CSE 3318

Week of 06/24/2024

Instructor : Donna French

Selection Sort

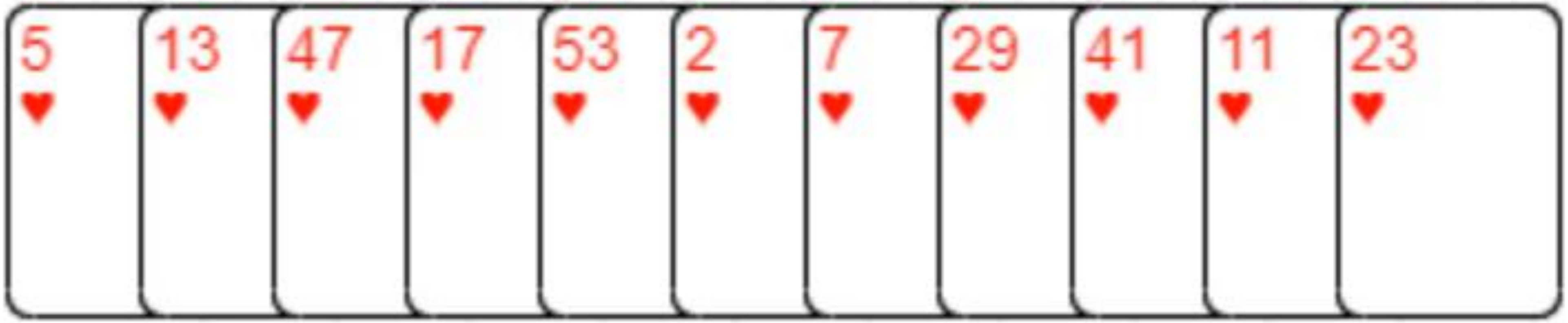
Insertion Sort sorts by taking an item "key" out and inserting it into the correct place (and moved everything else in the array over).

Merge Sort sorts by using divide and conquer with combine(merge) to break the problem down into its smallest pieces and then merge those pieces back together.

Selection Sort is another sort that uses a different technique.

Selection Sort

[redacted]



Selection Sort

1. Find the smallest card. Swap it with the first card.

2. Find the second-smallest card. Swap it with the second card.

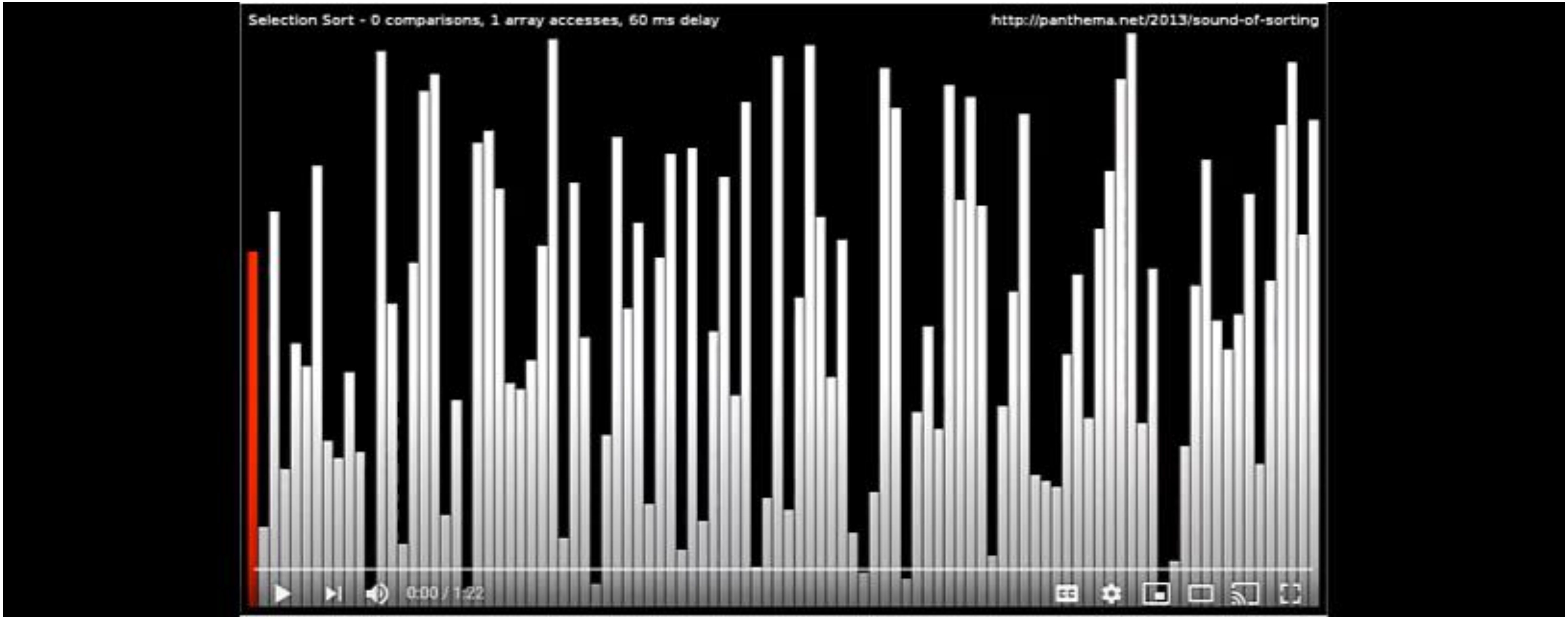
3. Find the third-smallest card. Swap it with the third card.

4. Repeat finding the next-smallest card and swapping it into the correct position until the array is sorted.

This algorithm is called **Selection Sort** because it repeatedly *selects* the next-smallest element and swaps it into place.

Selection Sort

Selection Sort



Selection Sort

What do you think about this algorithm?

What parts of it seem to take the longest?

How do you think it would perform on big arrays?

Keep those questions in mind as we analyze this algorithm.

Selection Sort

Swap

A key step in many sorting algorithms (including selection sort) is swapping the location of two items in an array. Here's a swap function that looks like it might work…

array

index of 1st element to swap

index of 2nd element to swap

```
void swap(int A[], int Swap1, int Swap2)
{
      A[Swap1] =  A[Swap2];
      A[Swap2] =  A[Swap1];
}
```

Selection Sort

```
void swap(int A[], int Swap1, int Swap2)
{
      A[Swap1] =  A[Swap2];
      A[Swap2] =  A[Swap1];
}
```

The first line is OK. Puts the element at `Swap2` into the array element of `Swap1`.

But what happens `A[Swap1]` is assigned to `A[Swap2]`?

`A[Swap1]` has already been updated with `A[Swap2]`; therefore, it just puts `A[Swap2]` back into `A[Swap2]`.

Selection Sort

```
void swap(int A[], int Swap1, int Swap2)
{
    A[Swap1] =  A[Swap2];
    A[Swap2] =  A[Swap1];
}
```

We need to keep track of A[Swap1] before overwriting it.

Selection Sort

student@cse1325: /media/sf_VM2320

File Edit Tabs Help

```c
// C program to demonstrate swap
#include <stdio.h>

void swap(int A[], int Swap1, int Swap2)
{
	A[Swap1] =  A[Swap2];
	A[Swap2] =  A[Swap1];
}

void printArray(int arr[], int size)
{
	int i;
	for (i=0; i < size; i++)
		printf("%d ", arr[i]);
	printf("\n");
}
"Swap.c" [dos] 38L, 720C                                  7,22-29       Top
```

Selection Sort

Selection Sort

Finding the index of the minimum element in a subarray

One of the steps in Selection Sort is to find the next-smallest element to put into its correct location.

For example, if the array initially has values

[13, 19, 18, 4, 10]

we first need to find the index of the smallest value in the array.

Selection Sort

Finding the index of the minimum element in a subarray

[13, 19, 18, 4, 10]

smallest element
index 3

4 is the smallest value and its index is 3.

Selection Sort would swap the value at [3] with the value at index [0]

[4, 19, 18, 13, 10]

Now we need to find the index of the second-smallest value to swap into index 1.

Selection Sort

Finding the index of the minimum element in a subarray

We could write code to find the index of the second-smallest value in an array.

It would be more complex than needed - there's a better way.

Notice that since the smallest value has already been swapped into index 0, what we really want is to find the smallest value in the part of the array that starts at index 1.

[13, 19, 18, 4, 10]

[4, 19, 18, 13, 10]

Selection Sort

Finding the index of the minimum element in a subarray

A section of an array is called a **subarray**, so that in this case, we want the index of the smallest value in the subarray that starts at index 1.

[4, 19, 18, 13, 10] ← Array element 10 at index 4 is the smallest element in the subarray

The smallest value in the subarray starting at index 1 is 10 at [4] in the original array.

So index 4 is the location of the second-smallest element of the full array.

```c
int main(void)
{
    int A[] = {64, 25, 12, 22, 11, 32, 67, 23, 99};
    int n = sizeof(A)/sizeof(A[0]);
    int min_ndx = 0;
    int sub_start = 0;
    int j = 0;

    printArray(A, n);

    printf("Enter index of start of subarray : ");
    scanf("%d", &sub_start);

    min_ndx = sub_start;

    for (j = min_ndx+1; j < n; j++)
    {
        if (A[j] < A[min_ndx])
            min_ndx = j;
    }

    printf("The smallest element in the subarray is %d at index %d\n", A[min_ndx], min_ndx);
```

```
 0     1     2     3    4      5    6      7    8
64, 25, 12, 22, 91, 32, 67, 23, 99


Enter index of start of subarray : 0
The smallest element in the subarray is 12 at index 2

Enter index of start of subarray : 1
The smallest element in the subarray is 12 at index 2

Enter index of start of subarray : 3
The smallest element in the subarray is 22 at index 3

Enter index of start of subarray : 4
The smallest element in the subarray is 23 at index 7

Enter index of start of subarray : 8
The smallest element in the subarray is 99 at index 8
```

0	1	2	3	4	5	6	7	8
64,	25,	12,	22,	91,	32,	67,	23,	99

```
min_ndx = sub_start;

for (j = min_ndx+1; j < n; j++)
{
   if (A[j] < A[min_ndx])
      min_ndx = j;
}
```

```
min_ndx  j
-------  ---------------
3        min_ndx+1=3+1=4
```

```
Enter index of start of subarray : 3
The smallest element in the subarray is 22 at index 3
```

0	1	2	3	4	5	6	7	8
64,	25,	12,	22,	91,	32,	67,	23,	99

```
min_ndx = sub_start;

for (j = min_ndx+1; j < n; j++)
{
   if (A[j] < A[min_ndx])
      min_ndx = j;
}
```

```
min_ndx  j
-------  ---------------
4        min_ndx+1=4+1=5
```

```
Enter index of start of subarray : 4
The smallest element in the subarray is 23 at index 7
```

Selection Sort

Going back to our definition of Selection Sort

1. Find the smallest element. Swap it with the first element.

2. Find the second-smallest element. Swap it with the second element.

3. Find the third-smallest element. Swap it with the third card.

4. Repeat finding the next-smallest element and swapping it into the correct position until the array is sorted.

We now have the code to find the smallest, second smallest, third smallest, etc…

We also have the code to swap two array elements.

```c
void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void selectionSort(int A[], int n)
{
    int i, j, min_idx;

    for (i = 0; i < n-1; i++)
    {
        min_idx = i;
        for (j = i+1; j < n; j++)
        {
            if (A[j] < A[min_idx])
                min_idx = j;
        }

        swap(&A[min_idx], &A[i]);
    }
}
```

Selection Sort

```
void selectionSort(int A[], int n)
{
  int i, j, min_idx;

  for (i = 0; i < n-1; i++)
  {
    min_idx = i;

    for (j = i+1; j < n; j++)
    {
      if (A[j] < A[min_idx])
        min_idx = j;
    }

    swap(&A[min_idx], &A[i]);
  }
}
```

```
{13, 9, 4, 1}

n = 4

 i    min_idx    j
---   -------   ---
 0
```

Selection Sort

```
void selectionSort(int A[], int n)
{
  int i, j, min_idx;

  for (i = 0; i < n-1; i++)
  {
    min_idx = i;

    for (j = i+1; j < n; j++)
    {
      if (A[j] < A[min_idx])
        min_idx = j;
    }

    swap(&A[min_idx], &A[i]);
  }
}
```

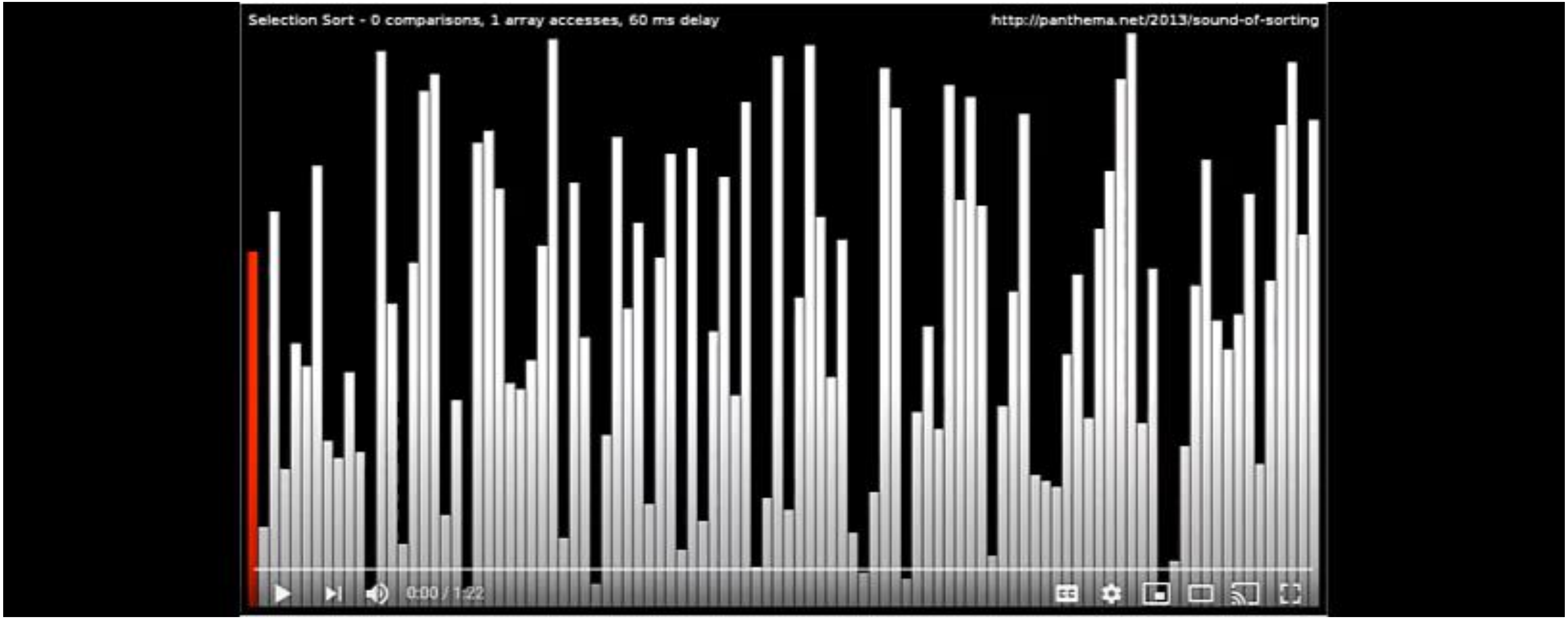
```
{1, 9, 4, 13}

n = 4

 i    min_idx     j
---   -------    ---
 1
```

Selection Sort

```
void selectionSort(int A[], int n)
{
  int i, j, min_idx;

  for (i = 0; i < n-1; i++)
  {
    min_idx = i;

    for (j = i+1; j < n; j++)
    {
      if (A[j] < A[min_idx])
        min_idx = j;
    }

    swap(&A[min_idx], &A[i]);
  }
}
```

```
{1, 4, 9, 13}

n = 4

 i    min_idx     j
---   -------    ---
 2
```

Selection Sort



Analysis of Selection Sort

Selection Sort loops over indices in the array

```
for (i = 0; i < n-1; i++)
```

Let's refer to this loop as the "outer i loop"

For each index, the code loops to find the minimum elements of the subarray

```
for (j = i+1; j < n; j++)
```

Let's refer to this loop as the "inner j loop"

and does a swap.

```
swap(&A[min_idx], &A[i]);
```

If the length of the array is *n*, then there are *n* indices in the array.

Analysis of Selection Sort

How many lines of code are executed by a single call to `swap()`?

```
void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}
```

In this implementation, three lines are ALWAYS executed.

We can say that each call to `swap()` takes constant time.

Analysis of Selection Sort

How many lines of code are executed by a single pass through the "outer i loop"?

We also have to account for the "inner j loop".

How many times does this loop execute in a given pass through the "outer i loop"?

It depends on the size of the subarray that it's iterating over. If the subarray is the whole array (as it is on the first step), the loop body runs *n* times.

If the subarray is of size 6, then the loop body runs 6 times.

Analysis of Selection Sort

```
for (i = 0; i < n-1; i++)
{
    min_idx = i;
    for (j = i+1; j < n; j++)
    {
        if (A[j] < A[min_idx])
            min_idx = j;
    }

    swap(&A[min_idx], &A[i]);
}
```

Let's examine an array of size 8.

1st call

When i = 0, the "inner j loop" will run from i+1 to j < n so from 1 to 7.

So we can say that for the first pass through the "outer i loop", the "inner j loop" will run 7 times.

Analysis of Selection Sort

```
for (i = 0; i < n-1; i++)
{
     min_idx = i;
     for (j = i+1; j < n; j++)
     {
          if (A[j] < A[min_idx])
               min_idx = j;
     }

     swap(&A[min_idx], &A[i]);
}
```

Let's examine an array of size 8.

2^{nd} call

When i = 1, the "inner j loop" will run from i+1 to j < n so from 2 to 7.

So we can say that for the first pass through the "outer i loop", the "inner j loop" will run 6 times.

Analysis of Selection Sort

```
for (i = 0; i < n-1; i++)
{
      min_idx = i;
      for (j = i+1; j < n; j++)
      {
            if (A[j] < A[min_idx])
                  min_idx = j;
      }

      swap(&A[min_idx], &A[i]);
}
```

Let's examine an array of size 8.

3rd call

When i = 2, the "inner j loop" will run from i+1 to j < n so from 3 to 7.

So we can say that for the first pass through the "outer i loop", the "inner j loop" will run 5 times.

Analysis of Selection Sort

1^{st} call (i = 0) for array of 8 elements	7 times
2^{nd} call (i = 1) for array of 8 elements	6 times
3^{rd} call (i = 2) for array of 8 elements	5 times

Noticing a pattern here?

"outer i loop" goes from 0 to n-1.

When i is 6, "inner j loop" will run from i+1 to j < n so from 7 to 7 < 8.

So "inner j loop" runs once when "outer i loop" is on the last element of the array.

Analysis of Selection Sort

1st call (i = 0) for array of 8 elements	7 times
2nd call (i = 1) for array of 8 elements	6 times
3rd call (i = 2) for array of 8 elements	5 times
4th call (i = 3) for array of 8 elements	4 times
5th call (i = 4) for array of 8 elements	3 times
6th call (i = 5) for array of 8 elements	2 times
7th call (i = 6) for array of 8 elements	1 time
8th call (i = 7) for array of 8 elements	Loop fails

Side Note : Arithmetic Series

How do you compute the sum 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 quickly?

8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 =

(8 + 1) + (7 + 2) + (6 + 3) + (5 + 4) =

9 + 9 + 9 + 9 = 4 * 9 = 36

1. Add the smallest and the largest number.
2. Multiply by the number of pairs.

Side Note : Arithmetic Series

What if the number of integers in the sequence is odd, so that you cannot pair them all up?

It doesn't matter!

Just count the unpaired number in the middle of the sequence as half a pair.

1 + 2 + 3 + 4 + 5

(1 + 5) + (2 + 4) + 3 = 2.5 pairs where each pair has a value of 6.

2.5 * 6 = 15

Side Note : Arithmetic Series

What if the sequence to sum up goes from 1 to *n*?

This an **arithmetic series**.

The sum of the smallest and largest numbers is *n*+1

Because there are *n* numbers total, there are $\frac{n}{2}$ pairs (whether *n* is odd or even).

Therefore, the sum of numbers from 1 to *n* is (*n* + 1)($\frac{n}{2}$) which is $\frac{n^2+n}{2}$

Side Note : Arithmetic Series

$$\frac{n^2+n}{2}$$

8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = (8 + 1) + (7 + 2) + (6 + 3) + (5 + 4) = 9 + 9 + 9 + 9 = 4 * 9 = 36

$$\frac{n^2+n}{2} = \frac{8^2+8}{2} = \frac{72}{2} = 36$$

1 + 2 + 3 + 4 + 5 = (1 + 5) + (2 + 4) + 3 = 2.5 pairs => 2.5 * 6 = 15

$$\frac{n^2+n}{2} = \frac{5^2+5}{2} = \frac{30}{2} = 15$$

Analysis of Selection Sort

The total running time for selection sort has three parts

1. The running time of the "outer i loop"

2. The running time for all the calls to `swap()`.

3. The running time for the "inner j loop"

Analysis of Selection Sort

Parts 1 and 2 are easy

1. The running time of the "outer i loop"

This loop is really just testing and incrementing the loop variable and running the "inner j loop" and calling `swap()`, so it takes constant time for each of the *n* iterations.

Using asymptotic notation, the time for all of these steps is $\Theta(n)$.

2. The running time for all the calls to `swap()`.

We know that there are *n-1* calls to `swap()` and each call takes constant time.

Using asymptotic notation, the time for all calls to `swap()` is $\Theta(n)$.

Analysis of Selection Sort

The running time for the "inner j loop"

Each individual iteration of the loop in "inner j loop" takes constant time. The number of iterations of this loop is n in the first call, then $n-1$, then $n-2$ and so on.

We know that this sum, $1 + 2 + \ldots + (n-1) + n$ is an arithmetic series and it evaluates to

$$\frac{n^2+n}{2}$$

Therefore, the total time for all calls to "inner j loop" is some constant, c_1, times $\frac{n^2+n}{2}$

Analysis of Selection Sort

Therefore, the total time for all calls to "inner j loop" is some constant, c_1, times $\frac{n^2+n}{2}$

$$c_1\left(\frac{n^2+n}{2}\right) = c_1\left(\frac{1}{2}n^2 + \frac{1}{2}n\right) = \frac{1}{2}c_1(n^2 + n) = \frac{1}{2}c_1 n^2 + \frac{1}{2}c_1 n$$

In terms of big-Θ notation, we can eliminate c_1 and the factor of $\frac{1}{2}$. We can also eliminate the low-order term of n.

The result is that the running time for all the calls to "inner j loop" is $\Theta(n^2)$.

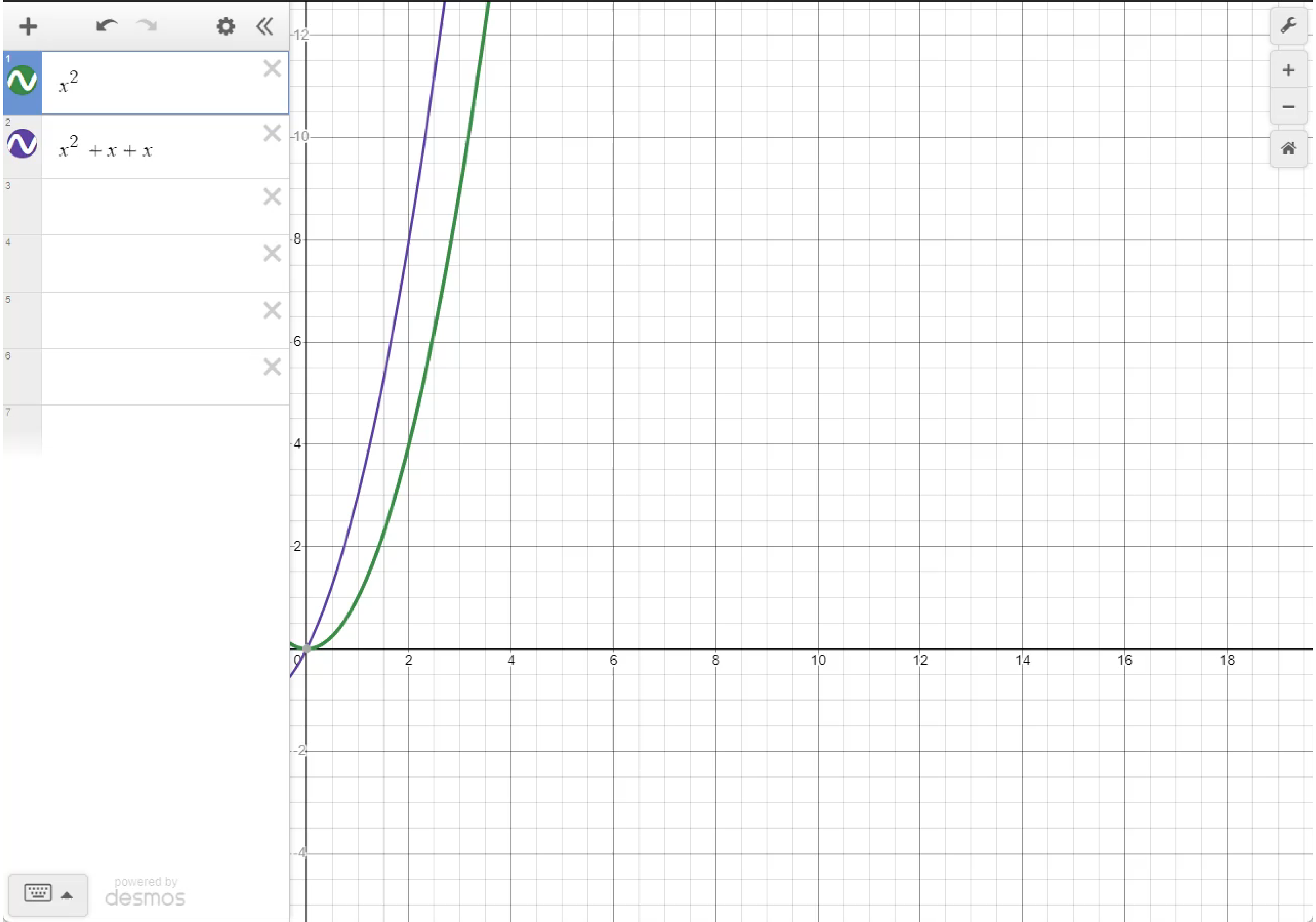
Analysis of Selection Sort

The total running time for selection sort has three parts

1. The running time of the "outer i loop" $\Theta(n)$
2. The running time for all the calls to `swap()`. $\Theta(n)$
3. The running time for the "inner j loop" $\Theta(n^2)$

$\Theta(n) + \Theta(n) + \Theta(n^2) = \Theta(n^2)$

What is big O and big Ω of Selection Sort?


x^2
$x^2 + x + x$
12
10
8
6
4
2
0
2
4
6
8
10
12
14
16
18
-2
-4
powered by
desmos

Analysis of Selection Sort

$\Theta(n^2)$

What is big O and big Ω of Selection Sort?

No case is particularly good or particularly bad for selection sort.

The loop in "inner j loop" will always make $\frac{n^2+n}{2}$ iterations regardless of the input.

Selection Sort runs in $\Theta(n^2)$ and $O(n^2)$ and $\Omega(n^2)$.

Analysis of Selection Sort

So what does a runtime of $\Theta(n^2)$ tell us about Selection Sort?

Let's use an example where the constant factor is $\frac{1}{10^6}$ so that selection sort takes approximately $(\frac{1}{10^6})n^2$ seconds to sort n values.

n = 100 The running time of selection sort is about $\frac{100^2}{10^6}$ which is $\frac{1}{100}$ seconds. That seems pretty fast.

n = 1000 The running time of selection sort is about $\frac{1000^2}{10^6}$ which is 1 second. Not bad??

n grew by a factor of 10 but the runtime of the sort increased by a factor of 100? Hmmm…

What if n = 1,000,000? $\frac{1000000^2}{10^6}$ = 1,000,000 seconds = 11 days and 14 hours.

Increasing the array size by a factor of 1000 increases the running time a million times!

Analysis of Selection Sort

Does the "sortedness" of the array affect the runtime of Selection Sort?

Does it run faster for a sorted array?

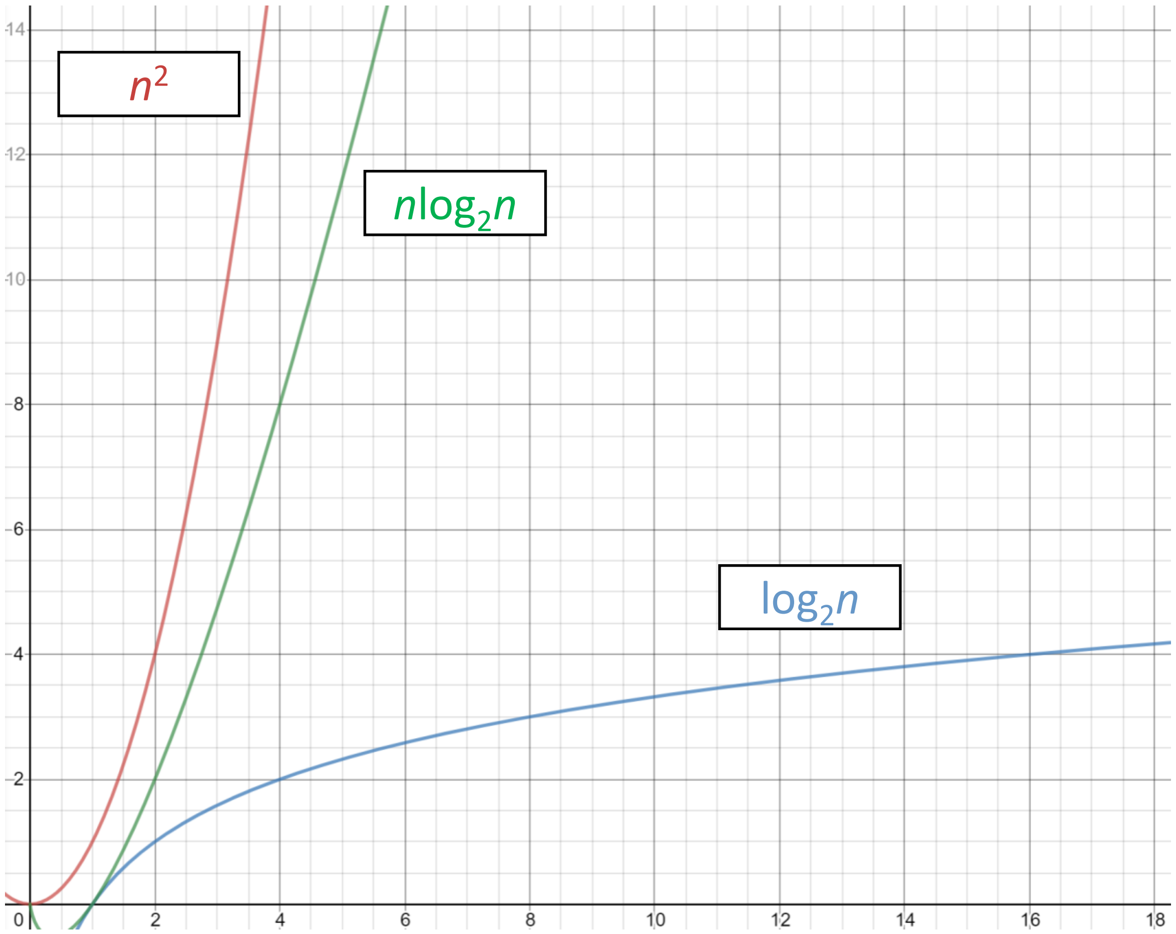
Does it run slower for an array in reverse sorted order?

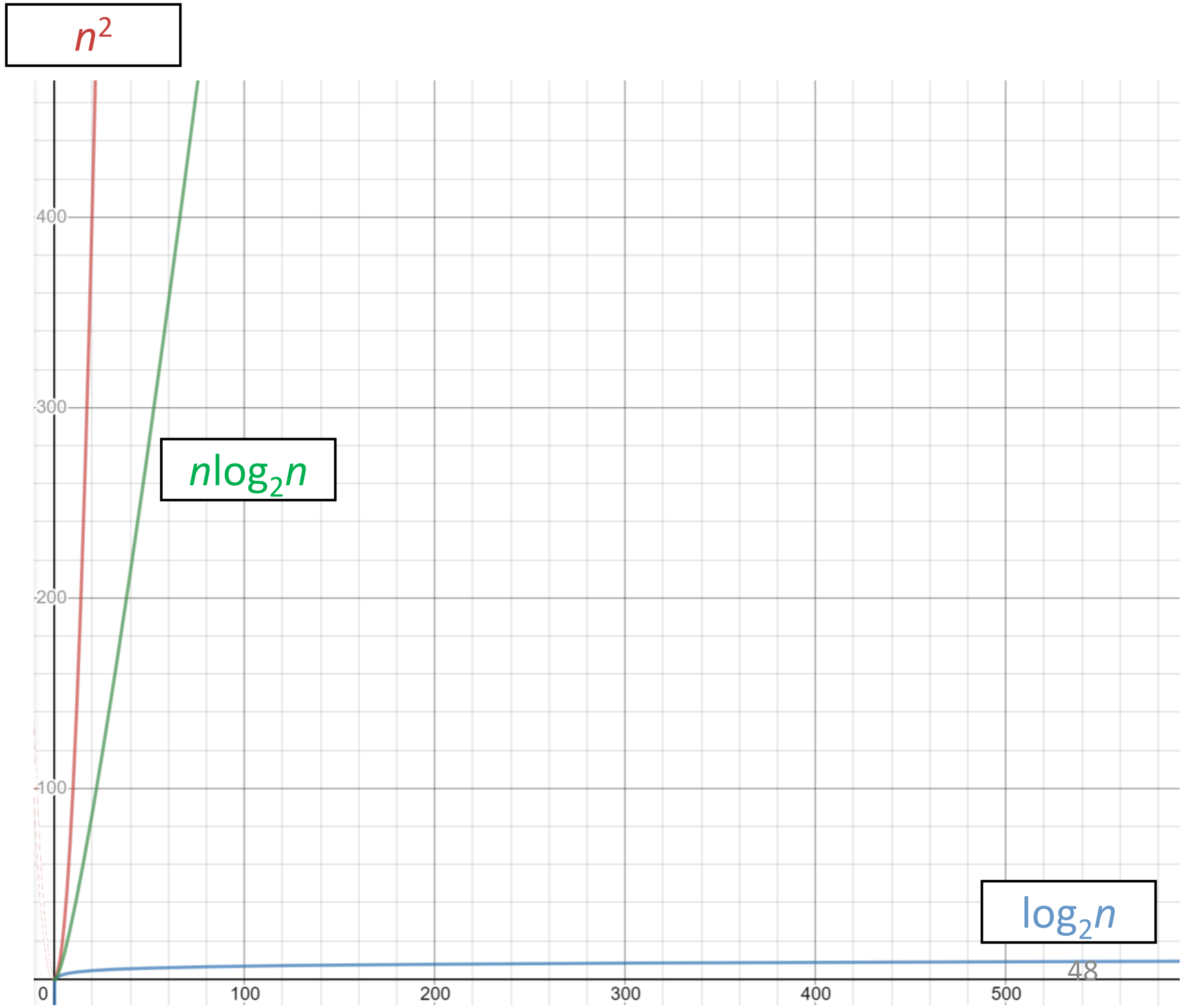
No.

Selection Sort will run through the same number of steps regardless.

Analysis of Selection Sort

$\Theta(n^2)$ is not that great of a run time.





Analysis of Selection Sort

Selection Sort shares many of same benefits of Insertion Sort …

- It performs well on small inputs

depending on the constant factors involved - it can beat $O(n\log_2 n)$ sorts

- It requires only constant extra space (unlike merge sort)

Selection Sort has some extra benefits

- The code is very simple; therefore, easy to program.

- It only requires *n-1* swaps (which is better than most sorting algorithms)

- For the same set of elements, it will take the same amount of time regardless of how they are arranged.
  - This can be good for real time applications.

Analysis of Selection Sort

Here are the cons

- $O(n^2)$ is slower than $O(n\log_2 n)$ algorithms (like merge sort) for large inputs.

- Insertion sort, which is also $O(n^2)$, is usually faster than it on small inputs.
    the constant factors would determine which is faster

So what are some situations when you want to use it?

Analysis of Selection Sort

- You need a sort algorithm that is easy to program

- You need a sort algorithm that requires a small amount of code

- You only have a small number of elements to sort, so you feel that it is quick enough and don't want to sacrifice more memory to get more speed.

- Swaps are expensive on your hardware, but you don't want to use a more complicated sort that cuts down on swaps.

- You need the sorting time to be consistent for a given size.

Coding Assignment 2

Formatting

Program is properly formatted – both alignment and formatting.

Student's name and id are at the top of the C file.

The file containing C code is named Code2_xxxxxxxxxx.c and the Excel file is named Output_xxxxxxxxxx.xls where xxxxxxxxxx is the student's 10 digit student id.

Formatting

```
abcd efghijklmnopq(rst u[], vwx y){zab c, d, efghijk;
lmn (o = q; r < stu; v++){xyzabcd = e;fgh (i = k+1; l < m; n++){
op (q[r] < s[tuvwxyz]){abcdefg = i;}}hijk(&l[mnopqrx], &t[u]);}}
```

Formatting

```
abcd efghijklmnopq(rst u[], vwx y){zab c, d, efghijk;
lmn (o = q; r < stu; v++){xyzabcd = e;fgh (i = k+1; l < m; n++){
op (q[r] < s[tuvwxyz]){abcdefg = i;}}hijk(&l[mnopqrx], &t[u]);}}

```

Coding Assignment 2

Execution 1

GTA - compile code with compiler directive of "-D PRINTARRAY". Run code with student's "TestFile.txt" on the command line. Confirm that program uses command line parameter of file name for input (file contents should print to the screen).

If the file name is missing from the command line, then the program should print a message and exit as it did in Coding Assignment 1.

If the file name given on the command line does not exist, then the program should print a message and exit as it did in Coding Assignment 1.

Coding Assignment 2

Execution 2

GTA - compile code with compiler directive of "-D PRINTARRAY".

Run code with student's "TestFile.txt" on the command line.

Confirm that the unsorted output is printed and then the sorted output is printed.

Coding Assignment 2

Execution 3

GTA - compile code without compiler directive of "-D PRINTARRAY".

Run code with a large input file name on the command line.

Confirm that program runs and outputs the timing results for a large file.

Coding Assignment 2

GTA Instructions - 1

Verify that insertion sort logic were used and is in its own function.

If any other sort logic was used, fail all rubric criteria to assign a grade of 0 to assignment.

Coding Assignment 2

GTA Instructions - 2

Verify that student's TestFile.txt belongs to that student and was created with the provided file generator.

Coding Assignment 2

Coding Requirement 1

Code should read though file once to count records and then use `fseek()` to return to the top of the file before reading it again to write the records into the array.

Code should NOT close and reopen the file in order to start back at the top of the file - `fseek()` MUST be used.

File should be closed when done. Close just can't be used to return the file pointer back to the top - `fseek()` must be used.

All of this code should be in the file handling/array filling function and NOT in `main()`.

Coding Assignment 2

Coding Requirement 2

Code should `malloc()` the array based on the number of records counted in the file.

Array should be `free()`d before the program ends.

The SAME array should be used for both the unsorted array and the sorted array.

Pointer for array must be declared in `main()` and passed to file handling/array filling function.

Call to `malloc()` must be in file handling/array filling function. Fail this rubric criterion if the return value of `malloc()` is cast.

Coding Assignment 2

Coding Requirement 3

Code should use the `clock()` function to time JUST the insertion sort process.

File handling and array handling should not be included in the timing.

The calls to `clock()` and insertion sort should be in `main()`.

The file handling and array filling must be in one function and not in `main()`.

Insertion sort code must be in its own function and not in `main()`.

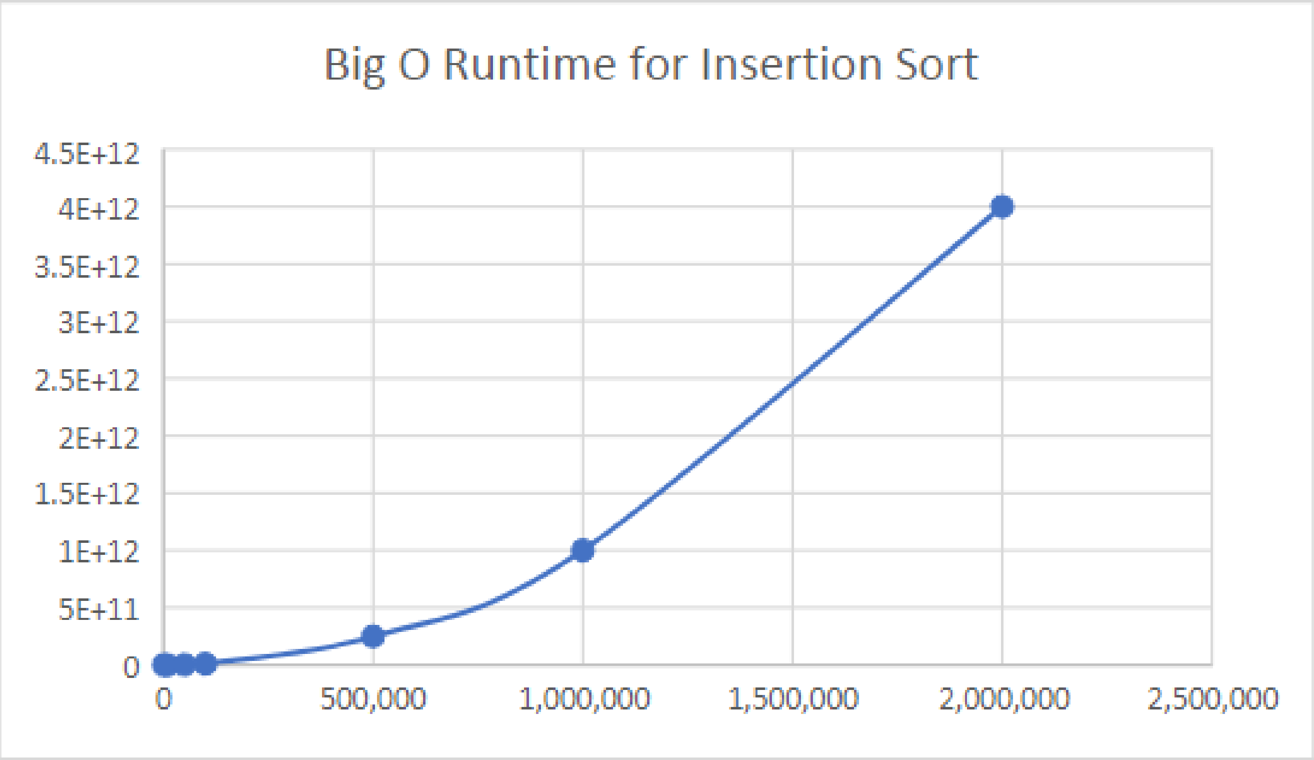
Coding Assignment 2

Final Results

Instructor will review final results and charts.

Points will be earned for correctly entering the tic information from the PC runs and the Omega runs and correctly graphing them.

	PC	Omega	Big O Runtime
n	Insertion Sort	Insertion Sort	Insertion Sort
1,024			1048576
10,000			100000000
50,000			2500000000
100,000			10000000000
500,000			2.5E+11
1,000,000			1E+12
2,000,000			4E+12



Coding Assignment 2

GTA Instructions

Compile the student's `Code2_xxxxxxxxxx.c` using a newer version of gcc on the PC and on Omega (add the `–std=c99` compiler flag). Perform all rubric criterion and pass/fail each one as needed. If code compiles with warnings/errors, fail all rubric points which will assign a grade of 0. Do not check any of the rubric criterion. Paste a copy of the compiler error/warning into the grader's notes for this criterion.

Assign a grade of 0 to assignment if any of the following coding constructs were used in the program - global variables (typedef structs are definitions - not global variables), goto, continue, break (except in a switch), exit (except when allowed by the assignment specifically) and return used inside any loop.

Coding Assignment 1

Create a function to read the file and call a function to add nodes to the linked list

Parameters : `argc`, `argv` and the linked list head

Return Value : `void`

Use `argv[1]` to get the filename from the command line if it exists. If the program is not run with a file name on the command line, then print the message shown in the sample output and exit*. If the program is run with a file name on the command line, then open the file with `"r"`. Check if the file opened and, if it did not, then print the message shown in the sample output and exit*. Use `fgets()` to loop through the file. See "File Handling in C" in the "Review Materials" module of Canvas. As each line of the file is read, call `AddNodeToLL()` to add the number read from the file as a node in the linked list. Count how many records you read from the file/added to the linked list. Sum up the numbers read from the file/added to the linked list. Print the count and the sum to the screen - see sample output.

* Use `exit(0);` in your code – this is the exception mentioned in the rubric.

Coding Assignment 2

Step 4

Add the ability to read a file name from the command line. **You already have this code from Coding Assignment 1.** Reuse the same file handling code including the allowed use of exit(0) when the file name is missing or invalid.

Using `fgets()`, loop through the file and count the number of lines in the file. Use `fseek()` to move the file pointer back to the beginning of the file (See "File Handling in C" in the "Review Materials" module of Canvas).

Coding Assignment 3

Make a copy of Coding Assignment 2.

Add a Merge Sort function to your code.

You will run your code on the same files from Coding Assignment 2.

You will graph the results of Insertion Sort and Merge Sort.

Quick Sort

Like Merge Sort, Quick Sort uses divide and conquer and so it's a recursive algorithm also.

The way that Quick Sort uses divide-and-conquer is a little different from how Merge Sort does.

In Merge Sort, the divide step does hardly anything and all the real work happens in the combine step.

Quick Sort is the opposite - all the real work happens in the divide step.

In fact, the combine step in Quick Sort does absolutely nothing.

Quick Sort

Quick Sort has a couple of other differences from Merge Sort.

Quick Sort works in place.

Its worst-case running time is as bad as Selection Sort and Insertion Sort $\Theta(n^2)$

But its average-case running time is as good as Merge Sort $\Theta(n\log_2 n)$

So why think about quicksort when merge sort is at least as good?

That's because the constant factor hidden in the big-Θ notation for Quick Sort is quite good.

In practice, Quick Sort outperforms Merge Sort and it significantly outperforms Selection Sort and Insertion Sort.

Quick Sort

Quick Sort uses divide-and-conquer.

Just like we did with Merge Sort, think of sorting a subarray

`array[p..r]`

where initially the subarray is `array[0..n-1]`.

Quick Sort

Divide

Choose any element in the subarray `array[p..r]` and call this element the **pivot**.

Rearrange the elements in `array[p..r]` so that all elements in `array[p..r]` that are less than or equal to the **pivot** are to its left and all elements that are greater than the **pivot** are to its right.

This procedure is called **partitioning**.

At this point, it doesn't matter what order the elements to the left/to the right of the pivot are in relation to each other.

We just care that each element is somewhere on the correct side of the pivot.

Quick Sort

As a matter of practice, we'll always choose the rightmost element in the subarray, `array[r]`, as the **pivot**.

So, for example, if the subarray consists of

[9, 7, 5, 11, 12, 2, 14, 3, 10, 6]

then we choose 6 as the pivot.

After **partitioning**, the subarray will look like

[5, 2, 3, 6, 12, 7, 14, 9, 10, 11]

Everything to the left of 6 is less than 6.
Everything to the right of 6 is greater than 6.

Let `q` be the index of where the **pivot** ends up.

Quick Sort

Conquer

Recursively sort the subarrays `array[p..q-1]`

all elements to the left of the pivot, which must be less than or equal to the **pivot**

and `array[q+1..r]`

all elements to the right of the pivot, which must be greater than the **pivot**

Quick Sort

Combine

Do nothing.

Once the conquer step recursively sorts, the sort is complete.

Why?

All elements to the left of the pivot, in `array[p..q-1]`, are less than or equal to the **pivot** and are sorted and all elements to the right of the **pivot**, in `array[q+1..r]`, are greater than the **pivot** and are sorted.

The elements in `array[p..r]` can't help but be sorted!

Quick Sort

Quick Sort is a divide and conquer recursion algorithm. So from `main()`, the quick sort function is called…

```
QuickSort(arr, 0, n-1);
```

with parameters of the array, the start of the array (index 0) and the last index (number of elements in array – 1)

Quick Sort

```
int main(void)
{
    int arr[] = {9, 6, 5, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    QuickSort(arr, 0, n-1);

    return 0;
}
```

Quick Sort

```
QuickSort(0,3)
{9, 6, 5, 7}

partition(0, 3)
ndx = ?

QuickSort(0,ndx-1)
QuickSort(ndx+1,3)
```

```
void QuickSort(int A[], int low, int high)
{
	if (low < high)
	{
		int ndx = partition(A, low, high);

		QuickSort(A, low, ndx - 1);
		QuickSort(A, ndx + 1, high);
	}
}
```

Note here that the 1st call alters `high` and the 2nd call alters `low`

Quick Sort

```
int partition (int A[], int low, int high)
{
   int i, j = 0;
   int pivot = A[high];

   i = (low - 1);

   for (j = low; j < high; j++)
   {
      if (A[j] < pivot)
      {
         i++;
         swap(&A[i], &A[j]);
      }
   }
   swap(&A[i + 1], &A[high]);

   return (i + 1);
}
```

```
void swap(int *SwapA, int *SwapB)
{
        int temp = *SwapA;
        *SwapA = *SwapB;
        *SwapB = temp;
}
```

```c
int partition (int A[], int low, int high)
{
   int i, j = 0;
   int pivot = A[high];

   i = (low - 1);

   for (j = low; j < high; j++)
   {
      if (A[j] < pivot)
      {
         i++;
         swap(&A[i], &A[j]);
      }
   }
   swap(&A[i + 1], &A[high]);

   return (i + 1);
}
```

0	1	2	3
{9,	6,	5,	7}
{6,	5,	7,	9}

1st call – `partition(A, 0, 3)`

i	j	pivot	low	high

Quick Sort

```
{9, 6, 5, 7}
QuickSort(0,3)

partition(0, 3)
ndx = 2
{6, 5, 7, 9}

QuickSort(0,1)
QuickSort(3,3)
```

```
void QuickSort(int A[], int low, int high)
{
    if (low < high)
    {
        int ndx = partition(A, low, high);

        QuickSort(A, low, ndx - 1);
        QuickSort(A, ndx + 1, high);
    }
}
```

Quick Sort

```
{9, 6, 5, 7}
QuickSort(0,3)
partition(0,3)
ndx = 2
{6, 5, 7, 9}

QuickSort(0,1)
QuickSort(3,3)
```

```
QuickSort(0,1)
{6, 5, 7, 9}
partition(0,1)
ndx = ?
{?, ?, ?, ?}
QuickSort(0,ndx-1)
QuickSort(ndx+1,1)
```

```
int partition (int A[], int low, int high)
{
   int i, j = 0;
   int pivot = A[high];

   i = (low - 1);

   for (j = low; j < high; j++)
   {
      if (A[j] < pivot)
      {
         i++;
         swap(&A[i], &A[j]);
      }
   }
   swap(&A[i + 1], &A[high]);

   return (i + 1);
}
```

0	1	2	3
{6,	5,	7,	9}
{5,	6,	7,	9}

2nd call – `partition(A, 0, 1)`

i	j	pivot	low	high

Quick Sort

```
{9, 6, 5, 7}
QuickSort(0,3)
partition(0, 3)
ndx = 2
{6, 5, 7, 9}

QuickSort(0,1)
QuickSort(3,3)
```

```
QuickSort(0,1)
{6, 5, 7, 9}
partition(0, 1)
ndx = 0
{5, 6, 7, 9}
QuickSort(0,-1)
QuickSort(1,1)
```

{9, 6, 5, 7}
QuickSort(0,3)
partition(0, 3)
ndx = 2
{6, 5, 7, 9}

QuickSort(0,1)
QuickSort(3,3)

QuickSort(0,1)
{6, 5, 7, 9}
partition(0, 1)
ndx = 0
{5, 6, 7, 9}
QuickSort(0,-1)
QuickSort(1,1)

QuickSort(0,-1)
{5, 6, 7, 9}
low = 0
high = -1
low $\nless$ high
return

QuickSort(1,1)
{?, ?, ?, ?}
low = 1
high = 1
low $\nless$ high
return

Quick Sort

```
int partition (int A[], int low, int high)
{
   int i, j = 0;
   int pivot = A[high];

   i = (low - 1);

   for (j = low; j < high; j++)
   {
      if (A[j] < pivot)
      {
         i++;
         swap(&A[i], &A[j]);
      }
   }
   swap(&A[i + 1], &A[high]);

   return (i + 1);
}
```

{9,6,5,7}	7 is pivot
	9 ≮ 7 => no swap
	6 < 7 => swap 9 and 6
{6,9,5,7}	5 < 7 => swap 9 and 5
{6,5,9,7}	final => swap 9 and 7
{6,5,7,9}	7 was pivot so divide {6,5} and {9}
{6,5}{7}{9}	9 does not process (because low ≮ high)
{6,5}	5 is pivot
	6 ≮ 5 => no swap
	final => swap 6 and 5
{5,6,7,9}	

Quick Sort

{6,9,7,5} 5 is pivot

6 ≮ 5 => no swap

9 ≮ 5 => no swap

7 ≮ 5 => no swap

final => swap 6 and 5

{5,9,7,6} 5 was pivot so divide into {} and

{}{5}{9,7,6} nothing to the left of 5

{9,7,6} 6 is pivot

9 ≮ 6 => no swap

7 ≮ 6 => no swap

final => swap 9 and 6

{6,7,9} 6 was pivot so divide into {} and {7,9}

{}{6}{7,9} nothing to the left of 6

{7,9} 9 is pivot – see next slide

{5,6,7,9}

```
int partition (int A[], int low, int high)
{
  int i, j = 0;
  int pivot = A[high];

  i = (low - 1);

  for (j = low; j < high; j++)
  {
    if (A[j] < pivot)
    {
      i++;
      swap(&A[i], &A[j]);
    }
  }
  swap(&A[i + 1], &A[high]);

  return (i + 1);
}
```

```
int partition (int A[], int low, int high)
{
   int i, j = 0;
   int pivot = A[high];

   i = (low - 1);

   for (j = low; j < high; j++)
   {
      if (A[j] < pivot)
      {
         i++;
         swap(&A[i], &A[j]);
      }
   }
   swap(&A[i + 1], &A[high]);

   return (i + 1);
}
```

0	1	2	3

{5, 6, 7, 9}

{5, 6, 7, 9}

partition(A, 2, 3)

i	j	pivot	low	high

Quick Sort

```
{9,7,6,5}       5 is pivot
                9 ≮ 5 => no swap
                7 ≮ 5 => no swap
                6 ≮ 5 => no swap
                final => swap 9 and 5
{5,7,6,9}       5 was pivot so divide into
                     {} and {7,6,9}
{}{5}{7,6,9}    nothing to the left of 5
{7,6,9}         9 is pivot
```

```
int partition (int A[], int low, int high)
{
  int i, j = 0;
  int pivot = A[high];

  i = (low - 1);

  for (j = low; j < high; j++)
  {
    if (A[j] < pivot)
    {
      i++;
      swap(&A[i], &A[j]);
    }
  }
  swap(&A[i + 1], &A[high]);

  return (i + 1);
}
```

```
int partition (int A[], int low, int high)
{
   int i, j = 0;
   int pivot = A[high];

   i = (low - 1);

   for (j = low; j < high; j++)
   {
      if (A[j] < pivot)
      {
         i++;
         swap(&A[i], &A[j]);
      }
   }
   swap(&A[i + 1], &A[high]);

   return (i + 1);
}
```

0	1	2	3
{5,	7,	6,	9}
{5,	7,	6,	9}

```
partition(A, 1, 3)
```

i	j	pivot	low	high

Quick Sort

{9,7,6,5}	5 is pivot
	9 ≮ 5 => no swap
	7 ≮ 5 => no swap
	6 ≮ 5 => no swap
	final => swap 9 and 5
{5,7,6,9}	5 was pivot so divide into {} and {7,6,9}
{}{5}{7,6,9}	nothing to the left of 5
{7,6,9}	9 is pivot
	7 < 9 => swap 7 and 7
{7,6,9}	6 < 9 => swap 6 and 6
{7,6,9}	final => swap 9 and 9
{7,6,9}	9 was pivot so divide into {7,6} and {9} and {}
{7,6}{9}{}	nothing to the right of 9
{7,6}	6 is pivot
{7,6}	7 ≮ 6 => no swap
{5,6,7,9}	final => swap 7 and 6

Quick Sort

{5,6,7,9}

9 is pivot

```c
int partition (int A[], int low, int high)
{
   int i, j = 0;
   int pivot = A[high];

   i = (low - 1);

   for (j = low; j < high; j++)
   {
      if (A[j] < pivot)
      {
         i++;
         swap(&A[i], &A[j]);
      }
   }
   swap(&A[i + 1], &A[high]);

   return (i + 1);
}
```

0	1	2	3

{5, 6, 7, 9}

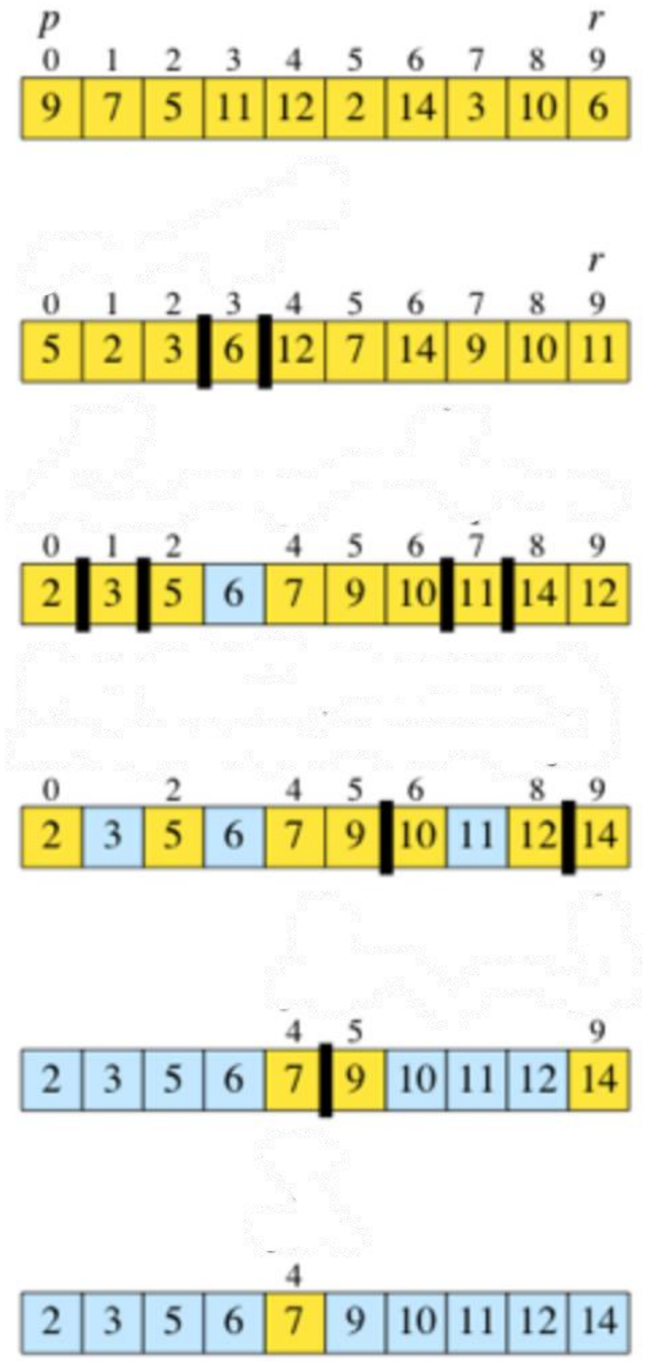
{5, 6, 7, 9}

partition(A, 0, 3)

i	j	pivot	low	high

Quick Sort

{5,6,7,9}	9 is pivot
	5 < 9 => swap 5 and 5
	6 < 9 => swap 6 and 6
	7 < 9 => swap 7 and 7
	final => swap 9 and 9
{5,6,7,9}	9 was pivot so divide into {5,6,7} and {9} and {}
{5,6,7}{9}{}	nothing to the right of 9
{5,6,7}	7 is pivot
	5 < 7 => swap 5 and 5
	6 < 7 => swap 6 and 6
	final => swap 7 and 7
{5,6,7}	7 was pivot so divide into {5,6} and {7} and {}
{5,6}{7}{}	nothing to the right of 7
{5,6}	6 is pivot
	5 < 6 => swap 5 and 5
{5,6,7,9}	final => swap 6 and 6


p
r
0 1 2 3 4 5 6 7 8 9
9 7 5 11 12 2 14 3 10 6
r
0 1 2 3 4 5 6 7 8 9
5 2 3 6 12 7 14 9 10 11
0 1 2 4 5 6 7 8 9
2 3 5 6 7 9 10 11 14 12
0 2 4 5 6 8 9
2 3 5 6 7 9 10 11 12 14
4 5 9
2 3 5 6 7 9 10 11 12 14
4
2 3 5 6 7 9 10 11 12 14

Quick Sort

9 7 5 11 12 2 14 3 10 6
5 7 9 11 12 2 14 3 10 6
5 2 9 11 12 7 14 3 10 6
5 2 3 11 12 7 14 9 10 6
5 2 3 6 12 7 14 9 10 11

2 5 3 6 12 7 14 9 10 11
2 3 5 6 7 12 14 9 10 11
2 3 5 6 7 9 14 12 10 11
2 3 5 6 7 9 10 12 14 11
2 3 5 6 7 9 10 11 14 12

2 3 5 6 7 9 10 11 14 12
2 3 5 6 7 9 10 11 14 12
2 3 5 6 7 9 10 11 14 12
2 3 5 6 7 9 10 11 14 12
2 3 5 6 7 9 10 11 14 12
2 3 5 6 7 9 10 11 12 14

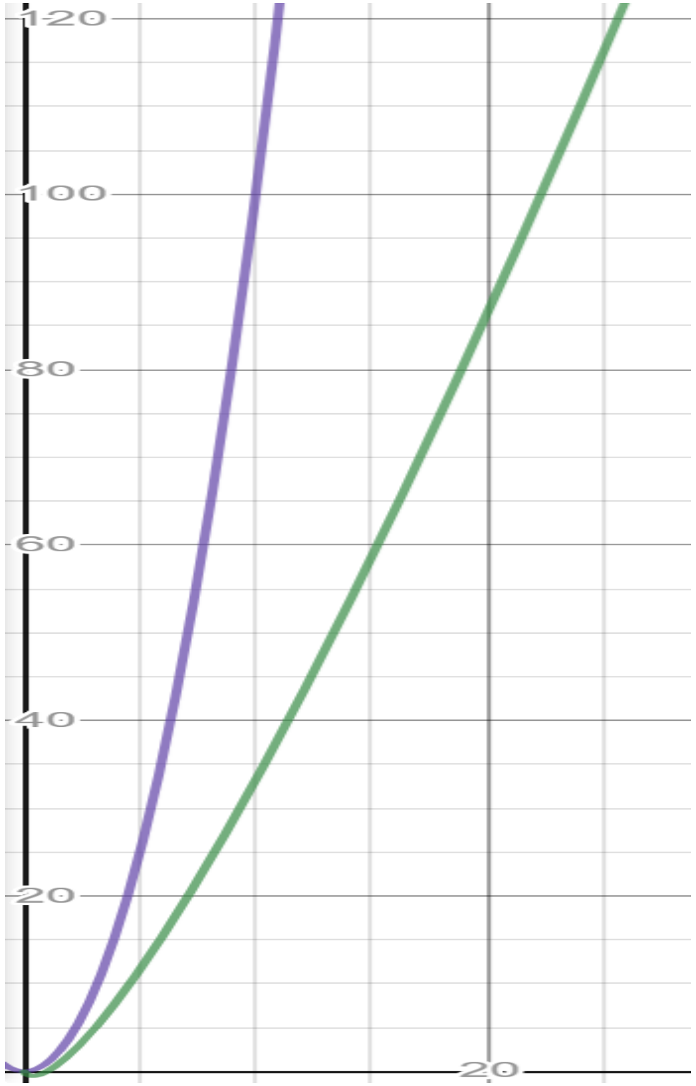
Quick Sort Analysis

Quick Sort's worst case run time is

$\Theta(n^2)$

Quick Sort's best case run time is

$\Theta(n\log_2 n)$



Quick Sort Analysis

So why think about Quick Sort when Merge Sort is at least as good?

Because the constant factor hidden in the big-Θ notation for Quick Sort is quite good.

In practice, Quick Sort outperforms Merge Sort and it significantly outperforms Selection Sort and Insertion Sort.

Quick Sort Analysis

How is it that Quick Sort's worst-case and best-case running times differ?

Let's start by looking at the worst-case running time.

Suppose that we're really unlucky and the partition sizes are really unbalanced.

In particular, suppose that the pivot chosen by the partition function is always either the smallest or the largest element in the n element subarray.

Quick Sort Analysis

In particular, suppose that the pivot chosen by the partition function is always either the smallest or the largest element in the *n* element subarray.

Let's start with the case there the pivot is the largest element

{1,2,4,6,7,8,14,18,19}

Pivot would be 19 so all elements are less than pivot so there would be multiple swaps of numbers with themselves but 19 would still remain on the far right.

{1,2,4,6,7,8,14,18} and {}.

{1,2,4,6,7,8,14} and {}.

{1,2,4,6,7,8} and {}

{1,2,4,6,7} and {}

{1,2,4,6} and {}

{1,2,4} and {}

{1,2} and {}

Quick Sort Analysis

As we can see, one of the partitions will contain no elements and the other partition will contain $n - 1$ (all but the pivot) every time.

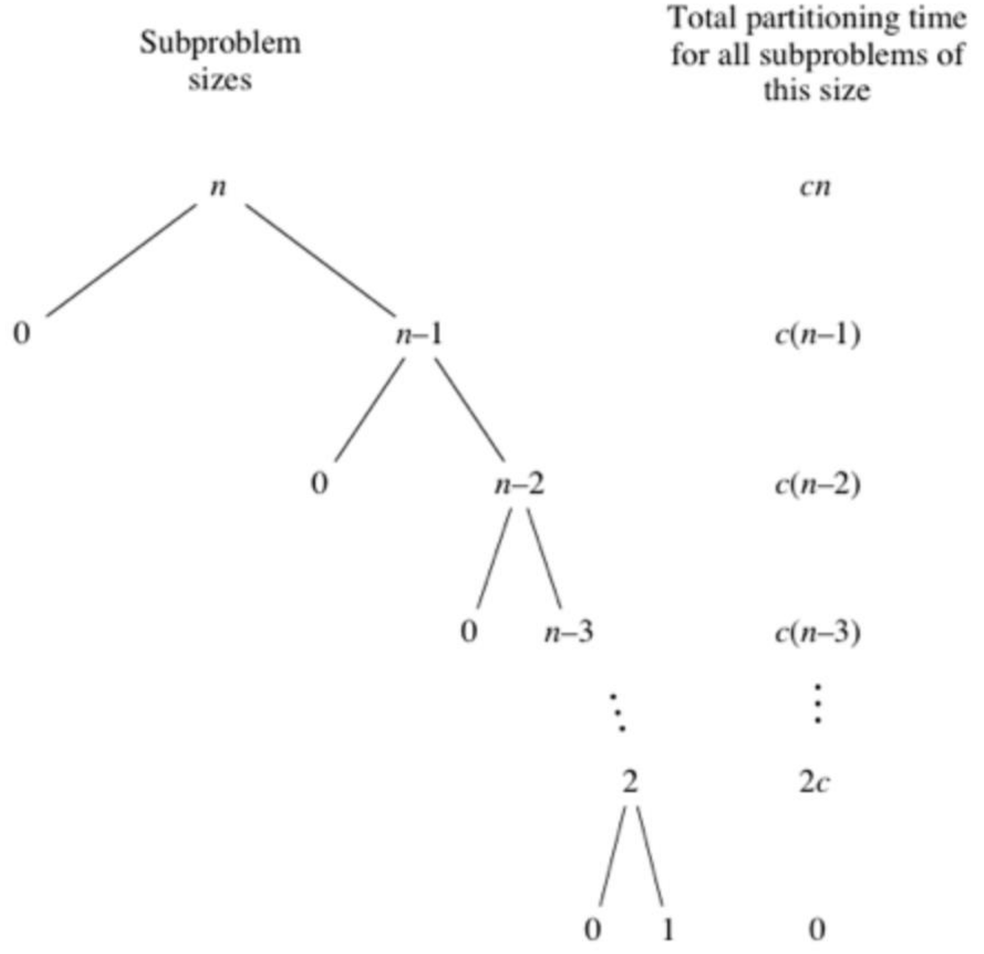
So the recursive calls will be on subarrays of sizes 0 and $n - 1$.

In this situation, divide and conquer with recursion does not help the run time so we don't get the benefit of it either - our runtime suffers.

An array in reverse sorted order would have the same issue.

Quick Sort Analysis

When Quick Sort always has the most unbalanced partitions possible, then the original call takes *cn* time for some constant *c*, the recursive call on *n*−1 elements takes *c*(*n*−1), the recursive call on *n*−2 elements takes *c*(*n*−2) time and so on … until we get to the final 2 elements.



Quick Sort Analysis

If n is 4

c(4 + (4-1) + (4-2) + (4-3)) =
c(4 + 3 + 2 + 1) = 10

c($\frac{n(n+1)}{2}$)= $\frac{4(4+1)}{2}$ = 10

We can add up the runtime of those partition steps

$cn + c(n-1) + c(n-2) + \cdots + 2c = c(n + (n-1) + (n-2) + \cdots + 2) =$

This pattern indiciates an arithmetic series that we sum with $\frac{n(n+1)}{2}$

Since our pattern ends with 2 instead of 1, we subtract 1

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

Worst case run time

Quick Sort Analysis

What does the best case look like?

Quick Sort's best case occurs when the partitions are as evenly balanced as possible

their sizes either are equal or are within 1 of each other.

Quick Sort Analysis

their sizes are either equal or are within 1 of each other.

if the subarray has an odd number of elements and the pivot is right in the middle after partitioning and each partition has $\frac{n-1}{2}$ elements

if the subarray has an even number of elements and one partition has $\frac{n}{2}$ elements with the other one having $\frac{n}{2}-1$

Quick Sort Analysis

In either of these cases, each partition has at most $\frac{n}{2}$ elements

The tree of subproblem sizes looks a lot like the tree of subproblem sizes for Merge Sort…

this is where $\log_2 n$ was introduce to the runtime

The partitioning times look like the merging times…

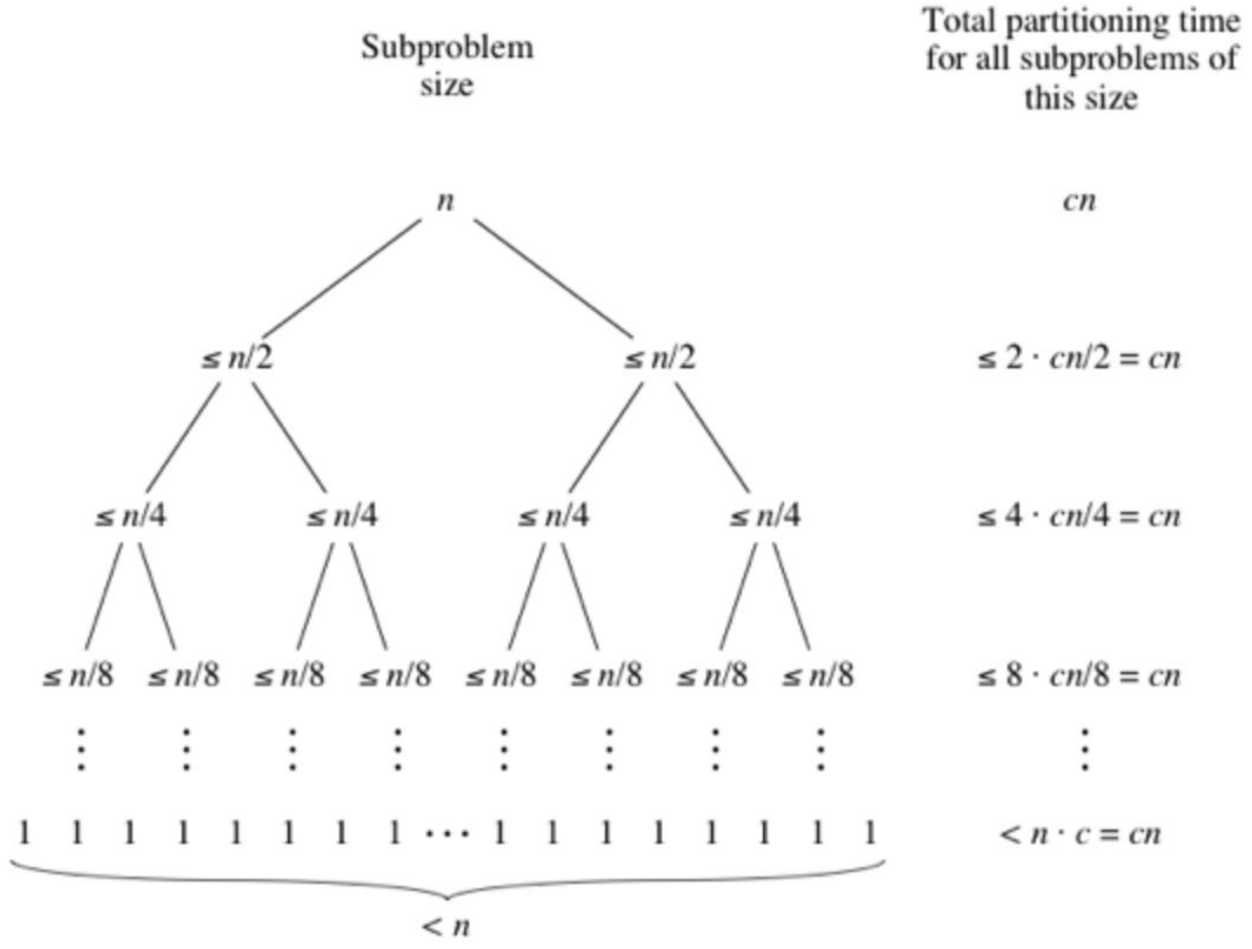
this is where *n* was introduced to the runtime

Quick Sort Analysis

This pattern tell us the same thing it told us about MergeSort.

Partitioning time (n) * number of partitions to make ($\log_2 n$)

$O(n\log_2 n)$



Suppose we run QuickSort on some input, and, magically, every recursive call chooses the median element of its subarray as its pivot. What's the running time in this case?

- Not enough information to answer question
- $\Theta(n)$
- $\Theta(n \log n)$
- $\Theta(n^2)$

Fix two elements of the input array. How many times can these two elements be compared during the execution of QuickSort ?

- 0
- 0 or 1
- 0 or 1 or 2
- Any number in between 0 and n-1

{9, 7, 15, 16, 12}	12 is pivot
{9, 7, 15, 16, 12}	9 < 12 swap 9 with 9
{9, 7, 15, 16, 12}	7 < 12 swap 7 and 7
	15 $\not<$ 12 so no swap
	16 $\not<$ 12 so no swap
	final swap of 15 and 12
	recursive call (left and right)
{9, 7} 12, {16, 15}	7 is pivot in left/15 is pivot in right
	9 $\not<$ 7 so no swap
	final swap of 9 and 7
{7,9} 12 {16,15}	16 $\not<$ 15 so no swap
	final swap of 16 and 15
{7,9} 12 {15,16}	

Quick Sort Analysis

Worst case is $\Theta(n^2)$ and best case is $\Theta(n\log_2 n)$

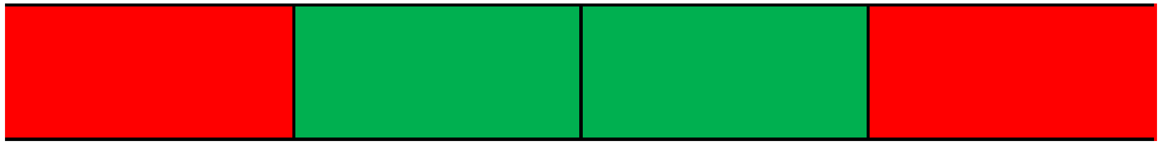
What is an average case?

In the average case, all elements are equally likely to be chosen as the pivot.

Quick Sort Analysis

Worst case is $\Theta(n^2)$ and best case is $\Theta(n\log_2 n)$

After partitioning, we would expect half the time the pivot to end up in the middle two quarters and half the time for it to end up in the outer two quarters.



It can be proven mathematically that this will result in a runtime of $\Theta(n\log_2 n)$ for the average case.

OLQ

- The Quick Sort code from lecture will be provided.

- You will be provided with a copy of this table for reference.

i	j	pivot	low	high

OLQ

You will be given an array to work with (for example)

{9, 7, 5, 12, 11}

Assume that a function to print the entire array is included the line after each call to function `swap()` inside `partition()`.

This call will be in the code provided with the quiz itself and will be in the OLQ review.

OLQ

```
int partition (int A[], int low, int high)
{
   int i, j = 0;
   int pivot = A[high];

   i = (low - 1);

   for (j = low; j < high; j++)
   {
      if (A[j] < pivot)
      {
         i++;
         swap(&A[i], &A[j]);
         printArray(A);
      }
   }
   swap(&A[i + 1], &A[high]);
   printArray(A);

   return (i + 1);
}
```

OLQ

For your given array, you will need to write the output of every call to this function - what does the array look like after each call to function `swap()`?

For example, if your given array is

`{9,7,5,12,11}`

then your answer will be

loop swap 9 9	{9,7,5,12,11}
loop swap 7 7	{9,7,5,12,11}
loop swap 5 5	{9,7,5,12,11}
final swap 12 11	{9,7,5,11,12}
final swap 9 5	{5,7,9,11,12}
loop swap 7 7	{5,7,9,11,12}
final swap 9 9	{5,7,9,11,12}

OLQ

{9, 7, 5, 12, 11}

loop swap 9 9	{9,7,5,12,11}
loop swap 7 7	{9,7,5,12,11}
loop swap 5 5	{9,7,5,12,11}
final swap 12 11	{9,7,5,11,12}
final swap 9 5	{5,7,9,11,12}
loop swap 7 7	{5,7,9,11,12}
final swap 9 9	{5,7,9,11,12}

```
i = low -1

for (j = low < high)
{
  if A[j] < pivot
     move i
     swap A[i] A[j]
     print
}
swap A[i+1] with A[high]
print
```

OLQ

```
{7,12,5,9,11}
```

loop swap 7 7	{7,12,5,9,11}
loop swap 12 5	{7,5,12,9,11}
loop swap 12 9	{7,5,9,12,11}
final swap 12 11	{7,5,9,11,12}
loop swap 7 7	{7,5,9,11,12}
loop swap 5 5	{7,5,9,11,12}
final swap 9 9	{7,5,9,11,12}
final swap 7 5	{5,7,9,11,12}

```
i = low -1

for (j = low -> j < high)
{
  if A[j] < pivot
    move i
    swap A[i] A[j]
    print
}
swap A[i+1] with A[high]
print
```

OLQ

Coding Assignment 4 will be using Quick Sort.

I HIGHLY recommend that you go ahead and start on the Coding Assignment and get the Quick Sort code work on a small hardcoded array. Add the print statements as shown here in the slides.

Practice by giving yourself an array and figuring out the prints. Then, run your program and check your answer.

Coding Assignment 4

Make a copy of Coding Assignment 3. Remove the Merge Sort and Insertion Sort code.

Add a Quick Sort function to your code.

You will run your code on the same files from Coding Assignment 3.

There will be other things to do but get this working to practice for the OLQ.

Coding Assignment 4

You will be making a copy of Coding Assignment 3.

Replace the MergeSort and Insertion Sort with Quick Sort.

The Quick Sort we did in class used the rightmost element as the pivot.

You will be using conditional compile to make your program run 3 different ways.

Coding Assignment 4

QSR

Run Quick Sort with rightmost element as pivot.

QSM

Run Quick Sort with middle element as pivot

QSRND

Run Quick Sort with a random pivot element

Coding Assignment 4

```
#ifdef QSM
int middle = (high+low)/2;
swap(&A[middle], &A[high]);
#elif QSRND
int random = (rand() % (high-low+1)) + low;
swap(&A[random], &A[high]);
#endif
int pivot = A[high];
```