# Analysing Algorithms (Part 2 - Examining Algorithms)

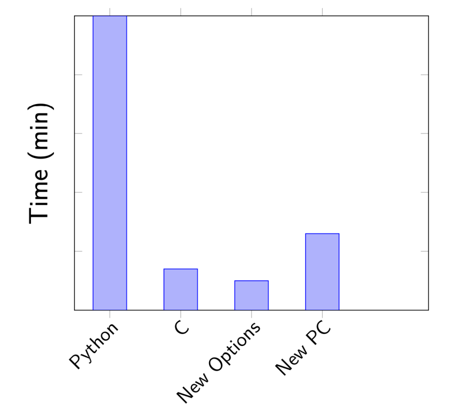
Daniel Lawson — University of Bristol

Lecture 08.1.2 (v1.0.2)

# Signposting

- Analysing Algorithms is split into three parts:
  - Part 1: Motivation and Algorithmic Complexity
  - Part 2: Examining algorithms
  - Part 3: Turing Machines and Complexity Classes
- This is Part 2
- Thanks to Turing Fellow and Computer Scientist Dan Martin for Tikz pictures and expertise

# Runtime vs Complexity - motivation




- Consider our algorithm run on data $D_1$:
  - In different programming languages, compile arguments and hardware
- What can be said in general?

# Algorithm Example (1)

- What is the complexity of the following algorithm?

**procedure** Example($a, b, n$)
  $i \leftarrow 1$
  **while** $i \leq n$ **do**
    $a \leftarrow f_1(b, n)$
    $b \leftarrow f_2(a, n)$
    $i \leftarrow i + 1$
  **end while**
  **return** $b$
**end procedure**

- $f_i(a, n)$ has runtime $T_i(n)$
- Inside loop is $\mathcal{O}(T_1(n) + T_2(n))$
- Total $\mathcal{O}[n(T_1(n) + T_2(n))]$

# Algorithm Example (2)

- Compare to the following algorithm?

**procedure** Example($a, b, n$)
  $i \leftarrow 1$
  **while** $i \leq n$ **do**
    $a \leftarrow f_1(b, n)$
    $b \leftarrow f_2(a, n)$
    $i \leftarrow 2 \cdot i$
  **end while**
  **return** $b$
**end procedure**

- Inside loop is $\mathcal{O}(T_1(n) + T_2(n))$
- Total $\mathcal{O}[\log(n)(T_1(n) + T_2(n))]$

# Sorting examples

- We have some data: $1, 4, 6, 2, 3, 7, 5, \cdots$
- We want to sort the data into ascending order: $1, 2, 3, 4, 5, 6, 7, \cdots$
- What is the best[1] algorithm?
  - **Insertion sort** is $\Theta(n^2)$, but operates in-place.
  - **Merge sort** is $\Theta(n \log(n))$, but memory requirements grow with data size.
  - **Heap sort** is $\Theta(n \log(n))$ and sorts in place.
  - **Quick sort** is $\Theta(n^2)$, but $\Theta(n \log(n))$ expected time, and is often fastest in practice.
  - **Counting sort** allows array indices to be sorted in $\Theta(n)$ by exploiting knowledge that all integers are present.
  - **Bucket sort** is $\Theta(n^2)$, though $\Theta(n)$ average case (if data are Uniform!)

[1] Cormen et al 2010 Introduction to Algorithms

# Quicksort: a Recursion Example

**procedure** QuickSort($A$)
  **if** $len(A) == 1$ **then**
    **return** $A$
  **else**
    $x \leftarrow A$
    $A_l \leftarrow \{a \in A : a < x\}$
    $A_h \leftarrow \{a \in A : a > x\}$
    $A_x \leftarrow \{a \in A : a = x\}$
    $S_l \leftarrow$ QuickSort($A_l$)
    $S_h \leftarrow$ QuickSort($A_h$)
    **return** $[S_l, A_x, S_h]$
  **end if**
**end procedure**

# Quicksort: a Recursion Example

**procedure** QuickSort($A$)
  **if** $len(A) == 1$ **then**
    **return** $A$
  **else**
    $x \leftarrow A$
    $A_l \leftarrow \{a \in A : a < x\}$
    $A_h \leftarrow \{a \in A : a > x\}$
    $A_x \leftarrow \{a \in A : a = x\}$
    $S_l \leftarrow$ QuickSort$(A_l)$
    $S_h \leftarrow$ QuickSort$(A_h)$
    **return** $[S_l, A_x, S_h]$
  **end if**
**end procedure**

What if we can choose the **median element** of $A$?

# Quicksort: a Recursion Example

```
procedure QuickSort(A)
    if len(A) == 1 then
        return A
    else
        x ← A
        A_l ← {a ∈ A : a < x}
        A_h ← {a ∈ A : a > x}
        A_x ← {a ∈ A : a = x}
        S_l ← QuickSort(A_l)
        S_h ← QuickSort(A_h)
        return [S_l, A_x, S_h]
    end if
end procedure
```

What if we can choose the **median element** of $A$?

$T(n)$

# Quicksort: a Recursion Example

**procedure** QuickSort($A$)
  **if** $len(A) == 1$ **then**
    **return** $A$
  **else**
    $x \leftarrow A$
    $A_l \leftarrow \{a \in A : a < x\}$
    $A_h \leftarrow \{a \in A : a > x\}$
    $A_x \leftarrow \{a \in A : a = x\}$
    $S_l \leftarrow$ QuickSort($A_l$)
    $S_h \leftarrow$ QuickSort($A_h$)
    **return** $[S_l, A_x, S_h]$
  **end if**
**end procedure**

What if we can choose the **median element** of $A$?

$$T(n)$$

$$= 2T(\frac{n}{2}) + n$$

# Quicksort: a Recursion Example

**procedure** QuickSort($A$)
  **if** $len(A) == 1$ **then**
    **return** $A$
  **else**
    $x \leftarrow A$
    $A_l \leftarrow \{a \in A : a < x\}$
    $A_h \leftarrow \{a \in A : a > x\}$
    $A_x \leftarrow \{a \in A : a = x\}$
    $S_l \leftarrow$ QuickSort($A_l$)
    $S_h \leftarrow$ QuickSort($A_h$)
    **return** $[S_l, A_x, S_h]$
  **end if**
**end procedure**

What if we can choose the **median element** of $A$?

$$
\begin{aligned}
T(n) & \\
&= 2T(\frac{n}{2}) + n \\
&= 2(2T(\frac{n}{4}) + \frac{n}{2}) + n
\end{aligned}
$$

# Quicksort: a Recursion Example

**procedure** QuickSort($A$)
  **if** $len(A) == 1$ **then**
    **return** $A$
  **else**
    $x \leftarrow A$
    $A_l \leftarrow \{a \in A : a < x\}$
    $A_h \leftarrow \{a \in A : a > x\}$
    $A_x \leftarrow \{a \in A : a = x\}$
    $S_l \leftarrow$ QuickSort$(A_l)$
    $S_h \leftarrow$ QuickSort$(A_h)$
    **return** $[S_l, A_x, S_h]$
  **end if**
**end procedure**

What if we can choose the **median element** of $A$?

$$
\begin{aligned}
T(n) & \\
&= 2T(\frac{n}{2}) + n \\
&= 2(2T(\frac{n}{4}) + \frac{n}{2}) + n \\
&= \dots
\end{aligned}
$$

# Quicksort: a Recursion Example

**procedure** QuickSort($A$)
  **if** $len(A) == 1$ **then**
    **return** $A$
  **else**
    $x \leftarrow A$
    $A_l \leftarrow \{a \in A : a < x\}$
    $A_h \leftarrow \{a \in A : a > x\}$
    $A_x \leftarrow \{a \in A : a = x\}$
    $S_l \leftarrow$ QuickSort$(A_l)$
    $S_h \leftarrow$ QuickSort$(A_h)$
    **return** $[S_l, A_x, S_h]$
  **end if**
**end procedure**

What if we can choose the **median element** of $A$?

$$
\begin{aligned}
T(n) & \\
&= 2T(\frac{n}{2}) + n \\
&= 2(2T(\frac{n}{4}) + \frac{n}{2}) + n \\
&= \dots \\
&= 2^{\log n} T(1) + \sum_{i=1}^{\log n} n
\end{aligned}
$$

# Quicksort: a Recursion Example

```
procedure QuickSort(A)
    if len(A) == 1 then
        return A
    else
        x ← A
        A_l ← {a ∈ A : a < x}
        A_h ← {a ∈ A : a > x}
        A_x ← {a ∈ A : a = x}
        S_l ← QuickSort(A_l)
        S_h ← QuickSort(A_h)
        return [S_l, A_x, S_h]
    end if
end procedure
```

What if we can choose the **median element** of $A$?

$$
\begin{aligned}
T(n) & \\
&= 2T(\frac{n}{2}) + n \\
&= 2(2T(\frac{n}{4}) + \frac{n}{2}) + n \\
&= \dots \\
&= 2^{\log n} T(1) + \sum_{i=1}^{\log n} n \\
&= \Theta(n \log n)
\end{aligned}
$$

# Quicksort: a Recursion Example

**procedure** QuickSort($A$)
  **if** $len(A) == 1$ **then**
    **return** $A$
  **else**
    $x \leftarrow A$
    $A_l \leftarrow \{a \in A : a < x\}$
    $A_h \leftarrow \{a \in A : a > x\}$
    $A_x \leftarrow \{a \in A : a = x\}$
    $S_l \leftarrow \text{QuickSort}(A_l)$
    $S_h \leftarrow \text{QuickSort}(A_h)$
    **return** $[S_l, A_x, S_h]$
  **end if**
**end procedure**

What if we always choose the **largest element** of $A$?

# Quicksort: a Recursion Example

```
procedure QuickSort(A)
    if len(A) == 1 then
        return A
    else
        x ← A
        A_l ← {a ∈ A : a < x}
        A_h ← {a ∈ A : a > x}
        A_x ← {a ∈ A : a = x}
        S_l ← QuickSort(A_l)
        S_h ← QuickSort(A_h)
        return [S_l, A_x, S_h]
    end if
end procedure
```

What if we always choose the **largest element** of $A$?

$T(n)$

# Quicksort: a Recursion Example

**procedure** QuickSort($A$)
  **if** $len(A) == 1$ **then**
    **return** $A$
  **else**
    $x \leftarrow A$
    $A_l \leftarrow \{a \in A : a < x\}$
    $A_h \leftarrow \{a \in A : a > x\}$
    $A_x \leftarrow \{a \in A : a = x\}$
    $S_l \leftarrow$ QuickSort$(A_l)$
    $S_h \leftarrow$ QuickSort$(A_h)$
    **return** $[S_l, A_x, S_h]$
  **end if**
**end procedure**

What if we always choose the **largest element** of $A$?

$$T(n) = T(n-1) + n$$

# Quicksort: a Recursion Example

**procedure** QuickSort($A$)
  **if** $len(A) == 1$ **then**
    **return** $A$
  **else**
    $x \leftarrow A$
    $A_l \leftarrow \{a \in A : a < x\}$
    $A_h \leftarrow \{a \in A : a > x\}$
    $A_x \leftarrow \{a \in A : a = x\}$
    $S_l \leftarrow$ QuickSort($A_l$)
    $S_h \leftarrow$ QuickSort($A_h$)
    **return** $[S_l, A_x, S_h]$
  **end if**
**end procedure**

What if we always choose the **largest element** of $A$?

$$
\begin{aligned}
T(n) \\
&= T(n-1) + n \\
&= (T(n-2) + n) + n
\end{aligned}
$$

# Quicksort: a Recursion Example

**procedure** QuickSort($A$)
  **if** $len(A) == 1$ **then**
    **return** $A$
  **else**
    $x \leftarrow A$
    $A_l \leftarrow \{a \in A : a < x\}$
    $A_h \leftarrow \{a \in A : a > x\}$
    $A_x \leftarrow \{a \in A : a = x\}$
    $S_l \leftarrow$ QuickSort$(A_l)$
    $S_h \leftarrow$ QuickSort$(A_h)$
    **return** $[S_l, A_x, S_h]$
  **end if**
**end procedure**

What if we always choose the **largest element** of $A$?

$$T(n)$$

$$\begin{aligned} &= T(n-1) + n \\ &= (T(n-2) + n) + n \\ &= \ldots \end{aligned}$$

# Quicksort: a Recursion Example

**procedure** QuickSort($A$)
  **if** $len(A) == 1$ **then**
    **return** $A$
  **else**
    $x \leftarrow A$
    $A_l \leftarrow \{a \in A : a < x\}$
    $A_h \leftarrow \{a \in A : a > x\}$
    $A_x \leftarrow \{a \in A : a = x\}$
    $S_l \leftarrow$ QuickSort($A_l$)
    $S_h \leftarrow$ QuickSort($A_h$)
    **return** $[S_l, A_x, S_h]$
  **end if**
**end procedure**

What if we always choose the **largest element** of $A$?

$T(n)$

$$\begin{aligned} &= T(n-1) + n \\ &= (T(n-2) + n) + n \\ &= \dots \\ &= T(1) + \sum_{i=1}^{n} i \end{aligned}$$

# Quicksort: a Recursion Example

**procedure** QuickSort($A$)
  **if** $len(A) == 1$ **then**
    **return** $A$
  **else**
    $x \leftarrow A$
    $A_l \leftarrow \{a \in A : a < x\}$
    $A_h \leftarrow \{a \in A : a > x\}$
    $A_x \leftarrow \{a \in A : a = x\}$
    $S_l \leftarrow$ QuickSort($A_l$)
    $S_h \leftarrow$ QuickSort($A_h$)
    **return** $[S_l, A_x, S_h]$
  **end if**
**end procedure**

What if we always choose the **largest element** of $A$?

$$
\begin{aligned}
T(n) & \\
&= T(n-1) + n \\
&= (T(n-2) + n) + n \\
&= \dots \\
&= T(1) + \sum_{i=1}^{n} i \\
&= n(n-1)/2 = \Theta(n^2)
\end{aligned}
$$

# Other types of complexity

- Complexity questions are primarily asked about:
  - **Computation** (time)
  - **Space** (memory)
  - **Communication** (data transfer)
- They are all studied analogously - it is the unit of counting that changes
- Despite that, the theory is quite different

# Space complexity

- Simply the amount of memory that an algorithm needs
- You can calculate it simply by adding the memory allocations
- Space required is **additional** to the input, which is **not** counted - this can conceptually not be stored at all, as in e.g. streaming algorithms
- Formally defined in terms of the Turing Machine (8.1.3)
- It can often be traded for time complexity, e.g. by storing intermediate results vs revisiting the calculation
- For a Data Scientist, this trade off is critical!
- We use the same notation

# Space complexity example (1)

- **Problem**: Find $x, y$ in $X$ s.t. $x + y = T$ (known to exist)
- Solution 1:

```
import heapq
heapq.heapsort(X)
i=0;j=n-1;
while(X[i]+X[j]!=T):
  if X[i]+X[j]<T:
      i=i+1
  else:
      j=j-1
```

- Heapsort has $\mathcal{O}(1)$ space complexity
- Therefore the whole algorithm is $\mathcal{O}(1)$ in space
- And time complexity $\mathcal{O}(n\log(n) + n) = \mathcal{O}(n\log(n))$

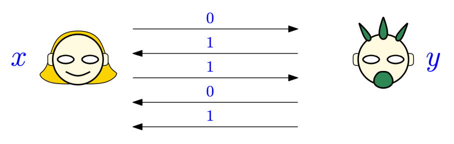
# Space complexity example (2)

- Find $x, y$ in $X$ s.t. $x + y = T$ (known to exist)
- Solution 2:

```
D={}
for i in range(len(X)):
    D[T-X[i]]=i
for x in X:
    y=T-x
    if y in D:
        return X[D[y]],x
```

- This is $\mathcal{O}(n)$ in space
- Hash lookups are $\mathcal{O}(1)$ **average case** complexity ($\mathcal{O}(n)$ worst case - which does not apply here!)
- So this algorithm is $\mathcal{O}(n)$ in time too

# Communication Complexity




- Alice knows $x \in X$, Bob knows $y \in Y$
- Together they want to compute $f(x, y)$ where $f \in X \times Y \to Z$
- Via a pre-arranged **protocol** $P$ determining what they send
- The **communication cost** is the number of bits sent [2]

[2] Raznorov 2015 Communication Complexity Lecture

## Communication Complexity

- The **Overall cost** of $P$ is $\mathrm{C}(P) = \max_{x,y}[P(x,y)]$, i.e. the maximum possible cost for all data
- The **Communication complexity** of $f$ is $\mathrm{C}(f) = \min_{P \in \mathcal{P}}(C[P(x,y)])$
- It is the minimum number of bits needed to compute $f(x,y)$ for any $x, y$
- Communication Complexity Theory describes $\mathrm{C}(f)$, typically by finding **bounds** (upper and lower) for a given $f$
  - Again typically as a function of the size of $x$ and $y$, and always for some well defined spaces $X$ and $Y$.
- Note that there is a trivial bound of $n+1$ for transferring all the data! (and then the answer back)

# Communication Complexity Examples

- $f(x, y) = \text{Parity}([x, y])$
  - Parity=$mod_2(\sum_{i=1}^{n} x_i)$
  - $\text{C}(f(x, y)) = 2$ because Alice calculates the Parity of $x$, Bob the Parity of $y$, and they each communicate their own parity
- $f(x, y) = \text{Equality}(x, y)$
  - i.e. $1$ if $x_i = y_i \quad \forall i$, and $0$ otherwise
  - $\text{C}(f(x, y)) = n$ because every bit must be compared
- Typically approximate algorithms allow dramatically lower complexity
  - All the interesting theory is in this space

# What is communication complexity theory good for?

- There are lots of immediate applications
  - Optimisation of computer networks
  - **Parallel algorithms**: communication between multiple cores on a CPU, or nodes of a cluster
  - And basically anything involving the internet!
  - Especially differential privacy (Block 12)
- There are many more less immediate applications
  - Particularly as a tool for algorithm and data structure lower bounds

# Reflection

- What are the main subjects of complexity theory, and in which ways are they similar?
- By the end of the course, students should be able to:
  - Define three subjects of complexity theory
  - Apply each to simple algorithms, including compound algorithms
  - Reason about their value at a high level

# Signposting

- Next up: Part 3: Turing Machines
- **References**:
  - Cormen et al 2010 Introduction to Algorithms
  - Toniann Pitassi Lecture on Communication Complexity: Applications and New Directions
  - Raznorov 2015 Communication Complexity Lecture
  - Arora and Barak Computational Complexity: A Modern Approach
    - One of few places to give space complexity much time (its always the poor cousin)