Analysing Algorithms (Part 3 - Turing Machines)

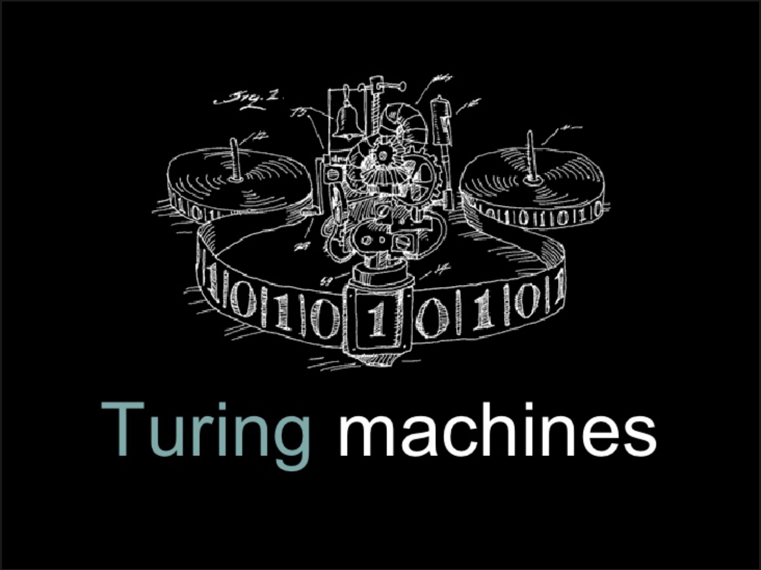
Daniel Lawson — University of Bristol

Lecture 08.1.3 (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 3
- Thanks to Turing Fellow and Computer Scientist Dan Martin for Tikz pictures and expertise

The Universal Turing Machine



High level description

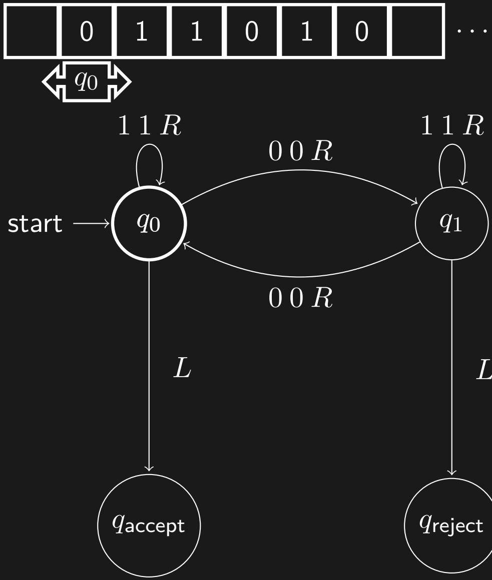
- Consider a function $f(\{x\}^d)$ where $\{x\}^d$ is a string of d bits (0 or 1)
- An algorithm for computing f is a set of rules such that we compute f for any $\{x\}^d$
- d is arbitrary
- The set of rules is fixed
- But can be arbitrarily complex and applied arbitrarily many times
- Rules are made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

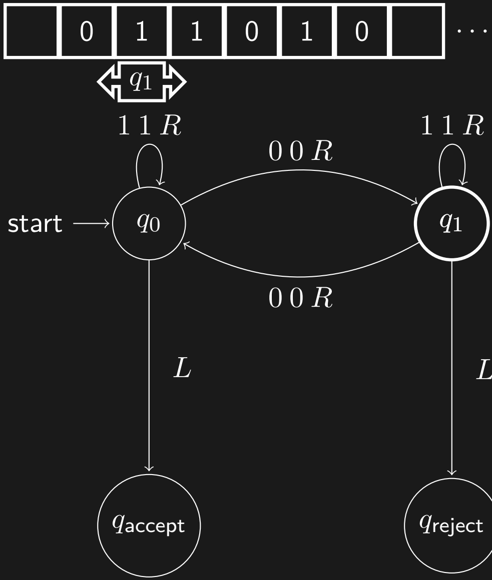
- A **Turing Machine** is a 3-tuple[1] (Q, Γ, δ):
- where Q, Γ are finite sets and:
  - Q is the set of all states, containing special states:
    - $q_0 \in Q$ is the start state
    - $q_{\text{accept}} \in Q$ is a set of accept states
    - $q_{\text{reject}} \in Q$ is a set of reject state where $q_{\text{accept}} \neq q_{\text{reject}}$
  - 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.

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

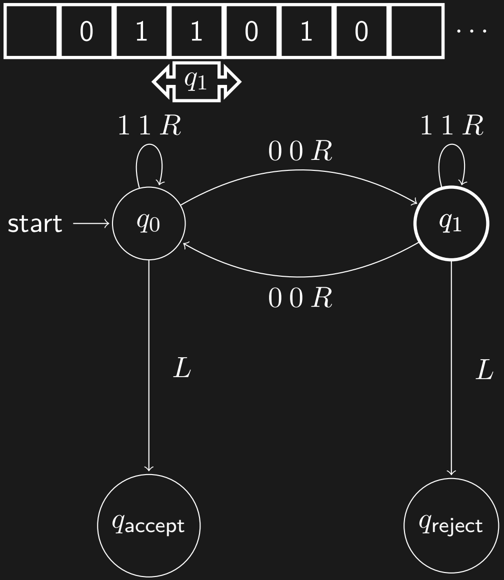
Turing Machine Example



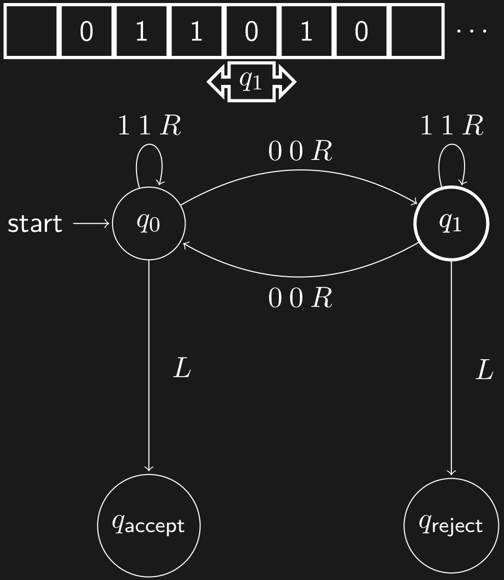
Turing Machine Example



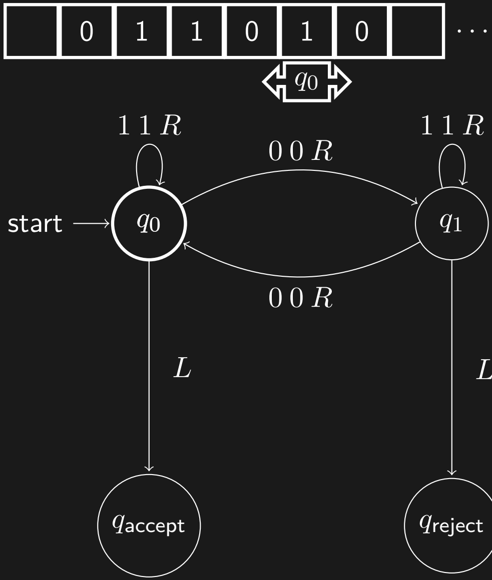
Turing Machine Example



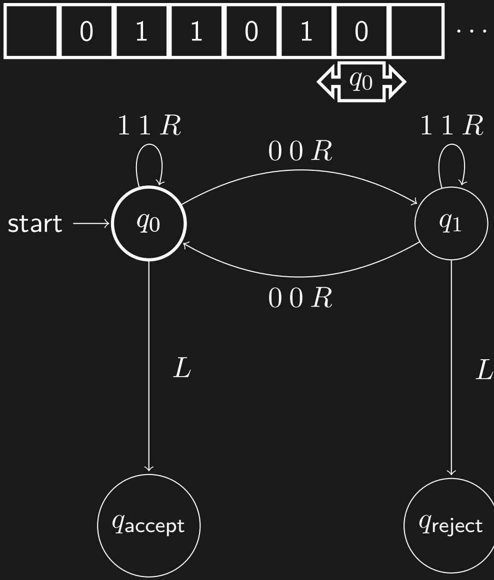
Turing Machine Example



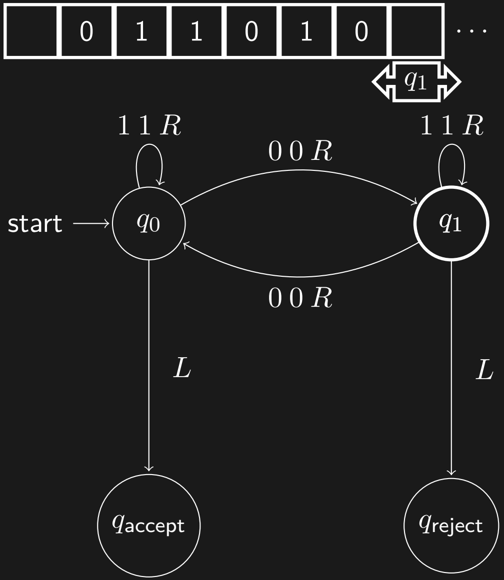
Turing Machine Example



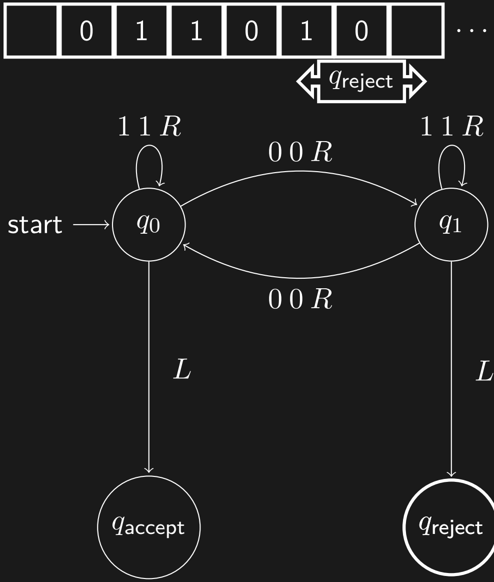
Turing Machine Example



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

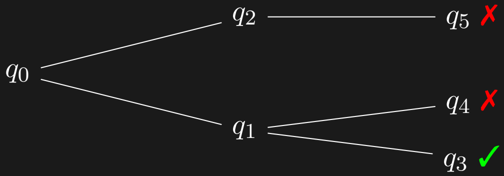
- An algorithm with time complexity $T(n)$ runs in **Polynomial Time** if $T(n) \in \cup_{i=1}^{\infty} \mathcal{O}(n^i)$.
- A language $L \in$ P if there exists a Turing machine M such that:
  - M runs in polynomial time for all inputs
  - $\forall x \in L : M(x) = 1$
  - $\forall x \notin L : M(x)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

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

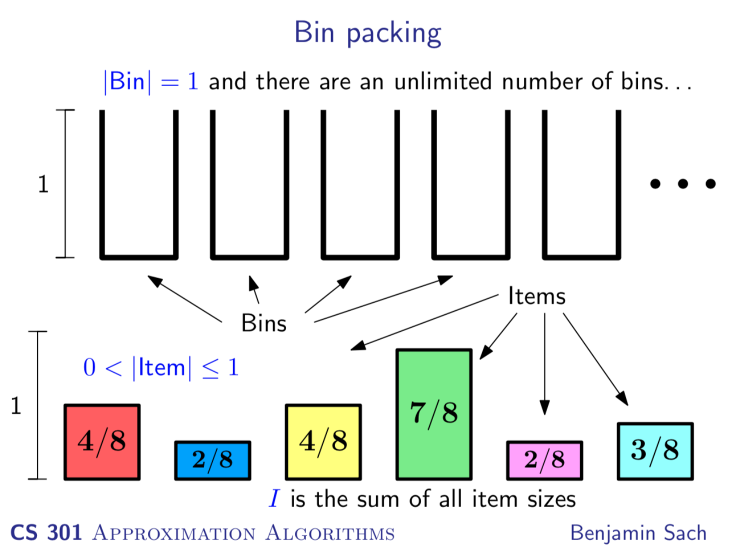
Examples of algorithms in NP

- **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

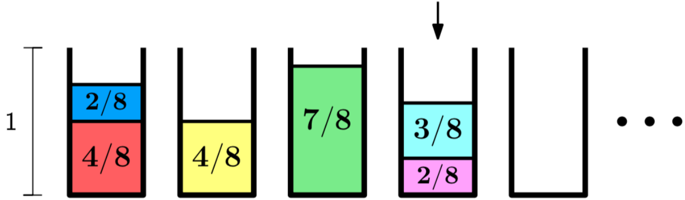
- Having an algorithm is the easiest way to prove that f is in a complexity class.
  - It is hard to prove that a problem is not in P!
- Many exact problems seem to be NP.
- We can sometimes do very well with an **approximate algorithm** in P. Examples:
  - Travelling salesman: exactly solved for Euclidean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

Bin packing problem

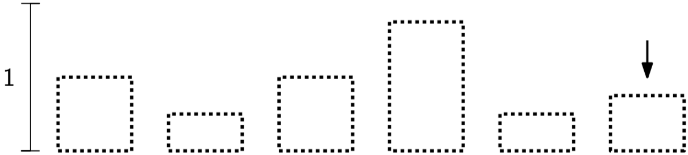


Bin packing: next fit

Next fit

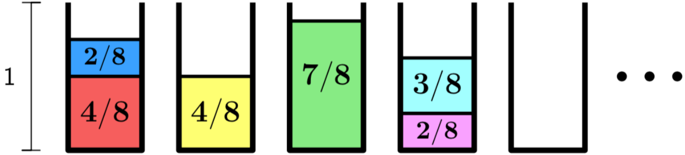


If item i fits into bin j: pack it; `i++`; else `j++`;



Bin packing: next fit

Next fit



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

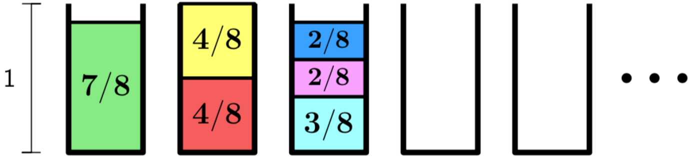
- Let fill(i) be the sum of item sizes in bin i
  and b the number of non-empty bins (using Next fit)
- Observe that $\text{fill}(2i-1) + \text{fill}(2i) > 1$ (for $1 \leq 2i \leq b$)

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

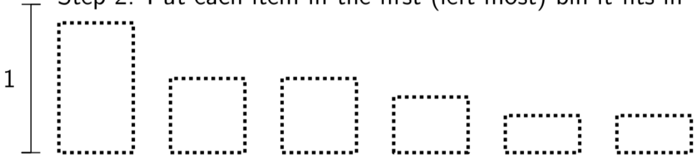
Bin packing: first fit decreasing

First fit decreasing (FFD)



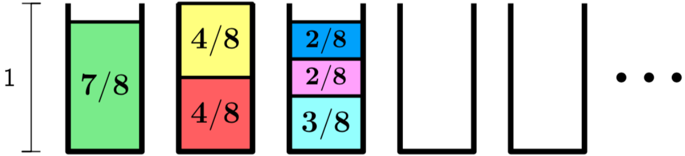
Step 1: Sort the items into non-increasing order

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[2]:
  - 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 $1176072512203653120000 n^{79}(l_{max}/d_{min}(\Theta_0))^{26}$
  - 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. $O(n^{1/\epsilon})$

[2] 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

Reflection

- In what sense is a Turing Machine Universal?
- Can we think of Turing Machines as having complex, compound states, or are we restricted to only simple bit operations?
- What role does Computational Complexity have in data science?
- By the end of the course, you should:
  - Understand the relationship between representations of algorithms
  - Be able to reason about the Turing Machine at a high level
  - Be able to describe the classes P and NP, and place complexity of algorithms in them

Signposting

- Next up: 8.2 Algorithms for Data Science

References

- Arora and Barak Computational Complexity: A Modern Approach
- Hopcroft and Ullman Introduction to Automata Theory, Languages, and Computation
- Annie Raymond's Lecture notes on bipartite matching
- Fan et al 2010 Graph Pattern Matching: From Intractable to Polynomial Time
- Stack Exchange Polynomial Time algorithms with huge exponent