Sorting

Sandra Batista, Mark Redekopp, and David Kempe

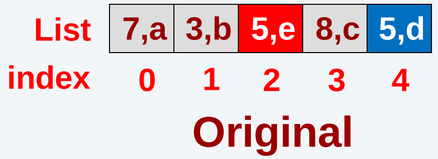
1.1–1.2

Applications of Sorting

- Binary Search
- Finding the set intersection

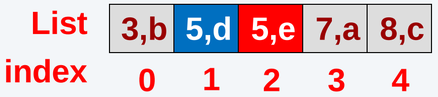
Sorting Stability

- A sort is stable if the order of equal items in the original list is maintained in the sorted list
  - Good for searching with multiple criteria



Original

Stable Sorting



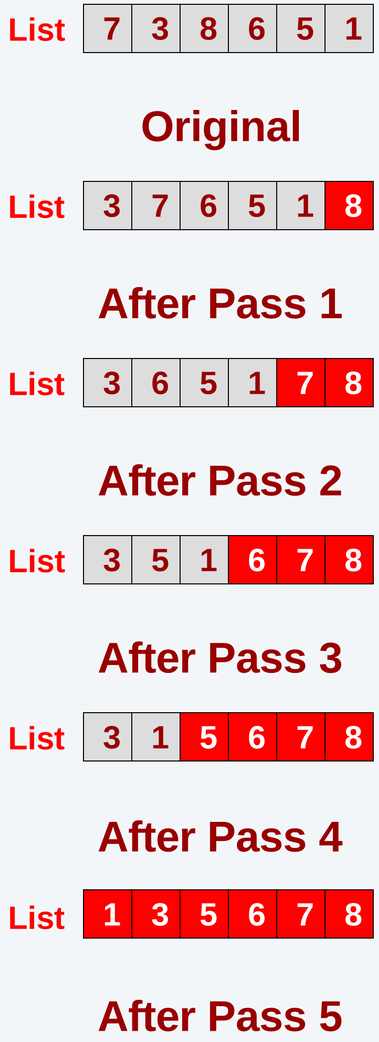
Unstable Sorting

Algorithm Efficiency

QUADRATIC COMPARISON SORTING

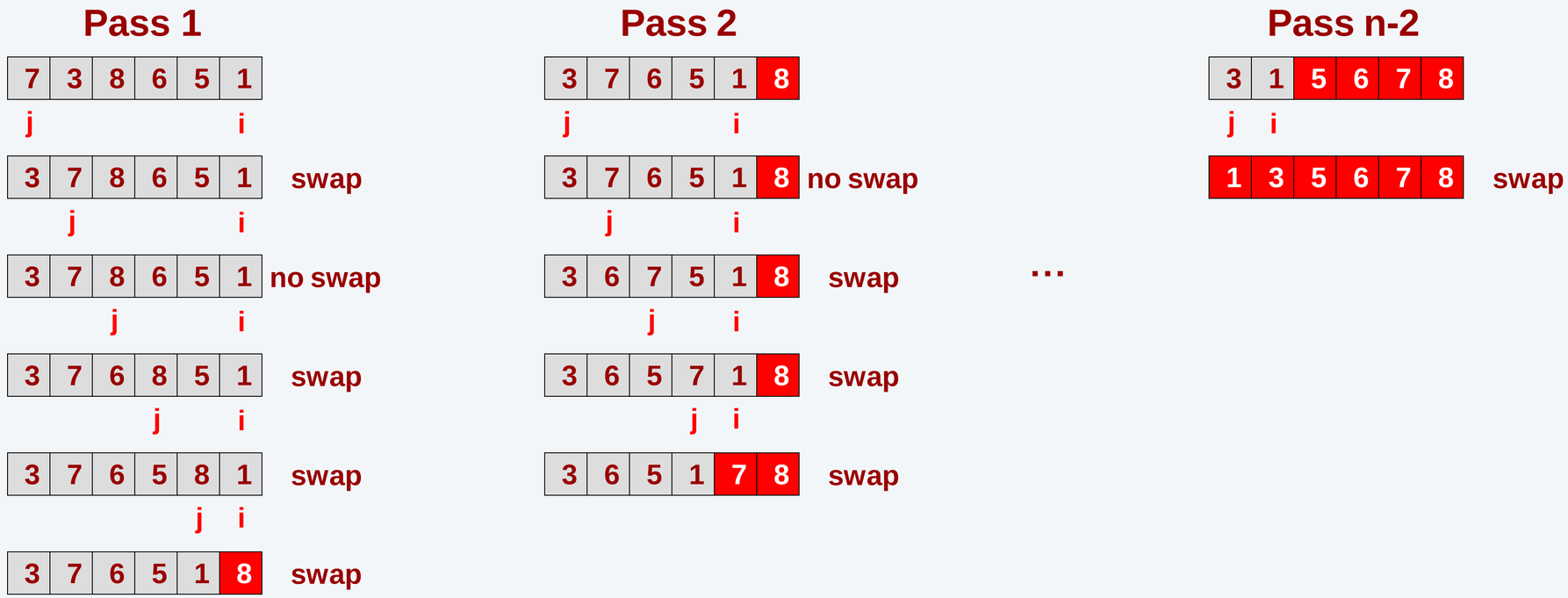
Bubble Sorting

- Main Idea: Bubble up the largest value to the greatest index on list



Bubble Sort Algorithm

```
void bsort(vector<int> mylist)
{
  int i ;
  for(i=mylist.size()-1; i > 0; i--){
     for(j=0; j < i; j++){
        if(mylist[j] > mylist[j+1]) {
           swap(j, j+1)
  }  }  }
}
```



Bubble Sort Analysis

- Best Case Complexity:
  - When already sorted (no swaps) but still have to do all compares
  - $O(n^2)$
- Worst Case Complexity:
  - When sorted in descending order
    - $O(n^2)$

```
void bsort(vector<int> mylist)
{
  int i ;
  for(i=mylist.size()-1; i > 0; i--){
     for(j=0; j < i; j++){
        if(mylist[j] > mylist[j+1]) {
           swap(j, j+1)
  }  }  }
}
```

Bubble Sort Analysis

- Is Bubble Sort stable?

```
void bsort(vector<int> mylist)
{
  int i ;
  for(i=mylist.size()-1; i > 0; i--){
     for(j=0; j < i; j++){
        if(mylist[j] > mylist[j+1]) {
           swap(j, j+1)
  }  }  }
}
```

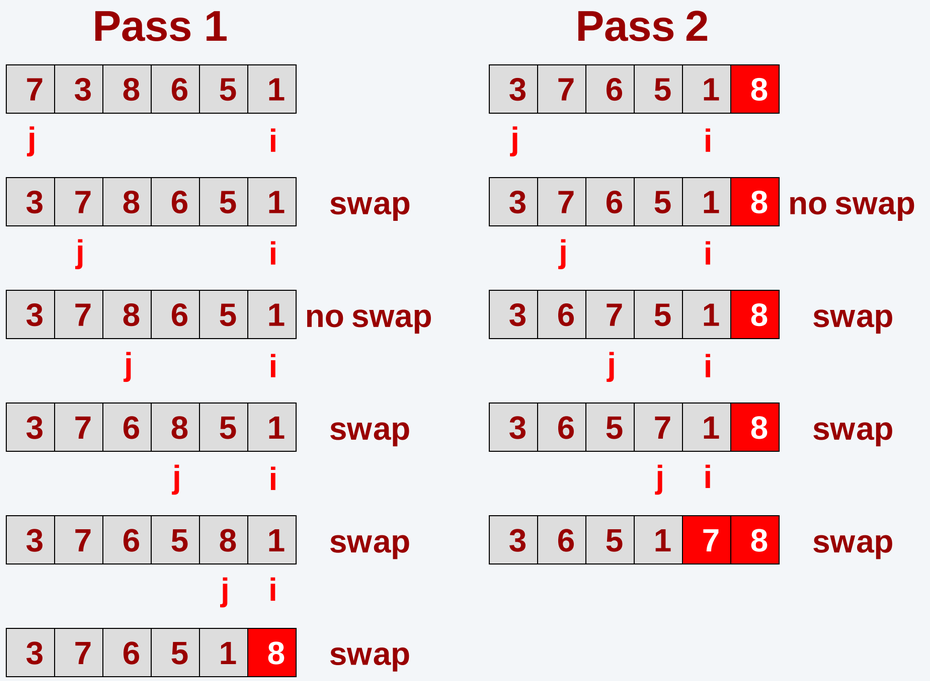
Loop Invariants

- Loop invariant:

  Logical predicate, P(k), that is true before loop begins and after each iteration

- What is loop invariant for outer loop of Bubble Sort?

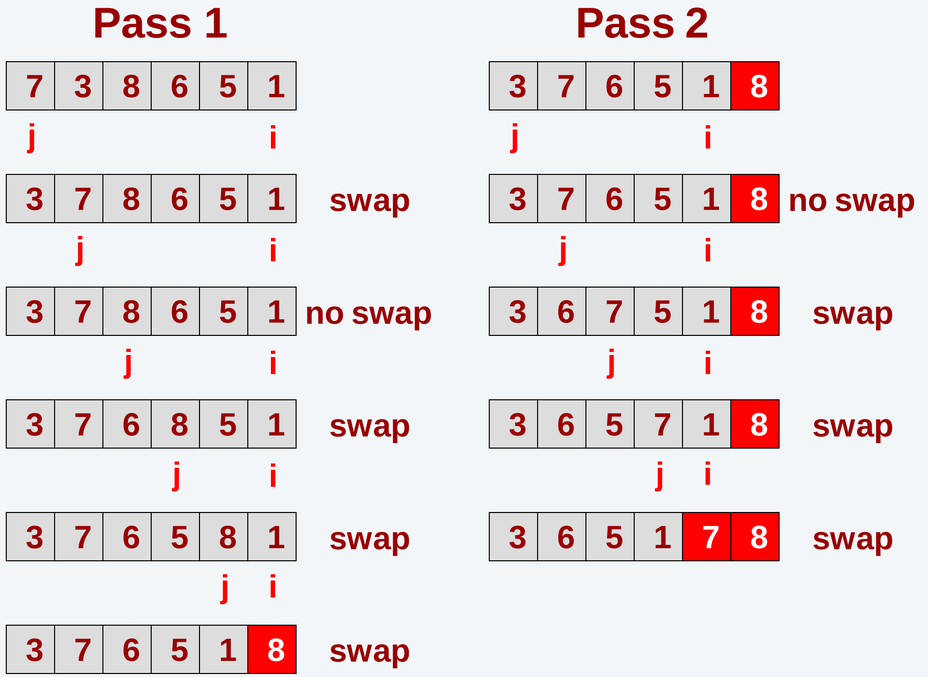
```
void bsort(vector<int> mylist)
{
  int i ;
  for(i=mylist.size()-1; i > 0; i--){
     for(j=0; j < i; j++){
        if(mylist[j] > mylist[j+1]) {
           swap(j, j+1)
  }  }  }
}
```



Bubble Sort Loop Invariant

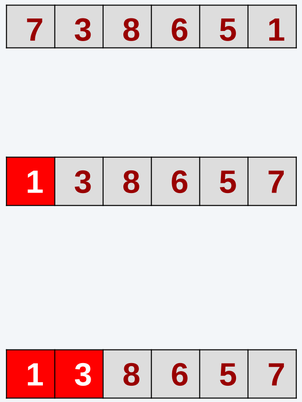
- What is true after the k-th iteration?
  - All data at indices n-k and above are sorted
    - $\forall i, i \geq n-k: a[i] < a[i+1]$
  - All data at indices below n-k are less than the value at n-k
    - $\forall i, i < n-k: a[i] < a[n-k]$

```
void bsort(vector<int> mylist)
{
  int i ;
  for(i=mylist.size()-1; i > 0; i--){
     for(j=0; j < i; j++){
        if(mylist[j] > mylist[j+1]) {
           swap(j, j+1)
  }  }  }
}
```



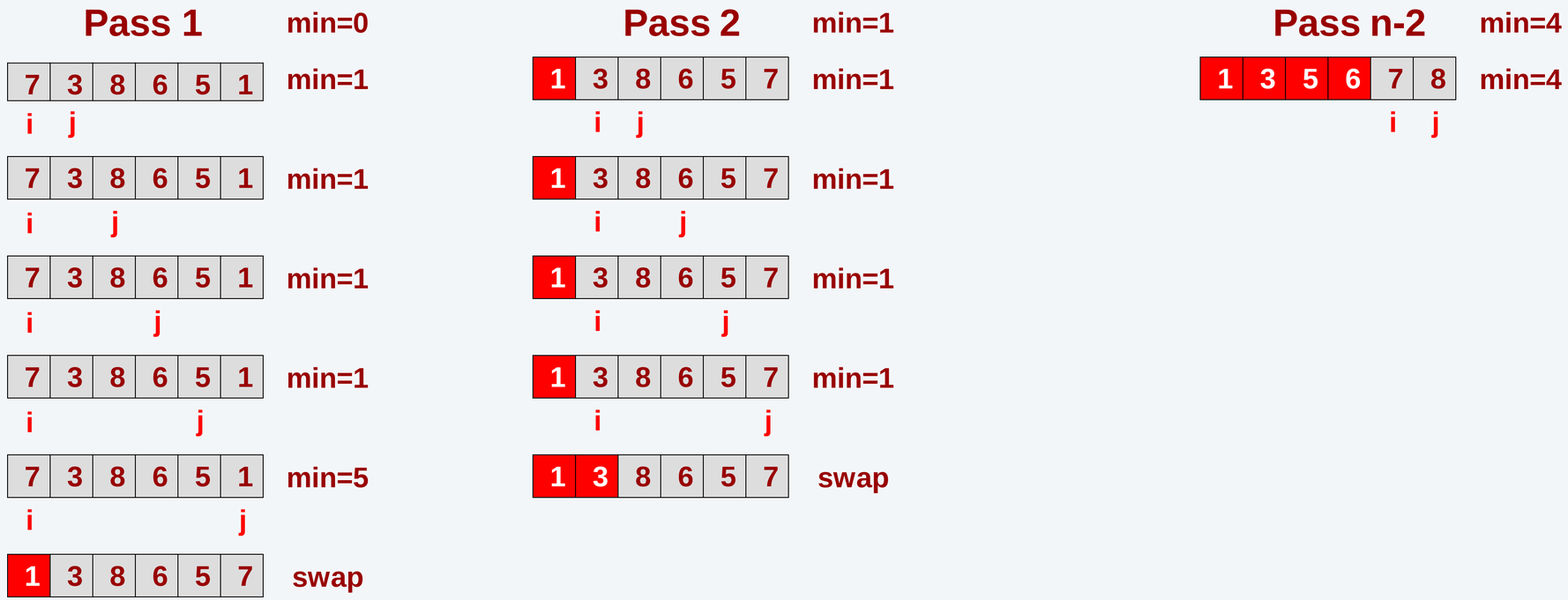
Selection Sort

- Selection sort finds min (or max) and puts at smallest unsorted or (greatest unsorted) index



Selection Sort Algorithm

```
void ssort(vector<int> mylist)
{
   for(i=0; i < mylist.size()-1; i++){
      int min = i;
      for(j=i+1; j < mylist.size; j++){
         if(mylist[j] < mylist[min]) {
            min = j
      }  }
      swap(mylist[i], mylist[min])
}}
```



Selection Sort Analysis

- Best Case Complexity:
  - Sorted already
  - $O(n^2)$
- Worst Case Complexity:
  - When sorted in descending order
  - $O(n^2)$

```
void ssort(vector<int> mylist)
{
  for(i=0; i < mylist.size()-1; i++){
     int min = i;
     for(j=i+1; j < mylist.size; j++){
        if(mylist[j] < mylist[min]) {
           min = j
     }  }
     swap(mylist[i], mylist[min])
}}
```

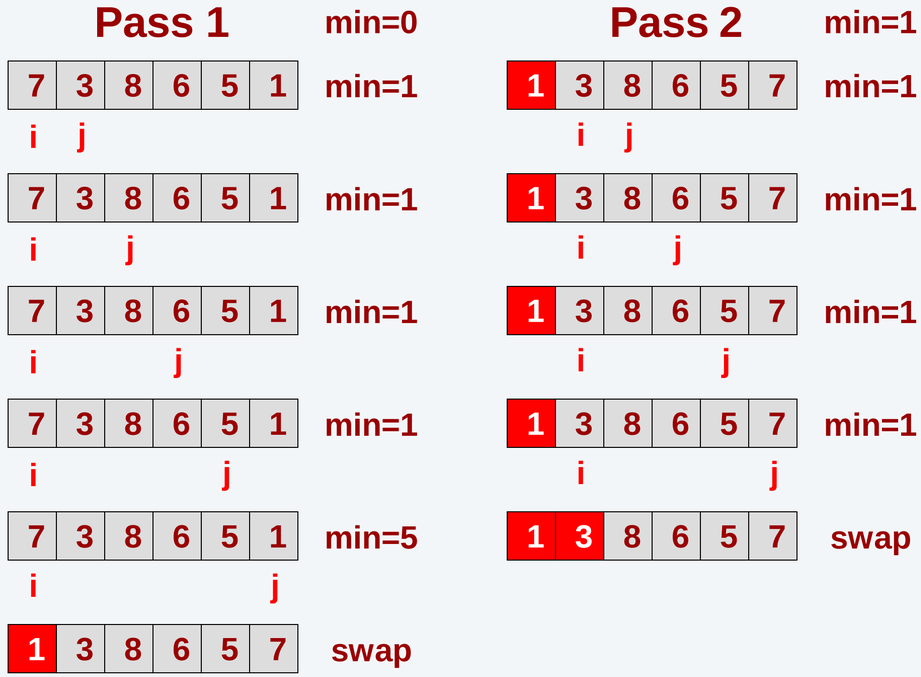
Is Selection Sort Stable?

```
void ssort(vector<int> mylist)
{
  for(i=0; i < mylist.size()-1; i++){
     int min = i;
     for(j=i+1; j < mylist.size; j++){
        if(mylist[j] < mylist[min]) {
           min = j
     }  }
     swap(mylist[i], mylist[min])
}}
```

Selection Sort Loop Invariant

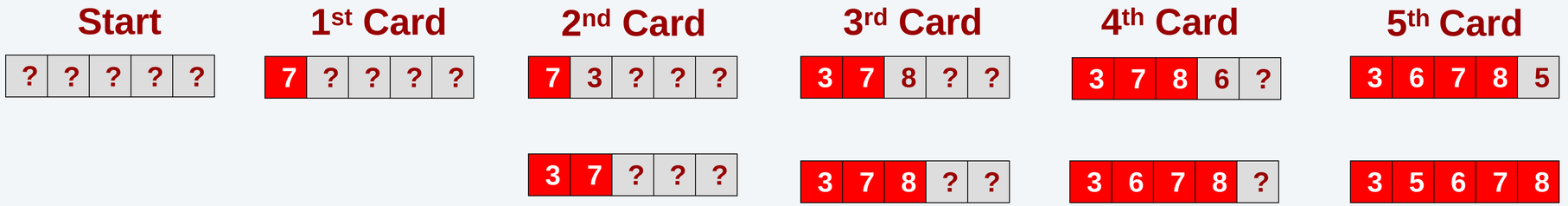
- What is true after the k-th iteration?
  - All data at indices less than k are sorted
    - $\forall i, i < k: a[i] < a[i+1]$
  - All data at indices k and above are greater than the value at k
    - $\forall i, i \geq k: a[k] < a[i]$

```
void ssort(vector<int> mylist)
{
   for(i=0; i < mylist.size()-1; i++){
      int min = i;
      for(j=i+1; j < mylist.size; j++){
         if(mylist[j] < mylist[min]) {
            min = j
      }  }
      swap(mylist[i], mylist[min])
}}
```



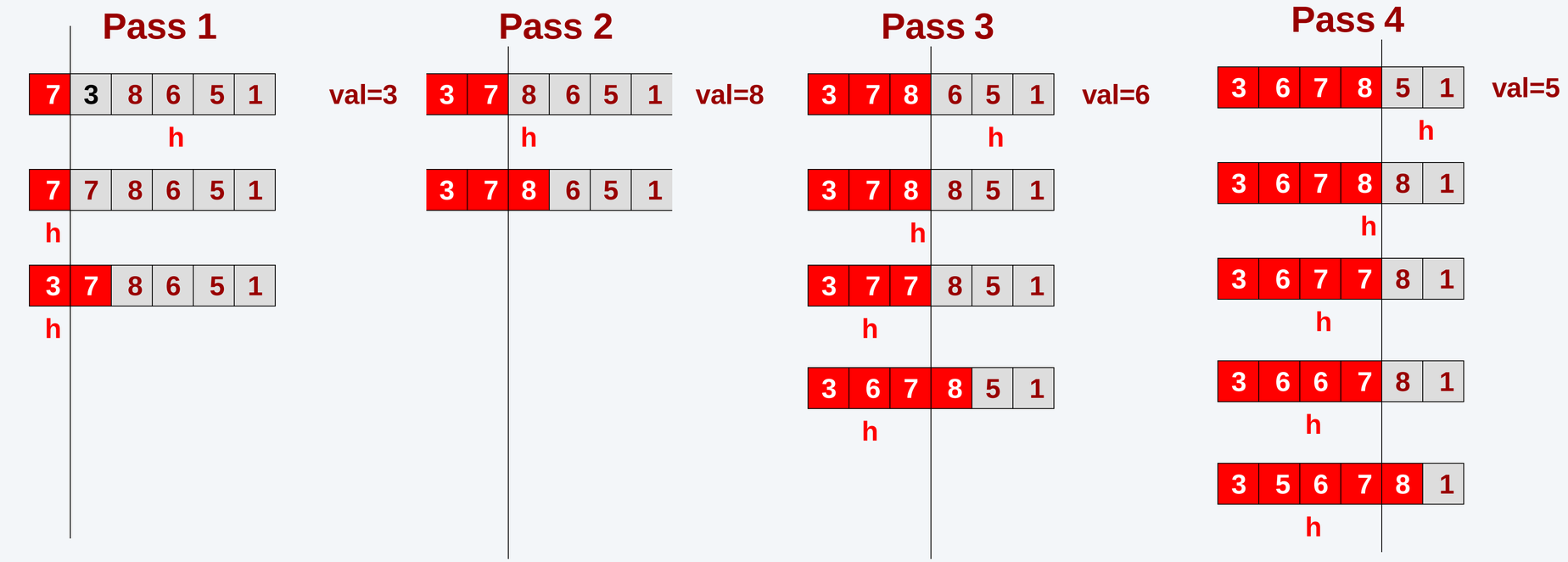
Insertion Sort Algorithm

- Sort one element of the array at a time inserting into the correct position
- Consider how you sort a hand of cards
  - You pick up the first and assume it is sorted
  - You pick up the second and insert it at the right position, etc.



Insertion Sort Algorithm

```
void isort(vector<int> mylist)
{  for(int i=1; i < mylist.size(); i++){
     int val = mylist[i];
     hole = i
     while(hole > 0 && val < mylist[hole-1]){
        mylist[hole] = mylist[hole-1];
        hole--;
     }
     mylist[hole] = val;}
}}
```



Insertion Sort Analysis

- Best Case Complexity:
  - Sorted already
  - O(n)
- Worst Case Complexity:
  - When sorted in descending order
  - $O(n^2)$

```
void isort(vector<int> mylist)
{  for(int i=1; i < mylist.size()-1; i++){
     int val = mylist[i];
     hole = i
     while(hole > 0 && val < mylist[hole-1]){
        mylist[hole] = mylist[hole-1];
        hole--;
     }
     mylist[hole] = val;
}}
```

Insertion Sort Analysis

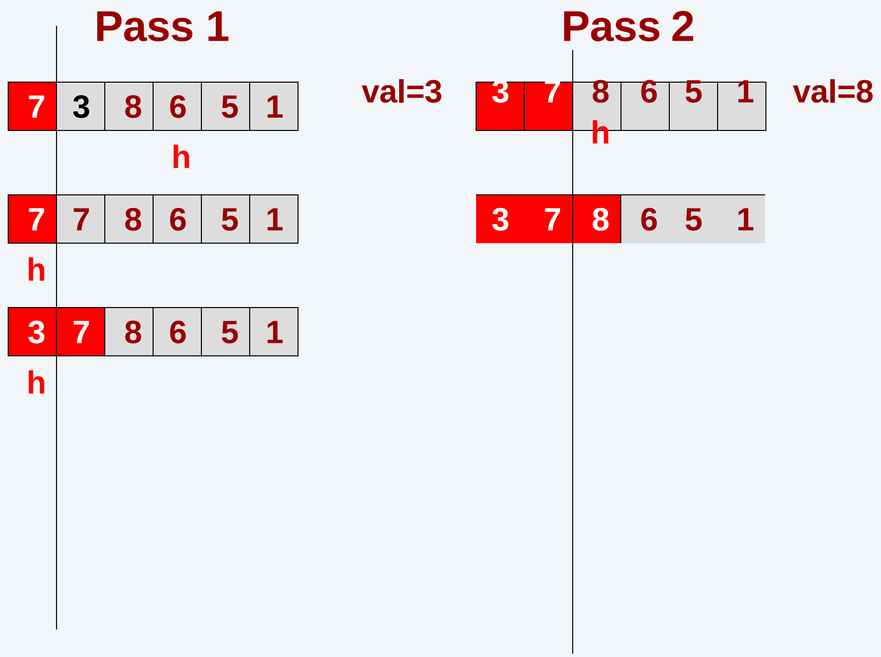
- Is Insertion Sort Stable?

```
void isort(vector<int> mylist)
{  for(int i=1; i < mylist.size()-1; i++){
     int val = mylist[i];
     hole = i
     while(hole > 0 && val < mylist[hole-1]){
        mylist[hole] = mylist[hole-1];
        hole--;
     }
     mylist[hole] = val;
}}
```

Insertion Sort Loop Invariant

- What is true after the k-th iteration?
- All data at indices less than k+1 are sorted
  - $\forall i, i < k + 1: a[i] < a[i + 1]$
- Can we make a claim about data at k+1 and beyond?
  - No, it's not guaranteed to be smaller or larger than what is in the sorted list

```
void isort(vector<int> mylist)
{  for(int i=1; i < mylist.size()-1; i++){
     int val = mylist[i];
     hole = i
     while(hole > 0 && val < mylist[hole-1]){
        mylist[hole] = mylist[hole-1];
        hole--;
     }
     mylist[hole] = val;
}}
```



Algorithm Efficiency

DIVIDE AND CONQUER SORTING

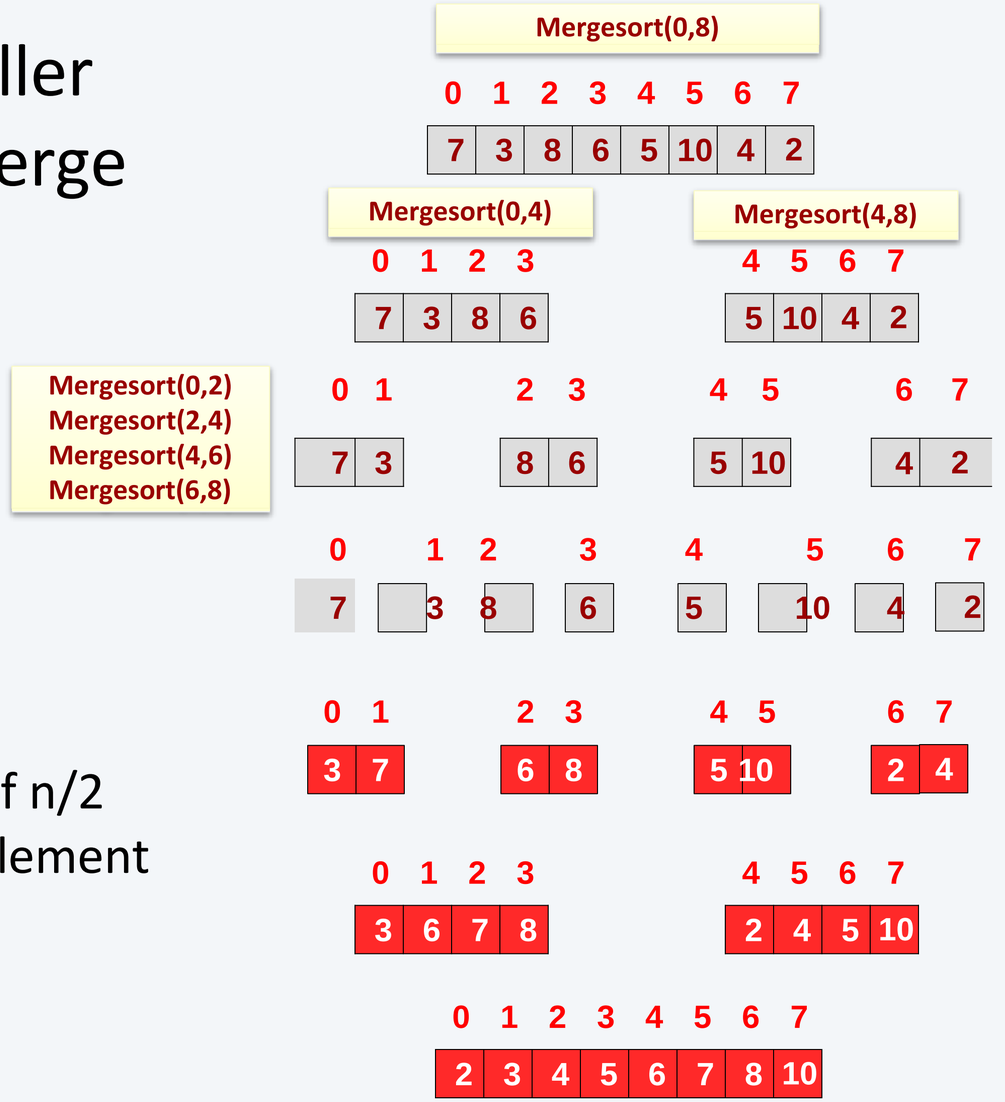
Divide & Conquer Strategy

- 3 Steps:
  - Divide
    - Split problem into smaller versions (usually partition the data somehow)
  - Recurse
    - Solve each of the smaller problems
  - Combine
    - Put solutions of smaller problems together to form larger solution

MERGESORT

MergeSort

- Break problem into smaller sorting problems and merge the results at the end
- Mergesort(0..n)
  - If list is size 1, return
  - Else
    - Mergesort(0..n/2 - 1)
    - Mergesort(n/2 .. n)
    - Combine each sorted list of n/2 elements into a sorted n-element list

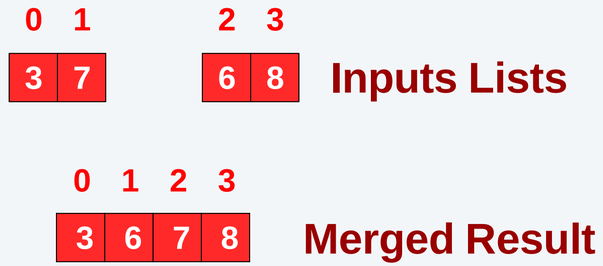


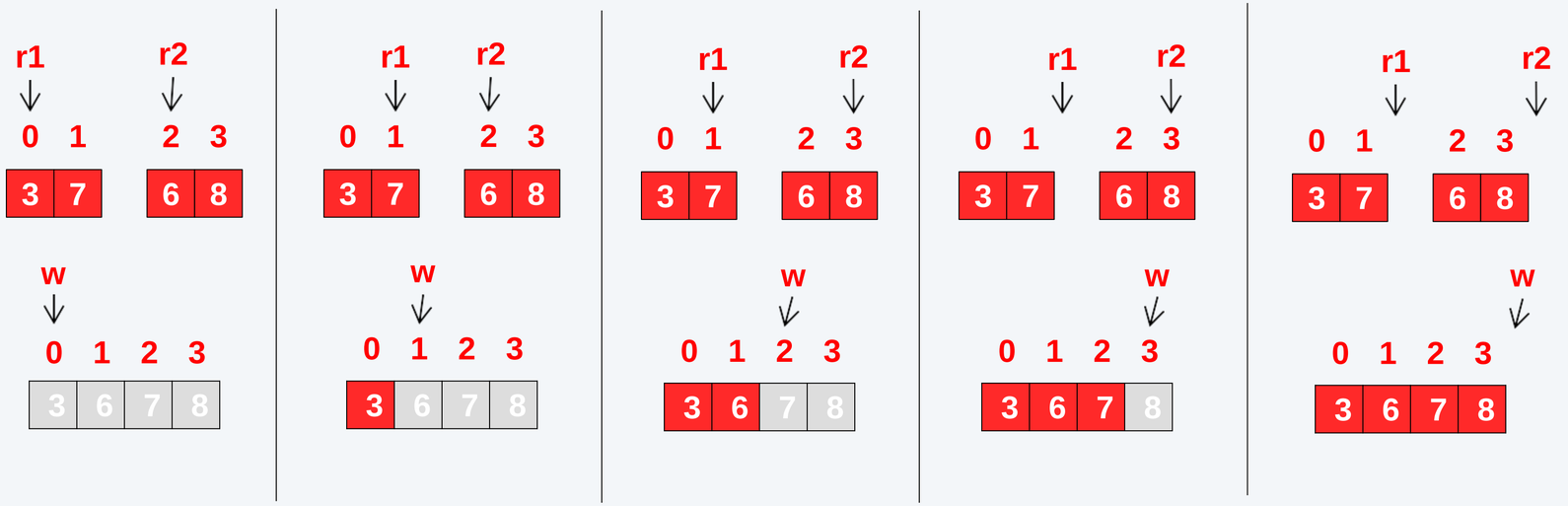
MergeSort

```
void mergesort(vector<int>& mylist,int l, int r)
{  if (l < r){
      int m = floor((l+r)/2);
      mergesort(mylist,l,m);
      mergesort(mylist,m+1,r);
      merge(mylist,l,r,m);
   }
}

void merge(vector<int>& mylist, int l, int r, int m){
   …
}
```

Merge Two Sorted Lists

- Consider the problem of merging two sorted lists into a new combined sorted list



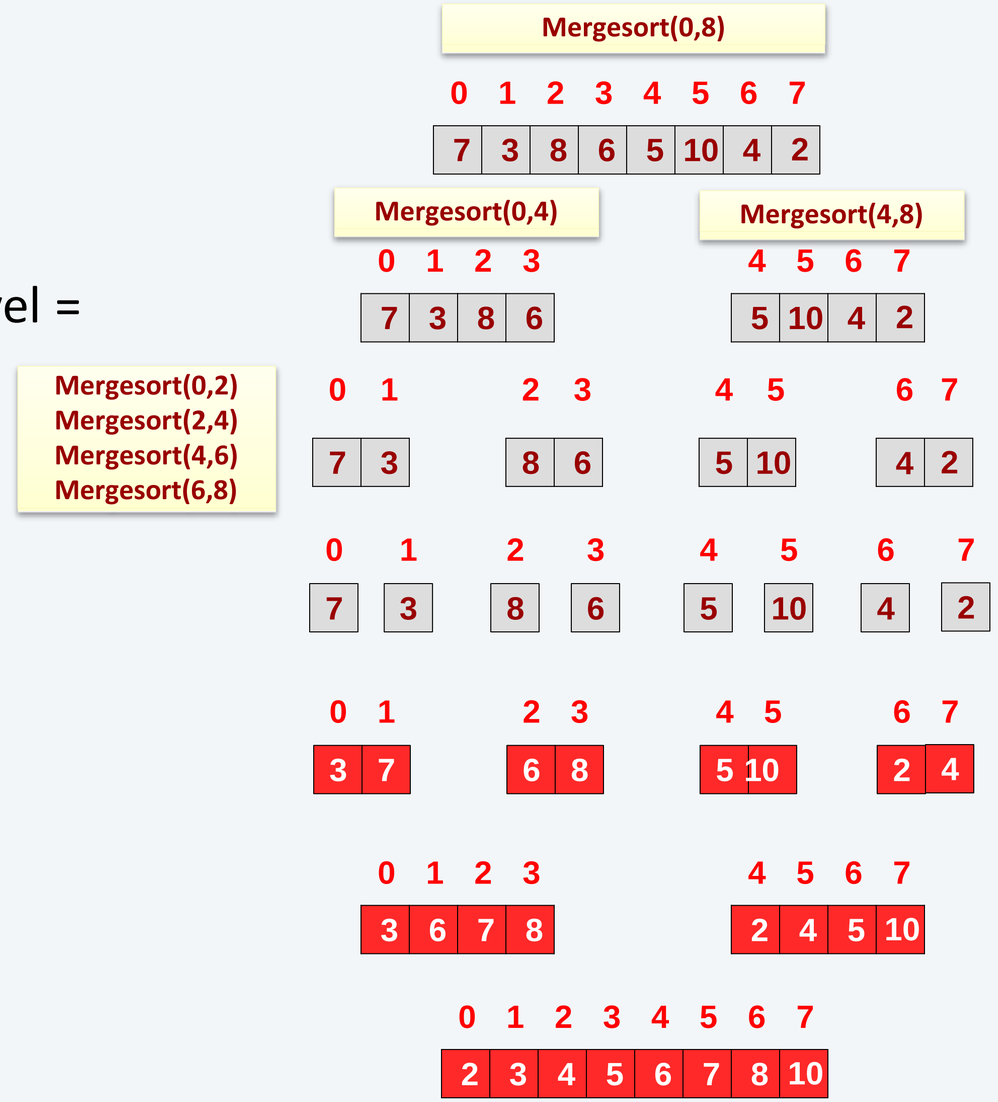


Merge of MergeSort

```
void merge(vector<int>& input,
                  int s1, int e1, int s2, int e2)
{
  vector<int> result;
   int start = s1;
   while(s1 < e1 && s2 < e2){
      if(input[s1] < input[s2]){
         result.push_back(input[s1++]);
      }
      else {
         result.push_back(input[s2++]);
      }
   }

  while(s1 < e1){
      result.push_back(input[s1++]);
   }

   while(s2 < e2){
      result.push_back(input[s2++]);
   }

   for (int i = 0; i < result.size() && start+i < e2; i++)
   {
      input[start+i] = result[i];
   }

     return;
}
```

MergeSort Runtime Analysis

- Run-time analysis
  - # of recursion levels =
    - $Log_2(n)$
  - Total operations to merge each level =
    - n operations total to merge two lists over all recursive calls at a particular level
- Mergesort = O(n * log (n))



MergeSort Recurrence

```
void mergesort(vector<int>& mylist,int l, int r)
{  if (l < r){
      int m = floor((l+r)/2);
      mergesort(mylist,l,m);
      mergesort(mylist,m+1,r);
      merge(mylist,l,r,m);
   }
}

void merge(vector<int>& mylist, int l, int r, int m){
   …
}
```

MergeSort Recursion Tree

- Let’s solve this using a recursion tree:

MergeSort Recursion Tree

- Let’s solve this using a recursion tree:

- To finish such proof with recurrence tree:

Unrolling the MergeSort Runtime Recurrence

- Let's prove this using the recurrence directly:

QUICKSORT

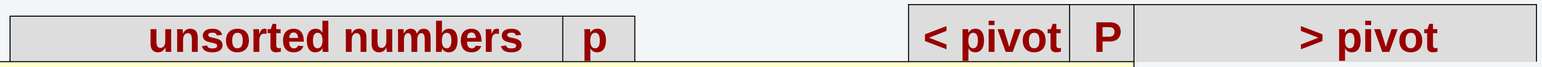
QuickSort

- Use the partition algorithm as the basis of a sort algorithm
- Partition on some number and the recursively call on both sides


```
// range is [start,end] where end is inclusive
void QuickSort(T a[], int start, int end)
{
   // base case - list has 1 or less items
   if(start < end){
   // pick last element as pivot
   / partition
   int loc = partition(a, start, end);
   // recurse on both sides
   QuickSort(mylist,start,loc-1);
   QuickSort(mylist,loc+1,end);
   }
   //base case returns
}
```

Partition of QuickSort

- Partition algorithm (arbitrarily) picks one number as the pivot and puts it into the correct location




```
int partition(T a[], int l, int r)
{
      int i= l;
      // set pivot
      T p = a [ r ] ;
     for ( int j = l ; j < r ; j++) {
           // moving pivot to correct position
           i f ( a [ j ] <= p ) {
                a . swap ( i , j ) ;
                i ++;
           }
      }
      a . swap ( i , r ) ;
     // return index where pivot is in correct position
      return i ;
}
```

Partition(mylist,0,5)

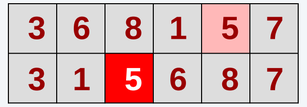
QuickSort Partition Trace Practice

Let’s trace partition on the array containing in this order:

vector = 7, 5, 3, 1, 4 The call to trace is partition(vector, 0, 4)

QuickSort Runtime Analysis

- Worst Case Complexity:
  - When pivot chosen ends up being min or max item
  - Runtime:
    - $T(n) = \Theta(n) + T(n-1)$



- Best Case Complexity:
  - Pivot point chosen ends up being the median item
  - Runtime:
    - Similar to MergeSort
    - $T(n) = 2T(n/2) + \Theta(n)$

QuickSort Runtime Analysis

- Worst Case Complexity:
  - When pivot chosen ends up being max or min of each list
  - $O(n^2)$
- Best Case Complexity:
  - Pivot point chosen ends up being the middle item
  - O(n*lg(n))
- Average Case Complexity: O(n*log(n))
  - Randomly choose a pivot

FUNCTORS

- A **functor** or **function object** is a class/struct that overloads the function call operator, operator()
- If the functor is used to compare other objects of a different class, it is called a comparator.
- Comparators can be passed to sorting algorithms in order to have a single templated sorting function.

Operator()

For most operators their number of arguments is implied

- operator+ takes an LHS and RHS
- operator-- takes no args

To overload operator() the number of arguments used may be arbitrary

```
class ObjA {
 public:
  ObjA();
  void action();
  void operator()() {
    cout << "I'm a functor!";
    cout << endl;
  }
  void operator()(int &x) {
    return ++x;
  }
};

int main()
{
  ObjA a;
  int y = 5;
  a();
  // prints "I'm a functor!"

  // This also makes sense !!
  a(y);
  // y is now 6
  return 0;
}
```

Purpose of Functors

Functors permit code to be generic as templates make type generic.

```
int count_if_neg (
   vector<int>::iterator first,
   vector<int>::iterator last)
{
  int ret = 0;
  for( ; first != last; ++first){
    if ( *first < 0 )
      ++ret;
  }
  return ret;
}
int count_if_even (
   vector<int>::iterator first,
   vector<int>::iterator last)
{
  int ret = 0;
  for( ; first != last; ++first){
    if ( *first % 2 == 0 )
      ++ret;
  }
  return ret;
}
```

Templates with Functors

Functors and templates can be used to make code generic.

```
struct isNeg {
 bool operator()(int x) { return x < 0; } };
struct isEven {
 bool operator()(int x) { return x % 2 == 0; } };


template <typename Comp>
int count_if (vector<int>::iterator first,
              vector<int>::iterator last,
              Comp c)
{ int ret = 0;
  for( ; first != last; ++first){
    if ( c(*first) )
      ++ret;
  }
  return ret;
}

int main()
{
  vector<int> v;   isNeg c1;   isEven c2;
  // fill data somehow
  int neg = count_if(v.begin(), v.end(), c1);
  int even = count_if(v.begin(), v.end(), c2);
  return 0;
}
```

Templates with Functors

- Define functor struct the operator()
- Declare struct and pass to function

```
struct NegCond {
  bool operator()(int val) { return val < 0; }
};

int main()
{ std::vector<int> myvec;

  // myvector: -5 -4 -3 -2 -1 0 1 2 3 4
  for (int i=-5; i<5; i++)
     myvec.push_back(i);
  NegCond c;
  int mycnt = count_if (myvec.begin(),
                        myvec.end(),
                        c);
  cout << "myvec contains " << mycnt;
  cout << " negative values." << endl;
  return 0;
}
```

std::count_if

Below is a modified count_if template function (from STL <algorithm>) that counts how many items in a container meet some condition

```
template <class InputIterator, class Cond>
int count_if (InputIterator first,
              InputIterator last,
              Cond pred)
{ int ret = 0;
  for( ; first != last; ++first){
    if ( pred( *first ) )
      ++ret;
  }
  return ret;
}
```

```
struct NegCond {
  bool operator()(int val)
   { return val < 0; }
};

int main()
{ std::vector<int> myv;

  // myvector: -5 -4 -3 ... 2 3 4
  for (int i=-5; i<5; i++)
     myvec.push_back(i);
  NegCond c;
  int mycnt =
     count_if(v.begin(), v.end(), c);
  cout << "myvec contains " << mycnt;
  cout << " negative values." << endl;
  return 0;
}
```

Functors for Maps and Sets

To use an object as a key for a map or set the class must have operator<

If class does not have operator<, use a comparator

```
class Pt {
 public:
  Pt(...);
  void action() { /* do stuff */ }
  int getX() { return x; }
  int getY() { return y; }
 private:
  int x, y;
};
```

```
int main()
{
  // I'd like to use Pt as a key
  // Can I?
  map<Pt, double> mymap;

  Pt p1(4,5);
  mymap[p1] = 6.7;
  return 0;
}
```

Map template takes as third template parameter a **comparator functor**

Functors are often used in C++ STL

References

- http://www.cprogramming.com/tutorial/functors-function-objects-in-c++.html

```
class Pt {
 public:
  Pt(...);
  void action() { /* do stuff */ }
  int getX() { return x; }
  int getY() { return y; }
 private:
  int x, y;
};
```

```
struct PtComparer
{
  bool operator()(const Pt& lhs,
                  const Pt& rhs)
  { return lhs.getX() < rhs.getX(); }
};

int main()
{ // Now we can use Pt as a key!!!!
  map<Pt, double, PtComparer> mymap;

  Pt a(4, 5), b(3, 7);
  mymap[a] = 6.7;   mymap[b] = 2.1;
  return 0;
}
```