Notes on Minimum Spanning Tree problem

A Priority queue trouble

Usual theoretical and conceptual descriptions of Prim's (Dijkstra's, etc.) algorithm say: "... store nodes in a priority queue". Technically, this is nearly impossible to do.

The graph and the nodes are defined separately and stored elsewhere in the memory.
The node has no reference to its position in the queue.
The programmer does not know where is the node in the queue.

So, how to move a node inside the queue according to the algorithm demands?
Standard solution:
Do not move a node, enqueue a "copy of a node", possibly more times.

When a copy of the node with the smallest value (=highest priority) among all its copies appears at the top of the queue it does its job exactly according to the algorithm prescription.
From that moment on, all other copies of the node which are still in the queue become useless and must be ignored. The easiest way to ignore a copy is to check it when it later appears at the top of the queue: If the node is already closed, ignore the copy, pop it and process the next top of the queue. If the node is still open, process it according to the algorithm.
q.insert(y);
q.insert(z);
qu.insert(w);

// push the nodes
// once more to the queue.

The older copies of nodes will get to the top of the queue later than the new copies (which have higher priority). The older copy gets to the top when the node had been processed and closed earlier. Thus:
If the node at the top of the queue is closed just pop it and do not process it any more.
Example of Prim's algorithm implementation using standard library priority queue

```java
void MST_Prim(Graph g, int start, final int[] dist, int[] pred) {
    // allocate structures
    int currnode = start, currdist, neigh;
    int INF = Integer.MAX_VALUE;
    boolean[] closed = new boolean[g.N];

    PriorityQueue<Integer> pq
        = new PriorityQueue<Integer>(g.N,
        new Comparator<Integer>() {
            @Override
            public int compare(Integer n1, Integer n2) {
                if( dist[n1] < dist[n2] ) return -1;
                if( dist[n1] > dist[n2] ) return 1;
                return 0;
            }
        });

    // initialize structures
    pq.add(start);
    for( int i = 0; i < g.N; i++ ) pred[i] = i;
    Arrays.fill(dist, INF);
    Arrays.fill(closed, false);
    dist[start] = 0;
}
```
Example of Prim's algorithm implementation using standard library priority queue

```java
for( int i = 0; i < g.N; i++ ) {
    // take the closest node and skip the closed ones
    while( closed[currnode = pq.poll()] == true );
    // and expand the closest node
    for( int j = 0; j < g.dg[currnode]; j++ ){
        neigh = g.edge[currnode][j];
        if( !closed[neigh] &&
            ( dist[neigh] > g.w[currnode][j]) ) {
            dist[neigh] = g.w[currnode][j];
            pred[neigh] = currnode;
            pq.add(neigh);
        }
    } // for j
    closed[currnode] = true;
} // for i
} // MST_Prim
```

A very small change produces Dijkstra's algorithm:

```java
if( !closed[neigh] &&
    ( dist[neigh] > g.w[currnode][j] + dist[currnode]) ) {
    dist[neigh] = g.w[currnode][j] + dist[currnode];
```
**Prim and Dijkstra Algorithms compared**

**The only difference:**

```c
void Dijkstra( Graph G, function weight, Node startnode )
    for each u in G.V:  u.dist = INFINITY; u.parent = NIL
    startnode.dist = 0; PriorityQueue Q = G.V
    while !Q.isEmpty()
        u = Extract-Min(Q)
        for each v in G.Adj[u]
            if (v in Q) and v.dist > weight(u,v) + u.dist
                v.parent = u
                v.dist = weight(u,v) + u.dist
```

```c
void MST_Prim( Graph G, function weight, Node startnode )
    for each u in G.V:  u.dist = INFINITY; u.parent = NIL
    startnode.dist = 0; PriorityQueue Q = G.V
    while !Q.isEmpty()
        u = Extract-Min(Q)
        for each v in G.Adj[u]
            if (v in Q) and v.dist > weight(u,v)
                v.parent = u
                v.dist = weight(u,v)
```
Disjoint-set data structure alias Union-Find structure

A graph with spanning trees of its subgraphs A, B, C.

Each subgraph is represented by one tree in the Union-Find structure.

Union-Find structure with trees corresponding to subgraphs A, B, C.

Tree roots represent the subgraphs and/or their spanning trees.

Implementation of the Union-Find structure.

A tree root is called here "boss", for brevity.

<table>
<thead>
<tr>
<th>node</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
<th>j</th>
<th>k</th>
<th>l</th>
</tr>
</thead>
<tbody>
<tr>
<td>boss</td>
<td>e</td>
<td>c</td>
<td>c</td>
<td>e</td>
<td>i</td>
<td>e</td>
<td>c</td>
<td>i</td>
<td>i</td>
<td>k</td>
<td>k</td>
<td>g</td>
</tr>
</tbody>
</table>
Disjoint-set data structure alias Union-Find structure

Only 3 operations are needed:

Initialize()
Union( representativeA, representativeB ) // merges the two sets represented
   // by the given two representatives
Find( nodeX )  // returns a representative of the set to which X belongs

```java
int [] boss; // = representative = tree root
int [] rank;

void UF_init( int n ) {
    boss = new int [n];
    rank = new int [n];
    for( int i = 0; i < n; i++ ) {
        boss[i] = i; // everybody's their own boss
        rank[i] = 0; // initial rank is 0
    }
}
```

Easy experiment, try it at home:
When the end nodes of the inspected edges are chosen uniformly randomly
then the average depth of a queried node in the Union-Find forest is less than 2.
void UF_union( int rootA, int rootB ) {
    if( rank[rootB] > rank[rootA] )
        boss[rootA] = rootB;
    else {
        boss[rootB] = rootA;
        if( rank[rootB] == rank[rootA] )
            // change rank?
            rank[rootA]++;
    }
}
Find with path compression

\begin{align*}
\text{node: } & a \quad b \quad c \quad d \quad e \\
\text{boss: } & \ldots \quad d \quad c \quad e \quad d \quad a \\
\text{node: } & a \quad b \quad c \quad d \quad e \\
\text{boss: } & \ldots \quad d \quad d \quad d \quad d \quad d \\
\end{align*}

```c
int UF_find( int a ) {
    int parent = boss[a];
    if( parent != a )
        boss[a] = UF_find( parent ); // path compression
    return boss[a];
}

int find( int a ) // for C experts
{ return( boss[a] == a ? a : (boss[a] = find(boss[a])) );}
```
Kruskal's algorithm and Union-Find scheme example

The diagram above illustrates a network of nodes and edges with weights. The node numbers are 0 to 17, and the edges are shown with their respective weights. The boss and rank tables below represent the Union-Find scheme:

<table>
<thead>
<tr>
<th>node</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>boss</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>rank</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Kruskal's algorithm and Union-Find scheme example
Kruskal's algorithm and Union-Find scheme example

```
node  boss  rank
  0   0  0
  1   1  1
  2   2
  3   3
  4   4
  5   5
  6   6
  7   7
  8   2
  9   9
 10  10
 11  11
 12  12
 13  13
 14  14
 15  14
 16  16
 17  17
```

Kruskal's algorithm and Union-Find scheme example

![Graph with nodes and edges labeled]

<table>
<thead>
<tr>
<th>node</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>boss</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>2</td>
<td>9</td>
<td>4</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>14</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>rank</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Kruskal's algorithm and Union-Find scheme example

node | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17
boss | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 2 | 9 | 4 | 11 | 12 | 14 | 14 | 16 | 17 | 17
rank  | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0
Kruskal's algorithm and Union-Find scheme example

![Graph](image-url)
Kruskal's algorithm and Union-Find scheme example

```
node   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
boss   0  1  2  3  4  5  6  7  2  9  4  4 12 14 14 14 16 16
rank   0  0  1  0  1  0  0  0  0  0  0  0  0  0  0  1  0  1  0
```
Kruskal's algorithm and Union-Find scheme example

node  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17
boss  | 0 | 1 | 2 | 4 | 4 | 5 | 6 | 7 | 2 | 9 | 4 | 4 | 12 | 14 | 14 | 14 | 16 | 16
rank  | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0
Kruskal's algorithm and Union-Find scheme example
Kruskal's algorithm and Union-Find scheme example
Kruskal's algorithm and Union-Find scheme example

```
Kruskal's algorithm and Union-Find scheme example

node    0  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
boss    0  0   2   4   4   5   6  14  2   4   4   4  12  14  14  14  16  16
rank    1  0   1   0   1   0   0   0   0   0   0   0   0   0   1   0   1   0
```
Kruskal's algorithm and Union-Find scheme example
Kruskal's algorithm and Union-Find scheme example

Node | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17
Boss | 0 | 0 | 2 | 4 | 4 | 4 | 14 | 14 | 2 | 4 | 4 | 4 | 12 | 14 | 14 | 14 | 16 | 16
Rank | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0

---

Rank | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0

---

22
Kruskal's algorithm and Union-Find scheme example

Node 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
Boss 0 0 2 4 2 4 14 14 2 4 4 4 12 14 14 14 16 16
Rank 1 0 2 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0
Kruskal's algorithm and Union-Find scheme example

\[
\text{node} \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \quad 16 \quad 17 \\
\text{boss} \quad 0 \quad 0 \quad 2 \quad 4 \quad 2 \quad 4 \quad 14 \quad 14 \quad 2 \quad 4 \quad 4 \quad 4 \quad 12 \quad 14 \quad 2 \quad 14 \quad 16 \quad 16 \\
\text{rank} \quad 1 \quad 0 \quad 2 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0
\]
Kruskal's algorithm and Union-Find scheme example

Node | Boss | Rank
--- | --- | ---
0   | 2    | 1
1   | 0    | 0
2   | 2    | 2
3   | 4    | 0
4   | 14   | 0
5   | 14   | 0
6   | 4    | 0
7   | 14   | 0
8   | 14   | 0
9   | 4    | 0
10  | 4    | 0
11  | 2    | 1
12  | 16   | 1
13  | 16   | 1
14  | 16   | 1
15  | 16   | 1
Kruskal's algorithm and Union-Find scheme example

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 | 8 | 1 | 15| 2 | 13| 3 | 7 | 4 | 11| 5 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 18| 27| 2 | 3 | 17| 10| 10| 5 | 20|
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 6 | 12| 7 | 14| 8 | 17| 9 | 10| 5 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 22| 9 | 9 | 23| 24| 25| 19|
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 12| 21| 4 | 1 | 24| 24| 1 | 4 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 13| 14| 14| 11| 16| 16| 6 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

**node**  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17
**boss** | 2 | 0 | 2 | 4 | 2 | 4 | 14| 14| 2 | 4 | 4 | 4 | 12 | 14 | 2 | 2 | 16 | 16
**rank**  | 1 | 0 | 2 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0

*path compression*
Kruskal's algorithm and Union-Find scheme example
Kruskal's algorithm and Union-Find scheme example

![Diagram showing Kruskal's algorithm and Union-Find scheme example]

<table>
<thead>
<tr>
<th>node</th>
<th>boss</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>16</td>
<td>0</td>
</tr>
</tbody>
</table>
Kruskal's algorithm and Union-Find scheme example

node | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
boss | 2  | 0  | 2  | 4  | 2  | 4  | 2  | 14 | 2  | 2  | 4  | 4  | 12 | 14 | 2  | 2  | 2  | 16 |
rank | 1  | 0  | 2  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  |
Kruskal's algorithm and Union-Find scheme example

```
node  0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17
boss  2  0  2  4  2  4  2  14  2  2  4  2  12  14  2  2  2  2
rank  1  0  2  0  1  0  0  0  0  0  0  0  0  0  0  0  1  0  1  0
```
Kruskal's algorithm and Union-Find scheme example

Node 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
Boss 2 0 2 4 2 2 2 14 2 2 4 2 12 14 2 2 2 2
Rank 1 0 2 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0

Path compression
Kruskal's algorithm and Union-Find scheme example

node: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
boss: 2 0 2 4 2 2 2 14 2 2 4 2 12 2 2 2 2 2
rank: 1 0 2 0 1 0 0 0 0 0 0 0 0 0 0 1 0 1 0
Kruskal's algorithm and Union-Find scheme example

![Graph Diagram]

<table>
<thead>
<tr>
<th>node</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>boss</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>14</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>rank</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

33
Kruskal's algorithm running time

When both union by rank and path compression are used, the running time spent on Union-Find operations in a graph with \( N \) nodes and \( M \) edges is \( O(M \cdot \alpha(N)) \), where \( \alpha(N) \) is the inverse function of \( f(x) = A(x, x) \), where \( A(x, y) \), is the Ackermann function, known to grow quite fast. In fact, for any graph representable in any conceivable machine \( \alpha(N) < 4 \).

Thanks to the inverse Ackermann function, all Union-Find operations run in amortized constant time in all practical situations. It means that the speed bottleneck in Kruskal's algorithm is the initial edge sorting. Sorting can be done in linear time when:

- edge values are strings (does not happen too often), apply Radix sort,
- edge values are integers in some (relatively) moderate range (e.g. 0..10 000, etc.), apply Counting sort,
- edge values are floats (more or less) uniformly distributed over some interval, apply Bucket sort.

Conclusion
In many practical situations, a careful implementation of Kruskal algorithm may run in \( \Theta(M) \) time.
### Ackermann function

\[ A(m, n) = \begin{cases} 
  n + 1 & \text{if } m = 0, \\
  A(m - 1, 1) & \text{if } m > 0, n = 0, \\
  A(m - 1, A(m, n - 1)) & \text{if } m > 0, n > 0.
\end{cases} \]

<table>
<thead>
<tr>
<th>(m) (\backslash) (n)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>(n + 1)</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>(n + 2)</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
<td>(2n - 3)</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>13</td>
<td>= 4(^2) − 3 = 2(^4) − 3</td>
<td>= 2(^5) − 3 = 2(^9) − 3</td>
<td>= 2(^6) − 3 = 2(^{12}) − 3</td>
<td>= 2(^7) − 3 = 2(^{16}) − 3</td>
<td>= 2(^8) − 3 = 2(^{20}) − 3</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>65533</td>
<td>= 2(^{22}) − 3</td>
<td>= 2(^{222}) − 3</td>
<td>= (#)</td>
<td>(##)</td>
<td>(###)</td>
</tr>
<tr>
<td>5</td>
<td>65533</td>
<td>= 2(^{222}) − 3</td>
<td>= 2(^{2222}) − 3</td>
<td>= 2(^{22222}) − 3</td>
<td>= 2(^{222222}) − 3</td>
<td>= 2(^{2222222}) − 3</td>
<td>= 2(^{22222222\ldots2}) − 3</td>
</tr>
<tr>
<td>6</td>
<td>(A(6,0))</td>
<td>= (A(5,1))</td>
<td>(2)(^{22222222\ldots2}) − 3</td>
<td>[{]</td>
<td>= 65536</td>
<td>= 65536</td>
<td>= too big to fit here …</td>
</tr>
<tr>
<td>7</td>
<td>()</td>
<td>()</td>
<td>()</td>
<td>()</td>
<td>()</td>
<td>()</td>
<td>()</td>
</tr>
</tbody>
</table>

1. 0-th value in each line = 1st value in the previous line.
2. Value \(X\) in the \(j\)-th column \((j > 0)\) on the current line is equal to the value on the previous line which column index is equal to the value written to left of the value \(X\) on the current line \((= \text{ in the } (j-1)\text{-th column})\).
\[ A(4, 2) = (\#) = 2^{2^{2^2}} - 3 = 2^{2^{2^4}} - 3 = 2^{2^{16}} - 3 = 2^{65536} - 3 = \]

= (next 3 slides with 19729 decimal digits) =

\[ A(4, 3) = (\#\#) = 2^{(\#)} - 3 = ? \]

\[ A(4, 4) = (\#\#\#) = 2^{2^{(\#)}} - 3 = ?? \]

Informal discussions concerning big integers:

http://www.scottaaronson.com/writings/bignumbers.html