# 15-150

# Principles of Functional Programming

Slides for Lecture 19

## Parallelism, Cost Graphs, Sequences

April 1, 2025

Michael Erdmann

# Lessons:

- Cost Semantics / Cost Graphs
- Brent's Theorem
- Sequences

# Parallelism:

Performing multiple computations simultaneously.

# Scheduling:

Telling each processor what to do when.

This course focuses on *deterministic parallelism*:

- We allow *independent expressions* in a program to evaluate in parallel.
- We require parallel evaluation to have *well-defined behavior*.
- We do not worry explicitly about scheduling, but we use ***cost semantics*** to write code that facilitates parallelism.

(Functional programming languages without side-effects facilitate this approach.)

## What can a programmer do to facilitate parallelism?

- Write code that does **not** *bake in* a schedule. (Lists bake in sequential evaluation. Trees facilitate parallelism. Today we will introduce an abstract datatype called ***sequences***. Sequences have a linear structure like lists but support the parallelism of trees.)

- Reason about time complexity (Work & Span) to write fast parallel code. (You have been doing that with recurrences. Today we will introduce ***cost graphs*** as another tool.)

# Cost Graphs

**Cost graphs** are a form of *series-parallel* graph.

Such a graph is a directed acyclic graph,
with designated ***source*** and ***sink*** nodes.

(***Source*** means there are no incoming edges.
***Sink*** means there are no outgoing edges.)
We draw graphs with source at top and sink at bottom.
All edges directed downward.)

We will use cost graphs to model computations
and to compute Work and Span.

# Basic Constructions

Base Case:  (single node, source=sink, modeling no computation)

# Basic Constructions

**Base Case:** ● (single node, source=sink, modeling no computation)

**Sequential Composition:**

$$\begin{array}{c} \mathbf{G_1} \\ \downarrow \\ \mathbf{G_2} \end{array}$$

(Edge from $\mathbf{G_1}$'s sink to $\mathbf{G_2}$'s source, modeling sequential computation: perform $\mathbf{G_1}$'s computation, then $\mathbf{G_2}$'s.)

# Basic Constructions

Base Case:

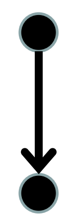
(single node, source=sink, modeling no computation)

Sequential Composition:

$$\begin{tikzcd} \mathbf{G_1} \arrow[d] \\ \mathbf{G_2} \end{tikzcd}$$

(Edge from $\mathbf{G_1}$'s sink to $\mathbf{G_2}$'s source, modeling sequential computation)

Special case:



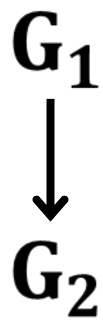
(one evaluation step)

# Basic Constructions

Base Case:

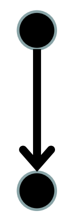
(single node, source=sink, modeling no computation)

Sequential Composition:



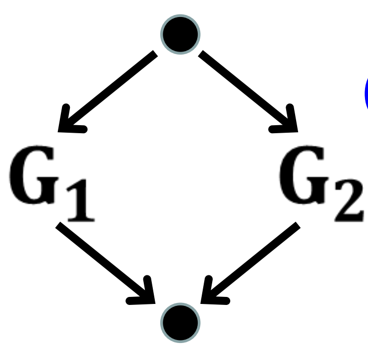

(Edge from $\mathbf{G_1}$'s sink to $\mathbf{G_2}$'s source, modeling sequential computation)

Special case:



(one evaluation step)

Parallel Composition:



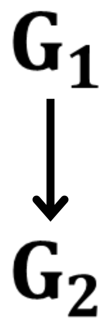

(Fork and Join: new source with edges to original sources of $\mathbf{G_1}$ and $\mathbf{G_2}$, then edges from their sinks to a new sink. Models parallel computation.)

# Basic Constructions

Base Case:

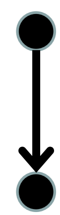
(single node, source=sink, modeling no computation)

Sequential Composition:




(Edge from $\mathbf{G_1}$'s sink to $\mathbf{G_2}$'s source, modeling sequential computation)

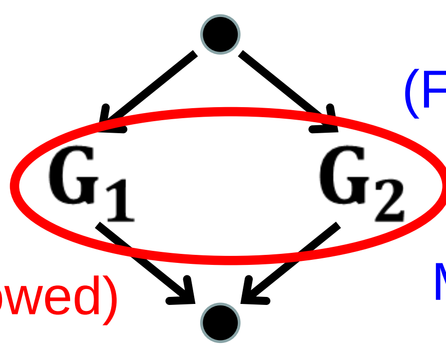
Special case:



(one evaluation step)

Parallel Composition:

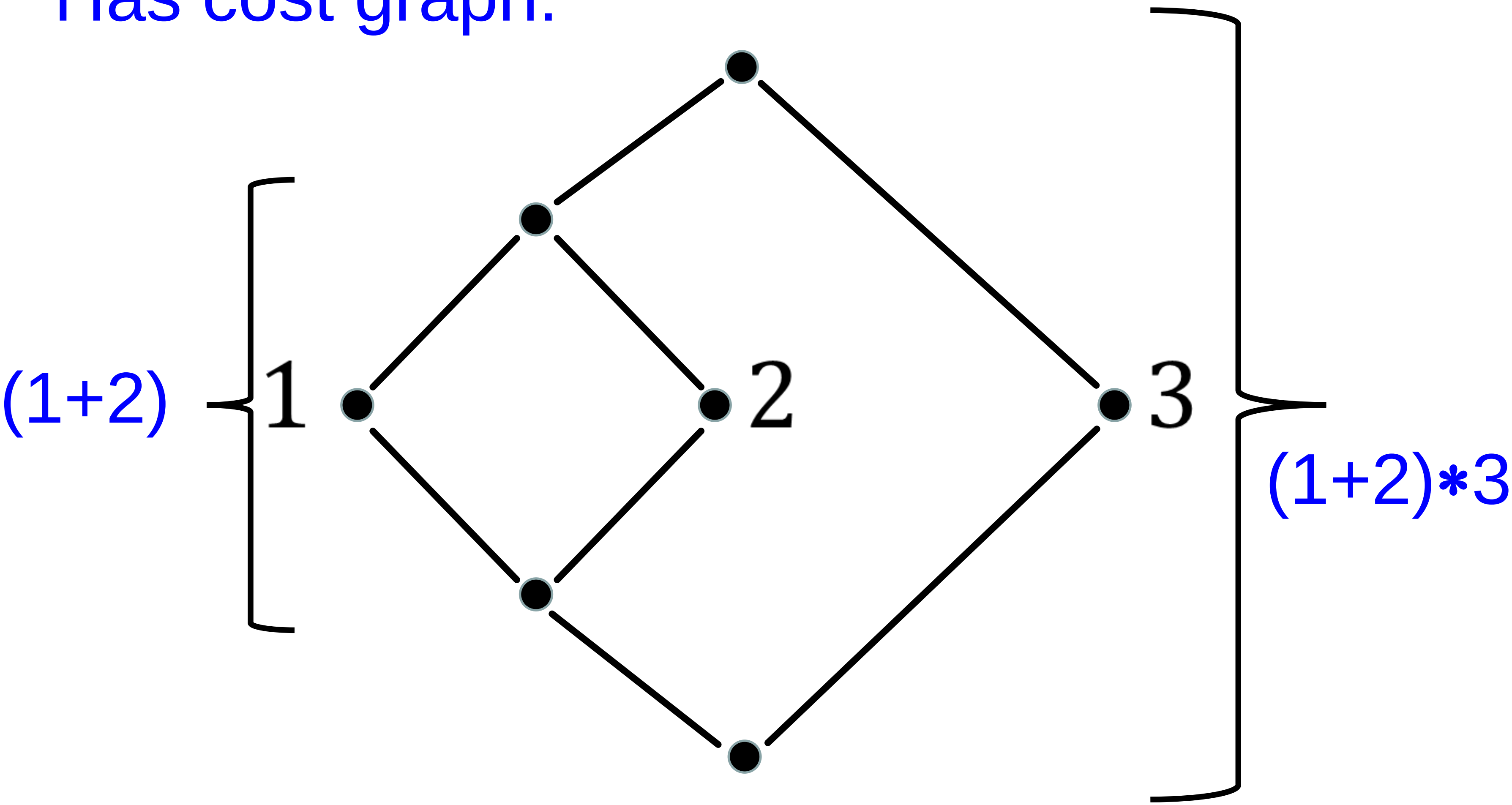
(n-ary parallelism allowed)




(Fork and Join: new source with edges to original sources of $\mathbf{G_1}$ and $\mathbf{G_2}$, then edges from their sinks to a new sink. Models parallel computation.)

Example: $(1+2) * 3$

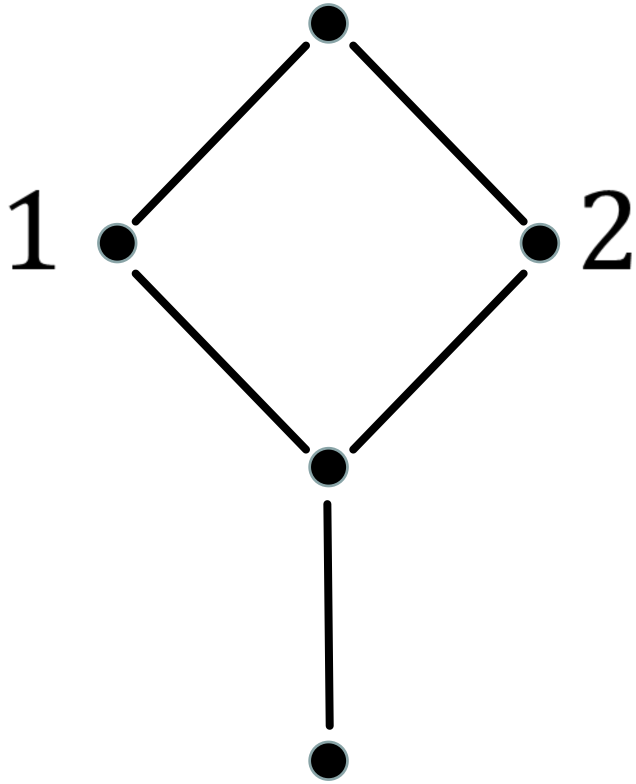
Has cost graph:



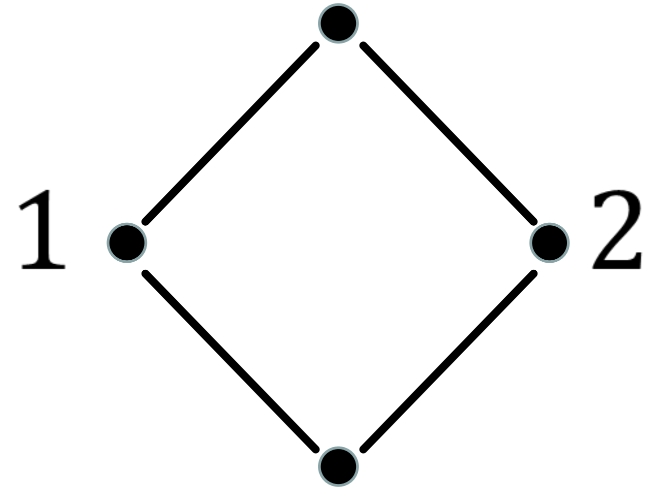

(Edges are implicitly directed downward.)

# We are being a little sloppy but it is fine.

Technically, $(1 + 2)$ has cost graph:




We elide that to:

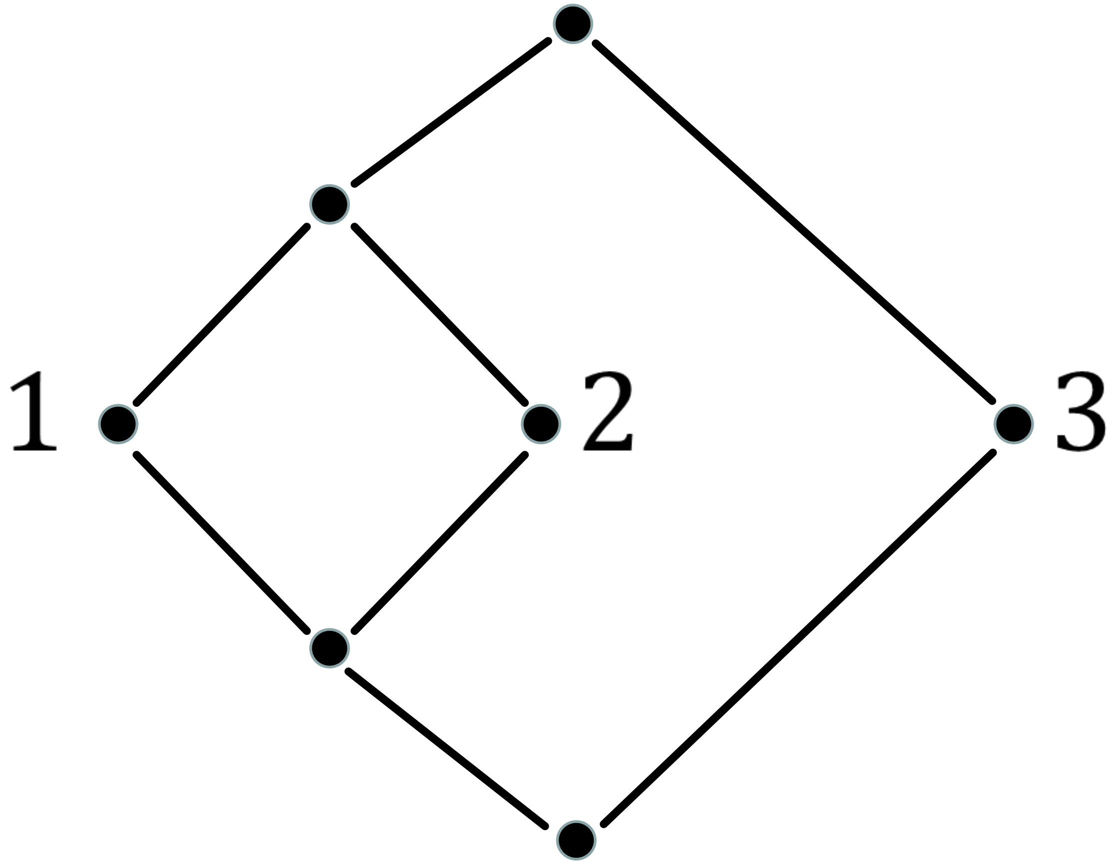


# Work and Span

- We define the ***work of a cost graph*** **G** to be the **number of nodes** in **G**.
- We define the ***span of a cost graph*** **G** to be the number of nodes on the **longest path** from **G**’s source to **G**’s sink.
- We now **re-define** the **work** and **span** of an expression **e** to be the work and span of the cost graph **G** representing **e**.

(These numbers differ by constant factors/terms from our earlier definitions, but will be the same asymptotically.)

Example: $(1 + 2) * 3$




Work = 7

Span = 5

# Brent's Theorem

An expression `e` with work ***W*** and span ***S*** can be evaluated on a ***p***-processor machine in time $\Omega(\max(W/p, S))$.

like "big-O" but now a lower bound

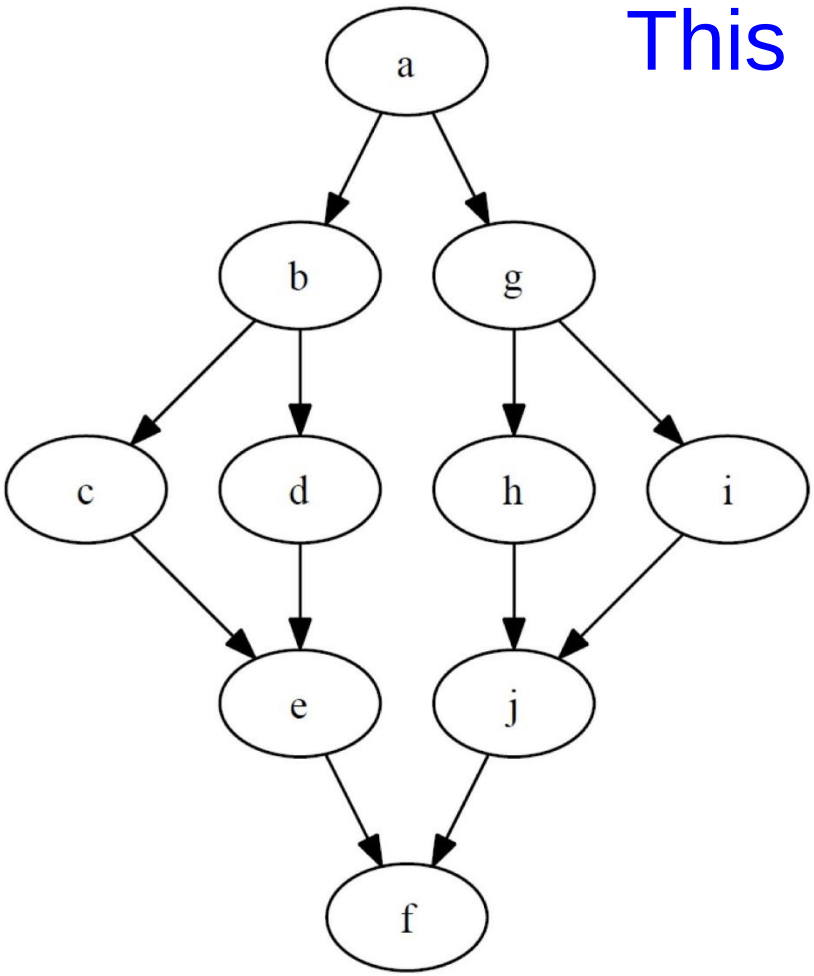
(we pretend it is an approximate equality)

# Scheduling

- (This is a bit of side-topic, just to show you how one might use cost graphs to schedule.)
- We will use *pebbling*:
  - $p$ pebbles, with $p$ the number of processors.
  - Start with one pebble on cost graph $\mathbf{G}$'s source.
  - Putting a pebble on a node *visits* the node.
  - At each time step, pick up all pebbles and put at most $p$ on the graph, no more than one per node. Can only put a pebble on an unvisited node all of whose ancestors have been visited.

(There are various kinds of pebbling strategies.)

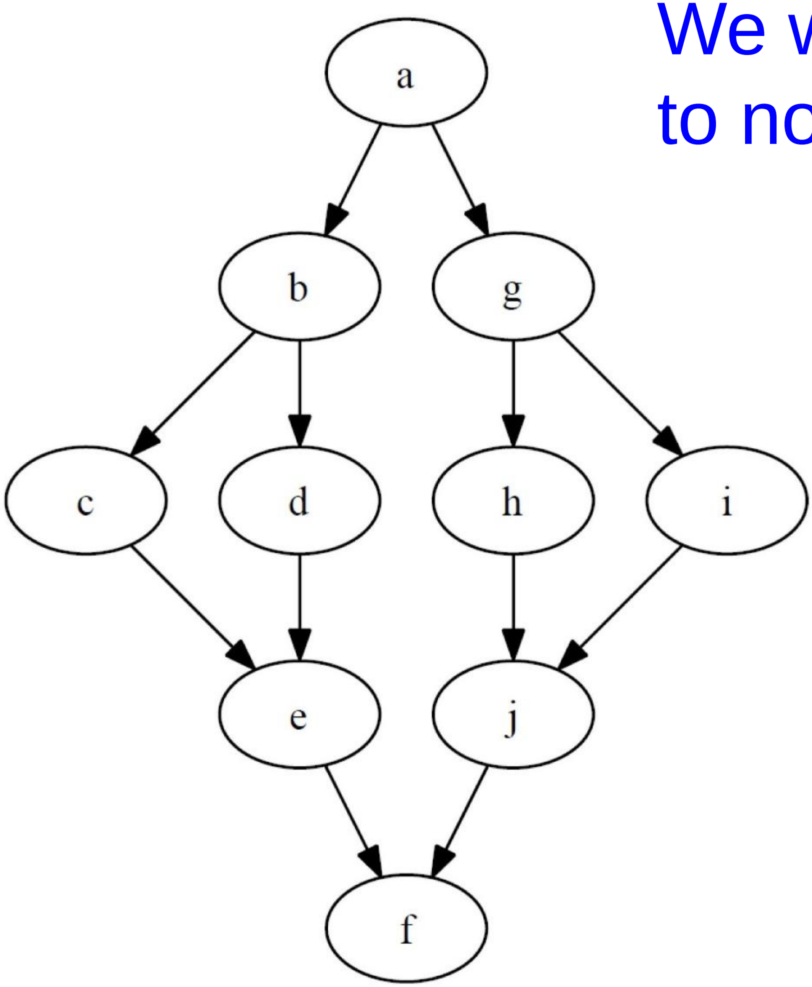
# Breadth-First Pebbling Algorithm




This might be a cost graph for

$$(1 + 2) * (3 + 4)$$

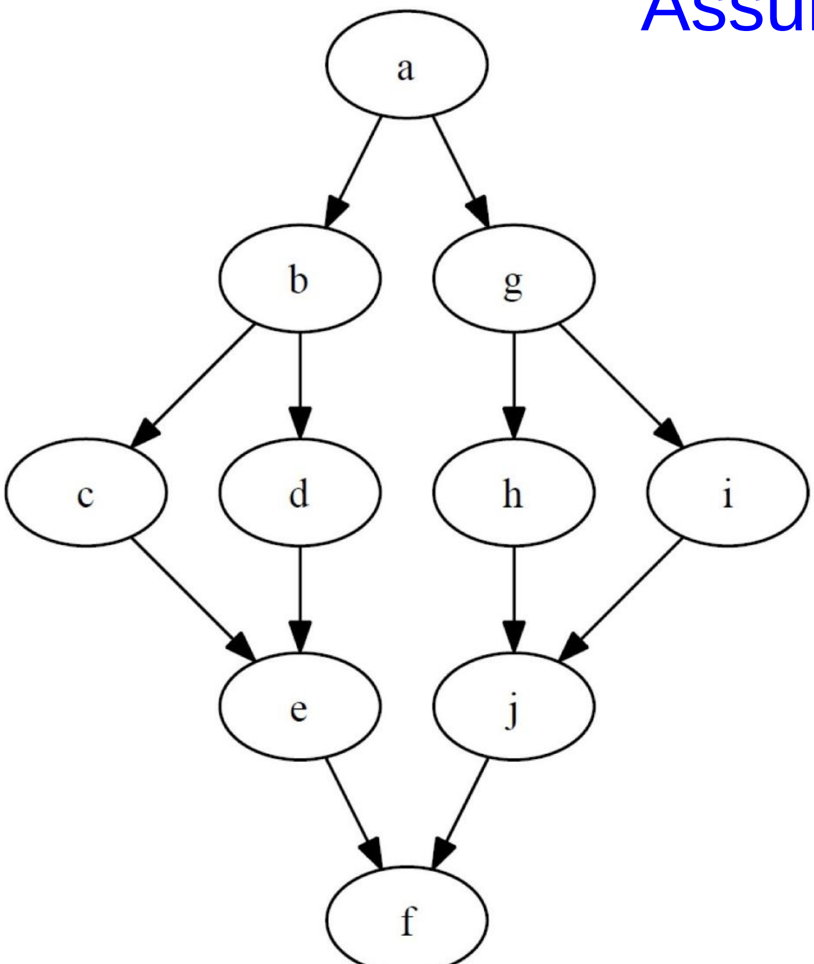
# Breadth-First Pebbling Algorithm




We wish to assign processors
to nodes at successive time steps.

[At each time step, the
processor assigned to a node
will perform the computation
represented by the node and
its incident edges
(e.g., fork, join, arithmetic).]

# Breadth-First Pebbling Algorithm

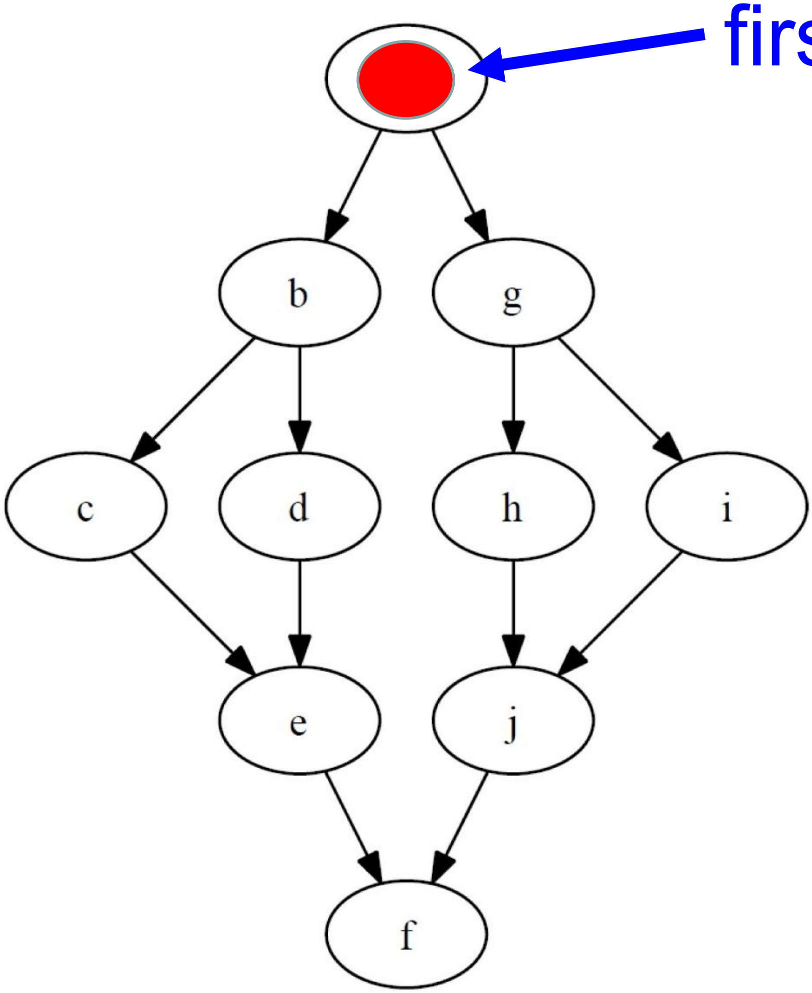



Assume we have 2 processors.

processors

| time | | 1 | 2 |
|---|---|---|---|
| | 1 | | |
| | 2 | | |
| | 3 | | |
| | 4 | | |
| | 5 | | |
| | 6 | | |

# Breadth-First Pebbling Algorithm

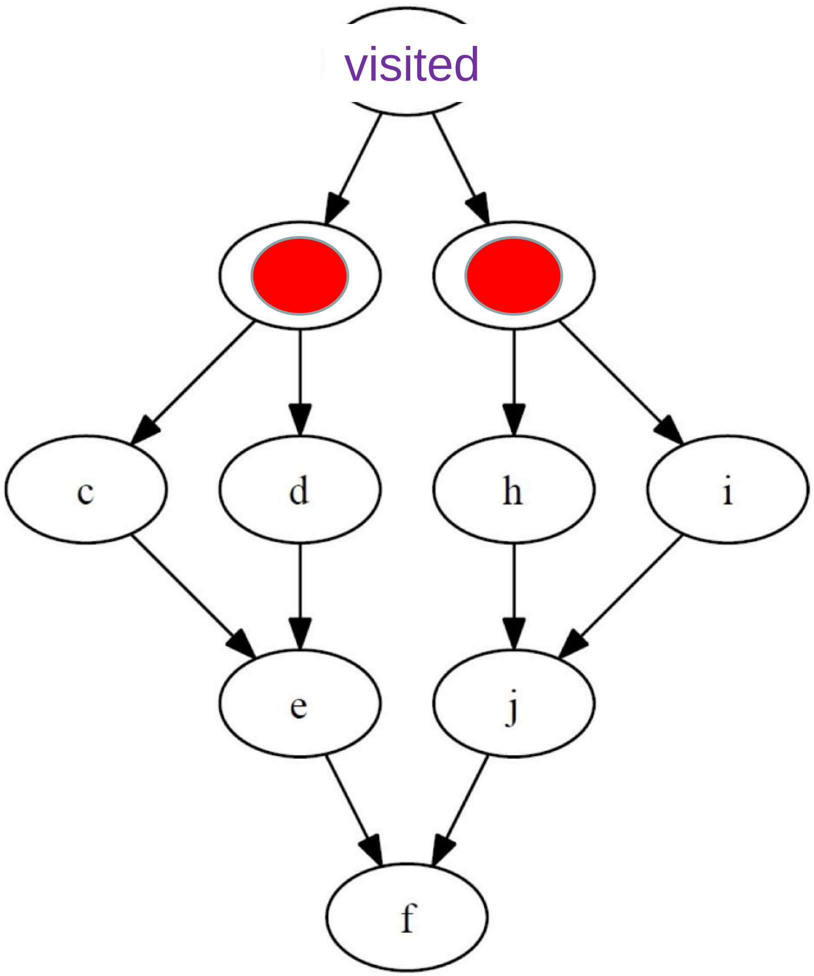



first pebble on node “a”

processors

| time | 1 | 2 |
|---|---|---|
| 1 | a | (idle) |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |
| 6 | | |

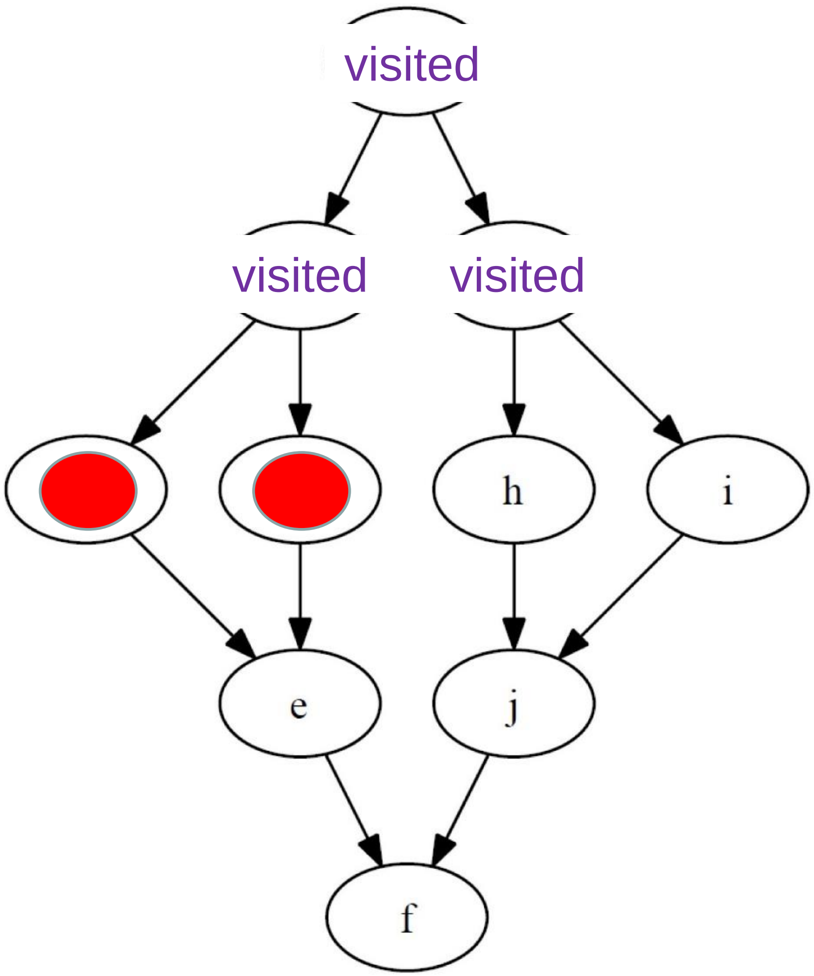
# Breadth-First Pebbling Algorithm




processors

| time | 1 | 2 |
|---|---|---|
| 1 | a | |
| 2 | b | g |
| 3 | | |
| 4 | | |
| 5 | | |
| 6 | | |

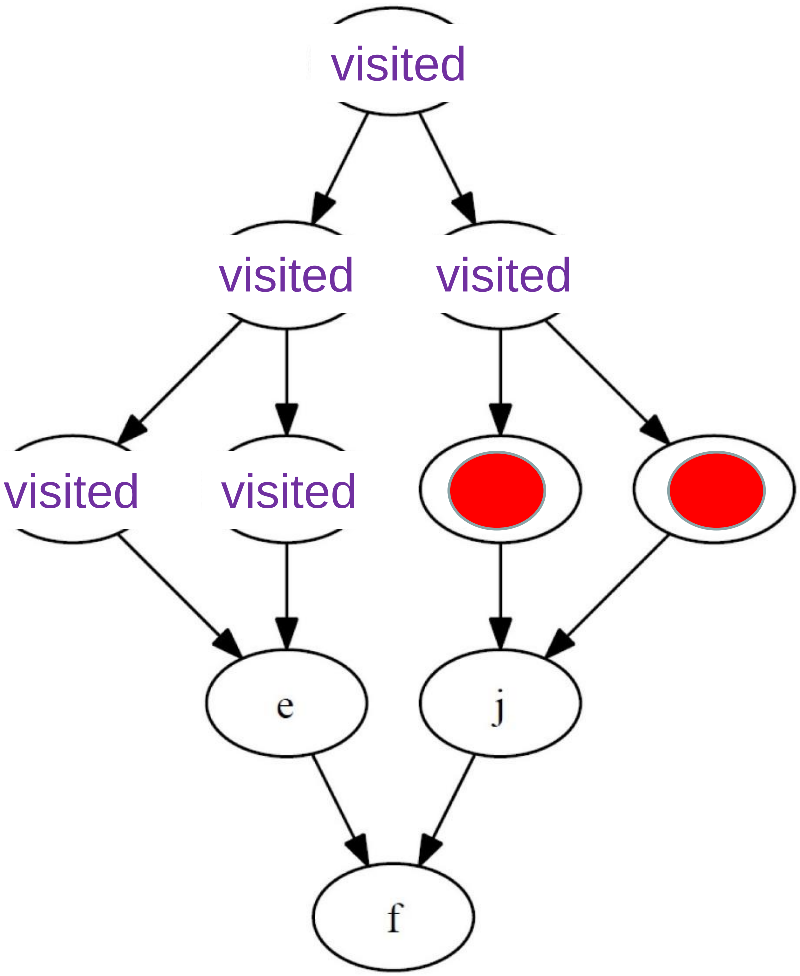
# Breadth-First Pebbling Algorithm




processors

| time | 1 | 2 |
|---|---|---|
| 1 | a | |
| 2 | b | g |
| 3 | c | d |
| 4 | | |
| 5 | | |
| 6 | | |

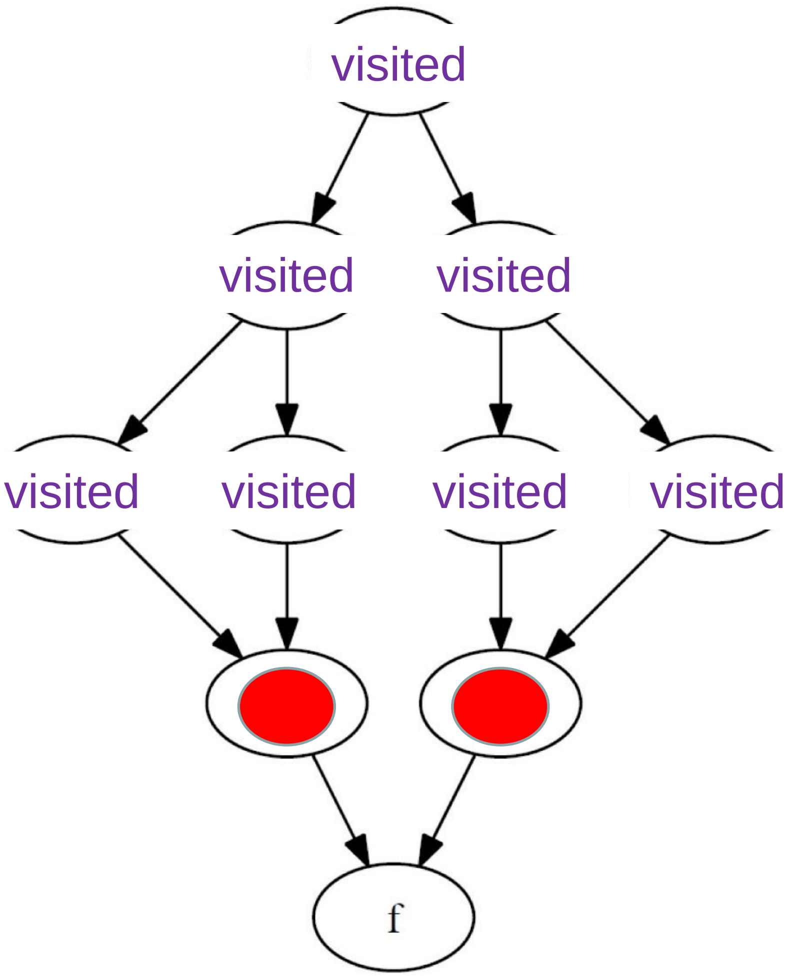
# Breadth-First Pebbling Algorithm




processors

| time | 1 | 2 |
|---|---|---|
| 1 | a | |
| 2 | b | g |
| 3 | c | d |
| 4 | h | i |
| 5 | | |
| 6 | | |

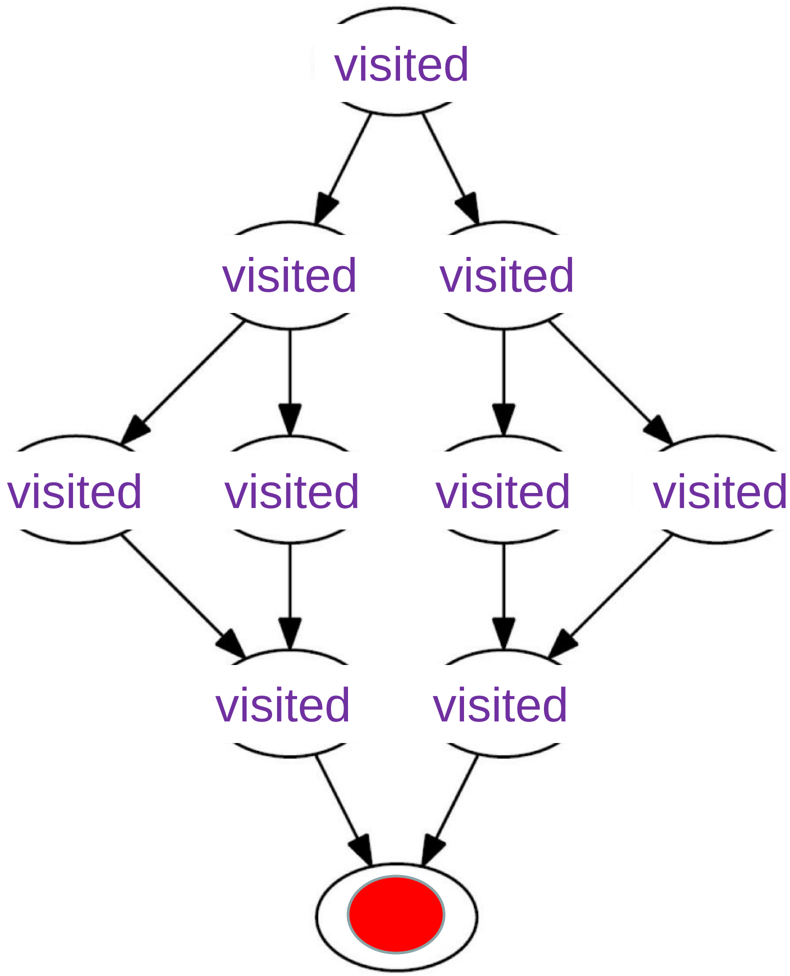
# Breadth-First Pebbling Algorithm




processors

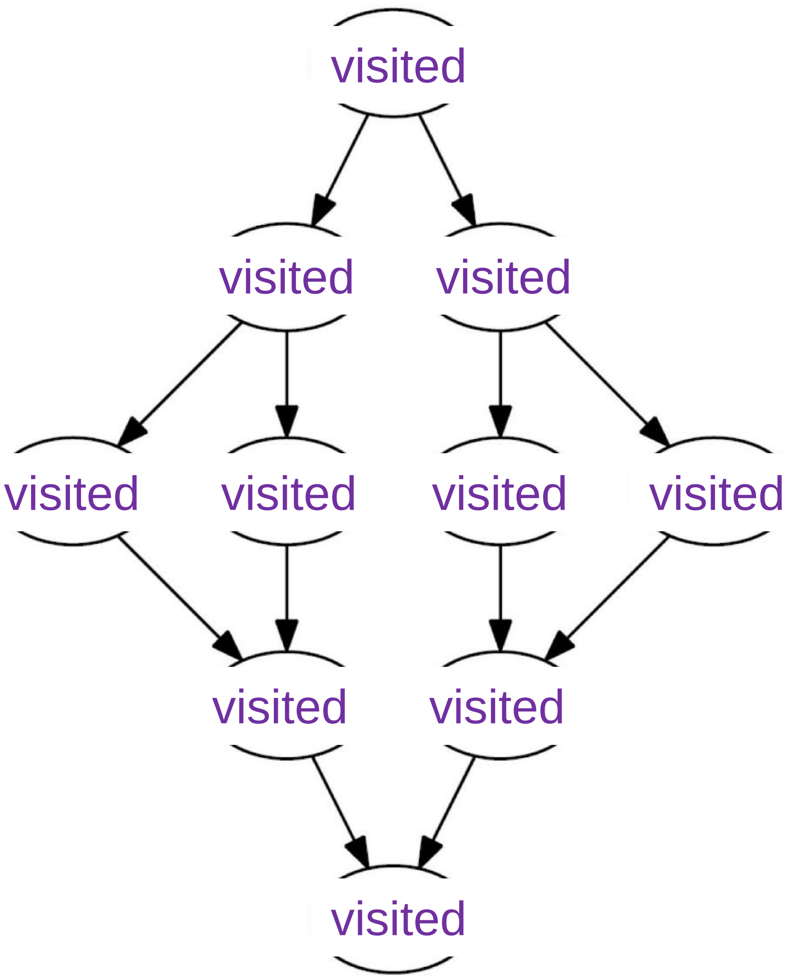
| time | 1 | 2 |
|---|---|---|
| 1 | a | |
| 2 | b | g |
| 3 | c | d |
| 4 | h | i |
| 5 | e | j |
| 6 | | |

# Breadth-First Pebbling Algorithm




| time | processors 1 | processors 2 |
|---|---|---|
| 1 | a | |
| 2 | b | g |
| 3 | c | d |
| 4 | h | i |
| 5 | e | j |
| 6 | | f |

Work = 10, Span = 5, so Brent's Theorem suggests
$5 = \max(10/2, 5)$ time steps might be possible.
But we have some idle times, so get $6$.
(Also: Brent's Theorem is only an asymptotic assertion.)




processors

| time | 1 | 2 |
|---|---|---|
| 1 | a | |
| 2 | b | g |
| 3 | c | d |
| 4 | h | i |
| 5 | e | j |
| 6 | | f |

# Sequences

- We will present (part of the) **`SEQUENCE`** signature.
- We will describe the **work** and **span** of some sequence functions via **cost graphs**.
- Sequences are abstract. Hidden implementation.
- For reasoning purposes, we write a sequence of length $\mathbf{n}$ containing elements $\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}$ as

  $$\langle \mathbf{x}_0, \ldots, \mathbf{x}_{n-1} \rangle .$$
- Two sequence values are extensionally equivalent iff they have the same length and contain extensionally equivalent values at corresponding positions.

```
signature SEQUENCE =
sig
  type 'a seq      (* abstract *)

  exception Range of string

  val empty : unit -> 'a seq

  val tabulate : (int -> 'a) -> int -> 'a seq
  val length : 'a seq -> int
  val nth    : 'a seq -> int -> 'a

  val map : ('a -> 'b) -> 'a seq -> 'b seq
  val reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a

  val mapreduce :
          ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a seq -> 'b

  val filter : ('a -> bool) -> 'a seq -> 'a seq

  ...
end
```

Most of those functions should seem familiar from lists.

One difference is that instead of `foldr` and `foldl` we now have `reduce`. We will talk more about that.

You probably never used `List.tabulate`. We will discuss `tabulate` for sequences.

**Unlike lists, sequences support parellization, giving good span costs for many functions.**

# sequence type

---

$$\texttt{<}\mathbf{x}_0\texttt{, ..., }\mathbf{x}_{n-1}\texttt{>} : \mathbf{t\ seq}$$

if $\mathbf{x}_i : \mathbf{t}$,

for $\mathbf{i} = 0, \ldots, \mathbf{n-1}$.

# sequence type

$\texttt{<x}_0\texttt{, ..., x}_{n-1}\texttt{>}$ : **t seq**

Reminder: A client would write something like

**t Seq.seq**

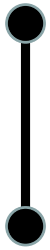
given structure **`Seq`** ascribing to signature **`SEQUENCE`**.

# `empty`

## `empty ()`

returns a sequence of length `0`, containing no elements.

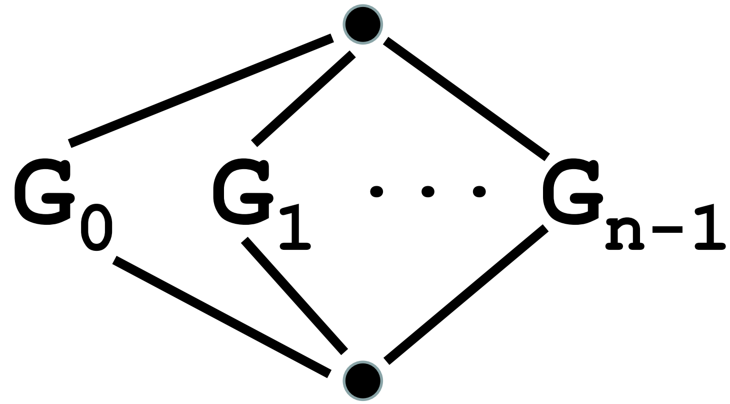
The type can be `t seq`, for any type `t`.

Cost Graph:  So O(1) work and span.

# `tabulate`

`tabulate f n` $\cong$ `<f(0), …, f(n-1)>`

Cost Graph:



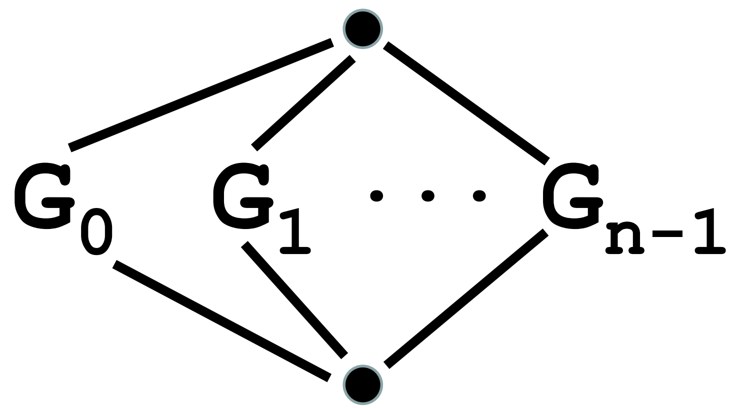

Here $G_i$ is the cost graph for evaluating `f(i)`.

If `f(i)` has O(1) work and span for all `i`, then `tabulate f n` has O(n) work and O(1) span.

# tabulate

**tabulate f n** ≅ **<f(0), …, f(n-1)>**

Cost Graph:



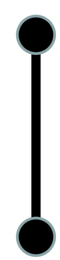

Here $G_i$ is the cost graph for evaluating **f(i)**.

IF **f(i)** has O(1) work and span for all **i**, then **tabulate f n** has O(n) work and O(1) span.

# nth

$\texttt{nth}\ \langle x_0, \ldots, x_{n-1}\rangle\ \texttt{i} \cong x_i$, if $0 \le \texttt{i} < \texttt{n}$,

raises **Range** otherwise.

Cost Graph:  So O(1) work and span.

In other words, constant time access to elements (unlike lists).

# length

$\texttt{length}\ \langle x_0,\ \ldots,\ x_{n-1}\rangle \cong n$ .

Cost Graph:

Again, O(1) work and span.

# `length`

$\texttt{length}\ \langle x_0, \ldots, x_{n-1} \rangle \cong n$ .

Cost Graph:  Again, O(1) work and span.

Question: How could one achieve this?

# `length`

$\texttt{length} \ \langle x_0, \ldots, x_{n-1} \rangle \cong n$.

Cost Graph:  Again, O(1) work and span.

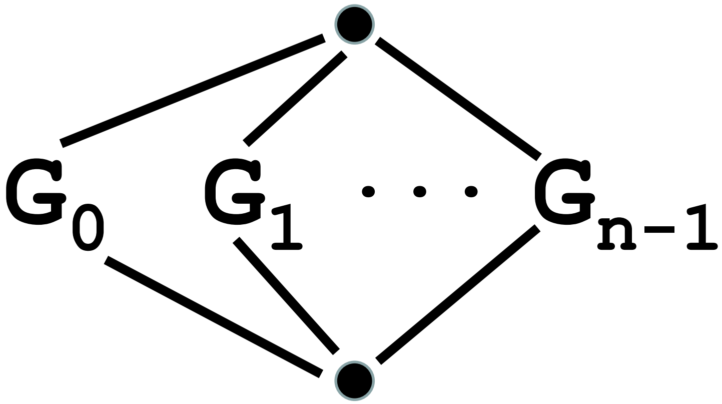
Question: How could one achieve this?

Answer: Keep track of length explicitly
in the underlying representation of sequences.

# `map`

$\texttt{map f <x}_0\texttt{,...,x}_{n-1}\texttt{>} \cong \texttt{<f x}_0\texttt{ ,..., f x}_{n-1}\texttt{>}$

Cost Graph:




Here $G_i$ is the cost graph for evaluating $\texttt{f(x}_i\texttt{)}$.

If `f(x)` has O(1) work and span for all `x`, then $\texttt{map f <x}_0\texttt{,...,x}_{n-1}\texttt{>}$ has O(n) work & O(1) span.

# `reduce`

Recall the type:

```
reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a
```

That is more restrictive than the type of **`foldr`** was:

```
foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

Let’s explore that.

`reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a`

# reduce

---

$$\texttt{reduce g z <}x_0,\ldots,x_{n-1}\texttt{>} \cong x_0 \odot \cdots \odot x_{n-1} \odot z$$

We assume that **g** is *associative*, meaning **g(g(x,y),w)** ≅ **g(x,g(y,w))**, for all values **x**,**y**,**w** of the correct type. So no parentheses are needed on the right, where we represent **g** by the infix operator ⊙.

[In 15-210 you will generally assume as well that **z** is an *identity* (also called a *zero*) for **g**, meaning **g(x,z)** ≅ **x** ≅ **g(z,x)**, for all values **x** of the correct type.]

Then:

$$\texttt{reduce g z <}x_0,\ldots,x_{n-1}\texttt{>} \cong x_0 \odot \cdots \odot x_{n-1}$$

$$\texttt{reduce g z <>} \cong \texttt{z}$$

`reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a`

# `reduce`

$$\texttt{reduce g z <}x_0,\ldots,x_{n-1}\texttt{>} \cong x_0 \odot \cdots \odot x_{n-1} \odot z$$

We assume that **`g`** is *associative*, meaning **`g(g(x,y),w) ≅ g(x,g(y,w))`**, for all values **`x`**,**`y`**,**`w`** of the correct type. So no parentheses are needed on the right, where we represent **`g`** by the infix operator $\odot$.

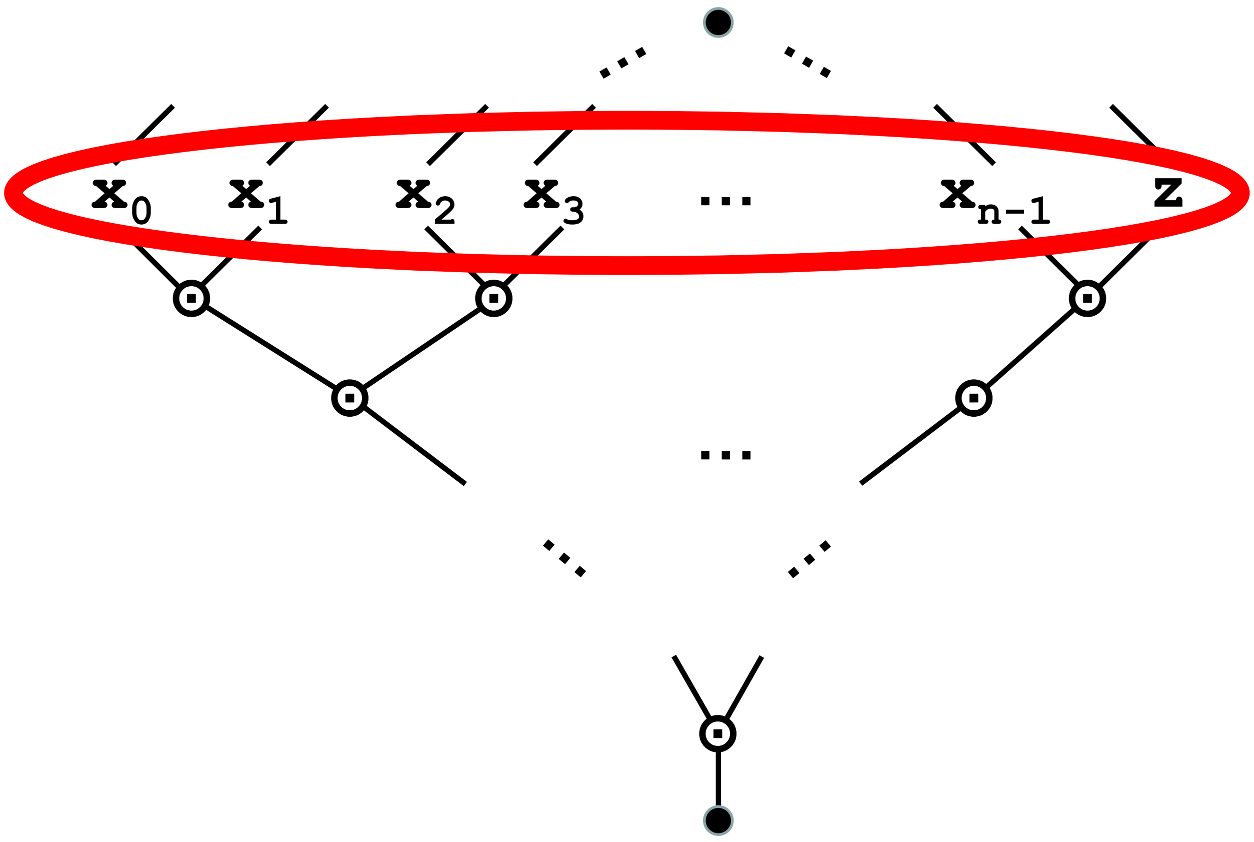
[In 15-210 you will generally assume as well that **`z`** is an *identity* (also called a *zero*) for **`g`**, meaning **`g(x,z) ≅ x ≅ g(z,x)`**, for all values **`x`** of the correct type. We do that sometimes in 15-150 but it can be useful to allow more general **`z`** (thus mimicking a list **`foldr`**).]

# reduce

$\texttt{reduce g z <}x_0,\ldots,x_{n-1}\texttt{>} \cong x_0 \odot \cdots \odot x_{n-1} \odot z$

Cost Graph:

(forking abbreviated)



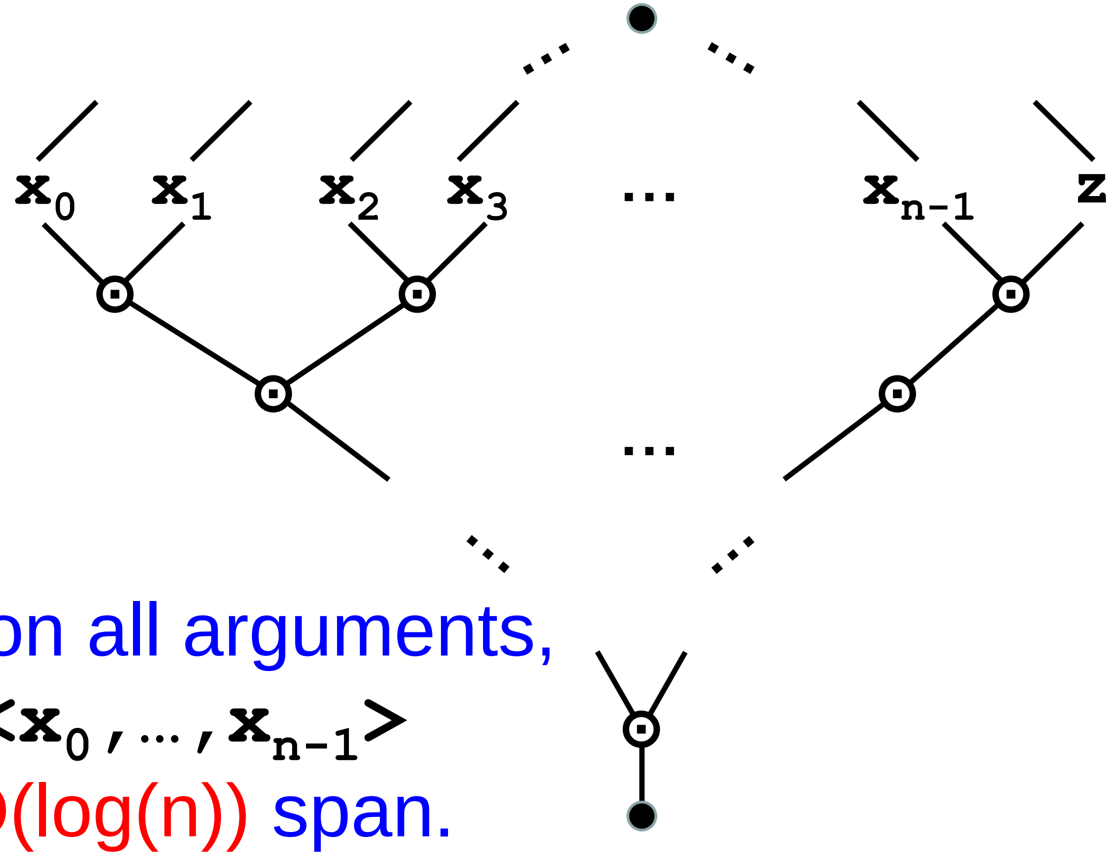
# `reduce`

$\texttt{reduce g z <}x_0,\ldots,x_{n-1}\texttt{>} \cong x_0 \odot \cdots \odot x_{n-1} \odot z$

Cost Graph:

(forking abbreviated)

O(log(n)) levels)




If **g** is constant time on all arguments, then $\texttt{reduce g z <}x_0,\ldots,x_{n-1}\texttt{>}$ has O(n) work and O(log(n)) span.

# mapreduce

**mapreduce** combines **map** and **reduce**:

$$\texttt{mapreduce f z g } \langle \texttt{x}_0, \ldots, \texttt{x}_{n-1} \rangle$$

$$\cong$$

$$(\texttt{f x}_0) \odot \cdots \odot (\texttt{f x}_{n-1}) \odot \texttt{z}$$

(here we again represent **g** by the infix operator $\odot$)

So, if **f** and **g** have $O(1)$ work and span on all arguments, then $\texttt{mapreduce f z g } \langle \texttt{x}_0, \ldots, \texttt{x}_{n-1} \rangle$ has $O(n)$ work and $O(\log(n))$ span.

# `filter`

`filter p s` $\cong$ `s'`,

with `s'` a sequence consisting of all $x_i$ in `s` such that `p(`$x_i$`)` $\cong$ `true`. The order of retained elements in `s'` is the same as in `s`.

If `p` has O(1) work and span on all arguments, then `filter p s` has O(n) work and O(log(n)) span (this is not obvious; you will learn more in 15-210).

# filter

One possible implementation that has O(log(n)) span (but O(nlog(n)) work):

```
fun filter p =
     let
        val nothing = empty()
        fun keep x =
             if p x then singleton x
                       else nothing
     in
        mapreduce keep nothing append
     end
```

These are also sequence functions.

# filter

One possible implementation that has $O(\log(n))$ span (but $O(n\log(n))$ work):

```
fun filter p =
     let
        val nothing = empty()
        fun keep x =
             if p x then singleton x
                    else nothing
     in
        mapreduce keep nothing append
     end
```

| | |
|---|---|
| `singleton x` ≅ `<x>` | `append` has $O(1)$ span |

Example (recall also Lecture 1):

```
fun sum (s : int Seq.seq) : int =
      Seq.reduce (op +) 0 s

type row = int Seq.seq
type room = row Seq.seq

fun count (class : room) : int =
      sum (Seq.map sum class)
```

(Here we are assuming a structure **`Seq`** ascribing to signature **`SEQUENCE`**.)

Example (recall also Lecture 1):

```
fun sum (s : int Seq.seq) : int =
      Seq.reduce (op +) 0 s

type row = int Seq.seq
type room = row Seq.seq

fun count (class : room) : int =
      sum (Seq.map sum class)
```

Let value `c:room` contain `m` `row`s of length `n` each.
What are the work and span of evaluating `count(c)`?

## Example (recall also Lecture 1):

```
fun sum (s : int Seq.seq) : int =
      Seq.reduce (op +) 0 s

type row = int Seq.seq
type room = row Seq.seq

fun count (class : room) : int =
      sum (Seq.map sum class)
```

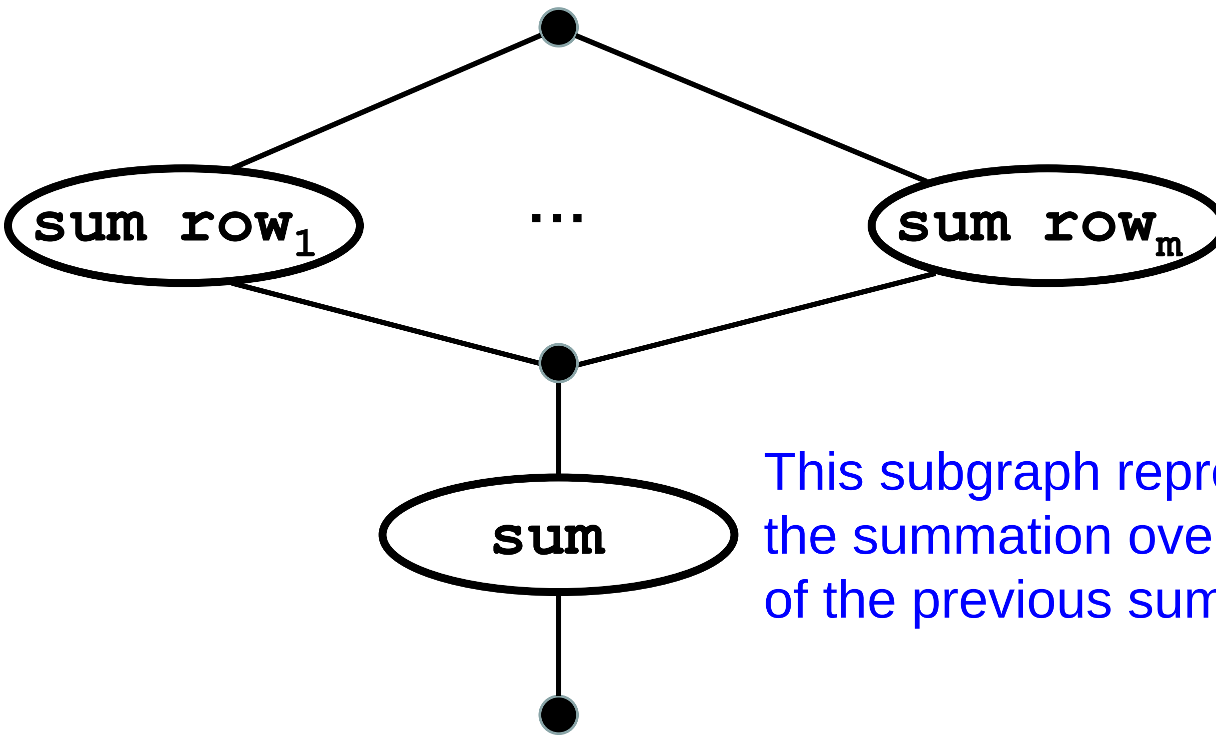
Let value `c:room` contain `m rows` of length `n` each.
What are the work and span of evaluating `count(c)`?

Answer: O(nm) work and O(log(n)+log(m)) span.

Answer: $O(nm)$ work and $O(\log(n)+\log(m))$ span.

To see that, construct a cost graph.

Suppose `c = <row`$_1$`, …, row`$_m$`>`:

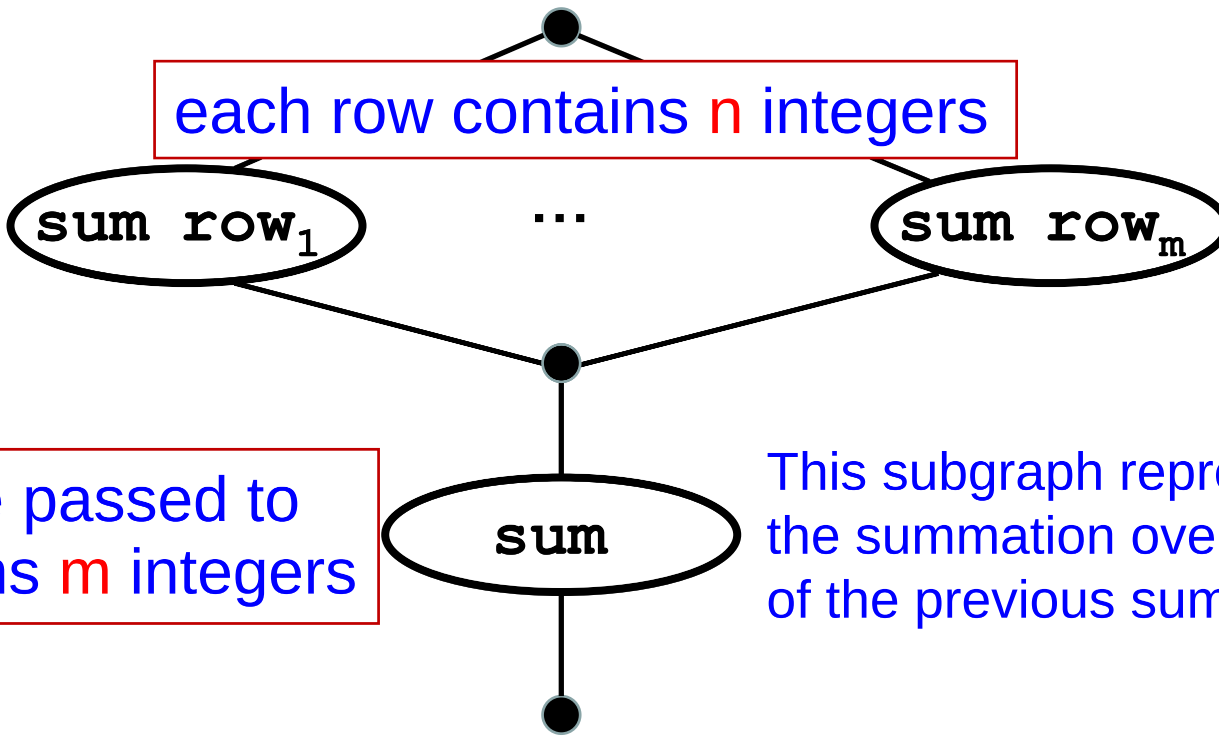



This subgraph represents the summation over the results of the previous summations.

Answer: O(nm) work and O(log(n)+log(m)) span.

To see that, construct a cost graph.

Suppose `c = <row_1, …, row_m>`:




sequence passed to **`sum`** contains m integers

This subgraph represents the summation over the results of the previous summations.

(Recall that **`sum`** has linear work and logarithmic span.)

## Example (recall also Lecture 1):

```
fun sum (s : int Seq.seq) : int =
      Seq.reduce (op +) 0 s

type row = int Seq.seq
type room = row Seq.seq

fun count (class : room) : int =
      sum (Seq.map sum class)
```

---

We could also have implemented `count` as:

```
val count : room -> int =
      Seq.mapreduce sum 0 (op +)
```

# That is all.

Please have a good Carnival!

See you next Tuesday, when we will talk about lazy evaluation.