Bin Sort

(a) Input chain

(b) Nodes in bins

(c) Sorted chain
Bin Sort Code (first take)

```c++
void binSort(chain<studentRecord>& theChain, int range)
{// Sort by score.
  // initialize the bins
  chain<studentRecord> *bin;
  bin = new chain<studentRecord> [range + 1];

  // distribute student records from theChain to bins
  int numberOfElements = theChain.size();
  for (int i = 1; i <= numberOfElements; i++)
  {
    studentRecord x = theChain.get(0);
    theChain.erase(0);
    bin[x.score].insert(0,x);
  }

  // collect elements from bins
  for (int j = range; j >= 0; j--)
  { while (!bin[j].empty())
    { studentRecord x = bin[j].get(0);
      bin[j].erase(0);
      theChain.insert(0,x);
    }
  }
  delete [] bin;
}
```
void binSort(chain<studentRecord>& theChain, int range) {
    // Sort by score.
    // initialize the bins
    chain<studentRecord> *bin;
    bin = new chain<studentRecord> [range + 1];

    // distribute student records from theChain to bins
    int numberOfElements = theChain.size();
    for (int i = 1; i <= numberOfElements; i++)
    {
        studentRecord x = theChain.get(0);
        theChain.erase(0);
        bin[x.score].insert(0, x);
    }

    // collect elements from bins
    for (int j = range; j >= 0; j--)
    {
        while (!bin[j].empty())
        {
            studentRecord x = bin[j].get(0);
            bin[j].erase(0);
            theChain.insert(0, x);
        }
    }

    delete [] bin;
}
Optimization

How can we avoid all these extra allocations and deletions?

And make the collection phase speedier too?

Make this a member function of Chain class

Keep track of both ends of each chain.

Rapid concatenation of chains in collection phase.
### Optimized Bin Sort

1. `template<class T>`
2. `void chain<T>::binSort(int range)`
3. `{ // Sort the nodes in the chain. //create and initialize the bins`  
4. `chainNode<T> **bottom, **top;`  
5. `bottom = new chainNode<T>* [range + 1];`  
6. `top = new chainNode<T>* [range + 1];`  
7. `for (int b = 0; b <= range; b++)`  
8. `bottom[b] = NULL;`  
9. `...`
Optimized Bin Sort (cont.)

1. // distribute to bins
2. for (; firstNode != NULL; firstNode=firstNode->next)
3. {
4.   // add firstNode to proper bin
5.     // type conversion to int
6.     int theBin = firstNode->element;
7.     if (bottom[theBin] == NULL) // bin is empty
8.         bottom[theBin] = top[theBin] = firstNode;
9.     else
10.        // bin not empty
11.        top[theBin]->next = firstNode;
12.        top[theBin] = firstNode;
13.   }
14. }
// collect from bins into sorted chain
chainNode<T> *y = NULL;
for (int theBin = 0; theBin <= range; theBin++)
    if (bottom[theBin] != NULL)
        { // bin not empty
            if (y == NULL) // first nonempty bin
                firstNode = bottom[theBin];
            else // not first nonempty bin
                y->next = bottom[theBin];
            y = top[theBin];
        }
    if (y != NULL)
        y->next = NULL;

delete [] bottom;
delete [] top;
Bin Sort is Stable

Elements with the same value will end up in the same order in the resulting sorted array as they were in originally.
What to do with larger ranges?

Radix Sort lets you sort integers in the range 0 through \( n^c - 1 \) in \( O(n) \).

Choose a radix \( r \), then use that \( r \) to decompose each number.

Apply Bin Sort to each digit of decomposed number.
Example with Radix 10

(a) Input chain

(b) Chain after sorting on least significant digit

(c) Chain after sorting on second-least significant digit

(d) Chain after sorting on most significant digit
Computing the Digits

With radix 10
\[ x \% 10; \frac{(x \% 100)}{10}; \frac{(x \% 1000)}{100}; \ldots \]

With radix 100
\[ x \% 100; \frac{(x \% 10000)}{100}; \frac{(x \% 1000000)}{10000}; \ldots \]

Generalized to any radix
\[ x \% r; \frac{(x \% r^2)}{r}; \frac{(x \% r^3)}{r^2}; \ldots \]
A **polygon** is a closed planar figure with three or more straight edges.

A polygon is **convex** iff all line segments that join two points on or in the polygon include no point that is outside the polygon.
Convex Hull (definition)

The **convex hull** of a set \( S \) of points in the plane is the smallest convex polygon that contains all these points. The vertices (i.e., corners) of this polygon are the **extreme points** of \( S \).

- ... Planar point
- ... Extreme point
Identifying Extreme Points

(a) Point ordering

(b) Counterclockwise angles
Convex Hull Algorithm

**Step 1:**
[Handle degenerate cases]
If $S$ has fewer than three points, return $S$.
If all points lie on a straight line, compute the endpoints of the smallest line that includes all points of $S$ and return these two points.

**Step 2:**
[Sort by polar angle]
Find a point $X$ that is inside the convex hull of $S$.
Sort $S$ by polar angle and within polar angle by distance from $X$.
Create a doubly linked circular list of points using the above order.
Let right link to the next point in the order and left link to the previous point.

**Step 3:**
[Eliminate nonextreme points]
Let $p$ be the point that has the smallest $y$-coordinate (break a tie, if any, by selecting the one with largest $x$-coordinate).
Pseudocode

for (x = p, rx = point to the right of x; x != rx;)
{
    rrx = point to the right of rx;
    if (angle formed by x, rx, and rrx is ≤ 180 degrees)
    {
        delete rx from the list;
        rx = x; x = point on left of rx;
    }
    else {x = rx; rx = rrx;}
}
Complexity

Step 1 (degenerate cases) takes $O(n)$ time

Step 2 (sort) can be done in $O(n \log n)$ time

Step 3 (elimination of non-extreme points) takes $O(n)$ time

Total Complexity: $O(n \log n)$
What is an equivalence class?

Suppose we have a set $U = 1, 2, \cdots, n$ of $n$ elements and a set $R = (i_1, j_1), (i_2, j_2), \cdots, (i_r, j_r)$ of $r$ relations

The relation $R$ is an equivalence relation iff the following conditions are true:

1. $(a, a) \in R$ for all $a \in U$ (the relation is reflexive).
2. $(a, b) \in R$ iff $(b, a) \in R$ (the relation is symmetric).
3. $(a, b) \in R$ and $(b, c) \in R$ imply that $(a, c) \in R$ (the relation is transitive).
Equivalence Classes

Two elements $a$ and $b$ are equivalent if $(a, b) \in R$.

An **equivalence class** is defined to be a maximal set of equivalent elements. 

*Maximal* means that no element outside the class is equivalent to an element in the class.

Since it is not possible for an element to be in more than one equivalence class, an equivalence relation partitions the universe $U$ into disjoint classes.
Circuit Example
Offline Equivalence Class Problem

We are given $n$ and $R$ and we need to determine the equivalence classes.

From the definition of an equivalence class, it follows that each element is in exactly one equivalence class.
Online Equivalence Class Problem

We begin with $n$ elements, each in a separate equivalence class.

We are to process a sequence of the operations:

1. combine(a,b) ⋯ combines the equivalence classes that contain elements a and b into a single class

2. find(theElement) ⋯ determines the class that currently contains element theElement

So combine(a,b) is equivalent to:
```
classA = find(a);
classB = find(b);
if (classA != classB)
    unite(classA, classB);
```
combine(a,b) is equivalent to

classA = find(a);
classB = find(b);
if (classA != classB)
    unite(classA, classB);
Union Find Solution Using Arrays

```c
int *equivClass,  // equivalence class array
    n;          // number of elements

void initialize(int numberOfElements)
{
    // Initialize numberOfElements classes with one element each.
    n = numberOfElements;
    equivClass = new int [n + 1];
    for (int e = 1; e <= n; e++)
        equivClass[e] = e;
}

void unite(int classA, int classB)
{
    // Unite the classes classA and classB.
    // Assume classA != classB
    for (int k = 1; k <= n; k++)
        if (equivClass[k] == classB)
            equivClass[k] = classA;
}

int find(int theElement)
{
    // Find the class that contains theElement.
    return equivClass[theElement];
}
```
Complexity using Arrays

The initialize and unite methods have complexity $\Theta(n)$

Complexity of find is $\Theta(1)$

Therefore, total complexity is given by:

$\Theta(n+u*n+f) = \Theta(u*n+f)$. 
An array node[1:n] of type equivNode is used to represent the n elements together with the equivalence class chains.

node[e].equivClass is both the value to be returned by find(e) and an integer pointer to the first node in the chain for the equivalence class node[e].equivClass.

node[e].size is defined only if e is the first node on a chain. In this case node[e].size is the number of nodes on the chain that begins at node[e].

node[e].next gives the next node on the chain that contains node e. Since the nodes in use are numbered 1 through n, a NULL pointer can be simulated by the integer 0.
Code for Chain Union-Find Solution

1. equivNode *node; // array of nodes
2. int n; // number of elements
3.
4. void initialize(int numberOfElements)
5. {
6.   // Initialize numberOfElements classes with one element each.
7.   n = numberOfElements;
8.   node = new equivNode [n + 1];
9. 
10.  for (int e = 1; e <= n; e++)
11.  {
12.    node[e].equivClass = e;
13.    node[e].next = 0; // no next node on chain
14.    node[e].size = 1;
15.  }
16. }
17.
void unite(int classA, int classB)
{// Unite the classes classA and classB.
// Assume classA != classB
// classA and classB are first elements in their chains

    // make classA smaller class
    if (node[classA].size > node[classB].size)
        swap(classA, classB);

    // change equivClass values of smaller class
    int k;
    for (k = classA; node[k].next != 0; k = node[k].next)
        node[k].equivClass = classB;
    node[k].equivClass = classB; // last node in chain

    // insert chain classA after first node in chain classB
    // and update new chain size
    node[classB].size += node[classA].size;
    node[k].next = node[classB].next;
    node[classB].next = classA;
}

int find(int theElement)
{// Find the class that contains theElement.
    return node[theElement].equivClass;
}
Complexity of Chain Solution

**Lemma 6.1**
*If we start with \( n \) classes that have one element each and perform \( u \) nonredundant unions, then*

1. No class has more than \( u + 1 \) elements.
2. At least \( n - 2u \) singleton classes remain.
3. \( u < n \).

And therefore, the time needed to perform the \( u \) unions is \( O(u \log u) \).

The complexity of the initialization and the sequence of \( u \) unions and \( f \) finds is \( O(n+u \log u+f) \).