STL vectors

random access
dynamically allocated
- increases size exponentially, so push_back become O(1)

push_back()
pop_back()
front() and back() retrn values of them
iterators random access
erase(itr) **O(n)**
- returns iterator pointing to new loc. of element following

insert(itr, val) **O(n)**
- returns itr pointing to first of newly inserted elements

begin() - iterator @ top
end() - iterator @ end
rbegin() - reverse itr
rend() - reverse itr

STL Lists

doubly linked
begin, end, rbegin, rend all work
front() and back() retrn values of them
iterators bidirectional
erase(itr) **O(1)**
- returns itr pointing to element following last element erased

insert(itr, val) **O(1)**
- returns itr pointing to first of new inserted elements

push_back()
push_front()
pop_back()
pop_front()
has all the same iterator functions

class Vec

```
template <class T> class Vec {
public:
  // TYPEDEFS
  typedef T* iterator;
  typedef const T* const_iterator;
  typedef unsigned int size_type;
  // CONSTRUCTORS etc
  Vec() { this->create(); }
  Vec(size_type n, const T& t = T()) { this->create(n, t); }
  Vec(const Vec& v) { copy(v); }
  Vec& operator=(const Vec& v);
  ~Vec() { delete [] m_data; }

  // MEMBER FNC & OPS
  T& operator[] (size_type i) { return m_data[i]; }
  const T& operator[] (size_type i) const { return m_data[i]; }
  void push_back(const T& t);
  iterator erase(iterator p);
  void resize(size_type n, const T& fill_in_value = T());
  void clear() { delete [] m_data;  create(); }
  bool empty() const { return m_size == 0; }
  size_type size() const { return m_size; }

  // ITERATOR OPERATIONS
  iterator begin() { return m_data; }
  const_iterator begin() const { return m_data; }
  iterator end() { return m_data + m_size; }
  const_iterator end() const { return m_data + m_size; }

private:
  // PRIVATE MEMBER FUNCTIONS
  void create();
  void create(size_type n, const T& val);
  void copy(const Vec<T>& v);

  // REPRESENTATION
  T* m_data; // Ptr to 1st loc in the allocated array
  size_type m_size;  // No el
  size_type m_alloc; // No array locs allocd  m_size <= m_alloc
};
```

class dslist

```
template <class T>
class Node {
public:
  Node() : next_(NULL), prev_(NULL) {}
  Node(const T& v) : value_(v), next_(NULL), prev_(NULL) {}
  // REPRESENTATION
  T value_;
  Node<T>* next_;
  Node<T>* prev_;
};

// A "forward declaration"
template <class T> class dslist;

// LIST ITERATOR
template <class T>
class list_iterator {
public:
  // constructors & destructor
  list_iterator(Node<T>* p=NULL) : ptr_(p) {}
  // NOTE: the implicit compiler defs of copy, assign, destr good
  // dereferencing operator
  T& operator*()  { return ptr_->value_;  }
  // increment & decrement
  list_iterator<T>& operator++() { // pre-increment, e.g., ++iter
    ptr_ = ptr_->next_;
    return *this;
  }
  list_iterator<T> operator++(int) { // post-increment,iter++
    list_iterator<T> temp(*this);
    ptr_ = ptr_->next_;
    return temp;
  }
  list_iterator<T>& operator--() { // pre-decrement, e.g., --iter
    ptr_ = ptr_->prev_;
    return *this;
  }
  list_iterator<T> operator--(int) { // post-decrement, e.g., iter--
    list_iterator<T> temp(*this);
    ptr_ = ptr_->prev_;
    return temp;
  }
  // the dslist class needs access to the private ptr_ member variable
  friend class dslist<T>;

  // Comparions operators
  bool operator==(const list_iterator<T>& r) const {
    return ptr_ == r.ptr_; }
  bool operator!=(const list_iterator<T>& r) const {
    return ptr_ != r.ptr_; }
private:
  // REPRESENTATION
  Node<T>* ptr_;    // ptr to node in the list
};

template <class T>
class dslist {
public:
  // default constructor, copy constructor, assignment operator, & destructor
  dslist() : head_(NULL), tail_(NULL), size_(0) {}
  dslist(const dslist<T>& old) { copy_list(old); }
  dslist& operator= (const dslist<T>& old);
  ~dslist() { destroy_list(); }

  typedef list_iterator<T> iterator;

  // simple accessors & modifiers
  unsigned int size() const { return size_; }
  bool empty() const { return head_ == NULL; }
  void clear() { destroy_list(); }

  // read/write access to contents
  const T& front() const { return head_->value_;  }
  T& front() { return head_->value_; }
  const T& back() const { return tail_->value_; }
  T& back() { return tail_->value_; }

  // modify the l.l. structur
  void push_front(const T& v);
  void pop_front();
  void push_back(const T& v);
  void pop_back();

  iterator erase(iterator itr);
  iterator insert(iterator itr, const T& v);
```

```
  iterator begin() { return
iterator(head_); }
  iterator end() { return
iterator(NULL); }

private:
  // private helper functions
  void copy_list(const
dslist<T>& old);
  void destroy_list();

  //REPRESENTATION
  Node<T>* head_;
  Node<T>* tail_;
  unsigned int size_;
};
```

Recursion

Binary Search: ***O(log(n))*** *to find value within sorted array*

```
template <class T>
bool binsearch(const
std::vector<T> &v, int low, int
high, const T &x) {
  if (high == low) return x ==
v[low];
  int mid = (low+high) / 2;
  if (x <= v[mid])
    return binsearch(v, low,
mid, x);
  else
    return binsearch(v, mid+1,
high, x);
}
template <class T>
bool binsearch(const
std::vector<T> &v, const T &x) {
   return binsearch(v, 0,
     v.size()-1, x);
}
```

Merge Sort: ***O(nlog(n))*** *always*

```
// We split the vector in half,
recursively sort each half, and
merge the two sorted halves into
a single sorted interval.
template <class T>
void mergesort(int low, int
high, vector<T>& values,
vector<T>& scratch) {
  if (low >= high) return;
  int mid=(low+high)/2;
  mergesort(low, mid, values,
scratch);
  mergesort(mid+1, high, values,
scratch);
  merge(low, mid, high, values,
scratch);
}
// Non-recursive function to
merge two sorted intervals
(low..mid & mid+1..high) of a
vector, using "scratch" as
temporary copying space.
template <class T>
void merge(int low, int mid, int
high, vector<T>& values,
vector<T>& scratch) {
  int i=low, j=mid+1, k=low;
  // still something left
  while (i <= mid && j <= high){
    // look at the top values,
grab the smaller one, store it
in the scratch vector
   if (values[i] < values[j]) {
    scratch[k] = values[i]; ++i;
   } else {
    scratch[k] = values[j]; ++j;
   }
++k;
  }
  // Copy the remainder of the
interval that hasn't been
exhausted
  for ( ; i<=mid; ++i, ++k )
scratch[k] = values[i];  // low
interval
  for ( ; j<=high; ++j, ++k )
scratch[k] = values[j]; // high
interval
  // Copy from scratch to values
  for ( i=low; i<=high; ++i )
    values[i] = scratch[i];
}
```

Quick Sort: ***O(nlog(n))****,* ***->n^2***

```
// Choose a "pivot" and
rearrange the vector. Returns
the location of the pivot,
separating top & bottom
(hopefully it's near the halfway
point).
int partition(vector<double>&
data, int start, int end, int&
swaps) {
  int mid = (start + end)/2;
  double pivot = data[mid];
  int i = start;
  int j = end;
  while ( i < j )
   while ( data[i] < pivot ) {
     i++;
   }
   while ( data[j] > pivot ) {
     j--;
   }
   if (i < j) {
     double tmp = data[i];
     data[i] = data[j];
     data[j] = tmp;
   }
  }
  return i; // where is pivot
}
void quickSort(vector<double>&
data, int start, int end) {
  if(start < end) {
    int pIndex = partition(data,
start, end);
    // after calling partition,
one element (the "pivot") will
be at its final position
    quickSort(data, start,
pIndex-1);
    quickSort(data, pIndex+1,
end);
  }
}
void quickSort(vector<double>&
data) {
  quickSort(data,0,
    data.size()-1);
}
```

generate ideas

• play with examples! develop a strategy for solving the problem? try any strategy on several examples.possible to map this strategy into algorithm?
• solving simpler version of the problem first and learn from the exercise or generalize.
• Does problem look like another problem you know how to solve?
• If given partial solution, could extend to complete sol'n?
• can split prob 1/2 and solve halves (recursively) separately?
• Does sorting the data help?
• Can split prob in different cases, handle cases separately?
• Can discover something fund. about prob that makes it easier to solve or makes you able to solve it more efficiently?
• have idea you think works, evaluate it: indeed works? other ways to approach might be better / faster? if not work, why not?

mapping to code

• How to represent the data? most efficient? what is easiest?
• use classes to organize data? What data stored and manipulated as unit? What info needs stored for each object? What ops (past simple accessors) helpful?
• How divide prob in2 units of logic that become functions? Can reuse any code u have already? any logic u write be re-usable?
• going to use recursion or iteration? What info need to handle during loops or recursive call, how is it "carried along"?
• How effective is sol'n? Is ur sol'n gen'l? How's performance? (order not'n of no. operations)? think - better ideas/approaches?
• Make notes about logic of code as you write. These become invariants - what should be true at begin and end of each iteration / recursive call.

details

• everything initialized correctly, e.g. bool flag vars, accumulation vars, max/min vars?
• Is logic of conditionals correct? Check several times and test examples by hand.
• have bounds on loops correct? Should you end at n, n-1 or n-2?
• Tidy "notes" to formalize invariants. Study code to ensure it in fact has it right. use asserts test invar's. (sometimes checking invar impossible or too costly to be practical.)
• works on corner cases; e.g. when answer is on start or end of data; when repeated values in data; when data set very small/ large?