437 – The Tower of Babylon

Task

- n types of blocks
- unlimited supply of blocks
- Rectangular, can be reoriented
- Base of a top block has a smaller base dimensions than the lower block

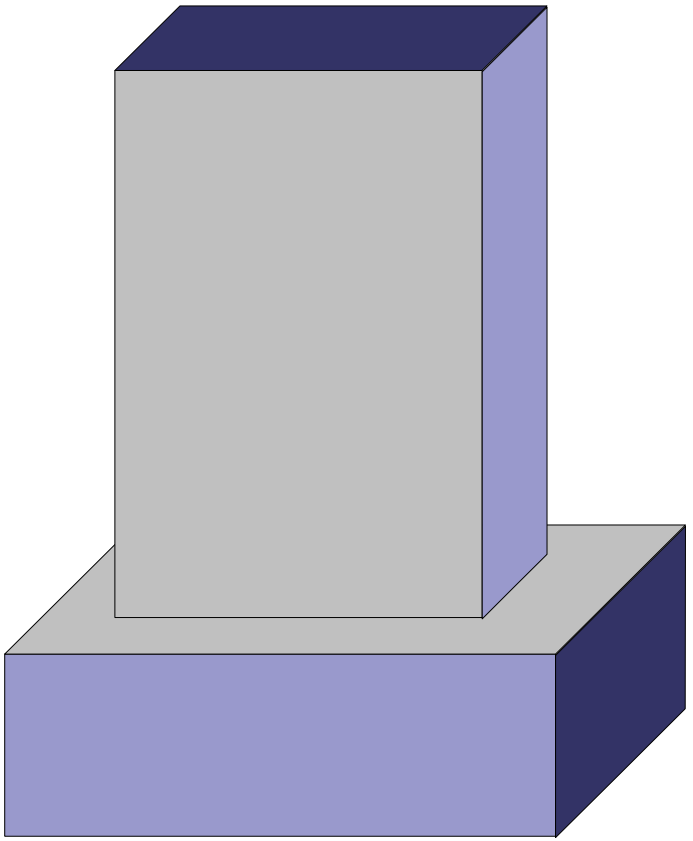
Purpose:

- Construct tallest tower possible

Beispiel

- [10, 20, 30]

  (Breite, Tiefe, Höhe)



Program-Input:

1

10 20 30

0

Beobachtung (1)

- [10, 20, 30]-Block
  - (10, 20, 30), (10, 30, 20)
  - (20, 10, 30), (20, 30, 10)
  - (30, 10, 20), (30, 20, 10)
    - Höchster Turm, Sequenz 1: 30
    - Höchster Turm, Sequenz 2: 40

      (30, 20, 10) → (20, 10, 30) <=>

      (20, 30, 10) → (10, 20, 30) <=>

      Höhe: 30 + 10 = 40
- Redundanz, Blöcke doppelt

Beobachtung (2)

- [10, 20, 30]-Block
  - (10, 20, 30)
  - (10, 30, 10)
  - (20, 30, 10)

- Ergebnis auch hier:

  (10, 20, 30) → (20, 30, 10) = 40

Erkenntnis (1)

- Relevante Blöcke:
  - Quader
    - (a, b, c), (a, c, b), (b, c, a)
      mit a < b < c
  - Quadratsäule
    - (a, a, b), (a, b, a)
      mit a < b
  - Würfel
    - (a, a, a)

Erkenntnis (2)

- Höchster Turm ungleich höchste Anzahl von Blöcken!
  - Beispiel:

    [1, 42, 42], [4, 4, 4], [3, 3, 3]

  - Lösung:

    maximum height = 42

Beispiel

- Eingabe →

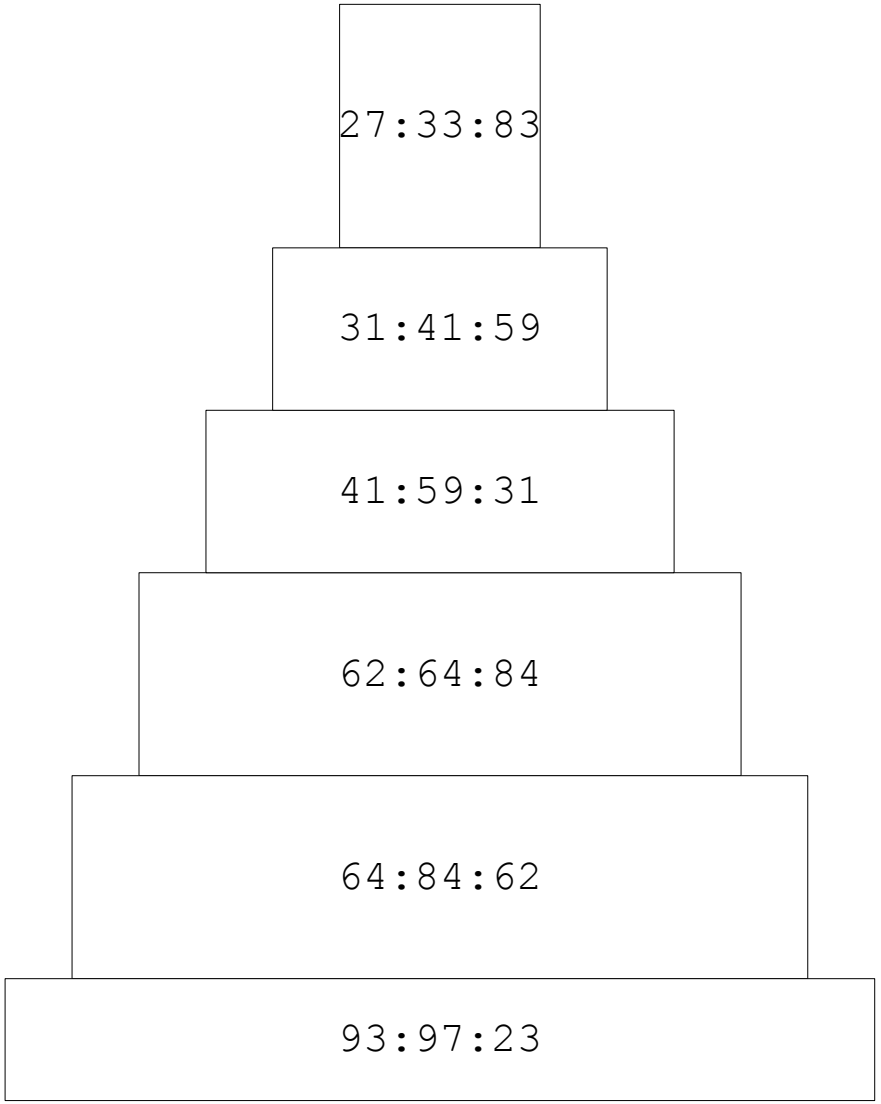
(5)

31	41	59
26	53	58
93	97	23
62	84	62
33	27	83

31	41	59
31	59	41
41	59	31
26	53	58
26	58	26
53	58	26
23	93	97
23	97	93
93	97	23
62	64	84
62	84	64
64	84	62
27	33	83
27	83	33
33	83	27

Lösung - Beispiel

- 6 Blöcke
- Höhe = 342



Lösungsarten

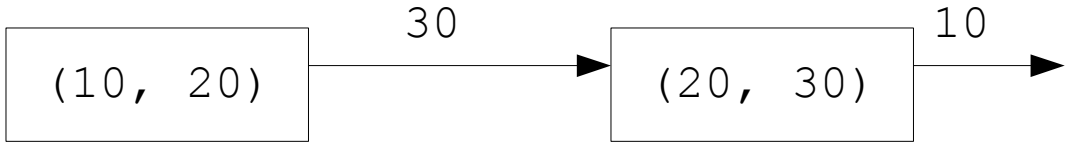
- Graphisch (Java)
- Dynamische Programmierung (C++)

  Longest Increasing Subsequence (LIS)

Graphische Lösung

- Jeder Block ist ein Knoten
- Kanten stellen die Höhe dar
- Längste Strecke im Graph äquivalent zu höchst möglicher Turm

- Beispiel:



Implementierung – class Block

- Implements Compareable<Block>
  - compareTo beachtet nur die Länge und Breite
- Variablen
  - final int: length, width
  - int: high, value
    - Ist ein neuer Block 'gleich' einen vorhandenen, high := ggfs neu setzen
    - value wird bei der Suche nach längsten Pfad benötigt
  - TreeSet<Block> next

Graph (1)

- Jeder Block wird in den Graph(en) einfügt

  (Quader: 3x, Quadratsäule: 2x, Würfel: 1x)
- Redundanz vermeiden, nur relevante Blöcke
- Falls (a, b) schon vorhanden, setze Höhe wie folgt:

  c := c_curr > c_new? c_curr : c_new
- Nach einfügen des letzten Blocks:

  → Längste Strecke berechnen

Graph (2)

- Mehrere Graphen
  - Liste von 'Knotenanfänge' verwalten
  - Neuen Knoten einfügen:
    - Durchlaufe die 'Knotenanfänge'-Liste
      - Kann Knoten dort eingefügt werden?

        (Vorne (→ neuer Knotenanfang), Mitte oder am Ende?
    - Falls Knoten nirgends eingefügt werden konnte → einfügen in die Liste (der 'Knotenanfänge')

Einlesen

```
private void addBlock(int a, int b, int c) {

    if(a == b && b == c) { //cube
        tryToAdd(a, a, a);
    }
    else if(a == b) { //square tube
        tryToAdd(a, a, c);
        tryToAdd(a, c, a);
    }
    else if(b == c) { //square tube
        tryToAdd(a, b, b);
        tryToAdd(b, b, a);
    }
    else { //cuboid
        tryToAdd(a, b, c);
        tryToAdd(a, c, b);
        tryToAdd(b, c, a);
    }
}
```

```
private void tryToAdd(int a, int b, int c) {
    Block block = new Block(a, b, c);

    if(blockSet.contains(block)) {
        Block tmp = blockSet.ceiling(block);
        tmp.high = tmp.high > c? tmp.high :c;
    }
    else {
        blockSet.add(block);
    }
}
```

Graph erstellen (1)

```
private void builtUpGraph() {
    if(blockSet.isEmpty())
        return;

    ArrayList<Block> tail = new ArrayList<Block>();
    tail.add(blockSet.pollFirst());

    for(Block newBlock : blockSet)
    {
        boolean isAdded = false;

        for(Block start : tail) {
            if(runThroughGraph(start, newBlock) && !isAdded) {
                isAdded = true;
            }
        }
        if(!isAdded) {
            tail.add(newBlock);
        }
    }
```

Graph erstellen (2)

```
private static boolean runThroughGraph(Block curr, Block newBlock)
{
    boolean sthFound = false;

    for (Block next : curr.next) {
        if (runThroughGraph(next, newBlock)) {
            sthFound = true;
        }
    }

    if(!sthFound) {
        if (newBlock.width > curr.width && newBlock.lenght > curr.lenght) {
            if (!curr.next.contains(newBlock)) {
                curr.next.add(newBlock);
            }
            return true;
        }
    }

    return sthFound;
}
```

Längsten Pfad finden

```
private void calcBestResult(Block start) {
    if(start.next.isEmpty()) {
        start.val += start.high;
        if(start.val > bestResult) {
            bestResult = start.val;
        }
    }
    else {
        for(Block next : start.next) {
            next.val = start.high + start.val;
            if(next.val > bestResult) {
                bestResult = next.val;
            }
            calcBestResult(next);
        }
    }
}
```

Dynamische Programmierung (1)

- Werte in Matrix speichern
  - Matrix[N <= x <= 3 * N][y = 3]
- Nach Länge und Breite aufsteigend sortieren
- Die Höhe jedes Blocks in Sequenz[x] separat speichern

Dynamische Programmierung (2)

- Aufsteigende Teilfolge suchen
  - In der Sequenz steht das jeweilige beste Resultat (height) für den momentanen Block
- Sequenz durchsuchen,

  höchster Wert := maximum height
- Lösung ausgeben

Einlesen

```
for (i = 0; i < quantity; i++) {
    cin >> temp[0] >> temp[1] >> temp[2];
    sort(temp, temp + DIMENSION);

    blocks[currIndex][0] = temp[0], blocks[currIndex][1] = temp[1], blocks[currIndex][2] = temp[2];
    currIndex++;

    if(!(temp[0] == temp[1] && temp[1] == temp[2])) {
        if(temp[0] == temp[1]) {
            blocks[currIndex][0] = blocks[currIndex][2] = temp[0];
            blocks[currIndex][1] = temp[2];
        }
        else if(temp[1] == temp[2]) {
            blocks[currIndex][0] = blocks[currIndex][1] = temp[1];
            blocks[currIndex][2] = temp[0];
        }
        else {
            blocks[currIndex][0] = temp[0];
            blocks[currIndex][1] = temp[2];
            blocks[currIndex][2] = temp[1];
            currIndex++;
            blocks[currIndex][0] = temp[1];
            blocks[currIndex][1] = temp[2];
            blocks[currIndex][2] = temp[0];
        }
        currIndex++;
    }
}
```

Sortieren und LIS anwenden

- Notwendigen Variablen noch anpassen
  - `length = currIndex`
  - `int sequence[length]`
- Ersten zwei Spalten sortiern
- LIS anwenden
- Ergebnis in sequence finden und ausgeben

Sortierung – Selection Sort

```
for (i = 0; i < length; i++) {
    k = i;
    for (j = i + 1; j < length; j++) {
        if (blocks[k][0] > blocks[j][0] && blocks[k][1] > blocks[j][1]) {
            k = j;
        }
    }

    temp[0] = blocks[k][0], temp[1] = blocks[k][1], temp[2] = blocks[k][2];
    blocks[k][0] = blocks[i][0], blocks[k][1] = blocks[i][1], blocks[k][2] = blocks[i][2];
    blocks[i][0] = temp[0], blocks[i][1] = temp[1], blocks[i][2] = temp[2];

    sequence[i] = blocks[i][2];
}
```

Longest Increasing Subsequence

```
for (i = 0; i < length; i++) {
    for (j = i + 1; j < length; j++) {
        if ((sequence[j] < sequence[i] + blocks[j][2]) && (blocks[j][0] > blocks[i][0] && blocks[j][1] > blocks[i][1])) {
            sequence[j] = sequence[i] + blocks[j][2];
        }
    }
    if (maxValue < sequence[i]) {
        maxValue = sequence[i];
    }
}
```

Run Time - UVa

- Graphische Lösung - Java:

  1.152 sec

- Dynamische Lösung (LIS) - C++:

  0.012 sec