Week 13: Randomised Algorithms

Randomised Algorithms

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

Randomness

Randomness is also useful

- 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*
    - gives average performance (e.g. in sorting algorithms)
- 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:

```
findKey(L,k):
|  Input  list L, key k
|  Output some element in L with key k
|
|  repeat
|     randomly select e∈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*: $\frac{1}{p}$ $\left(= \lim_{n\to\infty} \sum_{i=1..n} i \cdot (1-p)^{i-1} \cdot p\right)$
  - 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 …

```
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∈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

- *Probability of success*: $1 - p^d$
- *Expected runtime*: $\left(\sum_{i=1..d} i \cdot (1-p)^{i-1} \cdot p\right) + d \cdot (1-p)^{d-1}$
  - O(1) if d is a constant

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

... Non-randomised Quicksort

Divide ...

```
partition(array,low,high):
|  Input   array, index range low..high
|  Output  selects array[low] as pivot element
|          moves all smaller elements between low+1..high to its left
|          moves all larger elements between low+1..high to its right
|          returns new position of pivot element
|
|  pivot_item=array[low], left=low+1, right=high
|  while left<right do
|  |  left  = find index of leftmost element > pivot_item
|  |  right = find index of rightmost element <= pivot_item
|  |  if left<right then
|  |     swap array[left] and array[right]
|  |  end if
|  end while
|  array[low]=array[right] // right is final position for pivot
|  array[right]=pivot_item
|  return right
```

... Non-randomised Quicksort

... and Conquer!

```
Quicksort(array,low,high):
|  Input   array, index range low..high
|  Output  array[low..high] sorted
|
|  if high > low then        // termination condition low >= high
|  |  pivot = partition(array,low,high)
|  |  Quicksort(array,low,pivot-1)
|  |  Quicksort(array,pivot+1,high)
|  end if
```

... Non-randomised Quicksort


```
3  6  5  2  4  1

3  1  5  2  4  6

3  1  2  5  4  6

2  1   | 3 |   6  4  5

1  2   | 3 |   6  4  5

1  2   | 3 |   5  4   | 6 |

1  2   | 3 |   4  5   | 6 |
```

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
  $\Rightarrow$ running time is proportional to $n + n\text{-}1 + \ldots + 2 + 1$
- Hence the worst case for non-randomised Quicksort is $O(n^2)$

```
6  5  4  3  2  1

5  4  3  2  1  |  6

4  3  2  1  |  5  |  6

3  2  1  |  4  |  5  |  6

              ...

1  |  2  |  3  |  4  |  5  |  6
```

Randomised Quicksort

14/37

```
partition(array,low,high):
|  Input   array, index range low..high
|  Output  randomly select a pivot element from array[low..high]
|          moves all smaller elements between low..high to its left
|          moves all larger elements between low..high to its right
|          returns new position of pivot element
|
|  randomly select pivot_item∈array[low..high], left=low, right=high
|  while left<right do
|  |  left  = find index of leftmost element > pivot_item
|  |  right = find index of rightmost element <= pivot_item
|  |  if left<right then
|  |     swap array[left] and array[right]
|  |  end if
|  end while
|  array[right]=pivot_item // right is final position for pivot
|  return right
```

... Randomised Quicksort

15/37

Analysis:

- Consider a recursive call to `partition()` on an index range of size s
  - *Good call*:
    both `low..pivot-1` and `pivot+1..high` shorter than $¾ \cdot s$
  - *Bad call*:
    one of `low..pivot-1` or `pivot+1..high` greater than $¾ \cdot s$
- 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

6   3   5   2   4   1    | 7 |    8
```

Example of a good call:

```
6   3   5   2   4   1    | 7 |    8

2   1    | 3 |    6   5   4    | 7 |    8
```

... Randomised Quicksort

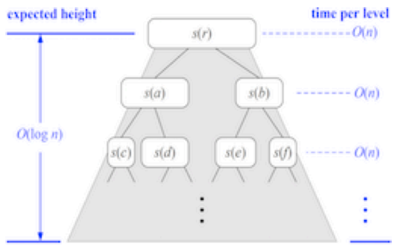
16/37

n … size of array

From probability theory we know that the expected number of coin tosses required in order to get k heads is 2·k

- For a recursive call at depth d we expect
  - $d/2$ ancestors are good calls
    $\Rightarrow$ size of input sequence for current call is $\leq (¾)^{d/2} \cdot n$
- Therefore,
  - the input of a recursive call at depth $2 \cdot \log_{4/3} n$ has expected size 1
    $\Rightarrow$ the expected recursion depth thus is $O(\log n)$
- The total amount of work done at all the nodes of the same depth is $O(n)$

Hence the expected runtime is $O(n \cdot \log n)$



Minimum Cut Problem

17/37

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

- set of vertices V
- set of edges E

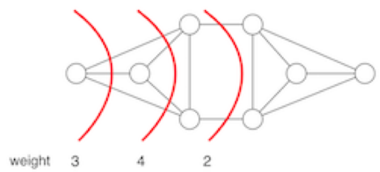
Cut of a graph …

- 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:

$$\omega(S,T) = \mid \{ \{s,t\} \in E : s \in S \wedge t \in T \} \mid$$

Minimum cut problem … find a cut of G with minimal weight

Example:



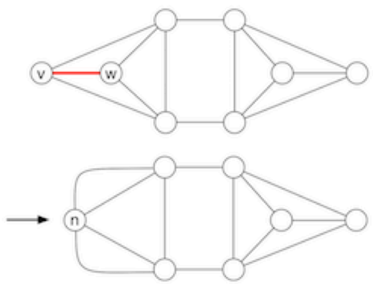
Contraction

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:



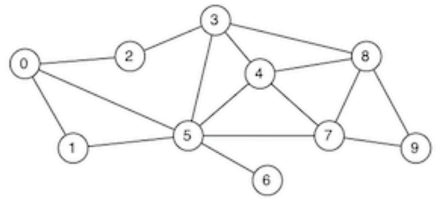
... Contraction

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

```
contract(G):
|  Input  graph G = (V,E) with |V|≥2 vertices
|  Output cut of G
|
|  while |V|>2 do
|     randomly select e∈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:



... Contraction

Analysis:

n … number of vertices

- Probability of `contract` to result in a minimum cut:
  $$\binom{n}{2} / (2^{n-1} - 1)$$
  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:
  $$1 / \binom{n}{2}$$
- 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

```
MinCut(G):
|  Input  graph G with n≥2 vertices
|  Output smallest cut found
|
|  min_weight=∞, d=0
|  repeat
|  |  cut=contract(G)
|  |  if weight(cut)<min_weight then
|  |     min_cut=cut, min_weight=weight(cut)
|  |  end if
|  |  d=d+1
|  until d > binomial(n,2)·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 $d = \binom{n}{2} \cdot \ln n$ times:

$$\left[1 - 1/\binom{n}{2}\right]^d \leq \frac{1}{e^{\ln n}} = \frac{1}{n}$$

- 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"

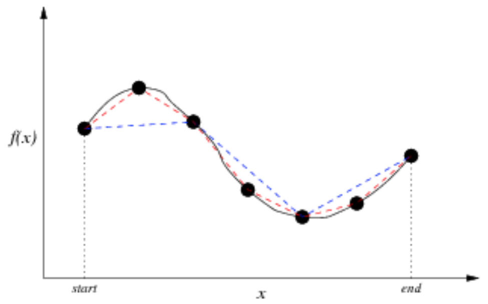
Examples:

- 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

```
curveLength(f,start,end):
|  Input   function f, start and end point
|  Output  curve length between f(start) and f(end)
|
|  length=0, δ=(end-start)/StepSize
|  for each x∈[start+δ,start+2δ,..,end] do
|     length = length + sqrt(δ^2 + (f(x)-f(x-δ))^2)
|  end for
|  return length
```


... Sidetrack: Approximation

Trade-offs in this method:

- large step size …
  - less steps, less computation (faster), lower accuracy
- small step size …
  - more steps, more computation (slower), higher accuracy

However, too many steps may lead to higher rounding error.

Each *f* has an optimal step size …

- but this is difficult to determine in advance

... Sidetrack: Approximation

```
Example: length = curveLength(0,π,sin);
```

Convergence when using more and more steps

```
steps =      0, length = 0.000000
steps =     10, length = 3.815283
steps =    100, length = 3.820149
steps =   1000, length = 3.820197
steps =  10000, length = 3.819753
```

```
steps =  100000, length = 3.820198
steps = 1000000, length = 3.820198
```

Actual answer is 3.820197789...

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: Gambling Game

We can get a reasonable approximation by simulation

- set our initial *balance* to $5
- generate two random numbers in range 1..6 (dice)
- adjust *balance* by payout or loss
- repeat above until *balance ≤ $0 or balance ≥ $20*
- run a very large number of trials like the above
- collect statistics on the outcome

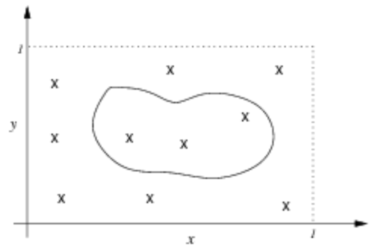
... Example: Gambling Game


```
gameSimulation:
|  Output likelihood of ending with a balance ≥$20
|
|  nwins=0
|  for a large number of Trials do
|  |  balance=$5
|  |  while balance>$0 ∧ balance<$20 do
|  |  |  balance=balance-$1
|  |  |  die1=random number∈[1..6], die2=random number∈[1..6]
|  |  |  if 7≤die1+die2≤11 then
|  |  |     balance=balance+$2
|  |  |  else if die1+die2=12 then
|  |  |     balance=balance+$6
|  |  |  end if
|  |  end while
|  |  if balance≥$20 then
|  |     nwins=nwins+1
|  |  end if
|  end for
|  return nwins/Trials
```

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?"



... Example: Area inside a 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