

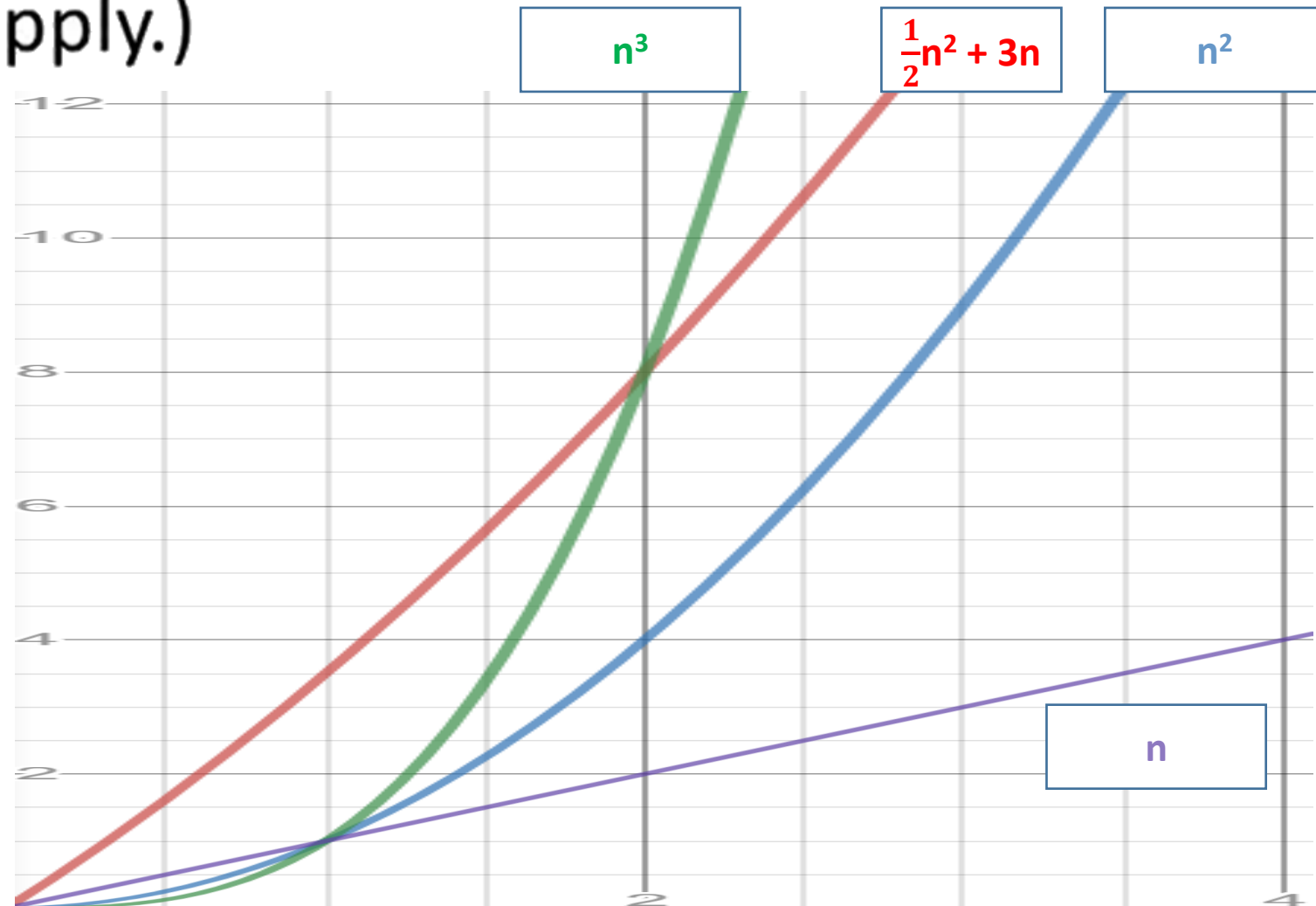
CSE 2320

Week of 06/29/2020

Instructor : Donna French

Let $T(n) = \frac{1}{2}n^2 + 3n$. Which of the following statements are true? (Check all that apply.)

- ☐ $T(n) = O(n)$.
- ☒ $T(n) = \Omega(n)$.
- ☒ $T(n) = \Theta(n^2)$.
- ☒ $T(n) = O(n^3)$.



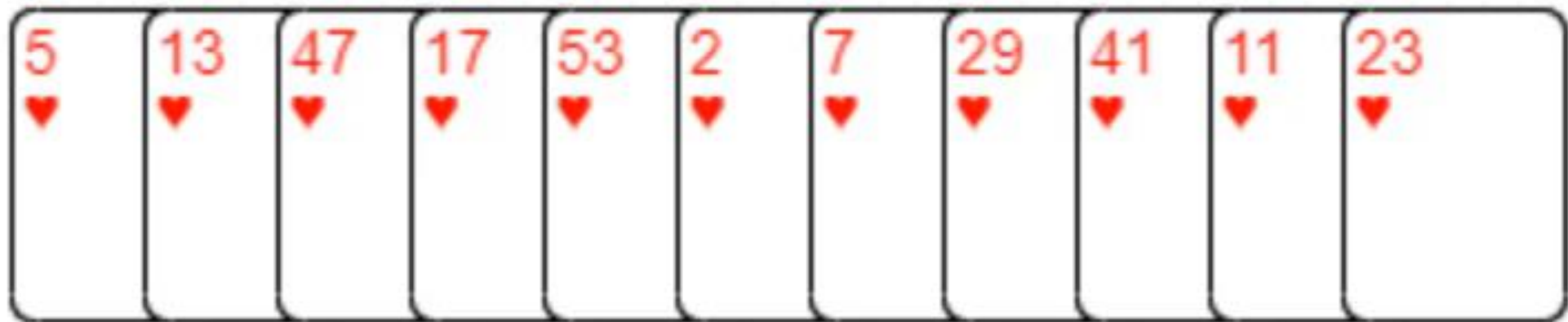
Selection Sort

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

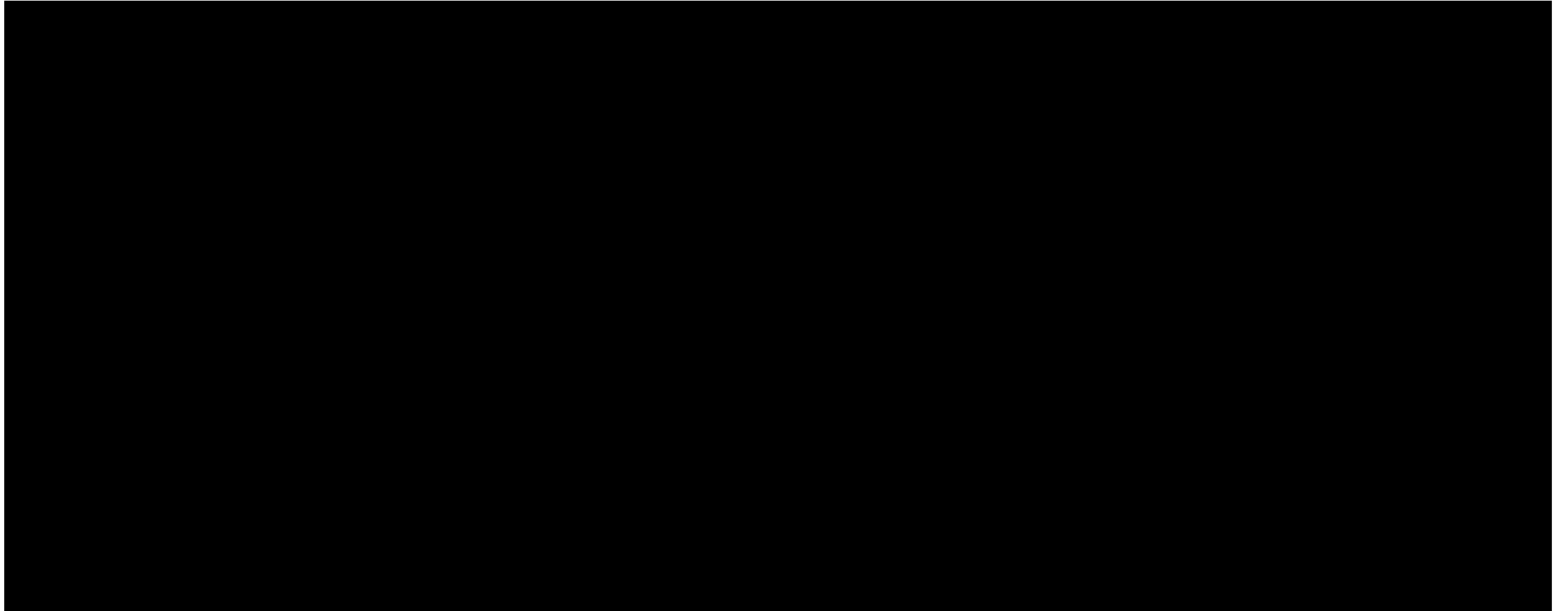


Selection Sort

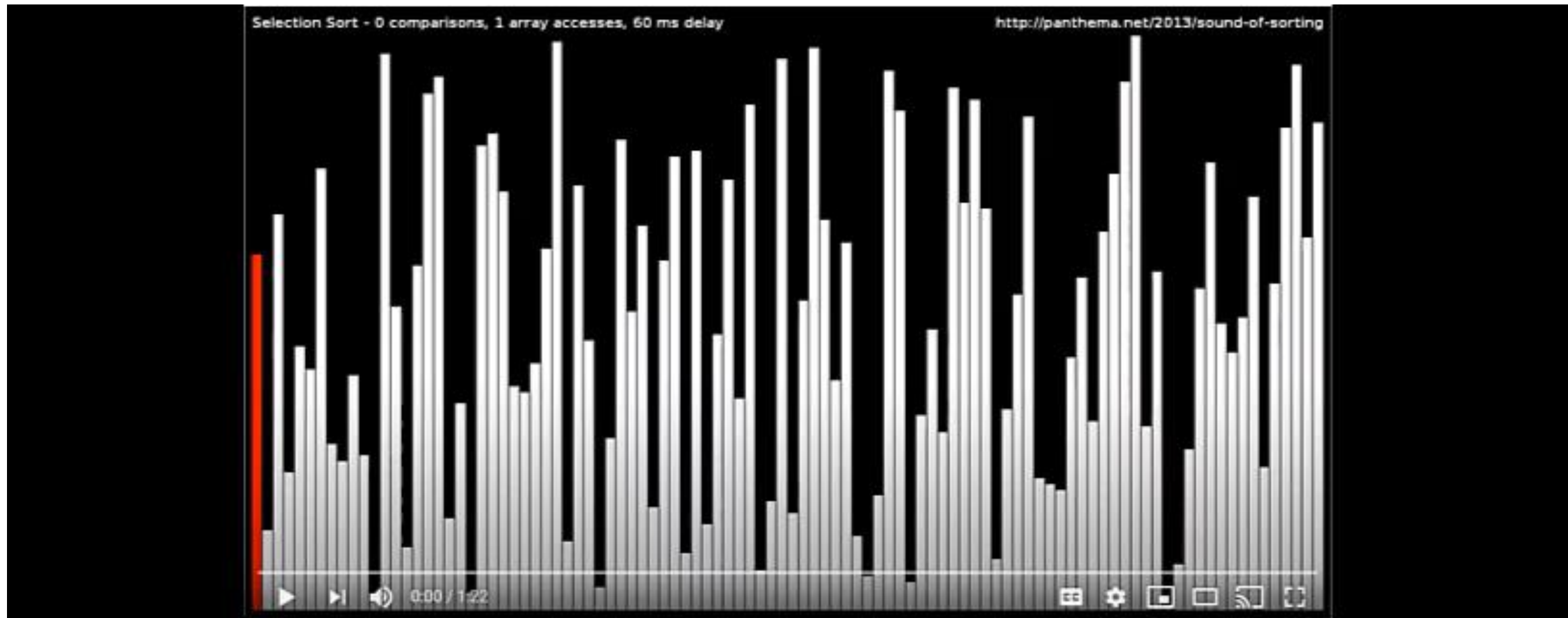
1. Find the smallest card. Swap it with the first card.
2. Find the second-smallest card. Swap it with the second card.
3. Find the third-smallest card. Swap it with the third card.
4. Repeat finding the next-smallest card and swapping it into the correct position until the array is sorted.

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?

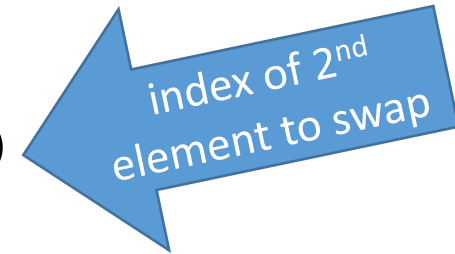
Keep those questions in mind as we analyze this algorithm.

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...

```
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 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

A screenshot of a terminal window. The title bar at the top reads "student@cse1325: /media/sf_VM2320" and includes standard window controls (minimize, maximize, close). Below the title bar is a menu bar with "File", "Edit", "Tabs", and "Help". The main area of the terminal shows a shell prompt "student@cse1325:/media/sf_VM2320\$" in green and blue text, followed by a black cursor. The terminal is flanked by black vertical bars on both sides.

```
student@cse1325: /media/sf_VM2320
File Edit Tabs Help
student@cse1325:/media/sf_VM2320$
```

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

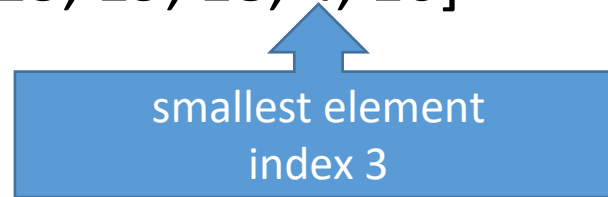
[13, 19, 18, 4, 10]

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]



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.

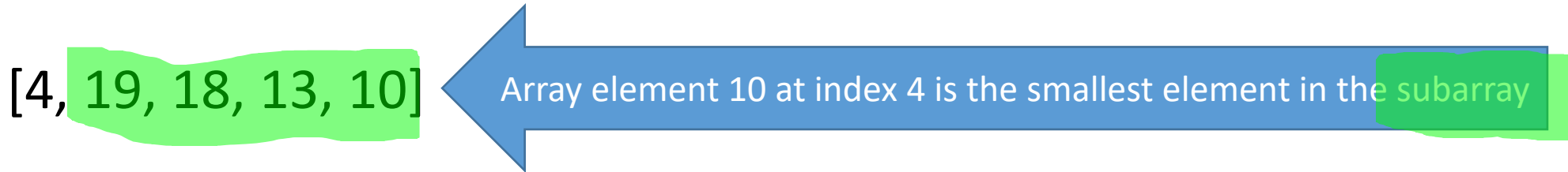
[4, 19, 18, 13, 10]

[13, 19, 18, 4, 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.



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.

```
13 int main(void)
14 {
15     int A[] = {64, 25, 12, 22, 11, 32, 67, 23, 99};
16     int n = sizeof(A)/sizeof(A[0]);
17     int min_ndx = 0;
18     int sub_start = 0;
19     int j = 0;
20
21     printArray(A, n);
22
23     printf("Enter index of start of subarray : ");
24     scanf("%d", &sub_start);
25
26     min_ndx = sub_start;
27
28     for (j = min_ndx+1; j < n; j++)
29     {
30         if (A[j] < A[min_ndx])
31             min_ndx = j;
32     }
33
34     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.

```
4 void swap(int *xp, int *yp)
5 {
6     int temp = *xp;
7     *xp = *yp;
8     *yp = temp;
9 }
10
11 void selectionSort(int A[], int n)
12 {
13     int i, j, min_idx;
14
15     for (i = 0; i < n-1; i++)
16     {
17         min_idx = i;
18         for (j = i+1; j < n; j++)
19         {
20             if (A[j] < A[min_idx])
21                 min_idx = j;
22         }
23
24         swap(&A[min_idx], &A[i]);
25     }
26 }
```

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

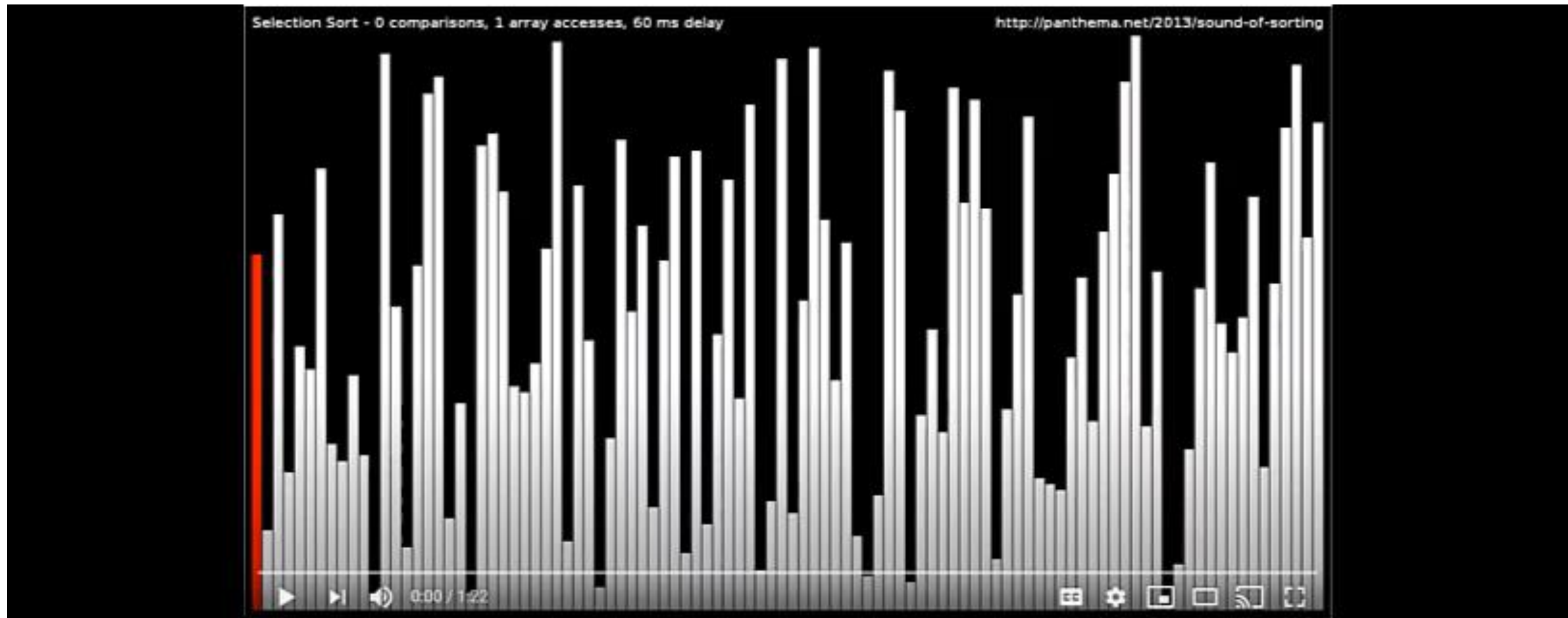
```
void selectionSort(int A[], int n)
{
    int i, j, min_idx;
    for (i = 0; i < n-1; i++)
    {
        min_idx = i;
        for (j = i+1; j < n; j++)
        {
            if (A[j] < A[min_idx])
                min_idx = j;
        }
        swap(&A[min_idx], &A[i]);
    }
}
```

{1, 4, 9, 13}

n = 4

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

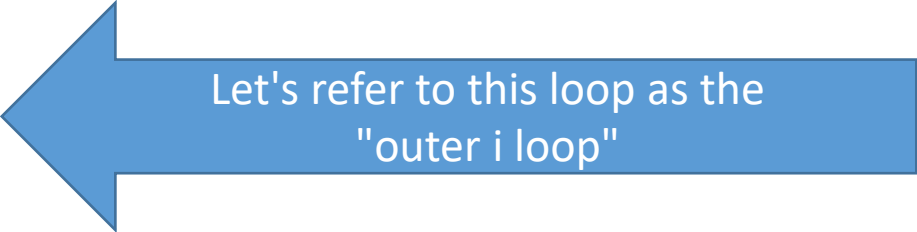
Selection Sort



Analysis of Selection Sort

Selection Sort loops over indices in the array

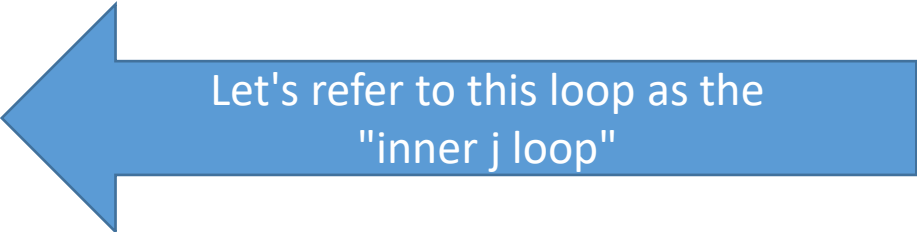
```
for (i = 0; i < n-1; i++)
```



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.

2nd 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

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
4 th call ($i = 3$) for array of 8 elements	4 times
5 th call ($i = 4$) for array of 8 elements	3 times
6 th call ($i = 5$) for array of 8 elements	2 times
7 th call ($i = 6$) for array of 8 elements	1 time
8 th 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

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

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

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

$$1 + 2 + 3 + 4 + 5 = (1 + 5) + (2 + 4) + 3 = 2.5 \text{ pairs} \Rightarrow 2.5 * 6 = 15$$

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

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 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, $1 + 2 + \dots + (n-1) + n$ 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 $\frac{n^2+n}{2}$

$$c_1 \left(\frac{n^2+n}{2} \right) = c_1 \left(\frac{1}{2}n^2 + \frac{1}{2}n \right) = \frac{1}{2}c_1(n^2 + n) = \frac{1}{2}c_1n^2 + \frac{1}{2}c_1n$$

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?

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?

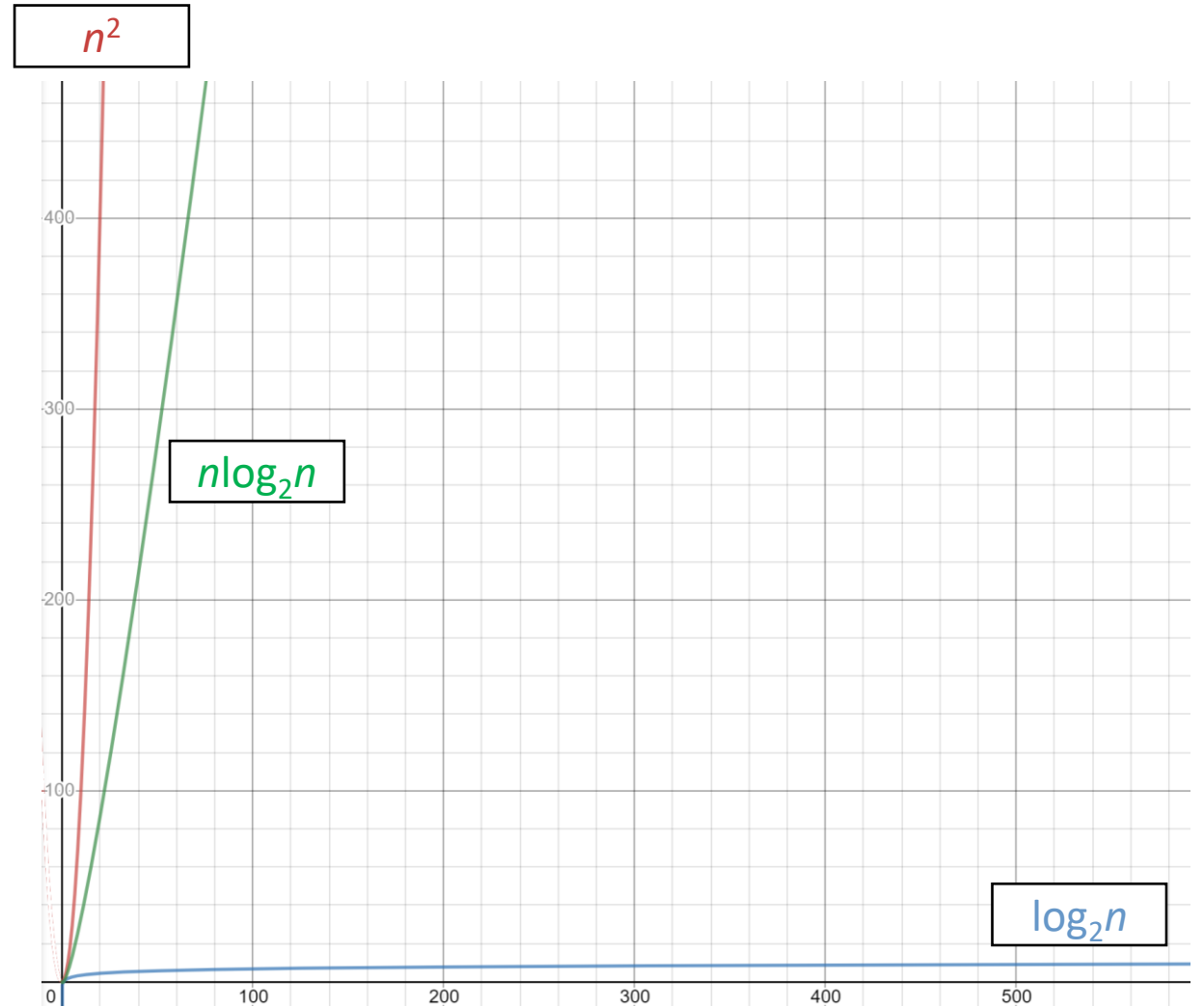
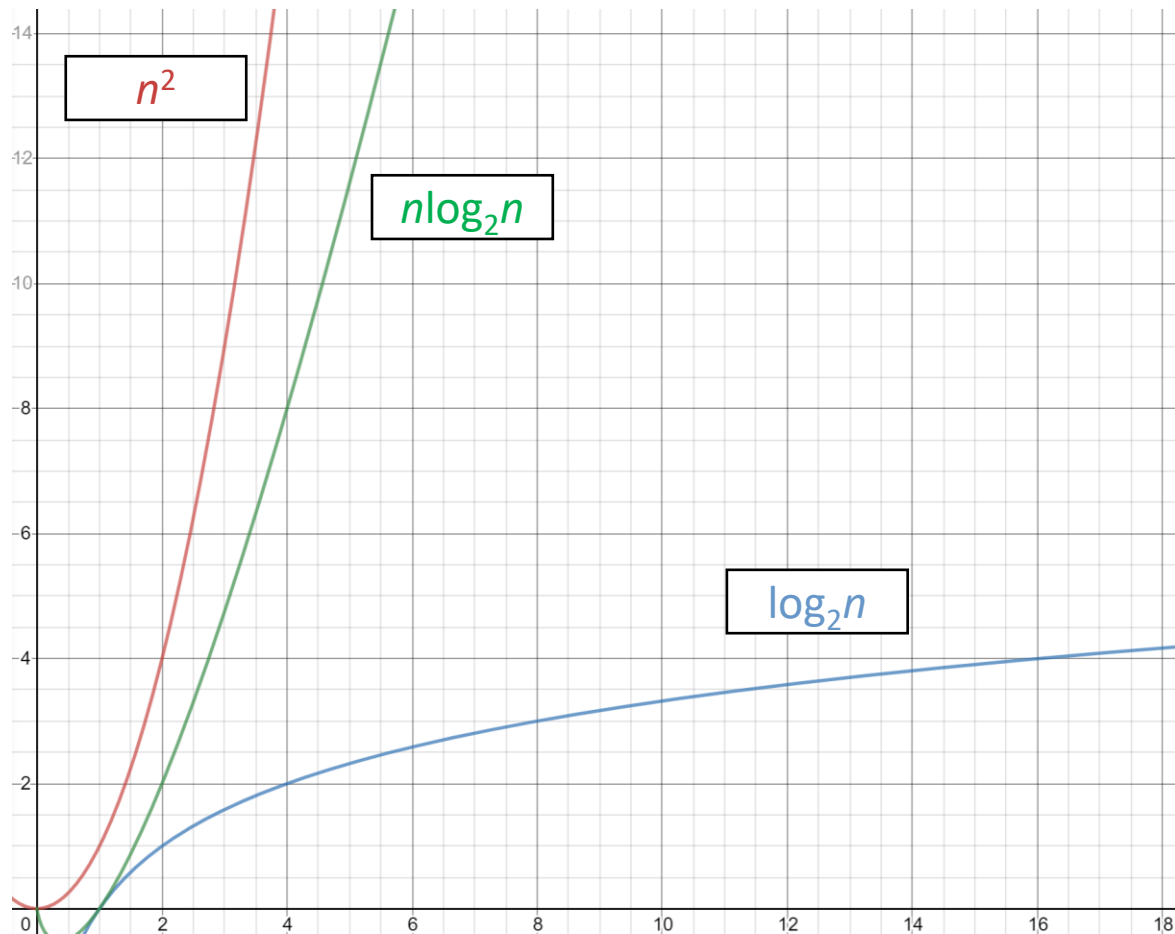
Does it run slower for an array in reverse sorted order?

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 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.

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

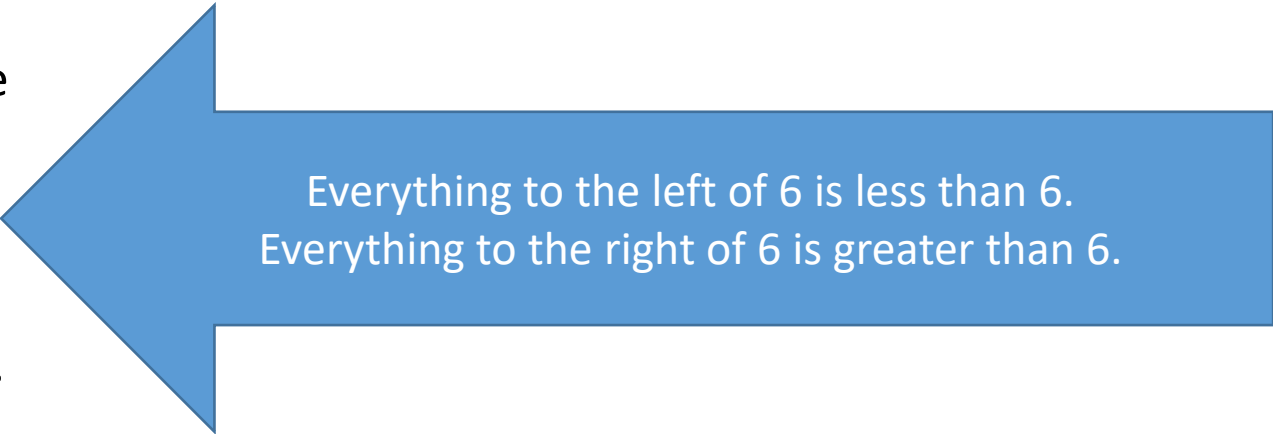
[9, 7, 5, 11, 12, 2, 14, 3, 10, 6]

then we choose 6 as the pivot.

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

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

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



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

Quick Sort

Conquer

Recursively sort the subarrays $\text{array}[p \dots q-1]$

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

and $\text{array}[q+1 \dots r]$

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)
```

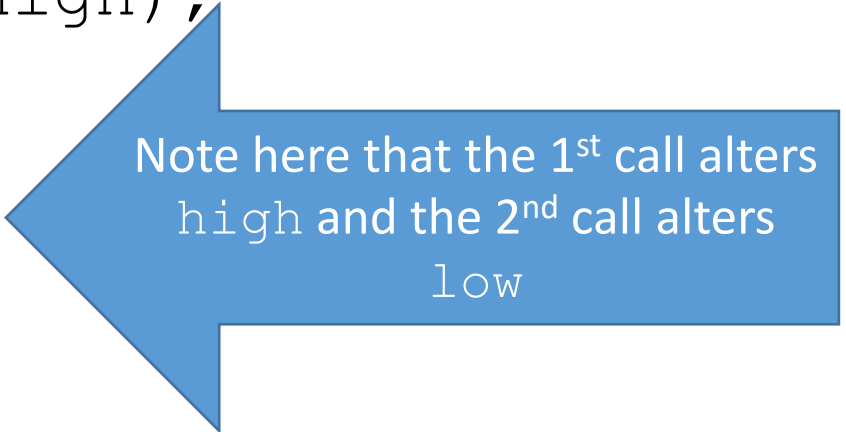
```
ndx = ?
```

```
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;
}
```

```

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

```
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);
    }
}
```

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

partition(0, 3)
 ndx = 2

{ 6, 5, 7, 9}

QuickSort(0, 1)

QuickSort(3, 3)

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)
```

```
QuickSort(3,3)
```

```
    low = 3
```

```
    high = 3
```

```
    low < high
```

```
    return
```

```

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)
```

```
QuickSort (3,3)  
    low = 3  
    high = 3  
    low  $\nless$  high  
    return
```

```
{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(3,3)  
    low = 3  
    high = 3  
    low <= high  
    return
```

```
QuickSort(0,-1)  
{5, 6, 7, 9}  
    low = 0  
    high = -1  
    low <= high  
    return
```

```
QuickSort(1,1)  
{?, ?, ?, ?}  
    low = 1  
    high = 1  
    low <= high  
    return
```

Quick Sort

{9, 6, 5, 7}

7 is pivot

9 \nless 7 \Rightarrow no swap

6 $<$ 7 \Rightarrow swap 9 and 6

{6, 9, 5, 7}

5 $<$ 7 \Rightarrow swap 9 and 5

{6, 5, 9, 7}

final \Rightarrow 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 \nless high)

{6, 5}

5 is pivot

6 \nless 5 \Rightarrow no swap

final \Rightarrow swap 6 and 5

{5, 6, 7, 9}

Quick Sort

{6, 9, 7, 5}

5 is pivot

6 \nless 5 \Rightarrow no swap

9 \nless 5 \Rightarrow no swap

7 \nless 5 \Rightarrow no swap

final \Rightarrow swap 6 and 5

{5, 9, 7, 6}

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

{} {5} {9, 7, 6}

nothing to the left of 5

{9, 7, 6}

6 is pivot

9 \nless 6 \Rightarrow no swap

7 \nless 6 \Rightarrow no swap

final \Rightarrow 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);
}

```

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

partition(A, 2, 3)

i	j	pivot	low	high