15 - 210

Parallel and Sequential Data Structures and Algorithms

Fall 2023

At Carnegie Mellon University

Notes by Lómenvoirë Mortecc.

Lec 1

Instructors : Guy Blelloch + Charlie Garrod
Today : Motivation for course content
Platform : Diderot
Lab credit caps at 80%
3 exams

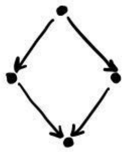
Deconstructing course title

Parallel

- Parallel
- Sequential – special case of parallel by having n = 1

Not using multiple cores → wasting your time
Many algorithms inherently parallel → use more cores

Dependency graph



← Recall work and span
work – total computation
span – longest path

Data structure & Algorithm

- Math Calculus, series, probability, linear algebra, proofs
- Abstraction Algorithm. interfaces, graphs, asymptotic analysis
- ~~Python~~ Problem solving Toolbox ↔ connections ⇒ problem, search for solution
  - connections – recognise similarity btwn problems
  - search for solution – trial and error with intuition

Example problem solving

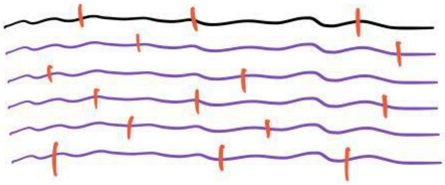
Problem : human genome sequencing (2001, 3.1 billions of nucleotide)

String of {A, C, G, T}, 3.1 billion in length

Constraints
- Can't read more than 2000 base pairs
- Sequential read takes 100s of years

→ Technique – Shotgun Method

make multiple copies
shatter into fragments } Done in lab
read each fragment — ~1000 long
reconstruct whole sequence } done in computer
↖ try find overlaps and combine



The algorithm

Get set of all sequences read
Get rid of sequences that are subset of another
Find best reconstruction
↖ Heuristic : find shortest superstring

Reduced problem : Shortest Substring (SS) Problem
└ Also good to check if sth is NP hard └ NP hard!
Informally : given set of strings, find shortest superstring that includes all

Problem solving

→ First try brute force solution, as long as correct
try all permutations, merge overlaps, pick shortest
Correct, but $O(n!)$

→ SS NP hard but has polynomial time approximation and not all possible input instances are hard
Connection : Travelling Salesman Problem (given graph and distances in edges, visit all nodes with lowest distance)
Reduction : String → vertex
$w(S_1, S_2) \rightarrow -\text{overlap}(S_1, S_2)$
add special vertex Λ, make $w(S_1, \Lambda) = w(\Lambda, S_1) = 0$ for all S_1, to fix cycles

Lec 2 Asymptotic Analysis, Recurrances

Asymptotic Analysis

- useful abstraction
    - simplifies expression
    - avoid machine detail / programming lang
    - focus on details of algorithm

Def $f(n)$ asymptotically dominates $g(n)$ if $\exists\, c, n_0$ s.t.

$$g(n) \leq c \cdot f(n) \quad \forall\; n > n_0$$

Ex	$f(n)$	$g(n)$	
	$2n$	n	Yes
	n	$2n$	Yes
	$n \lg n$	n	Yes
	2^n	$2^{1.1n}$	No

Notation
- lg means $\log_2$
- $O(f(n)) = \{ g \mid f$ asymptotically dominates $g \}$
- $\Omega(f(n)) = \{ g \mid g$ asymptotically dominates $f \}$
- $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$
- Notation abuses… Really means…

Notation abuses…	Really means…
* $n = O(n^2)$	$n \in O(n^2)$
* $f(n) = g(n) + O(n^2)$	$f(n) \in \{ g(n) + h(n) \mid h(n) \in O(n^2) \}$
* $O(n) = O(n^2)$	$O(n) \subseteq O(n^2)$

- $o(f(n)) = O(f(n)) \setminus \Theta(f(n))$
- $\omega(f(n)) = \Omega(f(n)) \setminus \Theta(f(n))$

Recurrences

- Base cases, recursive cases
- Modelling recurrent functions
- Harder to find closed form solution

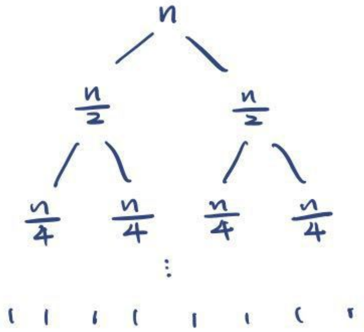
```
msort(A) = if |A| ≤ 1 then A else
           let (L,R) = msort(A[0 … $\frac{|A|}{2}$]),
                       msort(A[$\frac{|A|}{2}$ … |A|])
           in merge(L,R) end
```

$$W_{msort}(n) = \begin{cases} c_1 & \text{if } n \leq 1 \quad \leftarrow \text{Convention: drop this trivial base case} \\ 2W(\frac{n}{2}) + W(n) + c_2 & \text{if } n > 1 \end{cases} \quad \in O(n \lg n)$$

More abused notation: $W(n) = 2W(\frac{n}{2}) + O(n)$

Tree Method – Unfold level by level and sum them

msort



local costs

$O(n)$

$2O(\frac{n}{2}) = O(n)$

$O(n)$

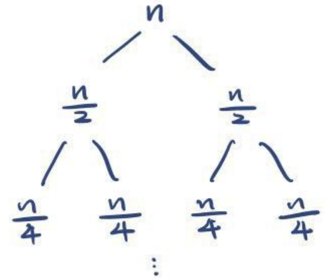
$\vdots$

$O(n)$

$\left.\right\}$ lg n levels

$$W(n) = \sum_{i=0}^{\lg n - 1} (c_1 n + 2^i c_2) \quad \ldots \in O(n \lg n)$$

Consider $W(n) = 2W(\frac{n}{2}) + n^2$



L_0 $\quad cost(L_1) = n^2$

L_1 $\quad cost(L_2) = 2(\frac{n}{2})^2$

L_2 $\quad cost(L_3) = 4(\frac{n}{4})^2$

$\vdots$

L_i $\quad cost(L_i) = 2^i(\frac{n}{2^i})^2 = \frac{n^2}{2^i}$

Root dominated recurrances

$$W(n) = n^2 + \underbrace{\frac{n^2}{2} + \frac{n^2}{4} + \frac{n^2}{8} + \cdots}_{\text{geometric decay}}$$
$\in O(n^2)$

other cases:
- leaf dominated
- balanced

Aside:

$1 + \alpha + \alpha^2 + \alpha^3 + \cdots$

$= \frac{1 - \alpha^{d+1}}{1 - \alpha}$ for $\alpha \neq 1$

Brick Method

Case 1: $cost(children(v)) \leq \alpha\, cost(v)$ $\forall$ node v and $\forall$ $0 < \alpha < 1$ (All children)
then overall cost $\in O(cost(root))$

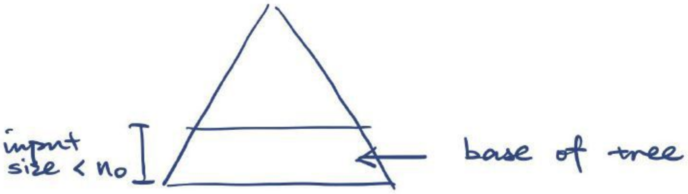
Proof: $cost(L_0) + \cdots + cost(L_d)$

$\leq cost(L_0) + \alpha\, cost(L_0) + \alpha^2 cost(L_0) + \alpha^d cost(L_0)$

$\leq \frac{1 - \alpha^{d+1}}{1 - \alpha} cost(L_0)$

$\leq \frac{1}{1 - \alpha} cost(L_0)$

Case 2: cost $(v) \leq \alpha$ cost (children(v)) for all nodes v with input size $> n_0$, $0 < \alpha < 1$.
then overall cost is $O($ cost $({}^{\text{base of}}_{\text{tree}}))$



Suppose same input size for each level, then overall cost

$= \text{cost}(L_0) + \cdots + \text{cost}(L_{d-1}) + \text{cost}(L_d) + \text{cost}\left({}^{\text{base of}}_{\text{tree}}\right)$

$\leq \alpha^d \text{cost}(L_d) + \cdots + \alpha\, \text{cost}(L_d) + \text{cost}(L_d) + \text{cost}\left({}^{\text{base of}}_{\text{tree}}\right)$

↑ Still need to compute this

Lec 3 Brick method balanced case & substitution method

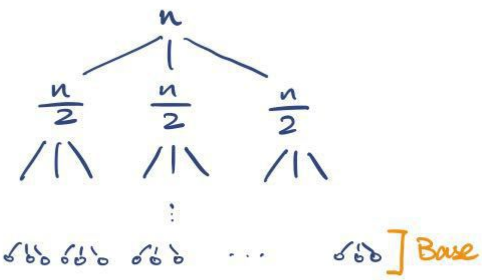
Brick method - more on leaf dominated case

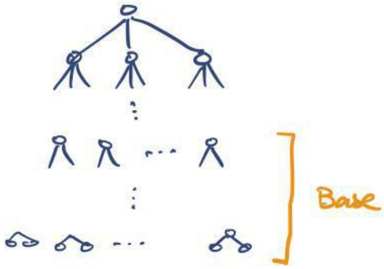
↱ Sum of all children

Thm Suppose cost(v) $\leq \alpha$ cost(children(v)) for all nodes v with input size greater than some n_0, $0 < \alpha < 1$. Then the overall cost is O(cost(base of tree))

Ex. $W(n) = 3W(\frac{n}{2}) + n$

$$W(n) = \begin{cases} 3W(\frac{n}{2}) + n & n > 42 \\ 2W(n-1) + 1 & 1 < n \leq 42 \\ 1 & n \leq 1 \end{cases}$$

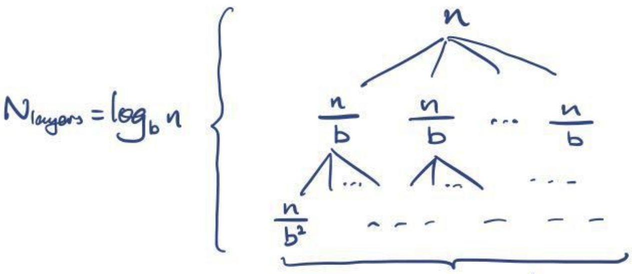




Computing cost of base

Usually ... if leaves has $O(1)$ and root is leaves,
cost(base of tree) = O(# leaves)

Ex. $W(n) = a\,W(\frac{n}{b}) + \cdots$

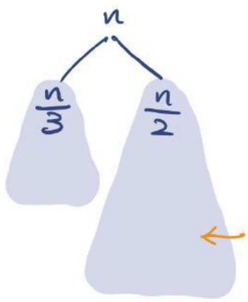


$$\begin{aligned} N_{leaves} &= a^{\log_b n} \\ &= n^{\log_b a} \\ &\in O(n^k) \end{aligned}$$

by $a^{\log_b c} = c^{\log_b a}$

But some cases are different...

Ex. $W(n) = W(\frac{n}{2}) + W(\frac{n}{3}) + \sqrt{n}$ ...



← and leaves scattered

↑ But more things here

local cost: $\text{cost}(v_n) = \sqrt{n}$

$\text{cost}(\text{children}(v_n)) = \sqrt{\frac{n}{2}} + \sqrt{\frac{n}{3}} = \sqrt{n}\left(\frac{1}{\sqrt{2}} + \frac{1}{\sqrt{3}}\right)$

$= 1.284\sqrt{n}$

↑ So increasing — leave dominated.

sometimes not the case

#leaves $L(n) = \begin{cases} \overset{\downarrow}{1} & \text{if } n \le 1 \\ L(\frac{n}{2}) + L(\frac{n}{3}) & \text{else} \end{cases}$

Substitution Method

— aka guess and check

guess: come up with some function

check: by induction

Ex. (cont) guess: $L(n) = n^b$ for some constant b

(BC) $L(n) = n^b = 1^b = 1$ for all b.

(IH) Assume for $0 \le k < n$, $L(k) = k^b$.

(IS) $L(n) = L(\frac{n}{2}) + L(\frac{n}{3})$

$= (\frac{n}{2})^b + (\frac{n}{3})^b$ by IH

$= n^b \cdot (\frac{1}{2^b} + \frac{1}{3^b})$

But if we want $n^b \cdot (\frac{1}{2^b} + \frac{1}{3^b}) = n^b$, we need:

$\frac{1}{2^b} + \frac{1}{3^b} = 1$

⇝ Wolfram alpha

$b \approx 0.788\ldots$

$\Rightarrow L(n) = n^{0.788\ldots}$

$W(n) \in O(n^{0.788\ldots})$ ← since leaves dominated

(if... say base case costs $O(m)$, $W(n,m) \in O(mn^{0.788\ldots})$)

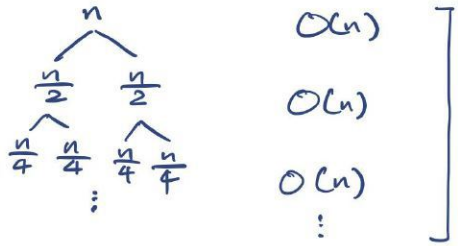
Brick method – balanced tree

If work balanced, across levels:
└ asymptotically the same as imprecise definition

overall cost $\le \underbrace{\max_{L_i} (\text{cost}(L_i))}_{\text{highest cost level}} \cdot$ # levels

Ex. Merge sort

$W(n) = 2W(\frac{n}{2}) + O(n)$



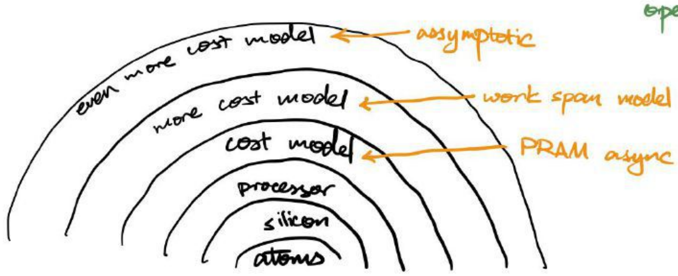
level ∈ O(log n)

overall ∈ O(n log n)

Note: not all recurrences fall in one of brick cases

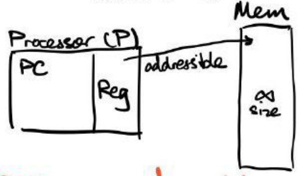
Cost models

— another layer of abstraction,
the question of asymptotic what?
time?
operations?
⋮



Some types of models...

- Random access machine (RAM) model ← Good enough for writing better algorithm



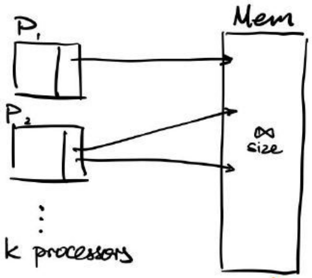
O(1) instructions: read, write, add, multiply,
jumps, conditionals ...

Sequential complexity in 122: #instructions on RAM model

Imperfection: read write may not be O(1) ... (think cache)

- IO model: non-constant read/write cost

- RAM model but multiple processors



- P-RAM model : that but all processors run synchronously
  - P-RAM (W) : variant to allow write at same time
  - P-RAM (exclusive W) : -- disallow ---

Problems: how do we model and partition?
maybe possible, but messy to work with
also synchronization is costly to implement
↳ but asynchronous makes it even harder to program

- On top of async PRAM — Nested Parallel Work-Span Model

More like a language cost model than machine model

expressions		Work	Span
$e ::= x$	(var)	1	1
$\mid c$	(constant)	1	1
$\mid e_1 + e_2$		$W(e_1) + W(e_2) + 1$	$S(e_1) + S(e_2) + 1$
$\mid e_1 \parallel e_2$	(parallel)	$W(e_1) + W(e_2) + 1$	$\max(S(e_1), S(e_2)) + 1$
$\mid e_1 , e_2$	(sequential)	$W(e_1) + W(e_2) + 1$	$S(e_1) + S(e_2) + 1$

Lec 4 More cost model & Array Sequence

Nested parallel model ← recursively define

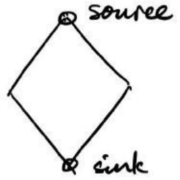
note jumps & exceptions break the model

e	W	S
x	1	1
c	1	1
$e_1 + e_2$	$1 + W(e_1) + W(e_2)$	$1 + S(e_1) + S(e_2)$
$e_1 ; e_2$	$c + W(e_1) + W(e_2)$	$c + S(e_1) + S(e_2)$
$e_1 \parallel e_2$	$c' + W(e_1) + W(e_2)$	$c' + \max(S(e_1), S(e_2))$
if e_1 then e_2 else e_3	$W(e_1) + \begin{cases} W(e_2) \\ W(e_3) \end{cases}$	$S(e_1) + \begin{cases} S(e_2) \\ S(e_3) \end{cases}$

This models Sync P-RAM pretty well

Dependence graph representation

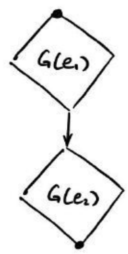
└ isomorphic to above representation



$G(x)$, $G(1)$

• source = sink

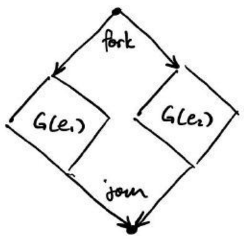
$G(e_1 ; e_2)$



W = # nodes

S = longest path

$G(e_1 \parallel e_2)$



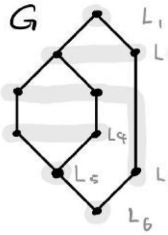
Mapping nested par model to hardware

pebble game – given depence graph G, put up to $\overset{\#\text{processors}}{p}$ pebble on nodes at each step s.t. all prerequisites have a pebble, with the goal of minimising # steps

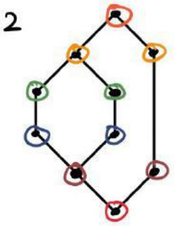
Greedy strategy – always put down min(r, p) pebbles, where r = number of ready pebbles

there's always an optimal sol that's greedy (but greedy not always optimal)

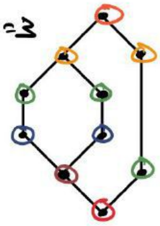
G



$p=2$



$p=3$



Let n = # nodes
d = len of path

<u>Claims</u>

1. It requires at least $\max(\lceil \frac{n}{p} \rceil, d)$ steps
2. Finding optimal is NP-hard ← But we can approximate quickly
3. Any greedy strategy will take at most $\frac{n}{p} + d$ steps, so always within factor of 2 to optimal, usually better

aka greedy scheduling theorem

<u>Mapping</u>

Nested model
W, S

PRAM (T = # of steps, p = # of processors)

$$\max(\frac{W}{P}, S) \leq T \leq \frac{W}{P} + S$$

We want this to dominate... this happens if:

$$p < \frac{W}{S}$$

$$\text{parallelism} = \frac{W}{S}$$

<u>Proof</u> for greedy scheduling theorem

Def: node is at level l if its longest path to root is l.

Lemma: on every step, either:
1. put down p pebbels
2. finish a level

<u>Proof</u> AFSOC let L_j be longest level that all nodes are covered
Then at L_{j+1} all nodes are either done or ready.
Then if we put less than p pebbels and not finish the level, we're not greedy

Array sequences , bottom up

Data structure : array

(other impl could use list , function , trees , …)

Primitives for array

		W	S
a[i]	get i-th elem	O(1)	O(1)
|a|	get length	O(1)	O(1)
alloc (n)	allocate array of length n	O(1)	O(1)
parallelFor (pFor)	i = x to y , evaluate e(i) in parallel	$\sum_{i=x}^{y} W(e(i)) + 1$	$\max_{i=x}^{y} S(e(i))$

↳ Has unavoidable side effect

Race condition : both write or one read one write

↳ Avoid this

Implementations

```
map f A =
  R = alloc |A|
  pFor i = 0..(|A|-1)
    R[i] = f A[i]
  ret R
```

```
tabulate f n =
  R = alloc n
  pFor i = 0..(n-1)
    R[i] = f i
  ret R
```

Lec 5 Sequences

\# Recall dependence graph & pebbel game

Greedy strat take at most $\frac{W}{P} + S$

Well then at each step we either :
- contribute to $\frac{W}{P}$ term
- contribute to S term
- contribute to both

So we fill $\frac{W}{P} + S$ by greedy scheduling

\# Work span trade off

→ Which to optimise?

$\frac{W}{P} + S$ ... usually W first. Usually give up no more than $O(\log n)$ work for better span

\# Array seqs

- Primitives A[i], alloc, |A|, parFor
- Assume parFor forks as many as it wants

```
append A B = tab (fn i ⇒ if  i < |A| then A[i]
                          else B[i - |A|] )   (|A| + |B|)
```

$W = O(|A| + |B|)$ $S = O(1)$

subseq -- tabulate and grab indices ?

$W = \cancel{O(\ell)}$ $S = O(1)$

Nope spec says $O(1)$.
Because values not mutable we can reference subseq

Efficient subseq & split mid

type α seq = (α array * start * end)

→ Then operation does index manipulation without necessarily copying part of the α array.

iterate , iteratePrefixes , reduce , scan

↙ It's just foldl

iterate : $\underset{F}{(\beta \times \alpha \to \beta)} \to \underset{init}{\beta} \to \underset{A}{\alpha\ seq} \to \beta$

$W = O(\cancel{\Sigma} \sum_{i=0}^{n-1} W(f(x_i, A[i]))$ $\quad S = W$

↑ Prof's new symbol , whoops

Consider :

```
x = <init>
B = alloc |A|
for i in 0..(n-1)
    B[i] = x
    x = f(x, A[i])
ret (B, x)
```

iteratePrefixes : $(\beta \times \alpha \to \beta) \to \beta \to \alpha\ seq \to (\beta\ seq , \beta)$

But if f associative and <init> is left identity of F, we can do things in parallel

⇒ iterate f I A ≡ reduce f I A

Associative funcs

+, *, ^, ...

$f((l_1, r_1), (l_2, r_2)) =$ if $(r_2 > l_2)$ then $(l_1 , r_1 - l_2 + r_2)$
else $(l_1 - r_1 + l_2 , r_2)$

copy(x, y) = case y of NONE ⇒ x
| _ ⇒ y

Examples

Assuming $W_{merge} = O(n)$ $S_{merge} = O(\log n)$

iterate (merge <) ⟨⟩ ⟨⟨x⟩ : x ∈ A⟩ ← insertion sort $W = O(n^2)$ $S = O(n \log n)$

reduce (merge <) ⟨⟩ ⟨⟨x⟩ : x ∈ A⟩ ← merge sort $W = O(n \log n)$ $S = O(\log^2 n)$

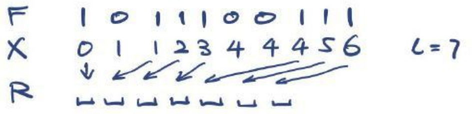
Lec 6 More on sequences + techniques

Filter

$W = O(n)$ $S = O(\log n)$

```
filter f A =                → notation ⟨x∈A | F(x)⟩
  F = map (fn x ⇒ 1 if f(x) else 0) A
  (X, l) = scan op+ 0 F
  R = alloc(l)
  pFor i = 0..(n-1) :
    if (F[i] = 1) then R[X[i]] = A[i]
  ret R
```

scan to get which index the element go



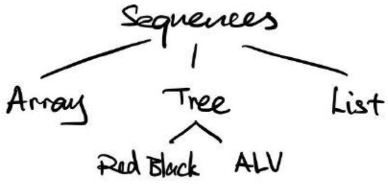
Flatten

```
flatten A =
  L = ⟨|s| | s ∈ A⟩
  (X, l) = scan op+ 0 L
  R = alloc l
  pFor i = 0..(|A|-1)
    pFor j = 0..( L[i] -1
      R[X[i] + 1] = (A[i])[j]
  ret R
```

	⟨⟨2,3⟩, ⟨7,8,1⟩, ⟨4⟩⟩
map length	⟨2, 3, 1 ⟩
scan (op+)	⟨0 2, 5 ⟩, 6
R indexes	⟨0, 1, 2, 3, 4, 5 ⟩

Sequences Abstractions

Definition



Cost model		
└ A[i] work	$O(1)$	$O(\log n)$
└ append A B	$O(\lvert A\rvert + \lvert B\rvert)$	$O(\log(\lvert A\rvert + \lvert B\rvert))$

Impl detail : ahh whatever

Math abstraction

Problem solving <
- toolbox
- connections
- search

Alg design techniques

- Reduction
- Brute force
- Divide and conquer
- Contraction
- Greedy alg
- Dynamic programming

Reduction

Def Problem A is reducible to problem B if ∀ instance of A we can :
- transform the instance to some instance of B
- solve it
- transform the solution back to a solution for A

Ex. Find max in input sequence

Bad reduction : sort it , grab last elem

$$\xrightarrow{\text{convert}} \text{sort} \xrightarrow{\text{convert}}$$

$O(1)$ $O(\ddot{\frown})$ $O(1)$

Brute - force

more confident , good parallelisation , use as baseline

— "Consider and check all possible solutions"

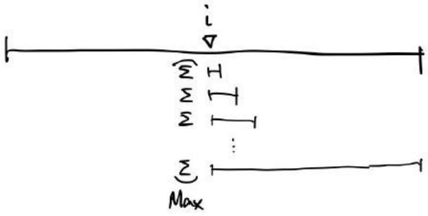
Ex. MCSS

→ Check all possible subseqs $O(n^3)$ work

then check redundant work

But wasted lots of work summing independently

Reduction : MCSS with start position i problem (MCSSwSP)



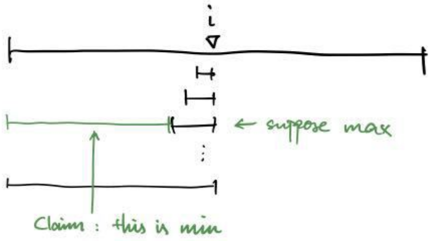
Scan then reduce — $O(n)$ work
$O(\lg n)$ span

Then solve MCSSwSP for all i , then reduce .
$O(n^2)$ work , $O(\lg n)$ span

Better ! ... but still some wasted work across different start positions

(MCSSwEP)

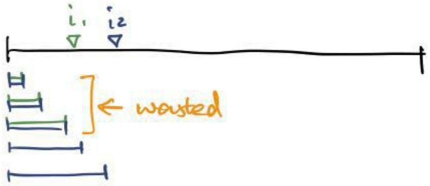
Further reduction: MCSS with end position i problem



```
MCSSwEP A i = let
   (b, v) = scan op+ A[0, i]
   prefixSum = append b <v>
   minPrefix = reduce min ∞ prefixSum
in
   v - minPrefix
end
```

O(n) work   O(log n) span

↑ Now redundant work have overlapping scans.



New alg

```
MCSS A = let
   (b, v) = scan op+ 0 A
   prefixSums = append b <v>
   (minPrefixes, _) = scan min ∞ prefixSum
   maxForEnds =
        < prefixSums [i] - minPrefix [i] | 0 ≤ i ≤ |A| >
in
   reduce max -∞ maxForEnds
end
```

Yay O(n) work   O(log n) span

↑

This sol took 9 years to find

Lec 7 Alg design techniques continued

More divide and conquer

Generally...
- Base case ...
- Inductive case
  1. Divide into $f(n)$ parts of $g(n)$ size
  2. Recurse
  3. Combine results

Skeleton

```
DC A =
  if |A| = 0   ⇒   ----
  if |A| = 1   ⇒   ----
  else let
    (L,R) = DC ( A[0, $\frac{|A|}{2}$] || B[$\frac{|A|}{2}$, |A|] )
  in
    combine (L , R)
  end
```

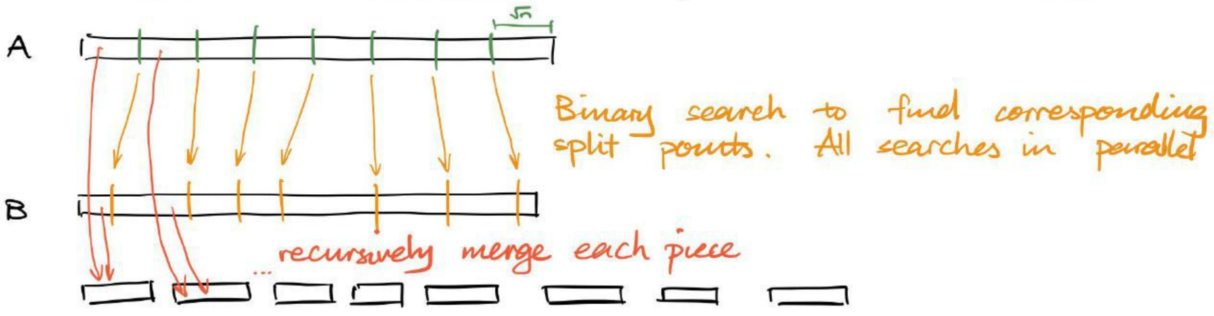
... but that's long we can actually do

```
reduce combine empty ⟨ base(x) | x ∈ A ⟩
```

Merge with $O(n)$ work $O(\log n)$ span

Let $n = |A| + |B|$, WLOG $|A| > |B|$

Break into $\sqrt{n}$ subinstances. Each piece also $O(\sqrt{n})$ in size



Binary search to find corresponding split points. All searches in parallel

... recursively merge each piece

Asuming...

$W_{split} = O(\sqrt{n}\ \lg n)$ $S_{split} = O(\lg n)$

$W_{combine} = O(\sqrt{n})$ $S_{combine} = O(\lg n)$

(for some tree sequence impl)

Then overall...

this is assuming B corresponds to A's split well

$$W(n) = \sqrt{n}\,W(\sqrt{n}) + W_{split}(n) + W_{combine}(n)$$
$$= \sqrt{n}\,W(\sqrt{n}) + O(\sqrt{n}\lg n)$$

parent $\sqrt{n}\log n$

children $\sqrt{n}\cdot(\sqrt{\sqrt{n}}\log\sqrt{n}) = n^{\frac{3}{4}}\cdot\frac{1}{2}\cdot\lg n$ Leave dominated

... $W(n) \in O(n)$

$$S(n) = S(\sqrt{n}) + S_{split}(n) + S_{combine}(n)$$
$$= S(\sqrt{n}) + O(\lg n)$$

parent $\lg n$

child $\lg\sqrt{n} = \frac{1}{2}\lg n$ Root dominated

So $S(n) \in O(\lg n)$

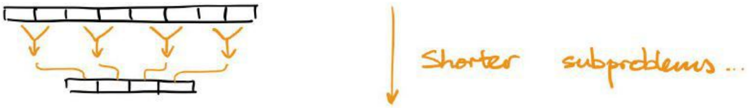
Contraction

Break into one piece ... but recursively solve the one piece

- Base case ...
- Inductive case
  1. Contract into one piece of size $g(n)$
  2. Recurse on subinstance
  3. Expand result to solve original problem

Ex.

```
reduce f I S = cas |S| of
  0 ⇒ I
  1 ⇒ f(I, S[0])
  _ ⇒ let
    B = ⟨ f(S[2i], s[2i+1]) | 0 ≤ i < |S|/2 ⟩
  in
    reduce f I B
  end
```



$$W(n) = W(\tfrac{n}{2}) + O(n) \in O(n)$$

$$S(n) = s(\tfrac{n}{2}) + O(1) \in O(\lg n)$$

Ex. scan [omitted]

scan op+ 0 ⟨2, 1, 1, 4, 3, 2, 1, 3⟩

contract 3 5 5 4

recursive scan (⟨0, 3, 8, 13⟩, 17)

expand (⟨0, 2 3, 4 8, 11 13 14⟩, 17)

$$W(n) = W(\tfrac{n}{2}) + O(n) \in O(n)$$

$$S(n) = S(\tfrac{n}{2}) + O(1) \in O(\lg n)$$

Lec 8 Probability for randomised algorithms

Exam: bring yourself, 1 handwritten sheet, 4 function calculator

Motivation for randomised algorithms

- Can be faster
  - sometimes faster by constant factor
  - sometimes faster asymptotically
- Can be simpler
- Break symmetry — hopefully low probalibity to choose badly

- Unpredictable
  - Running time
  - Inconsistent
  - Don't know how long each fork takes when parallelised
- Need source of randomness
- Hard to analyse

Ex. Prime test randomised algorithm
Polynomial time, simple implementation

Las Vegas algorithm random → always right answer

Monte Carlo alg simulate → generate something close

Ex. Random Distance Run

2 giant dice
1st roll : how many laps for one
2nd roll : how many more to run

Define rand vars : D_1 = value of 1st die
D_2 = value of 2nd die

Expected values... $E[D_1] = 3.5$
$E[D_2] = 3.5$

Multiplication works if independent ↓

What about expected sum of 2 dice $E[D_1 + D_2] = 7$
... expected product of 2 dice ... $E[D_1 D_2] = \cdots 12.25$
... expected max $E(\max(D_1, D_2)) = 4\,{}^{17}\!/_{36}$ ← Not clearly related to max of expectation.

Probability

Sample space Ω

Prob measure $P : \mathcal{P}(\Omega) \to \mathbb{R}$ with:

1. $\forall A,\quad 0 \leq P(A) \leq 1$
2. $\forall A, B,\quad A \cap B = \emptyset \Rightarrow P(A) + P(B) = P(A \cup B)$
3. $P(\Omega) = 1$

Random variable ... neither random nor variable

Deterministic function $X : \Omega \to \mathbb{R}$

Expected value of X $$E[X] = \sum_{\omega \in \Omega} \underbrace{P(\omega)}_{\text{prob of that outcome}} \cdot \overbrace{X(\omega)}^{\text{value of outcome}}$$

Independent X, Y indep if

$$P[X = a, Y = b] = P[X = a]\,P[Y = b] \quad \forall a, b$$

Linearity of expectation $E[X + Y] = E[X] + E[Y]$ ← (always true)

Expectation of product $E[X \cdot Y] = E[X] \cdot E[Y]$ ← (assuming independent)

Union bound $P(A) + P(B) \geq P(A \cup B)$

Conditional prob $P(A \mid B) = \dfrac{P(A \cap B)}{P(B)}$

Entangled dice

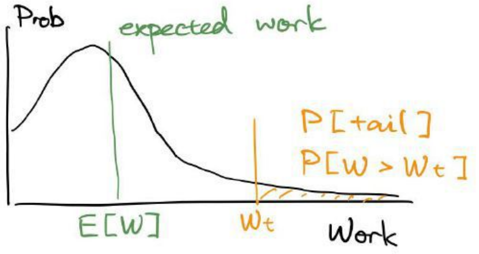
Suppose 2^{nd} die must be same as first die

Expected sum of dice → 7

Expected product → $15\,\frac{1}{6}$

Alg analysis with prob

Tail bound



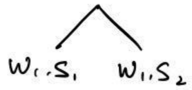
Markov's inequality tool for bounding tail

If $X \geq 0$ then $P[X > a] \leq \frac{E[X]}{a}$ $\forall a$ ↙ threshold

Quicksort

pick random pivot → partition → recur → append
Unlucky case : picking bad pivot.

Goal : analyse work & span of rand. alg.



$W = W_1 + W_2$ ← okay to bound
$S = \max(S_1, S_2)$ ← hard to bound

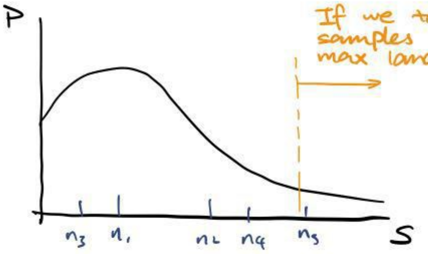
High probability bound

Say $W(n) \in O(f(n))$ with high probability (w.h.p) if
$W(n) \in O(\underset{\substack{\uparrow \\ \text{how much worse}}}{k} \cdot f(n))$ with probability $> 1 - \underset{\substack{\uparrow \\ \text{how often does it violate bound}}}{\left(\frac{1}{n}\right)^k}$

how much worse ⟵ We can define these differently ⟶ how often does it violate bound

Intuitively, $k \uparrow$ $1 - \left(\frac{1}{n}\right)^k \uparrow$
so the higher the violation the less often we are allowed to violate the bound

Consider max of n spans



If we take max of n samples how often the max land here

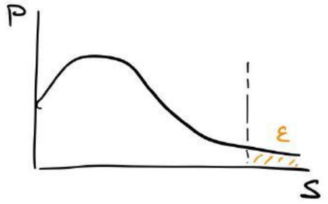
Lec 9 Probability Bound Analysis

High prob bound

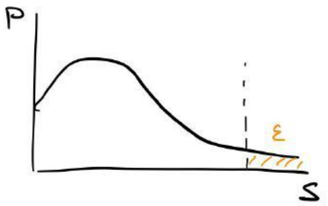
Redef Say $W(n) \in O(f(n))$ w.h.p. if $\exists$ constants c, n_0 s.t.
$\forall n > n_0$, $\forall k$ $W(n) \leq ckf(n)$ with probability $\geq 1-(\frac{1}{n})^k$

Max of spans

Consider n spans



……



Suppose there's ε prob. that a single span is bad

Prob that some of them bad is by union bound $\leq n\varepsilon$
$P[\text{Some bad}] = P[1^{st} \text{ being bad} \cup \cdots \cup n^{th} \text{ being bad}]$

Ex. Suppose each piece has $O(\lg n)$ w.h.p.

$P[\text{indiv good}] = 1-(\frac{1}{n})^k$

$P[\text{indiv bad}] = (\frac{1}{n})^k$

$P[\text{some bad}] \leq n(\frac{1}{n})^k = (\frac{1}{n})^{k-1}$
$= (\frac{1}{n})^{k'}$ $\forall k'$

$\Rightarrow$ Overall span is $O(\lg n)$ with prob $\geq 1-(\frac{1}{n})^{k'}$
so w.h.p. composed to $O(\lg n)$

Ex. toy alg. for skittles game

Game: jar start with n skittles
flip coin, if head eat [half of remaining]
else noop

Question: how many rounds before run out of skittles

... worse case ∞ ?

Define random var $X_d :=$ number of skittles at start of round d.

$$X_0 = n$$

$$\mathbb{E}[X_{d+1}] = \frac{1}{2}\mathbb{E}[X_d] + \frac{1}{2}\left\lfloor \frac{\mathbb{E}[X_d]}{2} \right\rfloor$$

$$\le \frac{3}{4}\mathbb{E}[X_d]$$

$$\Rightarrow \mathbb{E}[X_d] \le n\left(\frac{3}{4}\right)^d \quad \leftsquigarrow \text{ by induction}$$

Claim: num rounds $\le 10 \lg n$ with prob $1 - \left(\frac{1}{n}\right)^{3.15}$

Proof.

$$\mathbb{E}[X_{10\lg n}] \le n\left(\frac{3}{4}\right)^{10\lg n}$$

$$= n \cdot n^{10\lg\frac{3}{4}}$$

$$\approx n \cdot n^{-4.15}$$

$$= \frac{1}{n^{3.15}}$$

By Markov's inequality $P[X_{10\lg n} \ge 1] \le \dfrac{\mathbb{E}[X_{10\lg n}]}{1} = \dfrac{1}{n^{3.15}}$

$$\Rightarrow P[X_{10\lg n} < 1] = P[X_{10\lg n} = 0]$$

$$> 1 - P[X_{10\lg n} \ge 1]$$

$$= 1 - \frac{1}{n^{3.15}}$$

Lemma: num of rounds $\le \dfrac{-(k+1)}{\lg(\frac{3}{4})}\lg n$ with prob $\ge 1 - \left(\frac{1}{n}\right)^k$

let $c = \dfrac{-(k+1)}{\lg(\frac{3}{4})}$. $\mathbb{E}[X_{c\lg n}] \le n\left(\frac{3}{4}\right)^{c\lg n}$

$$= n \cdot n^{c\lg\frac{3}{4}}$$

$$= n \cdot n^{\left(\frac{-(k+1)}{\lg(\frac{3}{4})}\right)\lg\frac{3}{4}}$$

$$= n \cdot n^{-(k+1)}$$

$$= \frac{n}{n^{k+1}}$$

$$= \left(\frac{1}{n}\right)^k$$

markov stuff ... $\in O(\lg n)$

Analysing random select

An order statistics problem

Given seq A and rank k, return k^{th} smallest elem of A

→ One can simply sort, but not efficient enough

└ $W = O(n \lg n)$, $S = O(\lg^2 n)$

→ Goal: $W = O(n)$, $S = C(\lg^2 n)$ w.h.p.

```
rselect A k = let
    p = uniformly randomly selected elem
    (L,R) = ⟨x∈A : x<p⟩ || ⟨x∈A : x>p⟩
in
    if   k < |L| then select L k
    elif k = |L| then p
    else rselect R (k - |L| - 1)
```

Randomised select by contraction

Partition by pivot, then the cases...

① p is the k^{th}

② L longer than k ⇒ recurse on L

③ L shorter than k ⇒ recurse on R

Intuition for analysis

→ 50% of time picking p between Q1 and Q3
in that case we eliminate 25% of elems

Lec 10 Random Algorithm II – Order Stats Problem Analysis

Recall: skittle game, search for k-th rank in list

Randomised Select Analysis

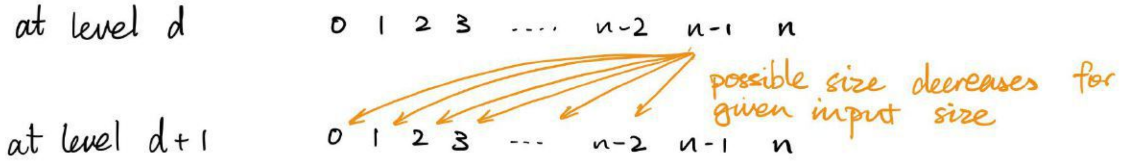
```
rselect A k = let
    p = uniformly randomly selected elem
    (L,R) = ⟨x∈A : x<p⟩ || ⟨x∈A : x>p⟩
in
    if k < |L| then select L k
    elif k = |L| then p
    else rselect R (k - |L| - 1)
```

Lucky: pick pivot close to median and eliminate $\frac{1}{2}$

Unlucky: pick close to min / max and eliminate 1

Midluck: pick sth between and eliminate $\frac{1}{4}$

Input size unknown...



Let Y_d be RV for input len at level d $(Y_0 = n)$

Z_d be RV for rank of pivot chosen at level d.

$$E[Y_{d+1}] = \sum_y \sum_z \underbrace{P[Y_d = y,\ Z_d = z]}_{\text{prob of having input size } y \text{ and picking rank } z \text{ at previous level. Corresponds to each edge!}} \underbrace{f(y,z)}_{\text{len of remaining input?}}$$

$$= \sum_y \sum_z P[Y_d = y] \underbrace{P[Z_d = z \mid Y_d = y]} f(y,z)$$

$$= \sum_y \sum_z P[Y_d = y] \frac{1}{y} f(y,z)$$

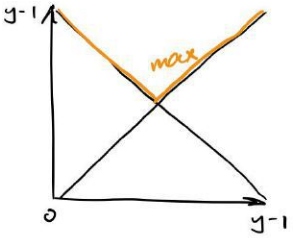
$$= \sum_y \left[P[Y_d = y] \sum_z \frac{1}{y} \underbrace{f(y,z)} \right]$$

$f(y,z)$ needs to return remaining input size

z	possible $f(y,z)$
0	$0, y-1$
1	$0, 1, y-2$
2	$0, 2, y-3$
$\vdots$	
z	$0, z, y-z-1$
$\vdots$	
$y-2$	$0, 1, y-2$
$y-1$	$0, y-1$

Worse case...

$$\sum_z f(y,z)$$
$$= \sum_{z=0}^{y-1} \max(0, z, y-z-1)$$
$$= 2\sum_{z=y/2}^{y-1} z$$
$$\vdots$$
$$\le \frac{3}{4}y^2$$



$$\le \sum_y \left[P[Y_d = y] \, \frac{1}{y} \, \frac{3}{4} y^2 \right]$$

$$= \frac{3}{4} \sum_y P[Y_d = y] \, y$$

$$= \frac{3}{4} \mathbb{E}[Y_d]$$

So $\mathbb{E}[Y_d] \le n \left(\frac{3}{4}\right)^d$

Expected work $\mathbb{E}[W] = \mathbb{E}[W_0 + \cdots + W_n]$

$$= \sum_{d=0}^{n} \mathbb{E}[W_d]$$

$$= \sum_{d=0}^{n} O\left(n \left(\frac{3}{4}\right)^d\right)$$

$$\in O(n)$$

Expected span $\mathbb{E}[S]$

$\mathbb{E}[\#\text{ of levels}] \in O(\lg n)$ w.h.p. (same as skittles game)

$\Rightarrow \mathbb{E}[S] \in O(\lg^2 n)$

Quicksort

```
qsort A =  if |A| ≤ 1 then A else  let
   p = uniformly selected pivot
   L,R =  partition in parallel
   L',R' =  qsort L || qsort R
in  L' ++ ⟨p⟩ ++ R'  end
```

Analysis by counting the number of comparisons

Define RVs $X_{i,j} = \begin{cases} 0 & \text{if keys ranked } i,j \text{ never compared} \\ 1 & \text{if } \cdots \text{ are compared} \end{cases}$

↑ Indicator RV

Observe : the pivot gets compared to everything
things only get compared if they get picked as pivot
and they they don't get compared in recursive calls

if $x<y<z$ and y is pivot, x and z never get compared

WLOG $i<j$

$$\mathbb{E}[X_{i,j}] = P[X_{i,j} = 1] = \frac{1}{j-i+1}(2!)$$

└ order for choosing i, j

└ chance for picking i, j in

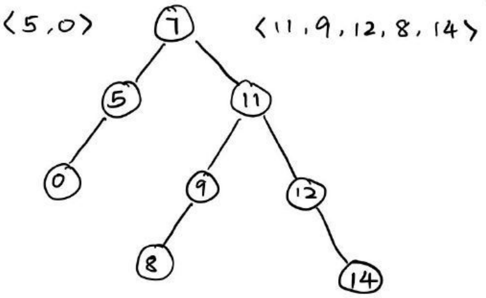
$$\begin{aligned} \mathbb{E}[W] &= O(\mathbb{E}[\#\text{ of comparisons}]) \\ &= O\Big(\sum_{i<j} \mathbb{E}[X_{i,j}]\Big) \\ &\leq 2\sum_{i=0}^{n} H_i \\ &\in O(n \log n) \end{aligned}$$

← harmonic number

$\mathbb{E}(S)$ analysis by <u>pivot tree</u>

└ recursion tree showing the pivot chosen at each node

$\langle 7, 5, 11, 0, 9, 12, 8, 14 \rangle$, always picking first



from randomised select, we found the length of one path is $O(\lg n)$ w.h.p.

$P[\text{ one path} > k_1 \lg n\,] \leq \frac{1}{n^{k_1}}$ for all constant k_1

WTS $P[\exists \text{ path} > k_2 \lg n] < \frac{1}{n^{k_2}}$ for all constants k_2

But there are $< n$ paths. By union bound:

$P[\exists \text{ path} \geq k_2 \lg n] \leq n \cdot \frac{1}{n^{k_1}}$ for all k_1.

$\leq \frac{1}{n^{k_2}}$ as long as we choose $k_1 = k_2 + 1$

$\boxed{\text{Lec 11}}$ Balanced Binary Tree I

Useful for
- ordered sets
- ordered tables
- sequences

  ⋮

Seen
- Remove, insert, find,
- AVL
- Redblack
- BSTs (binary search tree)

More bin tree ops

- insertAt *
- deleteAt *
- nth
- intersect
- union
- difference
- append *
- ranges
- split
- map
- reduce
- filter

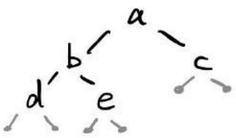
* will be better than ArraySeq

Tree options

- AVL
- Red Black
- Treaps
- Splay
- BTree
- Skapegoat
- Weight balanced
- 2-3 tree
- Skip-list

So many of them. Want to abstract all the options.
Assume there's joinMid for each tree type, implement general operations.

Binary tree



← "internal binary tree" as we don't store data on the leaves

Def Balanced := height $\in O(\log n)$
usually height $\leq 2\lg n$

Note this is always true:
height $\geq \lceil \lg(n+1) \rceil$
height $= \lceil \lg(n+1) \rceil$ when perfectly balanced

Store at nodes

- Value
- Key
- Balancing info
- Size of subtree
- Associative info (augmentation)

Binary Search Trees

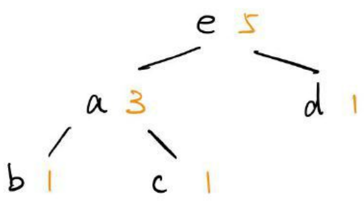
Def $$\forall_{\text{node}} , \begin{cases} \forall k \in \text{Left} , & k < \text{root} \\ \forall k \in \text{Right} , & \text{root} < k \end{cases}$$

Sequence Tree

Binary tree + size of subtree

Inorder traversal of tree is the sequence

$\langle b , a , c , e , d \rangle$



Exposing : get rid of extra info and return barebone tree

Lec 12 Balanced Binary Tree II (BT ADT, Treaps)

Today: split, union, filter, splitAt

Generic interface

```
struct
  type T                                      — tree with augmentation, etc.
  type E                                      — elem
  type N = Leaf | Node of T×E×T               — exposed form
  size : T → Z                  O(1)
  expose : T → N                O(1)
  empty : T
  joinM : T×E×T → T             usually O(|height L - height R|)
end
```

helpers:

```
singleton = λ x ⇒ joinM (empty, x, empty)
append    = λ A B ⇒ case expose A of          ← only preserves BST if L < R
                Leaf ⇒ B
              | Node (L, x, R) ⇒ joinM (L, x, append R B)
```

Impls filter ← works on both BST and tree seq

```
filter p A = case expose A of
  Leaf ⇒ empty
 | Node (L, x, R) ⇒
    let (L', R') = (filter L || filter R) in
    if p x then joinM (L', x, R')
           else append (L', R')
```

Assume for now:

joinM, append ← both sequential   W, S $O(\lg n)$ ($n = |L'| + |R'|$, assume $|L| = |R|$)

$W_{filter}(n = |L|+|R|) = 2W(\frac{n}{2}) + O(\lg n)$
$\in O(n)$

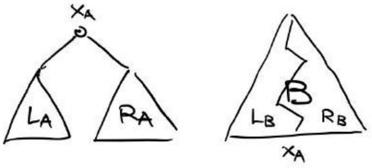
$S_{filter}(n) = S(\frac{n}{2}) + O(\lg n)$
$\in O(\lg^2 n)$

Impl split (BST only)

```
| REQ BST A                                                  : T × E
| ENS return ( {x∈A | x<k}, k∈A, {x∈A | x>k} )               : T×B×T
split A k = case expose A of                      W = O(lg n)  w.h.p.
  Leaf ⇒ (empty, false, empty)
 | Node (L, x, R) ⇒ case cmp x k of
     EQUAL ⇒ (L, true, R)
    | LESS ⇒ let
          (L_1, b, R_1) = split L k
       in
          (L_2, b, joinM (R_1, x, R))
       end
    | GREATER ⇒ [symmetry]
```

Impl union

```
| REQ A, B BSTs
| ENS BST with set of all keys in A, B
union A B = case (expose A, expose B) of
   (Leaf, _) ⇒ B
 | (_, Leaf) ⇒ A
 | (Node (L_A, x_A, R_A), _ ) ⇒ let
       (L_B, _, R_B) = split B x_A
       (L', R') = (union L_A L_B || union R_A R_B)
    in
       joinM( L', x_A, R')
    end
```



cost analysis (assume $|L'| = |R'|$, $|L_A| = |L_A|$, $|L_B| = |R_B|$, WLOG $\underbrace{|A|}_{m} \le \underbrace{|B|}_{n}$)

$W(n,m) = 2W(\frac{m}{2}, \frac{n}{2}) + O(\lg n)$ m-n ratio is same

$W(1, n_1) = \lg n_1$ ← Leaf-dominated but $n_1 = \frac{n}{m}$

$= \lg \frac{n}{m} \in O(\lg(\frac{n}{m}+1))$ ← in case $\frac{n}{m} = 1$

$\# \text{leafs} = 2^{\lg m} = m$

$\text{cost (base)} = m \lg(\frac{n}{m}+1)$

So $O(m \lg(\frac{n}{m}+1))$ ← in fact this is also lower bound.

(Span ... $\in O(\lg n \lg m)$)

Treaps aka Tree-Heap (with randomisation !)

Basically bin tree + heap ordering on priority

Priority $p : \mathbb{E} \to \mathbb{Z}$

elem $\mapsto$ unique int priority

"random hash"

can assume for large enough co-domain

Tree priority

```
pr A = case expose A of
         Leaf => -∞
       | Node (_, x, _) => p(x)
```

Def Treap A satisfies: A is bin tree s.t. $\forall \text{Node}(L, x, R) \in A$, $p(x) > pr(L)$ $p(x) > pr(R)$

Thm Treap has $O(\lg n)$ depth whp.

Proof sketch similar to quicksort

IRV $$A_{i,j} = \begin{cases} 1 & \text{if rank } i \text{ is ancestor of } j \\ 0 & \text{else} \end{cases}$$

rank in tree

$$\mathbb{E}[A_{i,j}] = P[i \text{ ancestor of } j] = \frac{1}{|i-j|+1}$$

$\boxed{\text{Lec 13}}$ Balanced Binary Tree III Treaps

Treaps : - in treap ordering
- optionally must be a BST] Given both, treap unique

Distribution of tree shape

think picking pivot by assigning highest priority so far.

It's same as distribution of quicksort recursion tree

⇓

height of treap $\in O(\lg n)$ w.h.p.

Create RIV $A_j^i = \begin{cases} 1 & \text{if } S[i] \text{ is ancestor of } S[j] \text{ (inclusive)} \\ 0 & \text{else} \end{cases}$

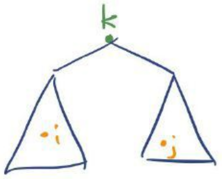
$$\text{depth}(j) = \sum_{i=0}^{n-1} A_j^i \qquad\qquad \text{size}(i) = \sum_{j=0}^{n-1} A_j^i$$

$$\mathbb{E}[A_j^i] = \mathbb{P}[\, S[i] \text{ is ancestor of } S[j]\,] = \frac{1}{|j-i|+1}$$

Intuition :

i and j are not ancestor of each other $\Leftrightarrow \exists\, k$ s.t. $p_k > \begin{cases} p_i \\ p_j \end{cases}$

⇒

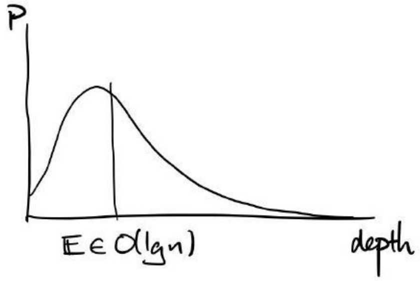


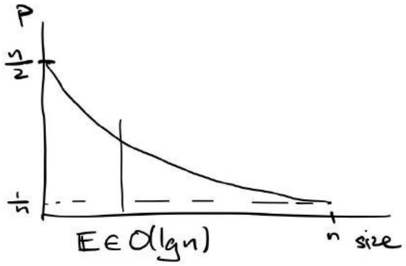
$$\mathbb{E}[\text{depth}(j)] = \sum_{i=0}^{n-1} \mathbb{E}[A_j^i] = \sum_{i=0}^{n-1} \frac{1}{|i-j|+1} = H_{j+1} + H_{n-j} - 1 \le 2H_n$$
$$\le 2\ln n + O(1)$$

$\mathbb{E}[\text{size}(i)] =$ --- sth similar --- $\le 2\ln n + O(1)$

Got to have depth$(j) \in O(\lg n)$ whp

BUT $\mathbb{E}[\text{size}(i)] \in O(\lg n) \not\Rightarrow \text{size}(i) \in O(\lg n)$ whp



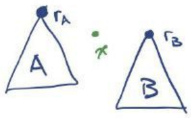


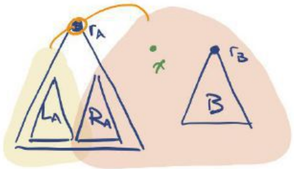
Treap Impl

track size

$T = \text{leaf} \mid \text{node of } T \times E \times \mathbb{Z} \times T$

```
mkNode (A, x, B) => node (A, x, size A + size B + 1, B)
joinM (A, x, B) => case
  p x > pr A and p x > pr B => mkNode (A, x, B)
  pr A > pr B => case expose A of
    leaf => raise Absurd   // we defined pr leaf = -∞
   | Node (L_A, r_A, R_A) => mkNode ( L_A, r_A, joinM( R_L, x, B )
```





$W \in O(\text{depth } A + \text{depth } B)$ ← each time we go down a level

$\subseteq O(\ln n \quad \ln n)$ $n = \text{depth } A + \text{depth } B$

$\subseteq O(\lg n)$ ← also $O(\lg n)$ whp.

joinM preserves BST property if $A < x < B$
preserves treap invar always

Table interface

Store key vals in tree, keep invariants by keys.
→ See documentation

Augmentation

Adding extra information in nodes (other than just balancing info)

Ex. dynamic paren matching

support: type paren = (|)
type dpm
insertAt dpm × paren × $\mathbb{Z}$ → dpm $O(\lg n)$
isMatched dpm → $\mathbb{B}$ $O(1)$?

→ Keep track of unmatched left & unmatched right at every node

Reduced value augmentation

1. Associate tree T with associative func $f : \mathbb{E} \times \mathbb{E} \to \mathbb{E}$ and its identity I.
2. Modify T to keep the "sum" of f at each node
3. Modify joinM to maintain the "sum"
4. Add func reduceVal : $T \to \mathbb{E}$ that returns the sum at root

<u>Impl</u>

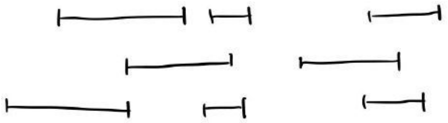
```
functor : E × f × I → augT
  T = leaf | node of T × E × E × Z × T
  reduceVal A = case expose A of leaf ⇒ I
                               | node(_, _, s, _, _) ⇒ s
  joinM (A, x, B) =
    node(L, x, f(x, f(reduceVal A, reduceVal B)), size A + size B + 1, R)
```

Lec 14 Treaps + Aug Table

Aug Tables

→ Treap with at each node : key, value, reduced value, size

Useful for ... eg. interval problems



Where is there 2 overlaps ?
Where is there ...

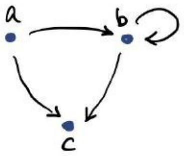
Graphs

Informal : verts connected by edges

More formally <u>directed graph</u> $G = (V, E)$, $n = |V|$, $m = |E|$

└ set of edges, represented by vert tuple
set of verts

Ex.



$G = (\{a,b,c\}, \{(a,b), (a,c), (b,c), (b,b)\})$

<u>Fact</u> $m \leq n^2$ (tight upper bound on num edges)

num distinct graphs with n verts ... $2^{(n^2)}$

<u>Undirected graph</u> $G = (V, E)$, $E \subseteq \binom{V}{2}$

└ set of sets with 2 verts

<u>Types of Graphs</u>

- Multgraph $G = (V, E)$, E is multiset

- Hypergraph $G = (V, E)$, $E = \mathcal{P}(V)$

↑
so an edge can link ≠ 2 verts
normal graph ⇒ 2-uniform hypergraph

- Bipartite graph $G = ((U, V), E)$, $E \subseteq U \times V$, $|U| = n_u$, $|V| = n_v$

  Fact there are $2^{(n_u \cdot n_v)}$ distinct undirected bipartite graphs

Applications

- Utility graph — electricity, internet, water, gas, ...
  verts ← location   edges ← connections
- Dependence graph — compiler control flow,
- Social network graph
- Taxonomy graph — phyrogenetics, evolution
- Mesh network
- Markov chain
- documents with links
- state graph

Mathematical Defs

Def $N_G(u)$ is neighbourhood of u in G $= \{v \in V \mid \{u, v\} \in E\}$

$N^+_G(u)$ is the outgoing nbors $\{v \in V \mid (u, v) \in E\}$
$N^-_G(u)$ ... $\{v \in V \mid (v, u) \in E\}$] for directed

$\deg(u) = |N_G(u)|$

$\deg^+(u) = |N^+_G(u)|$
$\deg^-(u) = |N^-_G(u)|$

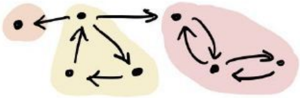
Def Path is an alternating seq of verts & edges
Length of path is num edges in path
Simple path is path without repeating vert nor edge
Cycle starts and end at same vert
Simple cycle cycle without repeating vert nor edge except at start

Def $\delta_G(s, v)$ = Len of shortest path from s to v in G

$R_G(u,v)$ = v reachable from u to v
viz. $\exists$ path from u to v

<u>Connected component</u> is a subset of verts s.t. every vert is reachable from every other vert] undirected

<u>Strongly connected component</u> is subset of verts s.t. every vert is reachable from every other vert] directed



<u>Def</u> <u>Forest</u> — graph without cycle] undirected

<u>Tree</u> — forest with one connected component]

<u>DAG</u> — directed acyclic graph

Graph representations

- Edge set

  (V set , (V×V) set) for some set repr

  Edge membership query → whatever lookup cost is in set repr
  Check nborhood → filter edges set then map to extract... $O(m)$

- Adjacency matrix

  ((V×V) key , $\mathbb{B}$ value) table

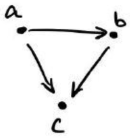
  Good for dense graph
  Bad for sparse graph

  <u>Def</u> G is dense if $m \simeq n^2$
  sparse otherwise

- Adjacency set

  (V key , V set) table

  {
  a : {b, c},
  b : {c},
  c : {}
  }

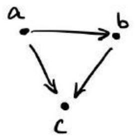


- Adjacency seq

  Define $\{0, \dots, n-1\} \leftrightarrow V$

  int seq seq

  $\langle \langle 1,2 \rangle, \langle 2 \rangle, \langle \rangle \rangle$



- Adjacency list

Lec 15 Graph Search / Graph Traversal

Recall: $G = (V, E)$
$N_G(u) = \{ \text{nbors of } u \}$
$d_G(u) = \text{degree of } u$
$\delta_G(u, v) = \text{distance from } u \text{ to } v$

Generic graph search

Def Graph search / traversal is when we systematically examine nodes in a graph starting from some vert v.

Def $R_G(s) = \{ v \in V \mid v \text{ reachable from } s \}$

Generic traversal

Keep track of:

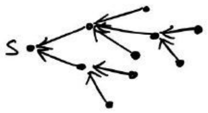
- visited $X = \{ \text{set of visited} \} \subseteq V$
- frontier $F = \{ \text{next to some visited node but not visited} \} \subseteq V \setminus X$

Alg:

```
search G s :
  X = {}
  F = {s}
  while |F| > 0 :
    U = some non-empty subset of F
    visit everything in U
    X = X ∪ U
    F = $N^+_G(X) \setminus X$
  return X
```

Thm search G s returns $R_G(s)$

Def graph search tree is a graph built by
for $v \in R_G(s)$:
create edge from v to the vertex that
added it to v (or some vert if race condition)



Cost Analysis

$X = X \cup U$ ← union
$F = N^+_G(X) \setminus X$ ← finding nbors & set diff

<u>Claim</u> cost is dominated by $N^+_G(X)$
(assume for now)

$$N^+_G(X) = \bigcup_{v \in X} N^+_G(v)$$

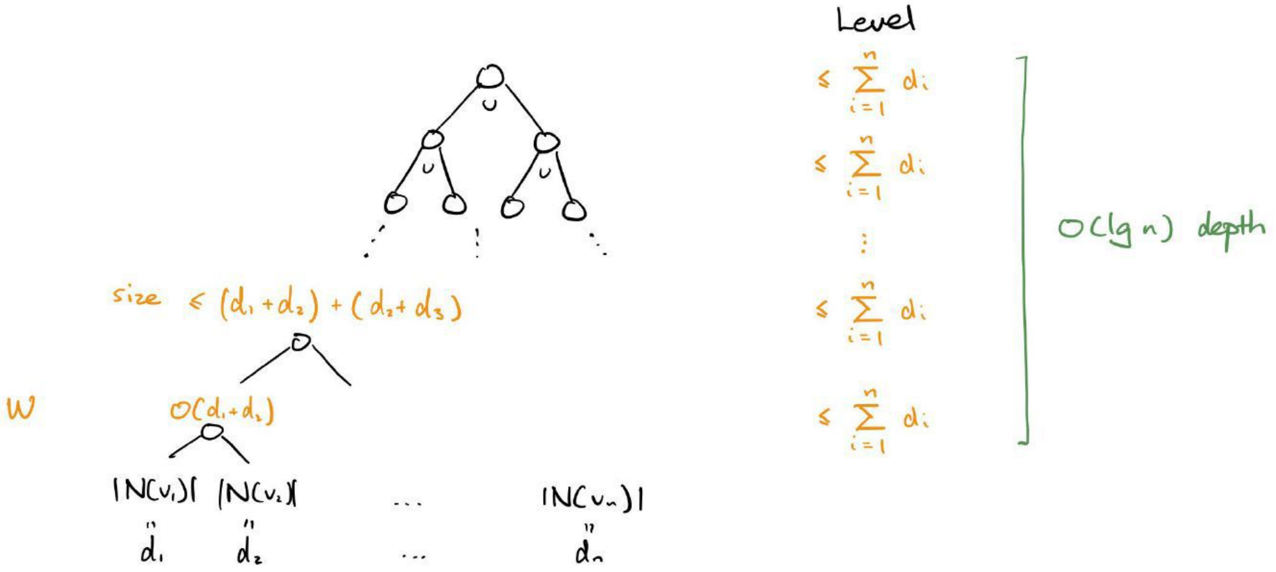
$$= \text{reduce union } \emptyset \ \langle N^+_G(v) : v \in X \rangle$$

Recall Assume $a = |A| \le b = |B|$

(exercise)

$$W_{union}(a,b) = O(a \lg(\tfrac{b}{a}+1)) \le O(a+b)$$
$$S_{union}(a,b) = O(\lg(a+b))$$

Reduce union recursion tree



But $\sum_{i=1}^{n} d_i \le O(m)$ so total work $O(m \lg n)$

⋮

Span : at level $\le O(\lg n)$
overall $O(\lg^2 n)$ ← (is tight bound)

Parallel BFS

When $U = F$

```
BFS G s = let
  loop (X, F, i) =
    if |F| = 0 then (X, i)
    else △ let
      $X' = X \cup F$
      $F' = \underbrace{N^+_G(F) \setminus X'}$
    in
      loop (X', F', i+1)
    end
  in
    loop ({}, {s}, 0)
  end
```

for BFS this ≡ $N^+_G(X') \setminus X'$
actually lower cost to get all nbors

Claim : at △ ,
$X_i = \{ v \in V, \ \delta(s,v) < i \}$
$F_i = \{ v \in V, \ \delta(s,v) = i \}$

$\cdot i := \cdot$ when loop called with counter i

Proof It feels right □ (no)

Proof by induction

BC $i = 0$, $X_0 = \emptyset$. ✓
$F_0 = \{s\}$. ✓

IS Assume for i , WTS for i+1

$$\begin{aligned} X_{i+1} &= X_i \cup F_i \\ &= \{ v \in V, \ \delta(s,v) \le i \} && \text{(IH)} \\ &= \{ v \in V, \ \delta(s,v) < i+1 \} && \checkmark \end{aligned}$$

$$\begin{aligned} F_{i+1} &= N^+_G(F_i) \setminus X_{i+1} \\ &= \{ v \in V, \ \delta(s,v) = i+1 \} && \checkmark \end{aligned}$$

BFS on line graph

iterations $O(n)$, $O(\lg n)$ at each iter $W = O(n \lg n)$
$S = O(n \lg n)$

BFS cost

let $\|F\| = \sum_{x \in F} (d^+(x) + 1)$ assume tree set

for iter i

	W	S
- $X_i \cup F_i$	$O(\lvert F_i \rvert \lg n)$	$O(\lg n)$
- $N_G^+(F_i)$	$O(\|F_i\| \lg n)$ same analysis as above	$O(\lg^2 n)$
- $\setminus X_{i+1}$	$O(\lvert F_i \rvert \lg n)$	$O(\lg n)$

$\boxed{\text{Lec 16}}$ Graph Search Cont

Parallel BFS cost analysis cont.

<u>BFS cost</u>

Let $\|F\| = \sum_{x \in F} (d^+(x) + 1)$ assume tree set

for iter i

	W	S
$- X_i \cup F_i$	$O(\|F_i\| \lg n)$ $\lvert F\rvert \leq \lvert X\rvert \Rightarrow O(\lvert F\rvert \lg(\frac{\lvert X\rvert}{\lvert F\rvert}+1)) \leq \lvert F\rvert \lg n$ $\lvert F\rvert > \lvert X\rvert \Rightarrow O(\lvert X\rvert \lg(\frac{\lvert F\rvert}{\lvert X\rvert}+1)) \leq \lvert F\rvert \lg \lvert F\rvert \leq \lvert F\rvert \lg n$	$O(\lg n)$
$- N_G^+(F_i)$	$O(\|F_i\| \lg n)$ same analysis as above	$O(\lg^2 n)$
$- \setminus X_{i+1}$	$O(\|F_i\| \lg n)$ similar to union	$O(\lg n)$

Over all loops. say max depth in search is d

$$W = O\left(\sum_{i=0}^{d} (\|F_i\| \lg n)\right)$$

$$= O\left(\lg n \sum_{i=0}^{d} \|F_i\|\right)$$

$$= O(\underline{\lg n} \cdot (m+n))$$

using $\|F\| = \sum_{x \in F} d^+(x) + 1$

$= \sum_{x \in F} d^+(x) + \sum_{x \in F} 1$

$\vdots$

$= m + n$

$S = O(d \, \underline{\lg^2 n})$ * one $\lg n$ can be eliminated using some set data struct

DFS

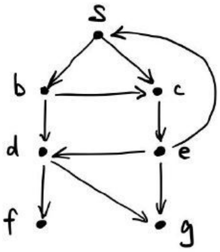
When v = most recently seen vert in frontier

Recursive impl

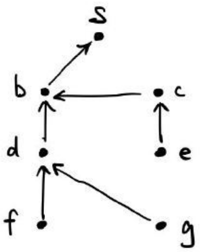
```
DFS G s =
  let
    DFS' (X, v) = if $\underbrace{v \in X}_{1}$ then X
                  else iterate DFS' $\underbrace{(X \cup \underbrace{\{v\}}_{2})\ N^+(v)}_{3}$
  in
    DFS' ({}, s)
  end
```

- inherently sequential!
- in fact DFS believed to be P-complete
- and believed that P-complete probs don't have polylog span sol

Example

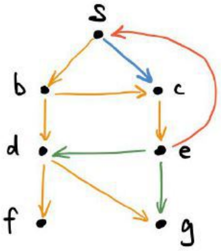


Possible order: s, b, d, f, g, c, e
↪ induced search tree:



DFS edge types

- Tree edge — u → v if v visited from u in DFS
  viz. reversed edges in search tree
- Back edge — edge that go back to ancestor in DFS tree that's not tree edge
- Forward edge — edge that go to descendent in DFS tree that's not tree edge
- Cross edge — none of above, they cross btwn branches



Note these four partition the edges in G

Generic DFS

Notice we may want to do some analysis while computing 1,2,3
We can let caller provide function for those computation

- application state Σ
- transition funcs $\Sigma \times V \to \Sigma$
  - visit — called when first visiting a vert
  - finished — called when done iterating over $N^+(v)$
  - revisit — if already visited

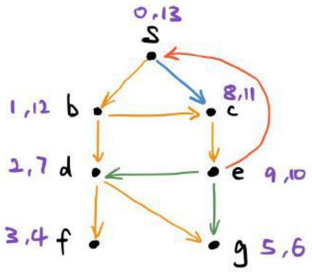
```
DFS G ((Σ, X), v) =
  if $v \in X$ then $\underbrace{(\text{revisit}(\Sigma, v), X)}_{1}$
  else let
    $\Sigma' = \underbrace{\text{visit}(\Sigma, v)}_{2}$
    $X' = X \cup \{v\}$
    $(\Sigma'', X'') = \text{iterate } (\text{DFS } G)\ (G', X')\ N^+(v)$
  in
    $\underbrace{(\text{finish}(\Sigma'', v), X'')}_{3}$
  end
```

Sometime we want

$$\text{DFSALL}\,(G = (V, E))\ \Sigma = \text{iterate}\ (\text{DFS } G)\ (\Sigma, \{\})\ V$$

Ex. application

→ DFS numbering : track visit / finish timestemp



Using our framework:

$$\Sigma : \underset{\substack{\uparrow \\ \text{curr time}}}{\text{int}} \times \underset{\substack{\uparrow \\ \text{visit time}}}{(V, \text{int})\ \text{table}} \times \underset{\substack{\uparrow \\ \text{finish time}}}{(V, \text{int})\ \text{table}}$$

$$\text{visit}\ ((t, V, F), v)$$
$$= (t+1,\ \text{insert}\ V\ (v, t),\ F)$$

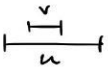
$$\text{finish}\ ((t, V, F), v)$$
$$= (t+1,\ V,\ \text{insert}\ F\ (v, t))$$

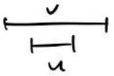
$$\text{revisit}\ ((t, V, F), v)$$
$$= (t, V, F)$$

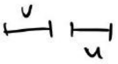
→ Determine edge types (reduced to start / finish time)

Keep set of tree edge in a set in Σ

<u>Claims</u>

e = (u, v) forward edge ⇔  ∧ e not tree edge

e = (u, v) back edge ⇔  ∧ e not tree edge

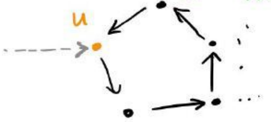
e = (u, v) cross edge ⇔  viz. finished searching target of e before going to the source

→ Cycle detection (reduced to edge type)

<u>Claim</u> G has cycle ⇔ ∃ back edge

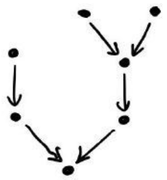
(⇐) Trivial

(⇒) Fix $\underbrace{\text{first time encountering vert}}_{u}$ in some cycle, then at later point we visit an incoming edge to that vert



← must visit this at later point

→ Topological sort



Given DAG G = (V, E)

Observe it define partial order

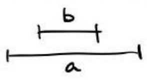
$a \leq_R b \iff b$ reachable from $a \wedge a \neq b$

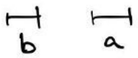
Want to sort V s.t. it respects $\leq_R$

<u>Lemma</u> DAG finish time:

if $a \leq_R b$, ∀ DFS, b finish ~~finish~~ before a

<u>C1</u> b visited before a <u>C2</u> a visited before b





<u>Alg</u> run DFSAll, return reverse finish time order

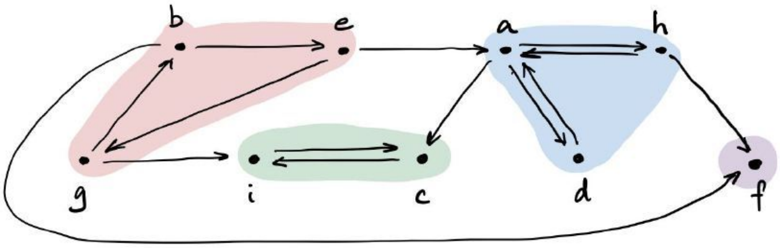
$\boxed{\text{Lec 17}}$ Kosaraju's Alg & Shortest path

Strongly connected component (SCC)

Def A subset of verts $S \subseteq V$ is strongly connected (SC) if $\forall v \in S, \forall u \in S$, u reachable from v

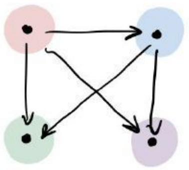
Def If $S \subseteq V$ is SC and is maximal, it's a strongly connected component

Ex.



Claim Contracting the SSCs gets you a DAG ← else one of those SSC not maximal.

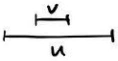
↳ turn SSCs into verts and add edge by reachbility btwn components



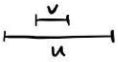
Def The SCCs problem: find the SCCs in graph and return them in topological order

Lemma For directed graph G, if $u \in V$ is first-visited vert in its SCC, all verts v reachable from u finish before u in a DFS

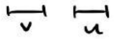
C1 v, u in same SCC $\Rightarrow$



C2 v, u in diff SCC $\wedge$ u visited before v $\Rightarrow$



C3 v, u in diff SCC $\wedge$ u visited after v $\Rightarrow$



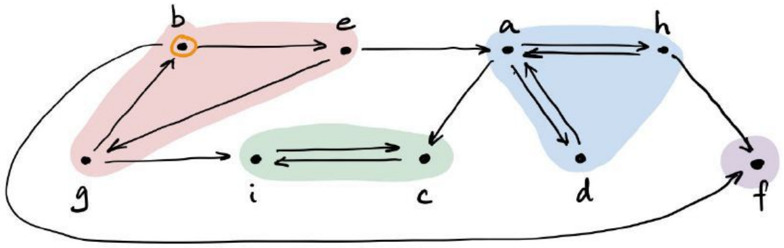
... otherwise u, v in same SCC so we can't go back

Kosaraju's Algorithm

```
SCC G = (V, E) =
  let
    F = reverseFinishTime G
    G^T = transpose G                    ( reverse the edges )
                 visited verts   SCC so far
    accumSCCs ( (X, L), u ) =
      let  overall visited
        (X', A) = reach G^T X u
             newly visited   search for all reachable from X
        ▷ If A = ∅ ⇒ we saw new SCC
      in
        if A = ∅ then (X, L) else
        (X', L ++ ⟨A⟩)
  in
    iterate accumSCC (∅, ⟨⟩) F
  end
```

<u>Cost</u> 2× DFS, so actually linear time

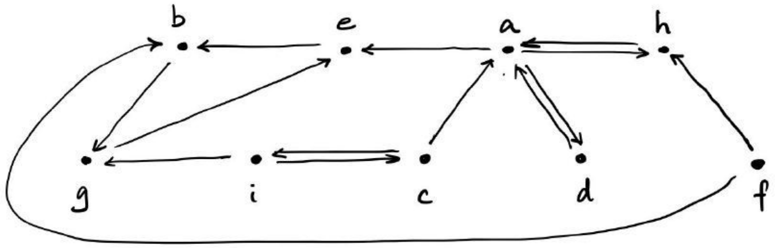
<u>Trace</u>



F ⇒ ⟨ b e g a h f d c i ⟩

latest finish

G^T ⇒



	A	L
reach G^T $\emptyset$ b $\Rightarrow$	$\{b, e, g\}$	$\langle \{b, e, g\} \rangle$
reach G^T $\{b, e, g\}$ e $\Rightarrow$	$\emptyset$	..
reach G^T $\{b, e, g\}$ g $\Rightarrow$	$\emptyset$	..
reach G^T $\{b, e, g\}$ a $\Rightarrow$	$\{a, d, h\}$	$\langle \{b, e, g\}, \{a, d, h\} \rangle$
.. $\{b, e, g, a, d, h\}$ h $\Rightarrow$	$\emptyset$	..
.. .. f $\Rightarrow$	$\{f\}$	$\langle \{b, e, g\}, \{a, d, h\}, \{f\} \rangle$
.. $\{b, e, g, a, d, h, f\}$ d $\Rightarrow$	$\emptyset$	..
.. .. c $\Rightarrow$	$\{c, i\}$	$\langle \{b, e, g\}, \{a, d, h\}, \{f\}, \{c, i\} \rangle$

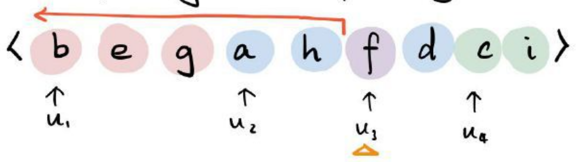
{ no more to add ↓

Correctness

Observe:
When first reach each SCC U_i via vert u_i in rev finish order:

1. SCCs left of U_i already completely visited
2. reach G^T u_i will not visit any SCCs to right of U_i
3. reach G^T u_i will visit all verts in U_i

where left / right identified by first appearance of vert in SCC in F



Notice 1-3 ⇒ reach G^T u_i visit exactly U_i

If △ is current, all of ⟵ unreachable from △ in G
otherwise F is not rev finish time order
⇒ △ unreachable from any of ⟵ in G^T

Shortest path problem

Def weighted graph has some weight for each edge

$G = (V, E, w)$ $w : E \to \mathbb{R}$

one representation G : $(V, (V, \mathbb{R})$ table $)$ table

$\delta(u, v) :=$ shortest path with min edge weights from u to v

Def Single-pair shortest path problem (SPSP)

Given u, v, find $\delta(u, v)$

Def Single-source -- (SSSP)

Given u, find $\delta(u, v)$ $\forall v$ reachable from u

Def All-pairs --

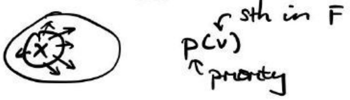
$\forall u, \forall v$, find $\delta(u, v)$

Priority-first search (PFS) aka best-first search

Decide where to search by order returned by some priority func

viz. pick $v \in F$ by highest priority

Ex. beam search, A*, dijkstra



Dijkstra's property:

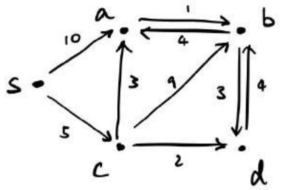
if $\nexists$ neg edge weights in G, let $p(v) = \min_{u \in X} (\delta(s, u) + w(u, v))$

then $v \in V \setminus X$ with smallest $p(v)$ has $\delta(s, v) = p(v)$

Dijkstra's algorithm:

use the above p as priority in PFS (record $p(v)$ for each vert v when we visit v)

Ex.



s	a	b	c	d
0	∞	∞	∞	∞
	10	∞	5	∞
	8	14		7
	8	11		
		9		

$\delta(s, s) = 0$

s c = 5

s d = 7

Lec 18 More Dijkstra & Bellman - Ford

Priority First Search

```
search G s =
  X = {}        F = {s}
  init
  while |F| > 0:
    v = $\underbrace{\min_{x \in F} \rho(x)}$
    visit v
    X = $X \cup \{v\}$
    F = $N_G(x) \setminus X \;=\; (F \setminus \{v\}) \cup (N^+_G(v) \setminus X)$
  return X
```

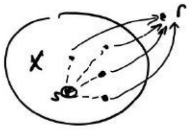
Dijkstra property

If no negative weight edges and define priority

$$p(v) = \min_{v \in X} (\delta(s,v) + w(v,r))$$

$$Y = \operatorname{argmin}_{v \in V \setminus X} p(v)$$

then $p(Y) = \delta(s, Y)$



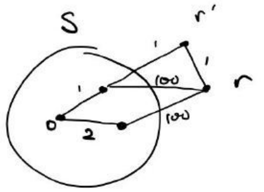
Dijkstra's init:

$d(s) := 0 \qquad d(x) := \infty$ for $x \in V \setminus \{s\}$

$p(v) = \min_{x \in X} (d(x) + w(x,v))$

visit $d(v) = p(v)$

Ensures: $d(v) = \delta(s,v)$



<u>Impl</u>

```
dijkstraPQ G s =
  let loop X Q =
    case delmin Q of
      (NONE, _) => X
     |(SOME(d,v), Q') =>
       if (v, _) ∈ X then loop X Q'
       else let
```

like an augmented frontier with extra verts, or even duplicate verts due to diff dist to source by diff paths

priority queue Q
insert $Q \times (\mathbb{Z} \times \mathbb{N}) \to Q$
delmin $Q \to (\mathbb{Z}, \mathbb{N})$ option

```
        X' = X ∪ {(v,d)}
        relax (Q, (u,w)) = insert (Q, (d + w, u))
        Q" = iterate relax Q' (N⁺_G(v))
      in
        loop X' Q"
      end
  in
    loop {} (insert emptyQ (0,s))
  end
```

Dijkstra cost analysis

Observe parallelism maybe possible in ——— for equal weight batches and ——— with batch insertion enabled queue, but in general sequential.

every edge could cause insert

Operation	Number	Work
Q. delmin	m	lg n (= lg m)
Q. insert	m	⋮
T. find	m	⋮
T. insert	n	⋮
$N^+_G(v)$ (T. find)	n	⋮
Calls to iterate	m	O(1)
Total work	$O((m+n)\lg n)$	

With fancy pqueue, $O(m + n\lg n)$ possible

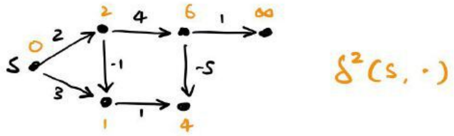
could be O(1), but above still $O(\lg m)$

Bellman - Ford — min dist on arbitrary graphs, but less efficient

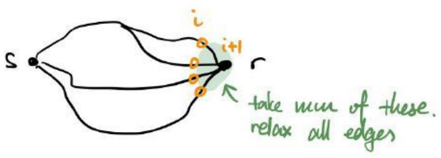
Ex. - convert from other probs lead to graph with neg edge
- find best way to convert currency
  when we want max product, take neg log and find min path, so we could get negatives

Intuition : $\delta^k(s,v)$ = shortest path $s \to v$ with max of k edges

given $\delta^i(s,v)$ for all v
get $\delta^{i+1}(s,v)$ by trying
all next edge ← parallel!



get global : find δ^{n-1} *
* if not done at $i \geq n$ then we got neg cycle



```
BF $G = (V, E)$ s =
  let loop (D: (V, $\mathbb{R}$) table) k =    just to prevent D[s] → ∞
    let $D' = \{ v \mapsto \min( D[v] , \min_{u \in N_G^-(v)} D[v] + w(u,v)) : v \in V \}$
    in
      if (k = |v|) then NONE    ← neg weight cycle
      else if D = D' then SOME D
      else loop D' (k+1)
  in
    loop $\{ s \mapsto 0 \} \cup \{ v \mapsto \infty \mid v \in V \setminus \{s\} \}$ 0
  end
```

$W(n,m) = O(mn)$ $S(n,m) = O(n \lg m) = O(n \lg n)$

Lec 19 Johnson's Algorithm & Graph Contraction

So far:

	Work	Span	Parallelism
Dijkstra	$O(m \lg n)$	$O(m \lg n)$	None
Bellmond-Ford	$O(mn)$	$O(n \lg n)$	$O(\frac{n}{\lg n})$

Those are single source shortest path (SSSP) problems
Asymptotically no better way to find SP given source and target than SSSP

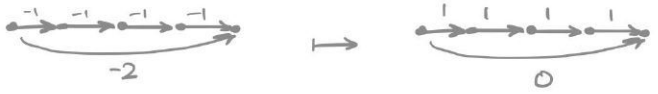
What if we want all pairs shortest path (APSP) ?

Brute force with Dijkstra ?

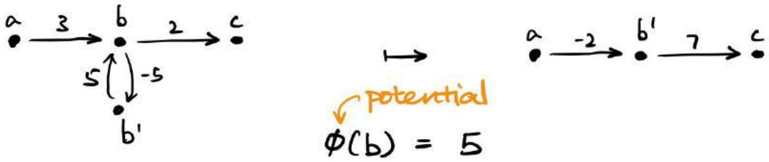
Johnson's Alg

Changing edge weight

Naive: add weight to each edge :C Bad

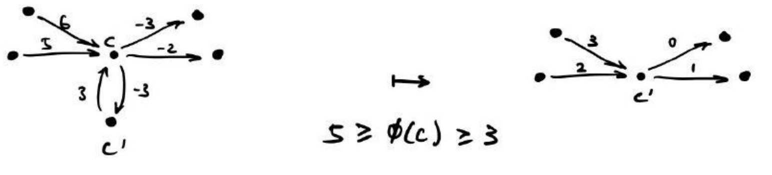


Potential property:

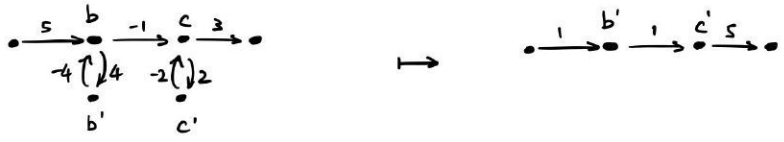


potential

$\phi(b) = 5$



$5 \geq \phi(c) \geq 3$



General case — reweight all edges by potential

$$w(u', v') = w(u, v) + \overset{\text{potential}}{\phi(u)} - \phi(v)$$

Claim $\delta(u', v') = \delta(u, v) + \phi(u) - \phi(v)$

$(\Leftrightarrow \delta(u, v) = \delta(u', v') - \phi(u) + \phi(v))$ △

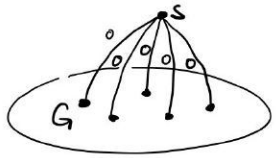
Shortest path relax property

(a,b) relaxed if $\delta(s,a) + w(a,b) \geq \delta(s,b)$

$\Leftrightarrow w(a,b) \geq \delta(s,b) - \delta(s,a)$

$\Leftrightarrow w(a,b) + \delta(s,a) - \delta(s,b) \geq 0$ ← looks like potential

Dummy Vertex



Reduction

1. Pick potentials so that all weights ≥ 0
   i. Add dummy verts s & edges of 0 weight
   ii. Run Bellmond-Ford to get $\delta(s, \cdot)$
   iii. $w(u', v') = w(u, v) + \delta(s, u) - \delta(s, v)$
2. Dijkstra from all source
3. Recover path lengths with △

Cost

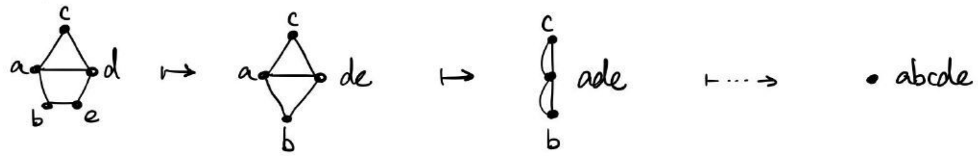
	Bellmond - Ford	Dijkstra	Overall
Work	$O(mn)$	$n\,O(m \lg n)$	$O(mn \lg n)$
Span	$O(n \lg n)$	$O(m \lg n)$	$O(m \lg n)$

Graph contraction — gives you polylog span

→ Generally search-based algs not good for parallelism. Think line graph causing troubles...

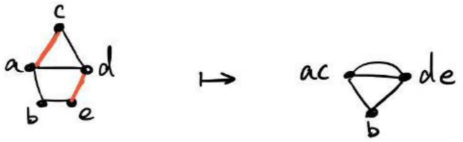
Useful for:
- Graph connectivity
- Min spanning tree
- biconnected components

Edge contraction



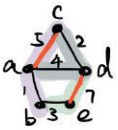
Contract by constant factor: find maximal matching and contract

greedy works, but is sequential :c



Parallel mostly maximal matching

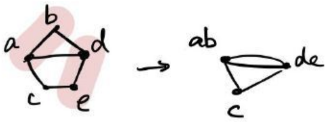
Random priority,
pick max among
touching edges



Lec 20 Star Contraction & Connectivity

Graph contraction

parallelism, polylog span, root dominated work
contract to get constant fraction smaller



Defs Graph partition := subgraph $H = (V', E')$ with $V' \subseteq V$ and $E' = \{\{u,v\} \in E \mid u, v \in V'\}$ viz. cut out verts and keep reasonable edges

Given partitions $H_1, \dots, H_{k-1}$, $\{u, v\} \in E$ is
- internal edge if $u, v \in V_i$
- cut edge if $u \in V_i$, $v \in V_j$, $i \neq j$

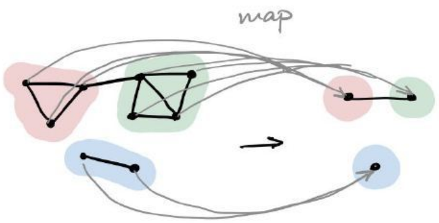
Quotient graph is contracted, smaller graph

Supervert is vert in quotient that verts in orig graph "merged" to

Repr
1. Label for each part
2. Map from vert to their part label

General Contraction

BC Small graph ⇒ compute result



IC **Contract** Make quotient
- Partition
- Turn part into vert
- Drop internal edges
- Point cut edges elsewhere (maybe remove dups)

Recur Solve on contracted graph

Expand Get result for bigger graph

Edge contraction

When each part is a vert or one edge.

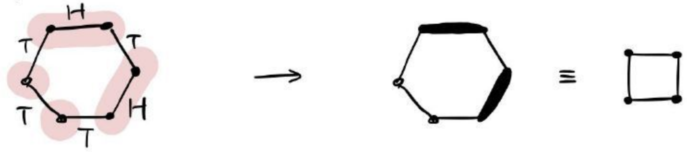
First we need to find matching

→ Greedy : for $e \in E$, keep adding e to M if possible ↙ sequential, always within factor of 2 of optimal

→ Random : parallel assignment, local decisions

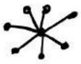
Coin flip

Flip coin for each edge, contract head s.t. no neighbouring edge is head



This gives constant fraction on some graph but not others

→ Cycle graph  each edge $\frac{1}{8}$ prob contracted , so $\mathbb{E}$ contract $\frac{m}{8}$

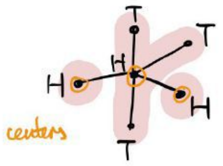
→ Star graph  then we only contract max 1 edge .

Star Contraction

Each part looks like a star with center & satellites

→ Sequential : pick center, add all nbors as satellites, remove, repeat

→ Random : flip coin on each vert, turn H into centers, for each T try to contract into neighbouring H, then turn T that failed to merge into center



```
starPart (G = (V, E)) =
  let
    TH = $\{(u,v) \in E \mid u \text{ tail} \wedge v \text{ head}\}$
    $P_s$ = $\bigcup_{(u,v) \in TH} \{u \mapsto v\}$        ← point sats to centers
    $V_c$ = $V \setminus \text{domain}(P_s)$        ← center verts
    $P_c$ = $\{u \mapsto u : u \in V_c\}$        ← center contract to center
  in
    $(V_c, P_s \cup P_c)$
  end
```

Cost $W = O(n+m)$] with array seq

$S = O(\lg n)$

Fact for G with n non-isolated verts, $\mathbb{E}$ satellites $\geq \frac{n}{4}$

Contraction alg

```
starContract base expand $(G = (V, E))$ =
  if $|E| = 0$ then base $V$
  else let
    $(V', P)$ = starPart $(V, E)$          remove celf edge
    $E' = \{(P[u], P[v]) : (u,v) \in E \mid P[u] \neq P[v]\}$
    $R$ = starContract base expand $(V', E')$
  in
    expand $(V, E, V', P, R)$
  end
```

Cost (star contract until $|E| = 0$)

Assume : $W_{base} = O(|V|)$ $S_{base} = O(1)$

$W_{expand} = O(|V| + |E|)$ $S_{expand} = O(\lg(|V| + |E|))$

$W = O((m+n) \lg n)$ $S = O(\lg^2 n)$

Application : Graph Connectivity

Prob Given undirected G, find all CCs by specifying them as vert set

→ Could do BFS or DFS, but slow

Contraction alg

```
connectedComponents $(G = (V, E))$ =
  if $|E| = 0$ then $(V, \{v \mapsto v : v \in V\})$
  else let
    $(V', P)$ = starPart $(V, E)$
    $E' = \{(P[u], P[v]) : (u,v) \in E \mid P[u] \neq P[v]\}$
    $(V'', C)$ = connectedComponents $(V', E')$
  in
    $(V'', \{u \mapsto C[v] : (u \mapsto v) \in P\})$
  end
```

Lec 21 Min Spanning Tree

MST

__Def__ Given undirected, connected graph $G = (V, E)$, a spanning tree is tree $G' = (V, E')$ with $E' \subseteq E$

A min spanning tree (MST) for undirected connected weighted graph $G = (V, E, w)$ is $ST = (V, E')$ of G with min weight sum for E'.

__Apps__
- Connecting things with min cost
- Approximate TSP

__Prop__ Given tree.

add edge creates exactly __one cycle__

then removing any edge in this cycle creates tree again

__Prop__ Light edge property

unique weights

$\forall$ undirected, conn, weighted G with $|V| \geq 2$,
$\forall\ U \subseteq V,\ |U| \geq 1$,
the min edge e from U to $V \setminus U$ is in MST

__Proof__ __C1__ If e is only edge btwn U and $V \setminus U$ then duh

__C2__ Else AFSOC $e \notin MST$, then $\exists\, e' \in MST$ that goes btwn U and $V \setminus U$, $e' \neq e$, and e' forms cycles with e if e added to MST

Add e to the MST and remove e'. We still get spanning tree but costs less. ※

__Prop__ Heavy edge prop

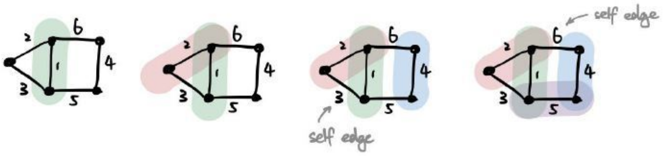
the heavest edge in any cycle is not in the MST

MST Algs

All $O(m \lg n)$ work, span maybe different

Kruskal

```
sort edges by weight
for i in 0..n-1 :
  if (u,v) = E[i] not self edge :      ← check using union find
    contract (u,v) , add (u,v) to MST
```



Prim

← same cost analysis as Dijkstra

PFS with $p(v) = \min_{x \in X} w(x,v)$

Sleator-Tarjan

← $\exists \lg n$ method to find heavy edge in cycle

```
for e = (u,v) ∈ E :
  add e to MST
  if new cycle formed :
    remove heaviest from that cycle
```

Borüvka (1926 ish), parallel

```
boruvka (G = (V, E, w)) =
  if |E| then ∅
  else
    for every vert find min weight e
    add e to MST
    G' = contract all edges identified
  in
    recur on G'
  end
```

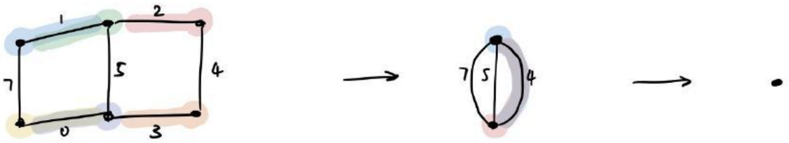
Cost:

Every step reduce by at least $\frac{1}{2}$
so worse case $\lg n$ steps

per iter

	W	S
Find min	m	$\lg n$
Contract	m	$\lg^2 n$
Total	$m \lg n$	$\lg^3 n$

↑ $\lg^2 n$ possible using star contraction



Lec 22 Dynamic Programming (DP)

Idea: solving subinstances and saving results in useful way

General structure

0. Start with some decision / optimisation / counting problem

is there path with len at most..

find shortest path

how many paths with given property

things like data transformation may not be one of these

1. Develop recursive solution

2. Recognise how to reuse results from subinstances

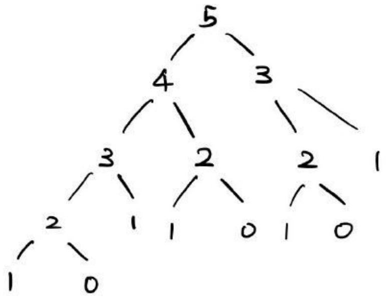
3. Count num of unique subinstances for analysis

4. Implement
   - Memoisation
   - Bottom-up

Fibonacci example

fib(n) = if $(n \le 1)$ then 1 else fib$(n-2)$ + fib$(n-1)$

Call tree fibb 5



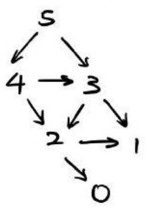
work $O(\phi^n)$

span $O(n)$

$\phi = \frac{\sqrt{5}+1}{2}$

Bad :C

Result dependency



So we need $6 = n+1$ unique instances

work $O(n)$

span $O(n)$

Bottom-up Impl

```
fib n =
  let
    loop a b k =
      if k = n then a
      else loop b (a+b) (k+1)
  in
    loop 1 1 0
  end
```

Subset sum problem (SS)   NP-hard

Given set $S \subseteq \mathbb{Z}^+$ and $k \in \mathbb{Z}^+$, is there $X \subseteq S$, $\sum_{x \in X} x = k$?

→ Actually the base for some crypto system that was broken

Even though NP-hard it's easy to find sol for some input.
Pseudopoly — polynomial to k, so if k itself poly to |S| we get poly to |S|

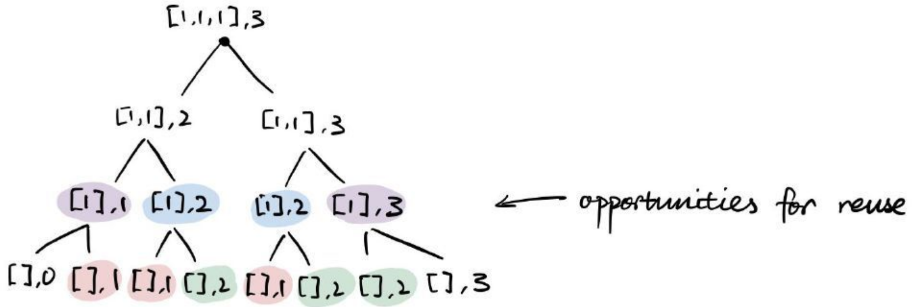
Recursive sol

hard to memoise as key

```
SS (S, k) = case (S,k) of
   (_, 0) => true
  | ([], _) => false
  | (x::xs, k) => if k < x then SS(xs, k) else
                  SS(xs, k-x) orelse SS(xs, k)
```

$W(n) = 2W(n-1) + O(1)$ exponential !

Ex. $S = [1,1,1]$ $k = 3$



$k' \in \{0, \ldots, k\}$, $|S'| \in \{0, \ldots, |S|\}$
so num unique subinstances is $(|S|+1)(k+1)$

If reuse results, work $O(|S| k)$
span $O(|S| + 1)$

Representing lookup table

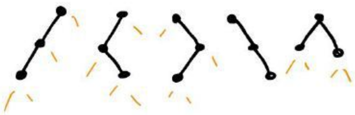
Our table wants to have subinstances as key, but if input has list how to hash / compare list? Expensive!

In practice try convert subinstance to integer

```
SS (S, k) =
  let
    n = |S|          use this as key to table, or even 2D array
    SS' (i, k') = (case (i, k') of
       (_, 0) => true
      | (n, _) => false                      ▷ should look up here
      | (i, k') ▽ => if (k < S[i]) then SS' (i+1, k) else
                     SS' (i+1, k'-S[i]) orelse SS' (i+1, k')
  in
    SS' (0, k)
  end
```

Counting problem example

Count number of rooted binary tree shape with size n

n	shapes	count
0	∅	1
1	•	1
2		1
3		5

$$T(n) = \begin{cases} 1 & \text{if } n \le 1 \\ \underset{\text{left subtree size}}{\sum_{i=0}^{n-1}} \underbrace{T(i)}_{\text{left subtree}} \underbrace{T(n-i-1)}_{\text{right subtree}} & \text{else} \end{cases}$$

Num unique subinstances = $n + 1$

Work per subinstance — $O(n)$ to do the sum
span — $O(\lg n)$ reduce op+

Overall work $\sum_{i=0}^{n} O(i) \in O(n^2)$

$\sum_{i=0}^{n} O(\lg i) \in O(n \lg n)$

Lec 23 More DP

Min Edit Dist Problem

Minimise the numbers of insertions and deletions to go from S:str to T:str

Ex. ABCADA → ABADC in 3 edits

Alg:

```
MED(S, T) =
  let
    MED'(i, j) = case (i, j) of                      len-i prefix of S
      (0, j) ⇒ j                                      "  -j    "   of T
     |(i, 0) ⇒ i
     |(i, j) ⇒ ⎧ MED'(i-1, j-1)               if S[i-1] = S[j-1]
             △ ⎨ min ⎧ MED'(i-1, j) + 1 ⎫     else
               ⎩     ⎩ MED'(i, j-1) + 1 ⎭
  in
    MED'(|S|, |T|)          △ Exponential, but allows subinstance result
  end                         reuse here
```

Analysis

$(|S|+1)(|T|+1)$ unique subinstances
each subinstance has constant local work
so $O(|S||T|)$ work, $O(|S|+|T|)$ span

Bottom up impl

		0	1	2	3	4	5	6
		-	A	B	C	A	D	A
0	-	0	1	2	3	4	5	6
1	A	1	0	1	2	3	4	5
2	B	2	1	0	1	2	3	4
3	A	3	2	1	2	1	2	3
4	D	4	3	2	3	2	1	2
5	C	5	4	3	2	3	2	3
6	A	6	5	2	3	2	3	$\boxed{2}$

Memoisation impl

~~(Magic)~~ Memoised version of f

```
fun f g (i,j) = case (i,j) of
    (0,j)  => j
  | (i,0)  => i
  | (i,j)  => { g (i-1, j-1)                      if  S[i-1] = S[j-1]
              { min { g(i-1, j) + 1 }             else
              {     { g(i, j-1) + 1 }
```

Memoiser lib

```
val MED' = memoiser.memoise (f)
```

Memoisation lib

```
fun memoise f =
  let
    val cache = ref (Table.empty ())
    fun g a = (case find (!cache, a) of
       SOME r => r
      |NONE => let
         val r = f g a
         val _ = cache := insert (!cache, a, r)
       in r end
  in
     g
  end
```

Not thread safe ⚠

$g : \alpha \to \beta$

$f : (\alpha \to \beta) \to \alpha \to \beta$

$memoise : ((\alpha \to \beta) \to \alpha \to \beta) \to (\alpha \to \beta)$

Ex. memoised fibb

```
fun f g n =
  if n ≤ 1 then 1 else
  g(n-1) + g(n-2)

val fib' = memoiser.memoise f
```

Lec 24 Meldable Priority Queues

Meld operation

meld : $Q \times Q \to Q$ that unions two priority queues

Possible impl: (m ≤ n

	insert	delMin	fromSeq	meld	
- balanced tree	$\lg n$	$\lg n$	$n \lg n$	$m \lg(\frac{n}{m}+1)$	impl dependent
- binary heap	$\lg n$	$\lg n$	n	$n+m$ or $m \lg n$	
- leftist heap	$\lg n$	$\lg n$	n	$\lg(m+n)$	

operations based on meld

```
datatype PQ = Empty | Node (k × PQ × PQ)
singleton x = Node (x, Empty, Empty)
insert (Q, x) = meld (Q, singleton x)
delMin Q = case Q of
             Empty ⇒ (Q, None)
           | Node (k, L, R) ⇒ (meld (L, R), Some k)
fromSeq S = reduce meld Empty ⟨singleton x : x ∈ S⟩
```

cost analysis assuming meld is $O(\lg(m+n))$

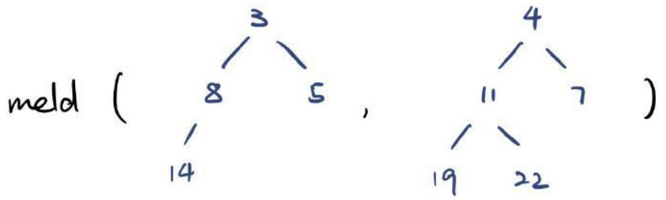
insert $\lg(n+1) = \lg n$

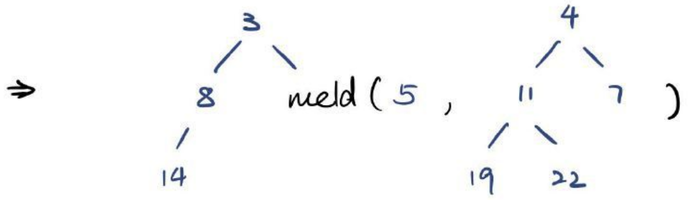
delMin $\lg n$

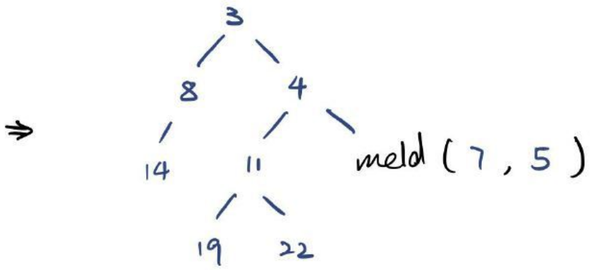
fromSeq
$$W(n) = 2W(\tfrac{n}{2}) + O(\lg n) \in O(2^{\lg n}) = O(n)$$
$$S(n) = S(\tfrac{n}{2}) + O(\lg n) \in O(\lg^2 n)$$

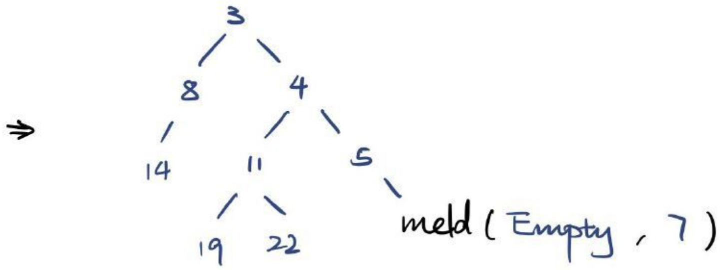
Bad meld (correct but out of bound)

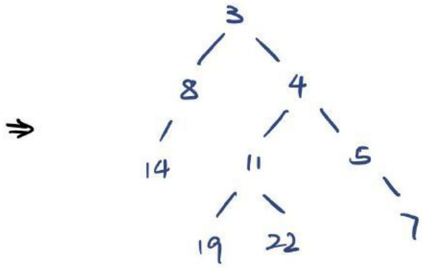
```
meld (A, B) = case (A, B) of
  (_, Empty) ⇒ A
 | (Empty, _) ⇒ B
 | (Node (kA, LA, RA), Node (kB, LB, RB)) ⇒
   if kA < kB then
     Node (kA, LA, meld (RA, B))
   else
     Node (kB, LB, meld (RB, A))
```







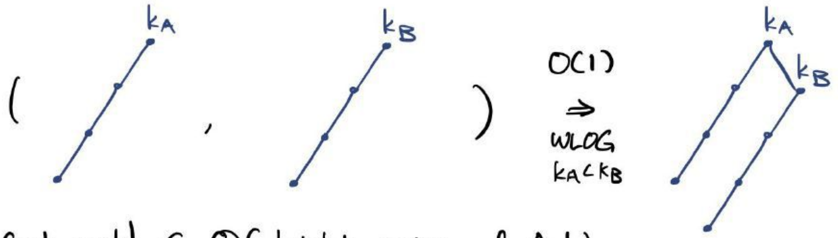




Cost analysis

Observe we only recurse down right subtrees (right spine)

So if right spine is short, we're efficient meld



Fact Cost meld $\in O(|\text{right spine of } A|)$

Leftist queue

Def rank Q := # nodes in right spine

Def leftist property:

$\forall \text{Node}(\_, L, R) \in PQ$, rank $R \leq$ rank L

Impl

```
datatype PQ = Empty | Node ( int x k x PQ x PQ )
rank Q  = case Q of
              Empty  => 0
            | Node (r, _, _, _) => r
node' (k, A, B) = if rank B < rank A then
                      Node ( rank B + 1 , k , A , B )
                  else
                      Node ( rank A + 1, k , B , A )
```

↑ maintains leftist property

Proof

Let $m(r)$ be min size of any leftist heap of rank r.

Claim : $m(r) = 2^r - 1$.

BC $r = 0 \Rightarrow m(0) = 0 = 2^0 - 1$ ✓

IC $m(r) = \overset{\text{root}}{1} + \overset{\text{left, smallest case}}{m(r-1)} + \overset{\text{min of right}}{m(r-1)}$

$= 1 + 2(2^{r-1} - 1)$

$= 2^r - 1$

So size is exponential to rank

Coro rank $Q \leq \lg(|Q| + 1)$

proof is that $|Q| \geq 2^{\text{rank } Q} - 1$

$|Q| + 1 \geq 2^{\text{rank} Q}$

$\lg(|Q| + 1) \geq \text{rank } Q$

Lec 25 Concurrent Data Structure & Work Stealing Scheduling

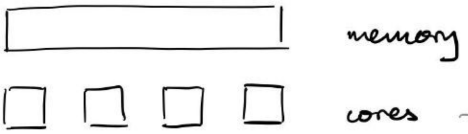
Key ideas
- Lock-free data structure
- Linearisation
- Compare and swap (CAS)
- Concurrent deque
- Randomised stealing

Recall greedy scheduling $T = \frac{W}{P} + S$

But in real world we need to find work to schedule

Working with async. parallel processors

Model



can delay
can get unscheduled
can have different clock rate
⋮

Assume arbitrary interleaving

```
P1   r1 ← mem[a]          P2   r2 ← mem[a]
     r1 = r1+1                 r2 = r2+1
     mem[a] ← r1               mem[a] ← r2
```

race condition possible
mem[a] can end up
+1 or +2

Lock-free data structure

Def Lock free data structure
- Supports certain operations
- Shared across processes
- At least one process making process
  (puts "lock-free lock" i.e. some lamda around critical code)

Linearisability

Operations: load, increment push, pop

→ They can appear interleaved but correctness captured sequentially

Compare and swap

On x86: CMPXCHG

Analogous to:

```
CAS : α ref → (α × α) → bool
CAS r (old, new) =
  let
    a = !r
  in
    if a = old then (r := new ; true)
    else false
  end
```

Note this code is not safe. Processor implements this on hardware level as instruction

Linearisable increment ← no lock involved

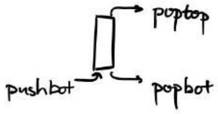
```
Inc (r : int ref) =
  let
    a = !r
  in
    if CAS r (a, a+1) then ()
    else Inc r
  end
```

Work stealing scheduler (randomised)

How to do $f \parallel g$?

→ Forking puts job into shared data structure
Idle threads find the job and do it

Each processor keeps a deque DQ
lock-free, linearisable



- when encountering $f \parallel d$ on processor p

```
DQ_p . pushbot(g)
run f
wait for result of g
```

- If processor p done or while waiting

```
case $DQ_p$.popbot() of
  Some f => run f
  None => repeat (randomly steal from top of another processor's DQ)
```

<u>Analysis</u> — why this works well

1. Most of the time stealing own work
   since we prioritise push and pop from bottom ⇒ good locality

2. Minimise sequentialisation of deque operations, low contention
   since most of the time DQs are long so 1 happens and we don't sequentialise
   also randomness helps spread out contentions

<u>Thm</u> number of steals to attempt is in $O(PS)$
time is in $O(\frac{W}{P} + S)$

P = num processors
S = span

Lec 26 Memoisation with parallelism

Things that can show up in 15-418, 15-312, 15-410

Sequential impl

```
fun memoise f =
  let
    val cache = ref Table.empty
    fun g a = case △Table.find !cache a of
                 SOME r => r
               ▷| NONE => let
                    val r = f g a ]
                    val _ = cache := Table.insert cache (a,r)
                 in◁                □
                    r
                 end
  in
     g
  end
```

← multi threads can call f if they have same a and a not in table

↑ two threads can insert at same time

Prob we often do ≥ 2 recursive calls and want to do them in parallel, but this impl not safe for parallelism

Idea suspend execution, make sure to not race compute

- at △, insert busy marker, at □, update actual result so lookup result can be busy, some, none.
- at ▷, handle busy case by
  - busy wait
  - sleep wait (OS could schedule some other work ?)
  - suspend job (SML built-in, but not Python / C++, etc.) give continuation / handle to another thread
    SML: callcc to suspend
    throw to wake up

  ↳ put self in some queue (could just be set)
  try wake things up at ◁

Impl initialise empty queue Q
states for table entry   state = wait of Q | full of β

```
fun g a = case Table.find !cache a of
  NONE => insert (a, wait (empty queue)) to table ;
          r = f g a ;
```

```
            get queue Q ;
            insert (a, full r) ;
            wake up everything in Q  ] ← O(1) span with some impl
                                         that supports parallel map
      | SOME (wait Q)  ⇒ suspend self in Q
      | SOME (full r)  ⇒ r
```

Also, make sure table and queue are linearisable concurrent data struct needed.

Concurrent table

```
insert :  ctable → (α × β) → β option
            T        (a , b)
          if a not in T, add (a, b) and return NONE
          if (a, b') in T, return SOME b'   ← can retry with update
                                               in this case

update :  ctable → (α, β) → ()

let
  cache = Table.empty
  fun g a =
    let
      Q = Queue.empty
    in
      case (CTable.insert cache (a, wait Q)) of
        SOME (full r) ⇒ r
      | SOME (wait Q) ⇒
          suspend self onto Q ;
          when wake up, find result and return
      | NONE ⇒
          ⋮ (as before)
```