Sorting

Insertion Sort, Merge Sort, & Quick Sort

Comparison Based Sorting, only operations are

< operator

assignment operator

STL Sorting

"The STL template function sort does not guarantee equal items retain their order
(if that is important use stable_sort instead of sort.)"

```
void sort( Iterator begin, Iterator end );
void sort( Iterator begin, Iterator end, Comparator cmp );
```

```
sort( v.begin( ), v.end( ) );
sort( v.begin( ), v.end( ), greater<int>( ) );
sort( v.begin( ), v.begin( ) + ( v.end( ) - v.begin( ) ) / 2 );
```

function template

std::sort

<algorithm>

```
default (1)   template <class RandomAccessIterator>
                void sort (RandomAccessIterator first, RandomAccessIterator last);
custom (2)    template <class RandomAccessIterator, class Compare>
                void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Sort elements in range

Sorts the elements in the range `[first,last)` into ascending order.

The elements are compared using `operator<` for the first version, and *comp* for the second.

Equivalent elements are not guaranteed to keep their original relative order (see `stable_sort`).

Parameters

`first, last`

Random-access iterators to the initial and final positions of the sequence to be sorted. The range used is `[first,last)`, which contains all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.
`RandomAccessIterator` shall point to a type for which `swap` is properly defined and which is both *move-constructible* and *move-assignable*.

`comp`

Binary function that accepts two elements in the range as arguments, and returns a value convertible to `bool`. The value returned indicates whether the element passed as first argument is considered to go before the second in the specific *strict weak ordering* it defines.
The function shall not modify any of its arguments.
This can either be a function pointer or a function object.

Return value

none

http://www.cplusplus.com/reference/algorithm/sort/

Why Sort?

Sorted data is easier to work with. Historically 25% of computer time was spent on sorting!

grouping

Finding which pair of items are closest to each other is easy if the items are sorted! O(n) time

Testing if all the items are unique is easy if sorted!

Finding which item occurs most frequently is much faster if the items are sorted! O(n)

preprocessing:

Searching through 100,000,000 items (under the assumption of each item is equally likely to be searched for) takes ~50,000,000 comparisons if the items are unsorted vs ~ 27 comparisons using binary search. O(n) vs O(log(n))!

Finding the smallest, largest, K'th largest is O(1) if sorted.

subroutine in other algorithms

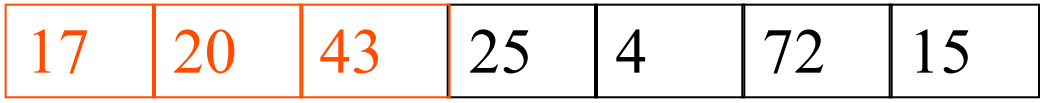
Optimal compression of text. Finding an optimal Huffman encoding starts by knowing the character frequency and sorting the characters by their frequencies.

Kruskal's algorithm for finding an optimal spanning tree starts by sorting the edge weights.

How can we sort the items?

A First Sort: Insertion Sort

Insertion Sort

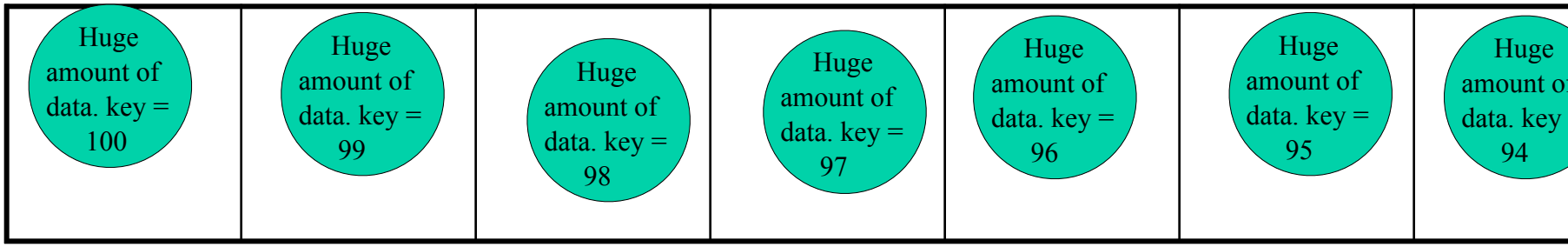


a[0..p-1] sorted rest of array

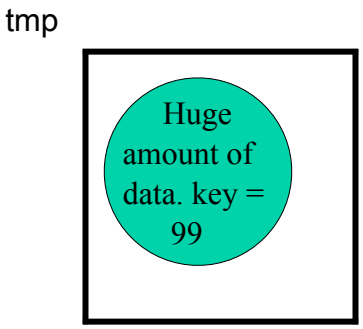
Insert element a[p] in a[0..p-1], sliding larger elements over to make room:

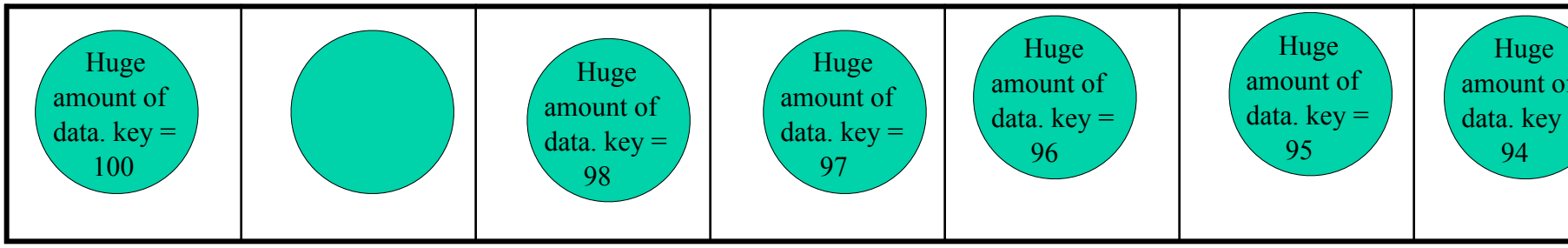
a[0..p] sorted rest of array

```
// insertionSort: sort items in array a
// Comparable: must have copy constructor, operator=,
//   and operator<
template <class Comparable>
void insertionSort( vector<Comparable> & a )
{
    int j;

    for( int p = 1; p < a.size( ); p++ )
    {
        Comparable tmp = std::move(a[ p ]);
        for( j = p; j > 0 && tmp < a[ j - 1 ]; j-- )
            a[ j ] = std::move(a[ j - 1 ]);
        a[ j ] = std::move(tmp);
    }
}
```

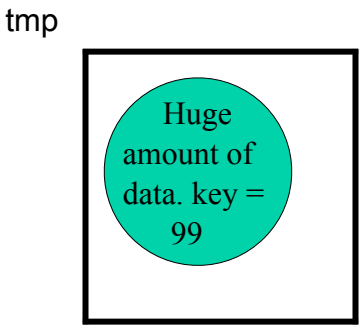


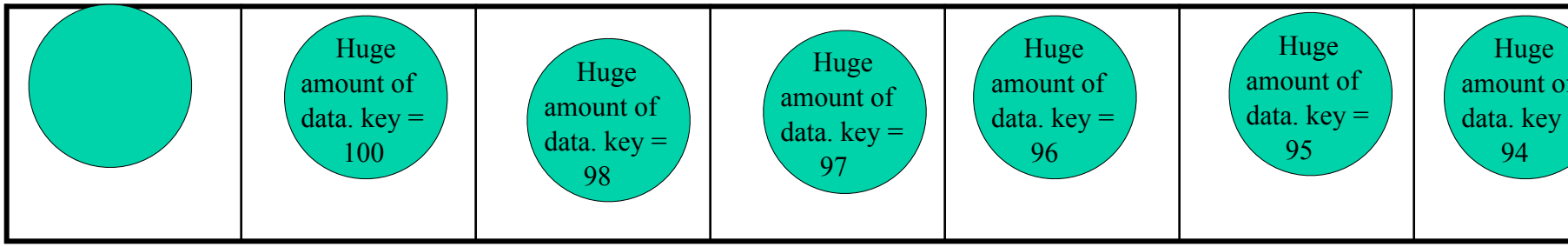
```
// insertionSort: sort items in array a
// Comparable: must have copy constructor, operator=,
//   and operator<
template <class Comparable>
void insertionSort( vector<Comparable> & a )
{
    int j;

    for( int p = 1; p < a.size( ); p++ )
    {
        Comparable tmp = std::move(a[ p ]);
        for( j = p; j > 0 && tmp < a[ j - 1 ]; j-- )
            a[ j ] = std::move(a[ j - 1 ]);
        a[ j ] = std::move(tmp);
    }
}
```





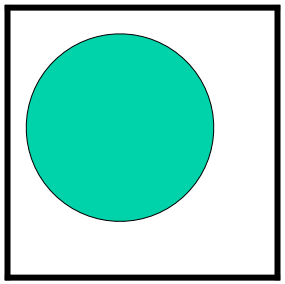
```
// insertionSort: sort items in array a
// Comparable: must have copy constructor, operator=,
//   and operator<
template <class Comparable>
void insertionSort( vector<Comparable> & a )
{
    int j;

    for( int p = 1; p < a.size( ); p++ )
    {
        Comparable tmp = std::move(a[ p ]);
        for( j = p; j > 0 && tmp < a[ j - 1 ]; j-- )
            a[ j ] = std::move(a[ j - 1 ]);
        a[ j ] = std::move(tmp);
    }
}
```

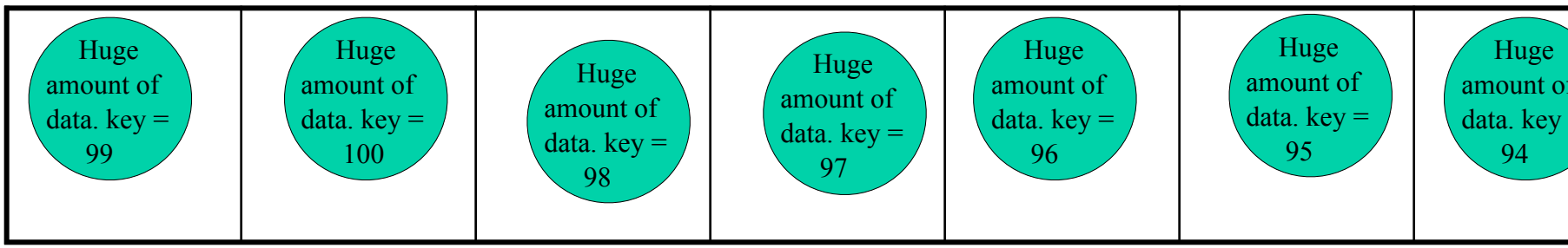




```
// insertionSort: sort items in array a
// Comparable: must have copy constructor, operator=,
//   and operator<
template <class Comparable>
void insertionSort( vector<Comparable> & a )
{
    int j;

    for( int p = 1; p < a.size( ); p++ )
    {
        Comparable tmp = std::move(a[ p ]);
        for( j = p; j > 0 && tmp < a[ j - 1 ]; j-- )
            a[ j ] = std::move(a[ j - 1 ]);
        a[ j ] = std::move(tmp);
    }
}
```

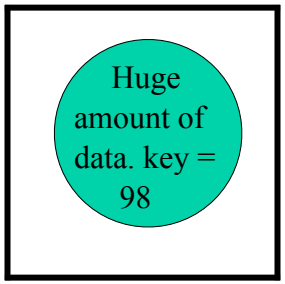
tmp

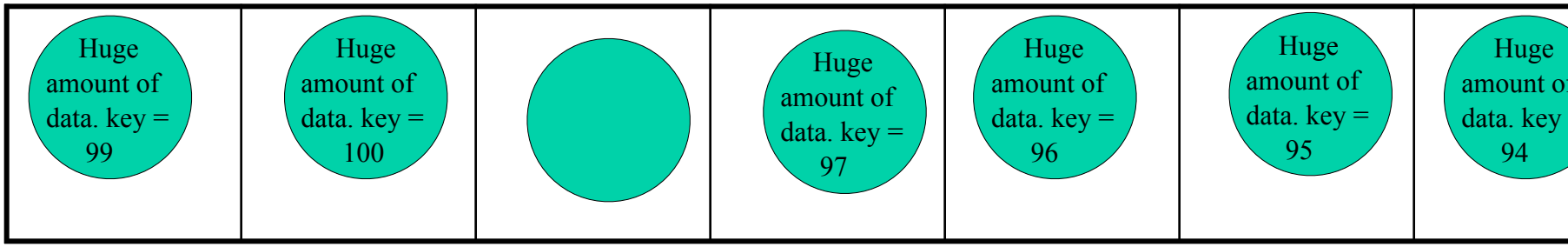




```
// insertionSort: sort items in array a
// Comparable: must have copy constructor, operator=,
//   and operator<
template <class Comparable>
void insertionSort( vector<Comparable> & a )
{
    int j;

    for( int p = 1; p < a.size( ); p++ )
    {
        Comparable tmp = std::move(a[ p ]);
        for( j = p; j > 0 && tmp < a[ j - 1 ]; j-- )
            a[ j ] = std::move(a[ j - 1 ]);
        a[ j ] = std::move(tmp);
    }
}
```

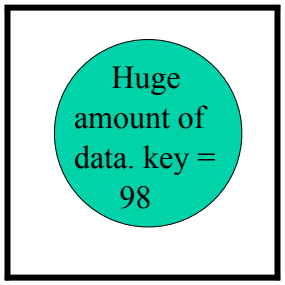
tmp

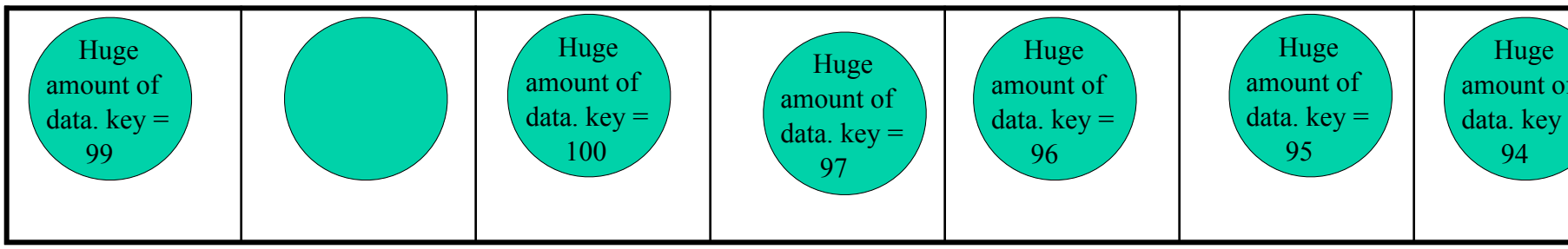




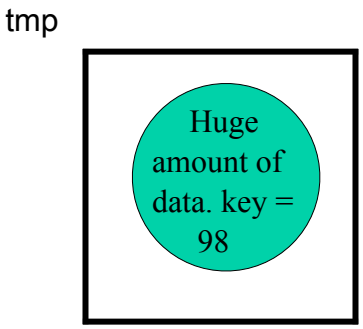
```
// insertionSort: sort items in array a
// Comparable: must have copy constructor, operator=,
//   and operator<
template <class Comparable>
void insertionSort( vector<Comparable> & a )
{
    int j;

    for( int p = 1; p < a.size( ); p++ )
    {
        Comparable tmp = std::move(a[ p ]);
        for( j = p; j > 0 && tmp < a[ j - 1 ]; j-- )
            a[ j ] = std::move(a[ j - 1 ]);
        a[ j ] = std::move(tmp);
    }
}
```

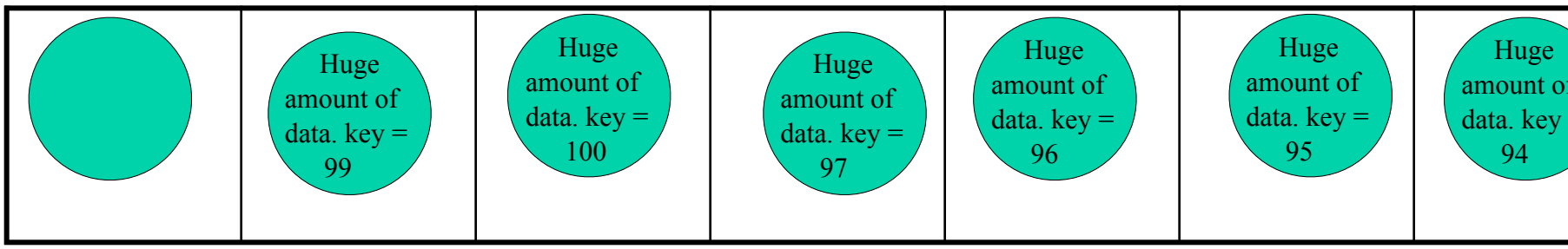
tmp





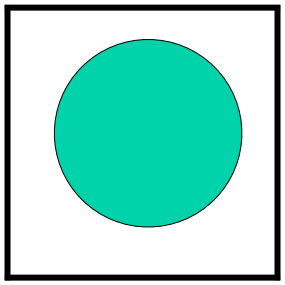
```
// insertionSort: sort items in array a
// Comparable: must have copy constructor, operator=,
//   and operator<
template <class Comparable>
void insertionSort( vector<Comparable> & a )
{
    int j;

    for( int p = 1; p < a.size( ); p++ )
    {
        Comparable tmp = std::move(a[ p ]);
        for( j = p; j > 0 && tmp < a[ j - 1 ]; j-- )
            a[ j ] = std::move(a[ j - 1 ]);
        a[ j ] = std::move(tmp);
    }
}
```

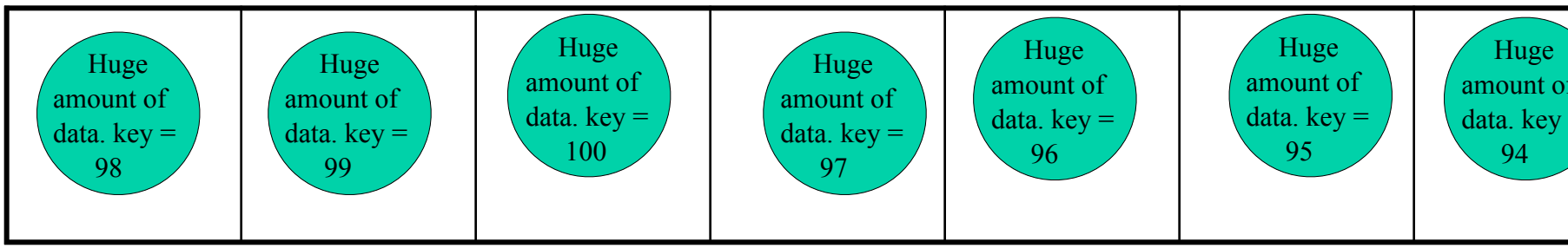




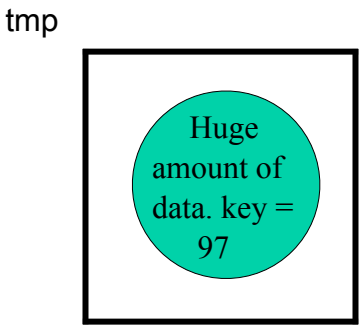
```
// insertionSort: sort items in array a
// Comparable: must have copy constructor, operator=,
//   and operator<
template <class Comparable>
void insertionSort( vector<Comparable> & a )
{
    int j;

    for( int p = 1; p < a.size( ); p++ )
    {
        Comparable tmp = std::move(a[ p ]);
        for( j = p; j > 0 && tmp < a[ j - 1 ]; j-- )
            a[ j ] = std::move(a[ j - 1 ]);
        a[ j ] = std::move(tmp);
    }
}
```

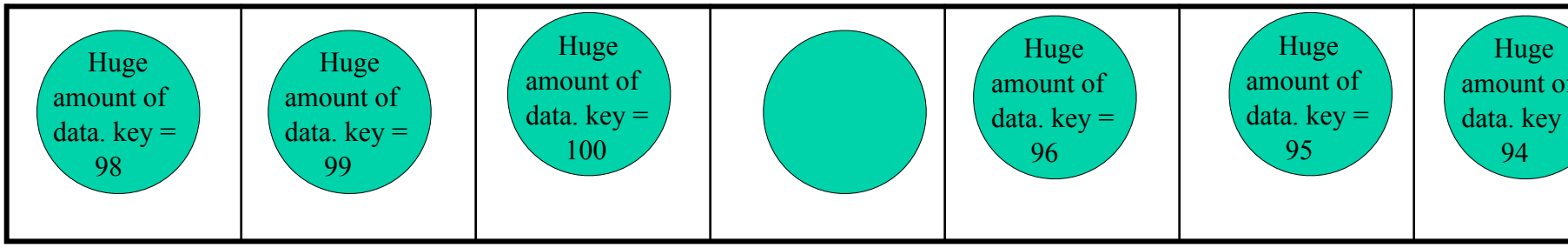
tmp



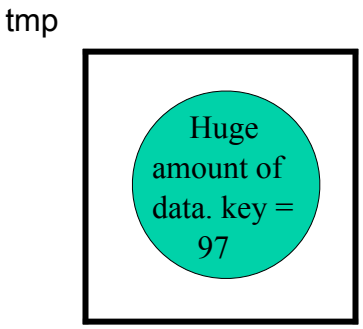


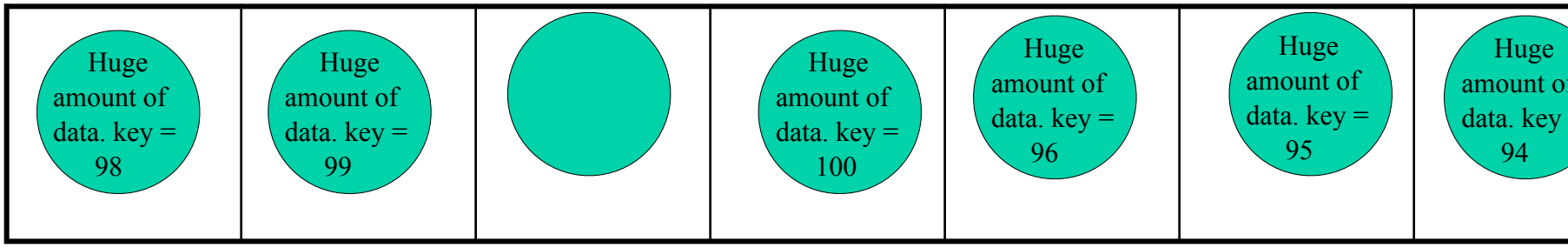
```
// insertionSort: sort items in array a
// Comparable: must have copy constructor, operator=,
//   and operator<
template <class Comparable>
void insertionSort( vector<Comparable> & a )
{
    int j;

    for( int p = 1; p < a.size( ); p++ )
    {
        Comparable tmp = std::move(a[ p ]);
        for( j = p; j > 0 && tmp < a[ j - 1 ]; j-- )
            a[ j ] = std::move(a[ j - 1 ]);
        a[ j ] = std::move(tmp);
    }
}
```



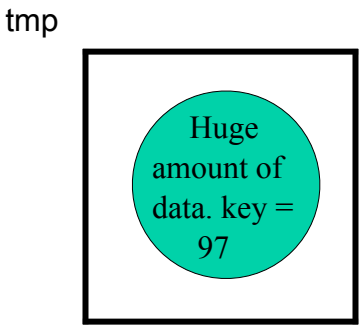


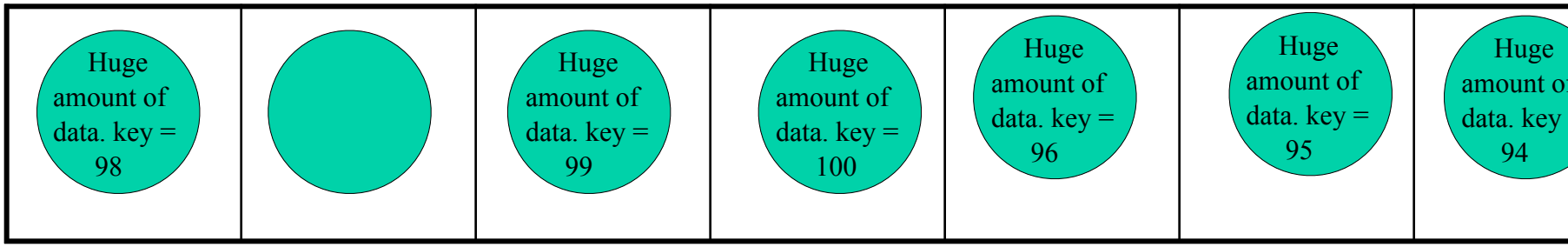
```
// insertionSort: sort items in array a
// Comparable: must have copy constructor, operator=,
//   and operator<
template <class Comparable>
void insertionSort( vector<Comparable> & a )
{
    int j;

    for( int p = 1; p < a.size( ); p++ )
    {
        Comparable tmp = std::move(a[ p ]);
        for( j = p; j > 0 && tmp < a[ j - 1 ]; j-- )
            a[ j ] = std::move(a[ j - 1 ]);
        a[ j ] = std::move(tmp);
    }
}
```



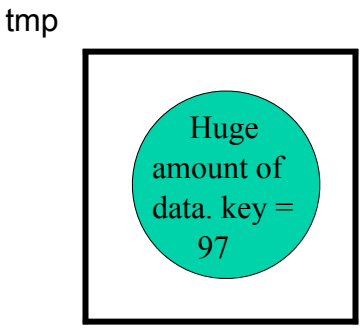


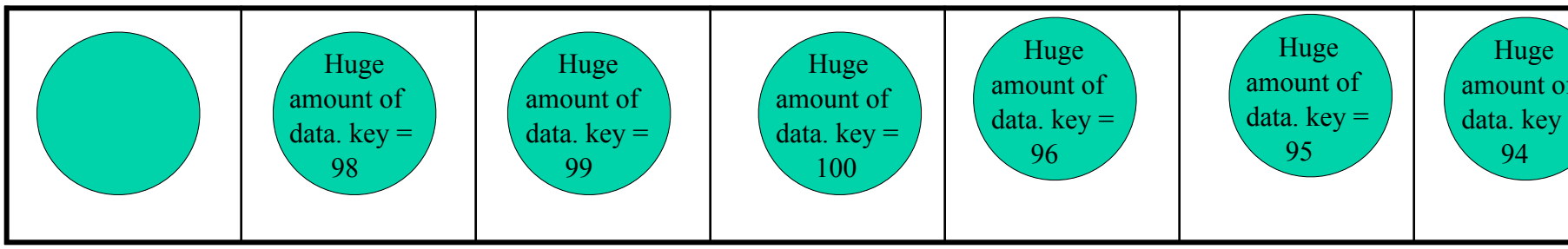
```
// insertionSort: sort items in array a
// Comparable: must have copy constructor, operator=,
//   and operator<
template <class Comparable>
void insertionSort( vector<Comparable> & a )
{
    int j;

    for( int p = 1; p < a.size( ); p++ )
    {
        Comparable tmp = std::move(a[ p ]);
        for( j = p; j > 0 && tmp < a[ j - 1 ]; j-- )
            a[ j ] = std::move(a[ j - 1 ]);
        a[ j ] = std::move(tmp);
    }
}
```





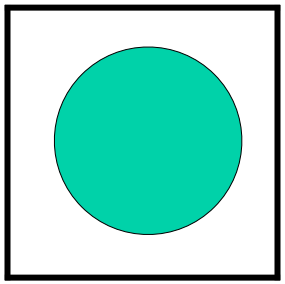
```
// insertionSort: sort items in array a
// Comparable: must have copy constructor, operator=,
//   and operator<
template <class Comparable>
void insertionSort( vector<Comparable> & a )
{
    int j;

    for( int p = 1; p < a.size( ); p++ )
    {
        Comparable tmp = std::move(a[ p ]);
        for( j = p; j > 0 && tmp < a[ j - 1 ]; j-- )
            a[ j ] = std::move(a[ j - 1 ]);
        a[ j ] = std::move(tmp);
    }
}
```

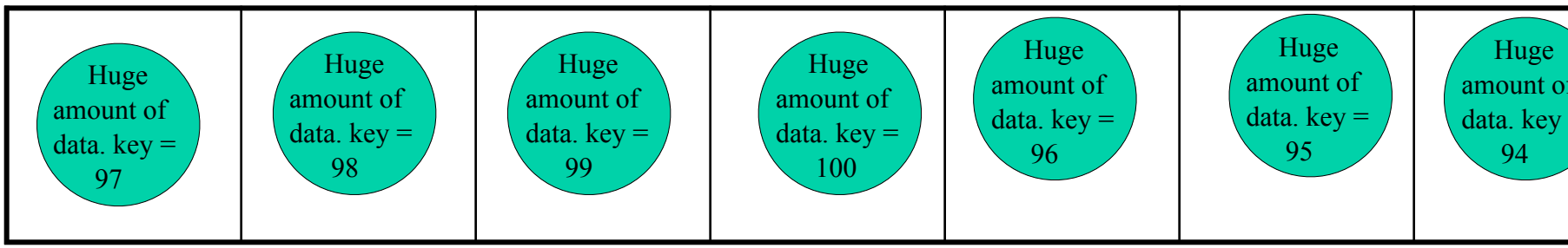




```
// insertionSort: sort items in array a
// Comparable: must have copy constructor, operator=,
//   and operator<
template <class Comparable>
void insertionSort( vector<Comparable> & a )
{
    int j;

    for( int p = 1; p < a.size( ); p++ )
    {
        Comparable tmp = std::move(a[ p ]);
        for( j = p; j > 0 && tmp < a[ j - 1 ]; j-- )
            a[ j ] = std::move(a[ j - 1 ]);
        a[ j ] = std::move(tmp);
    }
}
```

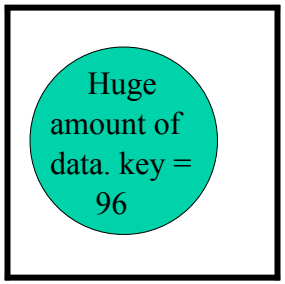
tmp

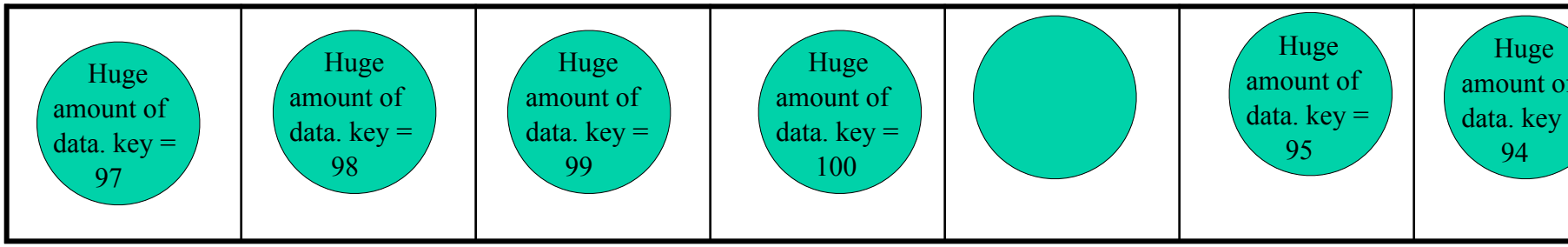




```
// insertionSort: sort items in array a
// Comparable: must have copy constructor, operator=,
//   and operator<
template <class Comparable>
void insertionSort( vector<Comparable> & a )
{
    int j;

    for( int p = 1; p < a.size( ); p++ )
    {
        Comparable tmp = std::move(a[ p ]);
        for( j = p; j > 0 && tmp < a[ j - 1 ]; j-- )
            a[ j ] = std::move(a[ j - 1 ]);
        a[ j ] = std::move(tmp);
    }
}
```

tmp





Running time?

STL Style Insertion Sort

- Range of items to sort [begin, end)
- Array access using iterators
- Use auto to deduce type
- Use a functor to compare items

17	20	25	43	4	72	15

```
template <class Comparable>
void insertionSort( vector<Comparable> & a )
{
    int j;

    for( int p = 1; p < a.size( ); p++ )
    {
        Comparable tmp = std::move(a[ p ]);
        for( j = p; j > 0 && tmp < a[ j - 1 ]; j-- )
            a[ j ] = std::move(a[ j - 1 ]);

        a[ j ] = std::move(tmp);
    }
}
```

```
template <typename Iterator, typename Comparator>
void insertionSort( Iterator begin, Iterator end, Comparator lessThan )
{
    if( begin == end )
        return;

    Iterator j;

    for( Iterator p = begin+1; p != end; ++p )
    {
        auto tmp =  std::move(*p) ;
        for( j = p; j != begin && lessThan( tmp, *( j-1 ) ); --j )
            *j = std::move(*(j-1));

        *j = std::move(tmp);
    }
}
```

Lets do it again! This time with two parameters and <

void insertionSort(Iterator begin, Iterator end);

Using:
void insertionSort(Iterator begin, Iterator end, Comparator cmp);

"In the C++ programming language, **decltype** is an operator for querying the type of an expression. It was introduced in the current version of the C++ standard, C++11. Its primary intended use is in generic programming, where it is often difficult, or even impossible, to express types that depend on template parameters."
http://en.wikipedia.org/wiki/Decltype

An STL Functor

```
template <class Object>
class less
{ public:
    bool operator()(const Object& lhs, const Object& rhs) const
      {return lhs < rhs;}
};

/*
* The two-parameter version calls the three-parameter version,
* using C++11 decltype
*/
template <typename Iterator>
void insertionSort( const Iterator & begin, const Iterator & end )
{
   insertionSort( begin, end, less< decltype(*begin)>( ) );
}
```

Operator yields the declared type of the expression

From Weiss, Mark A. (2013-06-11). Data Structures and Algorithm Analysis in C++ (4th Edition)

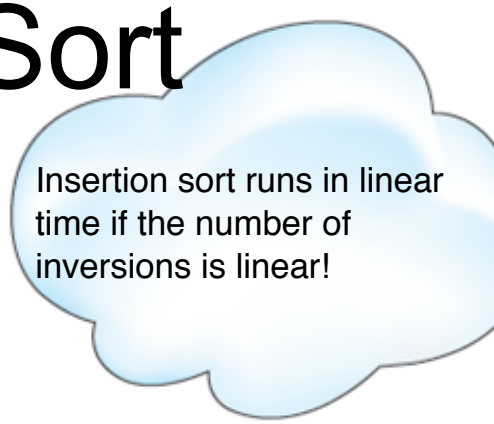
Inversions

- an inversion is a pair of elements out of order
- e.g. [8, 5, 9, 2, 6, 3] has 10 inversions:
  (8,5), (8,2), (8,6), (8,3), (5,2), (5,3),
  (9,2), (9,6), (9,3), (6,3)
- # of inversions is the number of times we move an item
- in our analysis we will assume there are no duplicate items

Average Case

- Given an array {8, 5, 9, 2, 6, 3} and an array in reverse order {3, 6, 2, 9, 5, 8}
- For every pair of numbers in an array, (x,y) it is an inversion in exactly one of the arrays. E.g. (8, 5) in an inversion in the first array and not the second.
- If we count the number of inversions in both arrays it equals the number of pairs, n(n-1)/2.
- An average array there for has n(n-1)/4

Analysis of Insertion Sort O(I + n)



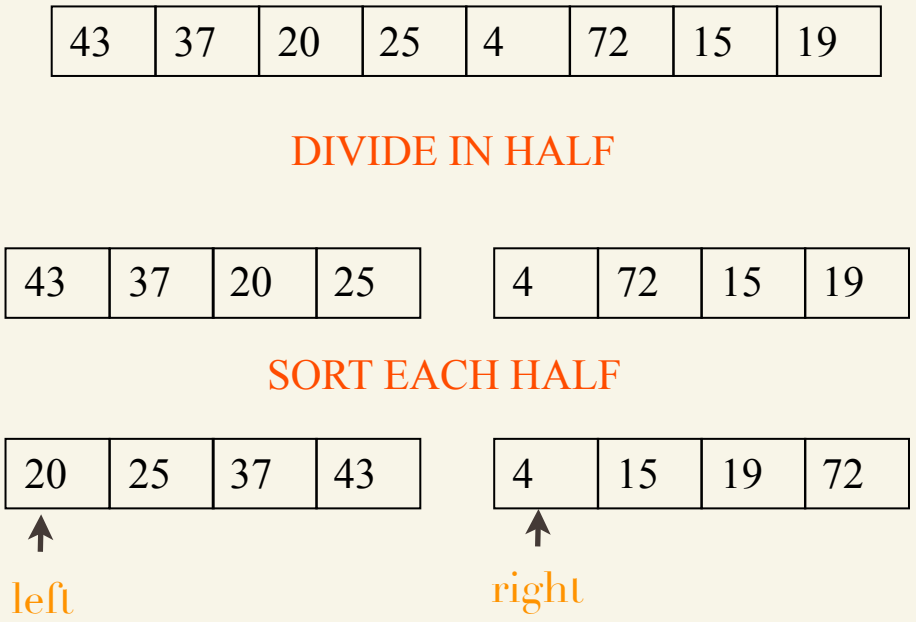
Number of inversions:

- Worst case:

  $1+ 2 + ... + n -1 = O(n^2)$
- Average case: similar ... $O(n^2)$
- Best case (Array is already sorted)
  - inner loop iterates 0 times for each p (1 test of inner for loop)
  - 0 inversions so running time is O(n) [!]
- Interesting special case: array is almost sorted ... similar to best case

Merge Sort
Divide and Conquer

Merge Sort


43 37 20 25 4 72 15 19
DIVIDE IN HALF
43 37 20 25
4 72 15 19
SORT EACH HALF
20 25 37 43
4 15 19 72
left
right
MERGE
4 15 19 20 25 37 43 72

Check out:

https://en.wikipedia.org/wiki/Merge_sort#/media/File:Merge-sort-example-300px.gif

http://www.sorting-algorithms.com/

https://www.youtube.com/user/AlgoRythmics

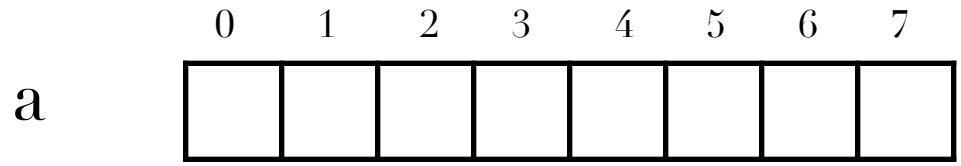
https://www.youtube.com/watch?v=kPRA0W1kECg

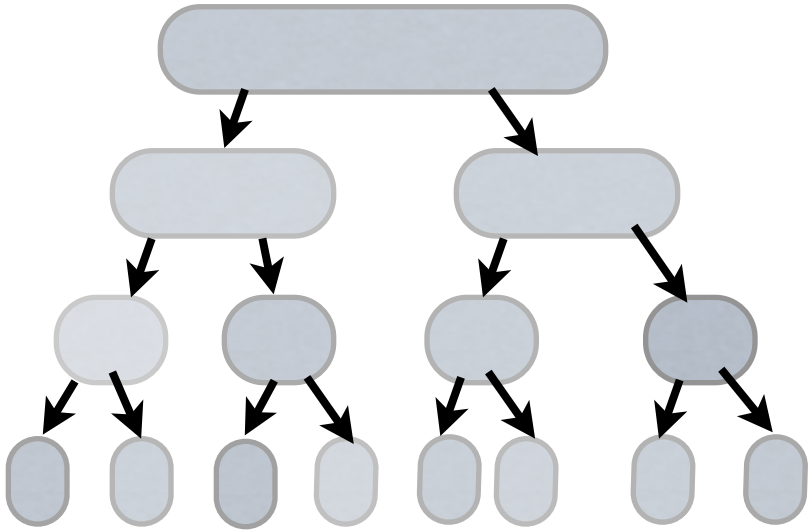
http://sortvis.org/

Merge Sort

- Divide array into two (almost) equal pieces
- Conquer:
  - sort each half recursively
  - merge the sorted arrays
- code
- Analysis:
  - log n stages, each of which is linear
  - O(n log n)

- Divide array into two (almost) equal pieces
- Conquer:
  - sort each half recursively
  - merge the sorted arrays



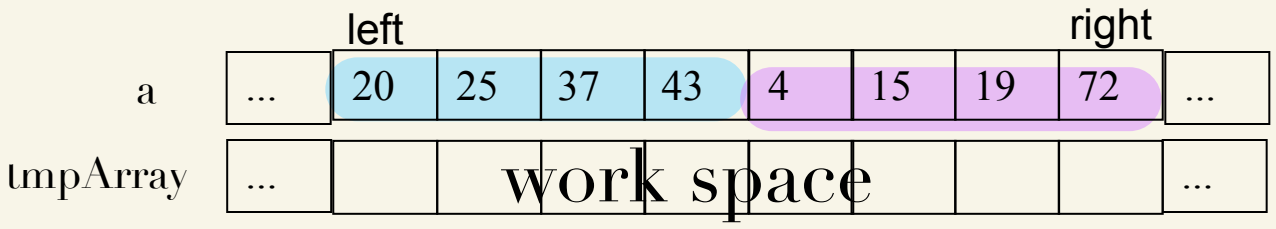


The Driver

```
// Mergesort algorithm (driver).
template<class Comparable>
void mergeSort( vector<Comparable> & a )
{
    vector<Comparable> tmpArray( a.size( ) );

    mergeSort( a, tmpArray, 0, a.size( ) - 1 );
}
```

```
void main()
{
    ...
    mergeSort(a);
    ...
}
```

```
template <class Comparable>
void mergeSort( vector<Comparable> & a,
        vector<Comparable> & tmpArray, int left, int right )
{
    if( left < right )
    {
      int center = ( left + right ) / 2;
      mergeSort( a, tmpArray, left, center );
      mergeSort( a, tmpArray, center + 1, right );
      myMerge( a, tmpArray, left, center + 1, right );

    }
}
```



Merge

- Given two sorted sub arrays, combine them into a single sorted array
  - Use auxiliary array
  - successively compare smallest elements
  - when one runs out, copy the rest of the other
- code for merge
- scans each element once: O(n) time
- also uses O(n) space

```
template <class Comparable>
void myMerge( vector<Comparable> & a, vector<Comparable>& tmpArray,
int leftPos, int rightPos, int rightEnd )
{
    int leftEnd = rightPos - 1;
    int tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;

// Main loop
   while( leftPos <= leftEnd && rightPos <= rightEnd )
      if( a[ leftPos ] <= a[ rightPos ] )
         tmpArray[ tmpPos++ ] = std::move(a[ leftPos++ ]);
      else
         tmpArray[ tmpPos++ ] = std::move(a[ rightPos++ ]);

    while( leftPos <= leftEnd )    // Copy rest of first half
       tmpArray[ tmpPos++ ] = std::move(a[ leftPos++ ]);

   while( rightPos <= rightEnd )  // Copy rest of right half
      tmpArray[ tmpPos++ ] =std::move(a[ rightPos++ ]);

     // Copy tmpArray back
   for( int i = 0; i < numElements; i++, - -rightEnd )
      a[ rightEnd ] = std::move(tmpArray[ rightEnd ];)
}
```

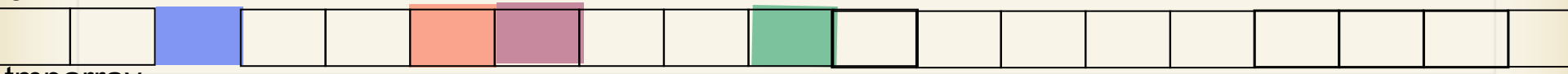
a

		15	25	37	43	4	12	18	19									

tmparray

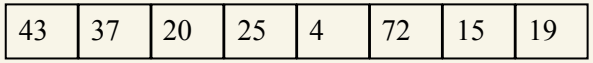

```
template <class Comparable>
void myMerge( vector<Comparable> & a, vector<Comparable>& tmpArray,
int leftPos, int rightPos, int rightEnd )
{
    int leftEnd = rightPos - 1;
    int tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;

// Main loop
   while( leftPos <= leftEnd && rightPos <= rightEnd )
      if( a[ leftPos ] <= a[ rightPos ] )
         tmpArray[ tmpPos++ ] = std::move(a[ leftPos++ ]);
      else
         tmpArray[ tmpPos++ ] = std::move(a[ rightPos++ ]);

    while( leftPos <= leftEnd )    // Copy rest of first half
       tmpArray[ tmpPos++ ] = std::move(a[ leftPos++ ]);

   while( rightPos <= rightEnd )  // Copy rest of right half
      tmpArray[ tmpPos++ ] = std::(move(a[ rightPos++ ]);

     // Copy tmpArray back
   for( int i = 0; i < numElements; i++, - -rightEnd )
      a[ rightEnd ] = std::move(tmpArray[ rightEnd ]);
}
```

a

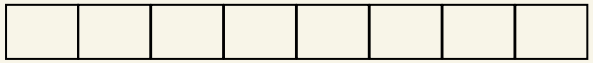


tmparray

a



tmpArray



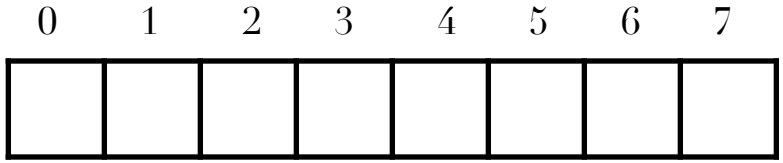
mergeSort(a, tmpArray, 0, 0)

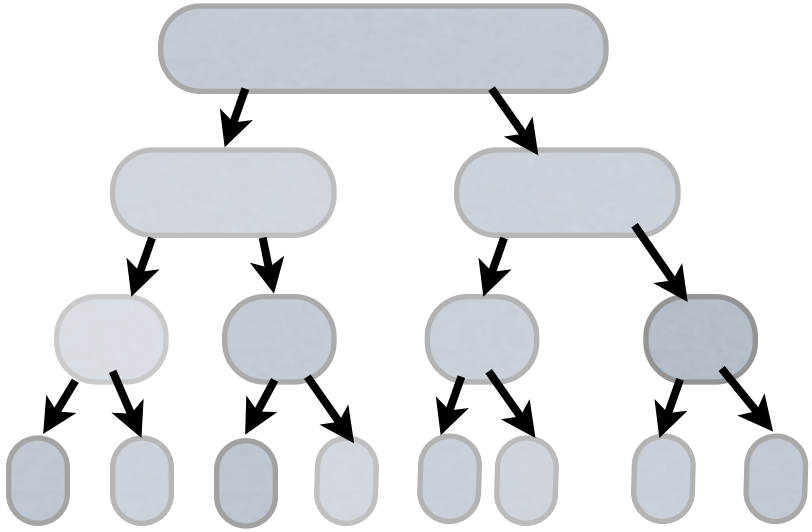
mergeSort(a, tmpArray, 0, 1)

mergeSort(a, tmpArray, 0, 3)

mergeSort(a, tmpArray, 0, 7)

a





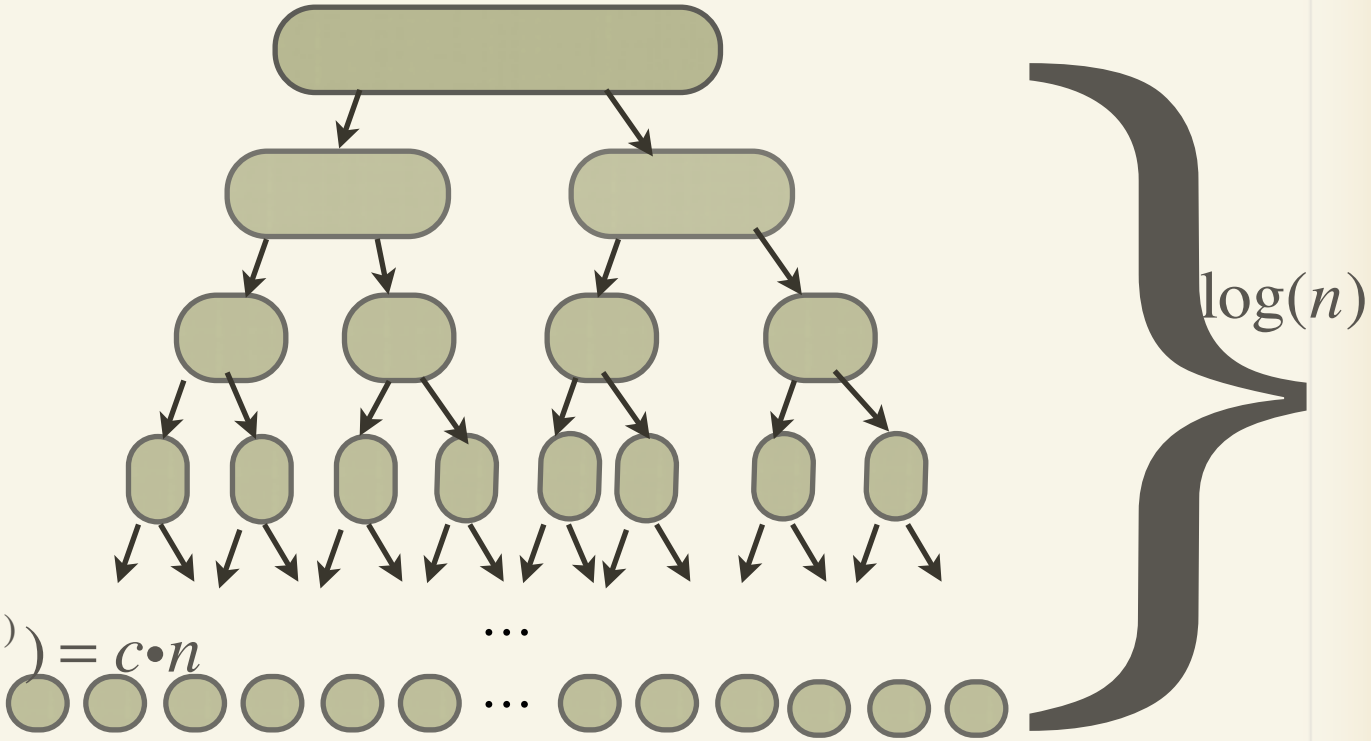
$c \bullet n$

$2(c \bullet n / 2) = c \bullet n$

$4(c \bullet n / 4) = c \bullet n$

$8(c \bullet n / 8) = c \bullet n$

$2^{\log(n)}(c \bullet n / 2^{\log(n)}) = c \bullet n$



$O(n \bullet \log(n))$ time

Homework

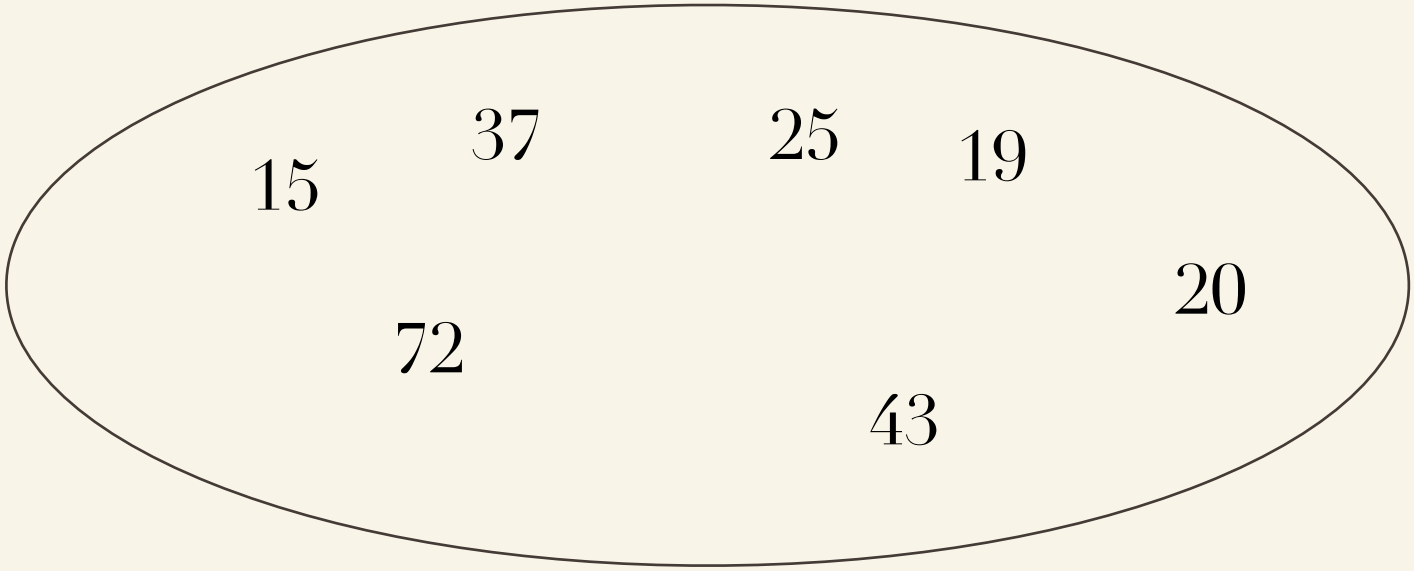
```
template<class InputIterator, class OutputIterator>
  OutputIterator copy (InputIterator first, InputIterator last, OutputIterator result)
{
  while (first!=last) {
    *result = *first;
    ++result; ++first;
  }
  return result;
}
```

```
template <class InputIterator1, class InputIterator2, class OutputIterator>
  OutputIterator merge (InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, InputIterator2 last2,
                        OutputIterator result)
{
  while (true) {
    if (first1==last1) return std::copy(first2,last2,result);
    if (first2==last2) return std::copy(first1,last1,result);
    *result++ = (*first2<*first1)? *first2++ : *first1++;
  }
}
```

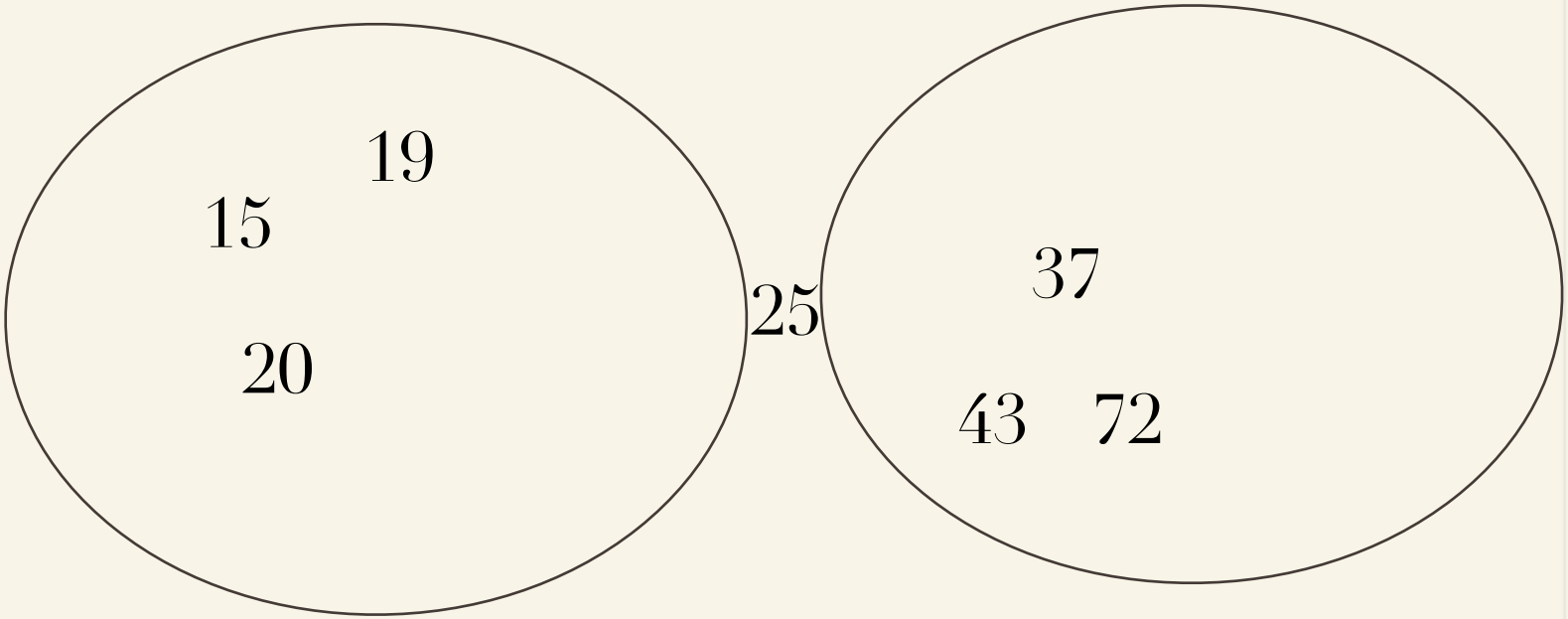
From: http://www.cplusplus.com/reference/algorithm/

Quick Sort
Divide and Conquer

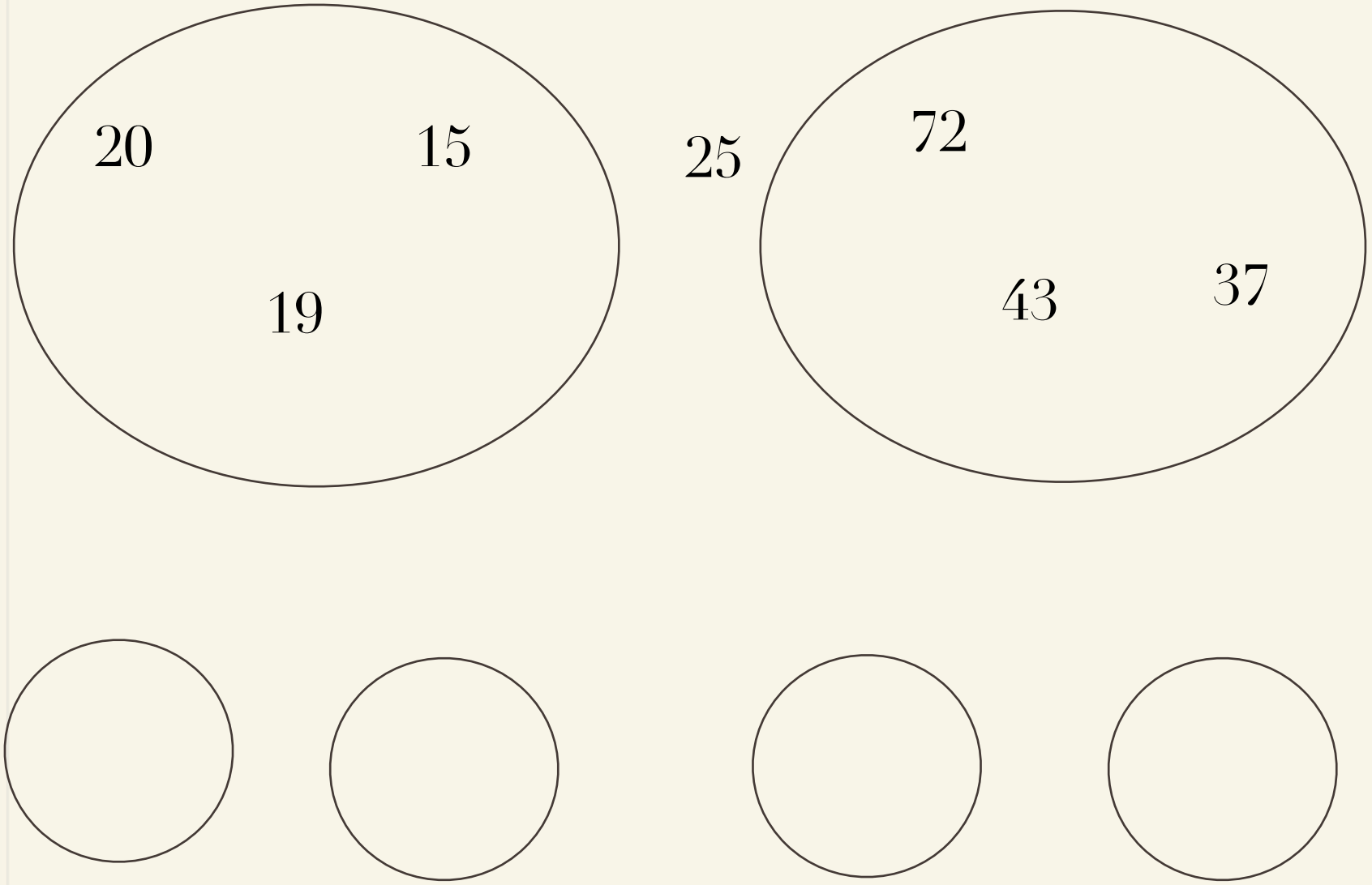
Reduce the original problem into subproblems that are easier to solve.



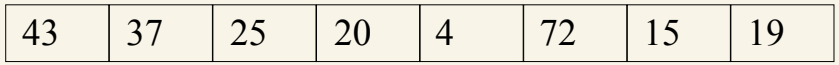
Solve the subproblems, and the combine to solve the original problem



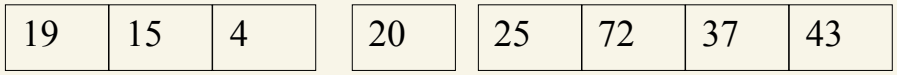
The subproblems are solved recursively



QuickSort



SELECT PIVOT; PARTITION



<= pivot pivot >= pivot

SORT EACH PIECE (Recursively)

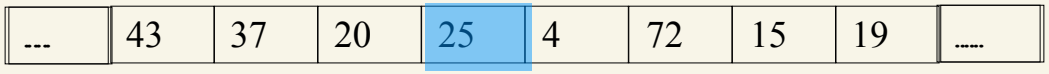
done!

Quick Sort

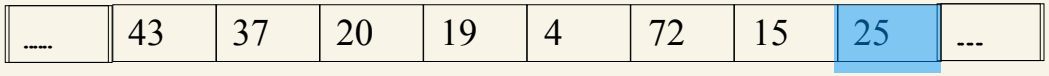
- Partition array into two pieces:
  - (elements <= pivot) , (elements >= pivot)
- Conquer:
  - sort each half recursively
- Analysis:
  - partition step can be done in linear time
  - Running time depends on choice of pivots at each stage
  - Equal sized piece → situation similar to merge sort
  - Very unequal sizes → poor performance

Implementation details on how we partition the items

First we select a pivot

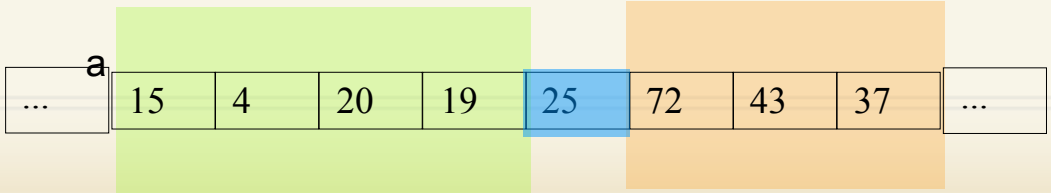


In our implementation we move the pivot to the the last position



Then we partition the items using the pivot

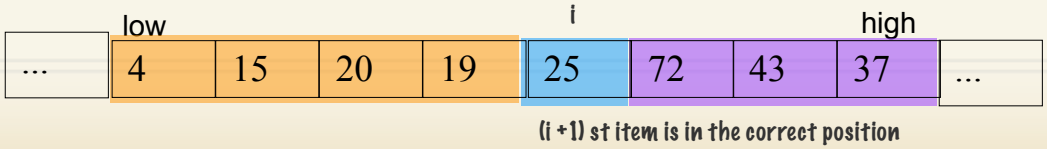
```
int i, j;
for( i = low, j = high - 1; ; )
{
   while ( a[i ] < pivot )  ++i;
  while(  j > i && pivot < a[j ] ) --j;
   if( i < j )
        std::swap( a[ i++ ], a[ j--] );
   else
        break;
}

swap( a[ i ], a[ high  ] );
```



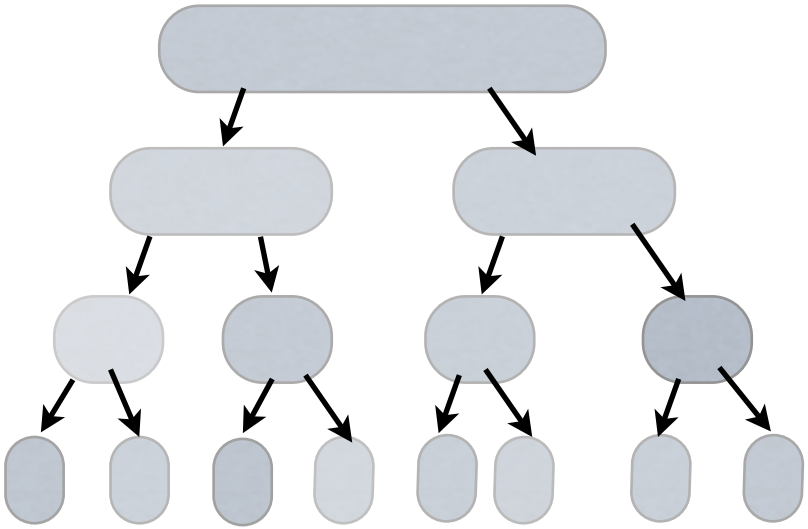
The Driver

```
/* driver for quicksort */
void quicksort(vector<int> &a)
{
   quicksort(a, 0, a.size()-1);
}


void main()
{
    ...
    quicksort(a);
    ...
}
```

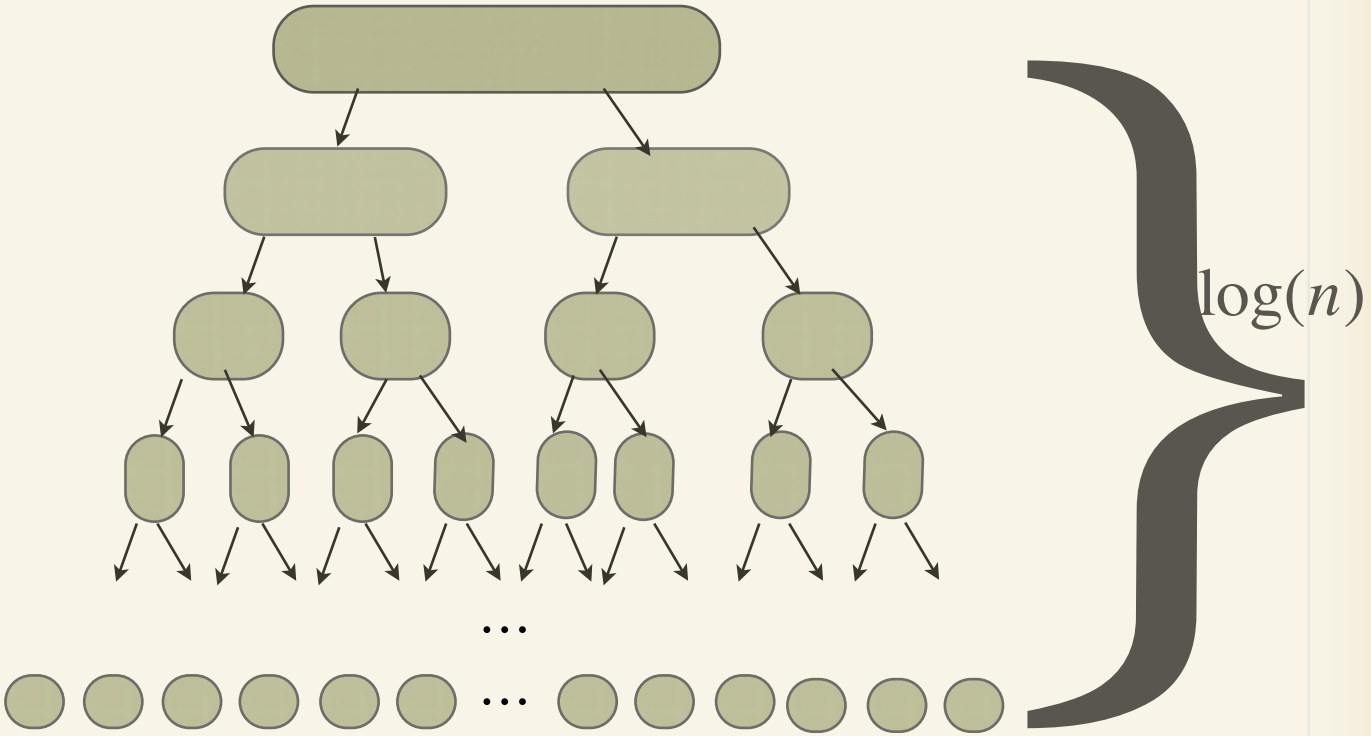
```
void quicksort( vector<int> & a, int low, int high)
{
  if (low < high)
  {
     // select pivot to be element in middle position
    int mid = (low + high)/2;
    int pivot = a[ mid ];
    // put pivot in a[high]
    swap(a[high], a[mid]);
     // Begin partitioning
    int i, j;
    for( i = low, j = high - 1; ; )
    {
       while ( a[i ] < pivot )  ++i;
       while( j > i && pivot < a[j ] ) --j;
       if( i < j )
            swap( a[ i++], a[ j-- ] );
       else
            break;
    }
    swap( a[ i ], a[ high  ] );  // Restore pivot

  quicksort( a, low, i - 1);
  quicksort( a, i + 1, high );
  }
}
```



Best Case: 50/50 Split!

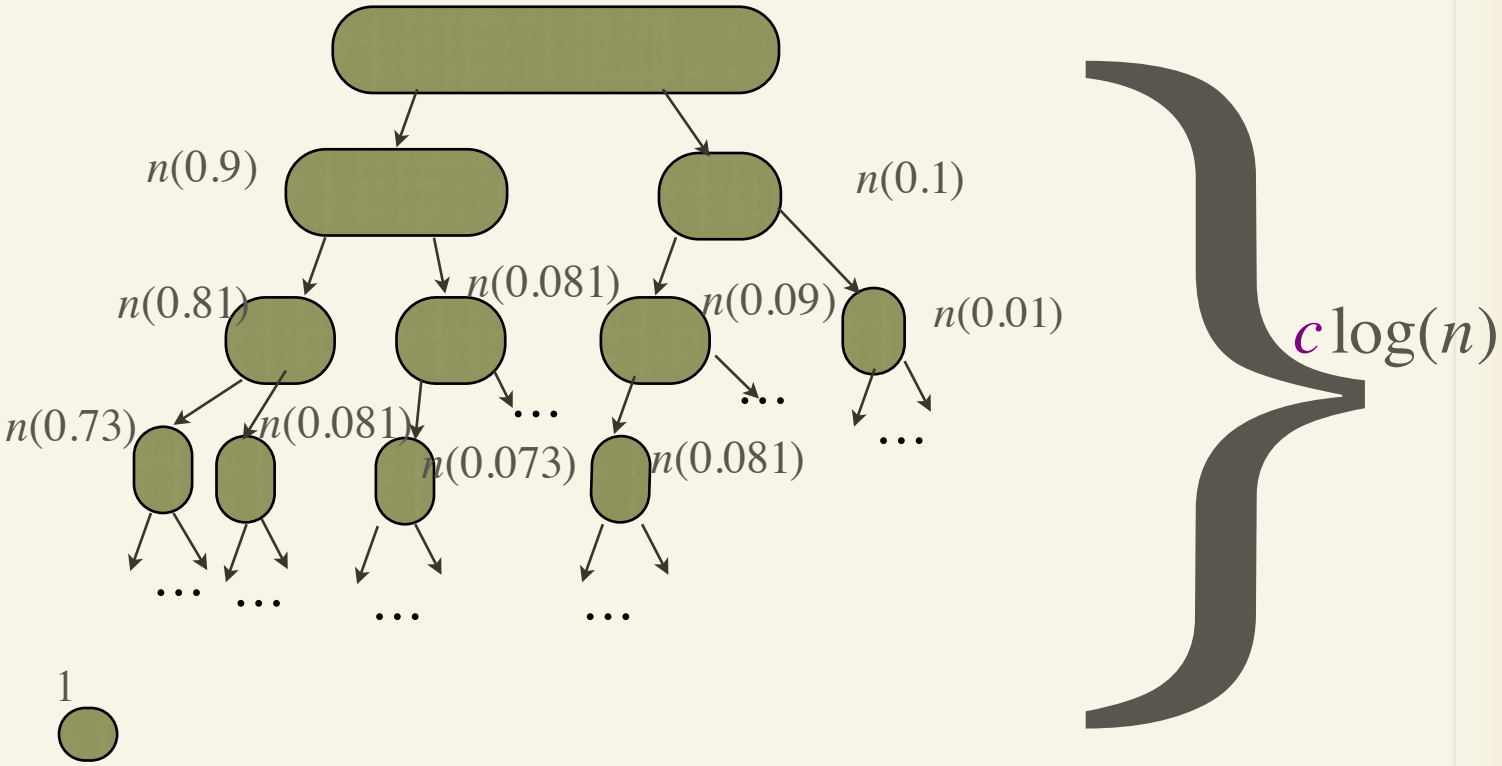


Best Case: 50/50 Split!



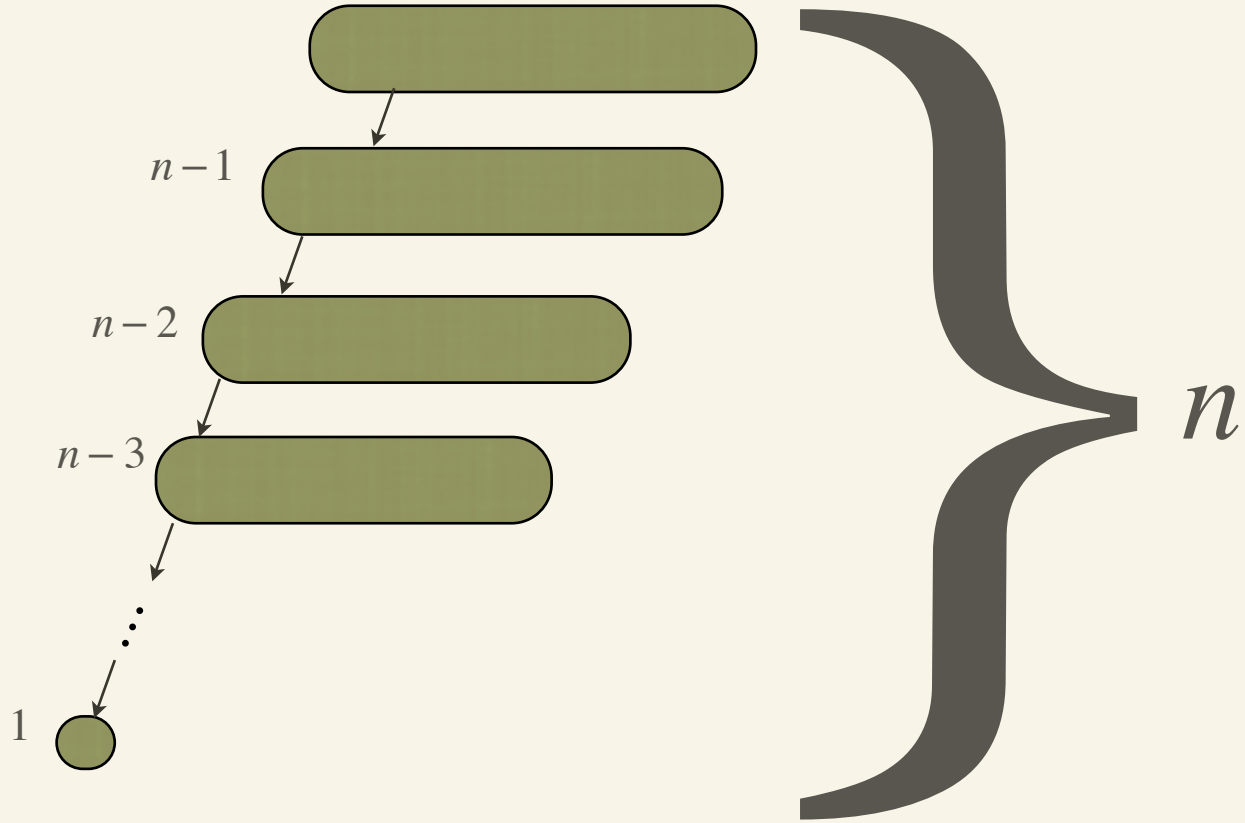
$O(n \cdot \log(n))$ time

Suppose: 90/10 Split...



$O(n \bullet \log(n))$ time

Worst Case: n-1/0 Split...



$O(n^2)$ time

Running Time

- Worst case:
  - suppose largest element is chosen as pivot on each call to Quicksort:
  - Quicksort(a, 0, n-1) calls

    Quicksort(a, 0, n-2) calls

    Quicksort(a, 0, n-3) calls

    ..

    Quicksort(a, 0, 0) calls
- $1 + 2 + \ldots + n = O(n^2)$

Running Time

- Best case:
  - suppose median element is chosen as pivot on each call to Quicksort:
  - Each instance calls Quicksort on two pieces of size roughly n/2
  - Picture similar to MergeSort analysis
- T(n) = 2T(n/2) + n
- O(n log n)
- Luckily AVERAGE CASE is close to BEST CASE.
  - With good pivot selection strategy, you have to be very unlucky to get worst case partition on many calls.

Pivot Selection Strategies

- Pivot = a[low]
  - bad if array is already sorted
- Pivot = a[(high +low)/2]
  - OK, unless you’re unlucky
- Pivot = median element of a[low .. high]
  - hard to compute
- Pivot = median of { a[high], a[middle],a[low]}
  - good compromise in practice

Comparison of Sorts

	worst	average	best
insertion sort	$O(n^2)$	$O(n^2)$	$O(n)$
MergeSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
QuickSort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$

Can we do better?

No! Any comparison based sorting algorithm has worst case at least n log n.

Running Times

n	insertion O(n^2)	merge O(nlog(n))	quick O(nlog(n)))ave
512	0.0018	0.000166	7.4e-05
1024	0.00747	0.000356	0.000157
2048	0.030801	0.000765	0.000325
4096	0.120905	0.001644	0.000697
8192	0.474183	0.003504	0.001444
16384	1.87247	0.007584	0.002947
32768	7.56883	0.015684	0.005933
65536	29.9251	0.033017	0.012211

Lower Bound on ANY comparison based sorting algorithm

Decision Tree for sorting 3 items

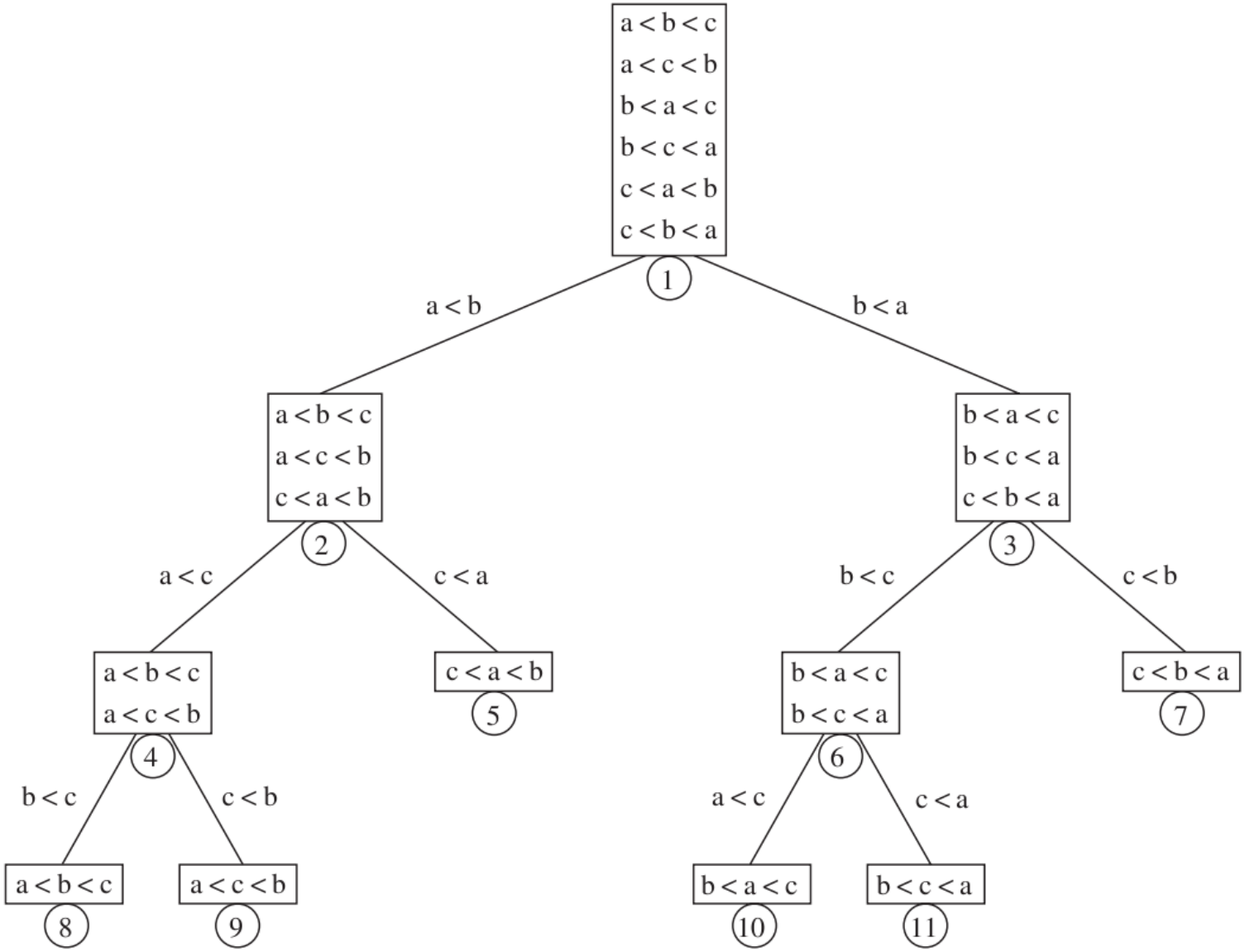


Figure 7.20 A decision tree for three-element sort

Any Algorithm that sorts by using comparisons between elements uses at least cn log(n) comparisons for some input.

(Worst case lower bound for sorting using comparisons)

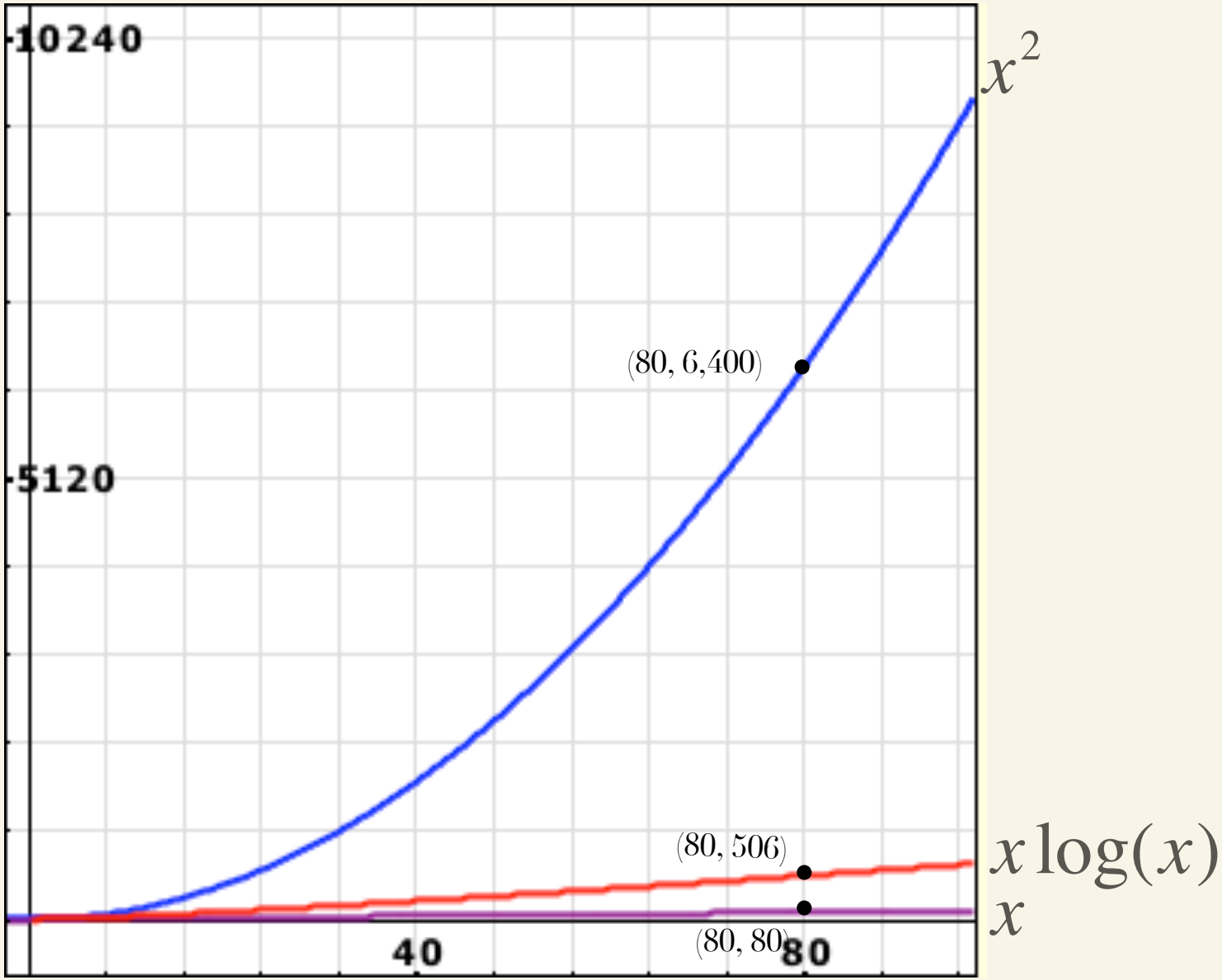
There are n! ways n items can be arranged.

After 0 comparisons all n! possible ways to order the items are possible.

After 1 comparison, the set of possible orders is divided into two groups: the passed the test group and the didn't pass the test group. One of these groups has at least half the permutations. The size of the larger group is at least n!/2

After j comparisons, one group will have at least $n!/2^j$ still-possible permutations.

The number of comparisons till we have only one possible ordering is thus log(n!) (this is about n log n - 1.44n)


10240
5120
40
80
(80, 6,400)
(80, 506)
(80, 80)
x^2
$x \log(x)$
x

Finding Order Stats

- kth order statistic is the kth smallest element in a collection
- special case: median is the (n/2)th order statistic
- Could find it by sorting, but this takes O(n log n) time.
- modification of QuickSort can be used to find kth order stat more efficiently (in average case).

The Driver

```
template <class Comparable>
void quickSelect( vector<Comparable> & a, int k )
{
    quickSelect( a, 0, a.size( ) - 1, k );
}
```

```
void main()
{
    ...
    quickSelect(a, k);
    ...
}
```

```
void quickselect( vector<int> & a, int low, int high, int k )
{
   if (low < high)
   {
      // select pivot to be element in middle position
      int mid = (low + high)/2;
      int pivot = a[ mid ];
      // put pivot in a[high]
      swap(a[high], a[mid]);
       // Begin partitioning
      int i, j;
      for( i = low, j = high - 1; ; )
      {
         while ( a[i ] < pivot )  ++i;
         while( j > i && pivot < a[j ] ) --j;
         if( i < j )
              swap( a[ i++], a[ j-- ] );
         else
              break;
      }
      swap( a[ i ], a[ high  ] );  // Restore pivot

    if (k <= i)
        quickselect( a, low, i - 1, k );
    else if(k > i + 1)
        quickselect( a, i + 1, high, k );
   }
}
```