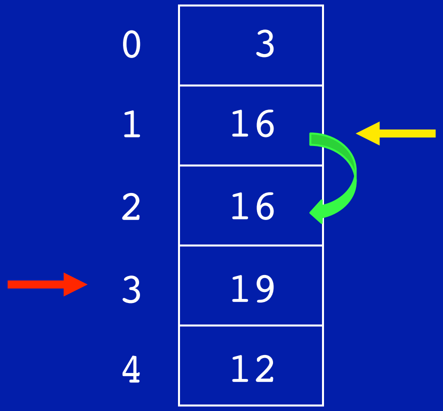
Insertion sort



Is 8 less than 19? Yes, so 19 moves down.
Is 8 less than 16?

The value to be inserted is 8.

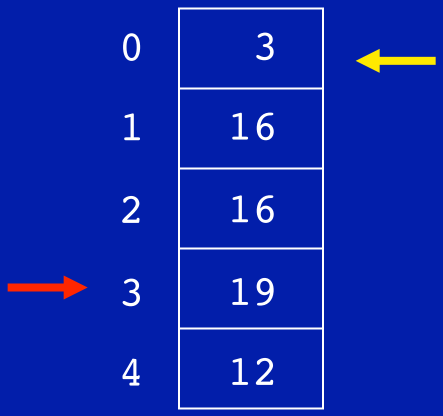
Insertion sort



Is 8 less than 19? Yes, so 19 moves down.
Is 8 less than 16? Yes, so 16 moves down.

The value to be inserted is 8.

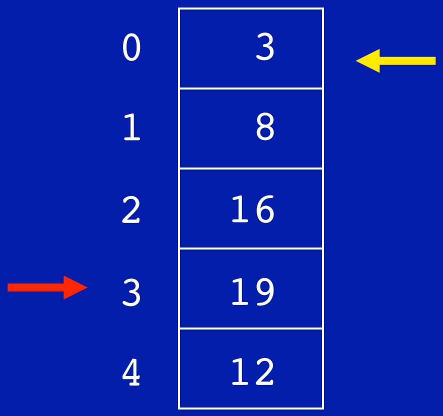
Insertion sort



Is 8 less than 19? Yes, so 19 moves down.
Is 8 less than 16? Yes, so 16 moves down.
Is 8 less than 3?

The value to be inserted is 8.

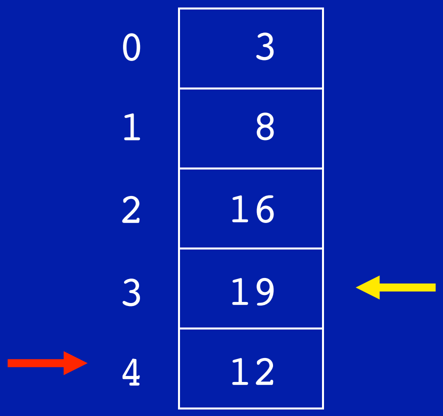
Insertion sort



Is 8 less than 19? Yes, so 19 moves down. Is 8 less than 16? Yes, so 16 moves down. Is 8 less than 3? No, so we copy 8 into the space that was last vacated (i.e., the empty space that's waiting to hold the value to be inserted).

The value to be inserted is 8.

Insertion sort

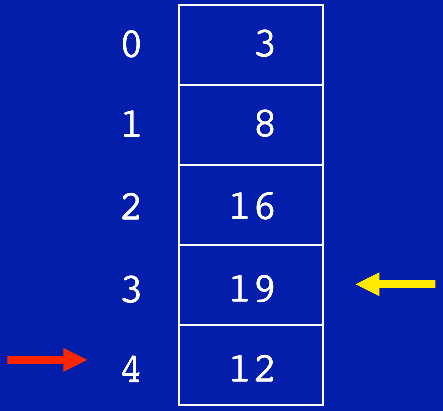


Is 8 less than 19? Yes, so 19 moves down.
Is 8 less than 16? Yes, so 16 moves down.
Is 8 less than 3? No, so we copy 8 into the space that was last vacated (i.e., the empty space that's waiting to hold the value to be inserted).

Reset the bottom of the sorted portion and the next value to be inserted.

The value to be inserted is 12.

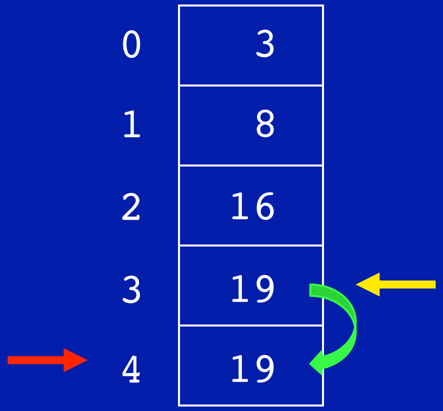
Insertion sort



Is 12 less than 19?

The value to be inserted is 12.

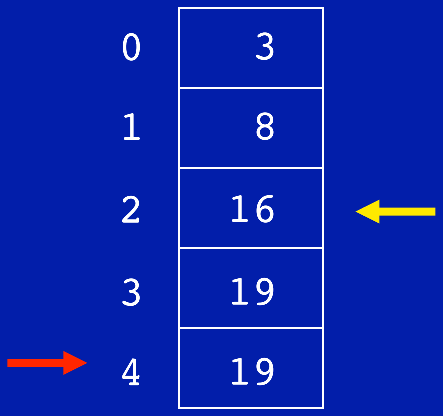
Insertion sort



Is 12 less than 19? Yes, so 19 moves down.

The value to be inserted is 12.

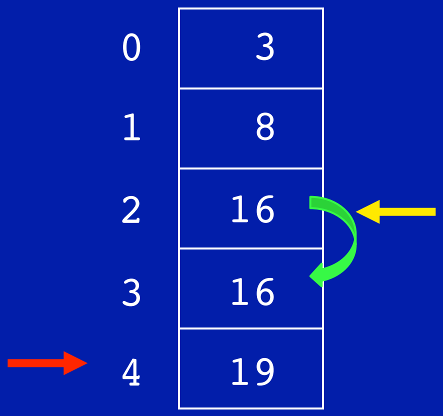
Insertion sort



Is 12 less than 16?

The value to be inserted is 12.

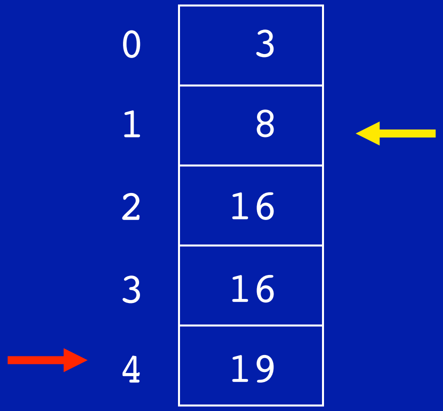
Insertion sort



Is 12 less than 16? Yes, so 16 moves down.

The value to be inserted is 12.

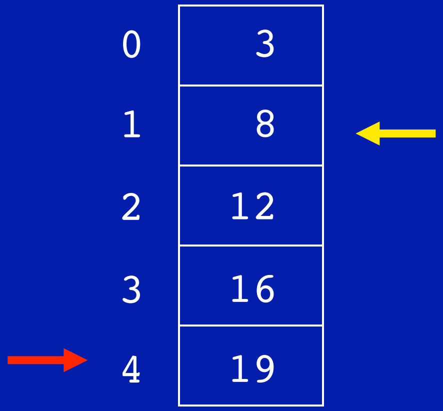
Insertion sort



Is 12 less than 8?

The value to be inserted is 12.

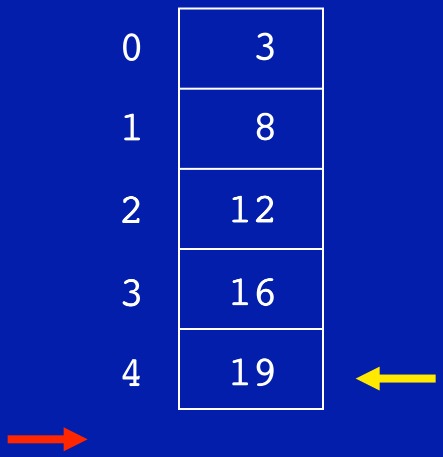
Insertion sort



Is 12 less than 8? No, so we copy 12 into the last vacated space.

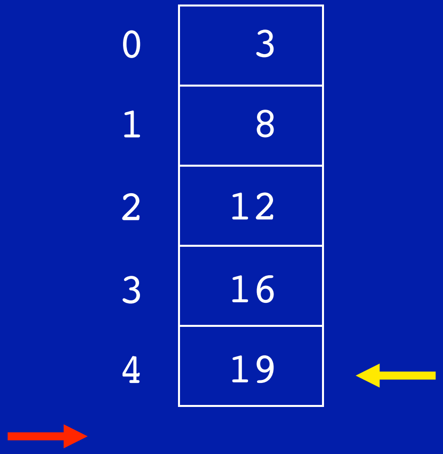
The value to be inserted is 12.

Insertion sort



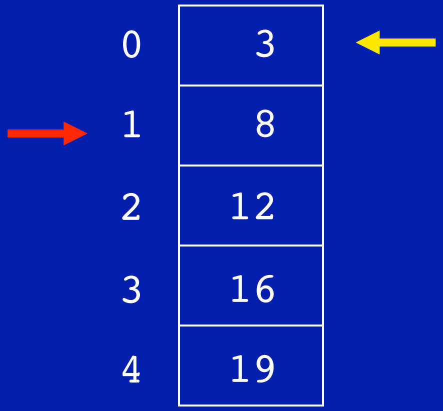
When we try to reset the pointers now, we can see that we're done. The entire array is sorted.

Insertion sort



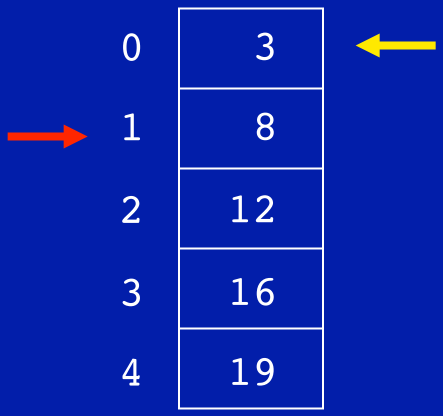
What input would let insertion sort show off its best possible performance?

Insertion sort



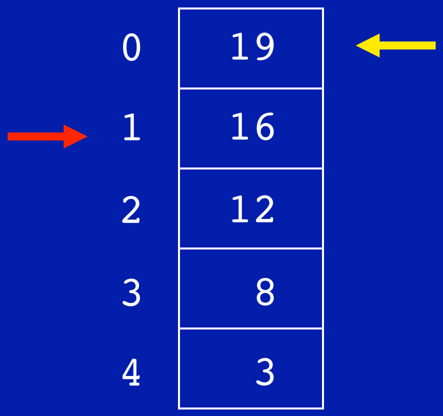
What input would let insertion sort show off its best possible performance? An array that's already sorted. How many comparisons would be made? How many elements would be moved?

Insertion sort



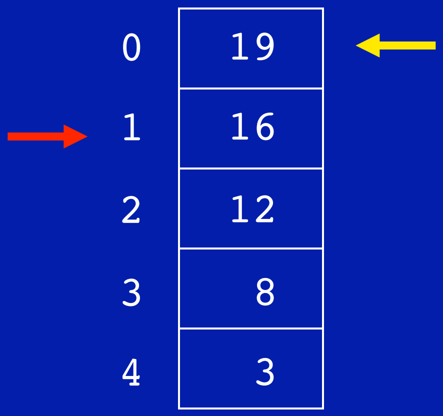
What input would be the worst possible case for insertion sort?

Insertion sort



What input would be the worst possible case for insertion sort? Sorted, but reversed. How many comparisons would be made? How many elements would be moved?

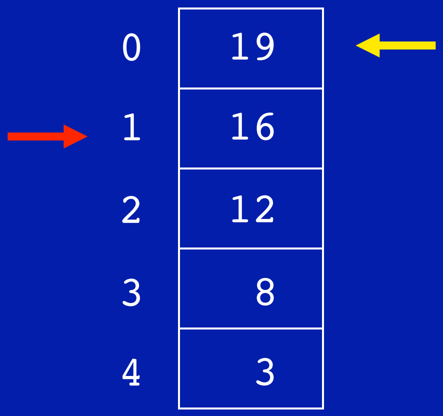
Insertion sort



What input would be the worst possible case for insertion sort? Sorted, but reversed. How many comparisons would be made? How many elements would be moved?

So what's the worst case Big-O for insertion sort?

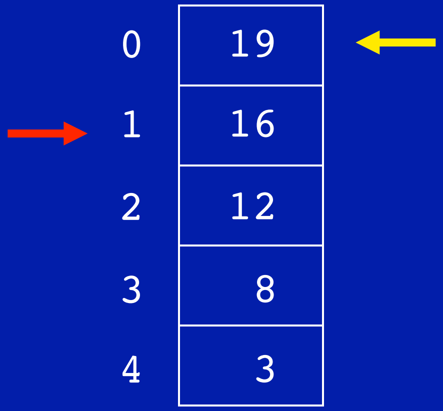
Insertion sort



What input would be the worst possible case for insertion sort? Sorted, but reversed. How many comparisons would be made? How many elements would be moved?

So what's the worst case Big-O for insertion sort? O(n^2), or pretty much the same as for selection sort.

Insertion sort



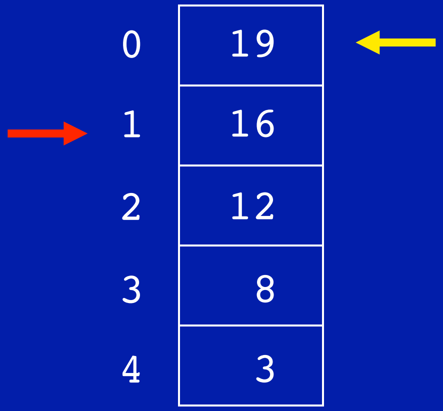
Pseudocode algorithm:

```
1. for each array element from the second
   element to the last element
2.     Insert the selected element where it
       where it belongs in the array by
       shifting all values larger than the
       selected element back by one location
```

Your textbook has a more detailed pseudocode algorithm for insertion sort that behaves sort of like what we just did.

Here's some C++ code that might do what we we're talking about too:

Insertion sort



```
#include <iostream>
using namespace std;

void insertSort(int array[], int n)
{
    int i, j;
    int key;

    for(i = 1; i < n; i++)
    {
        key = array[i];
        j = i - 1;
        while(j >= 0 && array[j] > key)
        {
            array[j + 1] = array[j];
            j--;
        }
    array[j + 1] = key;
    }
}
```

Questions?

Mergesort

Mergesort takes a different approach to the problem. It falls in the class of algorithms called “divide and conquer”.

In mergesort, the problem space is continually split in half by applying the algorithm recursively to each half, until the base case is reached.

Mergesort

Mergesort takes a different approach to the problem. It falls in the class of algorithms called “divide and conquer”.

In mergesort, the problem space is continually split in half by applying the algorithm recursively to each half, until the base case is reached.

A simple algorithm for mergesort is:

```
mergesort(unsorted_list)
    Divide the unsorted_list into two sublists of half the size of the unsorted_list
    Apply mergesort to each of the unsorted sublists
    Merge those two now-sorted sublists back into one sorted list
```

where ‘list’ could be any sequential data structure

Mergesort

If you're working with linked lists as in Racket/Scheme, the resulting code is simple:

```
(define (mergesort mlist)
  (cond [(< (length mlist) 2) mlist]
        [else
          (merge (mergesort (firsthalf mlist))
                 (mergesort (lasthalf mlist)))]))

(define (merge lst1 lst2)
 (cond [(empty? lst1) lst2]
       [(empty? lst2) lst1]
       [(< (first lst1) (first lst2))
        (cons (first lst1) (merge (rest lst1) lst2))]
       [else (cons (first lst2) (merge lst1 (rest lst2)))]))
```

where firsthalf and lasthalf are functions we would define to return the first and last halves of a list, respectively

Mergesort

Here’s how these functions would sort a list of integers:

```
(7  2  8  5  1  3  6  4)
```

Mergesort

Here’s how these functions would sort a list of integers:

$$\begin{tikzcd}
 & (7\ \ 2\ \ 8\ \ 5\ \ 1\ \ 3\ \ 6\ \ 4) \arrow[dl] \arrow[dr] & \\
(7\ \ 2\ \ 8\ \ 5) & & (1\ \ 3\ \ 6\ \ 4)
\end{tikzcd}$$

`mergesort`

Mergesort

Here’s how these functions would sort a list of integers:

$$\begin{tikzcd}
 & & & (7\ 2\ 8\ 5\ 1\ 3\ 6\ 4) \arrow[dll] \arrow[drr] & & & & \\
 & (7\ 2\ 8\ 5) \arrow[dl] \arrow[dr] & & & & (1\ 3\ 6\ 4) \arrow[dl] \arrow[dr] & & \texttt{mergesort} \\
(7\ 2) & & (8\ 5) & & (1\ 3) & & (6\ 4) & \texttt{mergesort}
\end{tikzcd}$$

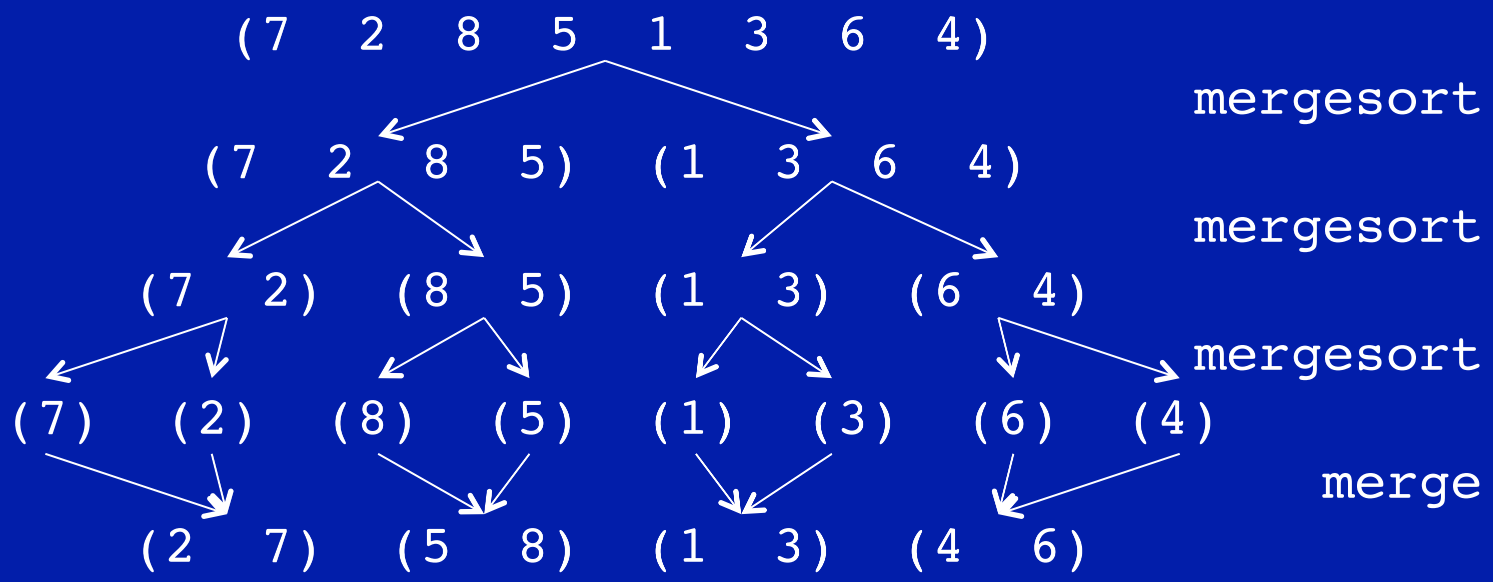
Mergesort

Here’s how these functions would sort a list of integers:

$$\begin{tikzcd}
&&&&&&& (7\ 2\ 8\ 5\ 1\ 3\ 6\ 4) \arrow[dllll] \arrow[drrrr] &&&&&&&&& \texttt{mergesort} \\
&&& (7\ 2\ 8\ 5) \arrow[dll] \arrow[drr] &&&&&&&& (1\ 3\ 6\ 4) \arrow[dll] \arrow[drr] &&&&& \texttt{mergesort} \\
& (7\ 2) \arrow[dl] \arrow[dr] &&&& (8\ 5) \arrow[dl] \arrow[dr] &&&& (1\ 3) \arrow[dl] \arrow[dr] &&&& (6\ 4) \arrow[dl] \arrow[dr] &&& \texttt{mergesort} \\
(7) && (2) && (8) && (5) && (1) && (3) && (6) && (4) &&
\end{tikzcd}$$

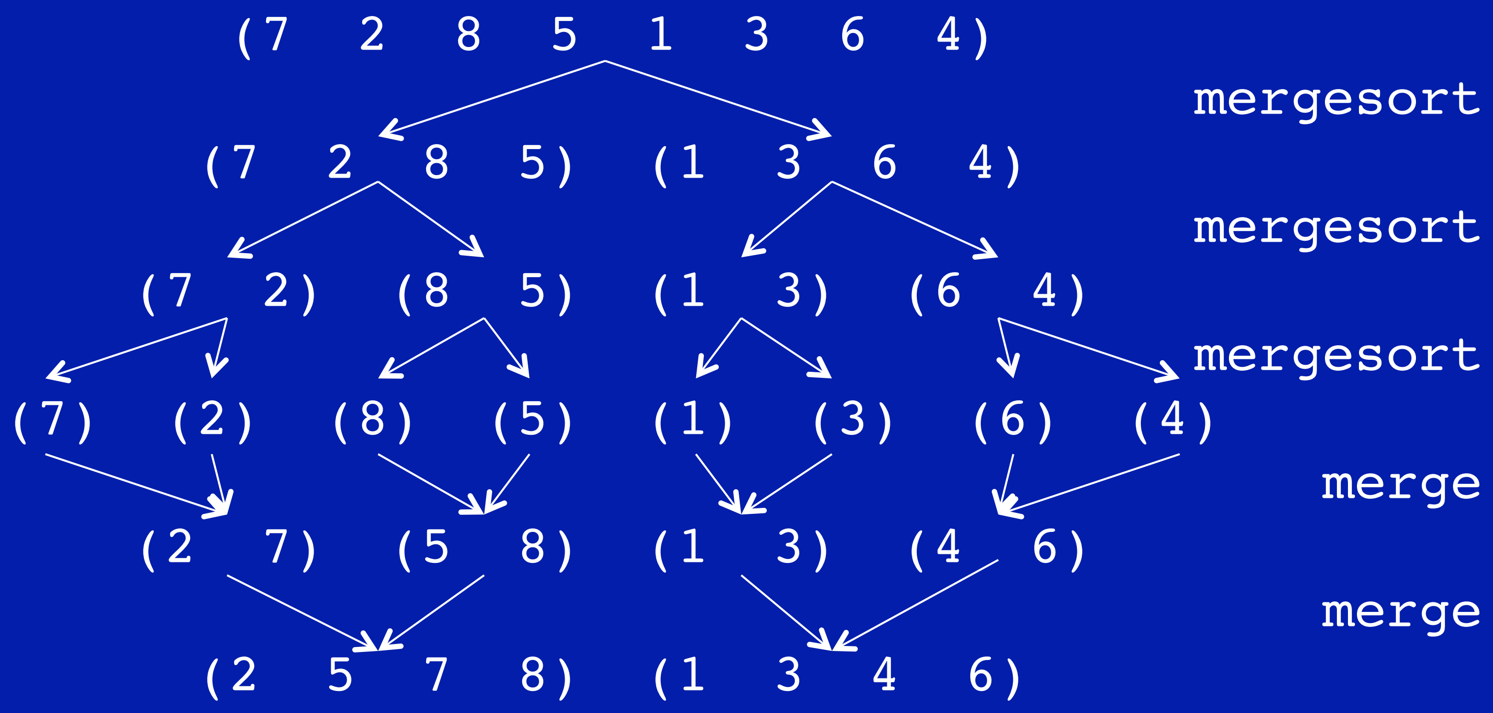
Mergesort

Here's how these functions would sort a list of integers:



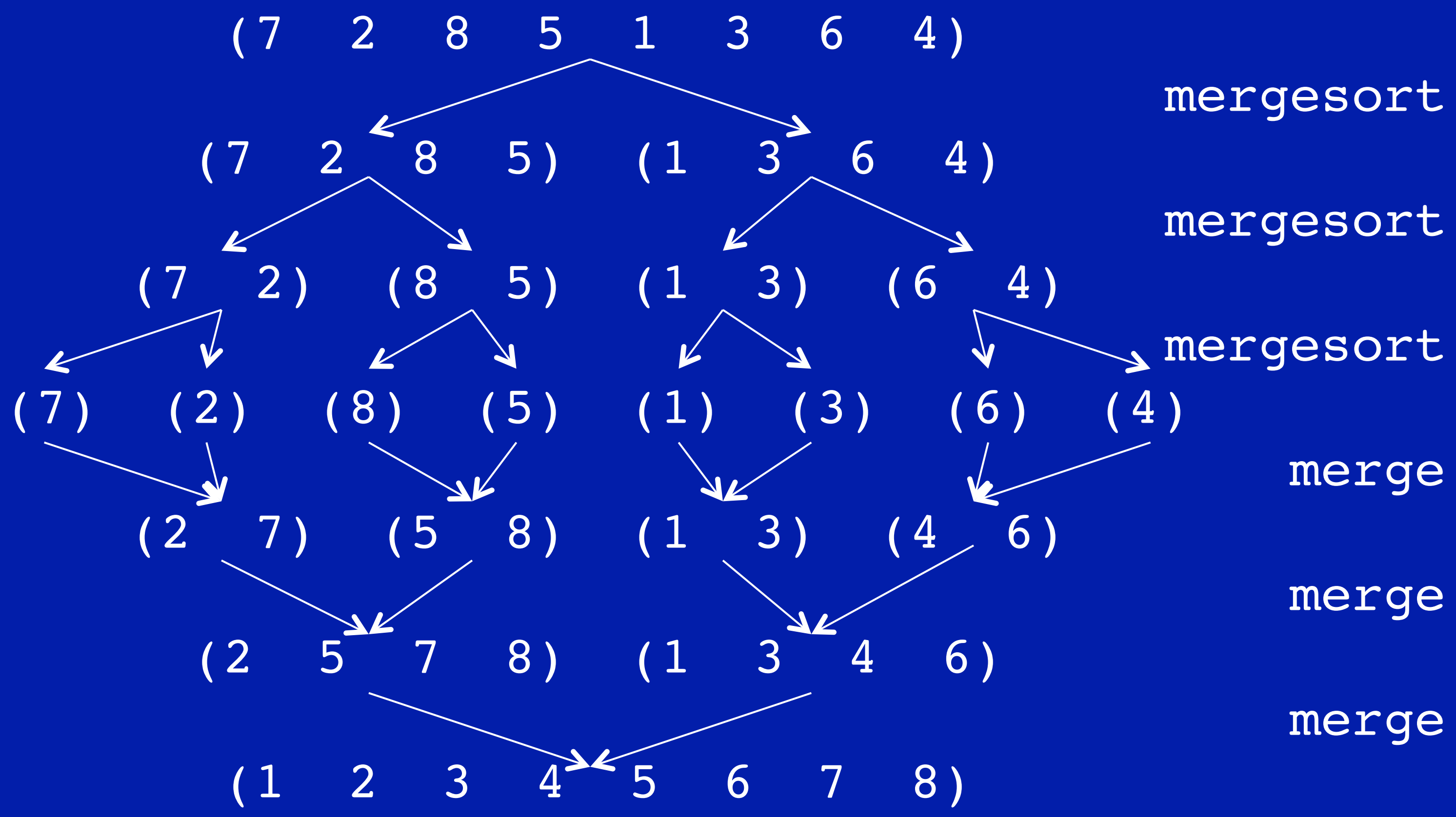
Mergesort

Here's how these functions would sort a list of integers:



Mergesort

Here’s how these functions would sort a list of integers:



Mergesort

Mergesort with an array, without making numerous copies of lists, gets a somewhat different implementation. The typical implementation involves creating a temporary array to hold the results of merging two sorted subarrays (demarcated in the original array). Then that sorted result is copied back to the original array.

Why the need for a temporary array? Because performing mergesort in place really goobers things up. Feel free to try it on your own.

Mergesort

Here's an imperfect trace of mergesorting just four values in an array (assume we've drawn little boxes to represent the arrays:

```
original           temp
[7  2  8  5]       _  _  _  _
```

Mergesort

Here’s an imperfect trace of mergesorting just four values in an array (assume we’ve drawn little boxes to represent the arrays:

original	temp
$\boxed{7\ \ 2\ \ 8\ \ 5}$	_ _ _ _
$\boxed{7\ \ 2}\ \ 8\ \ 5$	_ _ _ _

Mergesort

Here’s an imperfect trace of mergesorting just four values in an array (assume we’ve drawn little boxes to represent the arrays:

original	temp
$\boxed{7\ \ 2\ \ 8\ \ 5}$	_ _ _ _
$\boxed{7\ \ 2}$ 8 5	_ _ _ _
$\boxed{7}$ 2 8 5	_ _ _ _

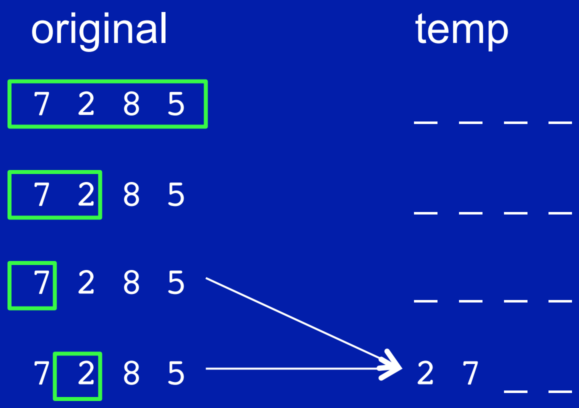
Mergesort

Here’s an imperfect trace of mergesorting just four values in an array (assume we’ve drawn little boxes to represent the arrays:

original	temp
$\boxed{7\ \ 2\ \ 8\ \ 5}$	_ _ _ _
$\boxed{7\ \ 2}\ \ 8\ \ 5$	_ _ _ _
$\boxed{7}\ \ 2\ \ 8\ \ 5$	_ _ _ _
$7\ \boxed{2}\ \ 8\ \ 5$	_ _ _ _

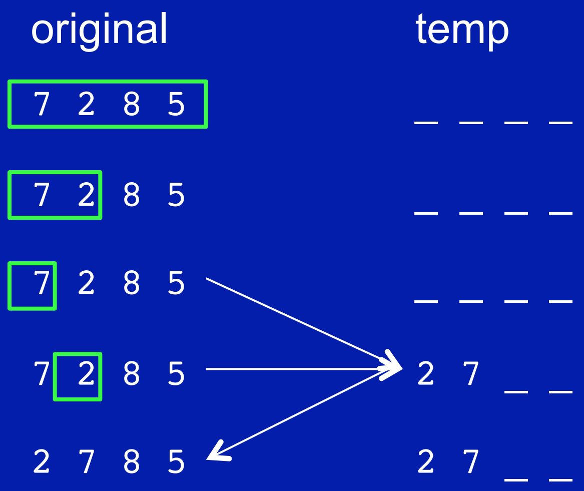
Mergesort

Here's an imperfect trace of mergesorting just four values in an array (assume we've drawn little boxes to represent the arrays:



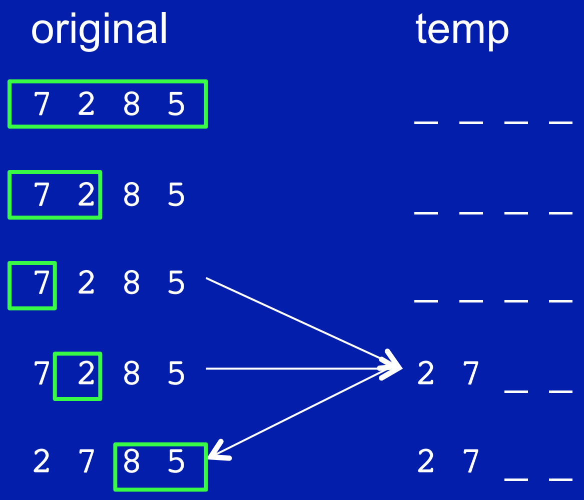
Mergesort

Here's an imperfect trace of mergesorting just four values in an array (assume we've drawn little boxes to represent the arrays:



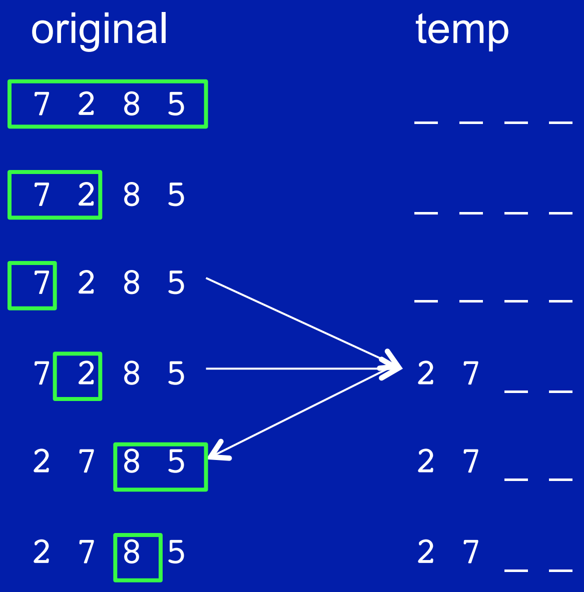
Mergesort

Here's an imperfect trace of mergesorting just four values in an array (assume we've drawn little boxes to represent the arrays:



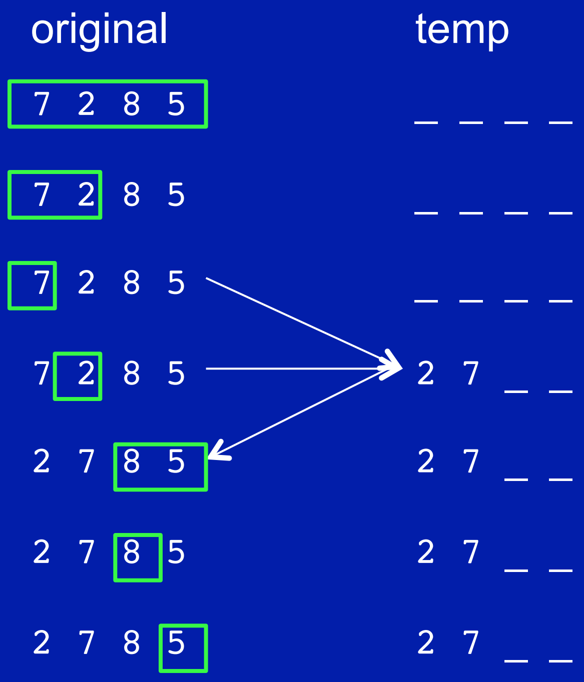
Mergesort

Here's an imperfect trace of mergesorting just four values in an array (assume we've drawn little boxes to represent the arrays:



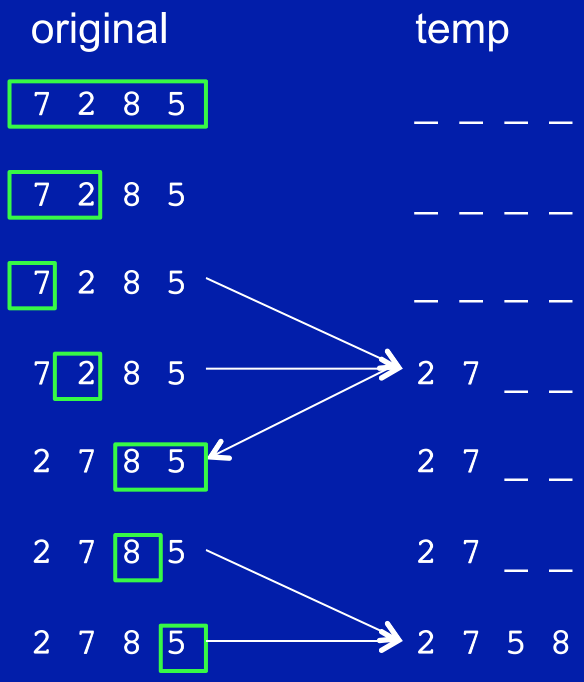
Mergesort

Here's an imperfect trace of mergesorting just four values in an array (assume we've drawn little boxes to represent the arrays:



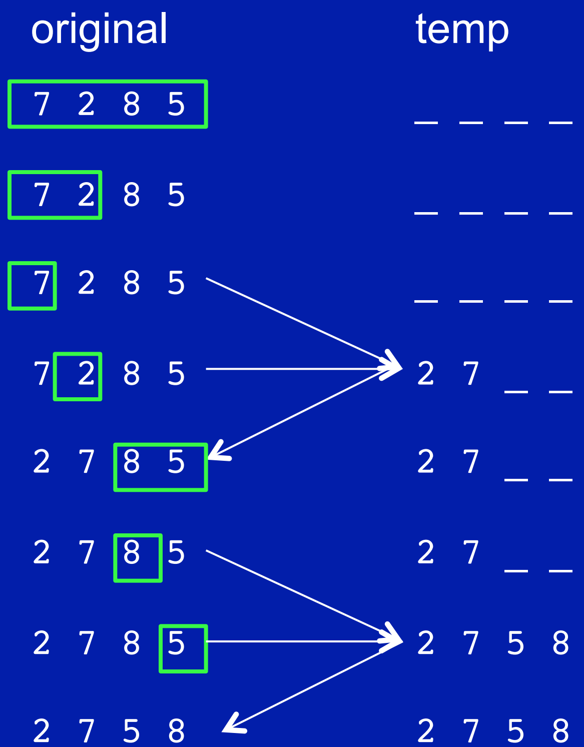
Mergesort

Here's an imperfect trace of mergesorting just four values in an array (assume we've drawn little boxes to represent the arrays:



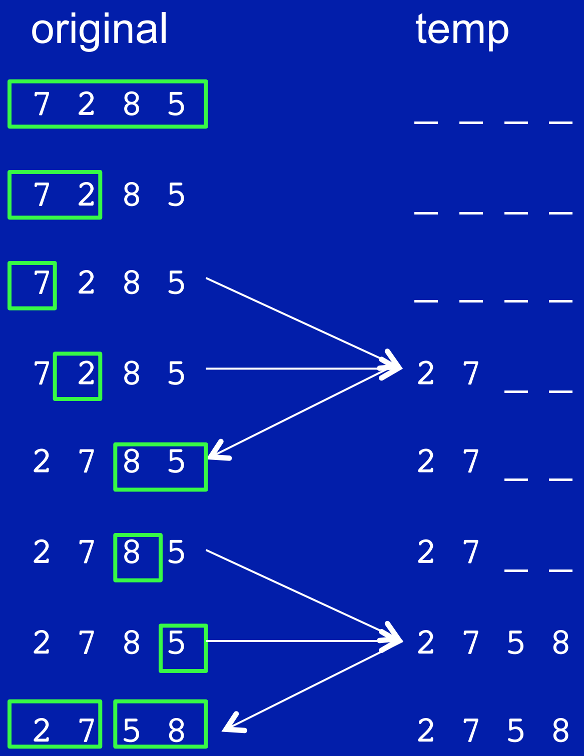
Mergesort

Here’s an imperfect trace of mergesorting just four values in an array (assume we’ve drawn little boxes to represent the arrays:



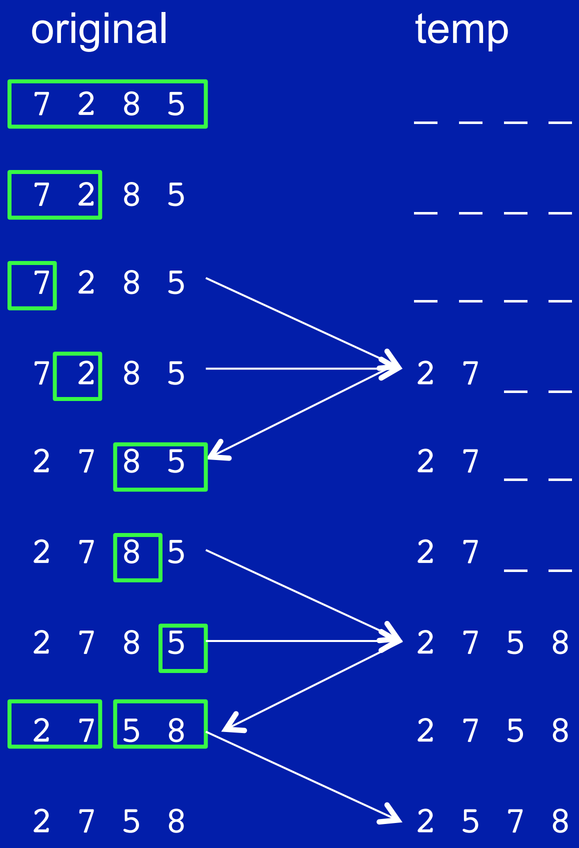
Mergesort

Here's an imperfect trace of mergesorting just four values in an array (assume we've drawn little boxes to represent the arrays:



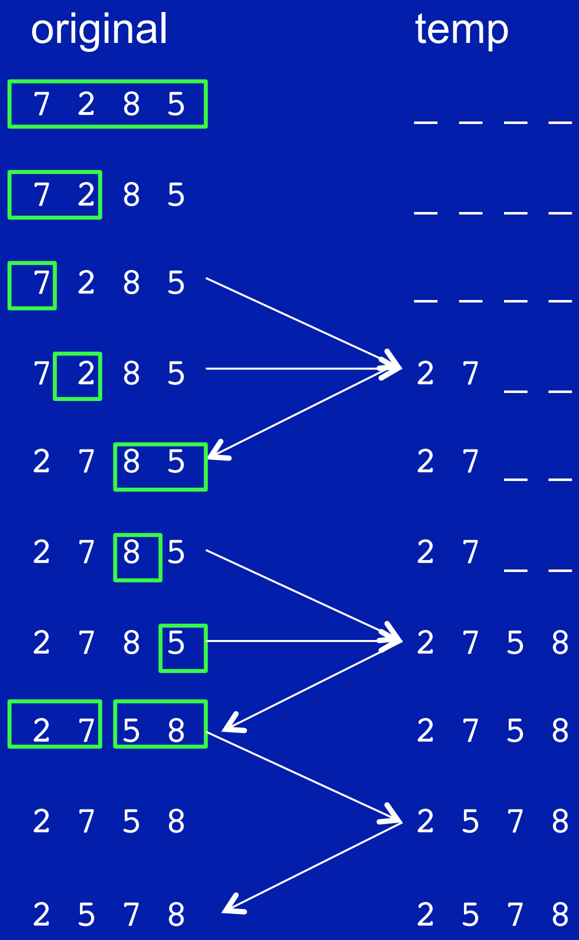
Mergesort

Here's an imperfect trace of mergesorting just four values in an array (assume we've drawn little boxes to represent the arrays:



Mergesort

Here's an imperfect trace of mergesorting just four values in an array (assume we've drawn little boxes to represent the arrays:



Mergesort

What are we imperfectly tracing? Code like this:

```
void mergesort(int array[], int n)
{
    int *tmp = new int[n];
    msort(array, 0, n - 1, tmp);
    delete[] tmp;
}

void msort(int array[], int lo, int hi, int tmp[])
{
    if (lo >= hi) return;
    int mid = (lo + hi)/2;
    msort(array, lo, mid, tmp);
    msort(array, mid + 1, hi, tmp);
    merge(array, lo, mid, hi, tmp);
}
```

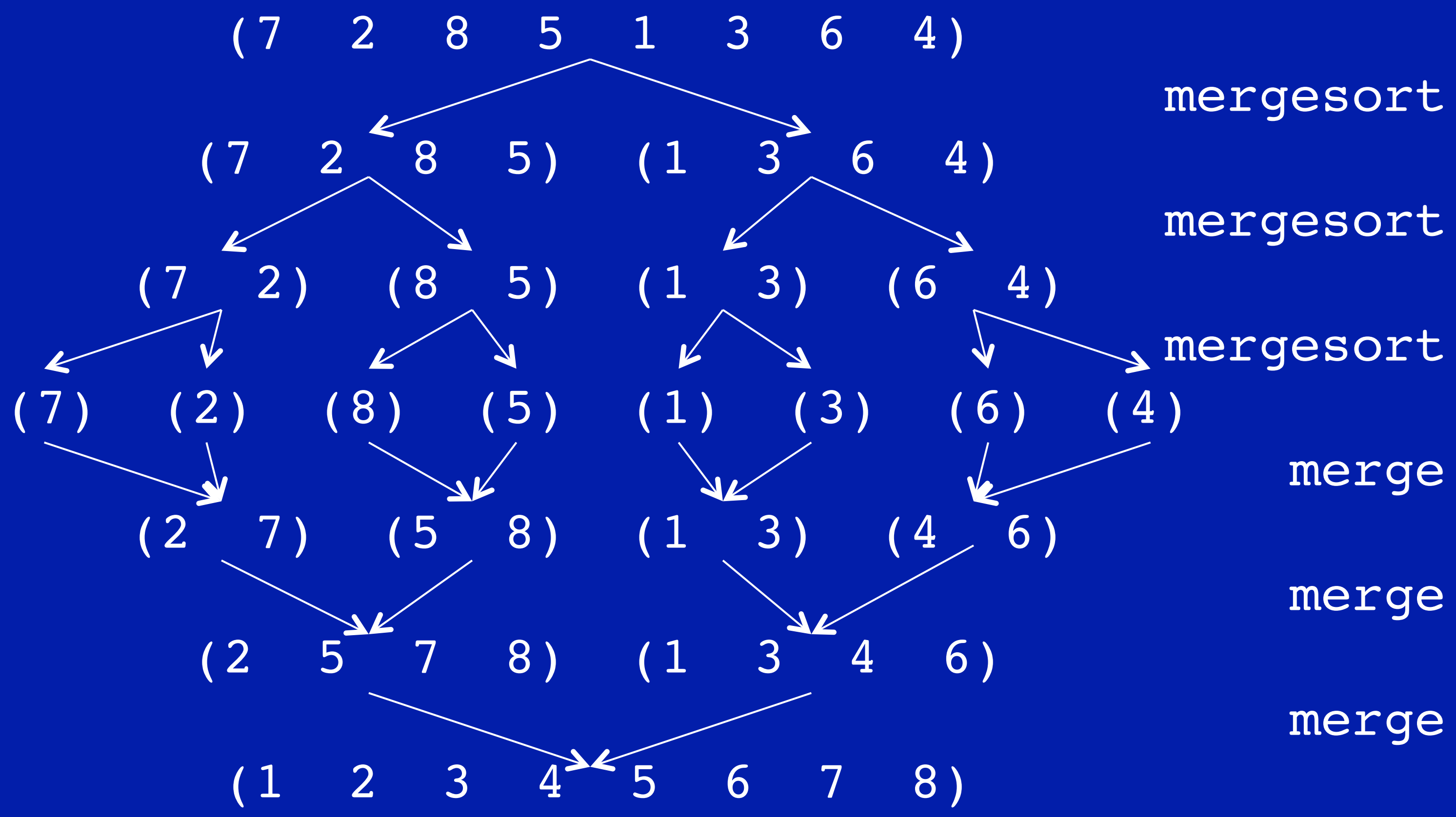
Mergesort

What are we imperfectly tracing? Code like this:

```
void merge(int array[], int lo, int mid, int hi, int tmp[])
{
    int a = lo, b = mid + 1;
    for(int k = lo; k <= hi; k++)
    {
        if(a <= mid && (b > hi || array[a] < array[b]))
            tmp[k] = array[a++];
        else
            tmp[k] = array[b++];
    }
    for(int k = lo; k <= hi; k++)
        array[k] = tmp[k];
}
```

Mergesort

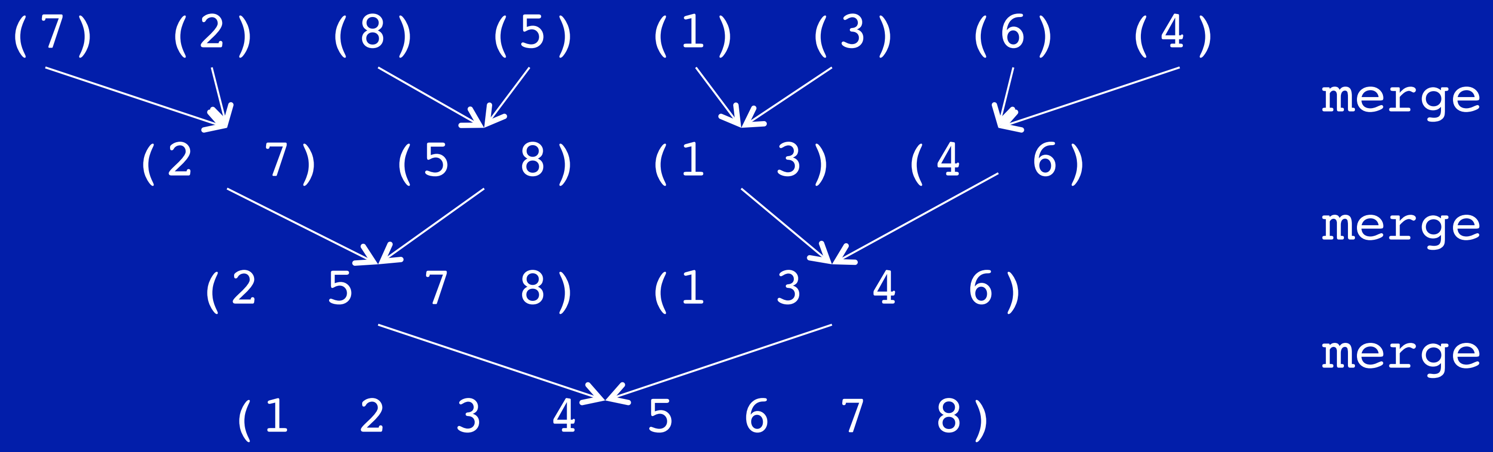
What kind of time complexity are we dealing with here?



Mergesort

What kind of time complexity are we dealing with here?

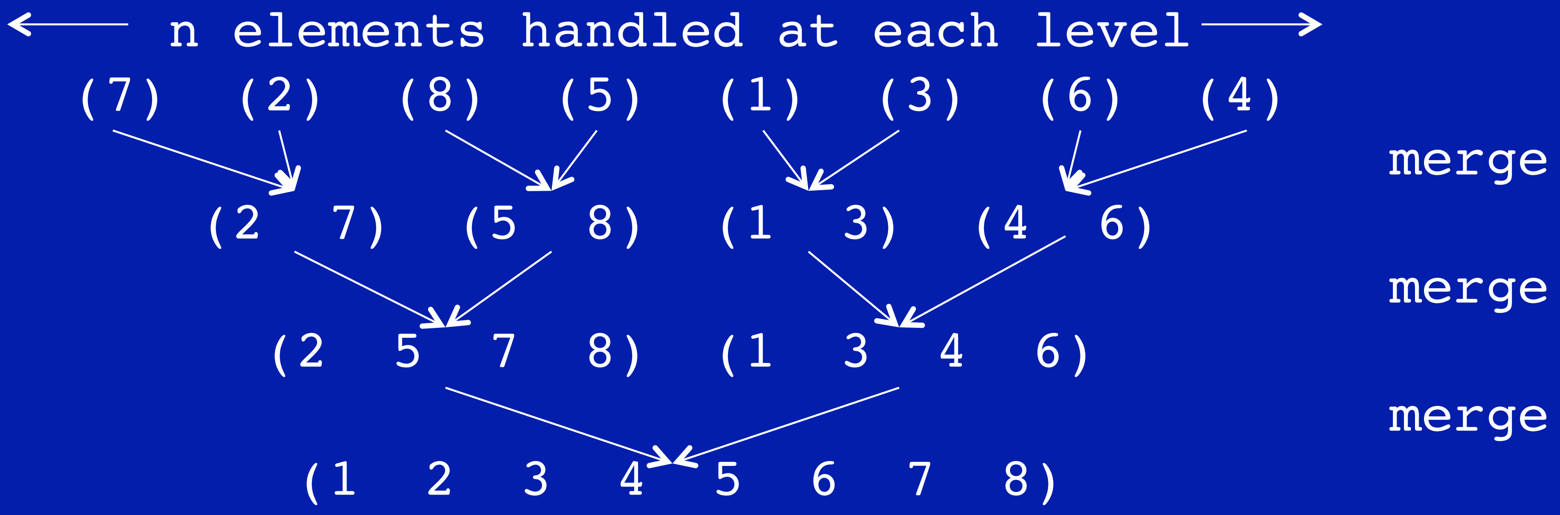
Let's just look at half the work. Multiply by two if you're concerned.



Mergesort

What kind of time complexity are we dealing with here?

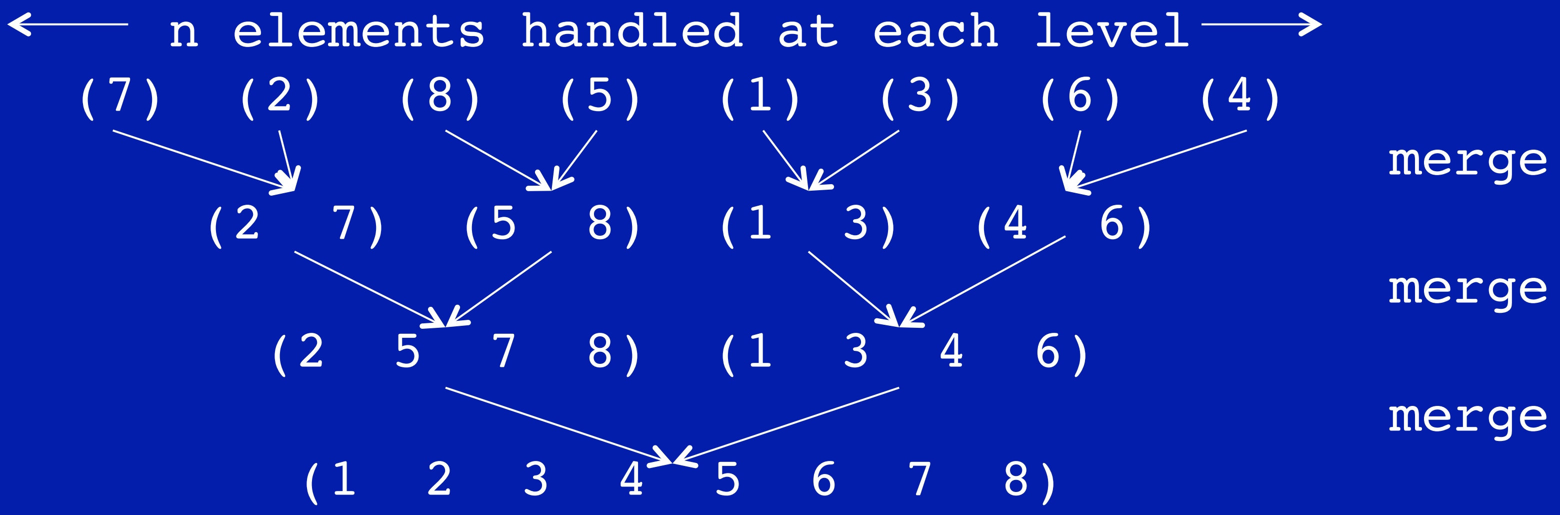
If were working with arrays, not lists, those last three lines represent the movement of n elements from the original array to the temporary array and back again, along with the required comparisons to maintain sorted order while merging. So each round of merging is O(n).



Mergesort

What kind of time complexity are we dealing with here?

There are 3 rounds of merging when n = 8. How many rounds of merging would be required if n = 16? n = 32? What does that tell you?

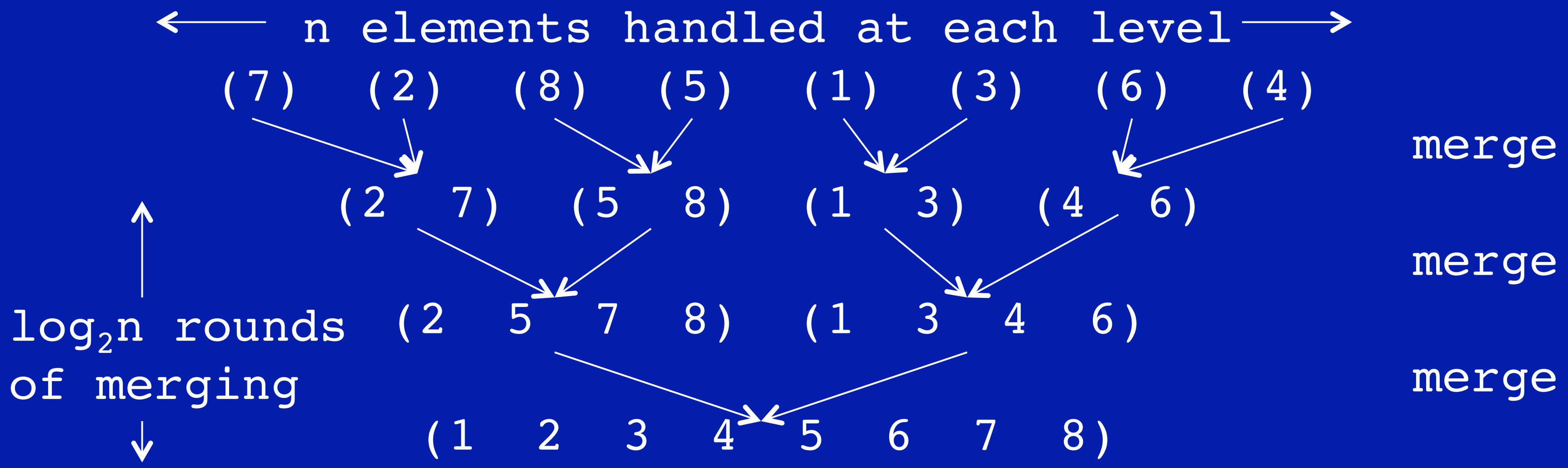


Mergesort

What kind of time complexity are we dealing with here?

There are 3 rounds of merging when n = 8. How many rounds of merging would be required if n = 16? n = 32? What does that tell you?

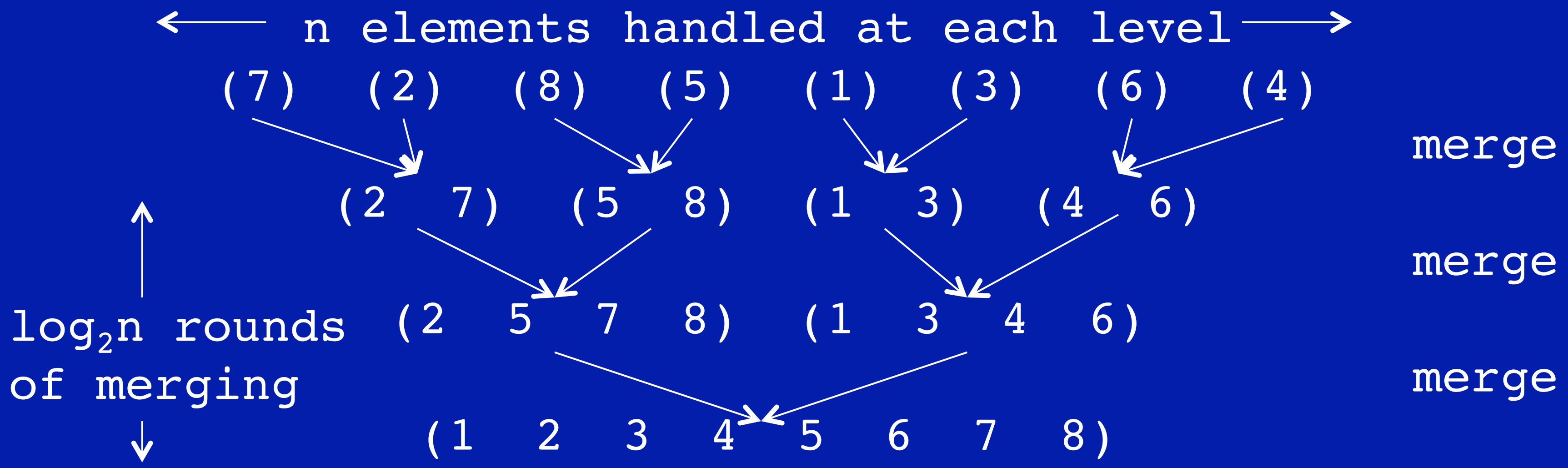
How does O(lg n) sound?



Mergesort

What kind of time complexity are we dealing with here?

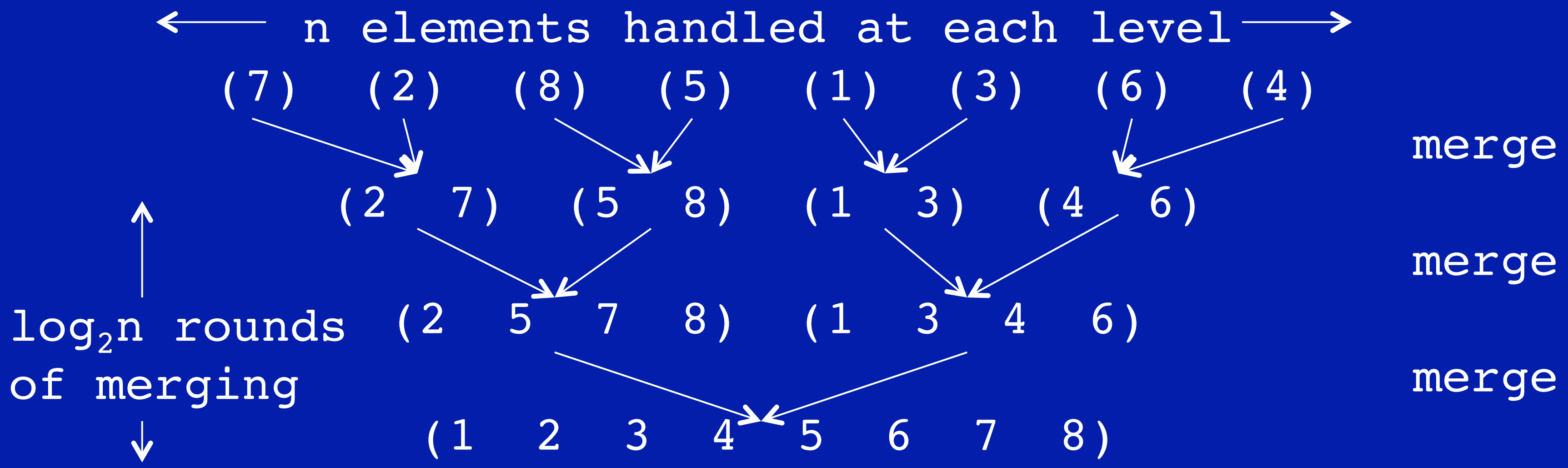
Worst case time complexity for mergesort is O(n lg n).
What about space complexity for the array-based approach?



Mergesort

What kind of time complexity are we dealing with here?

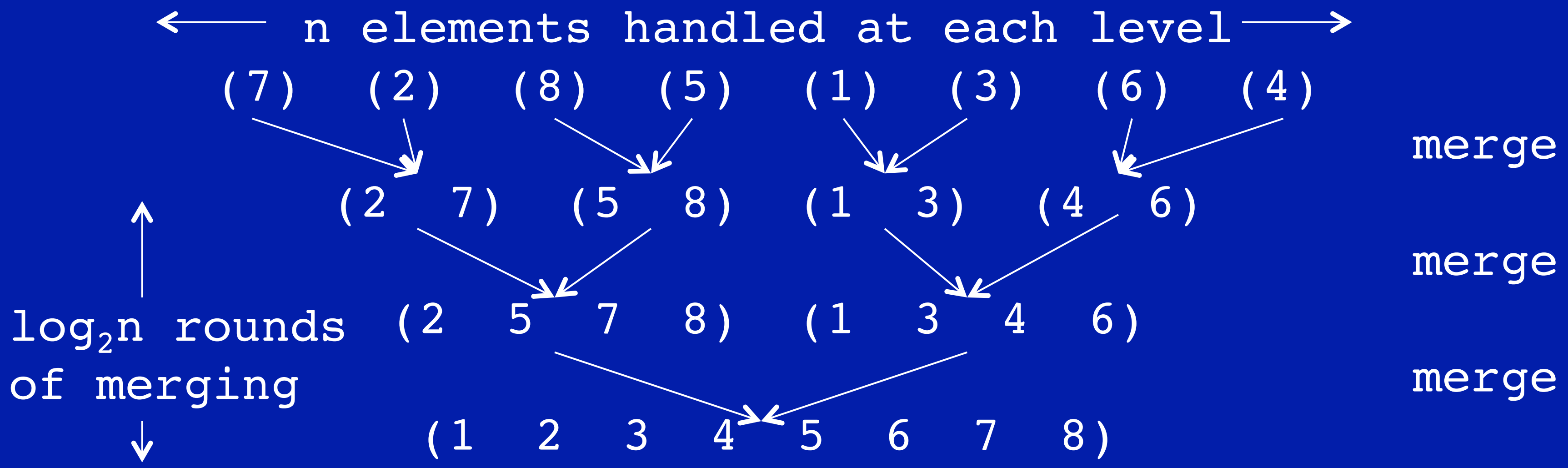
Worst case time complexity for mergesort is O(n lg n).
What about space complexity for the array-based approach?
The space requirements are two arrays of size n, so O(n).



Mergesort

What kind of time complexity are we dealing with here?

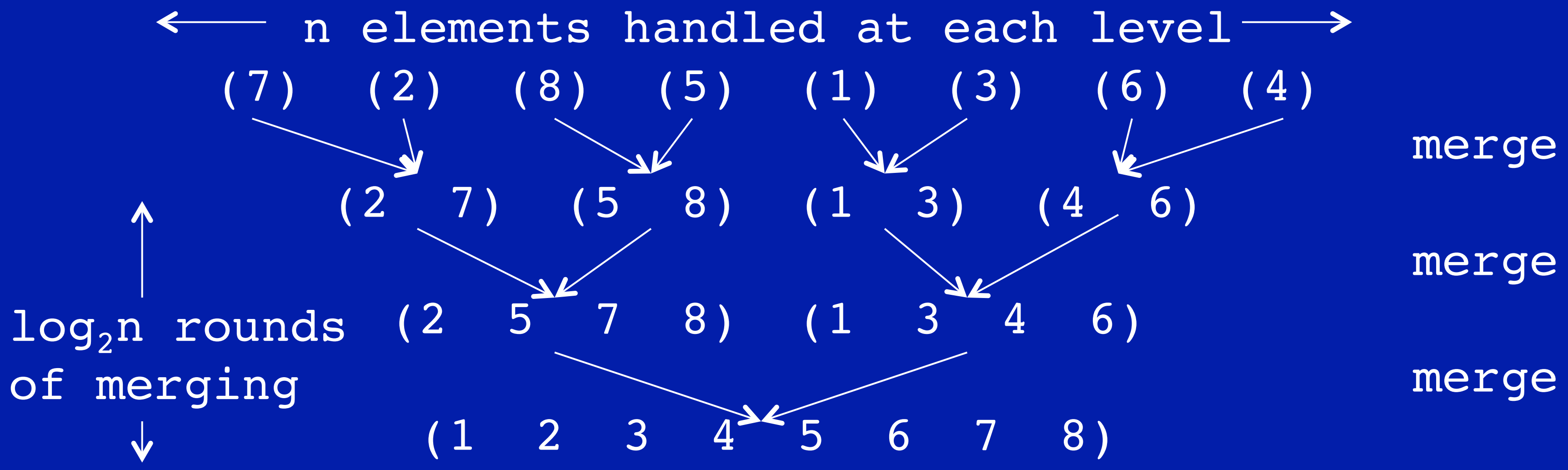
Worst case time complexity for mergesort is O(n lg n).
What about space complexity for the list-based approach with all the list copies?



Mergesort

What kind of time complexity are we dealing with here?

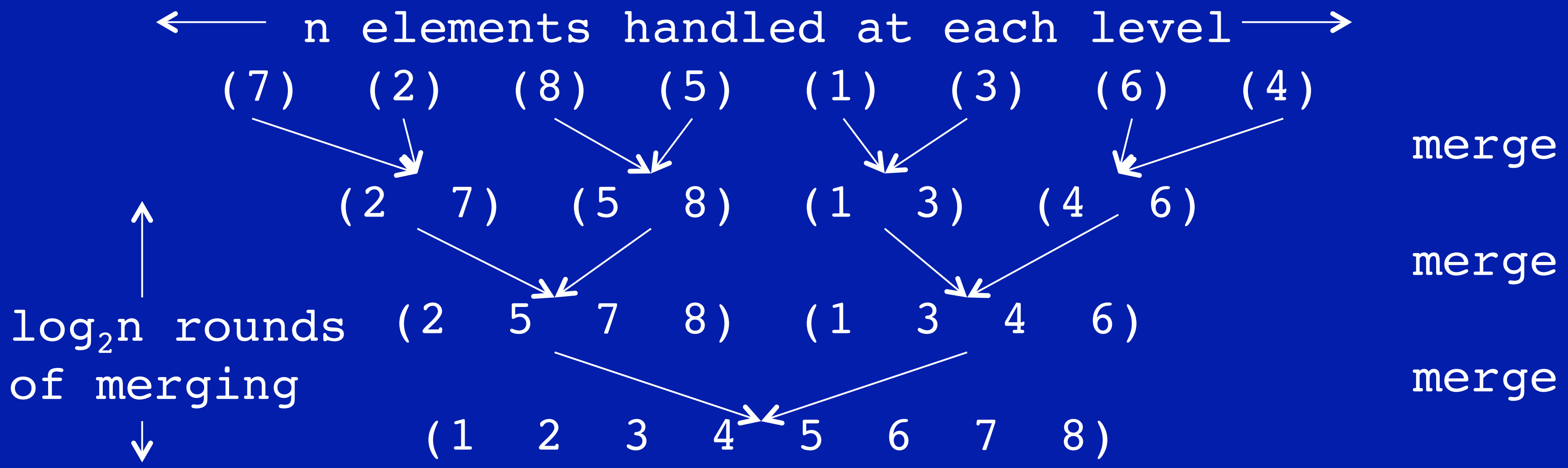
Worst case time complexity for mergesort is O(n lg n).
What about space complexity for the list-based approach with all the list copies? How many new cons cells were generated at each level of splitting? n new cons cells at each level.



Mergesort

What kind of time complexity are we dealing with here?

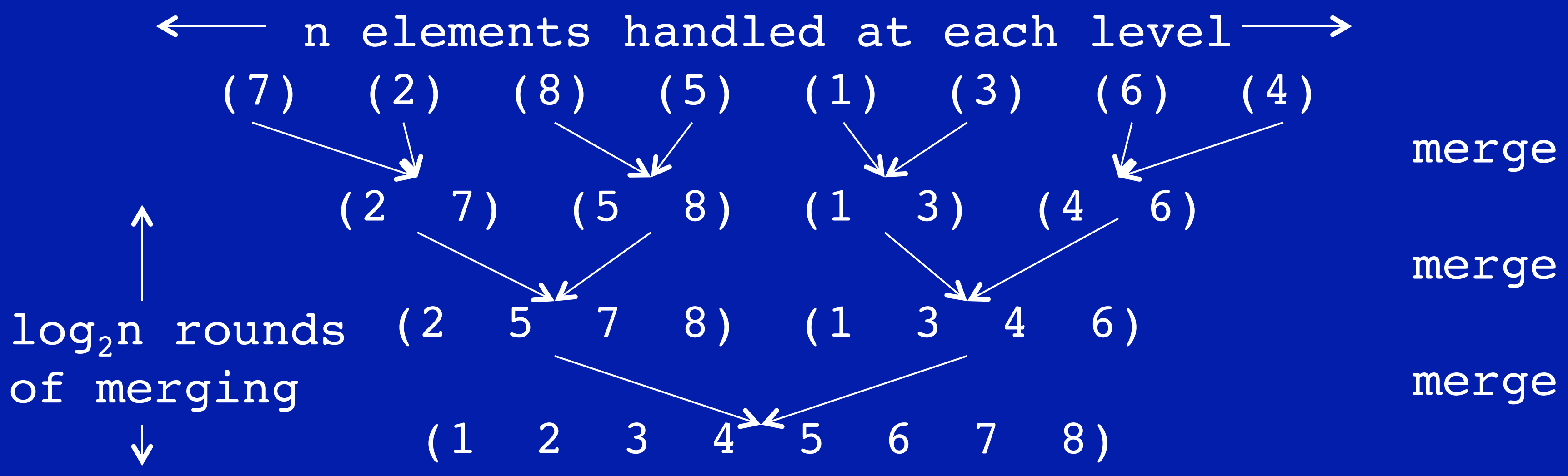
Worst case time complexity for mergesort is O(n lg n).
What about space complexity for the list-based approach with all the list copies? How many new cons cells were generated at each level of merging? n new cons cells at each level.



Mergesort

What kind of time complexity are we dealing with here?

Worst case time complexity for mergesort is O(n lg n).
What about space complexity for the list-based approach with all the list copies? n new cons cells at each level and 2 lg n levels implies O(n lg n) worst case space complexity.



Questions?

Quicksort

If we want to sort big sequences quickly, without the extra memory and copying back and forth of mergesort, the answer is quicksort. In practice, it is the fastest sorting algorithm known. While its worst-case run time is $O(n^2)$, its average run time is $O(n \lg n)$.

A simple algorithm for mergesort is:

```
quicksort(unsorted_list)
    Choose one element to be the pivot
    Partition (i.e. reorder) the list so that all elements < the pivot are to the left
      of the pivot, and all elements > the pivot are to the right of the pivot
    Apply quicksort recursively to the sublist to the left and the sublist to the right
```

where 'list' could be any sequential data structure

Quicksort

Here's an imperfect trace of quicksort on a small array (again assume we've drawn little boxes to represent the arrays:

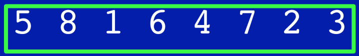
array

```
5 8 1 6 4 7 2 3
```

Quicksort

Here's an imperfect trace of quicksort on a small array (again assume we've drawn little boxes to represent the arrays:

array



Quicksort

Here's an imperfect trace of quicksort on a small array (again assume we've drawn little boxes to represent the arrays:

array

```
5 8 1 6 4 7 2 3
```

Quicksort

Here's an imperfect trace of quicksort on a small array (again assume we've drawn little boxes to represent the arrays:

```
    array

    5 |8 1 6 4 7 2 3|

|1 4 2 3| 5 |8 6 7|
```

Quicksort

Here's an imperfect trace of quicksort on a small array (again assume we've drawn little boxes to represent the arrays:

array

$5\ \boxed{8\ \ 1\ \ 6\ \ 4\ \ 7\ \ 2\ \ 3}$

$\boxed{1\ \ 4\ \ 2\ \ 3}\ 5\ \boxed{8\ \ 6\ \ 7}$

$1\ \boxed{4\ \ 2\ \ 3}\ 5\ \boxed{8\ \ 6\ \ 7}$

Quicksort

Here’s an imperfect trace of quicksort on a small array (again assume we’ve drawn little boxes to represent the arrays:

array

$5\ \boxed{8\ \ 1\ \ 6\ \ 4\ \ 7\ \ 2\ \ 3}$

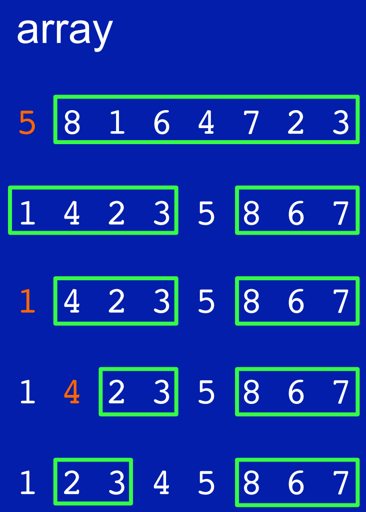
$\boxed{1\ \ 4\ \ 2\ \ 3}\ 5\ \boxed{8\ \ 6\ \ 7}$

$1\ \boxed{4\ \ 2\ \ 3}\ 5\ \boxed{8\ \ 6\ \ 7}$

$1\ \ 4\ \boxed{2\ \ 3}\ 5\ \boxed{8\ \ 6\ \ 7}$

Quicksort

Here’s an imperfect trace of quicksort on a small array (again assume we’ve drawn little boxes to represent the arrays:



Quicksort

Here’s an imperfect trace of quicksort on a small array (again assume we’ve drawn little boxes to represent the arrays:

array

$5\ \boxed{8\ 1\ 6\ 4\ 7\ 2\ 3}$

$\boxed{1\ 4\ 2\ 3}\ 5\ \boxed{8\ 6\ 7}$

$1\ \boxed{4\ 2\ 3}\ 5\ \boxed{8\ 6\ 7}$

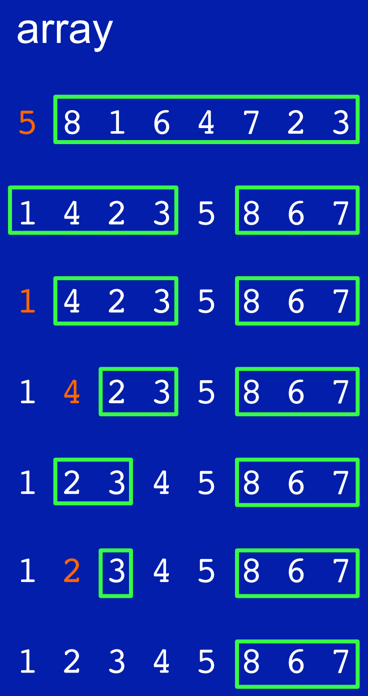
$1\ 4\ \boxed{2\ 3}\ 5\ \boxed{8\ 6\ 7}$

$1\ \boxed{2\ 3}\ 4\ 5\ \boxed{8\ 6\ 7}$

$1\ 2\ \boxed{3}\ 4\ 5\ \boxed{8\ 6\ 7}$

Quicksort

Here’s an imperfect trace of quicksort on a small array (again assume we’ve drawn little boxes to represent the arrays:



Quicksort

Here's an imperfect trace of quicksort on a small array (again assume we've drawn little boxes to represent the arrays:

array

$5\ \boxed{8\ \ 1\ \ 6\ \ 4\ \ 7\ \ 2\ \ 3}$

$\boxed{1\ \ 4\ \ 2\ \ 3}\ 5\ \boxed{8\ \ 6\ \ 7}$

$1\ \boxed{4\ \ 2\ \ 3}\ 5\ \boxed{8\ \ 6\ \ 7}$

$1\ \ 4\ \boxed{2\ \ 3}\ 5\ \boxed{8\ \ 6\ \ 7}$

$1\ \boxed{2\ \ 3}\ 4\ \ 5\ \boxed{8\ \ 6\ \ 7}$

$1\ \ 2\ \boxed{3}\ 4\ \ 5\ \boxed{8\ \ 6\ \ 7}$

$1\ \ 2\ \ 3\ \ 4\ \ 5\ \boxed{8\ \ 6\ \ 7}$

$1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 8\ \boxed{6\ \ 7}$

Quicksort

Here’s an imperfect trace of quicksort on a small array (again assume we’ve drawn little boxes to represent the arrays:

array

$5\ \boxed{8\ 1\ 6\ 4\ 7\ 2\ 3}$

$\boxed{1\ 4\ 2\ 3}\ 5\ \boxed{8\ 6\ 7}$

$1\ \boxed{4\ 2\ 3}\ 5\ \boxed{8\ 6\ 7}$

$1\ 4\ \boxed{2\ 3}\ 5\ \boxed{8\ 6\ 7}$

$1\ \boxed{2\ 3}\ 4\ 5\ \boxed{8\ 6\ 7}$

$1\ 2\ \boxed{3}\ 4\ 5\ \boxed{8\ 6\ 7}$

$1\ 2\ 3\ 4\ 5\ \boxed{8\ 6\ 7}$

$1\ 2\ 3\ 4\ 5\ 8\ \boxed{6\ 7}$

$1\ 2\ 3\ 4\ 5\ \boxed{6\ 7}\ 8$

Quicksort

Here's an imperfect trace of quicksort on a small array (again assume we've drawn little boxes to represent the arrays:

array

$5\ \boxed{8\ 1\ 6\ 4\ 7\ 2\ 3}$

$\boxed{1\ 4\ 2\ 3}\ 5\ \boxed{8\ 6\ 7}$

$1\ \boxed{4\ 2\ 3}\ 5\ \boxed{8\ 6\ 7}$

$1\ 4\ \boxed{2\ 3}\ 5\ \boxed{8\ 6\ 7}$

$1\ \boxed{2\ 3}\ 4\ 5\ \boxed{8\ 6\ 7}$

$1\ 2\ \boxed{3}\ 4\ 5\ \boxed{8\ 6\ 7}$

$1\ 2\ 3\ 4\ 5\ \boxed{8\ 6\ 7}$

$1\ 2\ 3\ 4\ 5\ 8\ \boxed{6\ 7}$

$1\ 2\ 3\ 4\ 5\ \boxed{6\ 7}\ 8$

$1\ 2\ 3\ 4\ 5\ 6\ \boxed{7}\ 8$

Quicksort

Here’s an imperfect trace of quicksort on a small array (again assume we’ve drawn little boxes to represent the arrays:

array

$5\ \boxed{8\ 1\ 6\ 4\ 7\ 2\ 3}$

$\boxed{1\ 4\ 2\ 3}\ 5\ \boxed{8\ 6\ 7}$

$1\ \boxed{4\ 2\ 3}\ 5\ \boxed{8\ 6\ 7}$

$1\ 4\ \boxed{2\ 3}\ 5\ \boxed{8\ 6\ 7}$

$1\ \boxed{2\ 3}\ 4\ 5\ \boxed{8\ 6\ 7}$

$1\ 2\ \boxed{3}\ 4\ 5\ \boxed{8\ 6\ 7}$

$1\ 2\ 3\ 4\ 5\ \boxed{8\ 6\ 7}$

$1\ 2\ 3\ 4\ 5\ 8\ \boxed{6\ 7}$

$1\ 2\ 3\ 4\ 5\ \boxed{6\ 7}\ 8$

$1\ 2\ 3\ 4\ 5\ 6\ \boxed{7}\ 8$ --> 1 2 3 4 5 6 7 8

Quicksort

Here’s a C++ implementation of quicksort (from Jon Bentley). Its behaviour may not match up exactly with what you just saw.

```
void qsort(int x[], int lo, int hi)
{
    int i, p;
    if (lo >= hi) return;
    p = lo;
    for( i=lo+1; i <= hi; i++ )
        if( x[i] < x[lo] )
            swap(x[++p], x[i]);
    swap(x[lo], x[p]);
    qsort(x, lo, p-1);
    qsort(x, p+1, hi);
}

void quicksort(int x[], int n)
{
    qsort(x, 0, n-1);
}
```

Quicksort

Here’s a Haskell implementation of quicksort that’s much prettier, but it involves lots of memory overhead because of list copies.

```
qsort []     = []
qsort (x:xs) =
  qsort (filter (< x) xs) ++ [x] ++ qsort (filter (>= x) xs)
```

(You’ve now seen sorting algorithms written in four different languages today.)

Quicksort complexity

Quicksort is comparison-based, so the operations are comparisons.

In the partitioning task, each element was compared to the pivot. So there are roughly n comparisons in each partitioning (really n – 1?).

Quicksort complexity

Quicksort is comparison-based, so the operations are comparisons.

In the partitioning task, each element was compared to the pivot. So there are roughly n comparisons in each partitioning (really n – 1?).

In the first step, we have n comparisons.
In the second step, we have floor(n/2) * 2 comparisons (for each half)
In the third step, we have floor(n/4) * 4 comparisons (for each quarter)

... or O(n lg n)

Quicksort complexity

What input will give quicksort its worst performance?

Quicksort complexity

What input will give quicksort its worst performance?
An array that's already sorted.

```
1 [2 3 4 5 6 7 8]   n – 1 comparisons
```

Quicksort complexity

What input will give quicksort its worst performance?
An array that's already sorted.

$1\ \boxed{2\ 3\ 4\ 5\ 6\ 7\ 8}$ n – 1 comparisons

$1\ 2\ \boxed{3\ 4\ 5\ 6\ 7\ 8}$ n – 2 comparisons

Quicksort complexity

What input will give quicksort its worst performance?
An array that's already sorted.

$1\ \boxed{2\ 3\ 4\ 5\ 6\ 7\ 8}$ n – 1 comparisons

$1\ 2\ \boxed{3\ 4\ 5\ 6\ 7\ 8}$ n – 2 comparisons

$1\ 2\ 3\ \boxed{4\ 5\ 6\ 7\ 8}$ n – 3 comparisons and so on...

Quicksort complexity

What input will give quicksort its worst performance?
An array that's already sorted.

$1\ \boxed{2\ 3\ 4\ 5\ 6\ 7\ 8}$ n – 1 comparisons

$1\ 2\ \boxed{3\ 4\ 5\ 6\ 7\ 8}$ n – 2 comparisons

$1\ 2\ 3\ \boxed{4\ 5\ 6\ 7\ 8}$ n – 3 comparisons and so on...

$(n - 1) + (n - 2) + (n - 3) + \ldots + 2 + 1 = n(n - 1)/2$

or $O(n^2)$

Performance depends on choosing good pivots. With good pivot choices, average case behaviour for quicksort is O(n lg n)