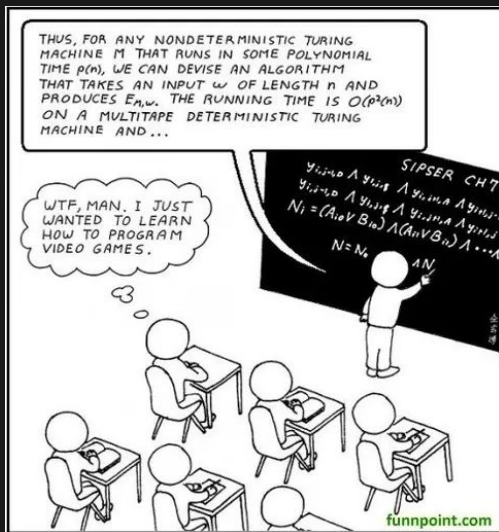


Analysing Algorithms

Daniel Lawson — University of Bristol

Lecture 08.1 (v4.0.0)

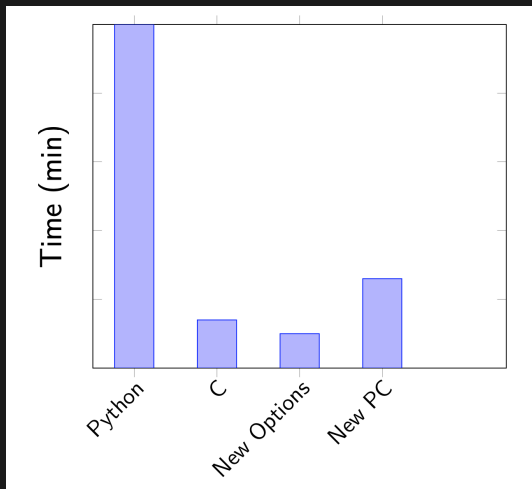
Shall we learn about Turing Machines?



Questions

- ▶ Can we prove that one algorithm is faster than another?
- ▶ What does *faster* mean?
- ▶ How do we describe random algorithms?
- ▶ What is the best algorithm? What is “best”?

Runtime - motivation



- ▶ Consider our algorithm run on data D_1 :
- ▶ Different programming languages/compiler/hardware
- ▶ How do we predict its runtime elsewhere?

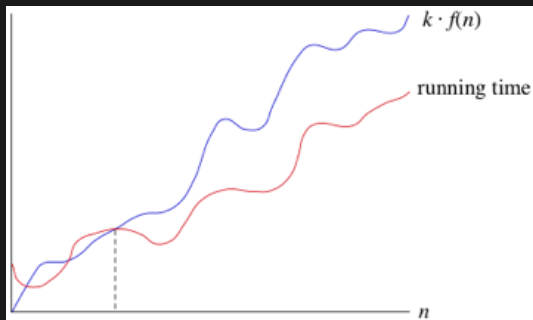
Why study algorithms?

- ▶ Algorithms underlie every machine-learning method.
- ▶ Theoretical statements about algorithms can be made, including:
 - ▶ How long does an algorithm take to run?
 - ▶ What guarantees can be made about the answer an algorithm returns?
- ▶ In some cases, **carefully chosen algorithms** can achieve either perfect or usefully good performance at a vanishing fraction of the run time of a naive implementation.
- ▶ This can lead to a **solution on a single machine** that is superior to that of a massively parallel implementation using distributed computing.

Algorithmic concerns

- ▶ We typically care about:
 - ▶ How long does the algorithm run for? Under which circumstances?
 - ▶ How do they trade off **runtime** and **memory requirement**?
- ▶ Some special values include **in-place** methods (which have a constant memory requirement) and **streaming** methods which visit the data exactly once each (usually with a constant-sized memory).
- ▶ Proofs typically describe the scaling of these properties, but in practice the constants are very important!

Algorithmic complexity: Big O Notation

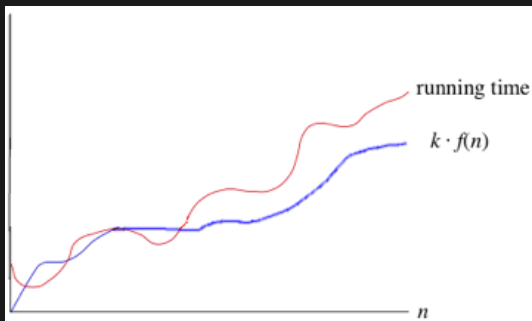


- ▶ $\mathcal{O}(n)$: An upper bound as a function of data size n
- ▶ $g(n) = \mathcal{O}(f(n))$:
 - ▶ $\exists n_0, k \in \mathbb{N}$ such that:
 - ▶ $\forall n \geq n_0$:
 - ▶ $g(n) \leq k \cdot f(n)$

Complexity examples - Big Oh

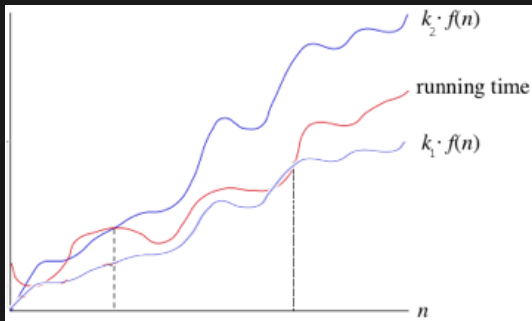
- ▶ Q: Is $4n \in \mathcal{O}(n^2)$?
 - ▶ Q: Is $n \in \mathcal{O}(n)$?
- ▶ Q: Is $2n^2 + n + 10 \in \mathcal{O}(n)$?
 - ▶ Q: Is $2n^2 + n + 10 \in \mathcal{O}(n^2)$?
- ▶ Q: Is $\log(n) \in \mathcal{O}(n^\epsilon)$?
 - ▶ A: ... for all $\epsilon > 0$.
- ▶ Extensions:
 - ▶ Let $g_1(n) \geq g_2(n)$ for all $n \geq n_0$.
 - ▶ Q: If $f_1(n) \in \mathcal{O}(g_1(n))$ and $f_2(n) \in \mathcal{O}(g_2(n))$ then is $f_1(n) \cdot f_2(n) \in \mathcal{O}(g_1(n))$?
 - ▶ Q: ... is $f_1(n) \cdot f_2(n) \in \mathcal{O}(g_1(n)g_2(n))$?
 - ▶ Q: ... is $f_1(n) + f_2(n) \in \mathcal{O}(g_1(n) \cdot g_2(n))$?
 - ▶ Q: ... is $f_1(n) + f_2(n) \in \mathcal{O}(g_1(n))$?

Algorithmic complexity: Big Omega Notation



- ▶ $\Omega(n)$: A lower bound a function of data size n
- ▶ $g(n) = \Omega(f(n))$:
 - ▶ $\exists n_0, k \in \mathbb{N}$ such that:
 - ▶ $\forall n \geq n_0$:
 - ▶ $g(n) \geq k \cdot f(n)$

Algorithmic complexity: Big Theta Notation



- ▶ $\Theta(n)$: A tight bound as a function of data size n
- ▶ $g(n) = \Theta(f(n))$:
 - ▶ $\exists n_0, k_1, k_2 \in \mathbb{N}$ such that:
 - ▶ $\forall n \geq n_0$:
 - ▶ $k_1 \cdot f(n) \leq g(n) \leq k_2 \cdot f(n)$
- ▶ i.e. the bound is strict.

Complexity examples

- ▶ Q: If $f(n) \in \mathcal{O}(g(n))$ then is $g(n) \in \Omega(f(n))$?
 - ▶ A: Yes, by definition.
- ▶ Q: What is Θ of $2n^2 + 3n + 1$?
 - ▶ A: $= 2n^2 + \Theta(n) =$
 - ▶ $\Theta(n^2)$
- ▶ Q: What is Θ of $3n^3 + \mathcal{O}(n^2)$?
 - ▶ A: $\Theta(n^3)$
- ▶ Q: What is Θ of $3n^3 + \Omega(n^2)$?
 - ▶ A: not enough information.
- ▶ Thoughts:
 - ▶ Q: What is $\mathcal{O}(f(n))$ when f can be run in $\mathcal{O}(n)$ if the input is sorted, and $\mathcal{O}(n^2)$ otherwise?

Algorithmic complexity: Probabilistic Analysis

- ▶ Sometimes we don't want the worst-case behaviour out of all possible inputs
- ▶ In these scenarios **average-case** run time is often reported
 - ▶ This is typically the average over the entire input space
 - ▶ This should make the statistician in you concerned!
- ▶ Randomized algorithms are also important
 - ▶ In these the answer may be random, and take a random amount of time, for a given input!
 - ▶ e.g. MCMC, etc
 - ▶ Again the **expected run time** is often reported
- ▶ We can discuss Θ , Ω and $\mathcal{O}$ of the expected runtime
- ▶ Clearly the distribution of the input data is important
- ▶ Some worst-case scenarios have “measure 0” (i.e. will never occur in practice)

Complexity and constants

- ▶ Consider the following functions:

```
import time
def constant_fun(n,k):
    time.sleep(k * k);
def linear_fun(n,k):
    for i in range(n):
        time.sleep(1);
```

- ▶ Clearly `linear_fun` is faster for $n < k^2$. We need to take into account k and whether it scales with n .
- ▶ In practice k is often truly a constant but can be any scale compared to n . The accounting therefore needs to retain it.
- ▶ Example: SVD is $\mathcal{O}(\min(mn^2, m^2n))$
- ▶ Complexity classes **only** describe **asymptotic behaviour** for large n

Divide and conquer

- ▶ One of the most popular strategies is **Divide and Conquer**, in which we make many sub-problems, each of which is solvable.
- ▶ This is typically valuable for parallelism
- ▶ It also makes sense to apply the algorithm **recursively**.
 - ▶ In which case we obtain expressions like:

$$T(n) = aT(n/k) + D(n) \quad \text{if} \quad n \geq c,$$

- ▶ and $T(n) = \Theta(1)$ otherwise.
- ▶ This recursion is a relatively straightforward infinite sum (exercises) and leads to $T(n) = \Theta(n \log_k(n))$

Other key concepts

- ▶ **Worst case cost** conditions: can require care when looking up the answer.
 - ▶ For example, some data structures have $\mathcal{O}(n)$ lookup cost if the data are missing, but much better if the data are present.
 - ▶ Also some costs are predictable and rare, leading to...
- ▶ **Amortised cost**: The long term, average worst case cost, which is often better than the single case cost.
 - ▶ For example, some data structures must be periodically rebuilt when they get too big, an expensive action. But this is done rarely by construction.

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, $\dots$
- ▶ We want to sort the data into ascending order:
1, 2, 3, 4, 5, 6, 7, $\dots$
- ▶ What is the best¹ 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!)

¹Cormen et al 2010 **Introduction to Algorithms**

Quicksort: a Recursion Example

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

Quicksort: a Recursion Example

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

What if we can pivot on the
median element of A ?

Quicksort: a Recursion Example

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

What if we can pivot on the
median element of A ?

$T(n)$

Quicksort: a Recursion Example

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

What if we can pivot on the
median element of A ?

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

Quicksort: a Recursion Example

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

What if we can pivot on the
median element of A ?

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

Quicksort: a Recursion Example

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

What if we can pivot on the
median element of A ?

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

Quicksort: a Recursion Example

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

What if we can pivot on the
median element of A ?

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

Quicksort: a Recursion Example

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

What if we can pivot on the
median element of A ?

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

Quicksort: a Recursion Example

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

What if we always pivot on
the **largest element** of A ?

Quicksort: a Recursion Example

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

What if we always pivot on
the **largest element** of A ?

$T(n)$

Quicksort: a Recursion Example

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

What if we always pivot on
the **largest element** of A ?

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

Quicksort: a Recursion Example

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

What if we always pivot on
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  $\text{len}(A) == 1$  then  
    return  $A$   
  else  
     $x \leftarrow \text{pivot}(x; X)$   
     $A_l \leftarrow \{a \in A : a < x\}$   
     $A_h \leftarrow \{a \in A : a >$   
 $x\}$   
     $A_x \leftarrow \{a \in A : a =$   
 $x\}$   
     $S_l \leftarrow \text{QuickSort}(A_l)$   
     $S_h \leftarrow \text{QuickSort}(A_h)$   
    return  $[S_l, A_x, S_h]$   
  end if  
end procedure
```

What if we always pivot on
the **largest element** of A ?

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

Quicksort: a Recursion Example

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

What if we always pivot on
the **largest element** of A ?

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

Quicksort: a Recursion Example

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

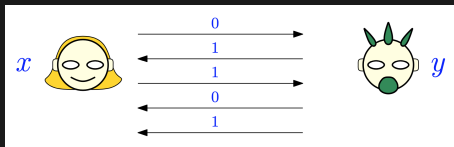
What if we always pivot on
the **largest element** of A ?

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

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

Communication Complexity



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

²According to Arora and Barak **Computational Complexity: A Modern Approach**. Hopcroft and Ullman **Introduction to Automata Theory, Languages, and Computation** use a 7-tuple.

Communication Complexity

- ▶ The **Overall cost** of P is $C(P) = \max_{x,y}[P(x,y)]$, i.e. the maximum possible cost for all data
- ▶ The **Communication complexity** of f is $C(f) = \min_{P \in \mathcal{P}}(C[P(x,y)])$
- ▶ It is the minimum number of bits needed to compute $f(x,y)$ for any x,y
- ▶ Communication Complexity Theory describes $C(f)$, 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

- ▶ $f(x, y) = \text{Parity}([x, y])$
 - ▶ $\text{Parity} = \text{mod}_2(\sum_{i=1}^n x_i)$
 - ▶ $C(f(x, y)) = 2$ 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
 - ▶ $C(f(x, y)) = n$ 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 11)
- ▶ There are many more less immediate applications
 - ▶ Particularly as a tool for algorithm and data structure lower bounds

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 (later)
- ▶ 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 $\mathcal{O}(n \log(n) + n) = \mathcal{O}(n \log(n))$

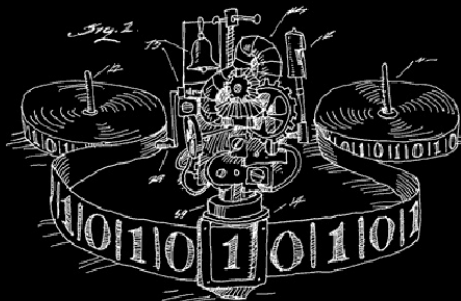
Space complexity example (2)

- ▶ 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

The Universal Turing Machine



Turing machines

High level description

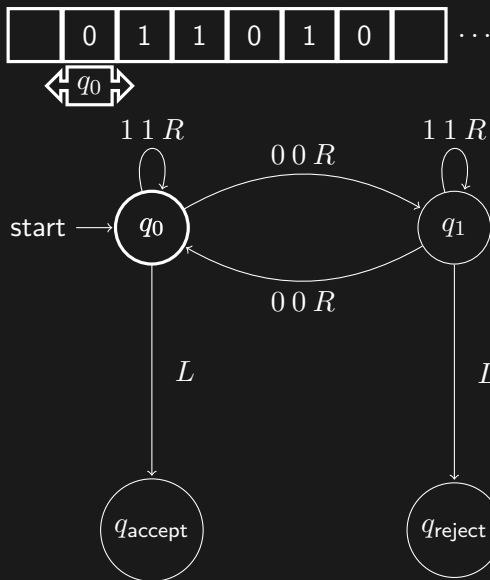
- ▶ Consider a function $f(\{x\}^d)$ where $\{x\}^d$ is a string of d bits (0 or 1)
- ▶ An algorithm for computing f is a set of rules such that we compute f for any $\{x\}^d$
- ▶ d is arbitrary
- ▶ The set of rules is fixed
- ▶ But can be arbitrarily complex and applied arbitrarily many times
- ▶ Rules are made up of **elementary operations**:
 1. Read a symbol of input
 2. Read a symbol from a “memory”
 3. Based on these, write a symbol to the “memory”
 4. Either stop and output TRUE, FALSE, or choose a new rule

Formal description

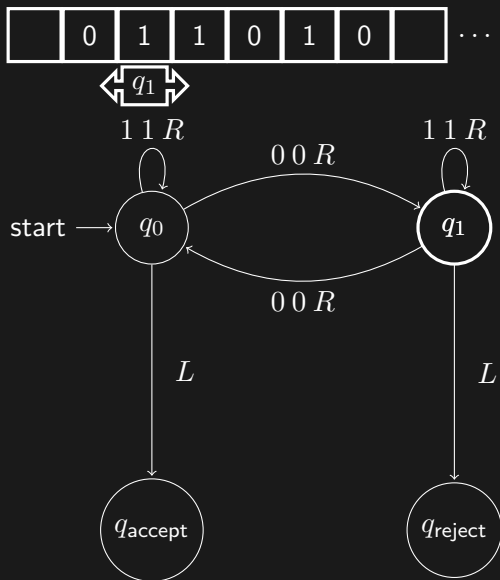
- ▶ A **Turing Machine** is a 3-tuple³ (Q, Γ, δ) :
- ▶ where Q, Γ are finite sets and:
 - ▶ Q is the set of all states, containing special states:
 - ▶ $q_0 \in Q$ is the start state
 - ▶ $q_{\text{accept}} \in Q$ is a set of accept states
 - ▶ $q_{\text{reject}} \in Q$ is a set of reject state where $q_{\text{accept}} \neq q_{\text{reject}}$
 - ▶ Γ is the tape (“memory”) alphabet with $\sqcup \in \Gamma$. The input space is $\Sigma \subset \Gamma$ excluding $\sqcup$ (the blank space).
 - ▶ $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is a transition function.

³According to Arora and Barak **Computational Complexity: A Modern Approach**. Hopcroft and Ullman **Introduction to Automata Theory, Languages, and Computation** use a 7-tuple.

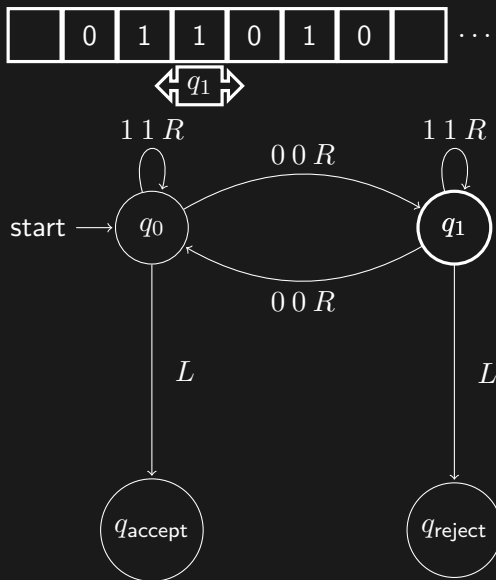
Turing Machine Example



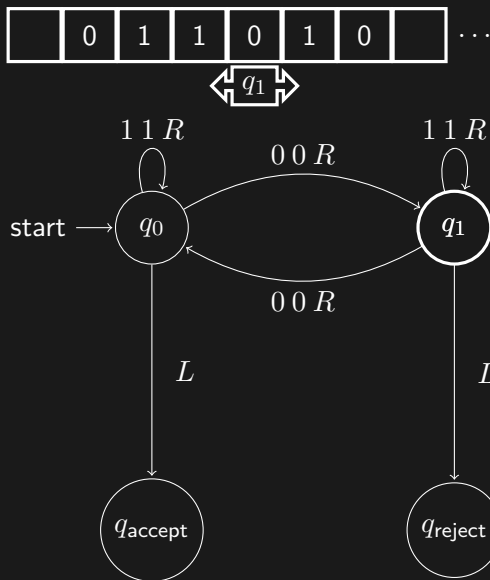
Turing Machine Example



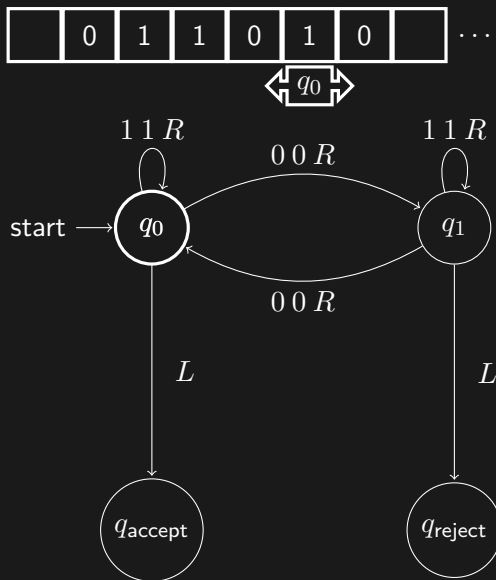
Turing Machine Example



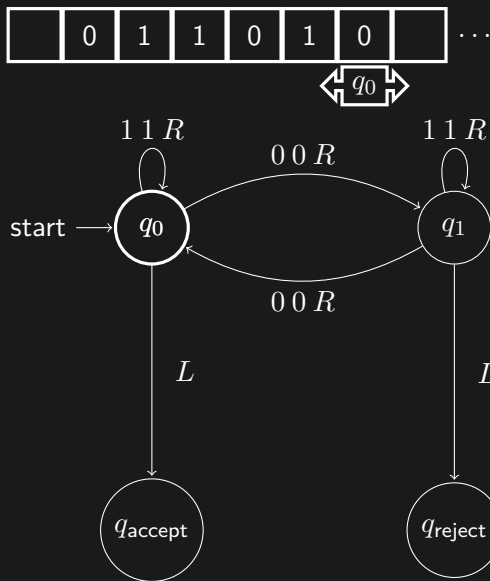
Turing Machine Example



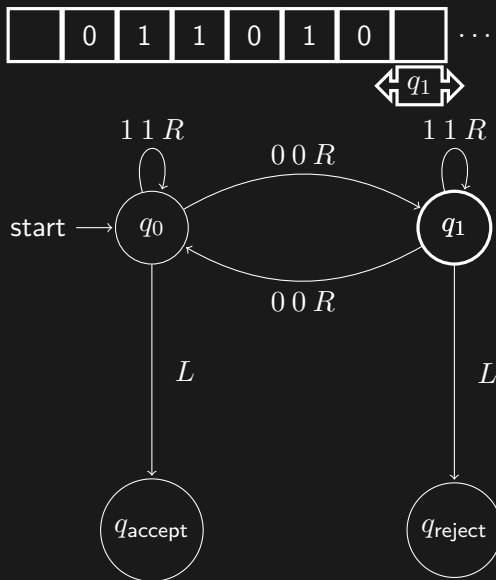
Turing Machine Example



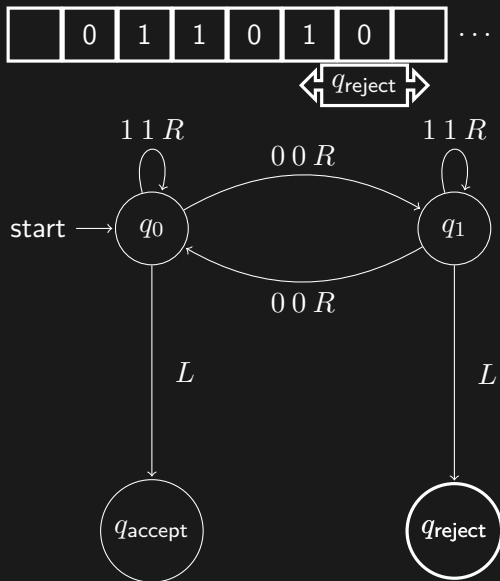
Turing Machine Example



Turing Machine Example



Turing Machine Example



Turing Machine Equivalence

- ▶ Turing Machines with the following properties are all equivalent:
 - ▶ A binary only alphabet
 - ▶ Multiple tapes
 - ▶ A doubly infinite tape
 - ▶ Designated input and/or output tapes
 - ▶ Universal Turing Machines

Conceptual objects in algorithms

- ▶ We have now met at least the following classes of object:
 1. **Functions**, which are conceptual mathematical objects
 2. **Algorithms**, which are implementations that compute a function, comprising:
 - a. **Pseudocode**, which are human-readable algorithms (though can still be precise)
 - b. **Computer code**, which is a machine-readable algorithm,
 - c. **Turing machines programmes**, which are mathematical representations of an algorithm.
- ▶ It takes proof to establish equivalence between classes of Algorithm
 - ▶ This is important for guaranteeing algorithms give the correct output
 - ▶ However, it has been proven that the correspondance between these exists.

Using Turing Machines

- ▶ Turing Machines are a tool for proving properties of Algorithms.
 - ▶ A wide class of computer architectures map to a Turing Machine
 - ▶ This allows proofs to **ignore implementation details**
 - ▶ For example: Programming language and CPU Chipset do not matter (Finiteness excepting)
- ▶ We will not use Turing Machines in proofs!
- ▶ What you need to know:
 - ▶ High level description of the Turing Machine
 - ▶ That it is used to make algorithmic proofs by connecting a Turing Machine to a particular algorithm
 - ▶ They enable a wide class of otherwise disparate computer architectures to be mapped and shown to be equivalent

Wrapup

- ▶ Complexity classes are important
- ▶ They apply to space, time, communication, memory
- ▶ Often we require approximate algorithms:
 - ▶ with better complexity
 - ▶ and quantifiable performance degradation
- ▶ However, empirical performance does not always match asymptotic complexity

References

► **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)