Priority Queue:

- Insert
- Delete Min
- Change Key
- Merge

Abstract Data Type:

Data class defined by behaviour.

Java - Interface.

C++ - virtual classes.

We only want the methods to be available.

Implementation is not enforced.

Heaps are an implementation:

Types of Heaps:

- Binary Heaps
  - $O(\log n)$
  - $O(\log n)$
  - $O(\log n)$
  - $O(n)$

- Binomial Heaps
  - $O(1)^*$
  - $O(\log n)$
  - $O(\log n)$
  - $O(\log n)$

- Fibonacci Heaps
  - $O(1)^*$
  - $O(\log n)$
  - $O(1)^*$
  - $O(1)^*$

[Amortized complexity]

*Not the usual time complexity.

This is the average over a lot of operations.
Heap is an almost complete binary tree with the heap invariant:

\( \forall x \in \text{Heap}, \forall y \in x.\text{subtree} \),

\( y.\text{key} \geq x.\text{key} \).

\[ \text{ex.} \]

\[ \text{almost-complete: all layers except the last have } 2^k \text{ nodes (at depth } k). \]

Implementing using array:

\[ 1, 5, 2, 7, 6, 3, 4, 10, 8, 7, 8, 5, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 \]
for $i$: children $\rightarrow 2i+1$, $2i+2$.
parent $\rightarrow \left\lfloor \frac{i-1}{2} \right\rfloor$

Because it is almost complete
height $= \Theta \left( \log n \right)$.

Insert:
1. insert 2.
2. insert 1
move Up (pos).

while pos ≠ 0
    parent = ⌈pos-1/2⌉
    if parent.key < pos.key
        break.
    swap (parent.key, pos.key).
    pos = parent.

Time taken to insert = O(lgn).

Delete Min.

```
1
\rightarrow \rightarrow
5 6 2 2
F F F 5 3
10 8 7 8 5 3
```

swap

```
\rightarrow \rightarrow
5 6 2 2
\rightarrow \rightarrow
1 6 2 2
10 8 7 8 5 3
```

pop

```
\rightarrow \rightarrow
4
\rightarrow \rightarrow
5
\rightarrow \rightarrow
1 6 2 2
10 8 7 8 5 3
```
move down (pos)
while \( \exists \) child of pos.
swap (pos, smaller child of pos).
\( \text{pos} = \text{smaller child of pos} \).

Time to delete min = \( O(\log n) \).

To change key. (assuming given the position)
use the appropriate move up/down.

Creating a heap in \( O(n) \).

Heapify: instead of doing \( n \)-inserts. \( \mathcal{O}(n \log n) \).
move down layer by layer.

Time taken to move layer at depth $j$

- # nodes = $2^j$
- Time for each node = $h-j$
- Time for layer = $2^j \cdot (h-j)$

Total $\sum_{j=0}^{h-1} 2^j (h-j)$.
\[
\sum_{j=1}^{2^n} \frac{1}{j} = \frac{\sum_{j=1}^{\infty} \frac{1}{j}}{2^n} \leq \frac{\sum_{j=1}^{\infty} \frac{1}{j}}{2}\]

Fact: \( \sum_{j=1}^{\infty} \frac{1}{j} = 2 \)

\[
\begin{align*}
\frac{1}{2} & \\
\frac{1}{4} & \\
\frac{1}{8} & \\
\frac{1}{8} & \\
\frac{1}{8} & \\
\vdots & \\
1 & \frac{1}{2} \quad \frac{1}{4} \\
\end{align*}
\]

\[
\geq 2.
\]

\[
\text{time to heapify} = \mathcal{O}(n).
\]
Application

Heap sort. (→ Inplace.)

Advantages.
- Cannot do faster, other than by a constant factor.
- Don't need to make a copy.

Build the heap in $O(n)$.
& when you do deleteMin. & need to pop the element from back of the array. [Do not remove it.] Just continue as if it is not there.

Time to do $n$-deleteMin

$= O(n \log n)$.

$\Rightarrow$ Time to sort $= O(n \log n)$. 
Dynamic Programming

A better word is memorization (memorization) in 1950.

Original DP - coined by Richard Bellman to confuse politicians.

[More etymology = waste of time on E.E.]

Idea is to keep track of problems that are already solved.

Change Making Problem

Given coins of n values, \( a_1, a_2, \ldots, a_n \) and a final amount \( P \), find the minimum number of coins needed to add up to \( P \).

e.g., U.S. system.

\[
\begin{align*}
1, 2, 5, 10, 20, 50, 100
\end{align*}
\]

\[
\begin{align*}
70 & \rightarrow 50 + 20. \\
80 & \rightarrow 50 + 20 + 10. \\
90 & \rightarrow 50 + 20 + 20.
\end{align*}
\]
Naive strategy
Pick the largest denomination available less than current amount & continue recursively.

90 → pick 50.
40 → pick 20
20 → pick 20.

This seems to work!

Counter example

1, 5, 8.

10 → 8 + 1 + 1. picking using naive strategy.

Better → 5 + 5.

So even when using recursion cannot
discount the fact of using any denomination.

Rec Change (P)
{a, a2, ..., an}

return 1 + min
n
P > a;

Rec Change (P - a i).
So at each point we potentially make $n$-recursive calls.

If height of tree $= h$, then time taken is at least $\sum_{i=0}^{h} N_i$ (total number of nodes) ($h$ can be $\frac{P}{\alpha N}$).

This is way too much.

Look at the calls that are repeated.

We are recomputing too many things.

So better to store them rather than recompute.
Store a table: \( \text{DP}[1, \ldots, P] \).

\[
\text{DP}[x] := \text{minimum number of coins needed to get } x.
\]

\[
\text{DP}[x] = \min_{i, a_i \leq x} \text{DP}[x-a_i] + 1
\]

E.g., \((1, 5, 8)\).

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
0 & 1 & 2 & 3 & 4 & 1 & 2 & 3 & 1 & 2 & 2
\end{array}
\]

Important: put \( \text{DP}[0] = 0 \).

Time taken = # of blanks to fill \( x \times \) time for each blank.

= \((P+1) \times n\).

= \(O(nP)\).
Proof.

\[ DP[x] \geq 0 \quad \forall \ x. \]
\[ \& \quad DP[x] = 0 \quad (\Rightarrow \quad x = 0). \]

1) \[ DP[x] \leq DP[x-ai] + 1 \quad \forall \ i \in \{1, \ldots, n\}. \]

As this is one way to get \( x \) & there may be more

2) to make \( x \) we need at least one coin of some denomination, say \( k \), & we can get \( x-ak \) either from \( x \) remove \( ak \) or otherwise

\[ \Rightarrow \quad DP[x-ak] \leq DP[x] - 1. \]

1) + 2) together

\[ \Rightarrow \quad DP[x] = \min_{i, ai \leq x} \{ DP[x-ai] + 1 \}. \]
0-1 Knapsack Problem

given $n$ items with weights $w_i$ and values $v_i$, and a knapsack with max weight $v_i$, ... , $v_n$ and capacity $W$.

find the maximum value of items you can fit in the knapsack.

E.g. $2 5 7 10 11 13 22$
$3 10 8 22 23 28$

$\rightarrow 10, 11$
$22, 23 = 45.$

Rec Knapsack. ($\{w_1, \ldots, w_n\}$, $\{v_1, \ldots, v_n\}$, $W$).

$v_1 = 0$

if $w_n \leq W$.

$v_i = \text{Rec. Knapsack} (\{w_1, \ldots, w_{i-1}\}, \{v_1, \ldots, v_{i-1}\}, W - w_n)$.

$v_2 = \text{Rec. Knapsack} (\{w_1, \ldots, w_{i-1}\}, \{v_1, \ldots, v_{i-1}\}, W)$.

Return $\max \{v_1, v_2\}$. 

Identifying recomputations (subproblems).

Inputs are of the form

\[ w_1, \ldots, w_k, C \]
\[ v_1, \ldots, v_k \]

\[ k = 0 \rightarrow n \]
\[ C = 0 \rightarrow W \]

\[ \Rightarrow \text{total inputs} = O(nW). \]

\[ \text{DP}[0 \ldots n][0 \ldots W] \]

\[ \text{DP}[k][C] = \text{max value possible using the first } k \text{ items \& capacity } C. \]

\[ \text{DP}[k][C] = \max \left( \text{DP}[k-1][C], \text{DP}[k-1][C-a_k] + v_k \right) \]

\[ \Rightarrow \text{total time} = O(nW). \]

Pseudo polynomial as it is not poly in length of \( W \rightarrow O(\log W) \).