Soft heap applications

Structured selection

binary heap

size: n (possibly infinite)
keys in min-heap order

task: find k\textsuperscript{th} smallest key
(or find k smallest keys)

(k ≤ n)

Problem: selection in a heap.

Very naive solution: extract-min k times \(O(k \cdot \log n)\)

Naïve solution:

only need to look at top k levels

Depth \(k\) = size \(2^{k-1}\)

Wait \(O(k)\)

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

Our problem: Selection in heaps

"Semi sorted" how to select efficiently?

Sorted input \(x_1 < x_2 < \ldots < x_n\) (Selection trivial)

Example application

Selection from

1) Sorted matrix

\(O(mn)\) \(\text{root}\) view as heap

\(\text{adj\matrix}\)

\(\text{every row, every column is sorted}\)

\(\text{which } x_{i,j} \text{ is the } k\text{th smallest}\)

\(\text{some children missing, we can decide which edge to add}\)
Given \( X = \{x_1, \ldots, x_n\} \) and \( Y = \{y_1, \ldots, y_m\} \),

\[ X + Y = \{a_{ij} \mid i \in [n], j \in [m]\} \]

Task: Find \( k \)th smallest sum in \( X + Y \) without sorting.

If \( X, Y \) are sorted:

Use approach 1:

- \( X_Y \) sorted

\[ x_1 < x_2 < \ldots < x_n \]
\[ y_1 < y_2 < \ldots < y_m \]

\( O(k) \) time

(See exercise)

Example application:

\( X = \{x_1, \ldots, x_n\} \) measurements.
\( Y = \{y_1, \ldots, y_m\} \) measurements.

Compare results:

Result: Angle number

Hodge-Lehmoun estimator

Pick line with median slope

Pick median from \( Y - X \)

"Robust estimator"

3. Given \( x_1 \leq x_2 \leq \ldots \leq x_n \)

Task: Find \( k \)th smallest sum on a subset of \( \{x_1, \ldots, x_n\} \)

\[ \phi_1(x_1, x_2, x_3, \ldots, x_n) \]
\[ \phi_2(x_1 + x_2, x_3, \ldots, x_n) \]
\[ \phi_3(x_1 + x_2 + x_3, \ldots, x_n) \]
\[ \vdots \]
\[ \phi_k(x_1 + x_2 + \ldots + x_k) \]

\( \leq \) possible subsets: \( 2^n \)

Wait time: \( O(k) \)
Idea: organize all sums in a heap
(built "on-the-fly" as needed)

- Top k levels:
  all sums with elements in 
  \( \{x_1, \ldots, x_k\} \)
- In k'th level:
  all sums containing \( x_k \)
  with elements in \( \{x_1, \ldots, x_k\} \)

how to build heap?

\((k+1)\)th level

Left

Same as (left but \( x_k \) replaced by \( x_{k+1} \))

Right

Same as parent, but \( x_{k+1} \) added

Obs: all possible sums appear in heap.

Induction.

- \( x_k \): largest element of sum

\[ S = x_{i_1} + x_{i_2} + \ldots + x_{i_j} + x_k \]

\[ 1 < i_2 < \ldots < i_j < k \]

Claim: \( S \) appears in heap at level \( k \).

\[ S' = x_{i_1} + x_{i_2} + \ldots + x_{i_j} + x_{k-1} \] if \( i_j < k-1 \)

\[ = x_{i_1} + x_{i_2} + \ldots + x_{i_j} \] if \( i_j = k-1 \)

\[ S' \in \text{the heap at level } k-1 \]

(Inductive claim)

Select k'th sum \( \Rightarrow \) way selection in a heap, will take \( O(k) \) time
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 $s \rightarrow t$
- 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 $x$, odd to set two children of $x$.

Use a heap* data structure to maintain candidate set.

- $H := \text{make-heap}()$
- $x := \text{extract-min}(H)$
- Print $x$.
- For $i = 1 \rightarrow k$
  - $H := \text{insert}(i, H)$
  - $x := \text{extract-min}(H)$
  - Print $x$
  - $H := \text{insert}(x, \text{left}(x), H)$ $\rightarrow O(\log k)$
  - $H := \text{insert}(x, \text{right}(x), H)$ $\rightarrow O(\log k)$

**Runtime**
- At each iteration, heap size decreases by 1.
- At most $k+1$ elements in heap $\rightarrow O(k \cdot \log k)$

**Note**: If a child is missing, add a dummy (✓).
Review of O(k\log k) time algorithm

Observation:
Algorithm returns top k elements in increasing order

We sorted top k elements

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

Second approach: O(k) time algorithm

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

\(\log^2 k \leq O(1)\), so running time \(O(k)\)

Modification: add children of \(x\) to \(H\) when \(x\) is extracted or when \(x\) is corrupted.

What we expect from soft heap?

\[ \begin{align*}
\text{make-heap} & \to O(1) \\
\text{insert} & \to O(\log 1/e) \\
\text{extract-min} & \to O(1/e) \\
\text{no corruption} & \text{ return } (x, C) \\
\text{list of corrupted items} & \end{align*} \]

Wait soft heap to tell us whenever it corrupts some element.

Recall: Corruption happens during sift-up

For each list move, one item corrupted, charge output cost to siftup cost.
Modify soft heap design, so that corruption happens only upon \textit{extract-min}.

- Insert should not cause corruption. Let's try:

  - An amortized analysis is unclear.

```latex
\begin{align*}
\text{insert buffer} & \quad \text{key}
\end{align*}
```

\(\text{insert}(x)\) \quad \text{insert buffer emptied}

\(\text{extract-min}\) everyday from buffer inserted

\(\text{then do} \quad \text{extract-min}\)

\(\text{insert buffer emptied}\)

<table>
<thead>
<tr>
<th>insert</th>
<th>(H \leftarrow \text{mole-heap}(\frac{1}{4}))</th>
<th>(L \leftarrow \text{mole-insert}(\frac{1}{4}))</th>
<th>insert ((x, H))</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i = \frac{1}{4}, \ldots, k)</td>
<td>((x, K) \leftarrow \text{extract-min}(H))</td>
<td>\text{insert} ((x, L))</td>
<td>\text{insert} ((x, \text{left}, H))</td>
</tr>
<tr>
<td>\text{for all} (c \in K)</td>
<td>\text{insert} ((c, \text{left}, H))</td>
<td>\text{insert} ((c, \text{right}, H))</td>
<td></td>
</tr>
</tbody>
</table>

\text{Select top} \(k\) \text{ from } L \cup H

\text{Proof:}

\(x\) has smallest current key in \(B\)

\(x.\text{key} \leq x.\text{current}\)

\(x.\text{key} \text{ smaller than all current keys in } B\)

\(x.\text{key} \leq y.\text{key} \quad \text{for all } y \in B\)

\(\forall y \in B, \quad \text{every element in } C \text{ has an ancestor in } B\)

\(\Rightarrow \text{top} \ k \text{ elements can only be in } A \cup B \quad (L \cup H)\)

\text{finish the selection step with a standard selection step.}
Show that $|A \cup B| \leq O(k)$

Let $I$ denote # insertion into $H$

Let $K$ denote # corrupted items ($\leq 3 \cdot I$)

$I \leq 2k + 2K$

$K \leq 3 \cdot (2k + 2K) + k$

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

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

$\Rightarrow I \leq 2k + \frac{1 + 2\varepsilon}{1 - 2\varepsilon} \cdot k \leq O(k)$

$\frac{1 - 2\varepsilon}{1 + 2\varepsilon} \Rightarrow I \leq 8k$

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

Last step:

Select top $k$ from $\leq 8k$ items (total $O(k)$ time)

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

Observation:

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

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

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