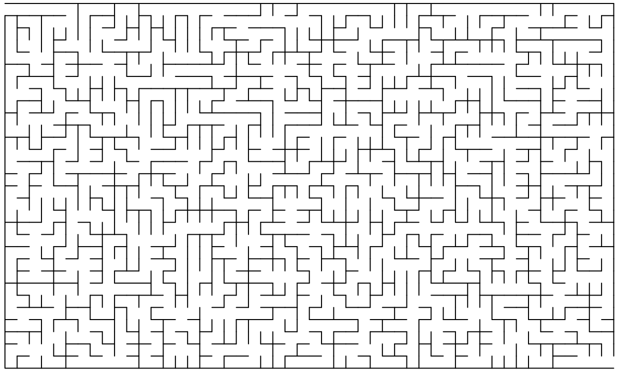
Algorithms and Analysis

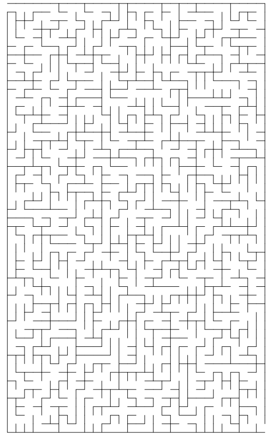
Lesson 12: *Use Arrays for Fast Set Algorithms*



Equivalent classes, Disjoint Set, Fast Sets

Outline

1. Equivalent Classes
2. Disjoint Sets
3. Fast Sets

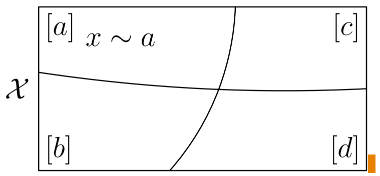


Equivalence Relations

- Given a set of elements $\mathcal{X} = \{x_1, x_2, \ldots\}$ and a binary relationship $\sim$ with the following properties

  (Reflexivity) For every element $x \in \mathcal{X}$, $x \sim x$
  (Symmetry) For every two elements $x, y \in \mathcal{X}$ if $x \sim y$ then $y \sim x$
  (Transitivity) For every three elements $x, y, z \in \mathcal{X}$ if $x \sim y$ and $y \sim z$ then $x \sim z$

- Then $\sim$ defines a partitioning of the set into **equivalence classes**



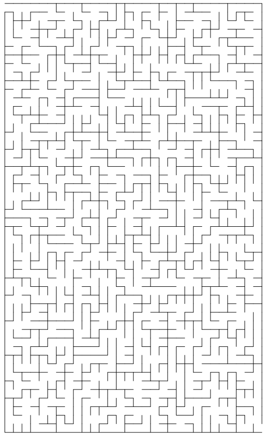
Example of Equivalence Classes

- Although, equivalent classes sound very mathematical they often provide a useful formalisation of the real world
- E.g. Pairs of web pages with a link in each direction between them
- Consider web pages in the same equivalence class if you can get from one to the other by clicking links
- Partitions the web into linked domains
- Friendship relations in social media

Dynamic Equivalence Classes

- Finding equivalence classes is rather easy using graph traversal algorithms
- However, as our web example suggests, there are applications where equivalence classes change over time
- Adding a link could join two domains which were separate
- We will see this is a useful idea both for building mazes and (in a later lecture) for finding minimum spanning trees
- Building a data structure which finds equivalence classes where the equivalence relation changes over time is challenging, but fortunately there is an elegant solution to this

Outline



Union-Find

- In the union-find algorithm we have a set of objects $x \in \mathcal{S}$ which are to be grouped into subsets $\mathcal{S}_1$, $\mathcal{S}_2$, . . .
- Initially each object is in its individual subset (no relationships)
- We want to make the **union** of two subsets (add relationship between elements)
- We also want to **find** the subset given an element
- This is a common problem for which we will write a class `DisjointSets` to perform fast `union`s and `find`s

DisjointSets

- We want to create a class

```
class DisjointSets
{
    DisjointSets(int numElements) {/* Constructor */}
    int find(int x) {/* Find root */}
    void union_(int root1, int root2) {/* Union */}

  private:
    int* s;
}
```

- Where `find(x)` returns a unique identifier for the subset which element `x` belongs to
- The array `s` contains labelling information to implement `find(x)`

The Union-Find Dilemma

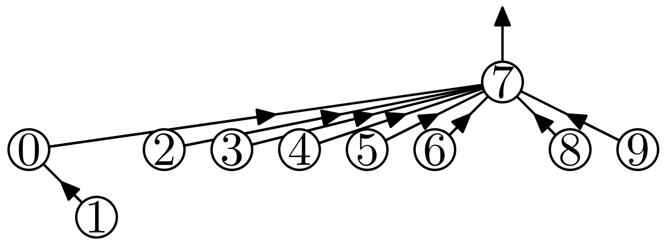
- A natural algorithm to perform finds is to maintain an array returning a subset label for each element—this makes `find` fast
- However, every time we combine two subset we have to change all the labels in this array (taking $O(n)$ operations)
- If we are unlucky the cost of performing n unions is $\Theta(n^2)$
- If we ensure that we relabel the smaller subset then the time complexity is $\Theta(n \log(n))$
- Fast *finds* seems to give slow(ish) *unions*
- What about the other way around?

Fast Union

- To achieve fast unions we can represent our disjoint sets as a forest (many disjoint trees)
- Every time we perform a union we make one of the trees point to the head of the other tree
- The cost of `find` depends on the depth of the tree
- To make unions efficient we make the shallow tree a subtree of the deeper tree

Putting it Together

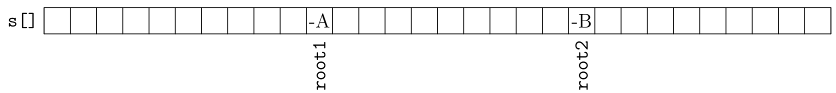
`find(6)=7`



0	1	2	3	4	5	6	7	8	9
7	0	7	7	7	7	7	-3	7	7

Smart Union

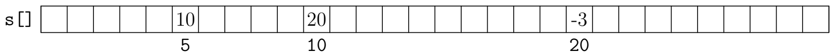
```
DisjointSets::DisjointSets(int numElements)
{
    s = new int[numElements];
    for(int i=0; i<numElements; i++)
        s[i] = -1;                      // roots are negative number
}

void DisjointSets::union_(int root1, int root2)
{
    if (s[root2]<s[root1]) {            // root2 is deeper
        s[root1] = root2;               // make root2 the root
    } else {
        if (s[root1]==s[root2])
            s[root1]--;                 // update height if same
        s[root2] = root1;               // make root1 new root
    }
}
```



Path Compression

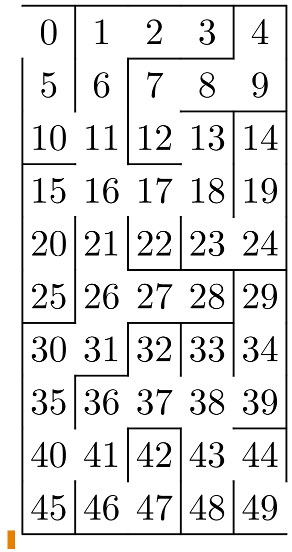
- To speed up `find` we relabel all nodes we visit during `find` by the root label

```
int DisjointSets::find(int index)
{
    if (s[index]<0)
        return index;
    else
        return s[index] = find(s[index]);
}
```



Mazes

- Union-Find is a data structure which can occur in very different applications
- One application is building a maze
- Start from a complete lattice
- Remove a randomly chosen edge if it connects two unconnected regions
- Stop when the start and end cell are connected
- Or better after all cells are connected



Time Complexity of Union-Find

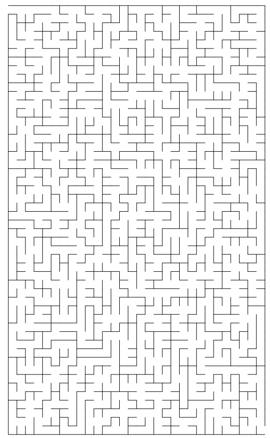
- If we perform M finds and N unions then the time complexity is $O\big(M\log_2^*(N)\big)$
- Where $\log_2^*(N)$ is the number of times you need to apply the logarithm function before you get a number less than 1
- In practice $\log_2^*(N) \leq 5$ for all conceivable N

$\log_2(\log$ [illegible]

- The proof of this time complexity is rather involved

Outline

1. Equivalent Classes
2. Disjoint Sets
3. **Fast Sets**



Comparison of Sets

- Binary Search Trees: $O(\log_2(n))$, general purpose
- Hash tables: $O(1)$, but need to compute hash, slow iterator when sparse, general purpose
- B-trees: $O((k-1)\log_k(n))$ very complicated, used for large amounts of data
- Tries: $O(\log_k(n))$ for large k expensive in memory, complicated to code efficiently

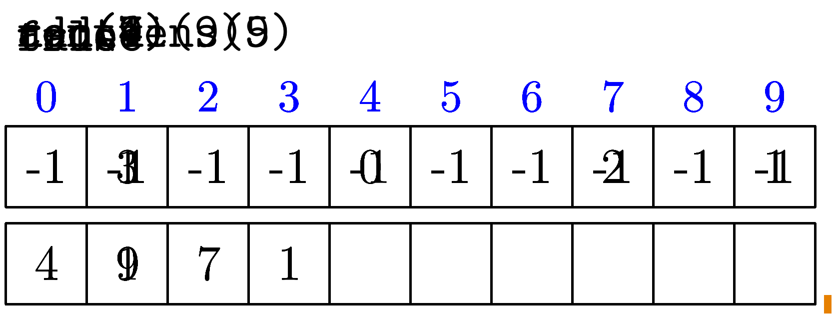
What Set to Use?

- A PhD student and I were working on writing a fast solver for a combinatorial optimisation problem
- We had to choose one variable to change out of a small number of possible variables
- Each time we changed a variable then we had to update the list of possible variables (remove some variables add others)
- We wanted a data structure which had quick add and remove and where we could choose a variable at random—what should we use?

Bounded Set

- One special feature is that we knew we only wanted the set to contain integers between 0 and n (where n might be 100 000)
- This allowed us to use an array to represent whether an integer belong to that set
- But how do we find a random element of the set quickly?
- Use another array of course!

FastSet



Implementation

```
class FastSet {
  private:
    int* indexArray;
    int* memberArray;
    int noMembers;

  public:
    FastSet(int n) {
      indexArray = new int[n];
      memberArray = new int[n];
      for(int i=0; i<n; i++) {
           indexArray [i] = -1;
      }
      noMembers = 0;
  }

  int size() {
     return noMembers;
  }
```


Add and Remove

```
bool add(int i) {
   if (indexArray[i]>-1)
      return false;
   memberArray[noMembers] = i;
   indexArray[i] = noMembers;
   ++noMembers;
   return true;
}

bool remove(int i) {
   if (indexArray[i]==-1)
         return false;
   --noMembers;
   memberArray[indexArray[i]] = memberArray[noMembers];
   indexArray[memberArray[noMembers]] = indexArray[i];
   indexArray[i] = -1;
   return true;
}
```


Collection Methods

```
void clear() {
   for(int i=0; i<noMembers; i++) {
      indexArray[memberArray[i]] = -1;
   }
   noMembers = 0;
}

bool isEmpty() {
   return noMembers==0;
}

int* begin() {return &memberArray[0];}
int* end() {return &memberArray[noMembers];}

}
```


And Random?

- We can add additional methods taking advantage of the classes strength

```
private:
  random_device rd;  // Seed for the random number engine
  mt19937 gen(rd()); // Mersenne Twister RNG

public:
  int getRandomElement() {
    return memberArray[uniform_int_distribution<int>(0, noMembers)];
  }
```

- Need to use FastSet signature to use this

```
  FastSet fastSet(n);
  ⋮
  int r = fastSet.getRandomElement();
```

Speed Up

- We compared our algorithm to a very highly regarded "state-of-the-art" algorithm
- For large problems we were over 10 times faster because of this data structure
- The competitor algorithm used a complex tree structure instead of the simple array
- Why? The array solution isn't in the books

Lessons

- If you have a bounded set then using an array is usually going to be very fast $O(1)$ (or $O(\log^*(n))$)
- These data structures are not general purpose for solving every day problems (c.f. `vector<T>`, `set<T>` and `map<T>`)
- They are "back pocket" data structures that solve problems that come up often enough that they are worth knowing about
- Sometimes good algorithms are not documented, but it doesn't mean they don't exist