CS 598sm

Probabilistic &
Approximate
Computing

http://misailo.web.engr.Illinois.edu/courses/cs598

(Recap)
Numerical Error Analysis:

Information-flow Type Systems

Relational Logic Reasoning

Numb3rs

Integers vs Machine Integers

- Precision
- Signed/unsigned

Reals vs Rationals vs Floats vs FixedPoint

- Precision
- Special values

Complex numbers etc.

Error Metric

For idealized computation P (running on idealized input x) and approximate computation P' (running on the approximated input x':

$$Err = \max_{x,x'} |P(x) - P'(x')|$$

Lipschitz Continuity

Sets a linear bound on error propagation:

$$\forall x_1, x_2 \ . \ |f(x_1) - f(x_2)| \leq K \cdot |x_1 - x_2|$$

Locally Lipschitz continuous in neighborhood U

$$\forall x_1, \forall x_2 \in U(x_1) \ . \ |f(x_1) - f(x_2)| \leq K \cdot |x_1 - x_2|$$

Error Propagation

f, x original, f', x' approximate/noisy

$$|f(x) - f'(x')| \leq |f(x) - f'(x) + f'(x) - f'(x')|$$

$$\leq |f(x) - f'(x)| + |f'(x) - f'(x')|$$

$$\leq \text{ErrorApprox} \quad + \text{ErrorPropagate*}$$

*assuming that f' propagates error
in the same way as f

Adding Some Randomness

Coin Flip:

- Heads,
- Tails
- Each outcome is uncertain

Fair Coin: Probability of heads/tails is equal, ½

Biased Coin: Probabilities of heads/tails differ

Dice

Select one of 6 options

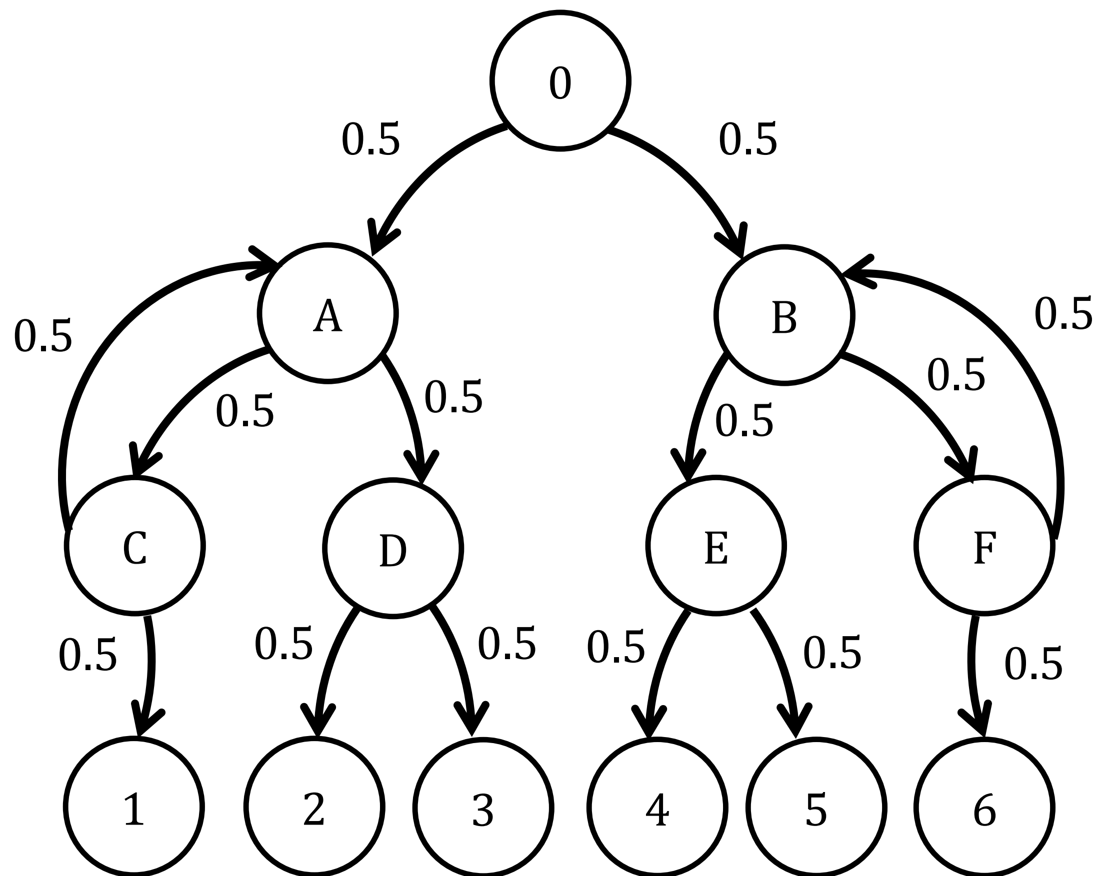
Fair or Loaded Dice

Can be simulated using the coin

- How many flips are necessary for a roll?

Example: Simulating a Dice Roll

Implement a dice roll using a fair coin (*)



* Example from "Probabilistic Programming", Gordon, Henzinger, Nori, Rajamani (ICSE-FoSE, 2014)

Random Number Generators

Most are not really random

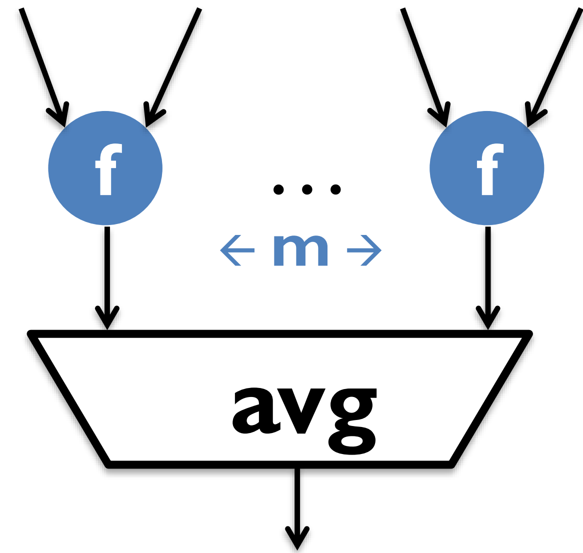
- However can have long cycles
- Pseudorandom number generators should satisfy various important properties
- Parallelization brings its own challenges

How much randomness do we need for approximation?

USING RANDOMNESS

Example Computation

```
for (i=0; i < m; i++) {
  sum = sum + f(i)
}

avg = sum / m
```



Selecting Implementations

Implementation A:
- Fully accurate
- Runs in time Ta

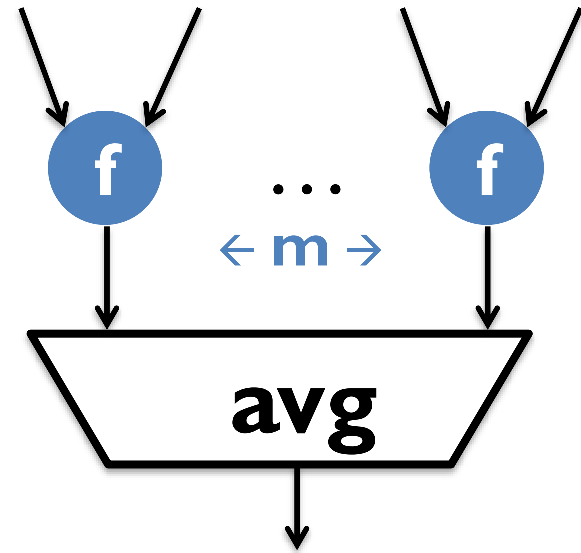
Implementation B:
- Has absolute error E
- Runs in time Tb

How do we implement a program that has an expected error of E/2 ?

For how much time does such a program run (on average)?

Selecting Implementation

```
for (i=0; i < m; i++) {
  t = 0;
  if (coinflip(pf_1))
     t = f_a(i);
  else
     t = f_b(i);

  sum = sum + t
}
avg = sum / m
```



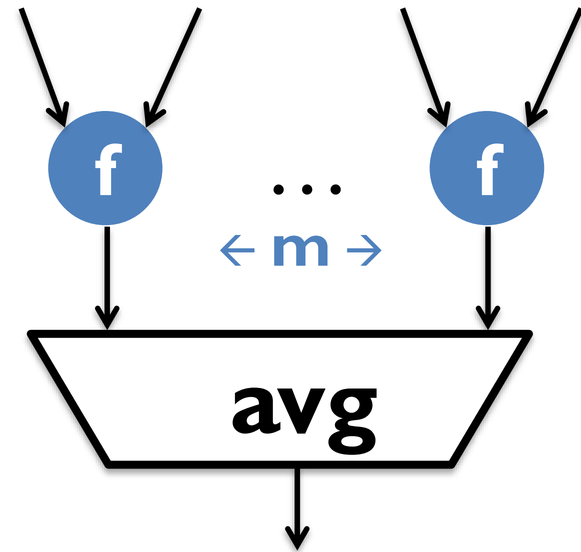
Sampling

Select a subset of elements from a list

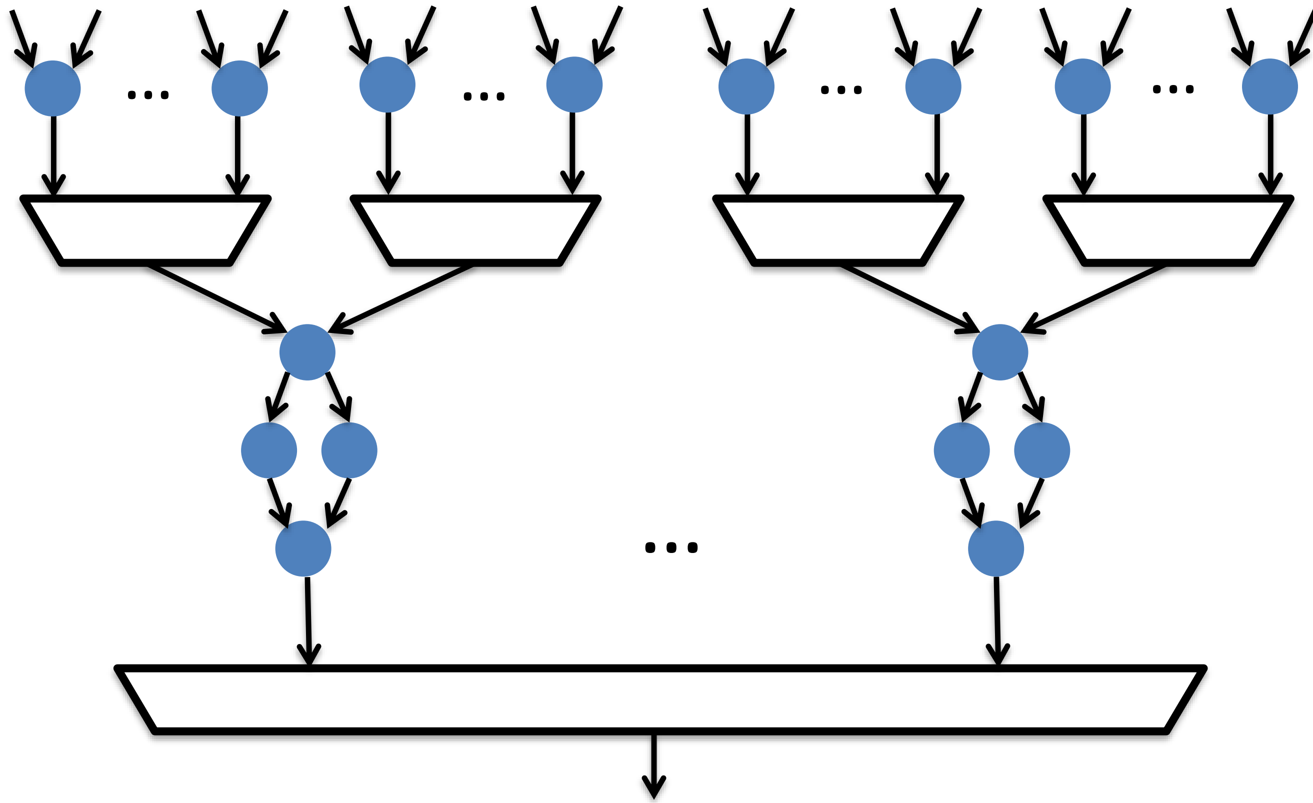
Does the order matter?

Sampling Example

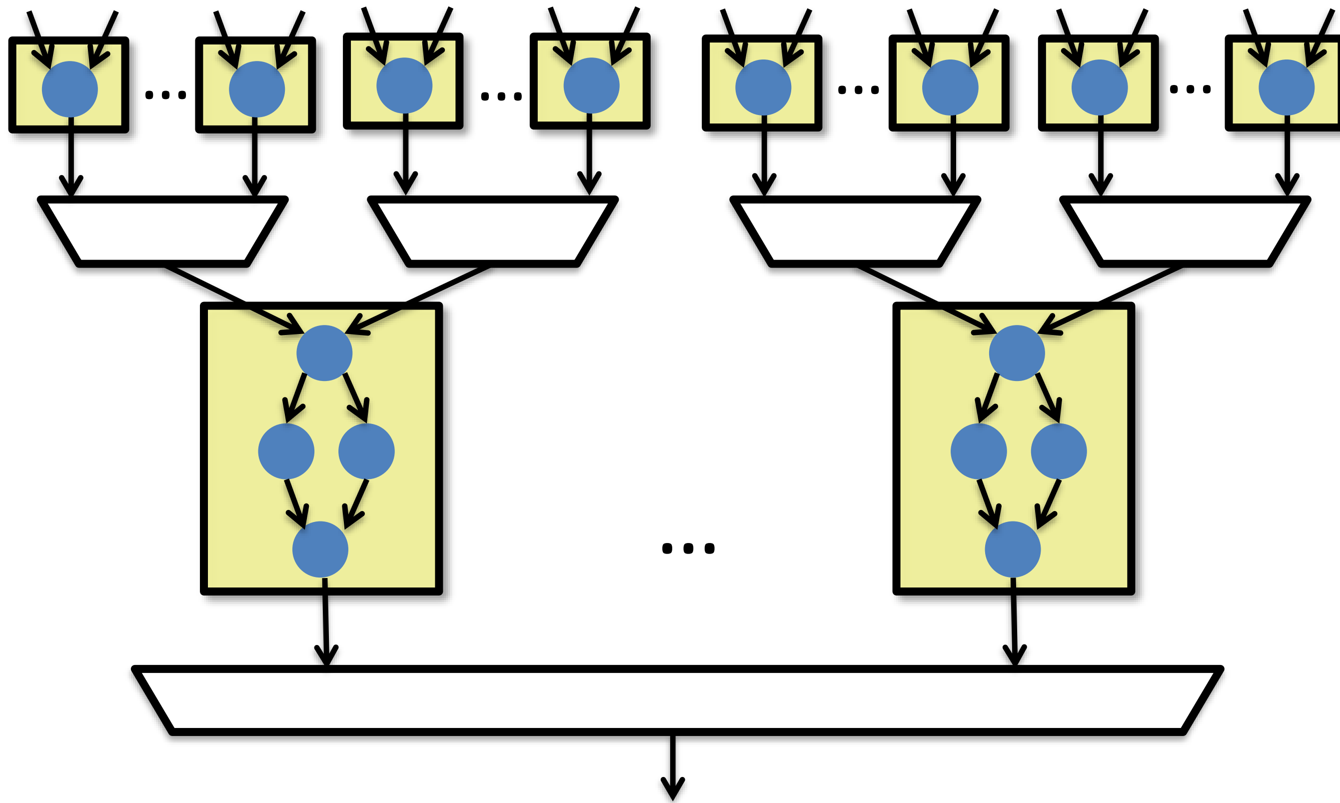
```
s = ...

for (i=0; i < m; i++) {
  if (!coinflip (s/m))
     continue;

  sum = sum + f(i)
}

avg = sum / s
```



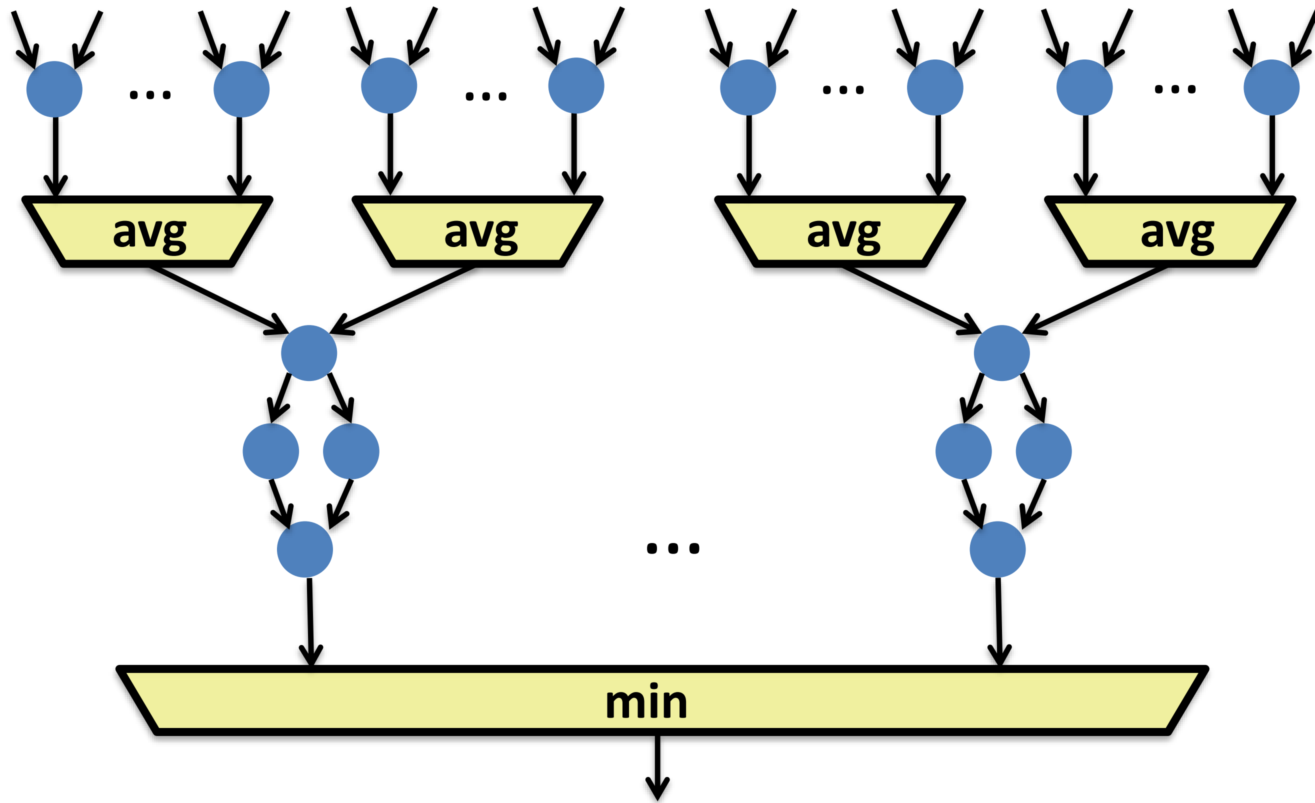
Bringing it all together



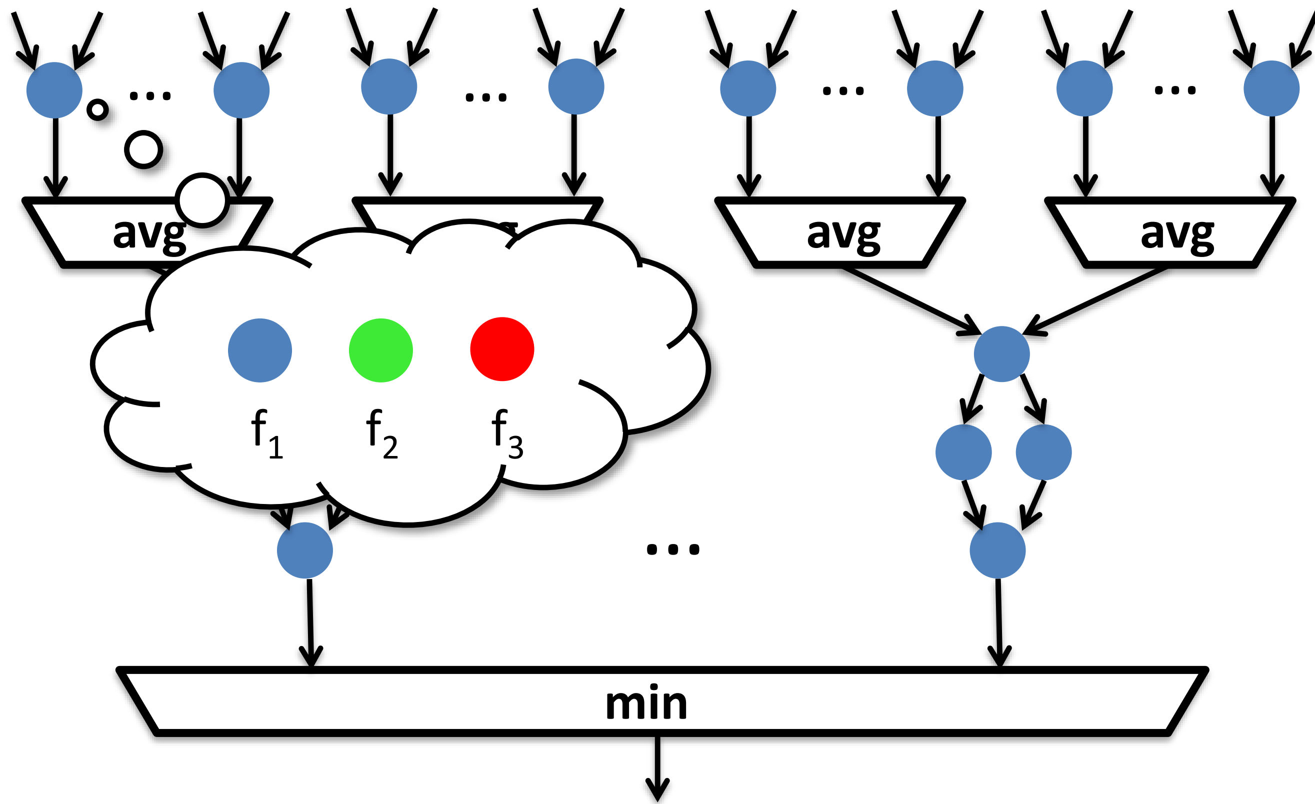
- Nodes represent computation
- Edges represent flow of data



- Functions – process individual data
- Reduction nodes – aggregate data

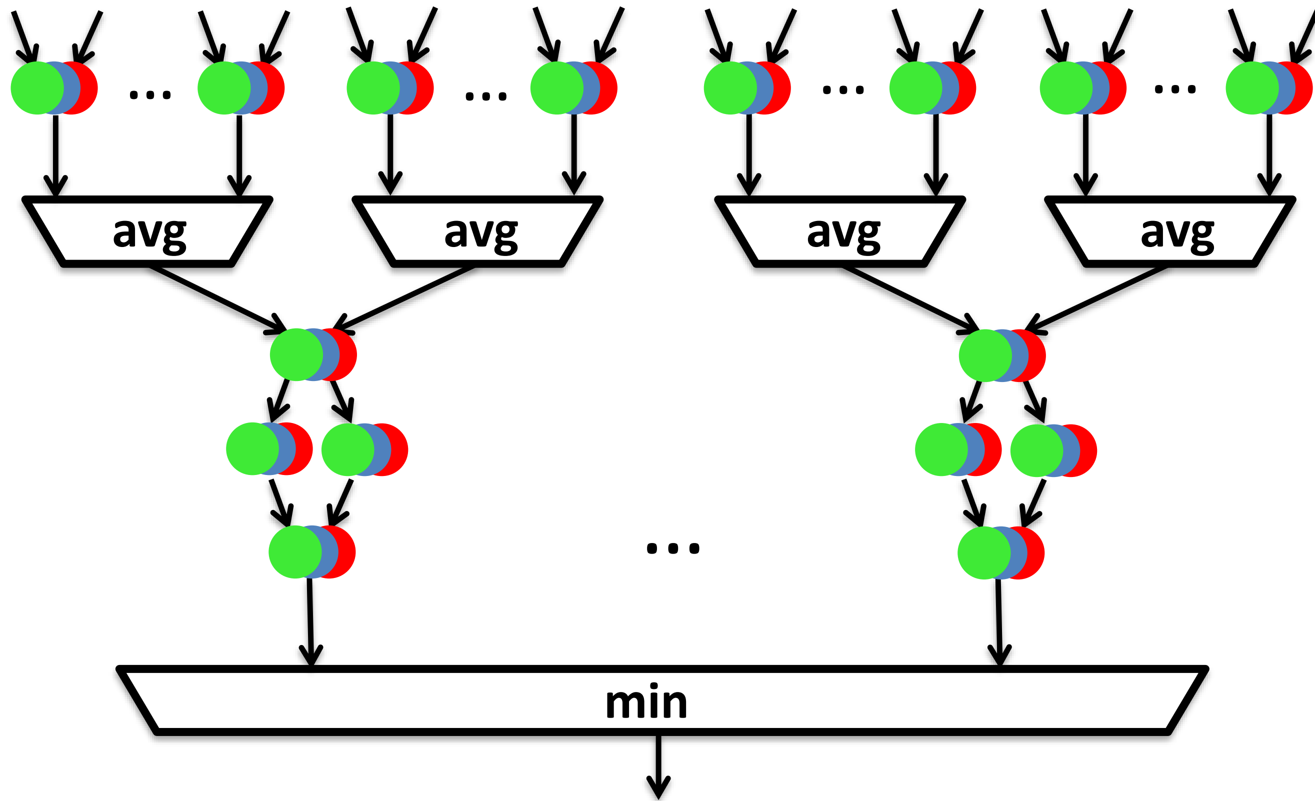


- Functions – process individual data
- Reduction nodes – aggregate data



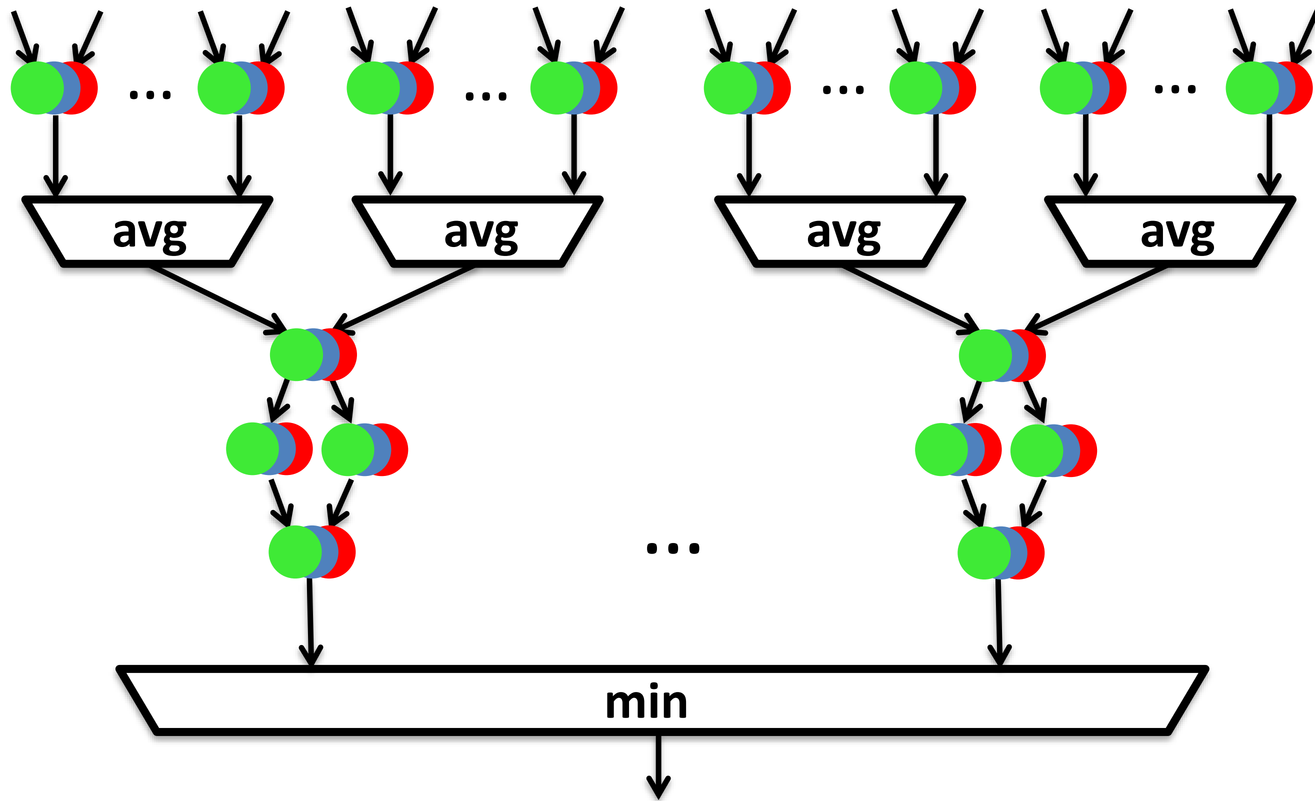
Function substitution

- Multiple implementations
- Each has expected error/time (E, T)



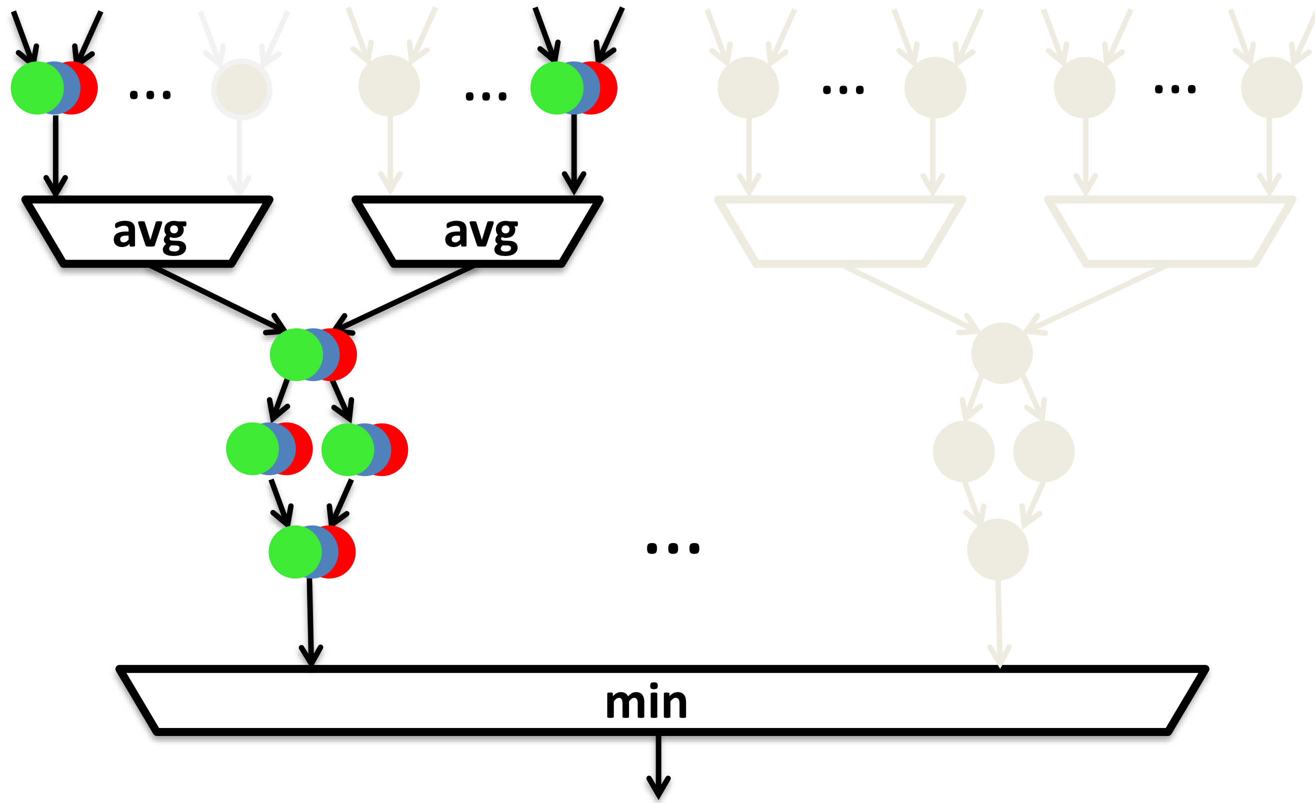
Function substitution

- Multiple implementations
- Each has expected error/time (E, T)



Sampling inputs of reduction nodes

- Reductions consume fewer inputs



Sampling inputs of reduction nodes

- Reductions consume fewer inputs

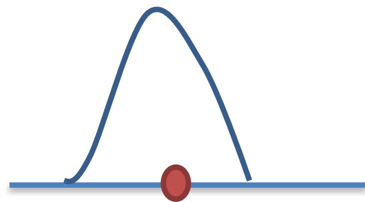
avg avg

...

min

Previously:

Now:



Analysis

- What if we know the distribution of the inputs?

- What if we just know that the procedure is random?

CASE 1: Sum Computation

- Original sum computation

```
s = 0;
for (i = 0; i < n; i++)    s = s + f(i);
```

- Perforated, extrapolated sum computation

```
s = 0;
for (i = 0; i < n; i += 2)    s = s + f(i);
s = s * 2;
```

Step 1: Represent Result Difference

- Original sum computation

```
s = 0;
for (i = 0; i < n; i++)    s = s + f(i);
```

- Perforated, extrapolated sum computation

```
s = 0;
for (i = 0; i < n; i += 2)    s = s + f(i);
s = s * 2;
```

- Perforation noise: $D = s_{original} - s_{perforated}$

Step 2: Probabilistic Modeling

- Original sum computation

```
s = 0;
for (i = 0; i < n; i++)    s = s + f(i);
```

- Perforated, extrapolated sum computation

```
s = 0;
for (i = 0; i < n; i += 2)    s = s + f(i);
s = s * 2;
```

- Perforation noise: $D = s_{original} - s_{perforated}$

Step 2: Probabilistic Modeling

- Original sum computation

```
s = 0;
for (i = 0; i < n; i++)    s = s + X_i;
```

- Perforated, extrapolated sum com

```
s = 0;
for (i = 0; i < n; i += 2)    s = s + X_i;
s = s * 2;
```

- Perforation noise: $D = s_{original} - s_{perforated}$

Analysis: Input/Output Relation

Perforation noise:

$$D = s_{\text{original}} - s_{\text{perforated}}$$

Analysis: Input/Output Relation

Perforation noise:

$$D = s_{\text{original}} - s_{\text{perforated}}$$

$$= \quad X_0 + X_1 + X_2 + X_3 + X_4 + X_5 + X_6 + X_7 + \ldots$$

$$- 2 \cdot (X_0 + X_2 + X_4 + X_6 + \ldots)$$

Analysis: Input/Output Relation

Perforation noise:

$$D = s_{original} - s_{perforated}$$

$$= \quad X_0 + X_1 + X_2 + X_3 + X_4 + X_5 + X_6 + X_7 + \ldots$$

$$- \; X_0 - X_0 - X_2 - X_2 - X_4 - X_4 - X_6 - X_6 - \ldots$$

Analysis: Input/Output Relation

Perforation noise:

$$D = s_{\text{original}} - s_{\text{perforated}}$$

$$= X_0 + X_1 + X_2 + X_3 + X_4 + X_5 + X_6 + X_7 + \ldots$$

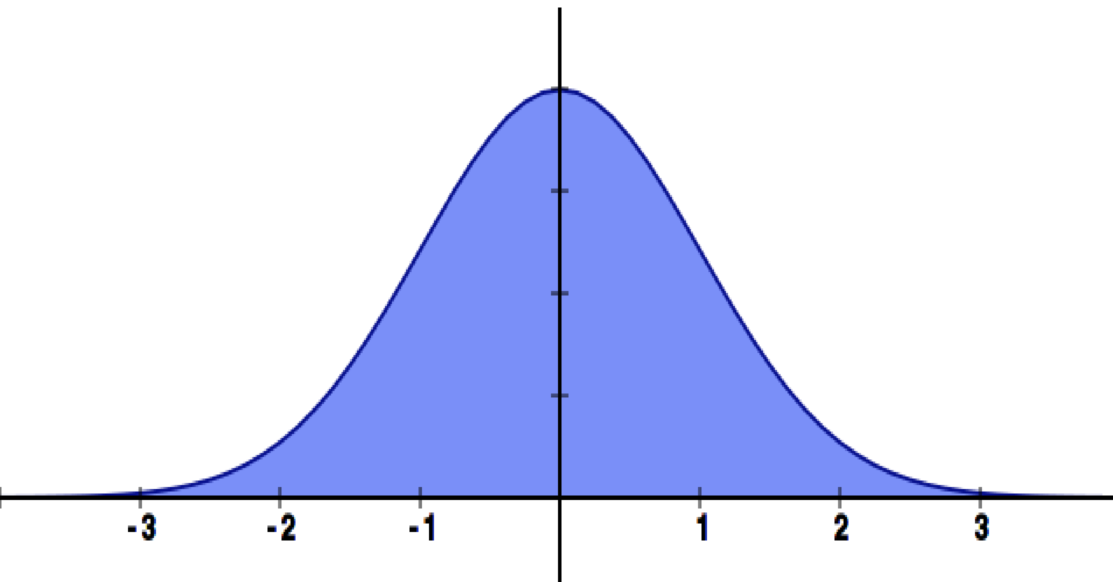
$$- X_0 - X_0 - X_2 - X_2 - X_4 - X_4 - X_6 - X_6 - \ldots$$

$$= \sum_{0 \le i < \frac{n}{2}} (X_{2i+1} - X_{2i})$$

Analysis Results

Perforation noise:

$$D = \varphi(X_0, X_2, \ldots, X_{n-1})$$



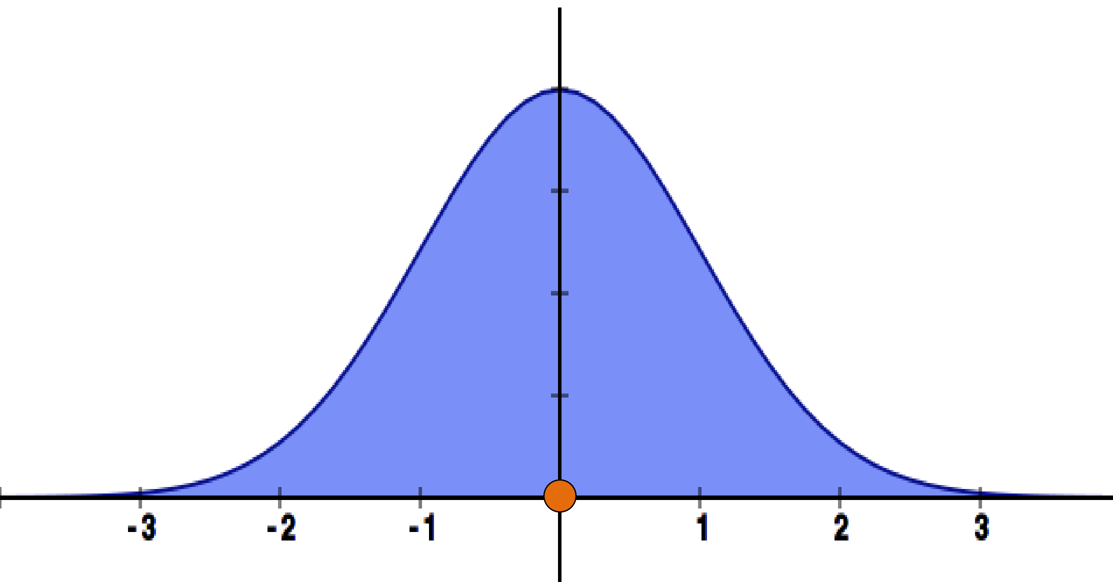
Analysis Results

Perforation noise:

$$D = \varphi(X_0, X_2, \ldots, X_{n-1})$$

Location: Mean

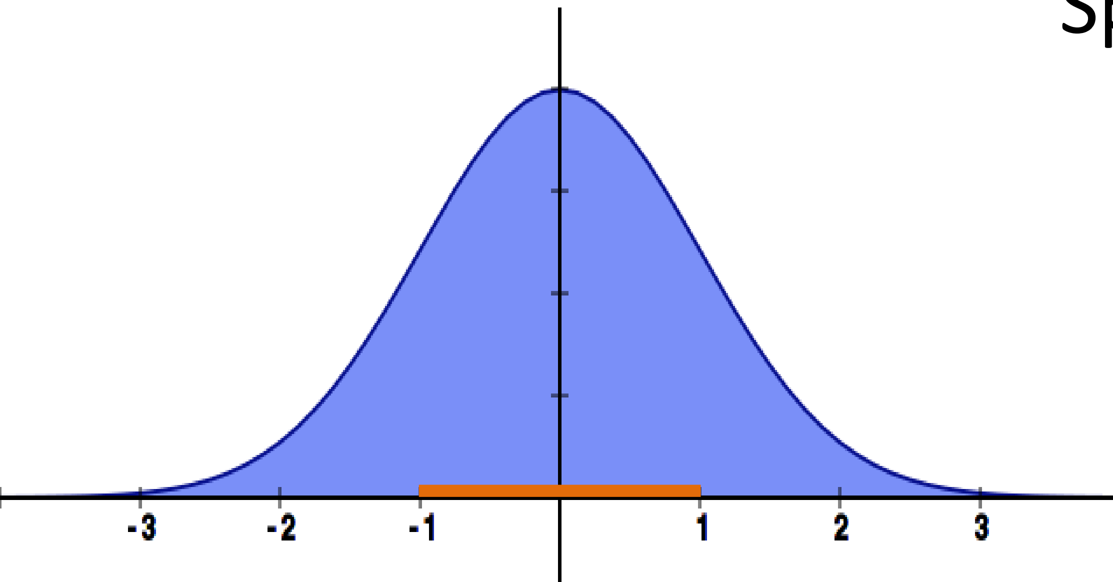
$$E(D) = \mu$$



Analysis Results

Perforation noise:

$$D = \varphi(X_0, X_2, \ldots, X_{n-1})$$



Location: Mean

$$E(D) = \mu$$

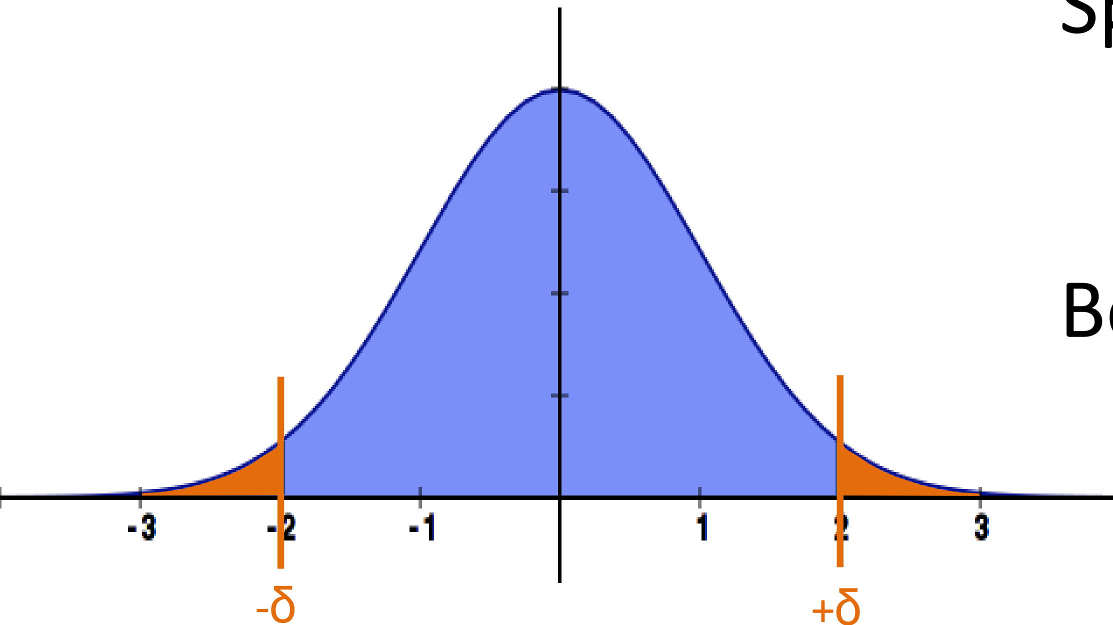
Spread: Variance

$$Var(D) = \sigma^2$$

Analysis Results

Perforation noise:

$$D = \varphi(X_0, X_2, \ldots, X_{n-1})$$



Location: Mean

$$E(D) = \mu$$

Spread: Variance

$$Var(D) = \sigma^2$$

Bound: Distribution tail

$$Pr\left[|D| > \delta \right] < \varepsilon$$

Case 2: Warmup

Case 2: Probabilistic Modeling

- Original sum computation

```
s = 0;
for (i = 0; i < n; i++)    s = s + X_i;
```

- Perforated, extrapolated sum com

```
s = 0;
for (i = 0; i < n; i ++)    s = s + X_i;
s = s * 2;
```

- Then derive in the similar manner as before

SUBLINEAR TIME ALGORITHMS

Property Checking

Main idea: make decisions just by visiting a subset of elements

- Sufficient to distinguish good elements from the clearly bad elements

It will give at most a probabilistic argument, but valid for all input sequences

Repeat multiple times for better effect.

See Ronit Rubenfeld's course on Sublinear time algorithms:
http://www.cs.tau.ac.il/~ronit/COURSES/F14sublin//

Property Checking: Input space

Checking Uniqueness

Input: x1, x2, … xn

Determine between:

1. All xi are distinct
2. The number of distinct elements is < (1-eps)n

Checking Uniqueness

Input: x1, x2, … xn

Determine between:

1. All xi are unique
2. The number of unique elements is < (1-eps)n

Algorithm:

1. Take s samples
2. If any duplicate in the sample, return FALSE
   else return TRUE

Checking Uniqueness

Input: x1, x2, … xn

Determine between:

1. All xi are unique
2. The number of unique elements is < (1-eps)n

Algorithm:

1. Take s samples
2. If any duplicate in the sample, return FALSE
   else return TRUE

Checking Sortedness

Input: x1, x2, …, xn
Bound: eps

Check Sortedness:

1. Select a random number i, 0<i<=n
2. Do a binary search for the element xi
3. If problems during binary search
   return FAIL
4. If ended at position i, return PASS
   else return FAIL
5. Repeat the procedure 2/eps times

Checking Sortedness

Check Sortedness:

1. Select a random number i, 0<i<=n
2. Do a binary search for the element xi
3. If problems during binary search
   return FAIL
4. If ended at position i, return PASS
   else return FAIL
5. Repeat the procedure 2/eps times

Time: O(log(n)/eps)

Accuracy: if input likely to pass the test, then at least
(1-eps)n elements are sorted with probability 2/3

Checking Sortedness

Time: O(log(n)/eps)

Accuracy: if input likely to pass the test, then at least (1-eps)n elements are sorted with probability 2/3

How useful is bound 2/3 ?

Can we get better probability?