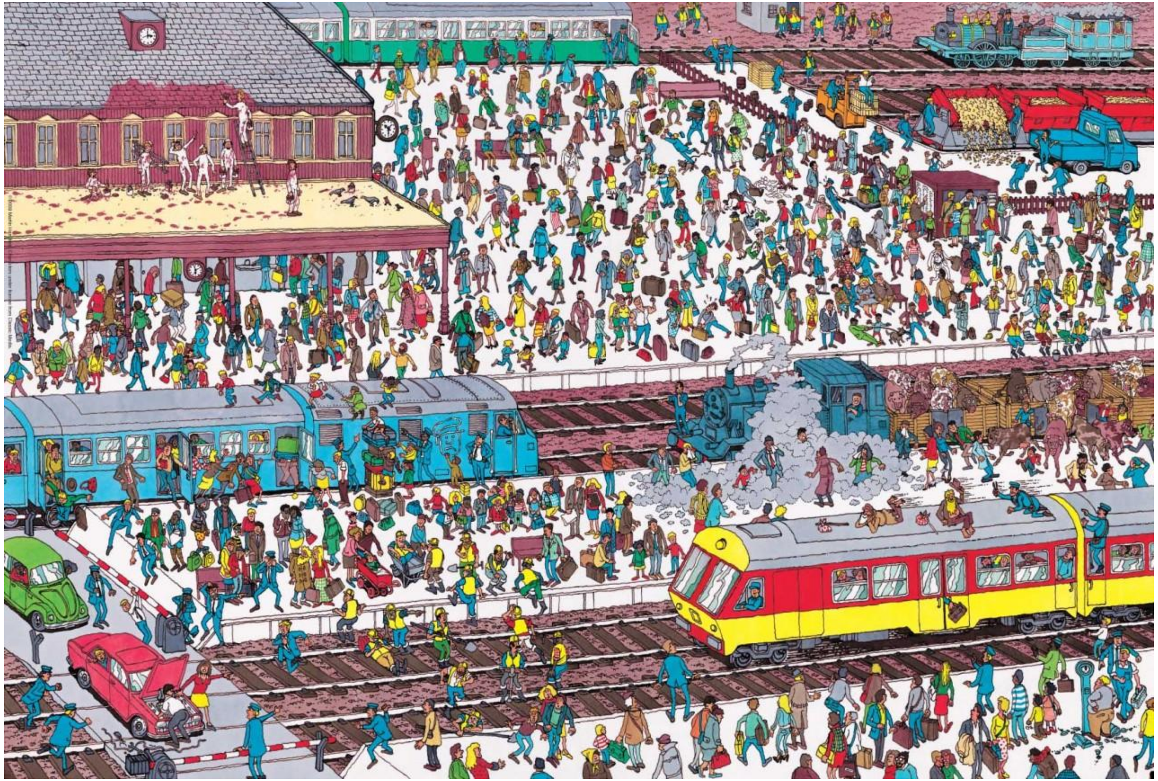
Searching, Sorting, and Algorithm Analysis

Chapter 9

Searching

Searching



Linear Search

- Often called sequential searching

Linear Search

- Often called sequential searching

- The list is searched one element at a time in order

Linear Search

- Often called sequential searching
- The list is searched one element at a time in order
  - Example: Line up every character in the Where’s Waldo picture and check if each one is Waldo
  - Always remember to stop searching when the result is found. i.e. break, return, ...

Linear Search

- How efficient is a linear search?

Linear Search

- How efficient is a linear search?
  - Where ‘N’ represents the number of elements, the algorithm may have to search all ‘N’ elements
  - What if the list is an ordered array of numbers 1 through 999 billion and we are searching for 999000000000?

Linear Search

- On average the search will find an item in N/2 attempts

Binary Search

- More efficient than linear searching

- Requirements
  - The array must be sorted

Binary Search Algorithm

- Where x is the desired item

1. Check the middle item, y
    1.1. If x == y, then return
    1.2. if x < y, go to step 1 with the smaller half of the array
    1.3. if x > y, go to step 1 with the larger half of the array

Binary Search Algorithm

Search for x = 6

1	2	3	4	5	6	7

Binary Search Algorithm

Search for x = 6

y = 4

1	2	3	4	5	6	7

Binary Search Algorithm

Search for x = 6

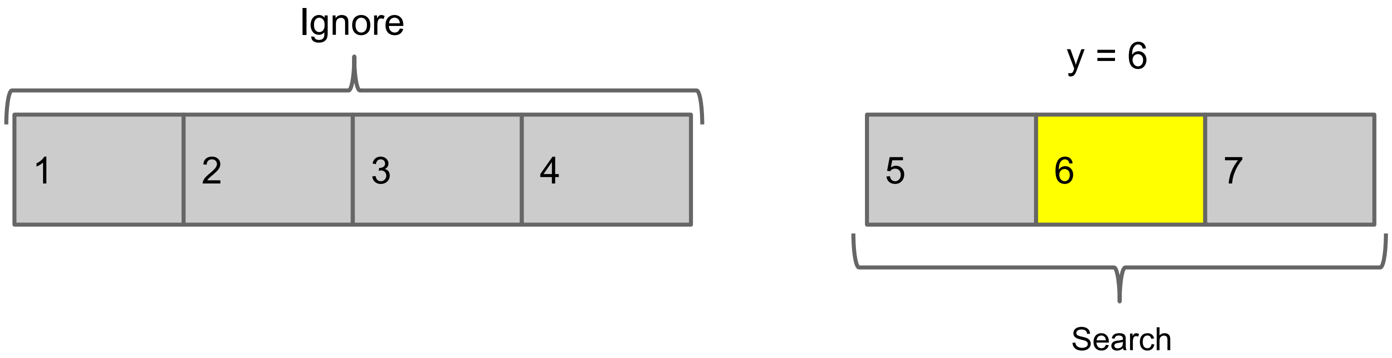
y = 4

1	2	3	4	5	6	7

Problem: 6 != 4, continue
6 > 4, so choose the larger half to search

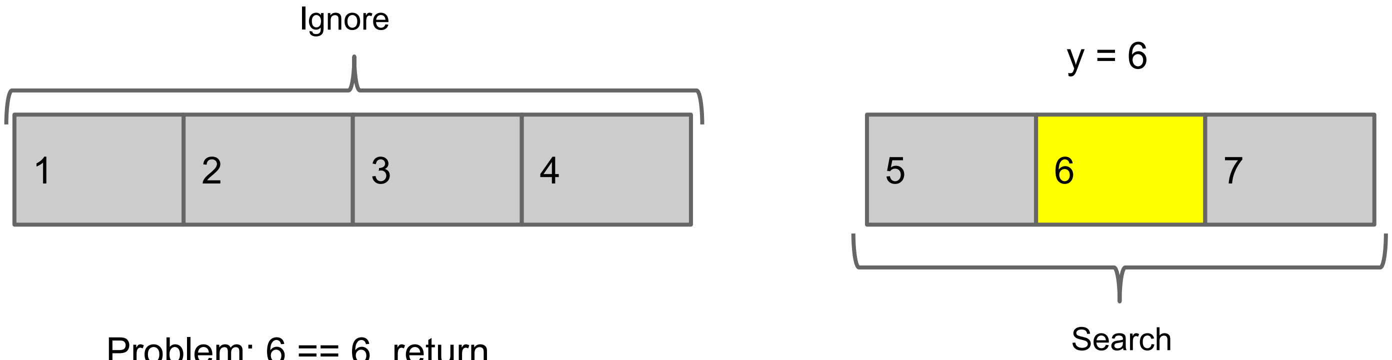
Binary Search Algorithm

Search for x = 6



Binary Search Algorithm

Search for x = 6



Problem: 6 == 6, return

Binary Search Algorithm

- Is the Binary Search more efficient than the Linear search?

Binary Search Algorithm

- Every iteration of the Binary Search Algorithm eliminates half of the list
  - List of 20,000 items
    - Linear: 20,000 possible comparisons
    - Binary search:

Binary Search Algorithm

- Every iteration of the Binary Search Algorithm eliminates half of the list
    - List of 20,000 items
        - Linear: 20,000 possible comparisons
        - Binary search: 15 possible comparisons

Binary Search Algorithm

- Every iteration of the Binary Search Algorithm eliminates half of the list
    - List of 20,000 items
        - Linear: 20,000 possible comparisons
        - Binary search: 15 possible comparisons
            - Thats a power of 2, ie Log2(20000) or log N

Searching

- Are there uses for both Linear and Binary searching?

Searching

- Are there uses for both Linear and Binary searching?

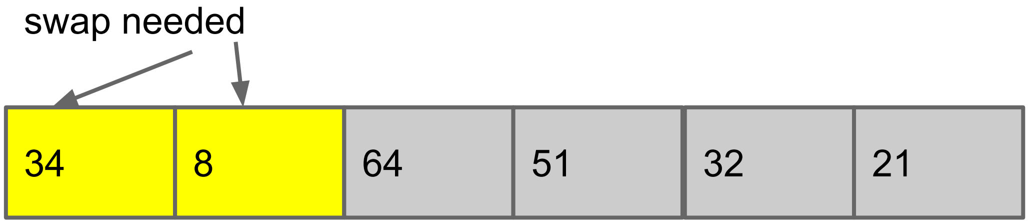
- What has to happen to take advantage of the binary search?

Sorting

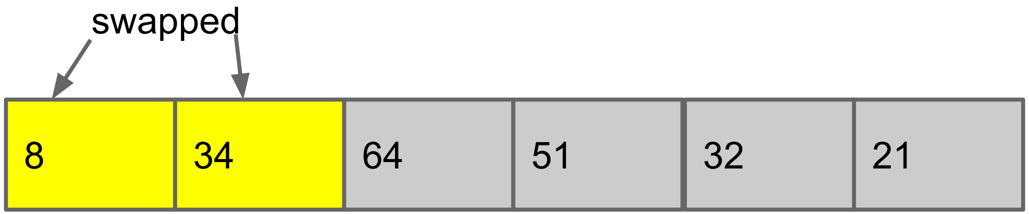
Bubble Sort

- Loop over the data N - 1 times
  - if current item is bigger than next item
    - swap

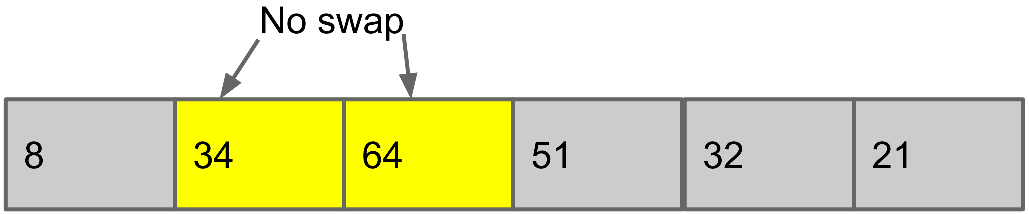
Bubble Sort



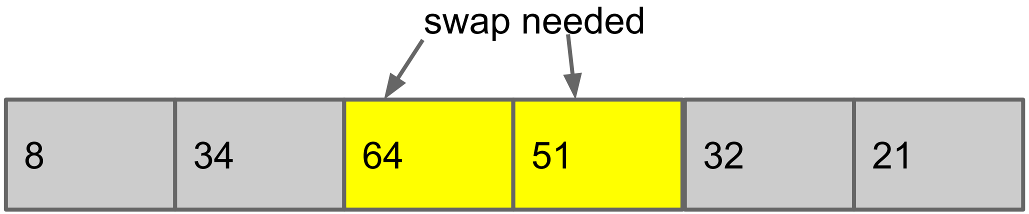
Bubble Sort



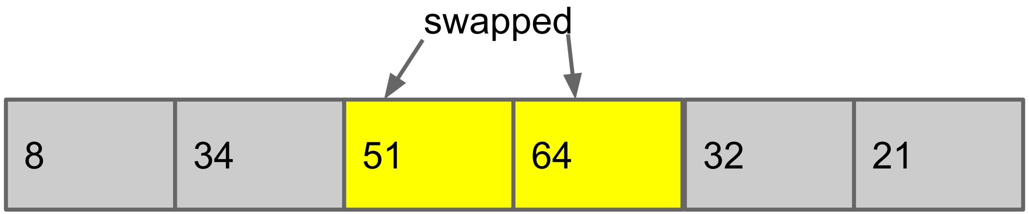
Bubble Sort



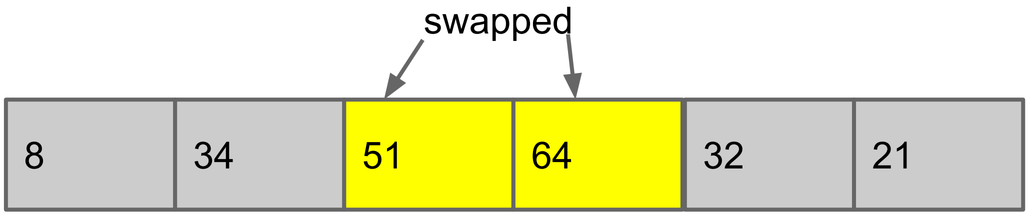
Bubble Sort



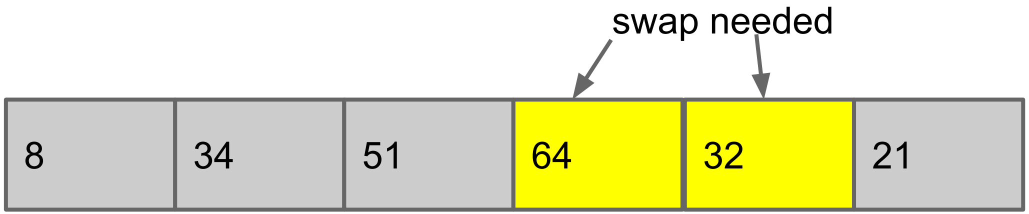
Bubble Sort



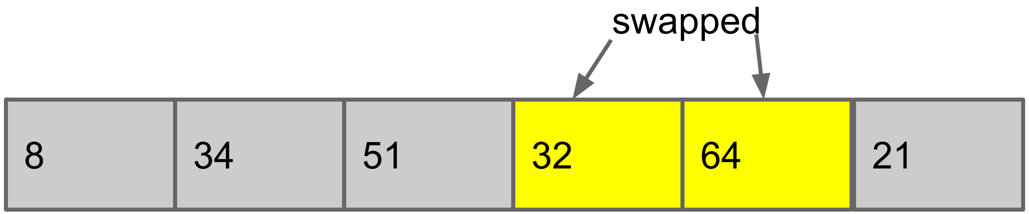
Bubble Sort



Bubble Sort



Bubble Sort

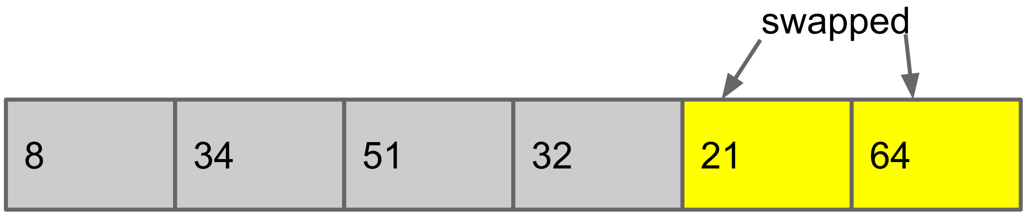


Bubble Sort

swap needed

8	34	51	32	64	21

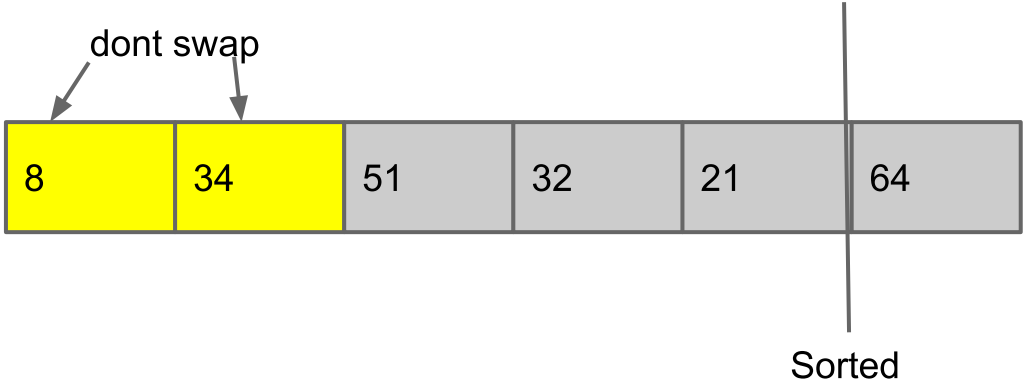
Bubble Sort



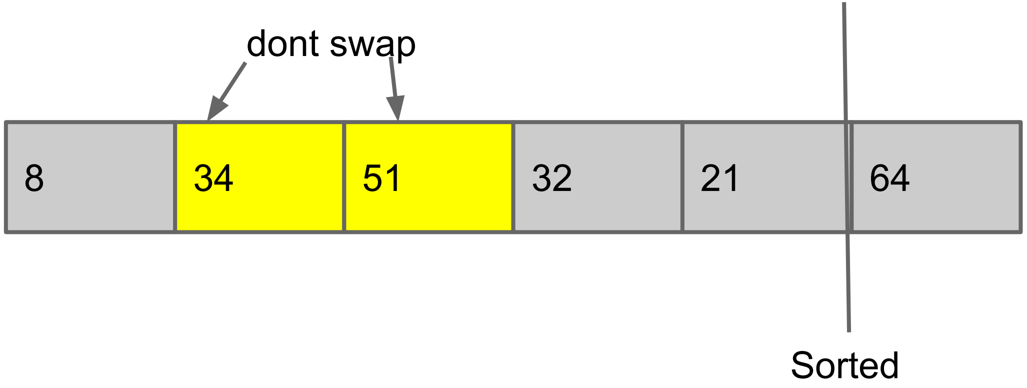
Bubble Sort

- What does the first pass guarantee?

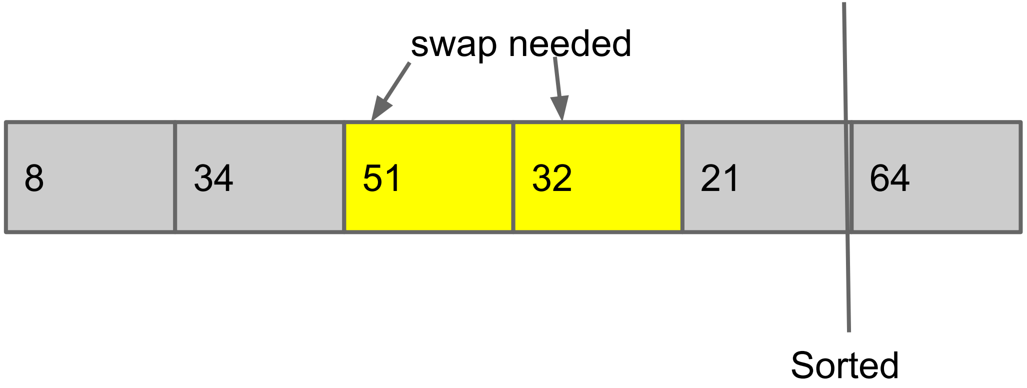
Bubble Sort



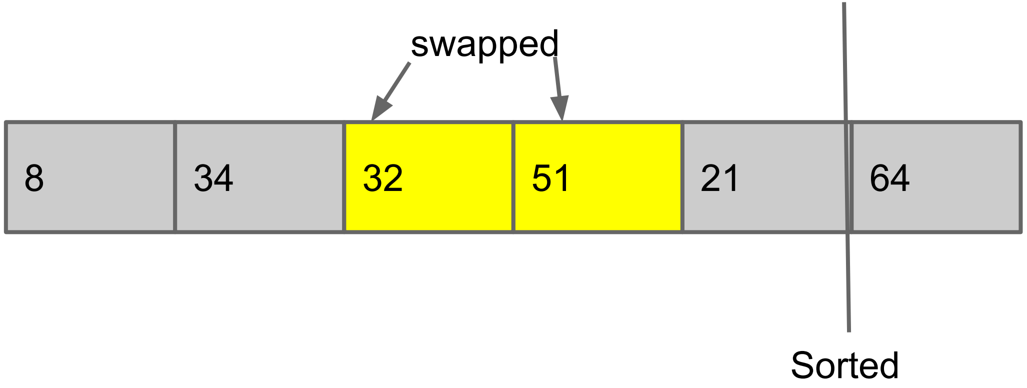
Bubble Sort



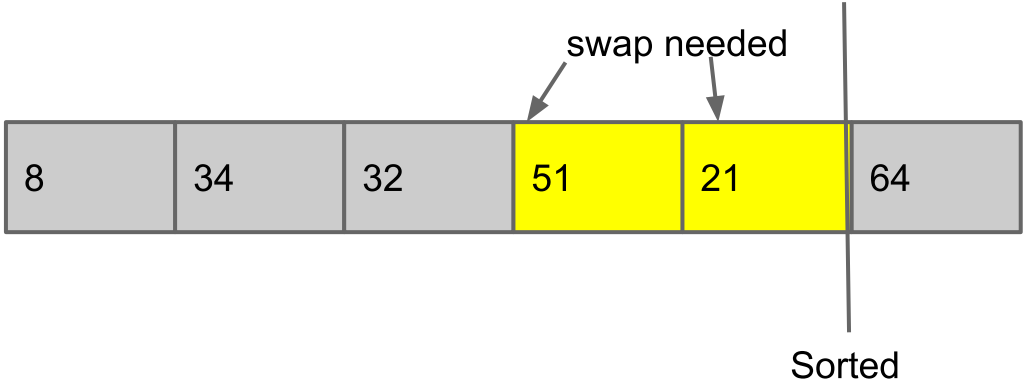
Bubble Sort



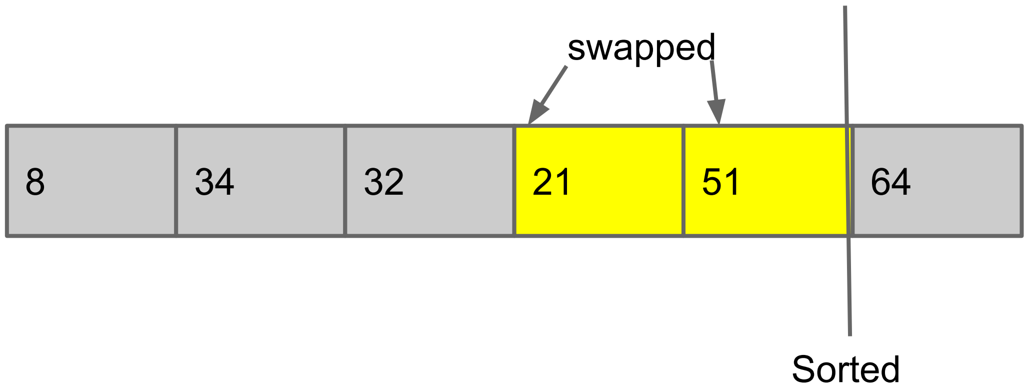
Bubble Sort



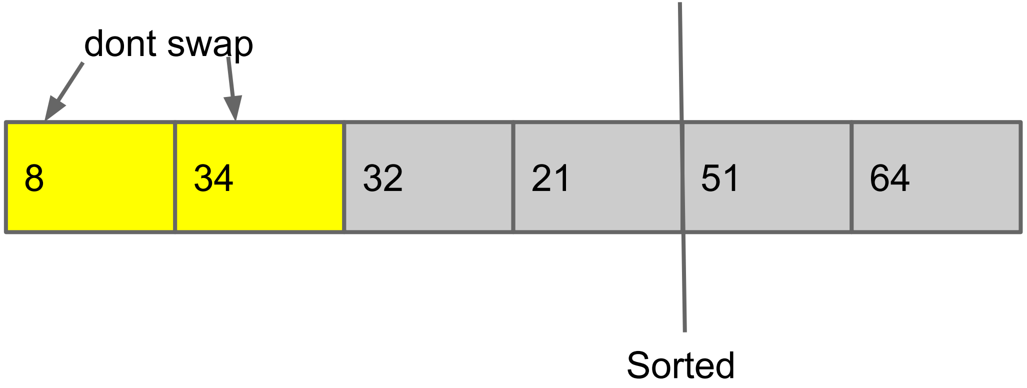
Bubble Sort



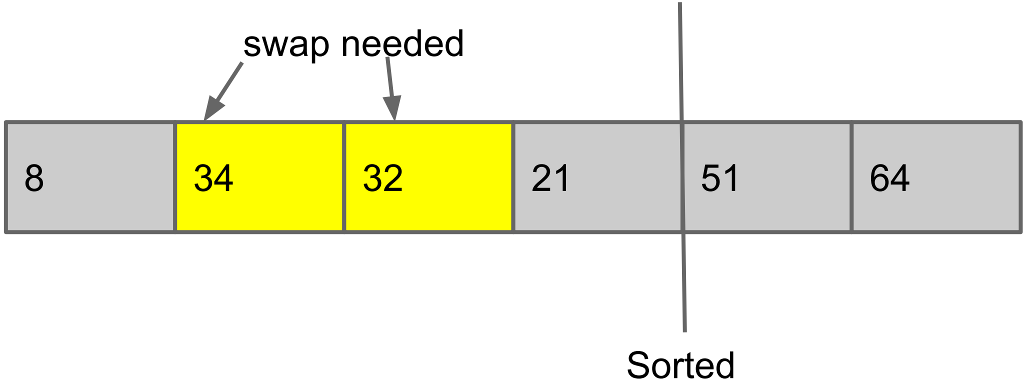
Bubble Sort



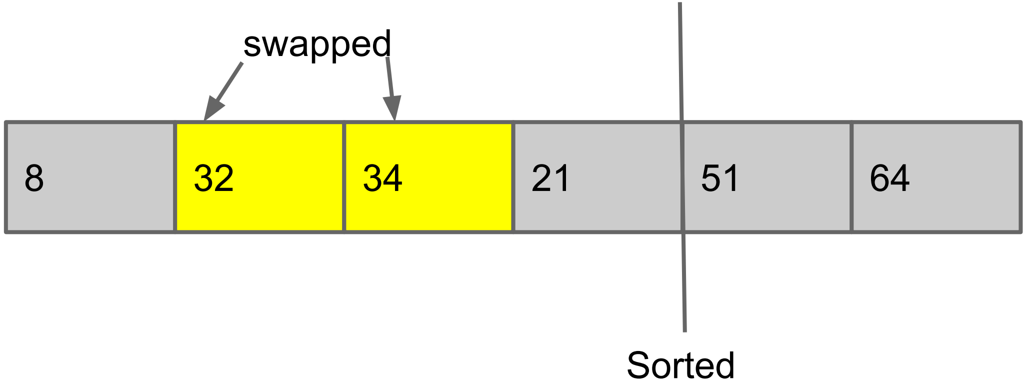
Bubble Sort



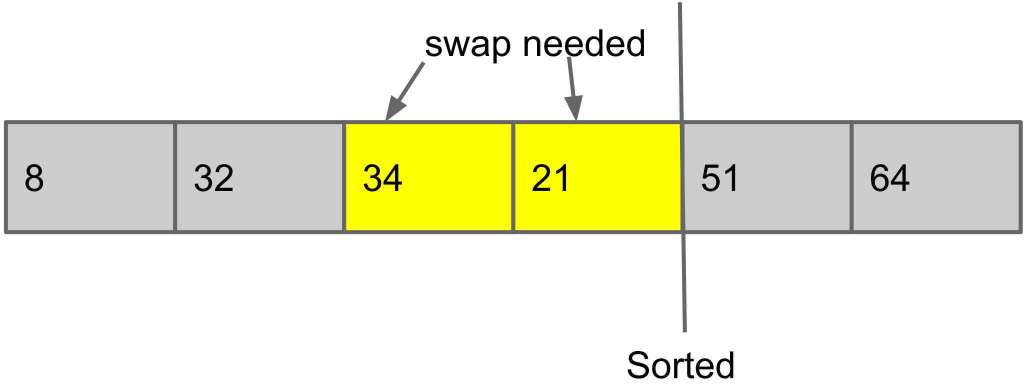
Bubble Sort



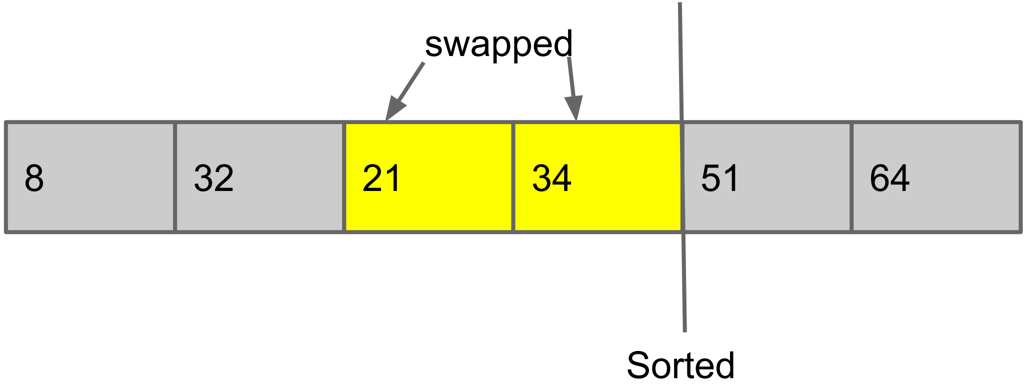
Bubble Sort



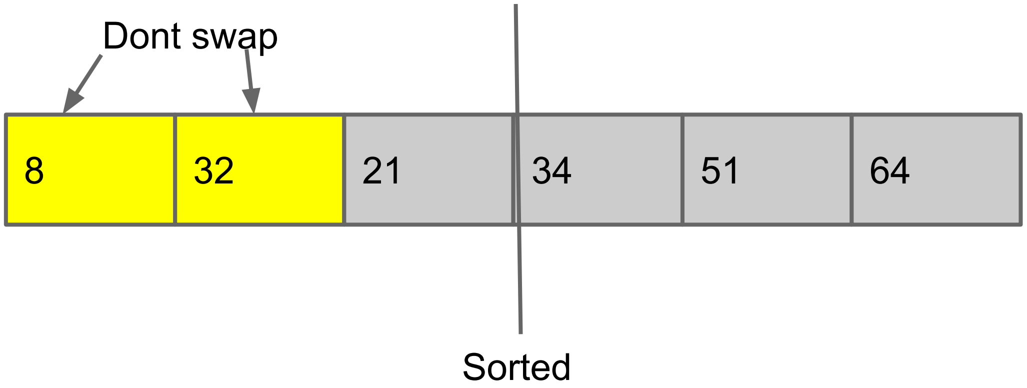
Bubble Sort



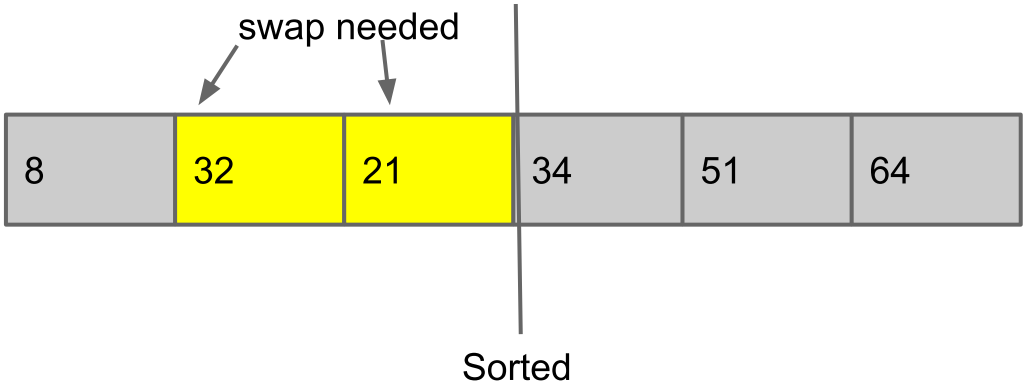
Bubble Sort



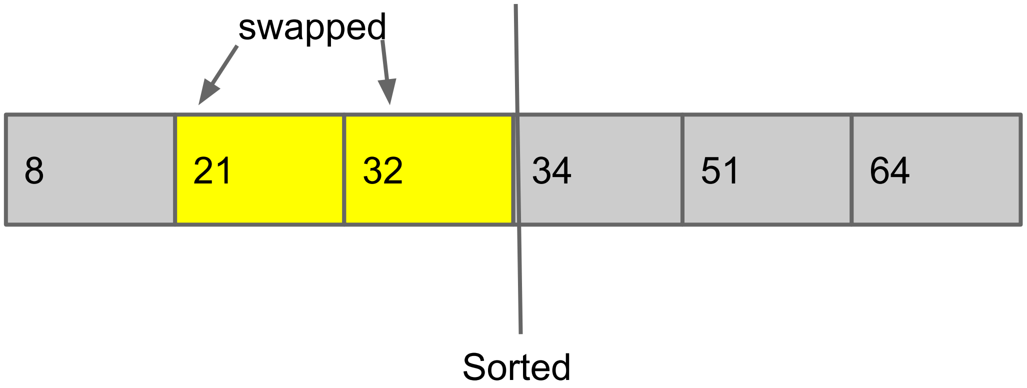
Bubble Sort



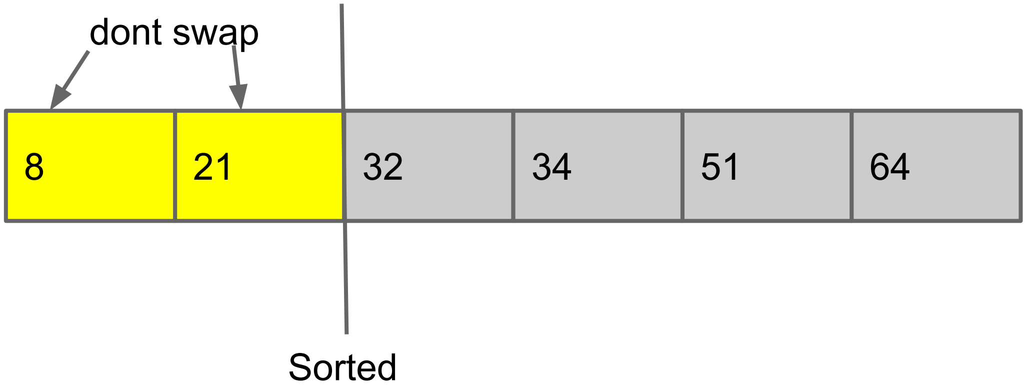
Bubble Sort



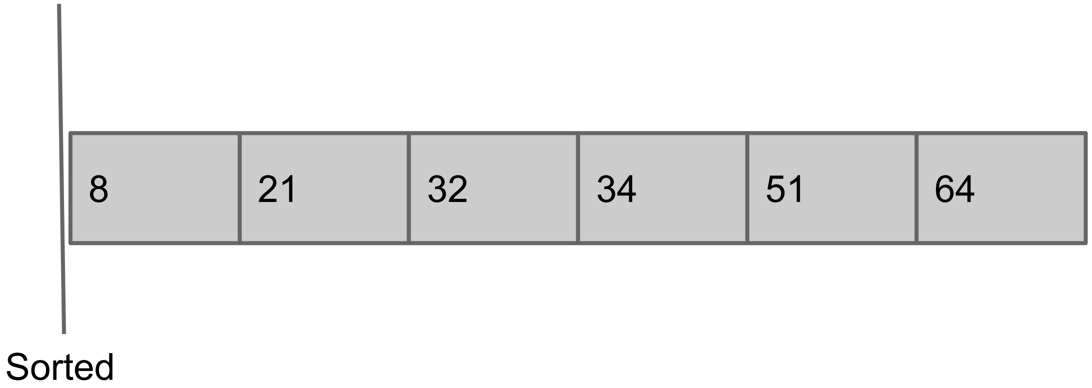
Bubble Sort



Bubble Sort



Bubble Sort



Bubble Sort

- What is the complexity

Bubble Sort

- What is the complexity
  - Best - N
  - Worst - N^2

Bubble Sort

- Is this a good algorithm for very large sets of data?

Bubble Sort

http://www.youtube.com/watch?v=lyZQPjUT5B4&feature=relmfu

Selection Sort

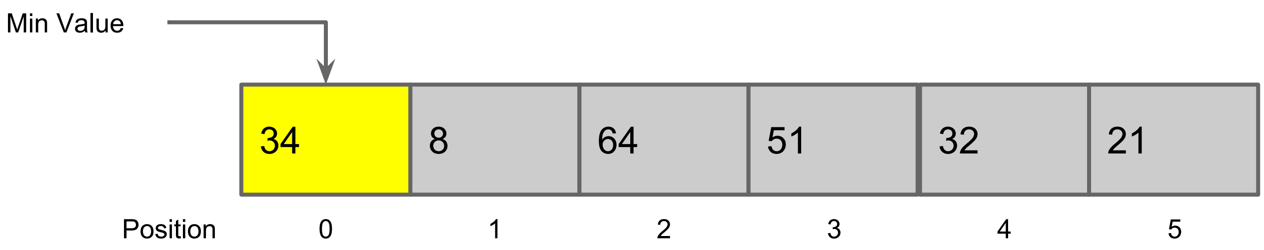
- The selection sort tries to minimize the number of swaps

Selection Sort

- For every position in the list, 0 through N-1 as ‘i’
  - loop over the list and find the smallest item
    - If item is the smallest, swap it into the current location (i) and continue outer loop

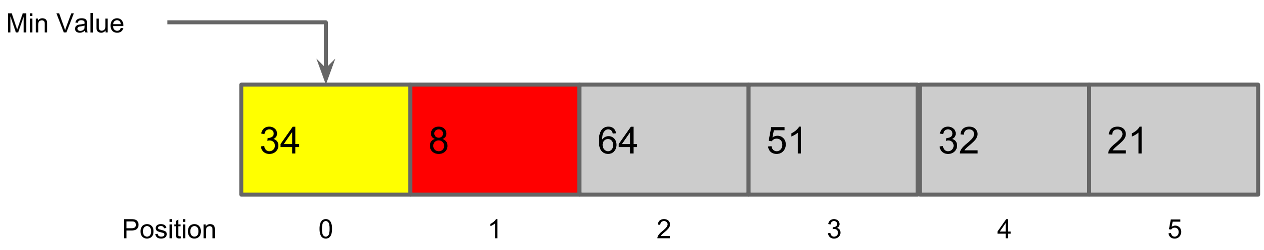
Selection Sort

Start at the beginning of the list. Start with i as index 0 and minValue as 34



Selection Sort

Loop over list to find the smallest item
and place it at position 0

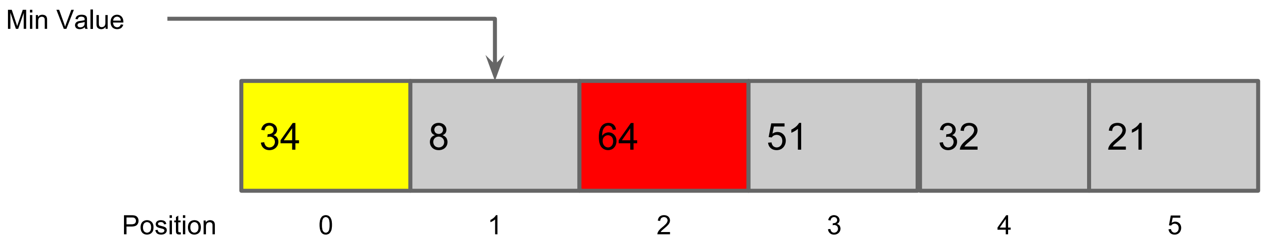


Is 34 < 8?

No, correct MinValue to 8

Selection Sort

Loop over list to find the smallest item
and place it at position 0

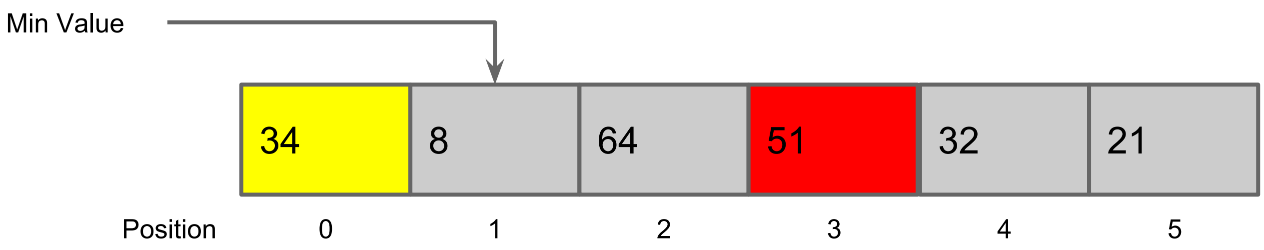


Is 8 < 64?

Yes, continue

Selection Sort

Loop over list to find the smallest item
and place it at position 0

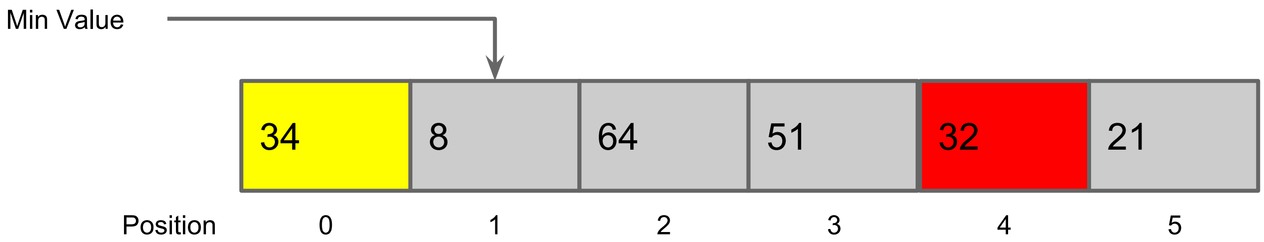


Is 8 < 51?

Yes, continue

Selection Sort

Loop over list to find the smallest item
and place it at position 0

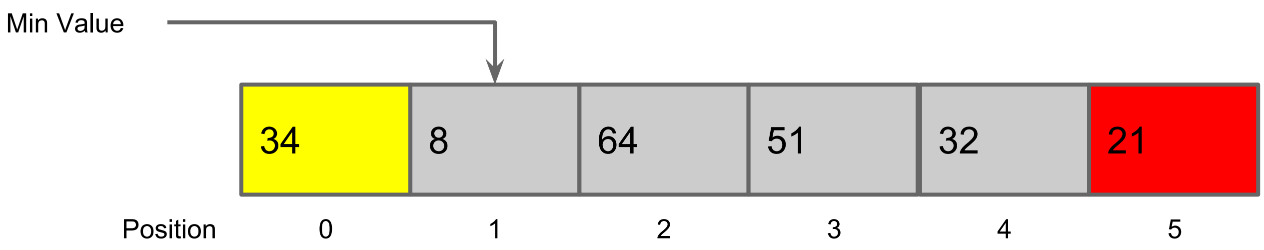


Is 8 < 32?

Yes, continue

Selection Sort

Loop over list to find the smallest item
and place it at position 0



Is 8 < 21?

Yes. Since we are at the end of the list, 8 must be the min value. Swap position 0 with position 1

Selection Sort

Loop over list to find the smallest item
and place it at position 0

Min Value

	8	34	64	51	32	21
Position	0	1	2	3	4	5

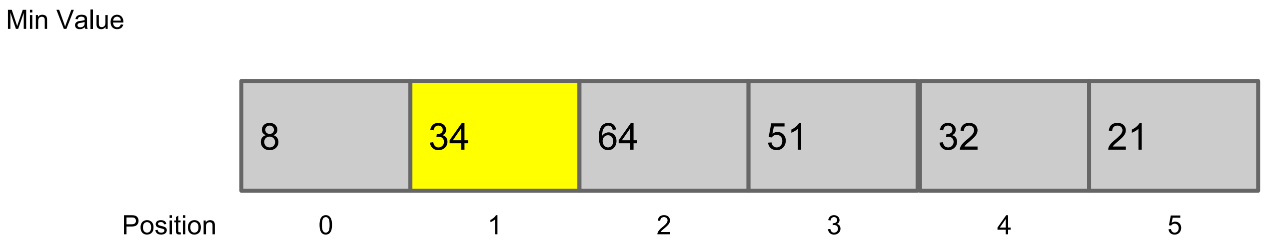
Selection Sort

- What is guaranteed after the first pass?

- What is guaranteed after any single pass?

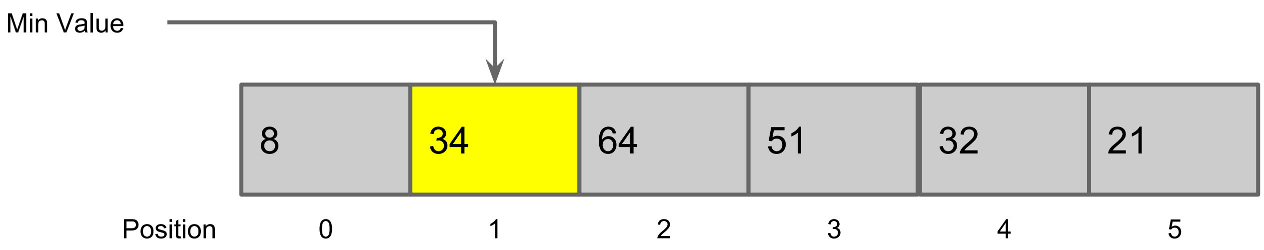
Selection Sort

Loop over list to find the next smallest
item and place it at position 1



Selection Sort

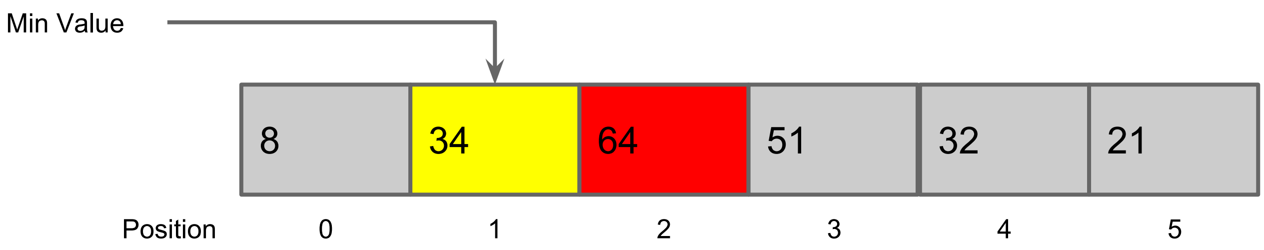
Loop over list to find the next smallest item and place it at position 1



Set Min Value to 34 and begin looping again

Selection Sort

Loop over list to find the next smallest item and place it at position 1

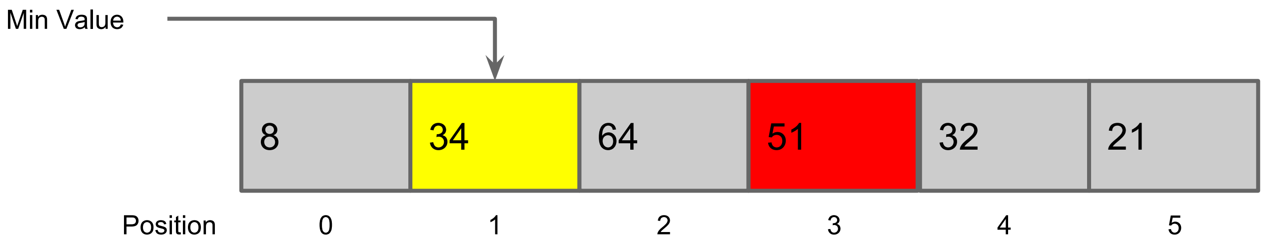


is 34 < 64?

Yes, continue

Selection Sort

Loop over list to find the next smallest item and place it at position 1

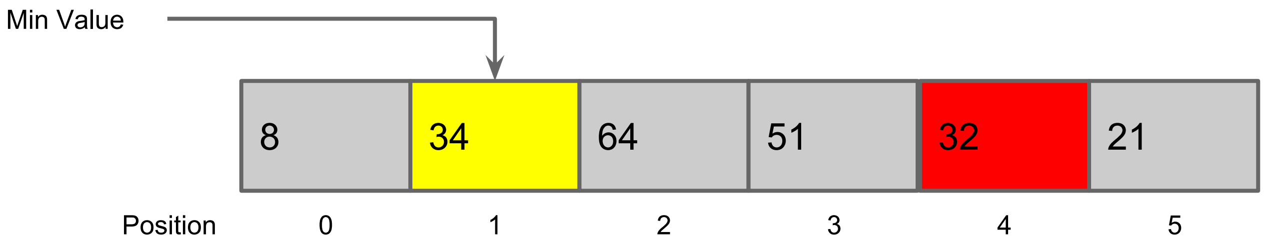


is 34 < 51?

Yes, continue

Selection Sort

Loop over list to find the next smallest item and place it at position 1

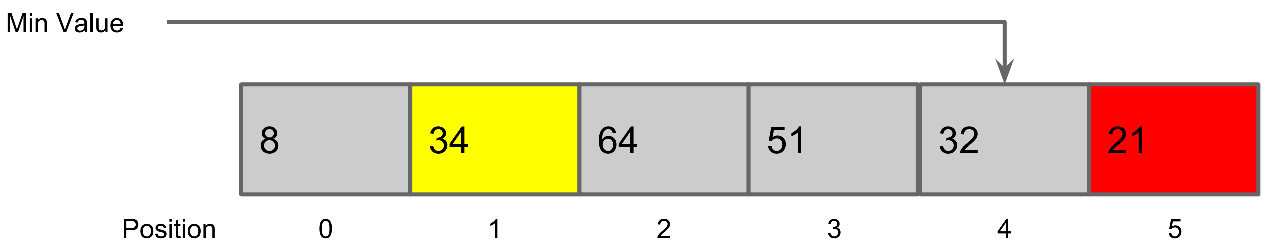


is 34 < 32?

No, set Min Value to 32

Selection Sort

Loop over list to find the next smallest
item and place it at position 1



is 32 < 21?
No, set Min Value to 21. There are no more elements to search, so swap 21 into position 1.

Selection Sort

Loop over list to find the next smallest item and place it at position 1

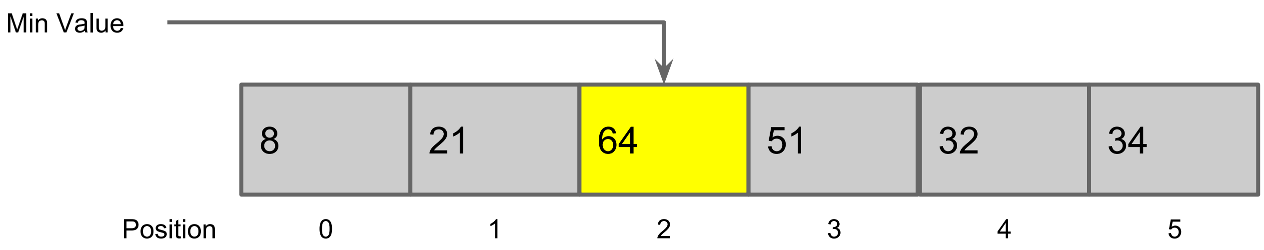
Min Value

Sorted

	8	21	64	51	32	34
Position	0	1	2	3	4	5

Selection Sort

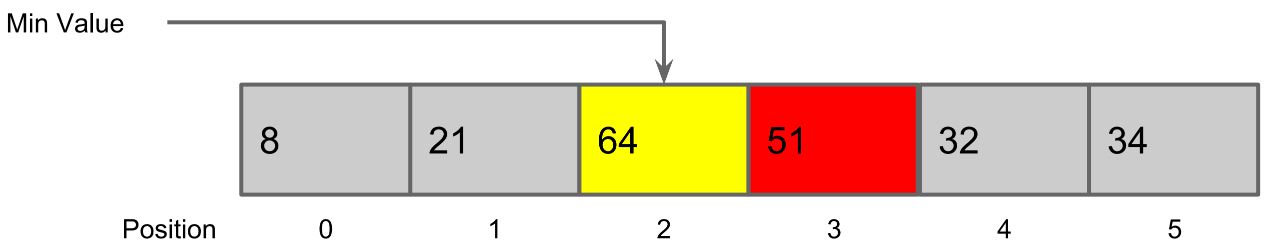
Loop over list to find the next smallest item and place it at position 2



Set Min Value to 64.

Selection Sort

Loop over list to find the next smallest item and place it at position 2



is 64 < 51?

No, set Min Value to 51.

Selection Sort

- How many swaps happen per main loop?

Selection Sort

- How many swaps happen per main loop?
  - 1
- How many comparisons are performed?
  - N^2

Analysis

Algorithm Analysis

- How can we compare one algorithm against another?

Algorithm Analysis

- How can we compare one algorithm against another?
    - Define the computational problem
    - Look at the requirements
        - Space
        - Time

Algorithm Analysis

Space-Time...

Algorithm Analysis

- Space
  - The amount of memory
- Time
  - Length of execution

Algorithm Analysis

- How could we analyse an algorithm?

Algorithm Analysis

- How could we analyse an algorithm?
  - Run the algorithm and time it

- Problems?

Algorithm Analysis

- How could we analyse an algorithm?
  - Run the algorithm and time it

- Problems?
  - Results depend on…
    - Hardware
    - Data sets
    - OS
    - ….

Algorithm Analysis

- Break the algorithms up into ‘steps’
  - A ‘step’ is a basic operation
    - Execution is time bounded by some constant regardless of input size
      - Swap vs find

- Define a generic size of the array or data set
  - Usually ‘N’

Algorithm Analysis

- Constants are ignored
  - One machine may execute the operation in one step, yet another takes multiple

Algorithm Analysis

- Measuring Complexity
  - Determine the number of steps required to run the algorithm

Algorithm Analysis

- Measuring Complexity
  - Determine the number of steps required to run the algorithm

Running Time Calculations: The Rules

1. FOR loops
   1. running time of the statements multiplied by the size of the loop
2. Nested loops
   1. running time of the statements multiplied by the product of the sizes of all loops
3. Consecutive Statements
   1. add them
4. If/Else
   1. never more than the running time of the test plus the larger of the running times

Example

```
int sum( int n)
{
    int partialSum;

    partialSum = 0;
    for (int i  = 1; i <= n; i++)
        partialSum += i * i * i;

    return partialSum;
}
```

Example

```
int sum( int n)
{
    int partialSum;                        //0

    partialSum = 0;                        //1
    for (int i  = 1; i <= n; i++)
        partialSum += i * i * i;

    return partialSum;                     //1
}
```

Example

```
int sum( int n)
{
    int partialSum;                    //0

    partialSum = 0;                    //1
    for (int i  = 1; i <= n; i++)
        partialSum += i * i * i;       //1 + 1 + 1 + 1 = 4

    return partialSum;                 //1
}
```

Example

```
int sum( int n)
{
    int partialSum;                       //0

    partialSum = 0;                       //1
    for (int i  = 1; i <= n; i++)        //1 (init i) +  N+1(compare) +
                                          //N (for i++) = 2N+2
        partialSum += i * i * i;         //1 + 1 + 1 + 1 = 4N

    return partialSum;                    //1
}
```

Example

```
int sum( int n)
{
    int partialSum;                    //0

    partialSum = 0;                    //1
    for (int i  = 1; i <= n; i++)      //1 (init i) +  N+1(compare) +
                                       //N (for i++) = 2N+2
        partialSum += i * i * i;       //1 + 1 + 1 + 1 = 4N

    return partialSum;                 //1
}
```

Total: 6N + 4

Algorithm Analysis

- Since inner sum is constant despite the size of the array, we can count it as a single step

Example

```
int sum( int n)
{
    int partialSum;                         //0

    partialSum = 0;                         //1
    for (int i  = 1; i <= n; i++)           //1 (init i) +  N+1(compare) +
                                            //N (for i++) = 2N+2
        partialSum += i * i * i;            //1

    return partialSum;                      //1
}
```

Total: 3N + 4 // a little simpler

Algorithm Analysis

- To compare algorithms, we must compare apples to apples. Thus, functions are usually found for:
  - Worse-Case Complexity
    - The number of steps performed on an input that requires the most steps
  - Average-Case Complexity
    - The number of steps performed on an input with an average number of steps (not the best indicator)
  - Best-Case Complexity
    - The number of steps performed on an input with the least number of steps (not the best indicator either)

Algorithm Analysis

- The worst case can be shown as a function of f(n)
  - f(n) = 6N + 4 // Previous algorithm

Algorithm Analysis

- The worst case can be shown as a function of f(n) or g(n) or ...
  - $f(n) = 6N + 4$ // Previous algorithm

- The worst case complexity of two algorithms can be compared by looking at the ration of f(n)/g(n) when n gets large

Algorithm Analysis

- Examples
  - $f(n) = n$
  - $g(n) = n^2$

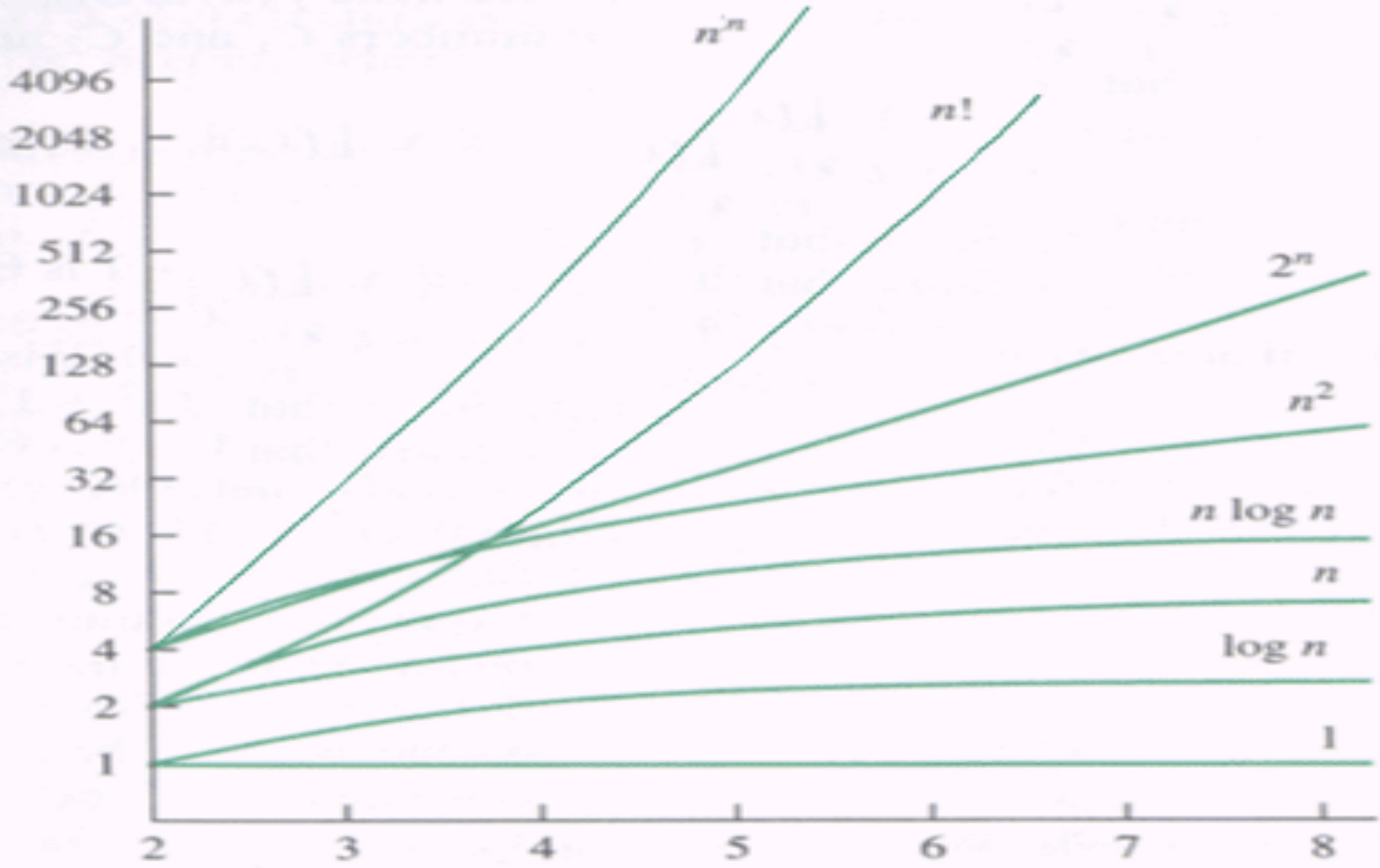
$$\frac{f(n)}{g(n)} = \frac{N}{N^2}$$

Algorithm Analysis

- Examples
  - $f(n) = n$
  - $g(n) = n^2$

$$\frac{f(n)}{g(n)} = \frac{N}{N^2} = \frac{1}{N}$$

- g(n) grows a lot faster than f(n)



4096
2048
1024
512
256
128
64
32
16
8
4
2
1
2
3
4
5
6
7
8
n^n
$n!$
2^n
n^2
$n \log n$
n
$\log n$
1

Algorithm Analysis

- What does this mean for us?
  - We really only care about the growth rate of algorithms
    - Welcome to Big O notation!

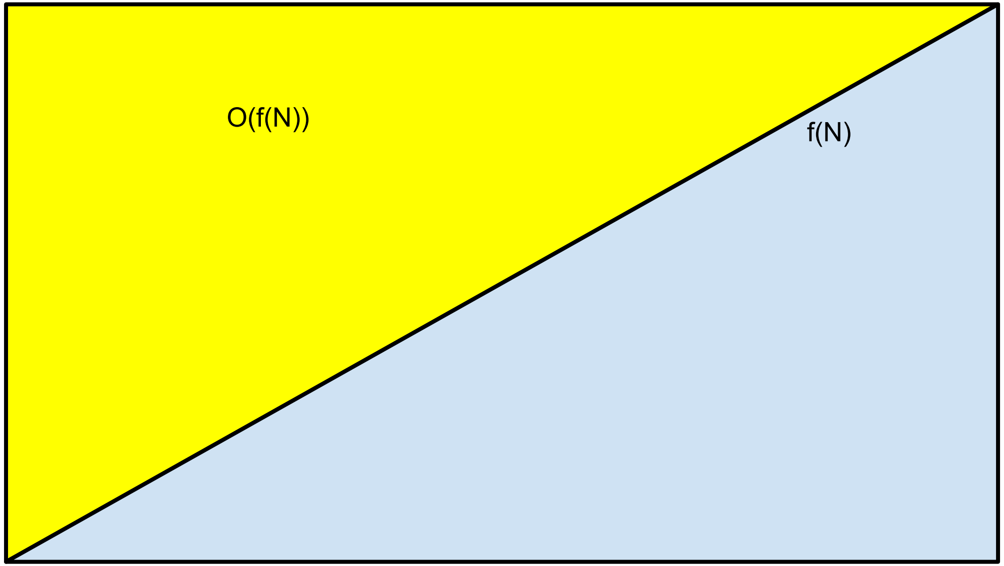
Algorithm Analysis: Big-Oh

- $T(N) = O(f(N))$
  - if there are positive constants c and n_0 such that
    $T(N) <= cf(N)$ when $N >= n_0$

- What does that mean?

Algorithm Analysis: Big-Oh

- $T(N) = O(f(N))$
  - if there are positive constants c and n_0 such that
    $T(N) <= cf(N)$ when $N >= n_0$



We only need to make sure that the true complexity function is above the Big O (With the biggest Big O possible)

Algorithm Analysis: Big-Oh

- $T(N) = O(f(N))$
  - if there are positive constants c and n_0 such that
    $T(N) <= cf(N)$ when $N >= n_0$

- The growth rate of T(N) is <= the growth of f(N)

- $T(N) = 1000N$
  - $f(N) = ?$
- $T(N) = 100N^2$
- $T(N) = 100N^2 + 3N + 300000000$

- Would it be correct to say that $f(N) = 2^N$ if $T(N) = 3N$?

Examples

```
void foo(int* paMyArray, int length)
{
    int sum;
    for (int i = 0; i < 7; i++)
    {
        sum += paMyArray[i];
    }
}
```

Examples

```
void foo(int* paMyArray, int length)
{
    int sum;
    for (int i = 0; i < 7; i++)
    {
        sum += paMyArray[i];
    }
}
```

Total: O(1)

Examples

```
void bar(int* paMyArray, int len)
{
    for (int i = 0; i < len; i++)
    {
        paMyArray[i] = 0;
    }

    for (int i = 0; i < len; i++)
    {
        for (int j = 0; j < len; j++)
        {
            paMyArray[i] = i + j;
        }
    }
}
```

Examples

```
void bar(int* paMyArray, int len)
{
    for (int i = 0; i < len; i++)
    {
        paMyArray[i] = 0;
    }

    for (int i = 0; i < len; i++)
    {
        for (int j = 0; j < len; j++)
        {
            paMyArray[i] = i + j;
        }
    }
}
```

Total: $O(N^2)$

Examples

```
int binarySearch(int * paArray, int len, const int x)
{
    int low = 0, high = len - 1;
    while (low <= high)
    {
        int mid = (low + high) / 2;
        if (paArray[mid] < x)
            low = mid + 1;
        else if (paArray[mid] > x)
            high = mid - 1;
        else
            return mid;
    }
    return -1;
}
```

Examples

```
int binarySearch(int * paArray, int len, const int x)
{
    int low = 0, high = len - 1;
    while (low <= high)
    {
        int mid = (low + high) / 2;
        if (paArray[mid] < x)
            low = mid + 1;
        else if (paArray[mid] > x)
            high = mid - 1;
        else
            return mid;
    }
    return -1;
}
```

Total: O(log N)