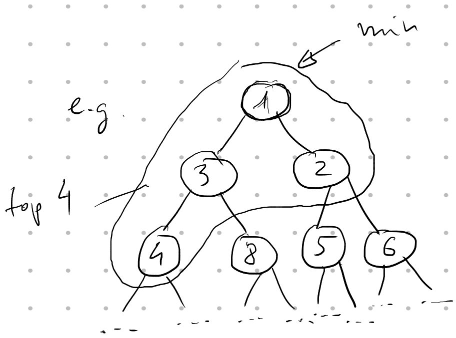
Soft heap applications

Structured selection

Historical note: optimal algorithms were given by [Fredrickson] [Frederickson-Johnson] these were very complicated, Soft heap makes it easier!

binary heap



size: n (possibly infinite)

keys in min-heap order

task: find k^{th} smallest key

(or find k smallest keys)

$(k \leq n)$

Note: there can be missing children in heap, just replace them by ∞

Problem: selection in a heap.

Very naive solution: extract-min k times

time: $O(k \cdot \log n)$

naive solution:

depth k

size $2^k - 1$

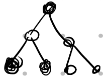
time: $O(k^2)$

only need to look at top k levels

Wait $O(k)$

General selection $x_1, \dots, x_n$, no info about ordering (selection $\Theta(n)$ time)

our problem: selection in heaps



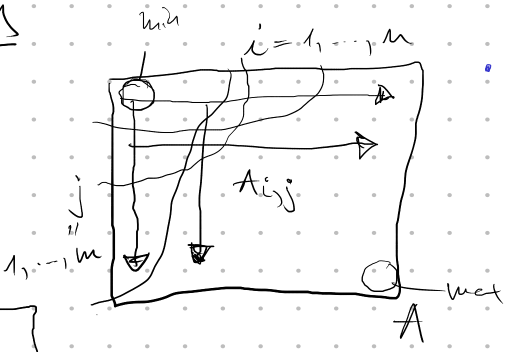
(partial order) "semisorted" how to select efficiently?

Sorted input $x_1 < x_2 < \dots < x_n$ (selection trivial)

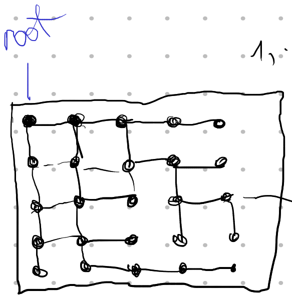
Example application

selection from

① sorted matrix



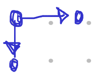
- every row, every column is sorted
- which $A_{i,j}$ is the k^{th} smallest



view as heap

O(k) instead of O(nm)

(some children missing, we can decide which edges to add



)

② Selection from X + Y

Given $X = \{x_1, \dots, x_n\}$

$Y = \{y_1, \dots, y_m\}$

$A_{i,j} = x_i + y_j \quad \forall i,j$

Task: find k^{th} smallest sum in $X+Y$

$X+Y = \left\{A_{ij} \;\middle|\; \begin{matrix} i \in [n] \\ j \in [m] \end{matrix}\right\}$

$\underline{n \cdot m}$ sums

If X, Y sorted

$x_1 < x_2 < \dots < x_n$

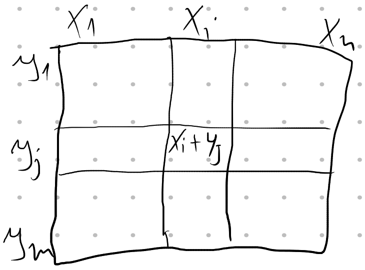
$y_1 < y_2 < \dots < y_m$

↓

use approach ①

↓

O(k) time



If X, Y not sorted, first sort in time $O(n\log n + m\log m)$

→ Sorted Matrix

(can solve more efficiently even if X, Y are not sorted.)

See exercise

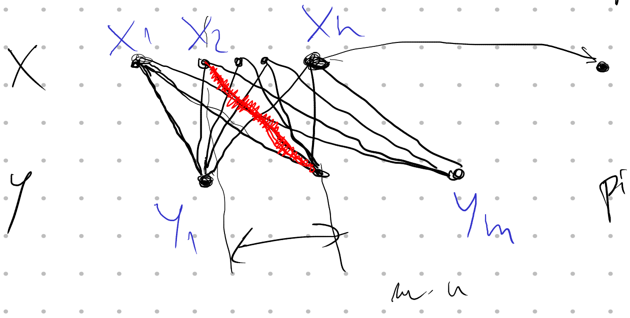
Example application.

$X = \{x_1, \dots, x_n\}$ measurements.

$Y = \{y_1, \dots, y_m\}$

Compare

Result = single number



Hodges-Lehmann estimator

pick line with median slope

pick median from $Y - X$

$((-X) + Y)$

"Robust estimator"

③ Given $x_1 < x_2 < \dots < x_n$

Task: Find k^{th} smallest sum on a subset of $\{x_1, \dots, x_n\}$

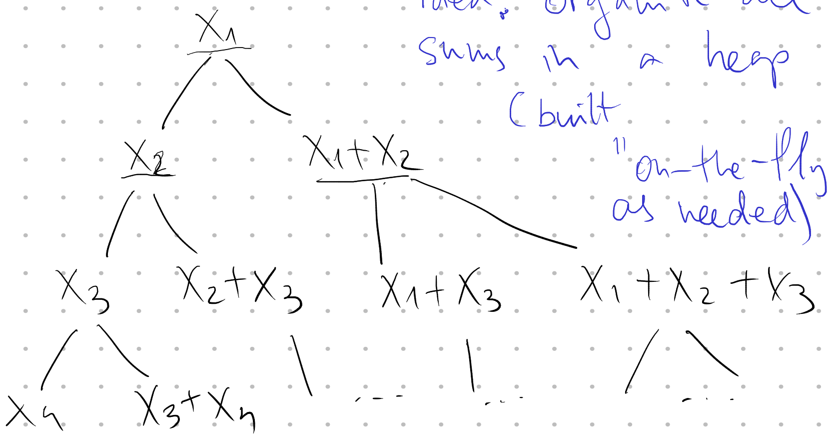
$\emptyset, \; x_1, \; x_2, \; \overset{x_3}{x_1 + x_2} \, ?, \; \dots, \; \overset{100^{th}}{x_1 + x_3 + x_5 + x_6} \, ??$

1st 2nd 3rd 4th

$x_{99} + x_{100}$?? ...

possible subsets 2^n

Want time $O(k)$

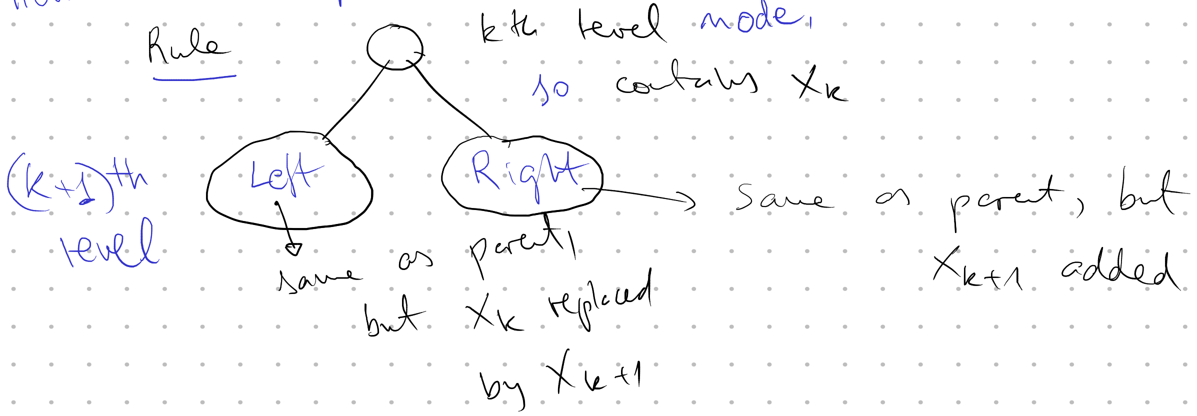


Idea: organize all sums in a heap (built "on-the-fly" as needed)

- top k levels: all sums with elements in $\{x_1, \dots, x_k\}$
- in kth level: all sums containing x_k with elements in $\{x_1, \dots, x_k\}$

How to build heap?

Rule



Obs all possible sums appear in heap.

Induction.

(consider an arbitrary sum S over a subset of $\{x_1, \dots, x_n\}$ and show that it appears in the heap)

x_k: largest element of sum

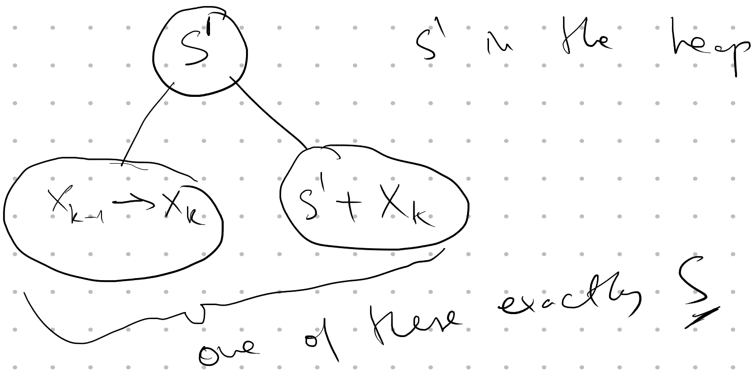
$$S = x_{i_1} + x_{i_2} + \dots + x_{i_j} + \underline{x_k}$$

$$i_1 < i_2 < \dots < i_j < k$$

Claim: S appears in heap at level k.

$$S' = x_{i_1} + x_{i_2} + \dots + x_{i_j} + x_{k-1}, \quad \text{if } i_j < k-1$$

$$= x_{i_1} + x_{i_2} + \dots + x_{i_j}, \quad \text{if } i_j = k-1$$



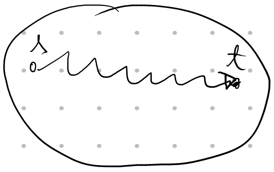
S' in the heap at level $k-1$ (inductive claim)

☒

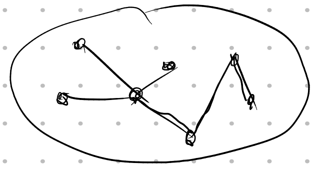
Selecting kth sum → using selection in a heap, will take $O(k)$ time

④ Selection in heaps can be used for structured selection problems, e.g. in graphs.

- shortest path from s,to t.

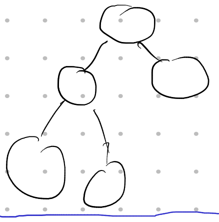


- MST



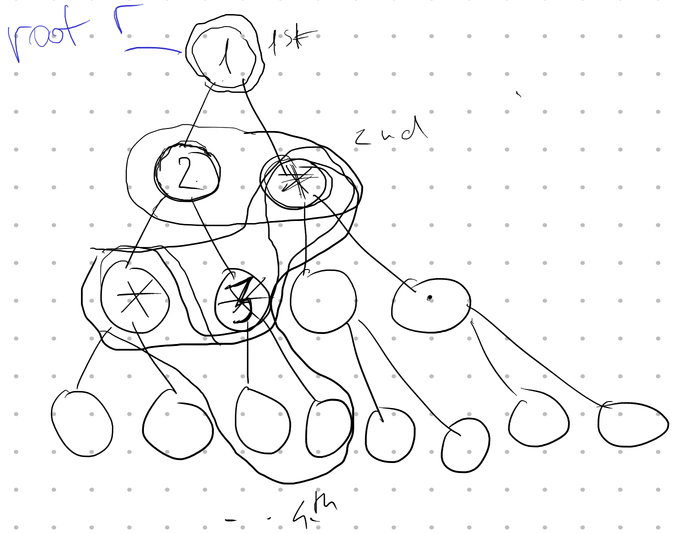
- k^{th} shortest path $s \to t$
- top k shortest path
- top k MST

- Can be reduced to selection in heaps



Algorithms for Selection from heap

First approach: $O(k \cdot \log k)$ time algorithm



Idea: maintain a set of candidate k^{th} elements.

When pop minimum x, add to set two children of x.

Use a heap* data structure to maintain candidate set.

* e.g. skew heap or binary heap

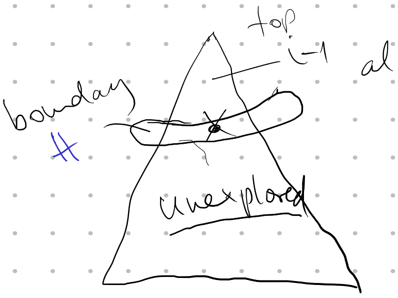
```
H := make-heap()
insert (r, H)
for i = 1,k
    x = extract_min (H)
    print (x)
    insert (x.left, H)
    insert (x.right, H)
```

— $O(\log k)$

Correctness

by induction

i^{th} print outputs i^{th} smallest element



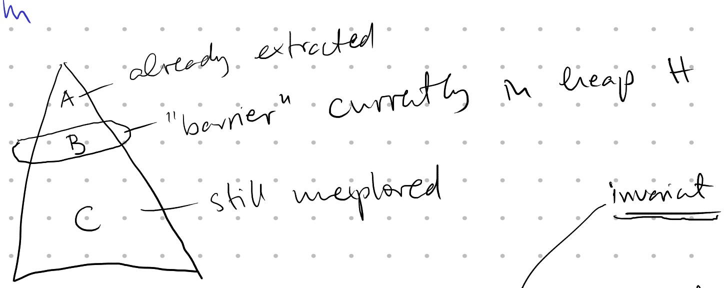
note: if a child missing, add a dummy ∞ node

running time

- in each iteration heap size increases by 1.
- At most k+1 elements in heap

$O(k \log k)$

Review of $O(k \log k)$ time algorithm



<u>invariant</u>: all <u>A</u> is smaller than all in <u>B, C</u>

$|A| = k$, done

return A

<u>induction</u>

- true when $A = \varphi$
- extract_min$(H) \to X$

$X.key$ min in $B \cup C$

(everyone in C has an ancestor in B)

moving X to A maintains invariant

Observation:

Algorithm returns top k elements <u>in increasing order</u>

↓

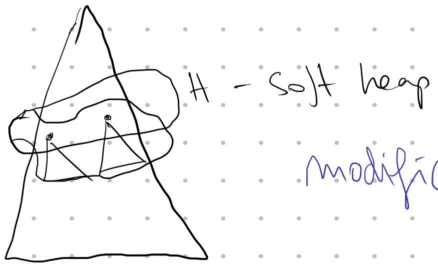
we sorted top k elements

↓

as long as we sort, we cannot do better than $\Theta(k \log k)$

Second approach: $O(k)$ time algorithm

<u>Idea</u>: Same as previous algorithm, but use soft-heap for maintaining H.



$\lg \frac{1}{\varepsilon} \in O(1)$, so running time $O(k)$

modification: add children of X to H, when X is <u>extracted</u> or when X is <u>corrupted</u>.

What we expect from soft heap?

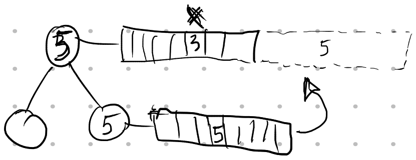
make-heap $\to \varepsilon$ $O(1)$

insert, extract-min $\}$ $O(\lg 1/\varepsilon)$

insert → no corruption

extract-min → return $(x, \underline{C})$ — list of corrupted items

→ Want soft heap to tell us whenever it corrupts some element.



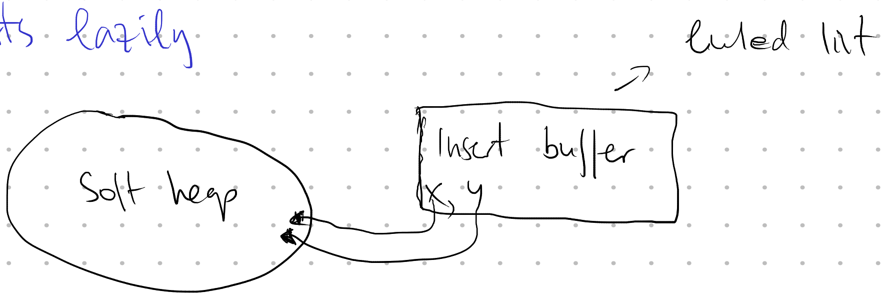
recall: corruption happens during sift-up

for each list move, <u>one</u> item corrupted, charge output cost to siftup cost.

→ Modify soft heap design, so that corruption happens only during extract-min

→ insert should not cause corruption

idea: do inserts lazily

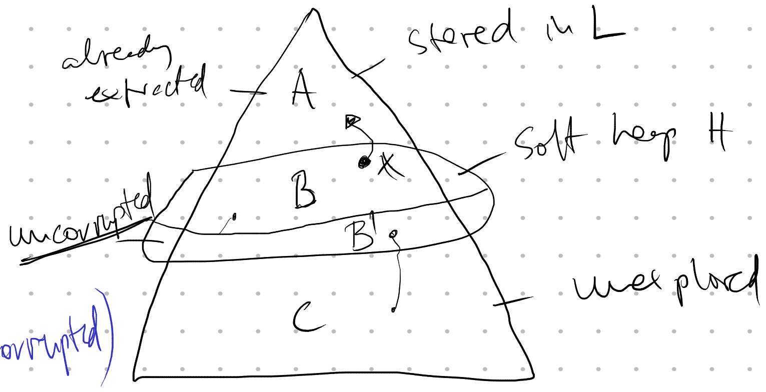


insert(x) → items put into insert buffer
insert(y)   no actual restructuring

| extract-min → everything from buffer inserted, then do extract-min (insert buffer emptied)

Amortized analysis is unchanged

```
H ← make_heap(1/4)  - - - → soft-heap w. error-param ε = 1/4
L ← make-list()
insert(r, H)
for i = 1,k
    (x, K) ← extract-min(H)
    insert(x, L)
    insert(x.left, H)
    insert(x.right, H)
    for all c ∈ K  - - - (newly corrupted items)
        insert(c.left, H)
        insert(c.right, H)
    // top k elements are in (L ∪ H) → size O(k)
    Select top k from L ∪ H
                        → from 9k items
```



Obs. all in A have smaller key than all in C.

if |A| = k
⇒ top k elements cannot be in C
⇒ top k can only be in A ∪ B (L ∪ H)

finish the job with a standard selection step

Proof of obs.: x has smallest current key in B

x.key ≤ x.current

x.key smaller than all current keys in B

x.key ≤ y.key for all y ∈ B' - - - (uncorrupted)

every element in C has an ancestor in B'

every key in C > x.key

Show that $|A \cup B| \in O(k)$

Let I denote # insertions into H

Let K denote # corruptions

(# corrupted items in heap $\le \varepsilon \cdot I$)

$$I \le 2k + 2K$$

$$K \le \underbrace{\varepsilon \cdot I}_{|B \setminus B'|} + k$$

everything extracted already

(Solve for I)

Observe that $I = |A \cup B|$ (everything in the heap and everything extracted must have been inserted at some point)

$$K \le \varepsilon \cdot (2k + 2K) + k$$

$$\Rightarrow K(1-2\varepsilon) \le k(1+2\varepsilon)$$

$$\Rightarrow K \le k \frac{1+2\varepsilon}{1-2\varepsilon}$$

$$\Rightarrow I \le \underline{2k} + \underline{2k} \underbrace{\frac{1+2\varepsilon}{1-2\varepsilon}}_{\varepsilon = \frac{1}{4}} \in O(k)$$

$$\frac{\frac{3}{2}}{\frac{1}{2}} = 3$$

$$I \le 8k$$

$$\Rightarrow |A \cup B| \le 8k$$

last step:

Select top k from $\le 8k$ items (takes $O(k)$ time.)

Overall $O(k)$ time to select top-k in heap.

Observation

We assumed that heap is binary. We could handle larger constant degree too.

- First approach extends easily to d-ary heap.
- Second approach also extends but calculation changes

now $I \le \underline{d} \cdot k + \underline{d} \cdot K$ (we may need to set ε smaller, e.g. $\varepsilon = \frac{1}{2d}$)