Algorithms and Analysis

Lesson 13: *Analyse!*

Pseudo code, binary search, insertion sort, selection sort, lower bound complexity

Outline

1. **Algorithm Analysis**
2. Search
3. Simple Sort
   - Insertion Sort
   - Selection Sort
4. Lower Bound

Algorithm Analysis

- We've covered most of the basic data structures
- The rest of the course is going to focus more on algorithms
- We will look predominantly at
  - Searching
  - Sorting
  - Graph Algorithms
- Emphasise general solution strategies

Algorithm Analysis

- We've covered most of the basic data structures
- The rest of the course is going to focus more on algorithms
- We will look predominantly at
  - Searching
  - Sorting
  - Graph Algorithms
- Emphasise general solution strategies

Algorithm Analysis

- We've covered most of the basic data structures
- The rest of the course is going to focus more on algorithms
- We will look predominantly at
  - ⋆ Searching
  - ⋆ Sorting
  - ⋆ Graph Algorithms
- Emphasise general solution strategies

Algorithm Analysis

- We've covered most of the basic data structures
- The rest of the course is going to focus more on algorithms
- We will look predominantly at
  - Searching
  - Sorting
  - Graph Algorithms
- Emphasise general solution strategies

Code and Pseudo Code

- C++ code is often difficult to read—there are often programming details we don't care abour
- It contains details such as throwing exception which are repetitive and often depends on who you are writing the code for
- Algorithms are not language dependent (data structures are a bit more language dependent)
- To focus on what is important we will use a stylised programming language called **pseudo code**

Code and Pseudo Code

- C++ code is often difficult to read—there are often programming details we don't care abour
- It contains details such as throwing exception which are repetitive and often depends on who you are writing the code for
- Algorithms are not language dependent (data structures are a bit more language dependent)
- To focus on what is important we will use a stylised programming language called **pseudo code**

Code and Pseudo Code

- C++ code is often difficult to read—there are often programming details we don't care abour
- It contains details such as throwing exception which are repetitive and often depends on who you are writing the code for
- Algorithms are not language dependent (data structures are a bit more language dependent)
- To focus on what is important we will use a stylised programming language called **pseudo code**

Code and Pseudo Code

- C++ code is often difficult to read—there are often programming details we don't care abour
- It contains details such as throwing exception which are repetitive and often depends on who you are writing the code for
- Algorithms are not language dependent (data structures are a bit more language dependent)
- To focus on what is important we will use a stylised programming language called **pseudo code**

Pseudo Code

- There is no standard for pseudo code
- The commands are not too dissimilar to C++
- The one strange convention is that assignments use an arrow $\leftarrow$
- Arrays are written in bold $\boldsymbol{a}$ with elements a_i
- In pseudo-code you are free to invent any operations that can be easily interpreted

Pseudo Code

- There is no standard for pseudo code
- The commands are not too dissimilar to C++
- The one strange convention is that assignments use an arrow $\leftarrow$
- Arrays are written in bold $\boldsymbol{a}$ with elements a_i
- In pseudo-code you are free to invent any operations that can be easily interpreted

Pseudo Code

- There is no standard for pseudo code
- The commands are not too dissimilar to C++
- The one strange convention is that assignments use an arrow $\leftarrow$
- Arrays are written in bold $\boldsymbol{a}$ with elements a_i
- In pseudo-code you are free to invent any operations that can be easily interpreted

Pseudo Code

- There is no standard for pseudo code
- The commands are not too dissimilar to C++
- The one strange convention is that assignments use an arrow $\leftarrow$
- Arrays are written in bold $\boldsymbol{a}$ with elements a_i
- In pseudo-code you are free to invent any operations that can be easily interpreted

Pseudo Code

- There is no standard for pseudo code
- The commands are not too dissimilar to C++
- The one strange convention is that assignments use an arrow $\leftarrow$
- Arrays are written in bold $\boldsymbol{a}$ with elements a_i
- In pseudo-code you are free to invent any operations that can be easily interpreted

Outline

1. Algorithm Analysis

2. **Search**

3. Simple Sort

   - Insertion Sort
   - Selection Sort

4. Lower Bound

Dumb Search

```
DUMBSEARCH(a, x)
{
  /* search array a = (a_1, ... a_n)  */
  /* for x return true */
  /* if  successful  else  false */
  for i←1 to n
    if (a_i = x)
      return true
    endif
  endfor

  return false
}
```

Dumb Search

```
DUMBSEARCH(a, x)
{
  /* search array a = (a_1, ... a_n)  */
  /* for x return true */
  /* if successful else false */
  for i←1 to n
    if (a_i = x)
      return true
    endif
  endfor

  return false
}
```

```
bool search(T a[], T x)
{
  for (int i=0; i<n; i++) {
    if (a[i] == x)
      return true;
  }

  return false;
}
```

Dumb Search

```
DumbSearch(a, x)
{
  /* search array a = (a_1, ... a_n)  */
  /* for x return true */
  /* if successful else false */
  for i←1 to n
    if (a_i = x)
      return true
    endif
  endfor

  return false
}
```

```
bool search(T a[], T x)
{
  for (int i=0; i<n; i++) {
    if (a[i] == x)
      return true;
  }

  return false;
}
```

56	26	62	60	53	53	77	91	60	41

Dumb Search

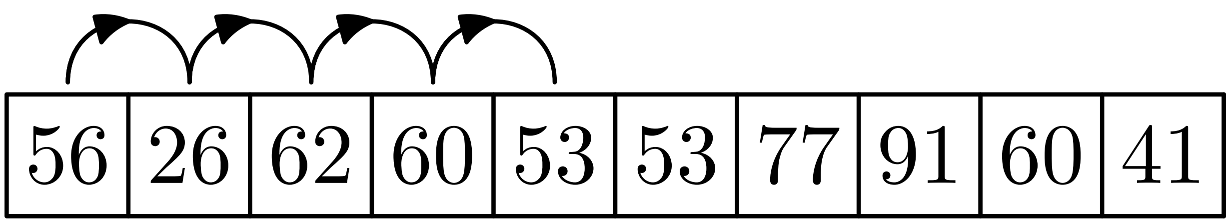
```
DumbSearch(a, x)
{
  /* search array a = (a_1, ... a_n)  */
  /* for x return true */
  /* if successful else false */
  for i←1 to n
    if (a_i = x)
      return true
    endif
  endfor

  return false
}
```

```
bool search(T a[], T x)
{
  for (int i=0; i<n; i++) {
    if (a[i] == x)
      return true;
  }

  return false;
}
```

find(53) ⟶ true



Dumb Search

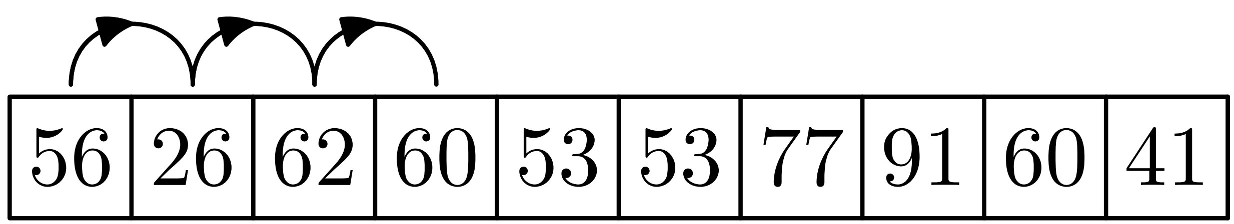
```
DUMBSEARCH(a, x)
{
  /* search array a = (a_1, ... a_n)  */
  /* for x return true */
  /* if successful else false */
  for i←1 to n
    if (a_i = x)
      return true
    endif
  endfor

  return false
}
```

```
bool search(T a[], T x)
{
  for (int i=0; i<n; i++) {
    if (a[i] == x)
      return true;
  }

  return false;
}
```

$$\text{find}(60) \longrightarrow \text{true}$$



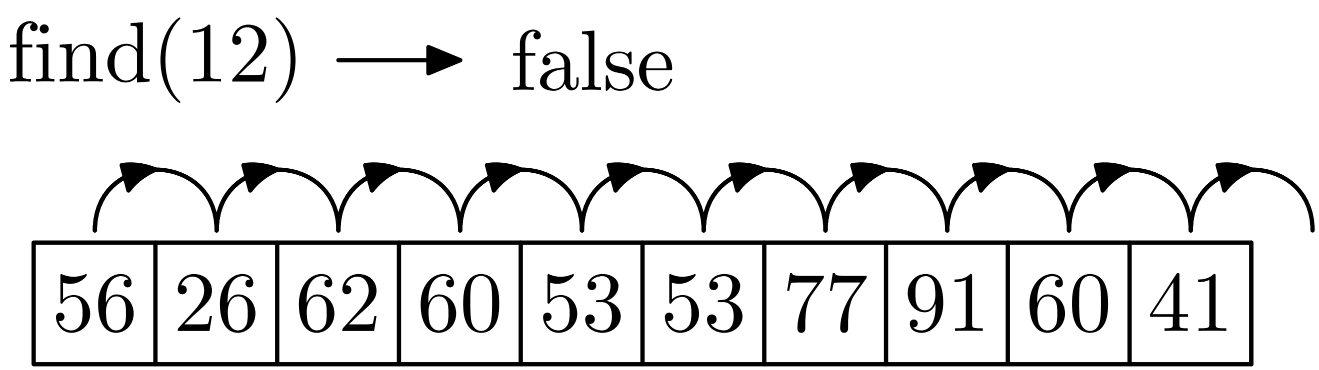
Dumb Search

```
DUMBSEARCH(a, x)
{
  /* search array a = (a_1, ... a_n)  */
  /* for x return true */
  /* if successful else false */
  for i←1 to n
    if (a_i = x)
      return true
    endif
  endfor

  return false
}
```

```
bool search(T a[], T x)
{
  for (int i=0; i<n; i++) {
    if (a[i] == x)
      return true;
  }

  return false;
}
```



Time Complexity

- Worst case:
  - ⋆ The worst case for a successful search is when the element is in the last location in the array
  - ⋆ This takes n comparisons: worst case is $\Theta(n)$
- Best case:
  - ⋆ The best case is when the element is in the first location
  - ⋆ This takes 1 comparison: best case is $\Theta(1)$
- Average case:
  - ⋆ Assume every location is equally likely to hold the key

$$\frac{1+2+\ldots+n}{n} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

- For an unsuccessful search n comparison are necessary

Time Complexity

- Worst case:
  - ⋆ The worst case for a successful search is when the element is in the last location in the array
  - ⋆ This takes n comparisons: worst case is $\Theta(n)$
- Best case:
  - ⋆ The best case is when the element is in the first location
  - ⋆ This takes 1 comparison: best case is $\Theta(1)$
- Average case:
  - ⋆ Assume every location is equally likely to hold the key

$$\frac{1+2+\ldots+n}{n} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

- For an unsuccessful search n comparison are necessary

Time Complexity

- Worst case:
  - ⋆ The worst case for a successful search is when the element is in the last location in the array
  - ⋆ This takes n comparisons: worst case is $\Theta(n)$
- Best case:
  - ⋆ The best case is when the element is in the first location
  - ⋆ This takes 1 comparison: best case is $\Theta(1)$
- Average case:
  - ⋆ Assume every location is equally likely to hold the key

$$\frac{1+2+\ldots+n}{n} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

- For an unsuccessful search n comparison are necessary

Time Complexity

- Worst case:
  - ⋆ The worst case for a successful search is when the element is in the last location in the array
  - ⋆ This takes n comparisons: worst case is $\Theta(n)$
- Best case:
  - ⋆ The best case is when the element is in the first location
  - ⋆ This takes 1 comparison: best case is $\Theta(1)$
- Average case:
  - ⋆ Assume every location is equally likely to hold the key

$$\frac{1+2+\ldots+n}{n} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

- For an unsuccessful search n comparison are necessary

Time Complexity

- Worst case:
  - ⋆ The worst case for a successful search is when the element is in the last location in the array
  - ⋆ This takes n comparisons: worst case is $\Theta(n)$
- Best case:
  - ⋆ The best case is when the element is in the first location
  - ⋆ This takes 1 comparison: best case is $\Theta(1)$
- Average case:
  - ⋆ Assume every location is equally likely to hold the key

$$\frac{1+2+\ldots+n}{n} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

- For an unsuccessful search n comparison are necessary

Time Complexity

- Worst case:
  - ⋆ The worst case for a successful search is when the element is in the last location in the array
  - ⋆ This takes n comparisons: worst case is $\Theta(n)$
- Best case:
  - ⋆ The best case is when the element is in the first location
  - ⋆ This takes 1 comparison: best case is $\Theta(1)$
- Average case:
  - ⋆ Assume every location is equally likely to hold the key

$$\frac{1+2+\ldots+n}{n} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

- For an unsuccessful search n comparison are necessary

Time Complexity

- Worst case:
  - ⋆ The worst case for a successful search is when the element is in the last location in the array
  - ⋆ This takes n comparisons: worst case is $\Theta(n)$
- Best case:
  - ⋆ The best case is when the element is in the first location
  - ⋆ This takes 1 comparison: best case is $\Theta(1)$
- Average case:
  - ⋆ Assume every location is equally likely to hold the key

$$\frac{1+2+\ldots+n}{n} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

- For an unsuccessful search n comparison are necessary

Time Complexity

- Worst case:
  - ⋆ The worst case for a successful search is when the element is in the last location in the array
  - ⋆ This takes n comparisons: worst case is $\Theta(n)$
- Best case:
  - ⋆ The best case is when the element is in the first location
  - ⋆ This takes 1 comparison: best case is $\Theta(1)$
- Average case:
  - ⋆ Assume every location is equally likely to hold the key

$$\frac{1+2+\ldots+n}{n} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

- For an unsuccessful search n comparison are necessary

Time Complexity

- Worst case:
  - ⋆ The worst case for a successful search is when the element is in the last location in the array
  - ⋆ This takes n comparisons: worst case is $\Theta(n)$
- Best case:
  - ⋆ The best case is when the element is in the first location
  - ⋆ This takes 1 comparison: best case is $\Theta(1)$
- Average case:
  - ⋆ Assume every location is equally likely to hold the key

$$\frac{1+2+\ldots+n}{n} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

- For an unsuccessful search n comparison are necessary

Time Complexity

- Worst case:
  - ⋆ The worst case for a successful search is when the element is in the last location in the array
  - ⋆ This takes n comparisons: worst case is $\Theta(n)$
- Best case:
  - ⋆ The best case is when the element is in the first location
  - ⋆ This takes 1 comparison: best case is $\Theta(1)$
- Average case:
  - ⋆ Assume every location is equally likely to hold the key

$$\frac{1+2+\ldots+n}{n} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

- For an unsuccessful search n comparison are necessary

Time Complexity

- Worst case:
  - ⋆ The worst case for a successful search is when the element is in the last location in the array
  - ⋆ This takes n comparisons: worst case is $\Theta(n)$
- Best case:
  - ⋆ The best case is when the element is in the first location
  - ⋆ This takes 1 comparison: best case is $\Theta(1)$
- Average case:
  - ⋆ Assume every location is equally likely to hold the key

$$\frac{1+2+\ldots+n}{n} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

- For an unsuccessful search n comparison are necessary

Time Complexity

- Worst case:
  - ⋆ The worst case for a successful search is when the element is in the last location in the array
  - ⋆ This takes n comparisons: worst case is $\Theta(n)$
- Best case:
  - ⋆ The best case is when the element is in the first location
  - ⋆ This takes 1 comparison: best case is $\Theta(1)$
- Average case:
  - ⋆ Assume every location is equally likely to hold the key

$$\frac{1+2+\ldots+n}{n} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

- For an unsuccessful search n comparison are necessary

Binary Search

- If the array is ordered we can do better
- At each step we bisect the array

```
BinarySearch(a, x)
{
  low ←1
  high ←n
  while (low ≤ high)
    mid ←⌊(low + high)/2⌋
    if x > a_mid
      low ←mid + 1
    elseif x < a_mid
      high ←mid −1
    else
      return true
    endif
  endwhile
  return false
}
```

⋆ Based on a **divide-and-conquer** strategy

⋆ We check the middle of the array

$$\underbrace{a_1, a_2, \cdots, a_{m-1},}_{x<a_m} \overbrace{a_m}^{x=a_m}, \underbrace{a_{m+1}, \cdots a_n}_{x>a_m}$$

⋆ Based on a recursive idea

Binary Search

- If the array is ordered we can do better
- At each step we bisect the array

```
BinarySearch(a, x)
{
  low ←1
  high ←n
  while (low ≤ high)
    mid ←⌊(low + high)/2⌋
    if x > a_mid
      low ←mid + 1
    elseif x < a_mid
      high ←mid -1
    else
      return true
    endif
  endwhile
  return false
}
```

⋆ Based on a **divide-and-conquer** strategy

⋆ We check the middle of the array

$$\underbrace{a_1, a_2, \cdots, a_{m-1},}_{x<a_m} \overbrace{a_m}^{x=a_m}, \underbrace{a_{m+1}, \cdots a_n}_{x>a_m}$$

⋆ Based on a recursive idea

Binary Search

- If the array is ordered we can do better
- At each step we bisect the array

```
BinarySearch(a, x)
{
  low ←1
  high ←n
  while (low ≤ high)
    mid ←⌊(low + high)/2⌋
    if x > a_mid
      low ←mid + 1
    elseif x < a_mid
      high ←mid −1
    else
      return true
    endif
  endwhile
  return false
}
```

⋆ Based on a **divide-and-conquer** strategy

⋆ We check the middle of the array

$$\underbrace{a_1, a_2, \cdots, a_{m-1}}_{x<a_m}, \overbrace{a_m}^{x=a_m}, \underbrace{a_{m+1}, \cdots a_n}_{x>a_m}$$

⋆ Based on a recursive idea

Binary Search

- If the array is ordered we can do better
- At each step we bisect the array

```
BinarySearch(a, x)
{
  low ←1
  high ←n
  while (low ≤ high)
    mid ←⌊(low + high)/2⌋
    if x > a_mid
      low ←mid + 1
    elseif x < a_mid
      high ←mid -1
    else
      return true
    endif
  endwhile
  return false
}
```

⋆ Based on a **divide-and-conquer** strategy

⋆ We check the middle of the array

$$\underbrace{a_1, a_2, \cdots, a_{m-1}}_{x<a_m}, \overbrace{a_m}^{x=a_m}, \underbrace{a_{m+1}, \cdots a_n}_{x>a_m}$$

⋆ Based on a recursive idea

Binary Search

- If the array is ordered we can do better
- At each step we bisect the array

```
BinarySearch(a, x)
{
  low ←1
  high ←n
  while (low ≤ high)
    mid ←⌊(low + high)/2⌋
    if x > a_mid
      low ←mid + 1
    elseif x < a_mid
      high ←mid -1
    else
      return true
    endif
  endwhile
  return false
}
```

⋆ Based on a **divide-and-conquer** strategy

⋆ We check the middle of the array

$$\underbrace{a_1, a_2, \cdots, a_{m-1}}_{x<a_m}, \overbrace{a_m}^{x=a_m}, \underbrace{a_{m+1}, \cdots a_n}_{x>a_m}$$

⋆ Based on a recursive idea

Binary Search

- If the array is ordered we can do better
- At each step we bisect the array

```
BinarySearch(a, x)
{
  low ←1
  high ←n
  while (low ≤ high)
    mid ←⌊(low + high)/2⌋
    if x > a_mid
      low ←mid + 1
    elseif x < a_mid
      high ←mid -1
    else
      return true
    endif
  endwhile
  return false
}
```

⋆ Based on a **divide-and-conquer** strategy

⋆ We check the middle of the array

$$\underbrace{a_1, a_2, \cdots, a_{m-1},}_{x<a_m} \overbrace{a_m}^{x=a_m}, \underbrace{a_{m+1}, \cdots a_n}_{x>a_m}$$

⋆ Based on a recursive idea

Binary Search

- If the array is ordered we can do better
- At each step we bisect the array

```
BINARYSEARCH(a, x)
{
  low ←1
  high ←n
  while (low ≤ high)
    mid ←⌊(low + high)/2⌋
    if x > a_mid
      low ←mid + 1
    elseif x < a_mid
      high ←mid −1
    else
      return true
    endif
  endwhile
  return false
}
```

⋆ Based on a **divide-and-conquer** strategy

⋆ We check the middle of the array

$$\underbrace{a_1, a_2, \cdots, a_{m-1}}_{x<a_m}, \overbrace{a_m}^{x=a_m}, \underbrace{a_{m+1}, \cdots a_n}_{x>a_m}$$

⋆ Based on a recursive idea

Binary Search

- If the array is ordered we can do better
- At each step we bisect the array

```
BinarySearch(a, x)
{
  low ←1
  high ←n
  while (low ≤ high)
    mid ←⌊(low + high)/2⌋
    if x > a_mid
      low ←mid + 1
    elseif x < a_mid
      high ←mid −1
    else
      return true
    endif
  endwhile
  return false
}
```

⋆ Based on a **divide-and-conquer** strategy

⋆ We check the middle of the array

$$\underbrace{a_1, a_2, \cdots, a_{m-1}}_{x<a_m}, \overbrace{a_m}^{x=a_m}, \underbrace{a_{m+1}, \cdots a_n}_{x>a_m}$$

⋆ Based on a recursive idea

Binary Search

- If the array is ordered we can do better
- At each step we bisect the array

```
BinarySearch(a, x)
{
  low ←1
  high ←n
  while (low ≤ high)
    mid ←⌊(low + high)/2⌋
    if x > a_mid
      low ←mid + 1
    elseif x < a_mid
      high ←mid -1
    else
      return true
    endif
  endwhile
  return false
}
```

⋆ Based on a **divide-and-conquer** strategy

⋆ We check the middle of the array

$$\underbrace{a_1, a_2, \cdots, a_{m-1}}_{x<a_m}, \overbrace{a_m}^{x=a_m}, \underbrace{a_{m+1}, \cdots a_n}_{x>a_m}$$

⋆ Based on a recursive idea

Binary Search

- If the array is ordered we can do better
- At each step we bisect the array

```
BinarySearch(a, x)
{
  low ←1
  high ←n
  while (low ≤ high)
    mid ←⌊(low + high)/2⌋
    if x > a_mid
      low ←mid + 1
    elseif x < a_mid
      high ←mid -1
    else
      return true
    endif
  endwhile
  return false
}
```

⋆ Based on a **divide-and-conquer** strategy

⋆ We check the middle of the array

$$\underbrace{a_1, a_2, \cdots, a_{m-1}}_{x<a_m}, \overbrace{a_m}^{x=a_m}, \underbrace{a_{m+1}, \cdots a_n}_{x>a_m}$$

⋆ Based on a recursive idea

Binary Search in Action

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95

Binary Search in Action

$\textsc{BinarySearch}(\mathbf{a}, 27)$

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
low																			high

Binary Search in Action

$\textsc{BinarySearch}(\mathbf{a}, 27)$

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
low									mid										high

Binary Search in Action

BinarySearch($\mathbf{a}$, 27)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
low								high											

Binary Search in Action

BinarySearch(**a**, 27)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
low				mid				high											

Binary Search in Action

BinarySearch($\mathbf{a}$, 27)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
low			high																

Binary Search in Action

BinarySearch($\mathbf{a}$, 27)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
low	mid		high																

Binary Search in Action

BinarySearch(**a**, 27)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
		low	high																

Binary Search in Action

BinarySearch($\mathbf{a}$, 27)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
		low mid	high																

Binary Search in Action

BinarySearch($\mathbf{a}$, 27)　　　found

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
		low mid	high																

Binary Search in Action

BinarySearch(**a**, 20)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
low																			high

Binary Search in Action

BinarySearch($\mathbf{a}$, 20)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
low									mid										high

Binary Search in Action

BinarySearch($\mathbf{a}$, 20)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
low								high											

Binary Search in Action

$\textsc{BinarySearch}(\mathbf{a},\, 20)$

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
low				mid				high											

Binary Search in Action

BinarySearch($\mathbf{a}$, 20)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
low			high																

Binary Search in Action

BinarySearch(**a**, 20)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
low	mid		high																

Binary Search in Action

BinarySearch($\mathbf{a}$, 20)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
		low	high																

Binary Search in Action

BinarySearch($\mathbf{a}$, 20)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
		low mid	high																

Binary Search in Action

BinarySearch($\mathbf{a}$, 20)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
	high	low																	

Binary Search in Action

$\textsc{BinarySearch}(\mathbf{a}, 20)$ not found

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
	high	low																	

Binary Search in Action

BinarySearch($\mathbf{a}$, 84)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95

low high

Binary Search in Action

$\textsc{BinarySearch}(\mathbf{a}, 84)$

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
low									mid										high

Binary Search in Action

BinarySearch($\mathbf{a}$, 84)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
										low									high

Binary Search in Action

BinarySearch($\mathbf{a}$, 84)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
										low				mid					high

Binary Search in Action

BinarySearch($\mathbf{a}$, 84)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
															low				high

Binary Search in Action

BinarySearch($\mathbf{a}$, 84)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
															low		mid		high

Binary Search in Action

BinarySearch($\mathbf{a}$, 84)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
															low	high			

Binary Search in Action

BinarySearch($\mathbf{a}$, 84)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
															low mid	high			

Binary Search in Action

BinarySearch($\mathbf{a}$, 84)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95

low high

Binary Search in Action

BinarySearch($\mathbf{a}$, 84)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
																low mid high			

Binary Search in Action

BinarySearch($\mathbf{a}$, 84) found

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
																low mid high			

Binary Search in Action

BinarySearch($\mathbf{a}$, 99)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
low																			high

Binary Search in Action

BinarySearch($\mathbf{a}$, 99)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
low									mid										high

Binary Search in Action

BinarySearch($\mathbf{a}$, 99)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
										low									high

Binary Search in Action

BINARYSEARCH($\mathbf{a}$, 99)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
										low				mid					high

Binary Search in Action

BinarySearch($\mathbf{a}$, 99)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
															low				high

Binary Search in Action

BinarySearch($\mathbf{a}$, 99)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
															low		mid		high

Binary Search in Action

BinarySearch($\mathbf{a}$, 99)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
																		low	high

Binary Search in Action

BinarySearch($\mathbf{a}$, 99)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
																		low mid	high

Binary Search in Action

BINARYSEARCH($\mathbf{a}$, 99)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
																			high low

Binary Search in Action

BinarySearch($\mathbf{a}$, 99)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
																			low mid high

Binary Search in Action

BinarySearch($\mathbf{a}$, 99)

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95

high low

Binary Search in Action

BinarySearch($\mathbf{a}$, 99) not found

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95

high low

Analysis

- We count the number of comparisons (counting each `if/else if` statement as a single comparison)

- Let $C(n)$ be the number of comparisons needed to search in an array of size n

- After one comparison we are left (in the worst case) with having to search an array not larger than $\lfloor n/2 \rfloor$, thus

$$C(n) < C(\lfloor n/2 \rfloor) + 1$$

- We've seen this relation before (lesson on Recursion)

- Easy to show $C(n) < \lfloor \log_2(n) \rfloor + 1 = O(\log(n))$

Analysis

- We count the number of comparisons (counting each `if/else if` statement as a single comparison)

- Let $C(n)$ be the number of comparisons needed to search in an array of size n

- After one comparison we are left (in the worst case) with having to search an array not larger than $\lfloor n/2 \rfloor$, thus

$$C(n) < C(\lfloor n/2 \rfloor) + 1$$

- We've seen this relation before (lesson on Recursion)

- Easy to show $C(n) < \lfloor \log_2(n) \rfloor + 1 = O(\log(n))$

Analysis

- We count the number of comparisons (counting each `if/else if` statement as a single comparison)

- Let $C(n)$ be the number of comparisons needed to search in an array of size n

- After one comparison we are left (in the worst case) with having to search an array not larger than $\lfloor n/2 \rfloor$, thus

$$C(n) < C(\lfloor n/2 \rfloor) + 1$$

- We've seen this relation before (lesson on Recursion)

- Easy to show $C(n) < \lfloor \log_2(n) \rfloor + 1 = O(\log(n))$

Analysis

- We count the number of comparisons (counting each `if/else if` statement as a single comparison)
- Let $C(n)$ be the number of comparisons needed to search in an array of size n
- After one comparison we are left (in the worst case) with having to search an array not larger than $\lfloor n/2 \rfloor$, thus

$$C(n) < C(\lfloor n/2 \rfloor) + 1$$

- We've seen this relation before (lesson on Recursion)
- Easy to show $C(n) < \lfloor \log_2(n) \rfloor + 1 = O(\log(n))$

Analysis

- We count the number of comparisons (counting each `if/else if` statement as a single comparison)
- Let $C(n)$ be the number of comparisons needed to search in an array of size n
- After one comparison we are left (in the worst case) with having to search an array not larger than $\lfloor n/2 \rfloor$, thus

$$C(n) < C(\lfloor n/2 \rfloor) + 1$$

- We've seen this relation before (lesson on Recursion)
- Easy to show $C(n) < \lfloor \log_2(n) \rfloor + 1 = O(\log(n))$

Analysis

- We count the number of comparisons (counting each `if/else if` statement as a single comparison)
- Let $C(n)$ be the number of comparisons needed to search in an array of size n
- After one comparison we are left (in the worst case) with having to search an array not larger than $\lfloor n/2 \rfloor$, thus

$$C(n) < C(\lfloor n/2 \rfloor) + 1$$

- We've seen this relation before (lesson on Recursion)
- Easy to show $C(n) < \lfloor \log_2(n) \rfloor + 1 = O(\log(n))$

Outline

1. Algorithm Analysis
2. Search
3. **Simple Sort**
   - Insertion Sort
   - Selection Sort
4. Lower Bound

Sort Characteristics

- Sort is one of the best studied algorithms. We care about stability, space and time complexity
- A sort algorithm is said to be **stable** if it does not change the order of elements that have the same value
- Space Complexity. Sort is said to be
  - ⋆ **In-place** if the memory used is $O(1)$
- Time Complexity. In particular we are interested in
  - ⋆ Worst case
  - ⋆ Average case
  - ⋆ Best case

Sort Characteristics

- Sort is one of the best studied algorithms. We care about stability, space and time complexity
- A sort algorithm is said to be **stable** if it does not change the order of elements that have the same value
- Space Complexity. Sort is said to be
  - ⋆ **In-place** if the memory used is $O(1)$
- Time Complexity. In particular we are interested in
  - ⋆ Worst case
  - ⋆ Average case
  - ⋆ Best case

Sort Characteristics

- Sort is one of the best studied algorithms. We care about stability, space and time complexity
- A sort algorithm is said to be **stable** if it does not change the order of elements that have the same value
- Space Complexity. Sort is said to be
  - ⋆ **In-place** if the memory used is $O(1)$
- Time Complexity. In particular we are interested in
  - ⋆ Worst case
  - ⋆ Average case
  - ⋆ Best case

Sort Characteristics

- Sort is one of the best studied algorithms. We care about stability, space and time complexity
- A sort algorithm is said to be **stable** if it does not change the order of elements that have the same value
- Space Complexity. Sort is said to be
  - ⋆ **In-place** if the memory used is $O(1)$
- Time Complexity. In particular we are interested in
  - ⋆ Worst case
  - ⋆ Average case
  - ⋆ Best case

Sort Characteristics

- Sort is one of the best studied algorithms. We care about stability, space and time complexity
- A sort algorithm is said to be **stable** if it does not change the order of elements that have the same value
- Space Complexity. Sort is said to be
  - **In-place** if the memory used is $O(1)$
- Time Complexity. In particular we are interested in
  - Worst case
  - Average case
  - Best case

Insertion Sort

- In insertion sort we keep a subsequence of elements on the left in sorted order
- This subsequence is increased by *inserting* the next element into its correct position

```
InsertionSort(a)
{
 for i←2 to n
   v←a_i
   j←i-1
   while j ≥ 1 and a_j>v
     a_{j+1}←a_j
     j←j-1
   endwhile
   a_{j+1}←v
 endfor
}
```

Insertion Sort

- In insertion sort we keep a subsequence of elements on the left in sorted order

- This subsequence is increased by *inserting* the next element into its correct position

```
InsertionSort(a)
{
  for i←2 to n
    v←a_i
    j←i-1
    while j ≥ 1 and a_j>v
      a_{j+1}←a_j
      j←j-1
    endwhile
    a_{j+1}←v
  endfor
}
```

Insertion Sort

- In insertion sort we keep a subsequence of elements on the left in sorted order

- This subsequence is increased by *inserting* the next element into its correct position

```
InsertionSort(a)
{
  for i←2 to n
    v←a_i
    j←i-1
    while j ≥ 1 and a_j>v
      a_{j+1}←a_j
      j←j-1
    endwhile
    a_{j+1}←v
  endfor
}
```

Insertion Sort

- In insertion sort we keep a subsequence of elements on the left in sorted order
- This subsequence is increased by *inserting* the next element into its correct position

```
INSERTIONSORT(a)
{
  for i←2 to n
    v←a_i
    j←i-1
    while j ≥ 1 and a_j>v
      a_{j+1}←a_j
      j←j-1
    endwhile
    a_{j+1}←v
  endfor
}
```

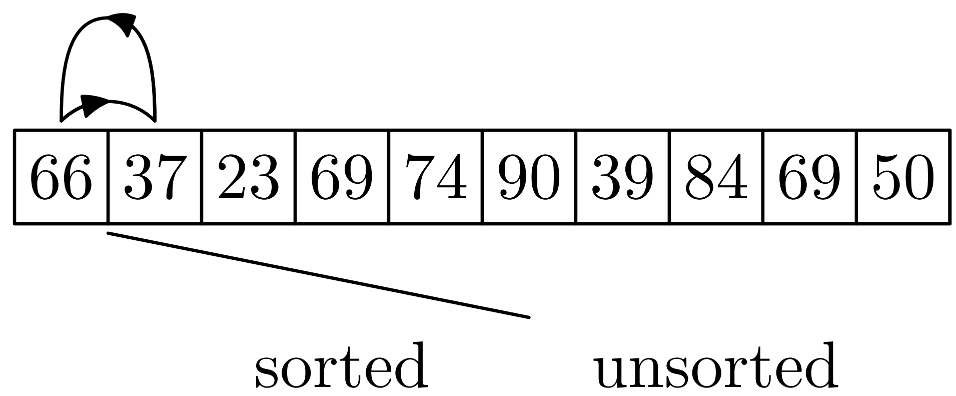
66	37	23	69	74	90	39	84	69	50

sorted unsorted

Insertion Sort

- In insertion sort we keep a subsequence of elements on the left in sorted order
- This subsequence is increased by *inserting* the next element into its correct position

```
InsertionSort(a)
{
  for i←2 to n
    v←a_i
    j←i-1
    while j ≥ 1 and a_j>v
      a_{j+1}←a_j
      j←j-1
    endwhile
    a_{j+1}←v
  endfor
}
```

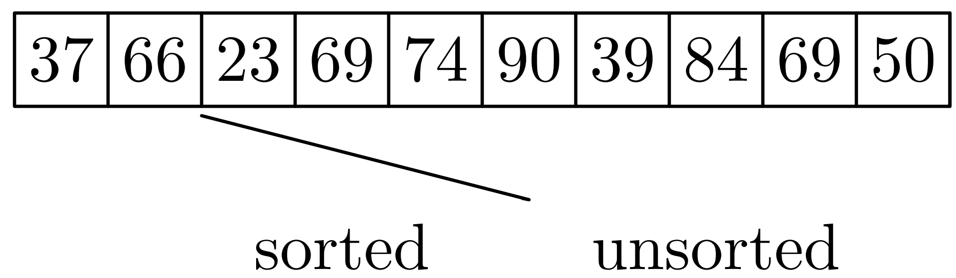


Insertion Sort

- In insertion sort we keep a subsequence of elements on the left in sorted order

- This subsequence is increased by *inserting* the next element into its correct position

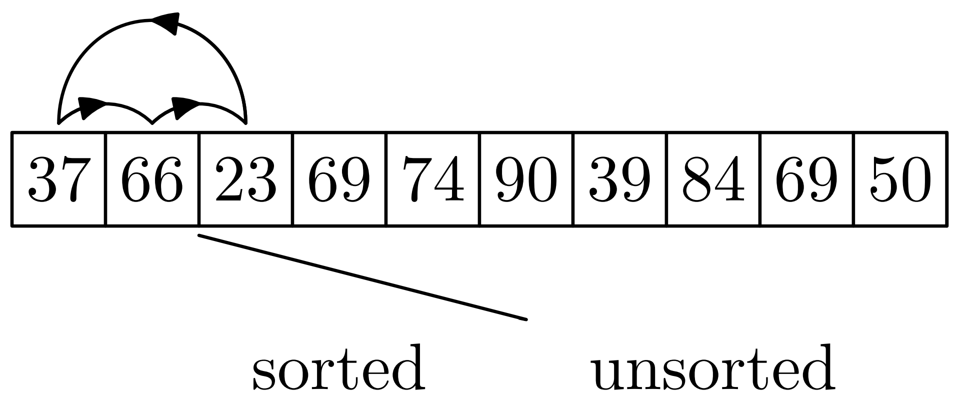
```
InsertionSort(a)
{
  for i←2 to n
    v←a_i
    j←i-1
    while j ≥ 1 and a_j>v
      a_{j+1}←a_j
      j←j-1
    endwhile
    a_{j+1}←v
  endfor
}
```



Insertion Sort

- In insertion sort we keep a subsequence of elements on the left in sorted order
- This subsequence is increased by *inserting* the next element into its correct position

INSERTIONSORT($\boldsymbol{a}$)
{
  for i $\leftarrow$ 2 **to** n
    v $\leftarrow a_i$
    j $\leftarrow$ i-1
    while j $\geq$ 1 **and** $a_j >$ v
      $a_{j+1} \leftarrow a_j$
      j $\leftarrow$ j-1
    endwhile
    $a_{j+1} \leftarrow$ v
  endfor
}



Insertion Sort

- In insertion sort we keep a subsequence of elements on the left in sorted order

- This subsequence is increased by *inserting* the next element into its correct position

```
InsertionSort(a)
{
  for i←2 to n
    v←a_i
    j←i-1
    while j ≥ 1 and a_j>v
      a_{j+1}←a_j
      j←j-1
    endwhile
    a_{j+1}←v
  endfor
}
```

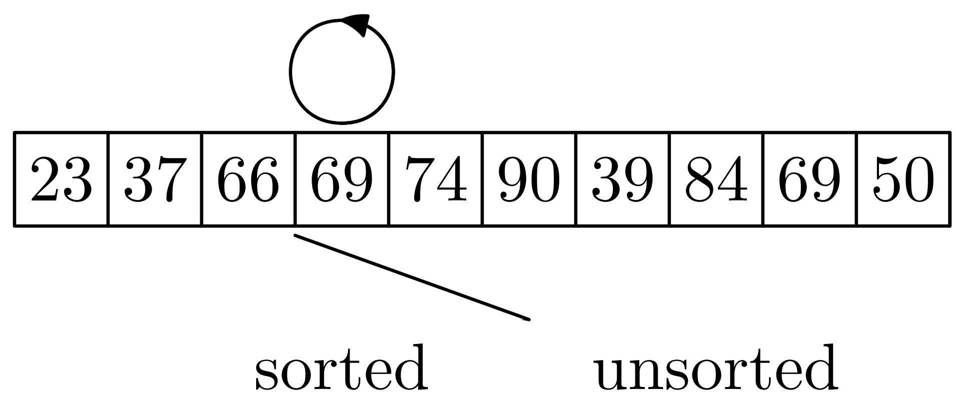
23	37	66	69	74	90	39	84	69	50

sorted unsorted

Insertion Sort

- In insertion sort we keep a subsequence of elements on the left in sorted order
- This subsequence is increased by *inserting* the next element into its correct position

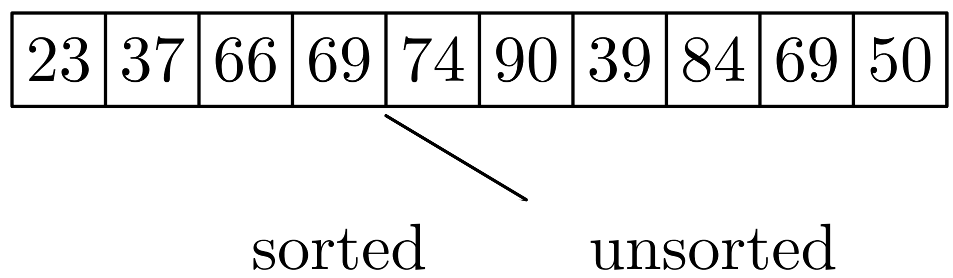
```
INSERTIONSORT(a)
{
  for i←2 to n
    v←a_i
    j←i-1
    while j ≥ 1 and a_j>v
      a_{j+1}←a_j
      j←j-1
    endwhile
    a_{j+1}←v
  endfor
}
```



Insertion Sort

- In insertion sort we keep a subsequence of elements on the left in sorted order
- This subsequence is increased by *inserting* the next element into its correct position

```
INSERTIONSORT(a)
{
  for i←2 to n
    v←a_i
    j←i-1
    while j ≥ 1 and a_j>v
      a_{j+1}←a_j
      j←j-1
    endwhile
    a_{j+1}←v
  endfor
}
```

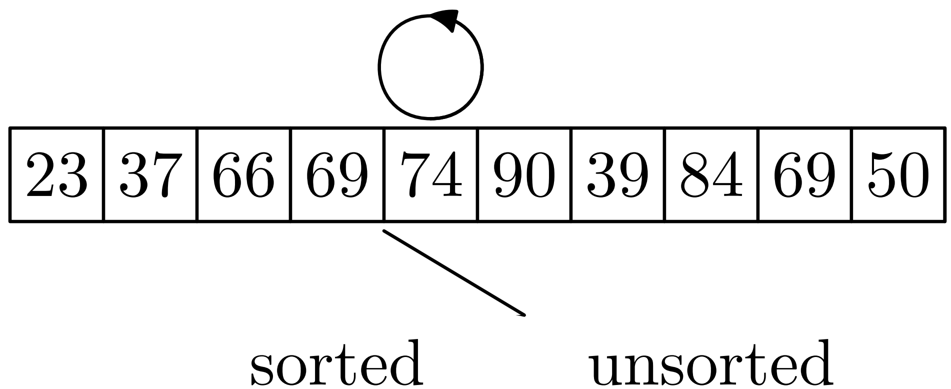


Insertion Sort

- In insertion sort we keep a subsequence of elements on the left in sorted order

- This subsequence is increased by *inserting* the next element into its correct position

```
InsertionSort(a)
{
  for i←2 to n
    v←a_i
    j←i-1
    while j ≥ 1 and a_j>v
      a_{j+1}←a_j
      j←j-1
    endwhile
    a_{j+1}←v
  endfor
}
```



Insertion Sort

- In insertion sort we keep a subsequence of elements on the left in sorted order
- This subsequence is increased by *inserting* the next element into its correct position

```
InsertionSort(a)
{
  for i←2 to n
    v←a_i
    j←i-1
    while j ≥ 1 and a_j>v
      a_{j+1}←a_j
      j←j-1
    endwhile
    a_{j+1}←v
  endfor
}
```

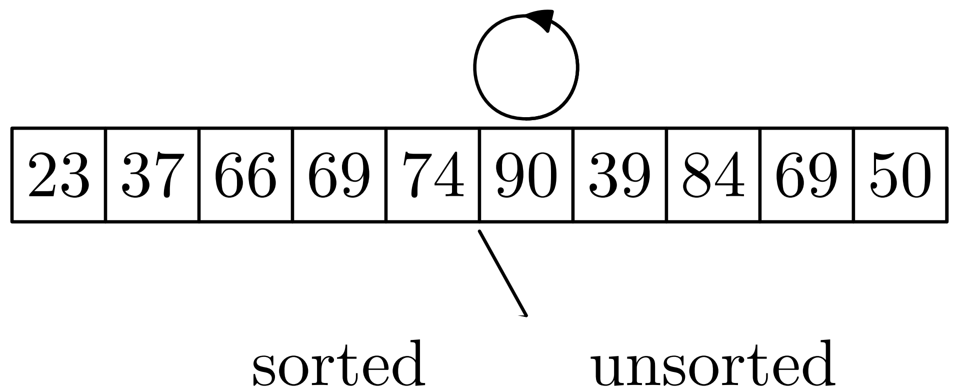
23	37	66	69	74	90	39	84	69	50

sorted unsorted

Insertion Sort

- In insertion sort we keep a subsequence of elements on the left in sorted order
- This subsequence is increased by *inserting* the next element into its correct position

```
InsertionSort(a)
{
  for i←2 to n
    v←a_i
    j←i-1
    while j ≥ 1 and a_j>v
      a_{j+1}←a_j
      j←j-1
    endwhile
    a_{j+1}←v
  endfor
}
```

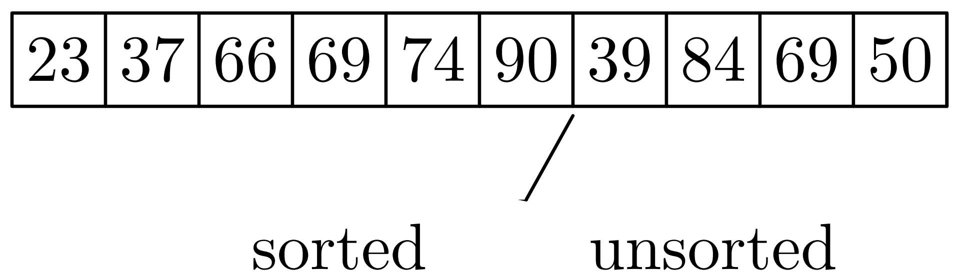


Insertion Sort

- In insertion sort we keep a subsequence of elements on the left in sorted order

- This subsequence is increased by *inserting* the next element into its correct position

```
InsertionSort(a)
{
  for i←2 to n
    v←a_i
    j←i-1
    while j ≥ 1 and a_j>v
      a_{j+1}←a_j
      j←j-1
    endwhile
    a_{j+1}←v
  endfor
}
```

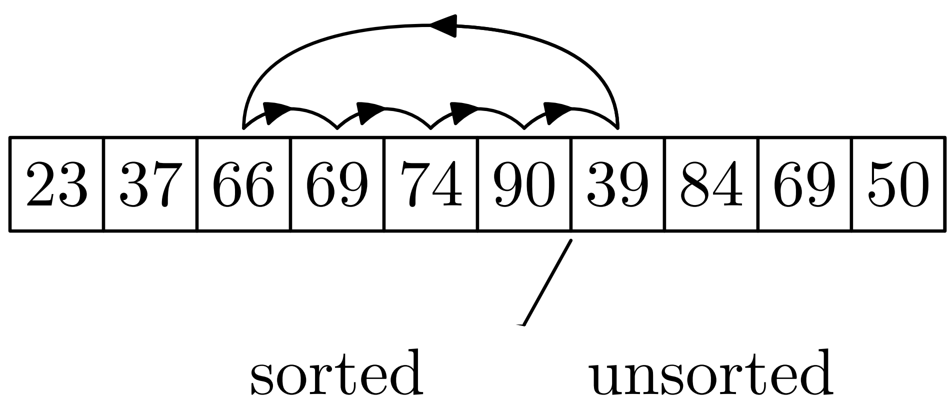


Insertion Sort

- In insertion sort we keep a subsequence of elements on the left in sorted order

- This subsequence is increased by *inserting* the next element into its correct position

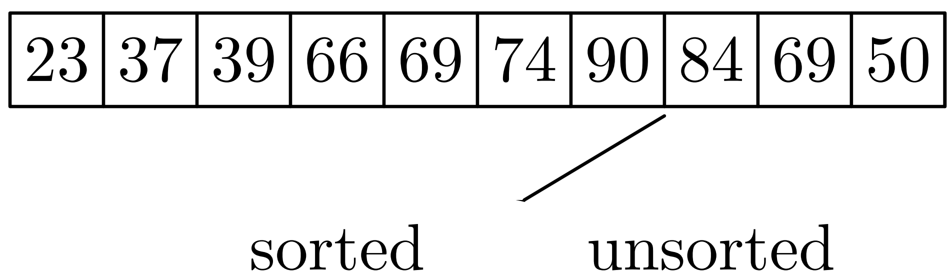
```
INSERTIONSORT(a)
{
  for i←2 to n
    v←a_i
    j←i-1
    while j ≥ 1 and a_j>v
      a_{j+1}←a_j
      j←j-1
    endwhile
    a_{j+1}←v
  endfor
}
```



Insertion Sort

- In insertion sort we keep a subsequence of elements on the left in sorted order
- This subsequence is increased by *inserting* the next element into its correct position

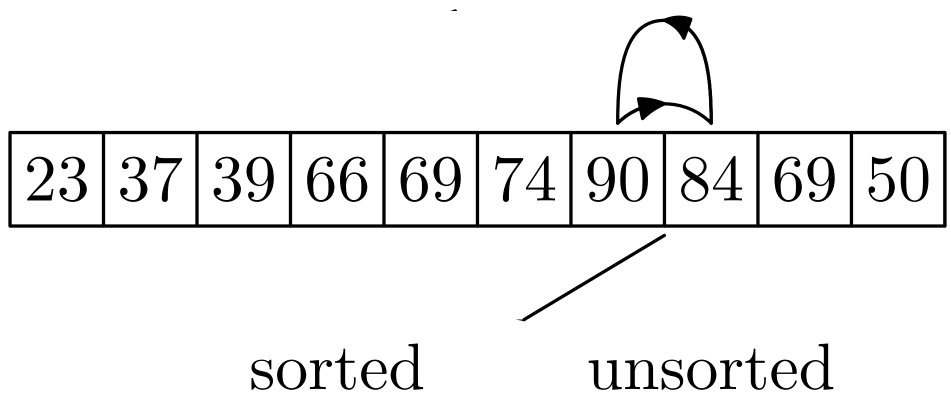
```
InsertionSort(a)
{
  for i←2 to n
    v←a_i
    j←i-1
    while j ≥ 1 and a_j>v
      a_{j+1}←a_j
      j←j-1
    endwhile
    a_{j+1}←v
  endfor
}
```



Insertion Sort

- In insertion sort we keep a subsequence of elements on the left in sorted order
- This subsequence is increased by *inserting* the next element into its correct position

```
InsertionSort(a)
{
  for i←2 to n
    v←a_i
    j←i-1
    while j ≥ 1 and a_j>v
      a_{j+1}←a_j
      j←j-1
    endwhile
    a_{j+1}←v
  endfor
}
```

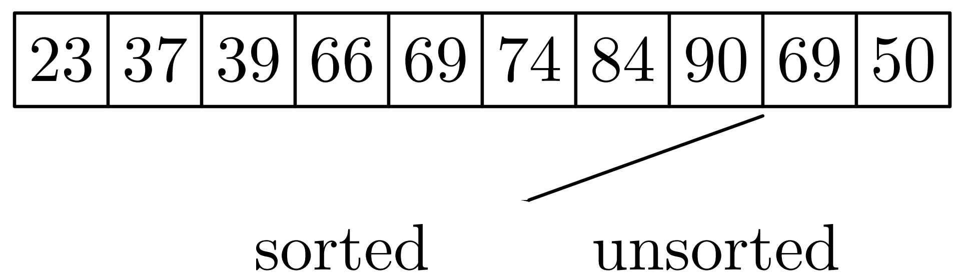


Insertion Sort

- In insertion sort we keep a subsequence of elements on the left in sorted order

- This subsequence is increased by *inserting* the next element into its correct position

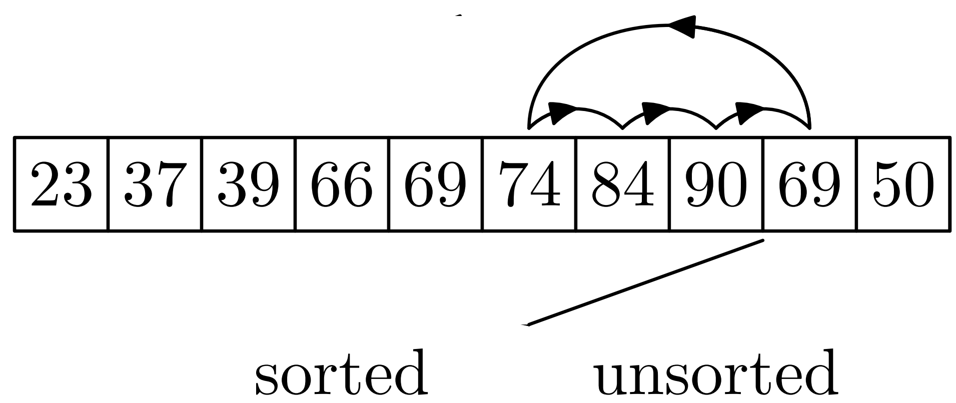
```
INSERTIONSORT(a)
{
  for i←2 to n
    v←a_i
    j←i-1
    while j ≥ 1 and a_j>v
      a_{j+1}←a_j
      j←j-1
    endwhile
    a_{j+1}←v
  endfor
}
```



Insertion Sort

- In insertion sort we keep a subsequence of elements on the left in sorted order
- This subsequence is increased by *inserting* the next element into its correct position

```
InsertionSort(a)
{
  for i←2 to n
    v←a_i
    j←i-1
    while j ≥ 1 and a_j>v
      a_{j+1}←a_j
      j←j-1
    endwhile
    a_{j+1}←v
  endfor
}
```



Insertion Sort

- In insertion sort we keep a subsequence of elements on the left in sorted order
- This subsequence is increased by *inserting* the next element into its correct position

```
InsertionSort(a)
{
  for i←2 to n
    v←a_i
    j←i-1
    while j ≥ 1 and a_j>v
      a_{j+1}←a_j
      j←j-1
    endwhile
    a_{j+1}←v
  endfor
}
```

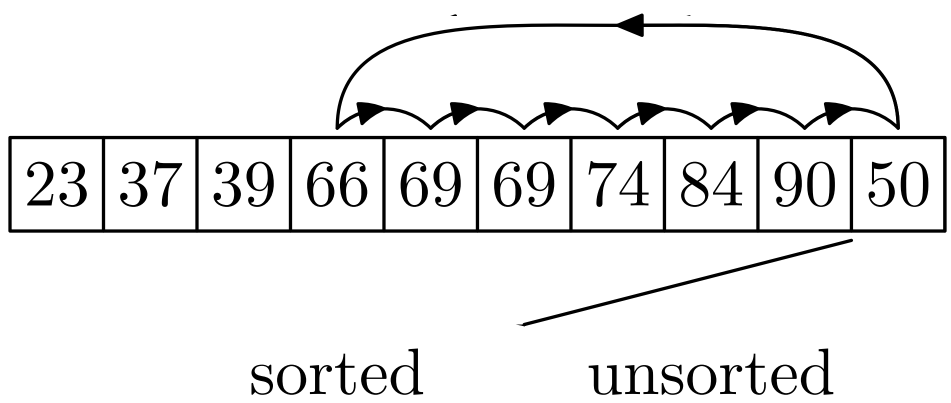
23	37	39	66	69	69	74	84	90	50

sorted unsorted

Insertion Sort

- In insertion sort we keep a subsequence of elements on the left in sorted order
- This subsequence is increased by *inserting* the next element into its correct position

```
INSERTIONSORT(a)
{
  for i←2 to n
    v←a_i
    j←i-1
    while j ≥ 1 and a_j>v
      a_{j+1}←a_j
      j←j-1
    endwhile
    a_{j+1}←v
  endfor
}
```



Insertion Sort

- In insertion sort we keep a subsequence of elements on the left in sorted order
- This subsequence is increased by *inserting* the next element into its correct position

```
INSERTIONSORT(a)
{
  for i←2 to n
    v←a_i
    j←i-1
    while j ≥ 1 and a_j>v
      a_{j+1}←a_j
      j←j-1
    endwhile
    a_{j+1}←v
  endfor
}
```

23	37	39	50	66	69	69	74	84	90

sorted unsorted

Properties of Insertion Sort

- Insertion sort is **stable**. We only swap the ordering of two elements if one is strictly less than the other

- It is **in-place**

- Worst time complexity
  - ⋆ Occurs when the array is in inverse order
  - ⋆ Every element has to be moved to front of the array
  - ⋆ Number of comparisons for an array of size $C_w(n)$

$$C_w(n) = \sum_{i=2}^{n} (i-1) = 1 + 2 + \cdots n - 1 = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Properties of Insertion Sort

- Insertion sort is **stable**. We only swap the ordering of two elements if one is strictly less than the other
- It is **in-place**
- Worst time complexity
  - ⋆ Occurs when the array is in inverse order
  - ⋆ Every element has to be moved to front of the array
  - ⋆ Number of comparisons for an array of size $C_w(n)$

$$C_w(n) = \sum_{i=2}^{n}(i-1) = 1 + 2 + \cdots n - 1 = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Properties of Insertion Sort

- Insertion sort is **stable**. We only swap the ordering of two elements if one is strictly less than the other
- It is **in-place**
- Worst time complexity
  - ⋆ Occurs when the array is in inverse order
  - ⋆ Every element has to be moved to front of the array
  - ⋆ Number of comparisons for an array of size $C_w(n)$

$$C_w(n) = \sum_{i=2}^{n}(i-1) = 1 + 2 + \cdots n - 1 = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Properties of Insertion Sort

- Insertion sort is **stable**. We only swap the ordering of two elements if one is strictly less than the other
- It is **in-place**
- Worst time complexity
  - ⋆ Occurs when the array is in inverse order
  - ⋆ Every element has to be moved to front of the array
  - ⋆ Number of comparisons for an array of size $C_w(n)$

$$C_w(n) = \sum_{i=2}^{n}(i-1) = 1 + 2 + \cdots n - 1 = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Properties of Insertion Sort

- Insertion sort is **stable**. We only swap the ordering of two elements if one is strictly less than the other

- It is **in-place**

- Worst time complexity
  - ⋆ Occurs when the array is in inverse order
  - ⋆ Every element has to be moved to front of the array
  - ⋆ Number of comparisons for an array of size $C_w(n)$

$$C_w(n) = \sum_{i=2}^{n}(i-1) = 1 + 2 + \cdots n - 1 = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Properties of Insertion Sort

- Insertion sort is **stable**. We only swap the ordering of two elements if one is strictly less than the other
- It is **in-place**
- Worst time complexity
  - ⋆ Occurs when the array is in inverse order
  - ⋆ Every element has to be moved to front of the array
  - ⋆ Number of comparisons for an array of size $C_w(n)$

$$C_w(n) = \sum_{i=2}^{n}(i-1) = 1 + 2 + \cdots n - 1 = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Properties of Insertion Sort

- Insertion sort is **stable**. We only swap the ordering of two elements if one is strictly less than the other
- It is **in-place**
- Worst time complexity
  - ⋆ Occurs when the array is in inverse order
  - ⋆ Every element has to be moved to front of the array
  - ⋆ Number of comparisons for an array of size $C_w(n)$

$$C_w(n) = \sum_{i=2}^{n}(i-1) = 1 + 2 + \cdots n - 1 = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Properties of Insertion Sort

- Insertion sort is **stable**. We only swap the ordering of two elements if one is strictly less than the other
- It is **in-place**
- Worst time complexity
  - ⋆ Occurs when the array is in inverse order
  - ⋆ Every element has to be moved to front of the array
  - ⋆ Number of comparisons for an array of size $C_w(n)$

$$C_w(n) = \sum_{i=2}^{n}(i-1) = 1 + 2 + \cdots n - 1 = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Properties of Insertion Sort

- Insertion sort is **stable**. We only swap the ordering of two elements if one is strictly less than the other
- It is **in-place**
- Worst time complexity
  - ⋆ Occurs when the array is in inverse order
  - ⋆ Every element has to be moved to front of the array
  - ⋆ Number of comparisons for an array of size $C_w(n)$

$$C_w(n) = \sum_{i=2}^{n}(i-1) = 1 + 2 + \cdots n - 1 = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Time Complexity

- Average Time Complexity
  - ⋆ On average we can expect that each new element being sorted moves half the way down sorted list
  - ⋆ This gives us an average time complexity, $C_a(n)$ of half the worst time

$$C_a(n) = \frac{n(n-1)}{4} \in \Theta(n^2)$$

- Best Time Complexity
  - ⋆ This occurs if the array is already sorted
  - ⋆ In this case we only need $C_b(n) = n - 1 \in \Theta(n)$ comparisons

- Insertion sort is a good sort for small arrays because it is stable, in-place and is efficient when the arrays are almost sorted

Time Complexity

- Average Time Complexity
  - ⋆ On average we can expect that each new element being sorted moves half the way down sorted list
  - ⋆ This gives us an average time complexity, $C_a(n)$ of half the worst time

$$C_a(n) = \frac{n(n-1)}{4} \in \Theta(n^2)$$

- Best Time Complexity
  - ⋆ This occurs if the array is already sorted
  - ⋆ In this case we only need $C_b(n) = n - 1 \in \Theta(n)$ comparisons

- Insertion sort is a good sort for small arrays because it is stable, in-place and is efficient when the arrays are almost sorted

Time Complexity

- Average Time Complexity
  - ⋆ On average we can expect that each new element being sorted moves half the way down sorted list
  - ⋆ This gives us an average time complexity, $C_a(n)$ of half the worst time

$$C_a(n) = \frac{n(n-1)}{4} \in \Theta(n^2)$$

- Best Time Complexity
  - ⋆ This occurs if the array is already sorted
  - ⋆ In this case we only need $C_b(n) = n - 1 \in \Theta(n)$ comparisons
- Insertion sort is a good sort for small arrays because it is stable, in-place and is efficient when the arrays are almost sorted

Time Complexity

- Average Time Complexity
  - ⋆ On average we can expect that each new element being sorted moves half the way down sorted list
  - ⋆ This gives us an average time complexity, $C_a(n)$ of half the worst time

$$C_a(n) = \frac{n(n-1)}{4} \in \Theta(n^2)$$

- Best Time Complexity
  - ⋆ This occurs if the array is already sorted
  - ⋆ In this case we only need $C_b(n) = n - 1 \in \Theta(n)$ comparisons

- Insertion sort is a good sort for small arrays because it is stable, in-place and is efficient when the arrays are almost sorted

Time Complexity

- Average Time Complexity
  - ⋆ On average we can expect that each new element being sorted moves half the way down sorted list
  - ⋆ This gives us an average time complexity, $C_a(n)$ of half the worst time

$$C_a(n) = \frac{n(n-1)}{4} \in \Theta(n^2)$$

- Best Time Complexity
  - ⋆ This occurs if the array is already sorted
  - ⋆ In this case we only need $C_b(n) = n - 1 \in \Theta(n)$ comparisons

- Insertion sort is a good sort for small arrays because it is stable, in-place and is efficient when the arrays are almost sorted

Time Complexity

- Average Time Complexity
  - ⋆ On average we can expect that each new element being sorted moves half the way down sorted list
  - ⋆ This gives us an average time complexity, $C_a(n)$ of half the worst time

$$C_a(n) = \frac{n(n-1)}{4} \in \Theta(n^2)$$

- Best Time Complexity
  - ⋆ This occurs if the array is already sorted
  - ⋆ In this case we only need $C_b(n) = n - 1 \in \Theta(n)$ comparisons

- Insertion sort is a good sort for small arrays because it is stable, in-place and is efficient when the arrays are almost sorted

Time Complexity

- Average Time Complexity
  - ⋆ On average we can expect that each new element being sorted moves half the way down sorted list
  - ⋆ This gives us an average time complexity, $C_a(n)$ of half the worst time

$$C_a(n) = \frac{n(n-1)}{4} \in \Theta(n^2)$$

- Best Time Complexity
  - ⋆ This occurs if the array is already sorted
  - ⋆ In this case we only need $C_b(n) = n - 1 \in \Theta(n)$ comparisons
- Insertion sort is a good sort for small arrays because it is stable, in-place and is efficient when the arrays are almost sorted

Time Complexity

- Average Time Complexity
  - ⋆ On average we can expect that each new element being sorted moves half the way down sorted list
  - ⋆ This gives us an average time complexity, $C_a(n)$ of half the worst time

$$C_a(n) = \frac{n(n-1)}{4} \in \Theta(n^2)$$

- Best Time Complexity
  - ⋆ This occurs if the array is already sorted
  - ⋆ In this case we only need $C_b(n) = n - 1 \in \Theta(n)$ comparisons

- Insertion sort is a good sort for small arrays because it is stable, in-place and is efficient when the arrays are almost sorted

Selection Sort

- A more direct **brute force** method is to find the least element iteratively

- We can make this an in-place method by swapping the least element with the first element, the second least element with the second element, etc.

```
SelectionSort(a)
{
  for i←1 to n-1
    min ←i
    for j←i+1 to n
      if a_j<a_min
        min←j
      end if
    end for
    swap a_i  and a_min
  end for
}
```

Selection Sort

- A more direct **brute force** method is to find the least element iteratively
- We can make this an in-place method by swapping the least element with the first element, the second least element with the second element, etc.

```
SelectionSort(a)
{
 for i←1 to n-1
   min ←i
   for j←i+1 to n
     if a_j<a_min
       min←j
     end if
   end for
   swap a_i  and a_min
 end for
}
```

Selection Sort

- A more direct **brute force** method is to find the least element iteratively

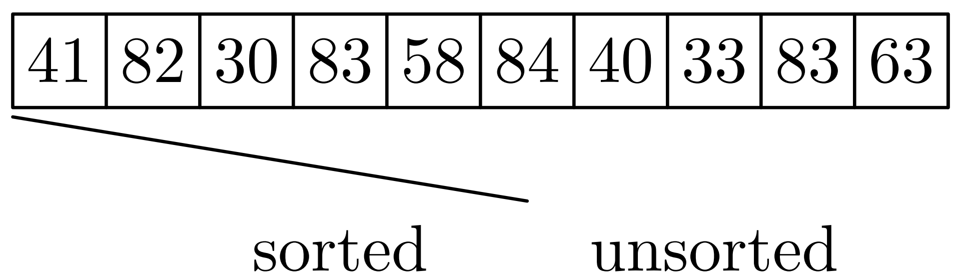
- We can make this an in-place method by swapping the least element with the first element, the second least element with the second element, etc.

```
SELECTIONSORT(a)
{
 for i←1 to n−1
   min ←i
   for j←i+1 to n
     if a_j<a_min
       min←j
     end if
   end for
   swap a_i  and a_min
 end for
}
```

Selection Sort

- A more direct **brute force** method is to find the least element iteratively
- We can make this an in-place method by swapping the least element with the first element, the second least element with the second element, etc.

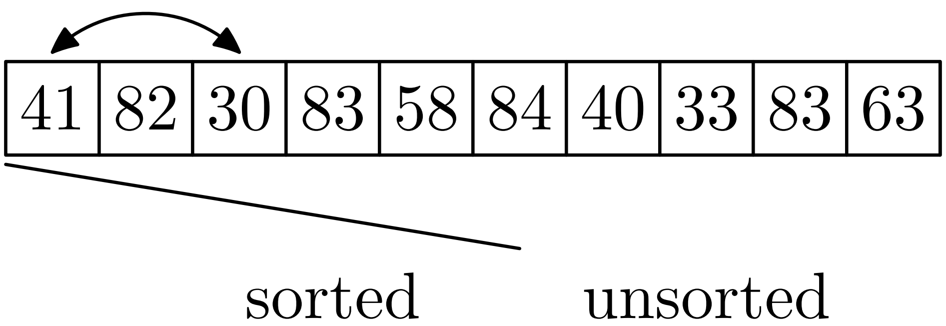
```
SelectionSort(a)
{
 for i←1 to n−1
  min ←i
  for j←i+1 to n
   if a_j<a_min
    min←j
   end if
  end for
  swap a_i and a_min
 end for
}
```



Selection Sort

- A more direct **brute force** method is to find the least element iteratively
- We can make this an in-place method by swapping the least element with the first element, the second least element with the second element, etc.

```
SelectionSort(a)
{
  for i←1 to n-1
    min ←i
    for j←i+1 to n
      if a_j<a_min
        min←j
      end if
    end for
    swap a_i  and a_min
  end for
}
```



Selection Sort

- A more direct **brute force** method is to find the least element iteratively

- We can make this an in-place method by swapping the least element with the first element, the second least element with the second element, etc.

```
SELECTIONSORT(a)
{
  for i←1 to n−1
    min ←i
    for j←i+1 to n
      if a_j<a_min
        min←j
      end if
    end for
    swap a_i  and a_min
  end for
}
```

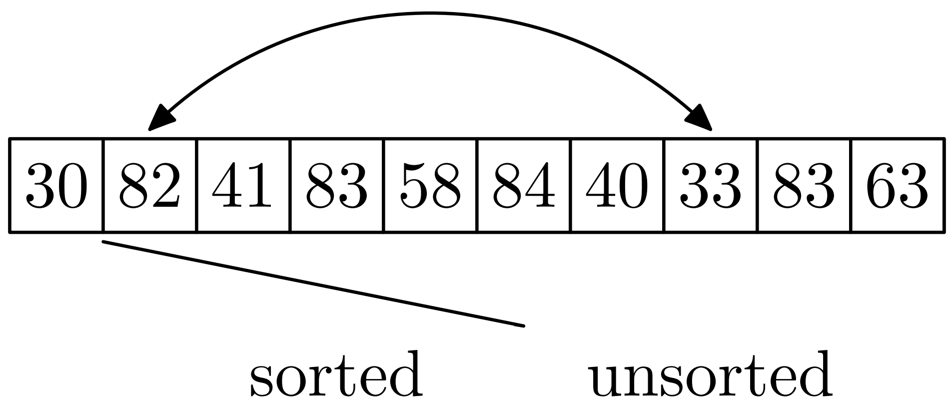
30	82	41	83	58	84	40	33	83	63

sorted unsorted

Selection Sort

- A more direct **brute force** method is to find the least element iteratively
- We can make this an in-place method by swapping the least element with the first element, the second least element with the second element, etc.

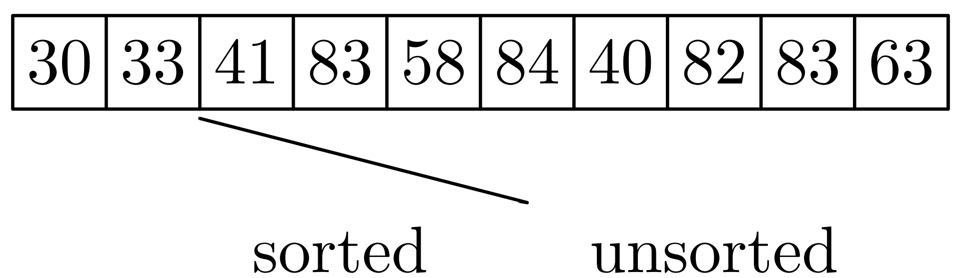
```
SelectionSort(a)
{
  for i←1 to n−1
    min ←i
    for j←i+1 to n
      if a_j<a_min
        min←j
      end if
    end for
    swap a_i  and a_min
  end for
}
```



Selection Sort

- A more direct **brute force** method is to find the least element iteratively
- We can make this an in-place method by swapping the least element with the first element, the second least element with the second element, etc.

```
SelectionSort(a)
{
  for i←1 to n−1
    min ←i
    for j←i+1 to n
      if a_j<a_min
        min←j
      end if
    end for
    swap a_i  and a_min
  end for
}
```

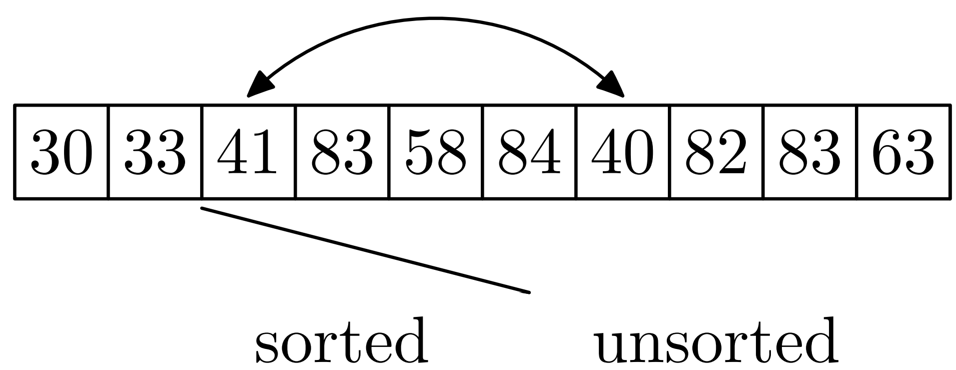


Selection Sort

- A more direct **brute force** method is to find the least element iteratively

- We can make this an in-place method by swapping the least element with the first element, the second least element with the second element, etc.

```
SelectionSort(a)
{
  for i←1 to n-1
    min ←i
    for j←i+1 to n
      if a_j<a_min
        min←j
      end if
    end for
    swap a_i  and a_min
  end for
}
```



Selection Sort

- A more direct **brute force** method is to find the least element iteratively
- We can make this an in-place method by swapping the least element with the first element, the second least element with the second element, etc.

```
SELECTIONSORT(a)
{
  for i←1 to n−1
    min ←i
    for j←i+1 to n
      if a_j<a_min
        min←j
      end if
    end for
    swap a_i  and a_min
  end for
}
```

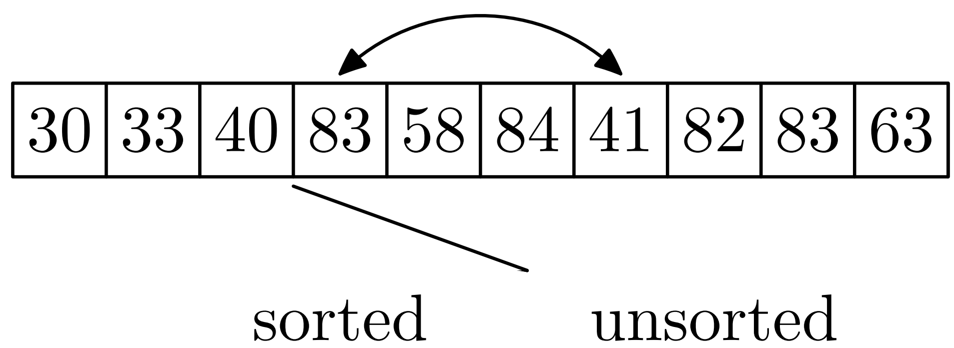
30	33	40	83	58	84	41	82	83	63

sorted unsorted

Selection Sort

- A more direct **brute force** method is to find the least element iteratively
- We can make this an in-place method by swapping the least element with the first element, the second least element with the second element, etc.

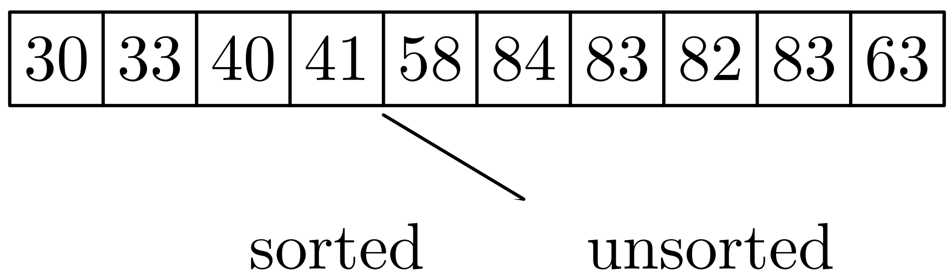
```
SelectionSort(a)
{
  for i←1 to n-1
   min ←i
   for j←i+1 to n
    if a_j<a_min
     min←j
    end if
   end for
   swap a_i  and a_min
  end for
}
```



Selection Sort

- A more direct **brute force** method is to find the least element iteratively
- We can make this an in-place method by swapping the least element with the first element, the second least element with the second element, etc.

```
SelectionSort(a)
{
  for i←1 to n−1
    min ←i
    for j←i+1 to n
      if a_j<a_min
        min←j
      end if
    end for
    swap a_i  and a_min
  end for
}
```

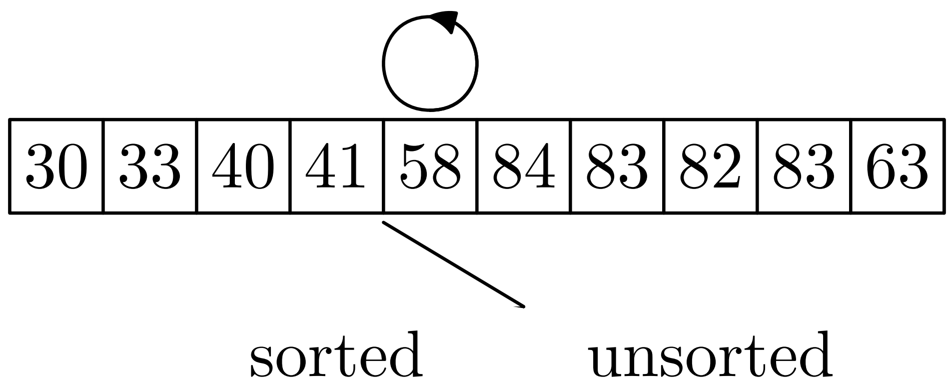


Selection Sort

- A more direct **brute force** method is to find the least element iteratively

- We can make this an in-place method by swapping the least element with the first element, the second least element with the second element, etc.

```
SELECTIONSORT(a)
{
  for i←1 to n-1
    min ←i
    for j←i+1 to n
      if a_j<a_min
        min←j
      end if
    end for
    swap a_i  and a_min
  end for
}
```



Selection Sort

- A more direct **brute force** method is to find the least element iteratively
- We can make this an in-place method by swapping the least element with the first element, the second least element with the second element, etc.

```
SelectionSort(a)
{
  for i←1 to n-1
    min ←i
    for j←i+1 to n
      if a_j<a_min
        min←j
      end if
    end for
    swap a_i  and a_min
  end for
}
```

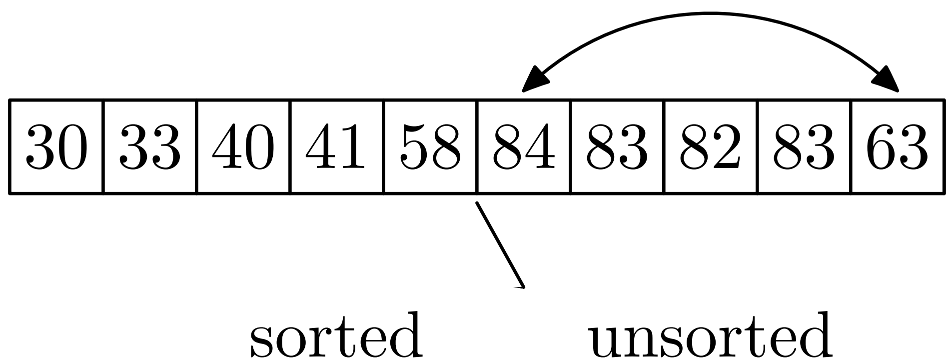
30	33	40	41	58	84	83	82	83	63

sorted unsorted

Selection Sort

- A more direct **brute force** method is to find the least element iteratively
- We can make this an in-place method by swapping the least element with the first element, the second least element with the second element, etc.

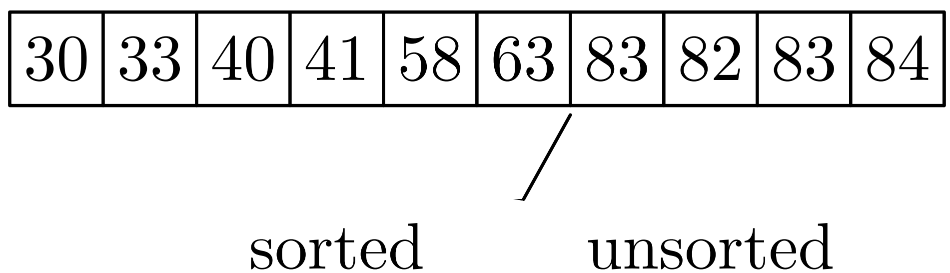
```
SELECTIONSORT(a)
{
 for i←1 to n−1
  min ←i
  for j←i+1 to n
   if a_j<a_min
    min←j
   end if
  end for
  swap a_i and a_min
 end for
}
```



Selection Sort

- A more direct **brute force** method is to find the least element iteratively
- We can make this an in-place method by swapping the least element with the first element, the second least element with the second element, etc.

```
SELECTIONSORT(a)
{
  for i←1 to n−1
    min ←i
    for j←i+1 to n
      if a_j<a_min
        min←j
      end if
    end for
    swap a_i  and a_min
  end for
}
```

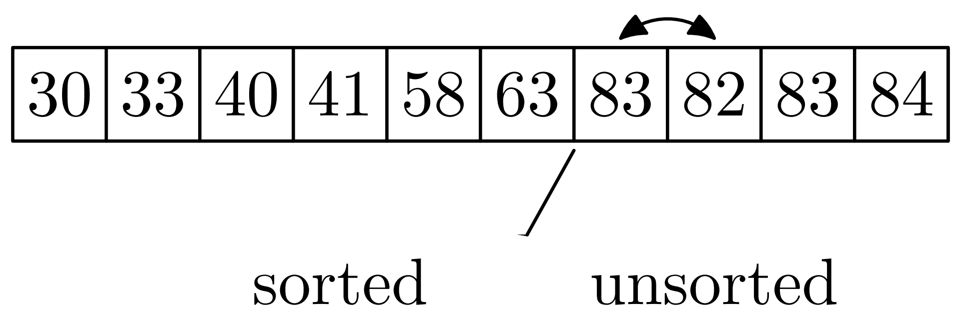


Selection Sort

- A more direct **brute force** method is to find the least element iteratively

- We can make this an in-place method by swapping the least element with the first element, the second least element with the second element, etc.

```
SELECTIONSORT(a)
{
  for i←1 to n−1
    min ←i
    for j←i+1 to n
      if a_j<a_min
        min←j
      end if
    end for
    swap a_i  and a_min
  end for
}
```



Selection Sort

- A more direct **brute force** method is to find the least element iteratively
- We can make this an in-place method by swapping the least element with the first element, the second least element with the second element, etc.

```
SelectionSort(a)
{
  for i←1 to n-1
    min ←i
    for j←i+1 to n
      if a_j<a_min
        min←j
      end if
    end for
    swap a_i  and a_min
  end for
}
```

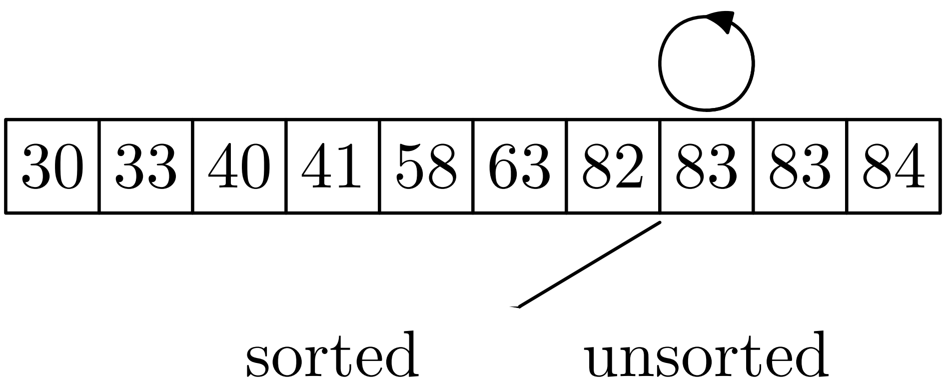
30	33	40	41	58	63	82	83	83	84

sorted unsorted

Selection Sort

- A more direct **brute force** method is to find the least element iteratively
- We can make this an in-place method by swapping the least element with the first element, the second least element with the second element, etc.

```
SelectionSort(a)
{
  for i←1 to n−1
    min ←i
    for j←i+1 to n
      if a_j<a_min
        min←j
      end if
    end for
    swap a_i  and a_min
  end for
}
```



Selection Sort

- A more direct **brute force** method is to find the least element iteratively
- We can make this an in-place method by swapping the least element with the first element, the second least element with the second element, etc.

```
SELECTIONSORT(a)
{
 for i←1 to n−1
   min ←i
   for j←i+1 to n
     if a_j<a_min
       min←j
     end if
   end for
   swap a_i  and a_min
 end for
}
```

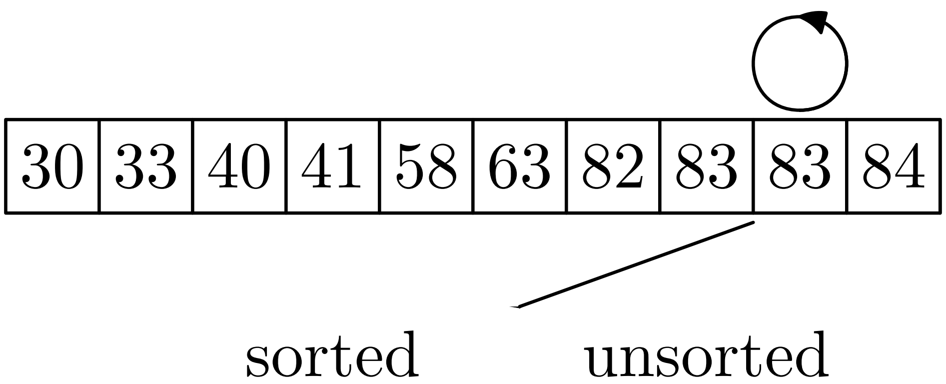
30	33	40	41	58	63	82	83	83	84

sorted unsorted

Selection Sort

- A more direct **brute force** method is to find the least element iteratively
- We can make this an in-place method by swapping the least element with the first element, the second least element with the second element, etc.

```
SELECTIONSORT(a)
{
  for i←1 to n−1
    min ←i
    for j←i+1 to n
      if a_j<a_min
        min←j
      end if
    end for
    swap a_i and a_min
  end for
}
```



Selection Sort

- A more direct **brute force** method is to find the least element iteratively

- We can make this an in-place method by swapping the least element with the first element, the second least element with the second element, etc.

```
SelectionSort(a)
{
  for i←1 to n−1
    min ←i
    for j←i+1 to n
      if a_j<a_min
        min←j
      end if
    end for
    swap a_i  and a_min
  end for
}
```

30	33	40	41	58	63	82	83	83	84

sorted unsorted

Analysis of Selection Sort

- Selection sort is in-place
- It isn't stable

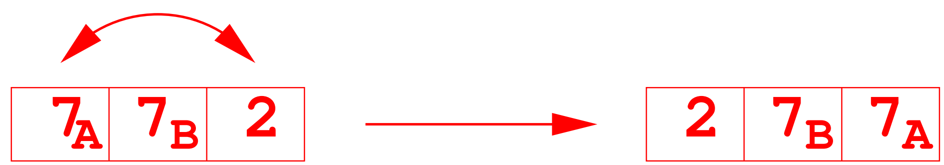
$$\boxed{7_A \mid 7_B \mid 2} \longrightarrow \boxed{2 \mid 7_B \mid 7_A}$$

- Selection sort always requires $n(n-1)/2$ comparisons so has the same worst case, but worse average case and best case complexity as insertion sort
- It only performs $n-1$ swaps—this makes it attractive (insertion sort moved more elements)

Analysis of Selection Sort

- Selection sort is in-place

- It isn't stable



- Selection sort always requires $n(n-1)/2$ comparisons so has the same worst case, but worse average case and best case complexity as insertion sort

- It only performs $n-1$ swaps—this makes it attractive (insertion sort moved more elements)

Analysis of Selection Sort

- Selection sort is in-place
- It isn't stable

- Selection sort always requires $n(n-1)/2$ comparisons so has the same worst case, but worse average case and best case complexity as insertion sort
- It only performs $n-1$ swaps—this makes it attractive (insertion sort moved more elements)

Analysis of Selection Sort

- Selection sort is in-place

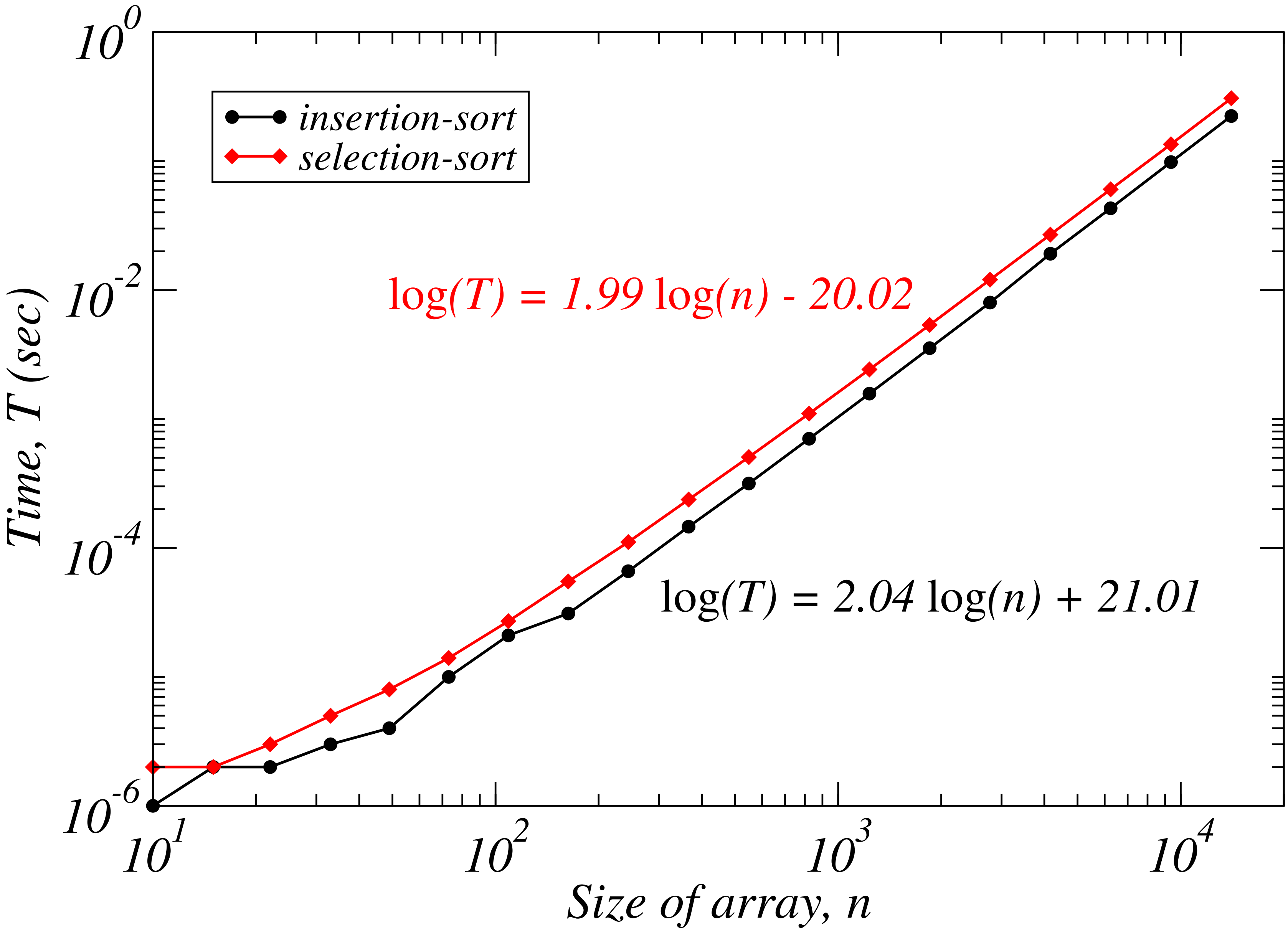
- It isn't stable

$$\boxed{7_A}\boxed{7_B}\boxed{2} \longrightarrow \boxed{2}\boxed{7_B}\boxed{7_A}$$

- Selection sort always requires $n(n-1)/2$ comparisons so has the same worst case, but worse average case and best case complexity as insertion sort

- It only performs $n-1$ swaps—this makes it attractive (insertion sort moved more elements)

Insertion versus Selection Sort



Bubble Sort

- There are many other simple sort strategies
- One popular one is bubble sort—keep on swapping neighbours until the array is sorted
- It is stable and in-place
- This again has $O(n^2)$ complexity
- This isn't bad for a simple sort, but it does do more work than insertion sort and selection sort
- Apart from its name it just doesn't have anything going for it

Bubble Sort

- There are many other simple sort strategies
- One popular one is bubble sort—keep on swapping neighbours until the array is sorted
- It is stable and in-place
- This again has $O(n^2)$ complexity
- This isn't bad for a simple sort, but it does do more work than insertion sort and selection sort
- Apart from its name it just doesn't have anything going for it

Bubble Sort

- There are many other simple sort strategies
- One popular one is bubble sort—keep on swapping neighbours until the array is sorted
- It is stable and in-place
- This again has $O(n^2)$ complexity
- This isn't bad for a simple sort, but it does do more work than insertion sort and selection sort
- Apart from its name it just doesn't have anything going for it

Bubble Sort

- There are many other simple sort strategies
- One popular one is bubble sort—keep on swapping neighbours until the array is sorted
- It is stable and in-place
- This again has $O(n^2)$ complexity
- This isn't bad for a simple sort, but it does do more work than insertion sort and selection sort
- Apart from its name it just doesn't have anything going for it

Bubble Sort

- There are many other simple sort strategies
- One popular one is bubble sort—keep on swapping neighbours until the array is sorted
- It is stable and in-place
- This again has $O(n^2)$ complexity
- This isn't bad for a simple sort, but it does do more work than insertion sort and selection sort
- Apart from its name it just doesn't have anything going for it

Bubble Sort

- There are many other simple sort strategies
- One popular one is bubble sort—keep on swapping neighbours until the array is sorted
- It is stable and in-place
- This again has $O(n^2)$ complexity
- This isn't bad for a simple sort, but it does do more work than insertion sort and selection sort
- Apart from its name it just doesn't have anything going for it

Bubble Sort

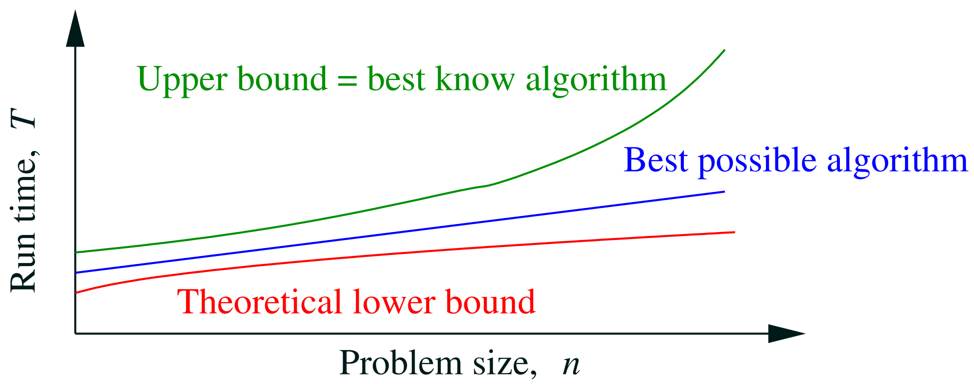
- There are many other simple sort strategies
- One popular one is bubble sort—keep on swapping neighbours until the array is sorted
- It is stable and in-place
- This again has $O(n^2)$ complexity
- This isn't bad for a simple sort, but it does do more work than insertion sort and selection sort
- Apart from its name it just doesn't have anything going for it

Outline

1. Algorithm Analysis
2. Search
3. Simple Sort
   - Insertion Sort
   - Selection Sort
4. **Lower Bound**

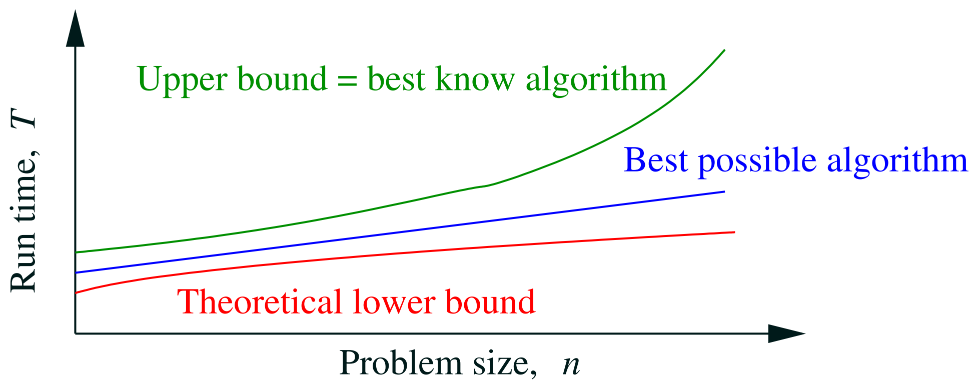
How Well Can You Do?

- Given a problem we would like to know what is the time complexity of the best possible program
- Usually there is no way of knowing this
- We can get an upper bound—if we know the time complexity of any algorithm that solves the problem we have an upper bound
- Lower bounds are far trickier
- A lower bound of $f(n)$ is a guarantee that we spend at least $f(n)$ operations to solve the problem



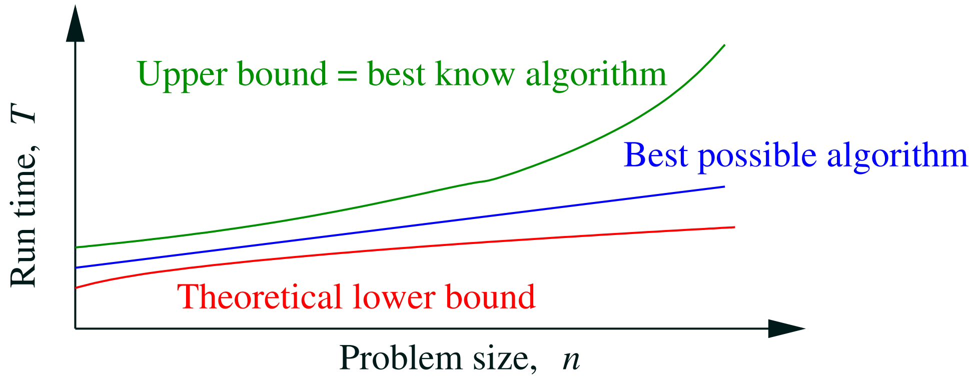
How Well Can You Do?

- Given a problem we would like to know what is the time complexity of the best possible program
- Usually there is no way of knowing this
- We can get an upper bound—if we know the time complexity of any algorithm that solves the problem we have an upper bound
- Lower bounds are far trickier
- A lower bound of $f(n)$ is a guarantee that we spend at least $f(n)$ operations to solve the problem



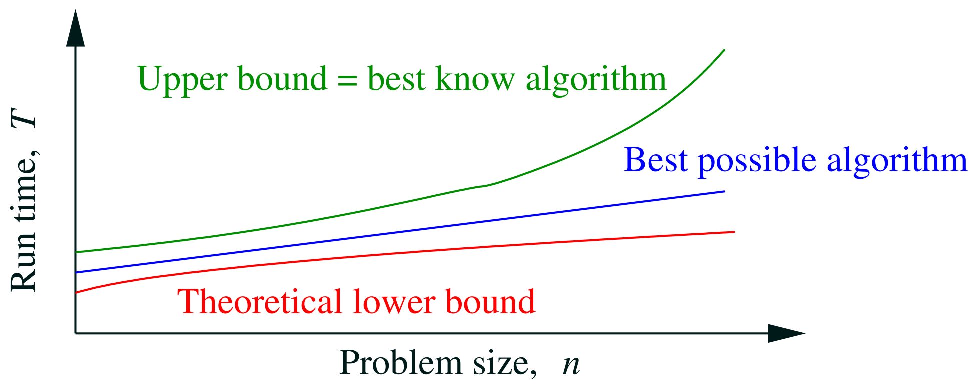
How Well Can You Do?

- Given a problem we would like to know what is the time complexity of the best possible program
- Usually there is no way of knowing this
- We can get an upper bound—if we know the time complexity of any algorithm that solves the problem we have an upper bound
- Lower bounds are far trickier
- A lower bound of $f(n)$ is a guarantee that we spend at least $f(n)$ operations to solve the problem



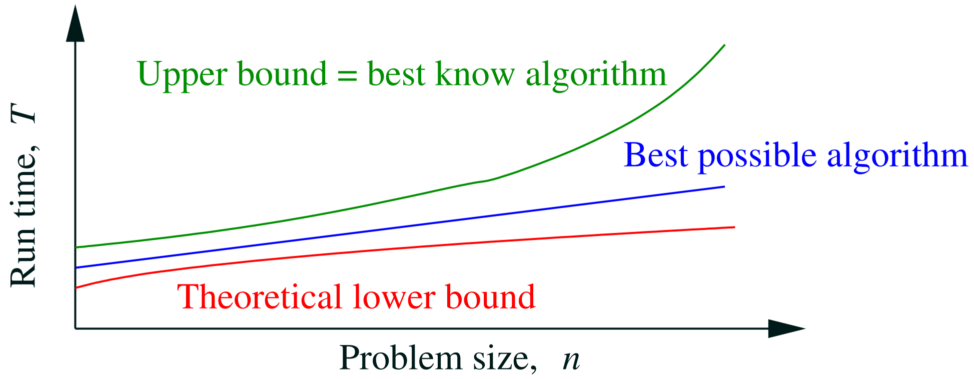
How Well Can You Do?

- Given a problem we would like to know what is the time complexity of the best possible program
- Usually there is no way of knowing this
- We can get an upper bound—if we know the time complexity of any algorithm that solves the problem we have an upper bound
- Lower bounds are far trickier
- A lower bound of $f(n)$ is a guarantee that we spend at least $f(n)$ operations to solve the problem



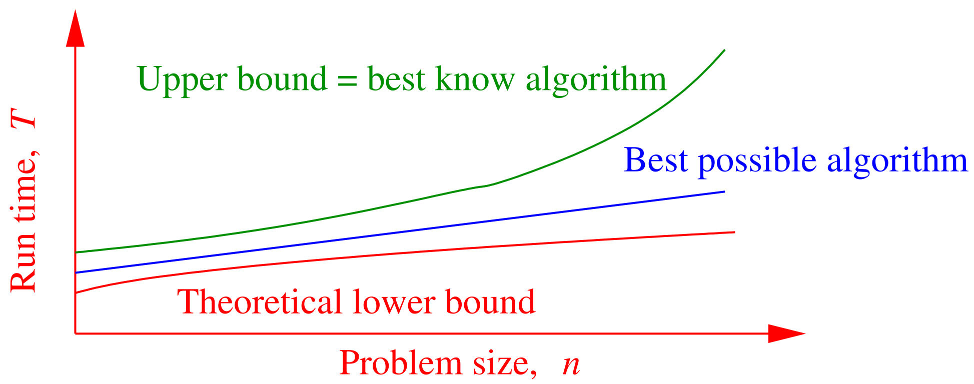
How Well Can You Do?

- Given a problem we would like to know what is the time complexity of the best possible program
- Usually there is no way of knowing this
- We can get an upper bound—if we know the time complexity of any algorithm that solves the problem we have an upper bound
- Lower bounds are far trickier
- A lower bound of $f(n)$ is a guarantee that we spend at least $f(n)$ operations to solve the problem



How Well Can You Do?

- Given a problem we would like to know what is the time complexity of the best possible program
- Usually there is no way of knowing this
- We can get an upper bound—if we know the time complexity of any algorithm that solves the problem we have an upper bound
- Lower bounds are far trickier
- A lower bound of $f(n)$ is a guarantee that we spend at least $f(n)$ operations to solve the problem



Decision Trees

- Decision trees are a way to visualise (at least, in principle) many algorithms
- They will eventually give us a lower bound on the time complexity of sort using binary decisions
- A decision tree shows the series of decisions made during an algorithm
- For sort based on binary comparisons the decision tree shows what the algorithm does after every comparison

Decision Trees

- Decision trees are a way to visualise (at least, in principle) many algorithms
- They will eventually give us a lower bound on the time complexity of sort using binary decisions
- A decision tree shows the series of decisions made during an algorithm
- For sort based on binary comparisons the decision tree shows what the algorithm does after every comparison

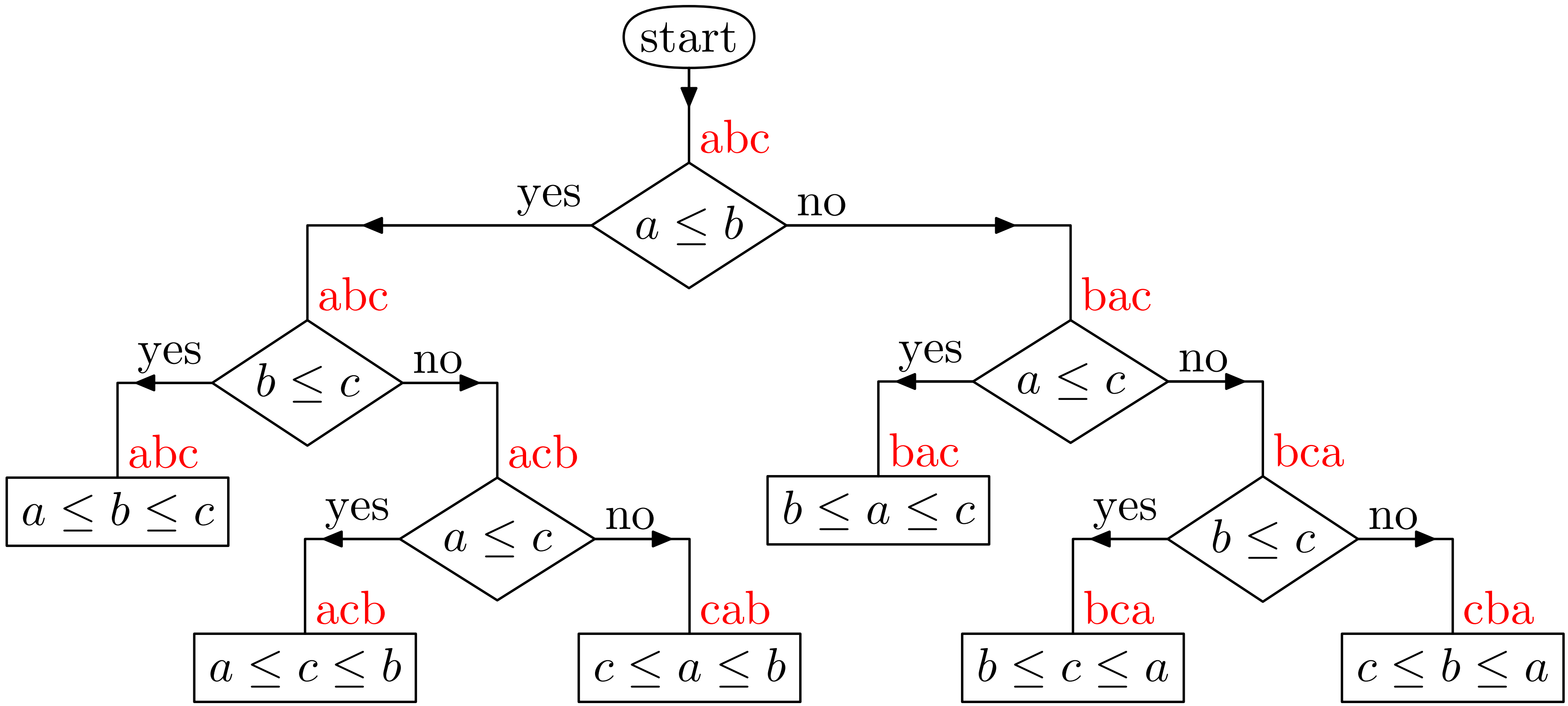
Decision Trees

- Decision trees are a way to visualise (at least, in principle) many algorithms
- They will eventually give us a lower bound on the time complexity of sort using binary decisions
- A decision tree shows the series of decisions made during an algorithm
- For sort based on binary comparisons the decision tree shows what the algorithm does after every comparison

Decision Trees

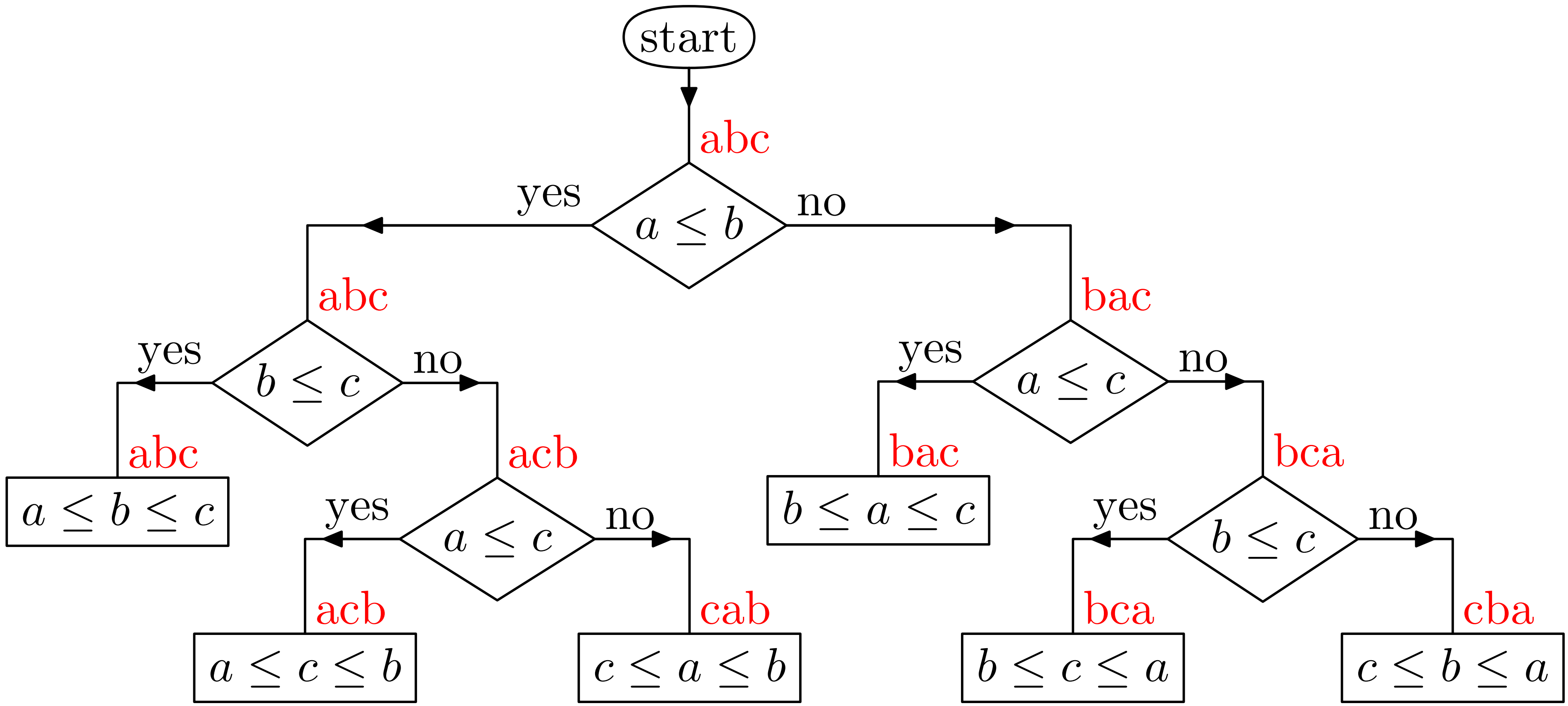
- Decision trees are a way to visualise (at least, in principle) many algorithms
- They will eventually give us a lower bound on the time complexity of sort using binary decisions
- A decision tree shows the series of decisions made during an algorithm
- For sort based on binary comparisons the decision tree shows what the algorithm does after every comparison

Decision Tree for Insertion Sort



- Note there is one leaf for every possible way of sorting the list

Decision Tree for Insertion Sort



- Note there is one leaf for every possible way of sorting the list

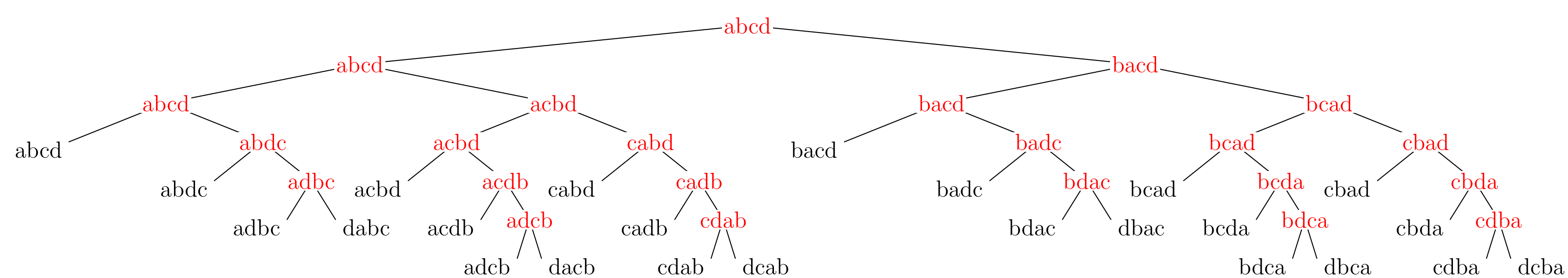
Decision Trees and Time Complexity

- The time taken to complete the task is the depth of the tree at which we finish (i.e. the leaf nodes)

- We can thus read of the time complexity
  - ⋆ worst case time: depth of the deepest of leaf
  - ⋆ best case time: depth of the shallowest of leaf
  - ⋆ average case time: average depth of leaves

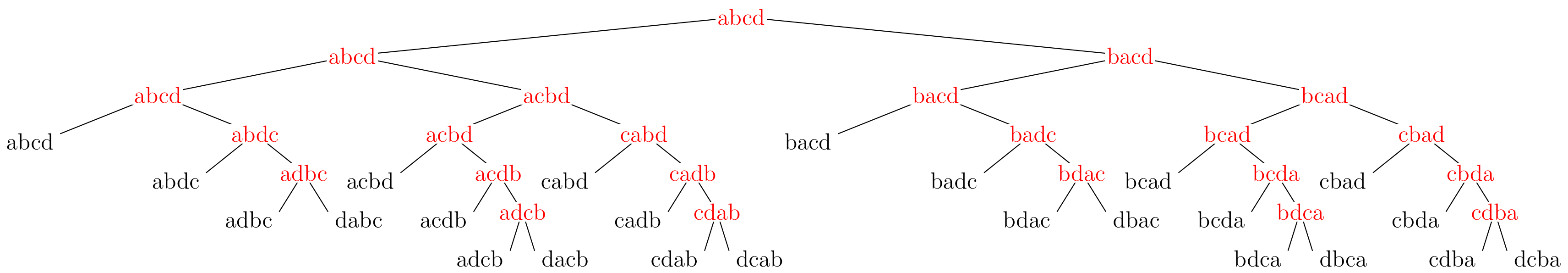
- Different sort strategies will have different decision trees

- Decision trees are usually far too large to write out ☹



Decision Trees and Time Complexity

- The time taken to complete the task is the depth of the tree at which we finish (i.e. the leaf nodes)
- We can thus read of the time complexity
  - ⋆ worst case time: depth of the deepest of leaf
  - ⋆ best case time: depth of the shallowest of leaf
  - ⋆ average case time: average depth of leaves
- Different sort strategies will have different decision trees
- Decision trees are usually far too large to write out ☹



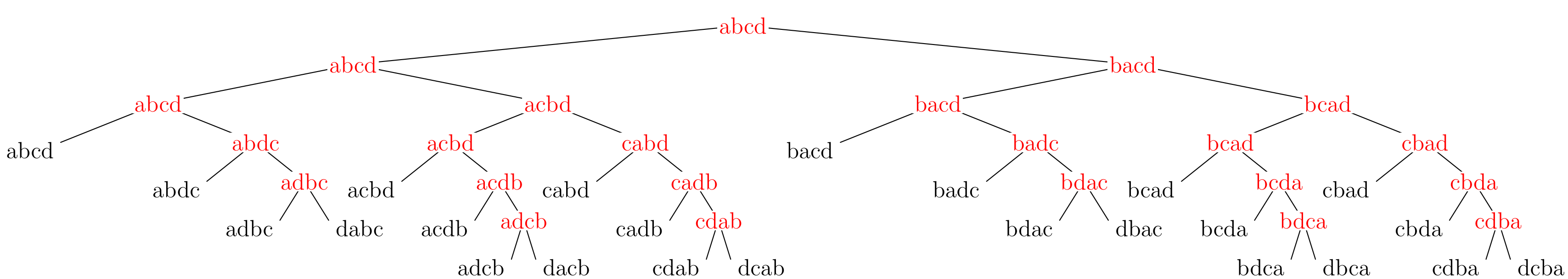
Decision Trees and Time Complexity

- The time taken to complete the task is the depth of the tree at which we finish (i.e. the leaf nodes)

- We can thus read of the time complexity
  - ⋆ worst case time: depth of the deepest of leaf
  - ⋆ best case time: depth of the shallowest of leaf
  - ⋆ average case time: average depth of leaves

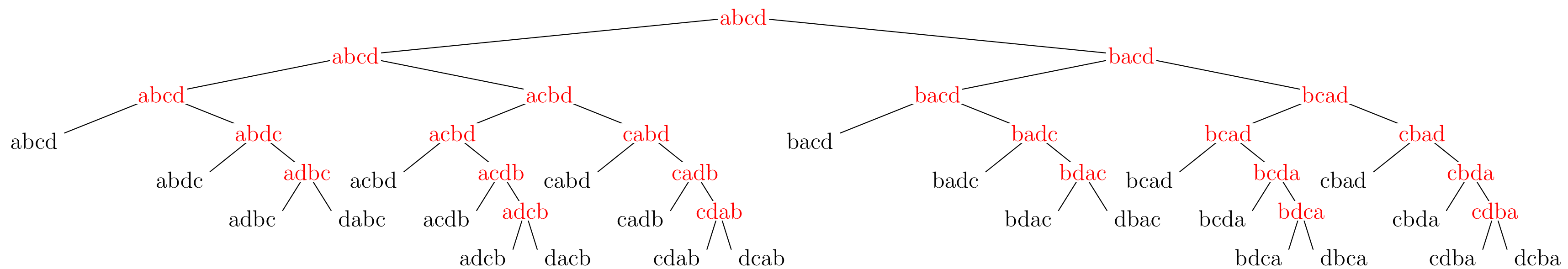
- Different sort strategies will have different decision trees

- Decision trees are usually far too large to write out ☹



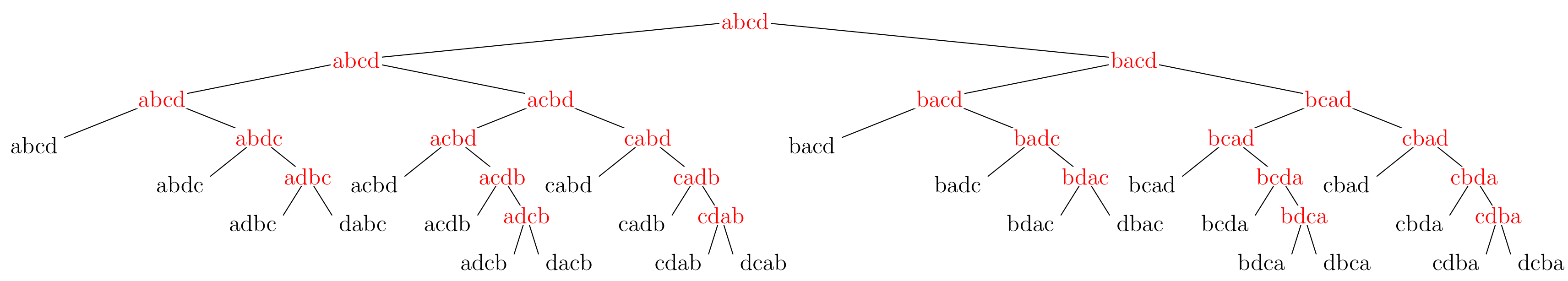
Decision Trees and Time Complexity

- The time taken to complete the task is the depth of the tree at which we finish (i.e. the leaf nodes)
- We can thus read of the time complexity
  - ⋆ worst case time: depth of the deepest of leaf
  - ⋆ best case time: depth of the shallowest of leaf
  - ⋆ average case time: average depth of leaves
- Different sort strategies will have different decision trees
- Decision trees are usually far too large to write out ☹



Decision Trees and Time Complexity

- The time taken to complete the task is the depth of the tree at which we finish (i.e. the leaf nodes)
- We can thus read of the time complexity
  - ⋆ worst case time: depth of the deepest of leaf
  - ⋆ best case time: depth of the shallowest of leaf
  - ⋆ average case time: average depth of leaves
- Different sort strategies will have different decision trees
- Decision trees are usually far too large to write out ☹



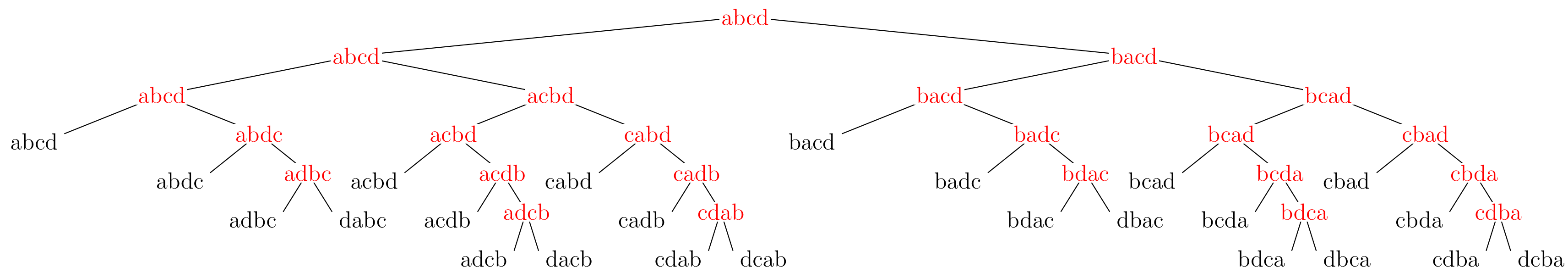
Decision Trees and Time Complexity

- The time taken to complete the task is the depth of the tree at which we finish (i.e. the leaf nodes)

- We can thus read of the time complexity
  - ⋆ worst case time: depth of the deepest of leaf
  - ⋆ best case time: depth of the shallowest of leaf
  - ⋆ average case time: average depth of leaves

- Different sort strategies will have different decision trees

- Decision trees are usually far too large to write out ☹



Requirements of Correct Sort

- Any sort based on binary comparisons must have a leaf of the tree for every possible way of sorting the list

- The array $[a, b, c]$ must be arranged differently for all combinations

$$[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]$$

- That is they must go through a different path of the decision tree

- If not sort won't work

Requirements of Correct Sort

- Any sort based on binary comparisons must have a leaf of the tree for every possible way of sorting the list

- The array $[a, b, c]$ must be arranged differently for all combinations

$$[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]$$

- That is they must go through a different path of the decision tree

- If not sort won't work

Requirements of Correct Sort

- Any sort based on binary comparisons must have a leaf of the tree for every possible way of sorting the list

- The array $[a, b, c]$ must be arranged differently for all combinations

$$[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]$$

- That is they must go through a different path of the decision tree

- If not sort won't work

Requirements of Correct Sort

- Any sort based on binary comparisons must have a leaf of the tree for every possible way of sorting the list
- The array $[a, b, c]$ must be arranged differently for all combinations

$$[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]$$

- That is they must go through a different path of the decision tree
- If not sort won't work

Minimum Number of Leaves

- There must be, at least, one leaf node of the decision tree for each possible permutation of the list
- How many permutations are there of a list of size n?
- Start with a sequence $(a_1, a_2, \ldots, a_n)$
- To create a new permutation we can choose any member of the list as the first element
- We can choose any of the remaining $n-1$ elements of the list as the second element of the permutation
- The total number of permutation is $n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1 = n!$

Minimum Number of Leaves

- There must be, at least, one leaf node of the decision tree for each possible permutation of the list
- How many permutations are there of a list of size n?
- Start with a sequence $(a_1, a_2, \ldots, a_n)$
- To create a new permutation we can choose any member of the list as the first element
- We can choose any of the remaining $n-1$ elements of the list as the second element of the permutation
- The total number of permutation is $n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1 = n!$

Minimum Number of Leaves

- There must be, at least, one leaf node of the decision tree for each possible permutation of the list
- How many permutations are there of a list of size n?
- Start with a sequence $(a_1, a_2, \ldots, a_n)$
- To create a new permutation we can choose any member of the list as the first element
- We can choose any of the remaining $n-1$ elements of the list as the second element of the permutation
- The total number of permutation is $n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1 = n!$

Minimum Number of Leaves

- There must be, at least, one leaf node of the decision tree for each possible permutation of the list
- How many permutations are there of a list of size n?
- Start with a sequence $(a_1, a_2, \ldots, a_n)$
- To create a new permutation we can choose any member of the list as the first element
- We can choose any of the remaining $n-1$ elements of the list as the second element of the permutation
- The total number of permutation is $n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1 = n!$

Minimum Number of Leaves

- There must be, at least, one leaf node of the decision tree for each possible permutation of the list
- How many permutations are there of a list of size n?
- Start with a sequence $(a_1, a_2, \ldots, a_n)$
- To create a new permutation we can choose any member of the list as the first element
- We can choose any of the remaining $n-1$ elements of the list as the second element of the permutation
- The total number of permutation is $n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1 = n!$

Minimum Number of Leaves

- There must be, at least, one leaf node of the decision tree for each possible permutation of the list
- How many permutations are there of a list of size n?
- Start with a sequence $(a_1, a_2, \ldots, a_n)$
- To create a new permutation we can choose any member of the list as the first element
- We can choose any of the remaining $n-1$ elements of the list as the second element of the permutation
- The total number of permutation is $n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1 = n!$

Lower Bound Time Complexity for Sorting

- Any sort algorithm using binary comparisons must have a decision tree with at least $n!$ leaf nodes
- This will be a binary tree with some depth d
- The number of leaves at depth d is 2^d
- Thus the smallest depth tree must have a depth d such that $2^d \geq n!$
- That is, the depth of the decision tree satisfies $d \geq \log_2(n!)$
- But this is the number of comparisons needed in our sort
- We are left with a lower bound on the time complexity of $\log_2(n!)$

Lower Bound Time Complexity for Sorting

- Any sort algorithm using binary comparisons must have a decision tree with at least $n!$ leaf nodes
- This will be a binary tree with some depth d
- The number of leaves at depth d is 2^d
- Thus the smallest depth tree must have a depth d such that $2^d \geq n!$
- That is, the depth of the decision tree satisfies $d \geq \log_2(n!)$
- But this is the number of comparisons needed in our sort
- We are left with a lower bound on the time complexity of $\log_2(n!)$

Lower Bound Time Complexity for Sorting

- Any sort algorithm using binary comparisons must have a decision tree with at least $n!$ leaf nodes
- This will be a binary tree with some depth d
- The number of leaves at depth d is 2^d
- Thus the smallest depth tree must have a depth d such that $2^d \geq n!$
- That is, the depth of the decision tree satisfies $d \geq \log_2(n!)$
- But this is the number of comparisons needed in our sort
- We are left with a lower bound on the time complexity of $\log_2(n!)$

Lower Bound Time Complexity for Sorting

- Any sort algorithm using binary comparisons must have a decision tree with at least $n!$ leaf nodes
- This will be a binary tree with some depth d
- The number of leaves at depth d is 2^d
- Thus the smallest depth tree must have a depth d such that $2^d \geq n!$
- That is, the depth of the decision tree satisfies $d \geq \log_2(n!)$
- But this is the number of comparisons needed in our sort
- We are left with a lower bound on the time complexity of $\log_2(n!)$

Lower Bound Time Complexity for Sorting

- Any sort algorithm using binary comparisons must have a decision tree with at least $n!$ leaf nodes
- This will be a binary tree with some depth d
- The number of leaves at depth d is 2^d
- Thus the smallest depth tree must have a depth d such that $2^d \geq n!$
- That is, the depth of the decision tree satisfies $d \geq \log_2(n!)$
- But this is the number of comparisons needed in our sort
- We are left with a lower bound on the time complexity of $\log_2(n!)$

Lower Bound Time Complexity for Sorting

- Any sort algorithm using binary comparisons must have a decision tree with at least $n!$ leaf nodes
- This will be a binary tree with some depth d
- The number of leaves at depth d is 2^d
- Thus the smallest depth tree must have a depth d such that $2^d \geq n!$
- That is, the depth of the decision tree satisfies $d \geq \log_2(n!)$
- But this is the number of comparisons needed in our sort
- We are left with a lower bound on the time complexity of $\log_2(n!)$

Lower Bound Time Complexity for Sorting

- Any sort algorithm using binary comparisons must have a decision tree with at least $n!$ leaf nodes
- This will be a binary tree with some depth d
- The number of leaves at depth d is 2^d
- Thus the smallest depth tree must have a depth d such that $2^d \geq n!$
- That is, the depth of the decision tree satisfies $d \geq \log_2(n!)$
- But this is the number of comparisons needed in our sort
- We are left with a lower bound on the time complexity of $\log_2(n!)$

How Big is $\log_2(n!)$

- We showed in the second lecture that

$$\left(\frac{n}{2}\right)^{n/2} < n! < n^n$$

- It is not too difficult to show that asymptotically (i.e. as $n \to \infty$) that $n!$ approaches $\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$—this is known as **Stirling's approximation**
- Thus

$$\begin{aligned}\log_2(n!) &\approx n\log_2(n) - n\log_2(e) + \frac{\log_2(n)}{2} + \frac{\log_2(2\pi)}{2} \\ &= \Theta(n\log_2(n))\end{aligned}$$

How Big is $\log_2(n!)$

- We showed in the second lecture that

$$\left(\frac{n}{2}\right)^{n/2} < n! < n^n$$

- It is not too difficult to show that asymptotically (i.e. as $n \to \infty$) that $n!$ approaches $\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$—this is known as **Stirling's approximation**

- Thus

$$\begin{aligned}\log_2(n!) &\approx n\log_2(n) - n\log_2(e) + \frac{\log_2(n)}{2} + \frac{\log_2(2\pi)}{2} \\ &= \Theta(n\log_2(n))\end{aligned}$$

How Big is $\log_2(n!)$

- We showed in the second lecture that

$$\left(\frac{n}{2}\right)^{n/2} < n! < n^n$$

- It is not too difficult to show that asymptotically (i.e. as $n \to \infty$) that $n!$ approaches $\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$—this is known as **Stirling's approximation**

- Thus

$$\begin{aligned}\log_2(n!) &\approx n\log_2(n) - n\log_2(e) + \frac{\log_2(n)}{2} + \frac{\log_2(2\pi)}{2} \\ &= \Theta(n\log_2(n))\end{aligned}$$

How Big is $\log_2(n!)$

- We showed in the second lecture that

$$\left(\frac{n}{2}\right)^{n/2} < n! < n^n$$

- It is not too difficult to show that asymptotically (i.e. as $n \to \infty$) that $n!$ approaches $\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$—this is known as **Stirling's approximation**
- Thus

$$\begin{aligned} \log_2(n!) &\approx n\log_2(n) - n\log_2(e) + \frac{\log_2(n)}{2} + \frac{\log_2(2\pi)}{2} \\ &= \Theta(n\log_2(n)) \end{aligned}$$

How Big is $\log_2(n!)$

- We showed in the second lecture that

$$\left(\frac{n}{2}\right)^{n/2} < n! < n^n$$

- It is not too difficult to show that asymptotically (i.e. as $n \to \infty$) that $n!$ approaches $\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$—this is known as **Stirling's approximation**
- Thus

$$\begin{aligned}\log_2(n!) &\approx n\log_2(n) - n\log_2(e) + \frac{\log_2(n)}{2} + \frac{\log_2(2\pi)}{2} \\ &= \Theta(n\log_2(n))\end{aligned}$$

Complexity of Sorting

- We therefore have a lower bound on the time complexity of $\Omega(n \log(n))$
- This is true for any sort using binary comparisons
- We will see in the next lecture there exists algorithms with time complexity $O(n \log(n))$
- This means our lower bound is tight—i.e. it is the true cost of the best algorithm
- Having a lower bound we know we are not going to obtain a substantially faster algorithm

Complexity of Sorting

- We therefore have a lower bound on the time complexity of $\Omega(n \log(n))$

- This is true for any sort using binary comparisons

- We will see in the next lecture there exists algorithms with time complexity $O(n \log(n))$

- This means our lower bound is tight—i.e. it is the true cost of the best algorithm

- Having a lower bound we know we are not going to obtain a substantially faster algorithm

Complexity of Sorting

- We therefore have a lower bound on the time complexity of $\Omega(n\log(n))$
- This is true for any sort using binary comparisons
- We will see in the next lecture there exists algorithms with time complexity $O(n\log(n))$
- This means our lower bound is tight—i.e. it is the true cost of the best algorithm
- Having a lower bound we know we are not going to obtain a substantially faster algorithm

Complexity of Sorting

- We therefore have a lower bound on the time complexity of $\Omega(n \log(n))$
- This is true for any sort using binary comparisons
- We will see in the next lecture there exists algorithms with time complexity $O(n \log(n))$
- This means our lower bound is tight—i.e. it is the true cost of the best algorithm
- Having a lower bound we know we are not going to obtain a substantially faster algorithm

Complexity of Sorting

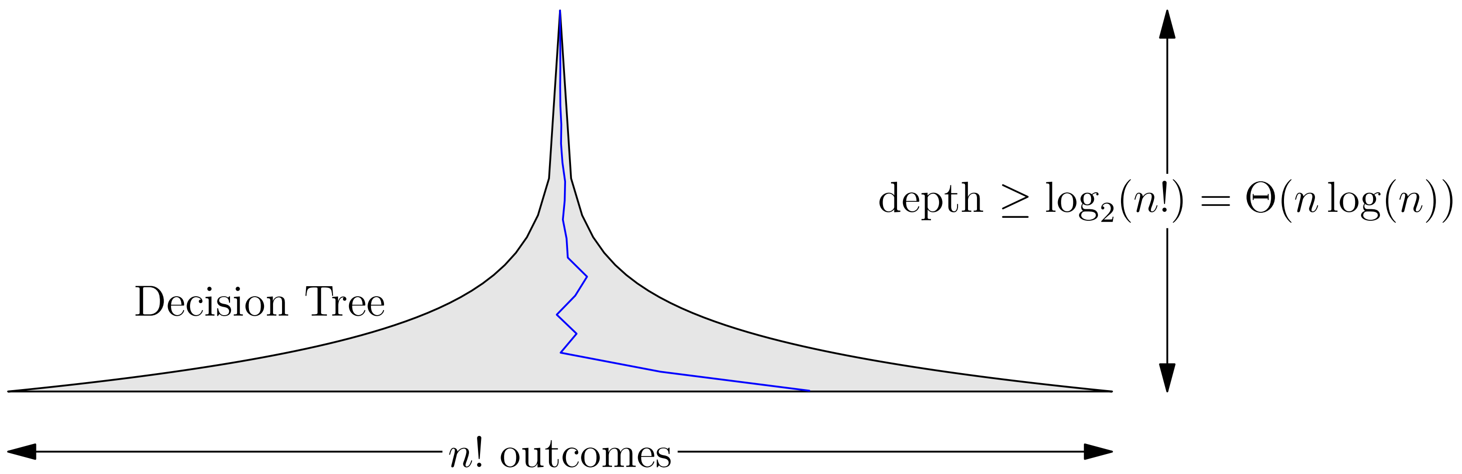
- We therefore have a lower bound on the time complexity of $\Omega(n\log(n))$
- This is true for any sort using binary comparisons
- We will see in the next lecture there exists algorithms with time complexity $O(n\log(n))$
- This means our lower bound is tight—i.e. it is the true cost of the best algorithm
- Having a lower bound we know we are not going to obtain a substantially faster algorithm

Complexity of Sorting

- We therefore have a lower bound on the time complexity of $\Omega(n \log(n))$
- This is true for any sort using binary comparisons
- We will see in the next lecture there exists algorithms with time complexity $O(n \log(n))$
- This means our lower bound is tight—i.e. it is the true cost of the best algorithm
- Having a lower bound we know we are not going to obtain a substantially faster algorithm

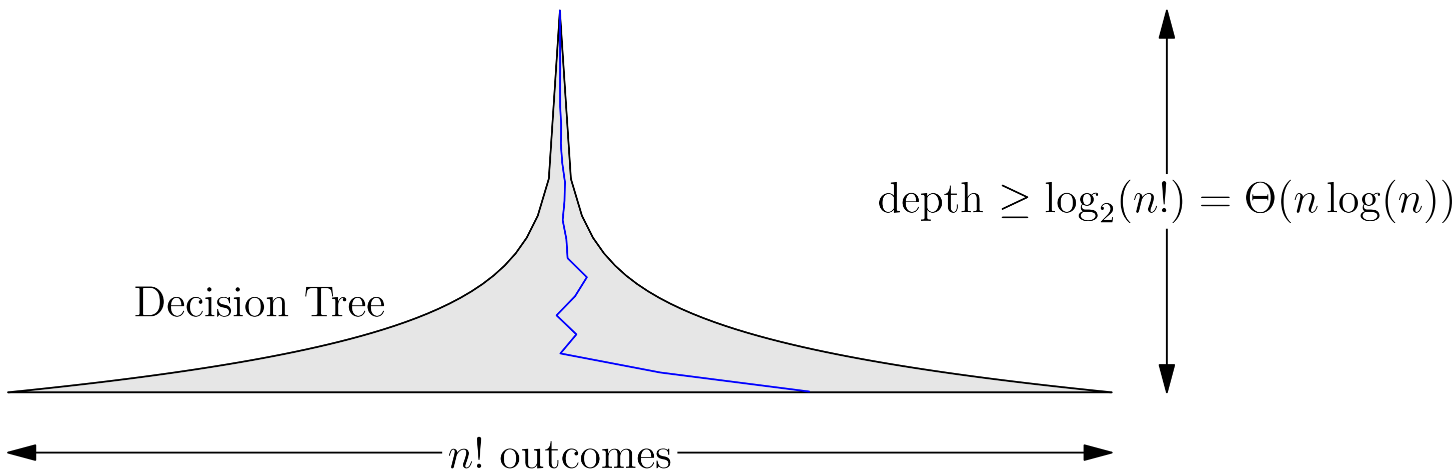
Lessons

- Analysis of algorithms is hard
- Analysis is important: without it we don't know if we have a good algorithm or whether we should try to find a more efficient one
- Lower bounds are particularly important



Lessons

- Analysis of algorithms is hard
- Analysis is important: without it we don't know if we have a good algorithm or whether we should try to find a more efficient one
- Lower bounds are particularly important



Lessons

- Analysis of algorithms is hard
- Analysis is important: without it we don't know if we have a good algorithm or whether we should try to find a more efficient one
- Lower bounds are particularly important