Algorithms employ randomness to improve worst-case runtime, compute correct solutions to hard problems more efficiently but with low probability of failure, compute approximate solutions to hard problems, and in computer games: may want aliens to move in a random pattern, the layout of a dungeon may be randomly generated, may want to introduce unpredictability in physics/applied maths: carry out simulations to determine behaviour e.g. models of molecules are often assume to move randomly, in testing: stress test components by bombarding them with random data, random data is often seen as unbiased data giving average performance (e.g. in sorting algorithms), and in cryptography.

Reminder: Random Numbers

In most programming languages, random() // generates random numbers in a given 0 .. RAND_MAX
where the constant RAND_MAX may depend on the computer, e.g. RAND_MAX = 2147483647
To convert to a number between 0 .. RANGE
compute the remainder after division by RANGE+1
Two functions are required:

srandom(int seed) // sets its argument as the seed
random() // uses a LCG technique to generate random numbers in the range 0 .. RAND_MAX

where the constant RAND_MAX is defined in stdlib.h (depends on the computer: on the CSE network, RAND_MAX = 2147483647)
The period length of this random number generator is very large approximately 16 \cdot (2^{31} - 1)

Analysis of Randomised Algorithms

Randomised algorithm to find some element with key $k$ in an unordered list:

```plaintext
findKey(L,k):
  Input list L, key k
  Output some element in L with key k
  repeat
    randomly select e \in L
    until key(e)=k
  return e
```

... Analysis of Randomised Algorithms

Analysis:

- $p$ ... ratio of elements in $L$ with key $k$ (e.g. $p = \frac{1}{3}$)
- Probability of success: 1 (if $p > 0$)
- Expected runtime: $\frac{1}{p} \left( = \lim_{n\to\infty} \sum_{i=0}^{n} i \cdot (1-p)^{i-1} \cdot p \right)$
  - Example: a third of the elements have key $k \Rightarrow$ expected number of iterations $= 3$

... Analysis of Randomised Algorithms

If we cannot guarantee that the list contains any elements with key $k$ ...

```plaintext
findKey(L,k,d):
  Input list L, key k, maximum #attempts d
  Output some element in L with key k
  repeat
    if d=0 then
      return failure
    end if
    randomly select e \in L
    d=d-1
  until key(e)=k
  return e
```

... Analysis of Randomised Algorithms

Analysis:

- $p$ ... ratio of elements in $L$ with key $k$
- $d$ ... maximum number of attempts
Randomised Algorithms

Non-randomised Quicksort

Reminder: Quicksort applies divide and conquer to sorting:

- **Divide**
  - pick a pivot element
  - move all elements smaller than the pivot to its left
  - move all elements greater than the pivot to its right
- **Conquer**
  - sort the elements on the left
  - sort the elements on the right

Worst-case Running Time

Worst case for Quicksort occurs when the pivot is the unique minimum or maximum element:

- One of the intervals low..pivot-1 and pivot+1..high is of size n-1 and the other is of size 0
  - running time is proportional to n + (n-1) + ... + 2 + 1
- Hence the worst case for non-randomised Quicksort is $O(n^2)$
Randomised Quicksort

\[ \text{partition}(\text{array, low, high):} \]
\[
\begin{align*}
\text{Input} & \quad \text{array, index range low..high} \\
\text{Output} & \quad \text{randomly select a pivot element from array[low..high]} \\
& \quad \text{moves all smaller elements between low..high to its left} \\
& \quad \text{moves all larger elements between low..high to its right} \\
& \quad \text{returns new position of pivot element} \\
\end{align*}
\]

\[ \text{randomly select pivot item=\text{array[low..high]}, left=low, right=high} \]

\[ \text{while left<right do} \]
\[ \begin{align*}
\text{left} & = \text{find index of leftmost element > pivot item} \\
\text{right} & = \text{find index of rightmost element <= pivot item} \\
\text{if} & \quad \text{left<right then} \\
& \quad \text{swap array[left] and array[right]} \\
\text{end if} \\
\end{align*} \]

\[ \text{end while} \]

array[right]=pivot_item // right is final position for pivot

\[ \text{return right} \]

Analysis:

- Consider a recursive call to \text{partition()} on an index range of size \( s \)
  - \text{Good call:} both low..pivot-1 and pivot+1..high shorter than \( \frac{3}{4} \cdot s \)
  - \text{Bad call:} one of low..pivot-1 or pivot+1..high greater than \( \frac{3}{4} \cdot s \)
- Probability that a call is good: 0.5 (because half the possible pivot elements cause a good call)

Example of a bad call:

6 3 7 5 8 2 4 1

Example of a good call:

6 3 5 2 4 1 7 8

Minimum Cut Problem

Reminder: Graph \( G = (V,E) \)

- \text{set of vertices} \( V \)
- \text{set of edges} \( E \)

\text{Cut} of a graph …

- a partition of \( V \) into \( S \cup T \)
  - \( S, T \) disjoint and both non-empty
  - its weight is the number of edges between \( S \) and \( T \):
    \[ \omega(S,T) = | \{ (s,t) \in E : s \in S \land t \in T \} | \]

\text{Minimum cut problem} … find a cut of \( G \) with minimal weight

Example:
**Contraction**

*Contracting edge* $e = \{v, w\}$ ...

- remove edge $e$
- replace vertices $v$ and $w$ by new node $n$
- replace all edges $\{x, v\}, \{x, w\}$ by $\{x, n\}$

... results in a *multigraph* (multiple edges between vertices allowed)

**Example:**

![Example Diagram]

**Randomised algorithm for graph contraction** = repeated edge contraction until 2 vertices remain

$\text{contract}(G)$:

| Input | graph $G = (V, E)$ with $|V| \geq 2$ vertices |
| Output | cut of $G$ |

| while $|V| > 2$ do |
| randomly select $e \in E$ |
| contract edge $e$ in $G$ |
| end while |
| return the only cut in $G$ |

**Exercise #1: Graph Contraction**

Apply the contraction algorithm twice to the following graph, with different random choices:

![Exercise Diagram]

---

**... Contraction**

Analysis:

$n$ ... number of vertices

- Probability of $\text{contract}$ to result in a minimum cut:

  \[
  \binom{n}{2} \left( \frac{2^{n-1} - 1}{2^n} \right)
  \]

  because every graph has $2^{n-1} - 1$ cuts, of which at most \( \binom{n}{2} \) can have minimum weight

- This is much higher than the probability of picking a minimum cut at random:

  \[
  \frac{1}{\binom{n}{2}}
  \]

- Single edge contraction can be implemented in $O(n)$ time on an adjacency-list representation ⇒
  total running time: $O(n^2)$

(Best known implementation uses $O(|E|)$ time)

---

**Karger's Algorithm**

Idea: Repeat random graph contraction several times and take the best cut found

$\text{MinCut}(G)$:

| Input | graph $G$ with $n \geq 2$ vertices |
| Output | smallest cut found |

| min_weight = $\infty$, $d=0$ |
| repeat |
| cut = $\text{contract}(G)$ |
| if $\text{weight}(cut) < \text{min_weight}$ then |
| min_cut = cut, min_weight = $\text{weight}(cut)$ |
| end if |
| $d = d+1$ |
| until $d > \binom{n}{2} \cdot \ln n$ |
| return min_cut |

---

**... Karger's Algorithm**

Analysis:
n … number of vertices
m … number of edges

- **Probability of success**: \(1 - \frac{1}{n}\)
  - probability of not finding a minimum cut when the contraction algorithm is repeated \(d = \binom{n}{2}\) times:
  \[
  \left[1 - \frac{1}{\binom{n}{2}}\right]^d \leq \frac{1}{e \ln n} = \frac{1}{n}
  \]
- **Total running time**: \(O(m \cdot d) = O(m \cdot n^2 \cdot \log n)\)
  - assuming edge contraction implemented in \(O(m)\)

### Randomised Algorithms for NP-Complete Problems

Many NP-complete problems can be tackled by randomised algorithms that

- compute nearly optimal solutions
  - with high probability

Examples:

- travelling salesman
- constraint satisfaction problems, satisfiability
- … and many more

### Simulation

#### Sidetrack: Approximation

**Approximation** is often used to solve numerical problems by

- solving a simpler, but much more easily solved, problem
- where this new problem gives an approximate solution
- and refine the method until it is “accurate enough”

Examples:

- length of a curve determined by a function \(f\)
- area under a curve for a function \(f\)
- roots of a function \(f\)

#### Sidetrack: Approximation

Example: Length of a Curve

Estimate length: approximate curve as sequence of straight lines.

#### Sidetrack: Approximation

Example: Length = curveLength(0, \(\pi\), \(\sin\));

Convergence when using more and more steps

<table>
<thead>
<tr>
<th>steps</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.000000</td>
</tr>
<tr>
<td>10</td>
<td>3.815283</td>
</tr>
<tr>
<td>100</td>
<td>3.820149</td>
</tr>
<tr>
<td>1000</td>
<td>3.820197</td>
</tr>
<tr>
<td>10000</td>
<td>3.819753</td>
</tr>
</tbody>
</table>
Simulation

In some problem scenarios

- it is difficult to devise an analytical solution
- so build a software model and run experiments

Examples: weather forecasting, traffic flow, queueing, games

Such systems typically require random number generation

- distributions: uniform, numerical, normal, exponential

Accuracy of results depends on accuracy of model.

Example: Gambling Game

Consider the following game:

- you bet $1 and roll two dice (6-sided)
- if total is between 8 and 11, you get $2 back
- if total is 12, you get $6 back
- otherwise, you lose your money

Is this game worth playing?

Test: start with $5 and play until you have $0 or $20.

In fact, this example is reasonably easy to solve analytically.

Example: Area inside a Curve

Scenario:

- have a closed curve defined by a complex function
- have a function to compute "X is inside/outside curve?"

Simulation approach to determining the area:

- determine a region completely enclosing curve
- generate very many random points in this region
- for each point x, compute inside(x)
- count number of insides and outsides
- areaWithinCurve = totalArea * insides/(insides+outsides)

I.e. we approximate the area within the curve by using the ratio of points inside the curve against those outside

Also known as Monte Carlo estimation
Summary

- Analysis of randomised algorithms
  - probability of success
  - expected runtime
- Randomised quicksort
- Karger's algorithm
- Approximation and simulation