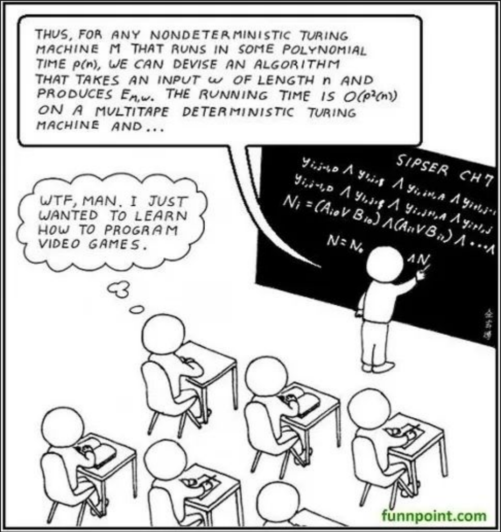
Analysing Algorithms

Daniel Lawson — University of Bristol

Lecture 09.1 (v2.0.0)

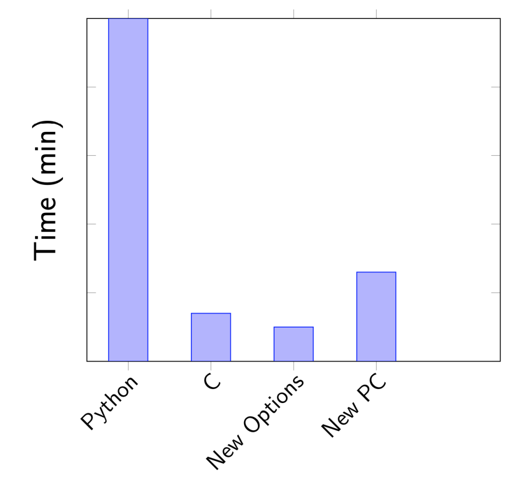
Shall we learn about Turing Machines?



Questions

- Can we prove that one algorithm is faster than another?
- What does $\mathcal{O}(f(n))$ mean?
- What is computational complexity?
- What is the best sorting 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?

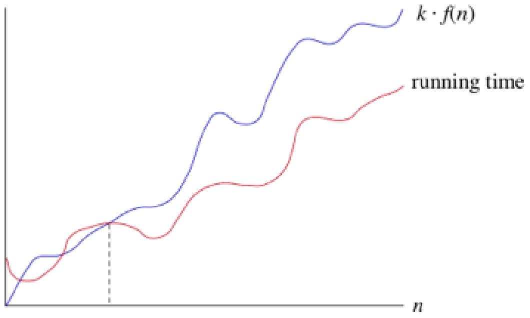
Why study algorithms?

- 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

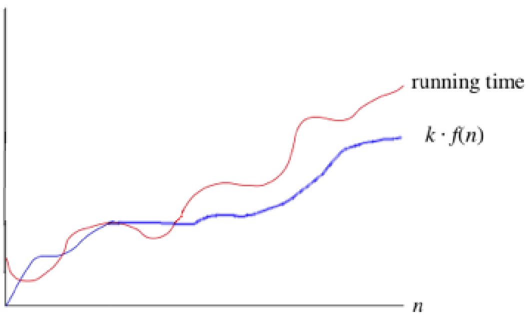
- We typically care about:
  - How long does the algorithm run for? Under which circumstances?
  - How do they trade off **runtime** and **memory requirement**?
- Some special values include **in-place** methods (which have a constant memory requirement) and **streaming** methods which visit the data 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



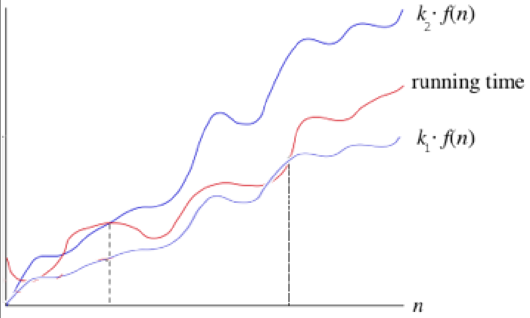
- $\mathcal{O}(n)$: An upper bound as a function of data size n
- $g(n) = \mathcal{O}(f(n))$:
  - $\exists n_0, k \in \mathbb{N}$ such that:
  - $\forall n \geq n_0$:
  - $g(n) \leq k \cdot f(n)$

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

- $n \in \mathcal{O}(n^2)$
  - $n \in \mathcal{O}(n)$ as well
  - $n \in \Omega(n)$
- $2n^2 + n + 10 \in \mathcal{O}(n^2)$
- $\log(n) \in \mathcal{O}(n^\epsilon)$ for all $\epsilon > 0$
- If $f(n) \in \mathcal{O}(g(n))$ then $g(n) \in \Omega(f(n))$
- If $f(n) \in \mathcal{O}(g(n))$ and $f(n) \in \Omega(g(n))$ then $f(n) \in \Theta(g(n))$
- If $f_1(n) \in \mathcal{O}(g_1(n))$ and $f_2(n) \in \mathcal{O}(g_2(n))$ then $f_1(n) \cdot f_2(n) \in \mathcal{O}(g_1(n) \cdot g_2(n))$
- If $f_1(n) \in \mathcal{O}(g_1(n))$ and $f_2(n) \in \mathcal{O}(g_2(n))$ then $f_1(n) + f_2(n) \in \mathcal{O}(max(g_1(n), g_2(n)))$
- $2n^2 + 3n + 1 = 2n^2 + \Theta(n) = \Theta(n^2)$

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 parallellism
- 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 ← 1
    while i ≤ n do
        a ← f_1(b, n)
        b ← f_2(a, n)
        i ← 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 $\textsc{Example}(a, b, n)$
    $i \leftarrow 1$
    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 $\textsc{QuickSort}(A)$
  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 $\textsc{QuickSort}(A)$
  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 can choose the **median element** of A?

Quicksort: a Recursion Example

procedure $\textsc{QuickSort}(A)$
  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?

$T(n)$

Quicksort: a Recursion Example

procedure $\textsc{QuickSort}(A)$
  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 $\textsc{QuickSort}(A)$
  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 can choose the **median element** of A?

$T(n)$

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

Quicksort: a Recursion Example

```
procedure QuickSort(A)
    if len(A) == 1 then
        return A
    else
        x ← 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?

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

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?

$T(n)$

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

Quicksort: a Recursion Example

procedure $\textsc{QuickSort}(A)$
  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 can choose the **median element** of A?

$T(n)$

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

Quicksort: a Recursion Example

procedure $\textsc{QuickSort}(A)$
    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?

Quicksort: a Recursion Example

procedure $\textsc{QuickSort}(A)$
    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?

$T(n)$

Quicksort: a Recursion Example

procedure $\textsc{QuickSort}(A)$
  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?

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

Quicksort: a Recursion Example

procedure $\textsc{QuickSort}(A)$
  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?

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

Quicksort: a Recursion Example

procedure $\textsc{QuickSort}(A)$
  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?

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

Quicksort: a Recursion Example

procedure $\textsc{QuickSort}(A)$
  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?

$T(n)$

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

Quicksort: a Recursion Example

```
procedure QuickSort(A)
    if len(A) == 1 then
        return A
    else
        x ← 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?

$T(n)$

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

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

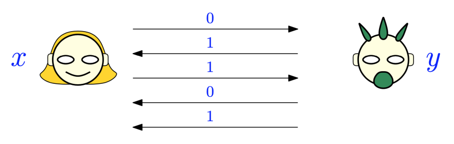
Space complexity example (2)

- 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] According 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 $\mathrm{C}(P) = \max_{x,y}[P(x, y)]$, 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)

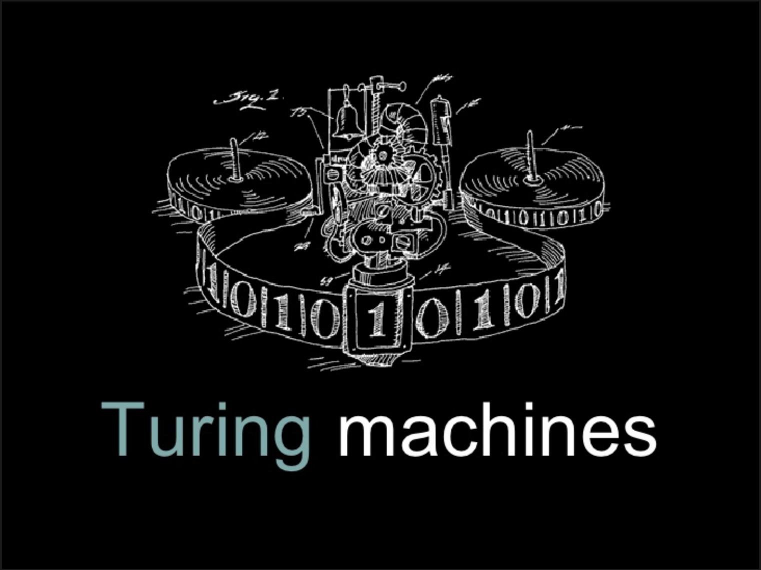
Communication Complexity Examples

- $f(x, y) = \text{Parity}([x, y])$
  - 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

The Universal Turing Machine



High level description

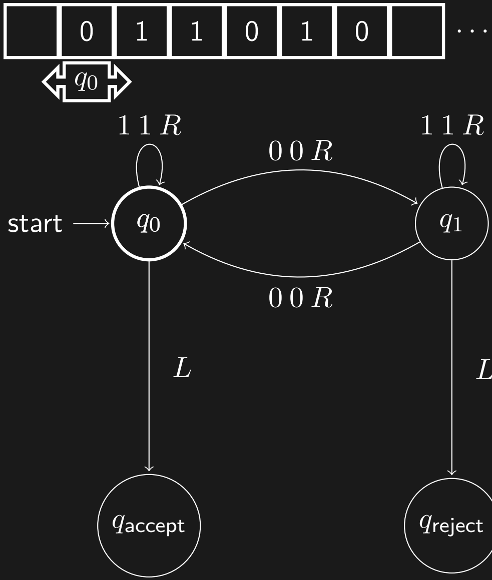
- Consider a function $f(\{x\}^d)$ where $\{x\}^d$ is a string of d bits (0 or 1)
- An algorithm for computing f is a set of rules such that we compute f for any $\{x\}^d$
- d is arbitrary
- The set of rules is fixed
- But can be arbitrarily complex and applied arbitrarily many times
- Rules are made 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

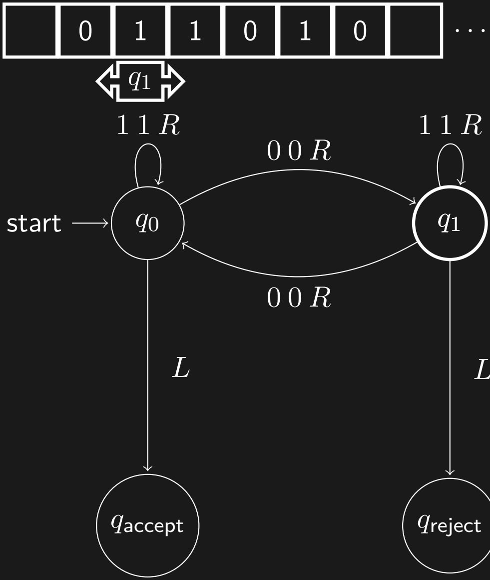
- A **Turing Machine** is a 3-tuple[3] (Q, Γ, δ):
- where Q, Γ are finite sets and:
  - Q is the set of all states, containing special states:
    - $q_0 \in Q$ is the start state
    - $q_{\text{accept}} \in Q$ is a set of accept states
    - $q_{\text{reject}} \in Q$ is a set of reject state where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

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

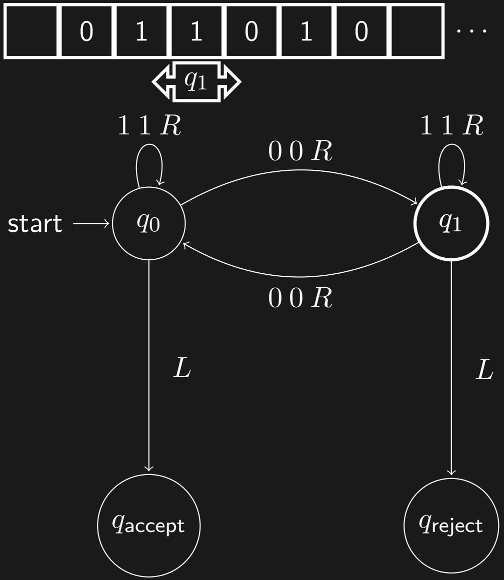
Turing Machine Example



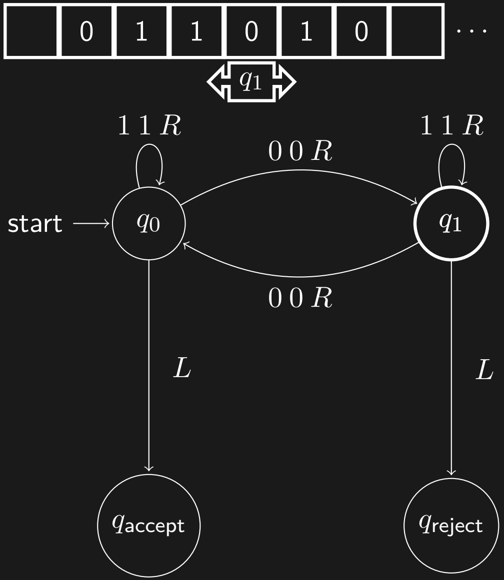
Turing Machine Example



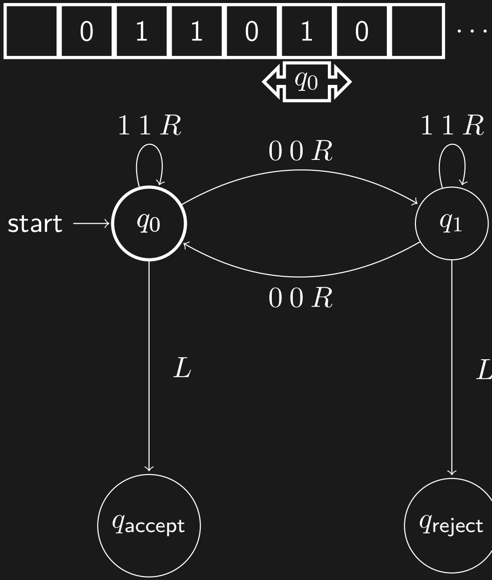
Turing Machine Example



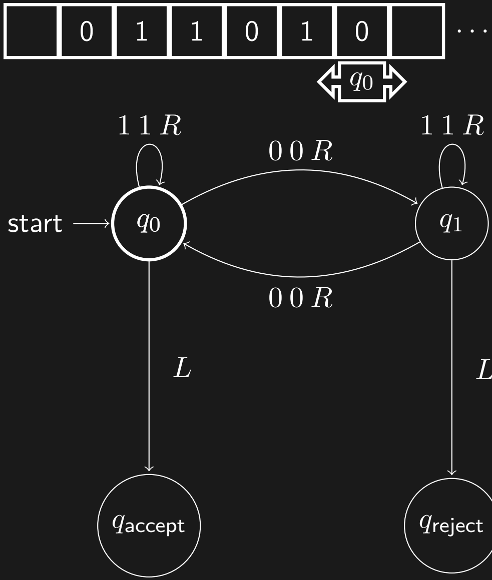
Turing Machine Example



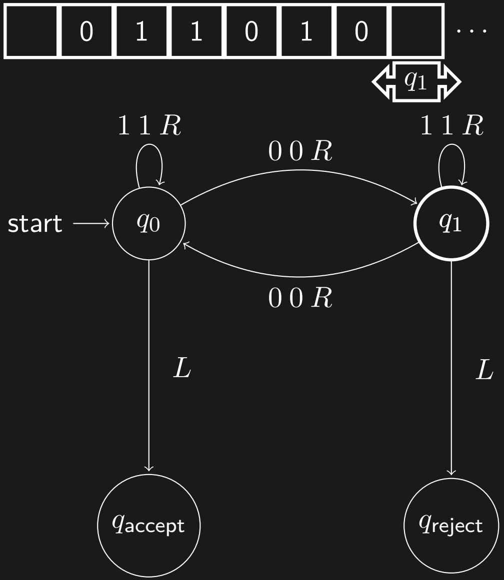
Turing Machine Example



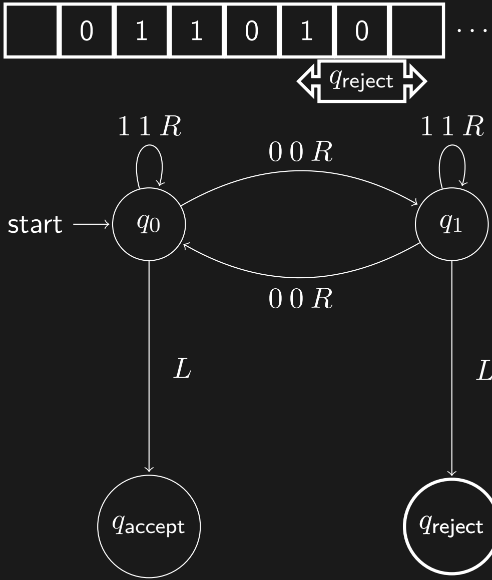
Turing Machine Example



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**
  - Fo 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 disperate computer architectures to be mapped and shown to be `equivalent`

Complexity Classes

- We often do not care about the details of a certain function
- We instead ask, "Is this function in a certain complexity class?"

Polynomial Time: P

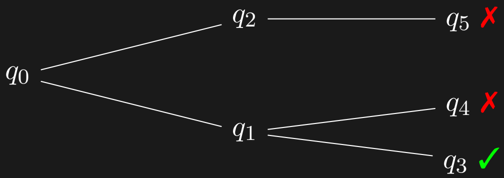
- An algorithm with time complexity $T(n)$ runs in **Polynomial Time** if $T(n) \in \cup_{i=1}^{\infty} \mathcal{O}(n^i)$.
- A language $L \in \mathsf{P}$ if there exists a Turing machine M such that:
  - M runs in polynomial time for all inputs
  - $\forall x \in L : M(x) = 1$
  - $\forall x \notin L : M(x) = 0$

Examples of algorithms in P

- **Primality Testing**: is a number x a prime number?
- **Shortest Path** in a graph: given two nodes, what is the shortest path? (for example, Dijkstra's Algorithm)
- **Minimal Weighted Matching**: Given n jobs on n machines with cost matrix c_{ij}, how do we allocate jobs? Solvable as an integer program.
- **Pattern Matching**: Asking, is a given pattern present in the data? The runtime depends on the data structure and pattern, but broad classes are solvable (e.g. graphs)

Non-Determinism

- A **Non-Deterministic** Turing machine is like a Turing Machine, except δ can go to multiple states for the same input.
- When a choice of transition is given, the Non-Deterministic Turing Machine “takes them all simultaneously’ ’.
- The machine accepts if any of the paths accept.



Non-Deterministic Polynomial Time: NP

- A language $L \in \mathrm{NP}$ if there exists a **Non-Deterministic** Turing machine M such that:
  - M runs in **Polynomial Time** for all inputs
  - $\forall x \in L : M(x) = 1$
  - $\forall x \notin L : M(x) = 0$

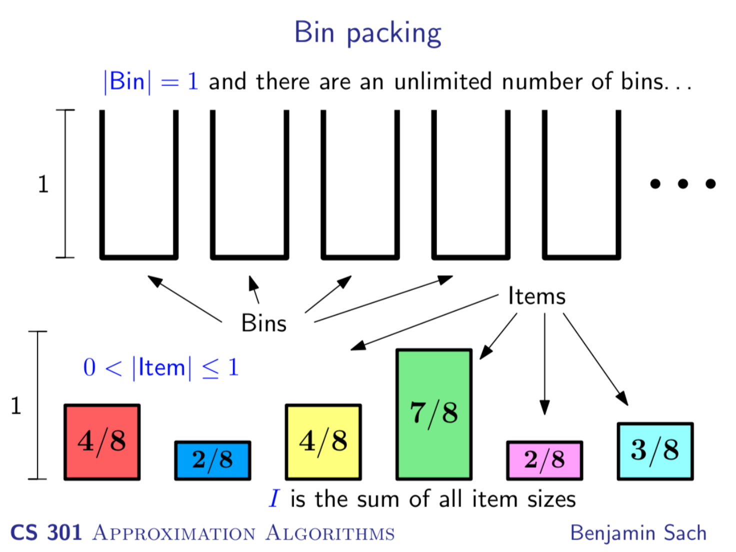
Examples of algorithms in NP

- **Travelling salesman problem**: Given a distance matrix between n cities, is there a route between them all with total distance less than D?
- **Bin packing**: Can you place n items into as few fixed-size bins as possible?
- **Boolean satisfiability**: Is a set of boolean logic statements true?
- **Integer factorisation**: Given a number x, what are its primes?

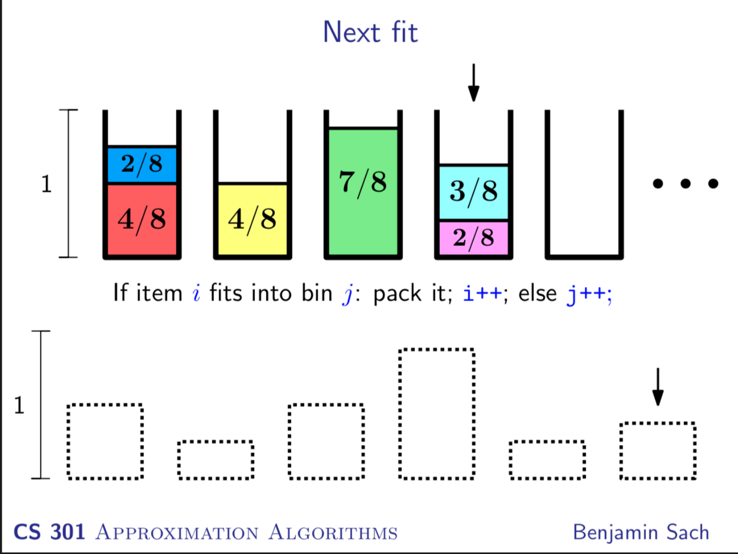
Data science consequences

- Having an algorithm is the easiest way to prove that f is in a complexity class.
  - It is hard to prove that a problem is not in P!
- Many exact problems seem to be NP.
- We can sometimes do very well with an **approximate algorithm** in P. Examples:
  - Travelling salesman: exactly solved for Euclidean distances, Christofides and Serdyukov's approximation using minimum weight perfect matching
  - Bin packing...
- Quantifying approximation error is therefore very important!

Bin packing problem

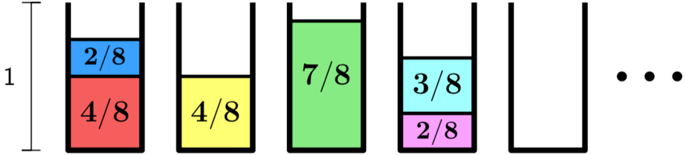


Bin packing: next fit



Bin packing: next fit

Next fit



Next fit runs in $O(n)$ time but how good is it?

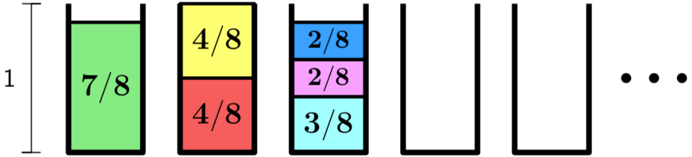
- Let fill(i) be the sum of item sizes in bin i and b the number of non-empty bins (using Next fit)
- Observe that $\text{fill}(2i-1) + \text{fill}(2i) > 1$ (for $1 \leq 2i \leq b$)

so $$\lfloor b/2 \rfloor < \sum_{1 \leq 2i \leq b} \text{fill}(2i-1) + \text{fill}(2i) \leq I \leq \text{Opt}$$

Next fit is an 2-approximation for bin packing which runs in linear time

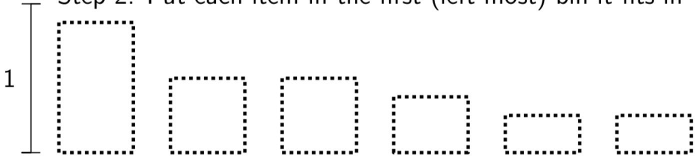
Bin packing: first fit decreasing

First fit decreasing (FFD)



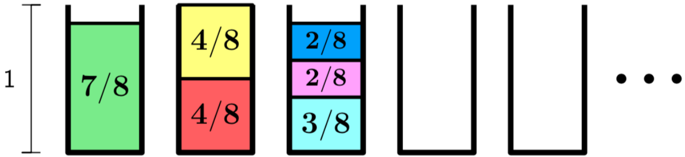
Step 1: Sort the items into non-increasing order

Step 2: Put each item in the first (left-most) bin it fits in



Bin packing: first fit decreasing

First fit decreasing (FFD)



- Consider bin $j = \lceil \frac{2b}{3} \rceil$ (FFD uses b bins on this instance)

Case 2: Bin j contains only items of size $\leq 1/2$

As $\lceil 2k/3 \rceil - 1 < I$

we have that $\lceil 2k/3 \rceil - 1 \leq 2k/3 \leq \text{Opt}$

- So FFD is a $3/2$-approximation for bin packing

Addendum

- Complexity classes are not everything!
- Some examples of algorithms in P[4]:
  - Max-Bisection is approximable to within a factor of 0.8776 in around $O(n^{10^{100}})$ time
  - Energy-driven linkage unfolding algorithm is at most $1176072512220365312000n^{79}(l_{max}/d_{min}(\Theta_0))^{26}$
  - The classic “picture dropping problem” for how to wrap string such that it that will drop when one nail is removed, with n nails, can be solved in $O(n^{43737})$
  - Approximate algorithms (accurate to within $(1+\epsilon)$ often scale badly, e.g. $O(n^{1/\epsilon})$

[4] Stack Exchange Polynomial Time algorithms with huge exponent

Wrapup

- Complexity classes are important
- They apply to space, time, communication, memory
- Often we require approximate algorithms:
  - with better complexity
  - and quantifiable peformance 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)