EECS 281

Preparation

Tools

Command Line

- Some tips and reminder:
  - I/O redirection: ./program < test.txt
  - ./program 2> test.txt (for cert)
  - `diff` to compare two files

Makefile

- Idea: make compiling / use of debugging tools / submission more convenient
- Configurating makefile: four modifications
  - Uniqname
  - Identifier
  - Executable (actually whatever)
  - Dependencies (optional)
- Using make:
  - make (make debug) (for valgrind and perf)
  - make clean
  - make partialsubmit
  - make fullsubmit (prepare for autograder)

Valgrind

- Idea: check undefined behavior (memory leak etc.)

Perf

- Idea: check the execution speed of different parts of the program
- Usage:

- make debug # must use this
  perf record -F 1000 --call-graph dwarf -e cycles:u ./[PROGRAM_NAME] < [INPUT]
  perf report
    - Samples: the more samples, the more precise
    - Check the run time proportion in the "self" column

I/O

- Outputs:
    - Cout - write to stdout; cerr - write to stderr
- Inputs: **Loop inputing: different methods**

```
erdao 123 erdao123 f

4.44 r4f -4 002 liang
```

- Using >>
    - **NEVER read any whitespace (jump if met)**

```
string s;
while(cin >> s){ // only executes if s is read in properly
cout << s;
}
// erdao123erdao123f4.44r4f-4002liang

char c;
while(cin >> c){
 cout << c;
}
// erdao123erdao123f4.44r4f-4002liang

int i;
while(cin >> i){
 cout << i;
}
// output nothing
```

- Using `getline()`
    - **read ALL characters (including white spaces) until a given one (DEFAULT '\n')**

- **Removes and discards the given character**
- **Possible to read an empty line (e.g. if the line is only "\n")**

```
string line;
while(getline(cin, line)){
 cout << line;
 // need to add '\n' manually if need to output the same as input
}
```

- Mix of using >> and getline()
  - **Make sure that all spaces are got rid of before using getline() for the next line**

Getopt_long

- Idea: easy pharsing of program options and arguments
- Using the function
- #include <getopt.h>
  getopt_long(int argc, char** argv, char * [A_STRING], struct option options, int &option_index)
  - The string:
    - With all short-version options; add ":" after if it requires arguments
  - The `option` struct (the struct is defined in getopt.h)

```
struct option longOpts[] = {{ "print", required_argument, nullptr, 'p' },
                              { "help", no_argument, nullptr, 'h' },
                              { "name", no_argument, nullptr, 'n'},
                              { "artist", no_argument, nullptr, 'a'},
                              { "listens", no_argument, nullptr, 'l'},
                              { nullptr, 0, nullptr, '\0' } // required to
read the terminator
                              };
// format: {long_option, required_argument/no_argument, nullptr,
short_option }
```

- Sample

```
int gotopt;
int option_index = 0;
```

```
option long_opts[] = {
 // ...
};

while ((gotopt = getopt_long(argc, argv, "an:", long_opts, &option_index))
!= -1){
 switch (gotopt){
   case 'a':
     //...
     break;
   case 'b':
     // ...
     break;
   default:
     // ...
     break;
} // switch
} // while
```

- Option arguments are automatically stored in "optarg", a char* global variable

Setups

CAEN Linux

- Idea: It is a remote Linux system (kind of like WSL). Autograder runs on this system. Better run the code on this system in order to make sure the code compiles
- Valgrind, perf etc. tools are also available on the system
- Accessing the system: using **ssh**
- ssh <uniqname>@login.engin.umich.edu
- Transmitting files to the remote system:
  - Way 1: using rsync (under ~ directory)
  - rsync -rtv --exclude '.git*' [LOCAL_DIRECTORY_NAME] <uniqname>@login.engin.umich.edu:[DEST_DIRECTORY_NAME]/
  - Way 2: make command (under ~ directory) (automatically filter out some files, only source file)
  - make sync2caen
- More: Setup CAEN | EECS 280 Tutorials (eecs280staff.github.io)

Autograder submission

- 3 submissions / day
- `make partialsubmmit` don't count as a submit, `make fullsubmit` counts
- Must do
    - Identifier to top of all project files (source code, header, makefile)

Optimization Tips

- Compile options: use **-O3** option in g++ (reorganize codes automatically to improve speed and memory)

Memory optimization

- Class & Struct
    - When creating a class or structure, create member variables in order from largest to smallest
    - NEVER use global variables
    - The `size_t` type should be used whenever you’ re referring to the size of a container
- Container
    - When creating a vector or deque, if possible create it with the correct size. This actually saves both time and memory.
    - Reduce the size of vectors: `vector<char>` better than `vector<int>`

Time optimization

- I/O
    - As the very first line of main(), before any other code, do the following: `std::ios_base::sync_with_stdio(false);` This turns off what is called “synchronized I/O”
    - Output `\n` instead of `endl` . Because endl flushes buffer every time
- Container
    - Reduce the size of vectors: vector<char> better than vector<int>
    - Reuse large arrays instead of declaring a new one
    - As few [] operator as possible. It takes much time
    - As few `.size()` as possible. Time consuming from experience

- Function call
    - If a function can be called once, don't call it twice. Instead, call it only once and save that value
    - For large object, use pass-by-reference instead of pass-by-value
- Code organization
    - When you’ re writing a class, implement small member functions (especially getters and setters) inside the header file.

There is no difference in memory or time efficiency between classes & structs

Containers, Data Structures, ADT

- Define a collection of valid operations and their behaviors on stored data, called **container**
- This interface to the data (the operations) is called an **abstract data type**.
- The implementation of the interface is called a **data structure**

Containers

- Types of containers

type	criteria	Is	Is not
searchable	Support find()	Vector, deque, list	Stack, queue
sequential	Allows iteration	Vector, deque, list	Stack, queue
ordered	Maintains current order as the time it is inserted Support insert() in any location Relative position of two elements can not change	Vector, deque, list	heap
sorted	Stored in a pre-defined order Not support insert() in any location	Map, set	Heap, list

    - Caution: these four types are not mutually-exclusive.
- Common operations: constructor, destructor, add, remove, get an elem, size, copy

- Access container items
    - Mainly two types: sequential vs. Random access
- Copy constructor & assignment operator
    - Best realization: copy-swap method

```
#include <utility> // Access to swap

Array(const Array &other) : length{other.length}, data{new double[length]} {
    for (size_t i = 0; i < length; ++i)
    data[i] = other.data[i];
}

Array &operator=(const Array &other) { // Copy-swap method
    Array temp(other); // use copy constructor to create object
    // swap this object's data and length with those from temp
    std::swap(length, temp.length);
    std::swap(data, temp.data);
    return *this; // delete original, return copied object
}
```

- Storing / getting from a container

	value	pointer	reference
Storing data type	Good	Used for shared data	NO
get() return type	Costly	Unsafe	Good (const &)

Data Structures

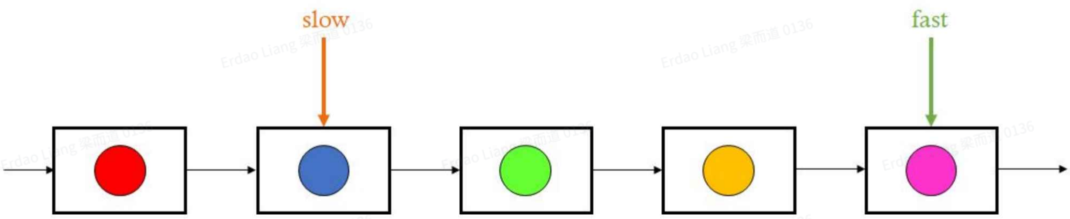
Arrays & Linked Lists

- Two major underlying inplementation of many ADTs
- Runtime Comparison

	Arrays	Linked Lists
Access	Random in O(1) time Sequential in O(1) time	Random in O(n) time Sequential in O(1) time
Insert and Append	Inserts in O(n) time Appends in O(n) time (O(1) amortized possible if vector)	Inserts in O(n) time Appends in O(n) time (O(1) with tail ptr)
Bookkeeping	Ptr to beginning CurrentSize or ptr to end of space used (optional) MaxSize or ptr to end of allocated space (optional)	Size (optional) Head ptr to first node Tail ptr to last node (optional) In each node, ptr to next node
Memory	Wastes memory if size is too large* Requires reallocation if too small*	Allocates memory as needed Memory overhead for pointers (wasteful for small data items)

- Arrays
    - Topic 1: relation between arrays & pointers
        - TODO
    - Topic 2: Array resizing - causes pointer invalidation
        - Mechanism: create a larger new array - copy data one by one to a.new array -> delete old array
        - This is the case for all "auto-resizing" containers - strings, vectors, etc.
        - Having a consistently larger capacity than size wastes memory, but repeatedly increasing the size beyond the capacity wastes time, so it is better if size is known in advanced and use the resize() / reserve() and don't change it
- Linked lists:
    - Topic 1: find middle element in a linked list - **the two pointer technique**

- How can you solve the problem? *Use two pointers!*
    - Start with two pointers, `fast` and `slow`.
    - Increment `fast` by two, then increment `slow` by one.
    - When `fast` reaches the end, `slow` must point to the middle node!



Heaps

- Heap properties: (1) completeness; (2) heap-ordering
- Binary Heaps: MIN heap & MAX heap
- Realization
    - Maintaining the heap property

	Priority of an element increases: fixUp	Priority of an element increases: fixDown
Logic	Swap the altered node with its parent, moving up until either 1. reach the root 2. reach a parent with a larger or equal key	Swap the altered node with the greater of its children, moving down until: 1. reach the bottom of the heap 2. Both children have a smaller or equal key
Complexity	Log n (number of levels of the heap)	Log n

- Insertion & removal

	Insertion	Removal
Logic	1. insert the new item into the bottom of the heap 2. call **fixUp()** on the newly inserted item	1. remove the root item by replacing it with the last element in heap 2. delete the last item in the heap 3. call **fixDown()** on the root element
Complexity	Log(n)	Log(n)

- Make heap / heapify

	Idea 1	Idea 2
Logic	repeatedly call **fixUp()** starting from the top of the array and moving down.	repeatedly call **fixDown()** starting from the bottom of the heap and moving up.
	equivalent to repeatedly inserting items into a heap.	equivalent to making many small heaps and gradually merging them by adding roots and finding the correct positions for them.

Complexity	**O(n log n)**	**O(n)**
		Why? The bottom level of the heap has the greatest number of items We are effectively building many small heaps and merging them by adding new nodes, which costs O(log n) - but mostly on very small heaps, limiting the work

- Heap Sort: heapify -> repeatedly remove items to the back
  - Complexity: O(n + n log n) = O(n log n)

Sets

- Set operation

Union (A ∪ B)	Iterate over each vector. Push the lower element to the output vector, and increment its iterator. If the elements are the same, only push one and increment both, to avoid duplication.
Intersection (A ∩ B)	Iterate over each vector. If the elements are the same, push one to the output vector and increment both. Otherwise, increment the iterator of the lower element (in case the next element matches the higher element).
Set Difference (A – B)	Iterate over each vector. If the elements are the same, increment both. If A's element compares lower, push it to the output vector. Increment the iterator of the lower element (in case the next element matches the higher element).

Union Find

- Goal: given some disjoint sets and two items, want to answer the question of whether the two items are in the same set
- Idea: Each item has a “representative” that helps identify the group they are in
  - **union(x, y)** joins x and y so that they become part of the same group - if x and y are in diffferent groups, the groups will be combined into a larger group
  - **find(x)** returns the “representative” of the group that x belongs to
- Realization

```
class UnionFind {
private:
    vector<size_t> reps;
```

```
public:
    UnionFind(size_t size) {
        reps.reserve(size);
        for (unsigned i = 0; i < size; ++i) {
        // at the beginning, every node represents itself!
        reps.push_back(i);
        }
    }

    size_t find(size_t x);

    void set_union(size_t x, size_t y);
};

UnionFind::find(size_t v){
    return (v == reps[v]) ? v : (reps[v] = find(reps[v]));
}

UnionFind:set_union(size_t x, size_t y){
    reps[find(y)] = find(x);
}
```

- Path compression: TODO

Hash Tables

- Motivation: need avg O(1) insert, O(1) search, O(1) delete (no container achieves this up til now)
- Hash function: h(key) = compress(translate(key))
- Load factor a = # elements / # buckets
- Collision resolution
  - Separate chaining (most common): use a linked list for each index
  - Open addressing: find another empty location
    - Setup: Probing outcome - empty / hit / full / deleted
    - operations

Operation	How to do	Time complexity	Space complexity

insert(x)	h(x), if empty/deleted, store it; if full, jump to next index; if hit, do nothing	• Avg: O(1+a) (still O(1) if keep a low a) • Worst: O(n)	• Avg: O(n) • Worst: O(n)
lookup(x)	h(x), if hit, return it; if full/deleted, jump to next index; if empty, not found		
remove(x)	first search(x); if found, mark it as deleted, else do nothing		

- Different ways of open addressing

Linear probing	if (t(key) % M) full then try ((t(key) + j) % M)
Quadratic probing	if (t(key) % M) full then try ((t(key) + j^2) % M)
Double hashing	if (t(key) % M) full then try ((t(key) + j*t'(key)) % M) (use a second hash function)

- # of keys increases -> insert/search performance decreases
- Clusters: the smaller the cluster, the lower the runtime

- Increase performance:
  - Use prime numbers for table sizes (reduce collision)
  - Design good hash functions
  - Keep load factor low - need dynamic hashing
- Dynamic hashing: increases the table size when it reaches some pre-determined load factor
  - Create a larger table -> insert all non-deleted elements
  - **Caution: Their positions might change**

Trees

- Tree concept
  - Simple trees vs. Rooted trees
  - Height vs. Depth
  - Internal nodes vs. External nodes (i.e. leaves)
  - Parent, children, ancestor,

Binary tree

- Implementation & complexities

  With n as the # of nodes,

	insert key		Search key	remove key	**Find parent**	Find children	**space**
	avg	worst	Avg, worst	avg			
Array implementation	O(1)	O(n)	O(n)	O(n)	**O(1)**	O(1)	**O(2^n)**
Linked list implementation	O(1)	O(n)	O(n)	O(n)	**O(n)**	O(1)	**O(n)**

- Translate from general trees to binary trees: 孩子兄弟表示法
- Binary tree Traversal
  - Types of traversal: pre-order / in-order / post-order / level-order
  - Variant: level-order traversal level-by-level

```
void levelTraverse(Node * root){
    if (!root) return;
    queue<Node *> q;
    q.push_back(root);
    while (!q.empty()){
        int levelSize = q.size();
        for (int i = 0; i < levelSize; i++){
            Node *curr = q.front();
            q.pop_front();
            cout << curr->val << " ";
            if (curr->left)
                q.push_back(curr->left);
            if (curr->right)
                q.push_back(curr->right);
        }
    }
}
```

  - tree reconstruction: **(pre-order OR post-order) AND (in-order OR (told that it is a BST))**

Binary search trees

- Operation complexities:

	avg	worst
Search/insert/remove	O(log n)	O(n)
when	Tree is balanced	Tree is sticky

- Insert
  - Caution: must have a consistent rule for determining where duplicates go
- Remove
  - Four cases: no child, only left child, only right child, both children
  - For the 4th case, find its in-order predecessor / successor

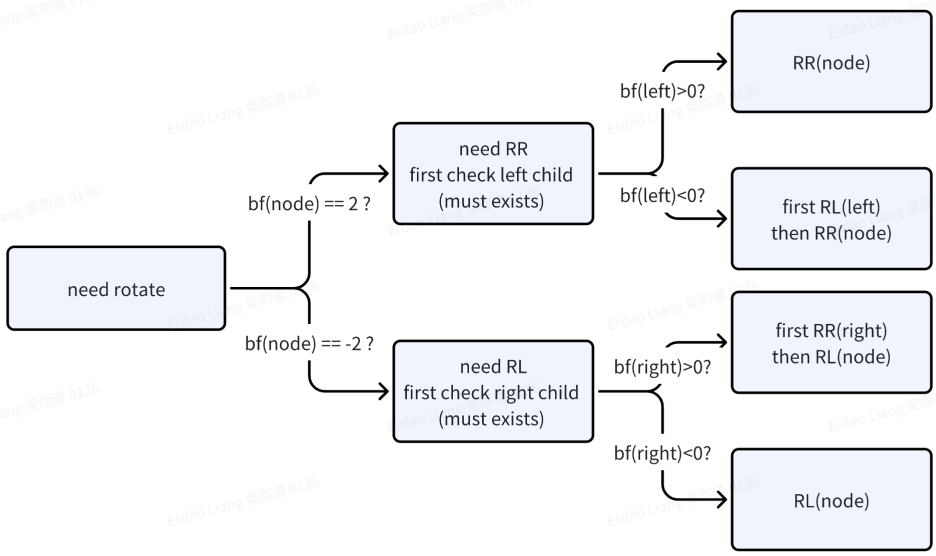
```
template <class T>
void BinaryTree<T>::remove(Node *&tree, const T &val) {
    Node *nodeToDelete = tree;
    Node *inorderSuccessor;
    // Recursively find the node containing the value to remove
    if (tree == nullptr)
        return;
    else if (val < tree->value)
        remove(tree->left, val);
    else if (tree->value < val)
        remove(tree->right, val);
    else {
        // Check for simple cases where at least one subtree is empty
        // case 1,2: no child, or only right child
        if (tree->left == nullptr) {
            tree = tree->right;
            delete nodeToDelete;
        } // if
        // case 3: only left child
        else if (tree->right == nullptr) {
            tree = tree->left;
            delete nodeToDelete;
        } // else if
        // case 4: both children exist
        else {
            // Node to delete has both left and right subtrees
```

```
            // here is how to find the successor:
            // turn right, and then always left
            inorderSuccessor = tree->right;
           while (inorderSuccessor->left != nullptr)
                inorderSuccessor = inorderSuccessor->left;
            // Replace value with the inorder successor's value
            nodeToDelete->value = inorderSuccessor->value;
            // Remove the inorder successor from right subtree
            remove(tree->right, inorderSuccessor->value);
        } // else
    } // else
} // BinaryTree::remove()
```

- Sort an array using binary search trees
    - Construct a BST; then in-order traversal
    - Time complexity O(n*Logn), space O(n) (not in-place sorting)

AVL trees

- Motivation: make worst case of a BST O(log n), instead of O(n)
- Property:
    a. Is a BST
    b. Balance factor of every node >= -1, <= 1
- Fix balancing: rotate




```
Algorithm checkAndBalance(Node *n)
    if balance(n) > +1
```

```
        if balance(n->left) < 0
            rotateL(n->left)
        rotateR(n)
    else if balance(n) < -1
        if balance(n->right) > 0
            rotateR(n->right)
        rotateL(n)
```

- Search - the same as BST
- Insert
  - First insert as that is done in BST
  - Back upward, if find a node unbalanced, rotate the node; **# of rotation <= 1**
- remove
  - First remove as that is done in BST
  - Back upward, if find a node unbalanced, rotate the node; **# of rotation can be any**

Graphs

- Concepts
  - simple graphs = no parallel edges + no self-loops
  - Directed graph vs. Undirected graph
  - Weighted graph vs. Unweighted graph
  - Dense graph (|E| = |V|^2) vs. Complete graph (|E| = |V|): it is relative

sparse graph	Graph without edge; hyperlinks between web pages in the internets
dense graph	cliques;

  - Cost:
    - unweighted graph - assume cost of each edge is 1
    - weighted graph - sum of weights on all edges
- Implementation

	Space	
Adjacency matrix	O(V^2)	

Adjacency list	O(V+E)	

Graph algorithms

- Common graph algorithms & complexity

Task	Adjacency matrix	Adjacency list
Task: Find whether an edge between v1 & v2 exists	O(1)	O(1+E/V)
Task: Determine the shortest edge going from V1	O(V)	O(1+E/V)
Task: Whether an edge goes from V1	O(V)	O(1)

- Graph traversals

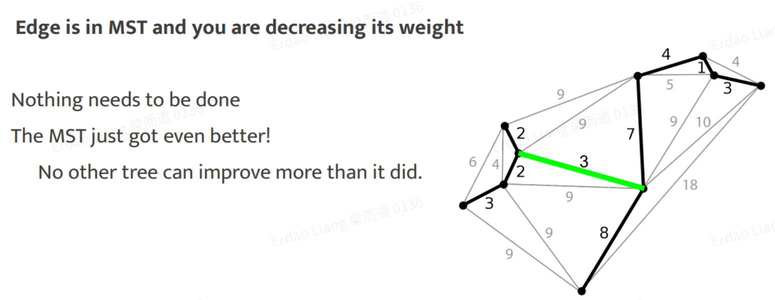
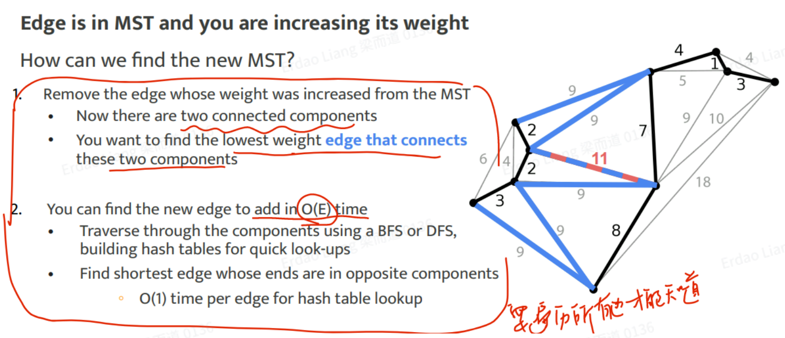
	Depth-first search (DFS)	Breadth-first search (BFS)
Analogy	Preorder search of trees	Level-order search of trees
Implmentation	**Use a stack / use recursion**	**Use a queue**
Complexity	• **O(|V|+|E|)** using adjacency list (each v/e is visited at most once) • **O(|V|^2)** using adjacency matrix (b.c. Iterating on edges of a vertex takes O(n))	
Pros & cons	Pros: space - at most O(log n) vertices in the stack (no more than the depth of the search tree)	Cons: Space - at most O(n) vertices in the queue
What it can do	• Find a vertex • Only find the shortest path for trees	• Find a vertex • For unweighted graphs, BFS returns the shortest path to that vertex

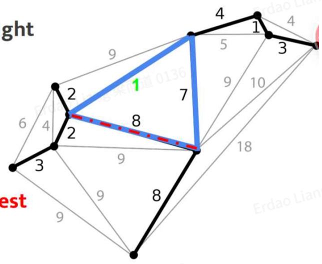
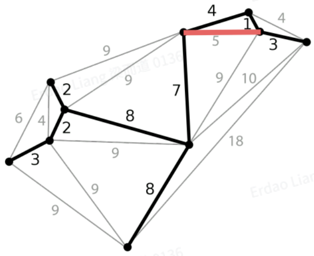
Minimal spanning trees

- MST theory
  - Definition of MST: connected; acyclic; adding any edge makes it cyclic
  - Properties:
    - The first & second shortest edge must be in MST, but not necessarily those later
- Constructing an MST

	Prim Algo (greedy)	Kruskal Algo (greedy)
Algorithm	Separate vertives into innies & outies Iteratively add nearest outie, converting to an inne Store k, d, p for each vertex • K (bool): an innie? • D (double): nearest distance to innie graph? • P (int): nearest innie neighbor?	Greddily select shortest edges that do not induce cycles
Complexity	• **O(|V|^2) for adj matrix & linear search** • **O((V+E)logV) = O(ElogV) for adj list & binary heap**	**O(ElogE) = O(ElogV)** • Sort edges in O(ElogE) • Use union-find to keep track of vertices in each component; 2 finds & at most 1 union for each edge
When to use	For dense graphs	For sparse graphs (for dense graphs, ElogE = V^2 log (V^2), worse than prim algo linear search)

- Modifying MST

<table>
<tr><th>Case</th><th>Modification</th></tr>
<tr><td>Case 1: edge in MST, weight decrease</td><td>
</td></tr>
<tr><td>Case 2: edge in MST, weight increase</td><td>
</td></tr>
</table>

Case 3: edge not in MST, weight decrease	**Edge is not in MST and you are decreasing its weight** How can we find the new MST? 1. Add the edge whose weight was decreased to the MST; now you've got exactly 1 **cycle** 2. Remove the edge in this cycle that has the **highest** weight; you can do this using DFS or BFS ◦ Complexity is O(|V|) 
Case 4: edge not in MST, weight increase	**Edge is not in MST and you are increasing its weight** **5->7 in the right** Nothing needs to be done; non-MSTs just got worse, but the MST is unchanged. 

Shortest Path

- Both greedy & dynamic programming

	Dijstra	Floyd
	1. Set the distance for s to be 0 2. Loop |V| times a. Find the unvisited vertex v with the shortest distance (to current innies) b. Mark k_v as visited c. For each v's adjacent unvisited vertex u: If (d_v + weight(u,v) < d_u) Update d_u = d_v + weight(u,v)	
complexity	Time: **O(|V|^2)** Space: O(n)	

Abstract Data Types

Stack & Queue

- Operations:

	size	empty	push	pop	End access	[]
	Y	Y	One end	One end	Y	NO
STL complexity	1	1	1	1	1	

- Implementation

	Stack	Queue
Possible implementation	Array Linked list	Circular buffer Doubly-linked list
STL implementation	Can choose underlying data structure (deque by default)	

  - Circular buffer method for queue: front_idx at the first element, back_idx at **one pass** the last element, front_idx==back_idx when [empty] or [array full]
- Interview Question
  - Topic 1: Sorting a stack:
  - Topic 2: Implementing a queue with stacks
- Using: simulate the feature of stack/queue; no need for random access

Deques

- Idea: a combination of stack & queue
  - In STL: also traverse using iterators and supports [] access
- Operations

	size	empty	push	pop	End access	[]
	Y	Y	Two ends	Two ends	Two ends	Y
STL complexity	1	1	1	1	1	1

- Implementation:

Possible implementation	• Doubly-linked list - cons: [] not O(1) • Circular arrays - cons: pointer invalidation
STL implementation	**essentially a deque of deques** dynamic array of pointers to dynamic arrays of a fixed size (the "chunk size") , which are allocated as necessary no reallocating -> no invalidation of pointers Support O(1) []

- Using: always better than stack & queue

Vector

- Operations

	size	empty	push	pop	End access	[]
	Y	Y	One end	One end	Y	Y
STL complexity	1	1	1	1	1	1

Similar to deques, but lose push_front(value) and pop_front() in exchange for better performance.

- STL implementation: fixed size array

.resize()		changes **size** (and increases capacity if needed)
.reserve()		Only changes compacity, doesn't change size

  - Capacity change causes reallocation - previous pointers are invalidated
- Using: always use vector unless fast push_front(value) and pop_front() need

Priority Queue

- Idea: support two operations (insert an item + remove an item with the highest priority)
- Operations:

	size	empty	top	push	pop

STL Complexity	1	1	1	**Log n**	**Log n**

- Implementaion

	Implementation	Insert	Remove
Possible implementation	Unordered array	1	n
	Sorted array	n	1
	Binary heap	Log n	Log n
STL implementation	Binary heap		

- Using priority_queue in STL:

```
template <class T,
    class Container = vector<T>,
    class Compare = less<typename Container::value_type>
> class priority_queue
```

  - Argument 1: the type of object stored in the priority queue
  - Argument 2: the underlying container used (the default is usually fine)
  - Argument 3: a function object (functor) type used to determine the priority between two objects
    - defaults to std::less<T>, which creates a MAX priority queue; can use greater<T>
- Application: find the kth largest element in an unsorted vector

Unordered Map/Set

- Operations & runtime

Function	Effect
`operator[]`	Gives a reference to the value-object with the corresponding key. Will create a **default** value-object if the key is not found. **Computes hash-function every time.**
`.find()`	Returns an iterator to the key-value pair matching a certain key, end() if it doesn't exist.
`.insert()`	Takes in a key-value pair, tries to insert the pair, and returns a pair containing an iterator to the key-value pair with a bool for whether the insertion actually took place (this function cannot change the existing value).
`.insert_or_assign()`	Same as insert, but will change an existing value.
`.erase()`	Removes a key-value pair from the hash table.
`.begin() and .end()`	Returns a **`ForwardIterator`** that will traverse key-value elements in *some* order.

- using & common pitfalls

```
// declaration
unordered_map<int, string> mp;

// in searching, avoid adding keys accidentally
auto findit = mp.find(3);
if (findit == map.end()) // sign of not found
    cout << "not found" << endl;

// if call mp[x] for not existing key x, then will create key x with
default value
// (usually 0 or "")
```

  - **Duplicate keys not allowed; actually updating values; do nothing if insert the same key**
  - Unsorted: can use a forward iterator, but order not guaranteed
- STL Implementation

  The actual implementation details can vary between different C++ standard library implementations. The choice of hash function and collision resolution strategy can affect the performance of the `std::unordered_map`.
  - Hash function: chosen by STL; vary depending on different container types, different standard library version
  - Collision resolution: use separate chaining by default
- Map, unordered_map, set, unordered_set comparisons
  - unordered_map vs. unordered_set:

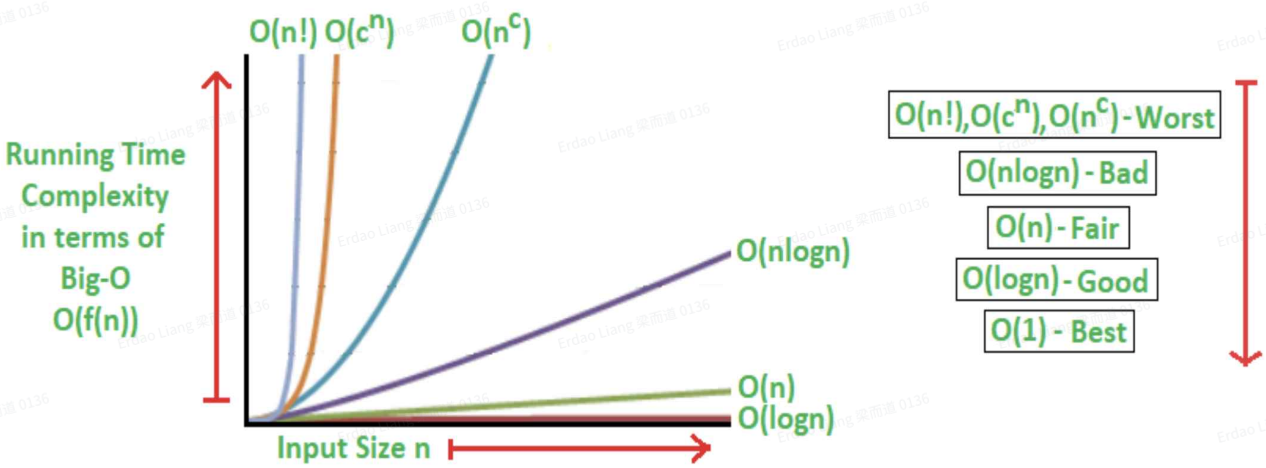
	unordered_map	unordered_set
What to store	Keys & values	Only keys, no value
When to use	implementing dictionaries, caches, or data structures that require fast key-based access.	checking for the existence of elements, maintaining a unique collection, and performing set operations like union, intersection, and difference.

- unordered_map vs. ordered_map
  - ordered_map is sorted -> complexities differ: O(log n) insertion and search

Complexity Analysis

Complexity Analysis

- Idea: a way to represent the asymptotic runtime of a program
- Terminology of Big-O notation
  - Big-O: an asymptotic upper bound to an algorithm.
  - Big-Ω: an asymptotic lower bound to an algorithm.
  - Big-Θ: an asymptotic tight bound to an algorithm.
- Asymptotic comparison



- Counting steps
  - For loop: initialization: 1, test: (num_of_loops + 1), update: num_of_loops

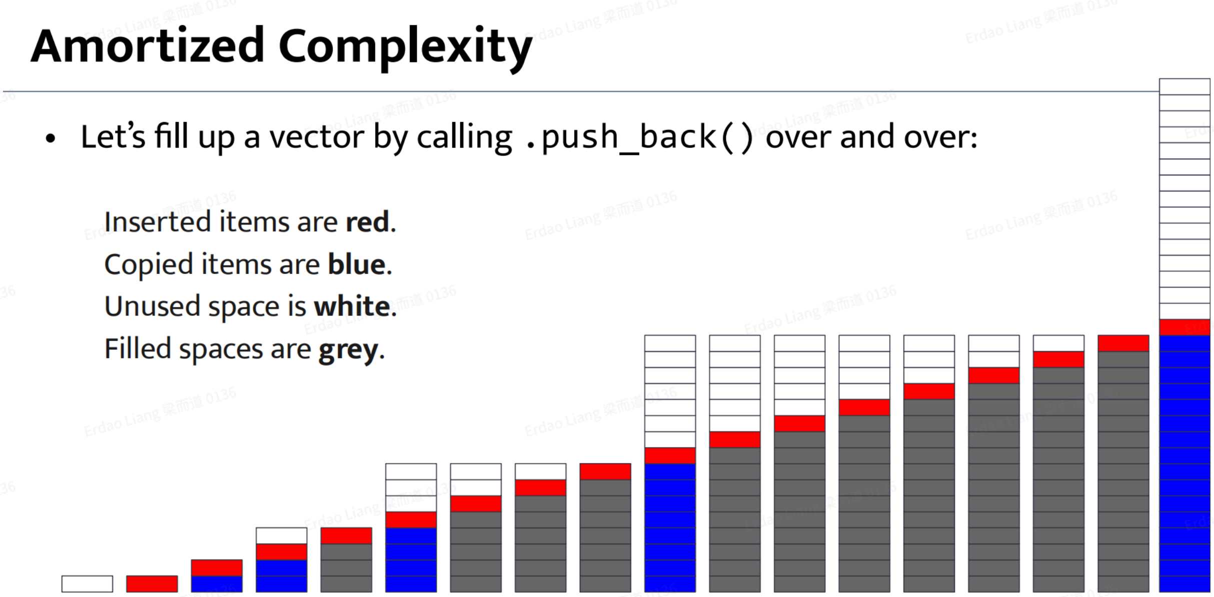
```
// polynomial
for (int i = 0; i < n; i++)

// log
for (int i = n; i > 1; i/2)
```

Amortized Complexity

- Idea: a principle to analyze the complexity of a program that worst cost is much worse than average cost
  - It is different from everage-case complexity

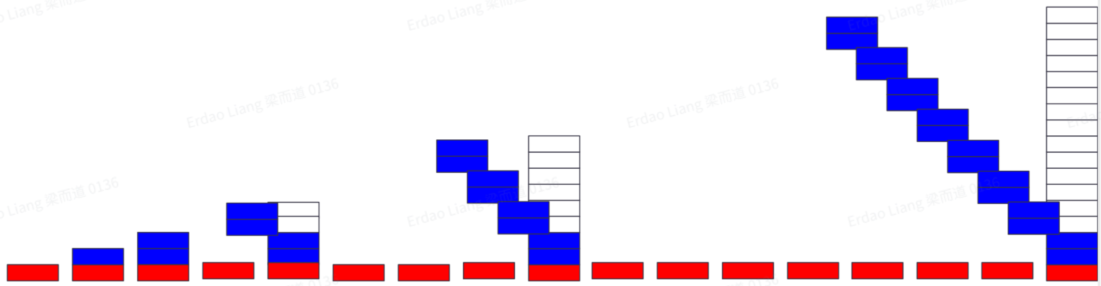
- Amortization:
  - I’m measuring the total cost of a sequence of operations.
  - Some of my operations are expensive, but the majority of my operations are cheap. So multiplying the largest cost by the number of operations gives a cost that is **too high**.
    - e.g. the worst case time complexity of pushing an element into a vector is $\Theta(n)$, but is the worst case time complexity of pushing n elements into a vector $\Theta(n^2)$, since you are doing a worst case $\Theta(n)$ operation n times? No!
  - When the excess work from the expensive is averaged out over the cheap operations, I find a more accurate upper bound for the complexity of that operation.

- Principle:



Amortized Complexity

- Now, we will **amortize the costs**! Let's spread the blue blocks around and see what happens...
- The total cost (number of blocks) stays the same.



- Application: constant growth & linear growth of vector capacity

Analyzing Complexity of Recursion

- Recursion
- Tail recursion: a recursion is tail recursion **if there is no pending computation at each recursive step**

```
// tail recursion
int factorial(int n, int res = 1){
        if (n == 0) return res;
        return factorial (n-1, res*n);
}
// not tail recursion
int factorial(int n){
        if (n==0)
                return 1;
        return n*factorial(n-1);
}
```

  - Advantages: The compiler will reuse the activation records instead of creating a new one - O(1) stack space

Recurrence Relation

- Idea: useful for describing the time complxity of a recursion function
- Identifying recurrence relations
- Solve recurrence relation

- **Master theorem** (not general)

  空白 TeX 公式

  - Master theorem with log factor
- **Substitution method** (general)
  1. Substitute the formula for T(n) into the recurrence terms on the RHS of the equation until a pattern is found
  2. Find a pattern that describes T(n) at the kth step
  3. Solve for k such that the base case is present on the RHS. This makes the recurrence easy to solve for in closed form because you know the value of your base case

- Common recurrence relation

Recurrence	Example	Big-O Solution
$T(n) = T(n / 2) + c$	Binary Search	$O(\log n)$
$T(n) = T(n - 1) + c$	Linear Search	$O(n)$
$T(n) = 2T(n / 2) + c$	Tree Traversal	$O(n)$
$T(n) = T(n - 1) + c_1 * n + c_2$	Selection/etc. Sorts	$O(n^2)$
$T(n) = 2T(n / 2) + c_1 * n + c_2$	Merge/Quick Sorts	$O(n \log n)$

- Example: 2D Table Search (different search method - different recurrence relation - different runtime)

C++

Data Types

Strings

- C strings:
- C++ strings: (STL)
  - Instantiation

    ```
    string str("string_content");
    ```

  - Functions:

        - iterator
        - Capacity (size, length, resize, reserve, clear)
        - Access: []
        - Operations: c_str, find, ...
    - .length(), .size() do the same thing

Pairs & Tuples

	Pairs	Tuples
	group two values	Group two or more values
Fetch	#include <utility>	#include <tuple>
Construction	• std::pair<std::string, int> pair1 = std::make_pair("eecs", 281); • std::pair<std::string, int> pair2{"eecs", 281}; • std::pair<std::string, int> pair3 = {"eecs", 281};	std::tuple<std::string, int, std::string, double> myTuple = std::make_tuple("eecs", 281, "paoletti", 3.14);
Access	.first, .second	get<*i*>(myTuple);
Comparison	Directly use expressions like (pair1 < pair2) Comparison starts from comparing the first elems, then the second in case of tie, (then the third, ...)	

C++ Features

Range-based for loop

```
for (int &item : array){
    // do something
}
```

Using

```
using type_name = std::vector<int>
```

Inline functions

- Making a function inline improves performance
- Case 1: if a function is specified as "inline", the compiler will consider it to be inline
- Case 2: if a class method has its definition written inside the class definition, it is automatically inline when compiled

Explicit

- Used in 1-parameter constructors to prevent implicit type conversion

```
explicit FeetInches(int feet); // ...

FeetInches a(3); // OK
FeetInches b = 3; // error
```

mutable

- Used in a class member variable
- Make it modifiable by a const member function

Functors

- Definition: A **function object, or functor**, is any type that implements operator().
    - Objects that can be called as if they were ordinary functions.
- Function objects provide two main advantages over a straight function call.
    - A function object can contain state
    - a function object is a type and therefore can be used as a template parameter.
- Application: Standard Library uses function objects primarily as sorting criteria for containers and in algorithms
    - More customizable than writing a comparison function
    - std::less<type> by default
- Example
    - compare two class objects by an ordinary comparison function

```
class Person {
    int age;
public:
```

```
    int get_age() const {return age;}
};

bool compare(Person &person1, Person &person2){
    return person1.get_age() < person2_get_age();
}

int main(){
    Person person1;
    Person person2;
    if (compare(person1, person2))
    cout << "Person1 is youngest";
    else
    cout << "Person2 is youngest";
}
```

- compare two class objects by using a functor

```
class Person {
    int age;
public:
    int get_age() const {return age;}
};
class PersonComparator {
public:
    bool operator()(const Person& p1, const Person& p2) const{
        return p1.get_age() < p2.get_age();
    }
};

int main(){
    Person person1;
    Person person2;
    PersonComparator my_functor;
    if (my_functor(person1, person2)) // can be called as if it is a
function
    cout << "Person1 is youngest";
    else
    cout << "Person2 is youngest";
}
```

Enum

```
enum class Nums{ zero, one, two, five = 5, hundred = 100};
```

- Declaring an enum type: rules
    - Each enumerator can be assigned a specific integer (can repeat)
    - If the first enumerator has no initializer, associated to 0
    - If a certain enumerator has no initializer, associated to previous + 1

Arrays in C/C++ (?)

Useful Libraries

- Randomization `<random>`
- Infinity `<limits>` - `numeric_limits<double>::infinity()`

STL

- Performance: best performance for general-purpose implementations
- Includes: Containers & iterators, Memory allocators, Function objects, Algorithms

Containers

- Container types

Sequential containers	vector<>, deque<>, list<>
Container adapters	stack<>, queue<> (underlying container default to be deque)
Associative containers	Map, set; unordered- version; multi- version

- In C++, when you declare a `std::vector<int>` in a function, the memory for the vector and its elements is typically allocated on the heap. This is because `std::vector` is a dynamic container, and it dynamically manages memory for its elements. The vector itself, which contains information such as the size, capacity, and a pointer to the dynamically allocated array of elements, is usually stored on the stack.

  vector Control Block (Metadata) in stack / dynamic array in heap

  in summary, the vector itself is a stack-allocated object that points to a dynamically allocated array of elements on the heap.

Iterators

- Iterators: Faster traversal

```
template <class InputIterator>
void genPrint(InputIterator begin, InputIterator end){
    while (begin != end){
        cout << *(begin++) << " " ;
    }
}
```

- Declaring an iterator
    - .begin(), .end() - just default iterators
    - .cbegin(), .cend() - const version of iterators
    - .rbegin(), .rend() - reverse iterators
    - std::begin(), std::end() for C arrays
- Types of iterators

	Input	Output	Forward	Bidirectional	Random
Supports dereference $(*)$ and read	✓		✓	✓	✓
Supports dereference $(*)$ and write		✓	✓	✓	✓
Supports forward movement $(++)$	✓	✓	✓	✓	✓
Supports backward movement $(--)$				✓	✓
Supports multiple passes			✓	✓	✓
Supports $==$ and $!=$	✓		✓	✓	✓
Supports pointer arithmetic $(+, -,$ etc.)					✓
Supports pointer comparison $(<, >,$ etc.)					✓

    - Different containers has different default iterators
    - Not all containers support all types of iterators

Container Type	Iterator Category
`std::vector<>`	Random Access
`std::deque<>`	Random Access
`std::string<>`	Random Access
`std::list<>`	Bidirectional
`std::set<>`	Bidirectional
`std::multiset<>`	Bidirectional
`std::map<>`	Bidirectional
`std::multimap<>`	Bidirectional
`std::unordered_set<>`	Forward
`std::unordered_multiset<>`	Forward
`std::unordered_map<>`	Forward
`std::unordered_multimap<>`	Forward
`std::forward_list<>`	Forward

Emplacement

- Problem:

for a container to insert/push a struct element, the following code call the struct's constructor twice (one default constructor, then one copy constructor to copy to the container): unnecessray work

```
struct Foo{
    int a;
    string b;
    Foo (int a_, string b_):a(a_), b(b_){}
    Foo (int a_): a(a_), b("awa") {}
}

vector<Foo> v;
v.push_back(Foo(1,"awa"));
```

- Solution:
  - Every STL container with push/insert support emplace() / emplace_back()
  - Have the container construct in-place, only call once; **parameters must match a constructor (any)**

```
vector<Foo> v;
v.emplace_back(1,"aa");
v.emplace_back(3);
```

Algorithm

Binary Search

```
int bsearch(double a[], double val, int left, int right){
    // search in [left,right)
    // suppose a[] is sorted in ascending order
    while (left < right){
        int mid = left + (right-left)/2;
        if (val == a[mid])
            return mid;
        if (val < a[mid])
            right = mid;
        else
```

```
            left = mid+1;
        }
        return -1;
    }
```

- Binary search in stl: (import <algorithm>)
  - binary_Search() returns a bool
  - lower_bound(): first item not less than target (return an iterator; return .end() if not found)
  - upper_bound(): first item greater than target

Sorting

- Considering a sort algorithm:
  - Time complexity
    - Worst
    - best (in what circumstances, why)?
  - Memory complexity?
  - Stable?
  - How adaptive is it on the inputs?
  - What cases is it suitable for?

Algorithm		Best case	Avg case	Worst case	Space	Stability
Elementary sort	Bubble	**O(n)**	O(n^2)	O(n^2)	O(1)	Yes
	Selection	O(n^2)	O(n^2)	O(n^2)	O(1)	No
	Insertion	**O(n)**	O(n^2)	O(n^2)	O(1)	Yes
Advanced sort	merge	O(nlog(n))	O(nlog(n))	O(nlog(n))	**O(n)**	Yes
	heap	O(nlog(n))	O(nlog(n))	O(nlog(n))	O(1)	**No**
	quick	O(nlog(n))	O(nlog(n))	**O(n^2)**	**O(log(n))**	**No**
	counting	O(n+k)	O(n+k)	O(n+k)	O(n+k)	Yes

- sorting in C++
  - #include <utility> swap(a,b)

    - #include <algorithm> std::sort(a,b), accepts two iterators
        - introsort, a combination of quick sort, heap sort, and insert sort
    - #include <algorithm> **std::nth_element**(a, a+n, b)
        - partially sort a range of elements such that the element at the nth position is in its sorted position if the range were fully sorted.
        - After calling `nth_element()`, the element at the `n`-th position (in this case, the 4th smallest element) is guaranteed to be in its correct sorted position within the range. You can access it using `numbers[n]`.
        - makes no guarantees about the relative order of other elements; the nth element is in its correct sorted position, and the elements to the left are less than or equal to it, while the elements to the right are greater than it
        - Runtime complexity O(n) (similar strategy to partition in quick sort)

Bubble sort

- Time complexity
    - Best case: O(n) (after optimization: keep a boolean swap flag) n comparison (only 1 outer iteration), ~n (0) swap
    - Worst case O(n^2), n^2 comparison n^2 swap
    - Quite adaptive
- Memory complexity: O(1)
- Stable: yes, in a single bubble process the latter one is not interchanged with the former one if they are equal
- Scenario: n small, nearly sorted
    - But consider [100 1 2 3 4 ... 99] do this in O(n^2)
    - Consider [2 3 4 5 ... 99 100 1] do this in O(n) (2 outer iterations)

Selection sort

- Time complexity
    - Best case: O(n^2) n^2 comparison, 0 swap (even when the array is already sorted, we still need n outer iterations and n inner iterations to determine the min)
    - WOrst case O(n^2), n^2 comparison **n-1 swap**
    - Nearly not adaptive at all
- Space capacity: O(1)
- Stable: no

- Scenario: good when auxiliary memory is limited;
  - Good when objects are large and copying is expensive

Insertion sort

- Time complexity
  - Best case: O(n): **n comparison** (1 comparison for all n outer iteration), ~n (0) swap
  - Worst case O(n^2), n^2 comparison n^2 swap
  - Very adaptive
- Space complexity O(1)
- Stable: yes, in a single insertion process the latter one is not interchanged with the former one if they are equal
- Scenario: the fastest algorithm on small input sizes
  - Good for nearly-sorted list: [100 1 2 3 4 ... 99] && [2 3 4 5 ... 99 100 1] both O(n)

Insertion sort and bubble sort are efficient on partially sorted data when each item is close to its final position

Insertion sort (but not bubble sort) is efficient on sorted data when a new item is added to the sorted data

Slecetion sort it expensive in comparsions but cheap in swaps

Merge sort

- Time complexity
  - Best case: O(nlogn): T(n) = 2T(n/2)+n (n for merge)
  - Worst case: also O(nlogn)
- Space complexity O(n): there must be a separate comtainer to store merged result
- Stable: yes (depend on Merge())
- Scenario: bad for large items

Heap sort

- Time complexity
  - Best case: O(nlogn)
  - Worst case: O(nlogn)

- Nearly not adaptive at all
- Space complexity O(1)
- Stable: No (jumping in fixUp and fixDown)
- Scenario
-

Quick sort

- Time complexity:
  - Best case: O(nlogn)
  - Worst case: O(n^2), n choosing of pivot (recursion call n times), n iterations per call (happens when either the max or the min element is chosen as pivot each time)
    - e.g. when the array is already sorted [1,2,3,4,5,6]
  - Very adaptive; the more uneven the partition, the worse the performance
    - The ideal pivot is the median element (but hard to find)
- Space complexity: O(log n) (for the function call stack)
- Stable: no
- Scenario: fastest algorithm in general

Counting sort

Alternations

- Lab question: 0-1-2 sort

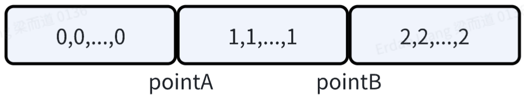
Counting sort

- Algorithm: two pass - first pass count the number of items - second pass copy records
- Only usable when the number of keys is limited and small
- Time complexity: O(n+k) (first pass + second pass)
- Space complexity: O(n+k) (result array + hash table)
- Not adaptive at all

Dutch National Flag Problem

- Only three values in the array: 0,1, or 2
- Sort it with O(n) time, **only one pass**, and O(1) additional memory

- Idea:
  - The sorted array looks like this



  - in that single pass, keep and move the current "pointA" and "pointB" (the end of "0" and the start of "2"); throw the 0 to pointA, throw the 2 to pointB, and move the pointer when necessary
- Code

```
void sort012(vector<int>& nums){
    int begin = 0, end = nums.size();
    int i = 0;
    while (i != end){
        // use while, because not every time i will be incremented
        if (nums[i] == 0){
            swap(nums[i], nums[begin]);
            begin++;
            i++;
            // the element that is swapped to position i can only be 1
        }
        else if (nums[i] == 2){
            swap(nums[i], nums[end];
            end--;
            // the element that is swapped to position i is either 0 or 1
           // so we do not ++i, because if that element is 0, we need to
stay here
           // and swap this 0 to the left in the next iteration
        }
        else{
            i++;
        }
    }
}
```

Backtracking and Branch&Bound

Brute force	Slow but gaurantee optimization
greedy	

Divide and conquer	**Non-overlapping** subproblems
backtracking	Constraint satisfaction problems
Branch and bound	Optimization problems
Dynamic programming	**Overlapping** subproblems

- Contraint satisfaction problems:
- Find a solution that satisfies given constraints, one solution is sufficient
- Rely on backtracking
- Optimization problems
- Cannot stop early, require a best solution
- Rely on branch and bound

- Generating permutaions

```
void gen_perms(vector<int> &items, int perm_length){
}
```

Dynamic Programming

- Sub problems are not independent
- Usually reduce from O(c^n) to O(n^c)
- **Memoization**
- Two methods: top-down & bottom-up

	top-down	Bottom up
	start with current problem, dig into smaller problems when necessary for a newly-encountered subproblem, store the result; for a subproblem that has met before, use the result directly	Start at samllest subproblem value build the subproblems up, until reaching the current value

	Implemented recursively	Implemented iteratively
	(Pros) only need to save value of needed subproblems - adaptive	(Cons) Must save all previously computed value (no matter whether it is actually needed) - may be not adaptive
	(cons) additional stack space needed (cons) no way to compactify memos, because don't know the order of evaluation beforehand	(pros) can recycle/collapse previous memo steps

Knapsack problem

- DP solution

```
uint32_t knapsackDP(const vector<Item> &items, const size_t m){
    const size_t n = items.size();
    vector<vector<uint32_t>> memo(n+1, vector<uint32_t>(m+1,0));

    for (size_t i = 0; i < n; i++){
        for (size_t j = 0; j < m+1; j++){
            if (j < items[i].size)
                memo[i+1][j] = memo[i][j]
            else
                memo[i+1][j] = max(memo[i][j], memo[i][j-items[i].size] +
items[i].value);
        }
    }

    return memo[n][m]; // right bottom corner
}
```

    - Complexity: O(NM)
- Reconstructing the solution
    - If a smaller solution plus an item is greater than or equal to a full solution without the item, it is included, otherwise excluded

```
vector<bool> knapDPReconstruct(const vector<Item> &items,
    const vector<vector<uint32_t>> &memo, const size_t m){

    const size_t n = items.size();
    size_t c= m; // current capacity
```

```
    vector<bool> taken(n, false);

    for (int i = n-1; i >=0; i--){
        if (items[i].size <= c){
            // if item fits
            if (memo[i][c - items[i].size] + items[i].value >= memo[i+1][c]){
                // if item added
                taken[i] = true;
                c -= items[i].size;
            }
        }
    }
    return taken;
}
```

- Complexity: O(N)