Algorithms

Algorithms is a set of steps to accomplish a task. A computer algorithm has the following properties:

- Precisely defined steps/conditions
- Defined input
- Defined output
- Correct (exactly correct or correct with some error)
- Terminates within a reasonable time

Algorithms are measured via:

- **Efficiency -** how do the time/space requirements of the algorithm scale with the input?
- **Correctness -** algorithm returns the desired output every time, for every possible input

Classifying algorithms
Algorithms can be classified by:

- Task - numeric, sorting, searching, routing, scheduling etc.
- Approach - brute force, divide & conquer, decrease & conquer, greedy etc.
- Solution type - exact, approximate, heuristic

Efficiency

Efficiency changes as the size of the input grows. It is important to:

- Understand how problems scale with input
- Choose which algorithm to use for a problem
- Know when a problem is intractable (not solvable in a reasonable time)

e.g. Fibonacci numbers $F_n = F_{n-1} + F_{n-2}$

Recursive Fibonacci in C:

```
int fib(int n){
    if (n==0) return 0;
    if (n==1) return 1;
    return fib(n-1) + fib(n-2);
}
```

Estimating efficiency

Efficiency is a property of the algorithm and rather than the hardware running, so efficiency is measured in steps rather than time. To compute efficiency:

1. Count operations instead of time
2. Express as a function of input size
3. T(N) = number of operations needed for input of size n

- Look for the most expensive operation
- Always be aware of the assumptions - are the operations really constant and what are the inputs and outputs?

For Fibonacci, adding two numbers takes constant time, but only true if both numbers fit within a word (32 bits or number < 2^{32}). Fibonacci numbers can get very large, therefore assumption not valid for large n (n>46).

e.g. Recursive Fibonacci numbers.

Memoization

Store previously computed values - saves time and space.

```
int fib(int n) {
    int result = 0;
    int preOldResult = 1;
    int oldResult = 1;

    if(n <= 0) return 0;
    if(n > 0 && n < 3) return 1;

    for( int i = 3; i <= n; i++ ){
        result = preOldResult + oldResult;
        preOldResult = oldResult; //Keeping last 2 results
        oldResult = result;
    }
    return result;
}
```

Complexity Analysis

Big-O Notation
$O(g(n))$ is a set of functions or upper-bound, including all functions that have the same or smaller growth rate

- Allows the growth rate of functions to be compared in a way that is independent of their numeric values

For functions $f(n)$ and $g(n)$, $f(n)$ is in $O(g(n))$ (in big O of g) if there are constants c and N such that:

$$f(n) < c \times g(n), \forall\, n > N_0$$

e.g. $T(n) = n^2 + 2n + 8$ is in $O(n^2)$
  - $n^2 + 2n + 8 < 11n^2, \forall\, n > 1$ (setting c = 11)

- Interested in asymptotic behaviour - as n approaches infinity
- When n is large, the highest order term always dominates
- General rule: ignore coefficients and drop lower-order terms

Common Big-O classes

Order	Examples
1	Constant-time operation that does not depend on n
$\log(n)$	Binary search
n	Looping through n items
$n\log(n)$	Efficient sorting algorithms
n^2	Pairwise comparisons, loop within loop
2^n	All possible subsets
$n!$	All possible permutations

Big-Omega
A lower bound of functions. $f(n)$ is $\Omega(g(n))$ if:

$g(n)$ is $O(f(n))$

e.g. $T(n) = n^2$ in $\Omega(n)$
  - $n < n^2, \forall\, n > 0$ (setting c = 1)

Big-Theta
A tighter bound of functions. $f(n)$ is $\Theta(g(n))$ when:

$f(n)$ is $O(g(n))$ **and** $f(n)$ is $\Omega(g(n))$ or
$f(n)$ is $O(g(n))$ **and** $g(n)$ is $O(f(n))$ or

e.g. $T(n) = 2n^2 + 4$ in $\Theta(n^2)$
  - $2n^2 + 4 < 6n^2, \forall\, n > 1$ (setting c = 6)

        - $n^2 < 1 \times (2n^2 + 4), \forall\, n > 0$ (setting c = 0)

Note: complexity analysis often focuses on the worst-case scenario, but best and average case are also informative.

Pointers

Random access memory and pointers
Memory is stored in bytes, and a computer can access any location of memory to read or write data.

A pointer is a type of variables that stores the memory address of another variable. The addresses of these variables are printed as base-16 hexadecimal integers.

- Pointers with different types cannot be assigned together as each type takes a different number of bytes on the memory

Operator	Name	Returns
*	Value at	Value stored at the address
&	Address of	Address of the variable

e.g.

```
int i, j=10;
int *ptr;
ptr = &j; // ptr == 1004
i = *ptr; // i == 10
```

Pointers as arguments
When a function needs to alter its arguments, it should be designed to receive pointers. When called, any functions with pointers should be sent an address `&a`.

e.g. Swapping two variables.

```
void int_swap(int *p1, int *p2);

int
main(int argc, char *argv[]) {
    int x=2, y=3, z=4;
    printf("main: x=%2d, y=%2d, z=%2d\n", x, y, z);
    int_swap(&x, &y);
    printf("main: x=%2d, y=%2d, z=%2d\n", x, y, z);
    return 0;
}

void
int_swap(int *p1, int *p2) {
    int tmp;
    tmp = *p1;
    *p1 = *p2;
    *p2 = tmp;
}
>>> x= 2, y= 3, z= 4
>>> x= 3, y= 2, z= 4
```

Dynamic Memory

Each declaration in C takes up a certain number of bytes. Dynamic arrays allow runtime memory to be efficiently used, as programs can adjust their memory consumption to the scale of the input data without needing to be recompiled.

When variables are initialised, these declarations take up a certain area in memory which is a multiple of its number of bytes.

Declaration	Bytes
`char`	1
`int`	4
`float`	4
`double`	8

There are various memory management functions provided in C.

Function	**Purpose**
`size_t sizeof()`	Returns number of bytes required to store the type or variable
`(type*)malloc(size_t size)`	Allocates a segment of memory containing size bytes and return a pointer to this segment, or NULL if not allocated.
`(void*)calloc(size_t num, size_t size)`	Allocates a fresh segment of memory, initializing all bits to 0.
`void free(void *ptr)`	Deallocates the segment of memory indicated by ptr and returns it to the pool of available memory. • Each call to `malloc` should be matched by a later call to `free` to prevent a memory leak
`void *realloc(void *ptr, size_t size)`	Used to relocate information from a previously allocated block of memory to a new larger or smaller memory block. • The first array must be created with malloc • Values are copied and reallocated

Memory allocation - standard recipe.

```
type_t *ptr;

/* figure how big the array needs to be */
n = ... ;

/* and ask for the right amount of space */
ptr = (type_t*)malloc(n*sizeof(*tptr));
assert(tptr);

...

/* free the memory */
```

```
free(tptr);
tptr = NULL;
```

Dynamic Memory - getword example

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>

#define MAXCHARS 1000    /* max chars per word */
#define INITIAL   100    /* initial size of word array */

typedef char word_t[MAXCHARS+1];
int getword(word_t W, int limit);
void exit_if_null(void *ptr, char *msg);

int
main(int argc, char *argv[]) {
    word_t one_word;
    char **all_words; // equivalent to (char*)[]
    size_t current_size=INITIAL;
    int numdistinct=0, totwords=0, i, found;

    all_words = (char**)malloc(INITIAL*sizeof(*all_words));
    assert(all_words)

    /* iterating over inputted words */
    while (getword(one_word, MAXCHARS) != EOF) {
        totwords = totwords+1;

        /* linear search in array of previous words */
        found = 0;
        for (i=0; i<numdistinct && !found; i++) {
            found = (strcmp(one_word, all_words[i]) == 0);
        }

        /* if new word exists, assign */
        if (!found) {

            /* a new word exists, but is there space? */
            if (numdistinct == current_size) {
                current_size *= 2; // Always multiply size to minimise 
iteration time

                all_words = realloc(all_words, 
current_size*sizeof(*all_words));
                assert(all_words);
            }

            /* there is definitely space in the array. 1+ allows for null
                byte */
            all_words[numdistinct] = (char*)malloc(1+strlen(one_word));
            assert(all_words)

            /* and there is also a space for the new word */
            strcpy(all_words[numdistinct], one_word);
            numdistinct += 1;
        }
    }

    printf("%d words read\n", totwords);
    for (i=0; i<numdistinct; i++) {
        printf("word #%d is \"%s\"\n", i, all_words[i]);
        free(all_words[i]); 
        all_words[i] = NULL;
    }
    free(all_words);
    all_words = NULL;
    return 0;
}

#include "getword.c"
```

```
>>> Mary had a little lamb, little lamb, little lamb,
>>> Mary had a little fourleggedwhitefluffything.
14 words read
word
```

Data Structure Types

Two types of data structures:

- **Abstract data type -** describes what it does, but does not specify an implementation e.g. stack, queue, dictionary
  - Must separate the implementation from the concept
  - Nothing outside of definitions of the data type should refer to anything inside
- **Concrete data structure -** implements an abstract data type e.g. array, linked list, tree

Abstract data structures

- FIFO/queue (first-in first-out) - if insert at tail and extract from head
- LIFO/stack (last-in first-out,) - if insert at head and extract from head
- Dictionary - access items by key

Arrays

A concrete data structure that stores a group of items (a pointer constant).

- Fixed number of items, all the same type
- Any position can be accessed in constant time via an index
- e.g. `int A[5]` will reserve a block of memory large enough for 5 integers (20 bytes)

Accessing arrays

Since arrays are a pointer constant, there are various ways to access a value of the array:

```
A[i] = *(A+0)
```

Complexity analysis

- Reading or writing one item: $O(1)$
- Building an array of n items: $O(n)$
- Finding an item in an unsorted array: $O(n)$ (worst-case scenario check all items)

Building and searching

- Unsorted array, linear search:
  1. If there are n items, constant time to build - $O(n)$
  2. If there are m lookups, n steps to search - $O(m \times n)$
  3. $O(n + m \times n) = O(mn)$
- Sorted array, binary search:
  1. If there are n items, n comparisons and n data movements to insert - $O(n^2)$
  2. If there are m lookups, log n steps to search - $m \times \log_2 n$
  3. $O\left(n^2 + m \times \log(n)\right) = O\left(n^2\right)$

Sorted array better for small n and large m, unsorted array better for large n and small m.

Circular Array

- Concrete data structure
- Keep track of start and end indices
- Delete at start index and move start along
- Insert at end index and increment by one, then get remainder from dividing by array size
- If run out of space and add another value, copy from start to end in order to new array

Structures

Unlike an array, a structure combines underlying variables of differing types, with individual elements identified by component name.

Declaring a single structure variable

```
    struct {
        type1 var1;
        type2 var2;
    } var_t;
```

Naming structures for later use by suppling a tag

```
    struct tag_t {
        type1 comp1;
        dec2 comp2;
    };
    struct tag_t var;
```

Usually it is common practice to create a full type name for the structure being presented, and the resulting `tag_t` can be used as a declaration.

```
    typedef struct {
        type1 comp1;
        type2 comp2;
    } tag_t;
    tag_t var = ...;
```

Operations

Variables in a structure are accessed by the '.' selection operator.

```
    type(tag_t) var.component = ... ;
```

Unlike arrays, structures of the same type can be assigned in the same way as variables, but not compared.

- Structures are read and written component by component, as all elements are different types
- Structure declarations can make use of other structures
- An array of structures with declaration `tag_t` can be created to store data, they are passed into functions with `tag_t` as the declaration
- Structures of one type cannot be cast into another

Pointers and functions

When a structure is passed into a function, it is copied into a local argument variable. Therefore, it is usual to pass structure pointers to functions instead to alter the components e.g. `termdeposit_t *d`.

When a variable of structure type is dereference, it yields an object of type `tag_t`.

To access a component's field, the expression `pointer->component` is used i.e. "the component variable of the structure pointed at by pointer". Also expressed as `(*pointer).component`

- It is usually desirable to pass a pointer instead as it allows the function to alter the components in the underlying variable e.g.

```
int
read_planet_ptr(planet_t *planet) {
    int nvals_read;
    printf("Enter %s:\n", PLANETPROMPT);
    nvals_read = scanf("%s %s %lf %lf %lf",
        planet->name,
        planet->orbits,
        &planet->distance,
        &planet->mass,
        &planet->radius);
    if (nvals_read != 5) {
        return EOF;
    } else {
        return 0;
    }
}
```

instead of:

```
planet_t
read_planet(void) {
    planet_t new_planet;
    printf("Enter %s:\n", PLANETPROMPT);
    scanf("%s %s %lf %lf %lf",
        new_planet.name,
        new_planet.orbits,
        &new_planet.distance,
        &new_planet.mass,
        &new_planet.radius);
    return new_planet;
}
```

Summary

	Array	Structure
Assigned (=)	No	Yes
Compared (==)	Yes (as pointer)	No
Argument to function	Yes (as pointer)	Yes (copied)
Returned from function	Yes (as pointer)	Yes (copied)
Take address of (&)	No (is already)	Yes
Use as pointer(, [])*	Yes	No

Lists and Linear Linked Structures

A linked list is a one-dimensional data structure in which objects are threaded together using a pointer in each node.

```
typedef struct node node_t;

struct node{
    data_t data;
    node_t *next;
};

typedef struct {
     node_t *head;
     node_t *foot;
} list_t;
```

- Type `node_t` is a recursive data structure
- The base case is a `next` that has the value NULL
- The first pointer in the chain must be a variable, but nodes thereafter can be created through malloc
- Variable `list_t` is of type struct with a head and foot node (start and end of nodes)
- `list_t` is the list, whereas `node_t` are the individual elements within the list

List generation and removal functions

Processing

Generate empty list

```
list_t
*make_empty_list(void) {
    list_t *list;

    /* allocate memory for list*/
    list = (list_t*)malloc(sizeof(*list));
    assert(list!=NULL);

    /* set head and foot as NULL */
    list->head = list->foot = NULL;
    return list;
}
```

Determine if list is empty

Free the list from memory

List Manipulation Functions

Insert node at head

```
list_t
*insert_at_head(list_t *list, data_t value) {
    /* create new node and allocate memory */
    node_t *new;
    new = (node_t*)malloc(sizeof(*new));
    assert(list!=NULL && new!=NULL);

    new->data = value; // assign value to data of new node
    new->next = list->head; // assign head of list to be next of new
node
    list->head = new; // assign new node to be head of list

    if (list->foot==NULL) {
        /* this is the first insertion into the list */
        list->foot = new;
    }
    return list;
}
```

Insert node at foot

```
list_t
*insert_at_foot(list_t *list, data_t value) {
    /* create new node and allocate memory */
    node_t *new;
    new = (node_t*)malloc(sizeof(*new));
    assert(list!=NULL && new!=NULL);

    new->data = value; // assign value to data of new node
    new->next = NULL; // assign NULL to next of new node, as this is
the end

    if (list->foot==NULL) {
        /* this is the first insertion into the list */
        list->head = list->foot = new;
    } else {
        /* assign next of last node in list to be new node */
        list->foot->next = new;
        list->foot = new;
    }
    return
```

Get value of head

Get tail of list (delete first node)

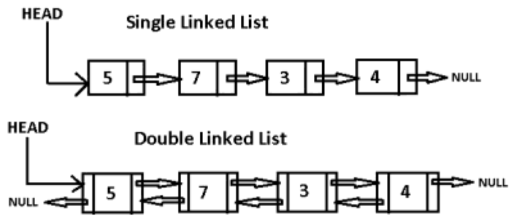
```
list_t
*get_tail(list_t *list) {
    node_t *oldhead;
    assert(list!=NULL && list->head!=NULL);

    oldhead = list->head;
    list->head = list->head->next; // get second node in list

    if (list->head==NULL) {
        /* the only list node just got deleted */
        list->foot = NULL;
    }

    free(oldhead);
    return list;
}
```

Linked List Details

Linked lists

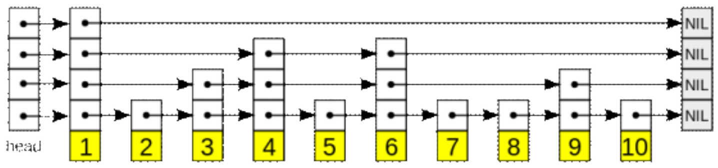
There are two types of linked lists:

- Singly linked list: each node has a pointer to the next node
- Doubly linked list: each node has a pointer to next node and a pointer to previous node



Skip lists

Linked list with lots of extra pointers ($O(\log n)$) search but tricky to build (probabilistically).



Stacks and queues

- Stack or LIFO (last-in first-out, or stack) - insert at head and extract from head
  - Dynamic arrays, linked lists
- Queue or FIFO (first-in first-out) - insert at tail and extract from head
  - Dynamic arrays, linked lists, circular arrays
- Push - add item to top of stack
- Pop - take item from top of stack

Linked List vs Arrays

Arrays

Advantages	**Disadvantages**
• Compact • Convenient to access • O(1) access to any item	• Items need to be of same type • Fixed size - static structure

Linked lists

Each item is located in an arbitrary place in memory with a pointer to the next item

Advantages	**Disadvantages**
• Flexible - can resize list as needed • Easy to insert or delete new items anywhere by rearranging links	• No random access to items - must traverse list to access items • Takes up more space and extra time

Action/complexity time	**Array**	**Linked list**
Space	$O(n)$	$O(n)$
Time	$O(n)$	$O(n)$
Time: search unsorted	$O(n)$	$O(n)$
Time: search sorted	$O(\log n)$	$O(n)$
Time: build sorted	$O(n^2)$	$O(n^2)$

	Static Array	Dynamic Array	Sorted dynamic array	Linked list
Insert	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Search	$O(n)$	$O(n)$	$O(\log n)$	$O(n)$

Binary Search Trees

```
struct node {
    record r;
    struct node *left;
    struct node *right;
    struct node *parent;
};
```

A binary tree is a two-dimensional data structure in which objects are threaded using two pointers in each node.

- Objects are ordered from left to right across the tree
- A binary tree is complete if every level except (optionally) the last is completely filled with all nodes as far left as possible
- When inserting/searching, start from top of tree and determine if number is smaller/larger, and traverse down according to left (smaller) or right (larger)

Time complexity

	Height	Root to node longest	Root to node average
Best case (balanced)	$\log_2 n$	$\log_2 n$	$\log_2 n$
Worst case (linked list)	n	n	$\frac{n}{2}$

Traversal and Deletion

Traversal

Recursive in-order traversal	Print all nodes in key order. • Good for sorting	`traverse(struct node *t){ if(t != NULL){ traverse(t->left); visit(t); traverse(t->right); } }`
Pre-order traversal	Do something at current node then recurse on left and right nodes. • Good for copying tree	`traverse(struct node *t){ if(t != NULL){ visit(t); traverse(t->left); traverse(t->right); } }`
Post-order traversal	Recurse on left and right nodes, then do something at current node. • Good for freeing	`traverse(struct node *t){ if(t != NULL){ traverse(t->left); traverse(t->right); visit(t); } }`

Deletion

Case 1: node is a leaf	Just delete the node
Case 2: node has one child	Replace node with the child
Case 3a: case has two children with one child having no children	Replace node with the childless child
Case 3b: case has two children and both children have children	Replace node with either in-order successor or in-order predecessor

Steps

1. Find the node to be deleted
2. Delete it
3. Replaced based on case

Time complexity

	Find node	Find in-order predecessor/successor	**Total time**
Worst case	$O(n)$	$O(n)$	$O(n)$
Average case	$O(\log n)$	$O(\log n)$	$O(\log n)$

AVL Trees

Balanced trees keep the height of the tree $O(\log n)$ and should not increase time complexity to build tree.

- Add steps during insertion to ensure tree doesn't become unbalanced
- Order preserved: left child < parent < right child

AVL trees (Adelson-Velskii & Landis)

Insert node and keep track of height of subtrees of every node using a balance factor.

- **Balance factor** difference between height of left and right (left-right), it is balanced only if the factor of all nodes is <= 1
- Non-AVL trees are caused by outside or inside insertions

Balancing trees

Trees can be balanced using rotations.

- Outside imbalance:
    - Rotate to rebalance
    - e.g. left subtree > right subtree: rotate right
- Inside imbalance:
    - Similar, but requires two rotations

Rotations

Right rotation	
	```
	RotateR(node){
	    //Auxiliary variables
	    left = node.Left;
	    leftRight = left.Right;
	    parent = node.Parent;

	    //Operations
	    left.Parent = parent;
	    left.Right = node;
	    node.Left = leftRight;
	    node.Parent = left;
	}
	```

Left rotation	
	```
	RotateL(node){
	    //Auxiliary variables
	    right = node.Right;
	    rightLeft = right.Light;
	    parent = node.Parent;

	    //Operations
	    right.Parent = parent;
	    right.Left = node;
	    node.Right = rightLeft;
	    node.Parent = right;
	}
	```

Insertion

```
node* insert(node*tree, node*new_node){
    if(tree == NULL){
        tree = new_node;
    }else if(new_node->key < tree->key) {
        tree->left = insert(tree->left, new_node);
        /* Fifty lines of left balancing code */
    }
    else{
        tree->right = insert(tree->right, new_node);
        /* Fifty lines of right balancing code */
    }
    return tree;
}
```

Advantages	**Disadvantages**
• Tree is always nearly balanced • Height < $1.44 \log_2 n$ • Complexity for any search is $O(\log n)$	• Must keep track of insertion path and height • Balancing adds time to insertion (constant time)

Distribution Counting

A stable sorting technique based on keys between a specific range.

- **Stable sort -** maintain the relative order of records with equal keys (values)

1. Take a count array to store the count of each unique object - index = actual value in original array and value = number of times this value occurs
2. Modify count array to contain the cumulative sum starting from index 1 - each element at each index stores the sum of previous counts
3. Modified array indicates the position of each object in output sequence
4. Traverse original array copying each item to its position and then increasing the cumulative count by one

Complexity analysis

Time	$O(n) + O(range)$
Space	$O(n) + O(range)$

Drawbacks

- Takes extra space
- Generally less flexible than comparison-based

Dictionaries and Hashing

Dictionaries
An abstract data type that defines an unordered collection of data as a set of key-value pairs.

- So far have been based on key comparisons - linked list, array, BST etc.
- Best search performance is $O(\log n)$

Hashing
The main idea of hashing is to take each key, and use a hash function to deterministically construct a seemingly random integer in a constrained range.

- Dictionary implementation with mostly $O(1)$ search (only if not too full)
- Data structure is an array
- e.g. Python's dictionary data type is a hash tabel

Implementation
A common implementation is using a circular array, squashing keys to fit into an array f limited size and storing each key in `A[key%mod]`

- For modulo mapping, prime numbers are better to disrupt patterns
- Using a function that maps keys evenly to the decided range
- Having a method to handle collisions e.g. using modulo 10 keys 1, 11, 21 all map to A[1]

Hash functions
A function that can be used to map data of arbitrary size to fixed-size values.

- Maps key to an array slot `A[hash(item->key)] = item;`
- Output value must be within bounds of array
- Should minimise collusions by spreading keys evenly throughout the table
- A popular and effective hash function is MD5

e.g. `A[97]; hash(key) = (key * BIGPRIME)%97`

Hashing for strings
Usually will use string to number mapping.

- Efficient to cumpte
- Unique value for each string
- Not guaranteed to break up patterns

Collision Handling & Hashing Properties

When a location indicated by the hash function is already occupied.

- When array size < number of records, collisions are guaranteed
- For converse, collisions still possible

Linear chaining	Implementing collisions as a linked list
Linear probing	Moves to the next empty element in the hash table
Double hashing	Moving to an empty element by adding on another key to the failed index until an empty element (hash1(key) + hash2(key))

Open addressing analysis (linear probing & double hashing)
Measured using load factor:

$$\alpha = n/m$$

load factor = number of keys / number of cells

	Linear probing	Double hashing
Insertion	$\frac{1}{(1-\alpha)^2}$	$\frac{1}{1-\alpha}$
Lookup	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)}\right)$

Chaining vs open addressing

- Space: chaining requires extra space for linked list nodes whereas open addressing requires extra space in table
- Time: similar for lightly loaded table, but open addressing is slower as α approaches 1

Hash table properties

- Performance degrades as table fills up
    - Open addressing tables must be reallocated - hash must be recomputed
    - Open addressing degrades much more rapidly when table is full
- No efficient way to retrieve items in sorted order
- Good hash functions are typically slow
- Also used for file verification, efficient duplicate/plagiarism detection, encryption

Sorting, Selection and Insertion Sort

Methods to sort

Non-comparison sorting	Comparison sorting
• Maps item to sorted positions • Requires known (generally small) range	• Only reading the list elements through a single abstract comparison

Selection sort

Finding the smallest element in the array and putting it at the beginning of the array, and repeating process until all sorted.

```
void selection(item* A, int n){
      int i, j, min;
      for( i = 0; i < n-1; i++ ){
            min = i;
            for( j = i+1; j < n; j++ ){
                  if( A[j] < A[min] ) min = j;
            }
            swap( &(A[i]), &(A[min]) );
      }
}
```

- Worst case complexity $O(n^2)$
- Releatively few swaps - each item only moved once
- $O(n)$ number of swaps
- Useful when moving items in memory is expensive

Insertion sort

Iterating through the array, compare each element to its previous elements and keep swapping to find the smallest position possible.

```
void insertion(item* A, int n){
      int i,j,val;
      for( i=1; i < n; i++ ){
            val = A[i]; j=i;
            /* swap A[i] left into correct position if j is
                  bigger than right of j */
            while( j > 0 && A[j-1] > val ){
                  A[j] = A[j-1]; j--;
            }
            A[j] = val;
      }
}
```

- Worst case complexity $O(n^2)$
- Useful due to few steps, and generally outperforms $O(n \log n)$ algorithms when n is small

Quicksort & Divid and Conquer

Divide instance of problems into smaller instances, and solve smaller instances - usually recursively.

- Generally complexity of $O(n \log n)$

There are two main approaches:

- Sorting while splitting, use a trivial method to join groups e.g. quicksort
- Use a trivial method to split, sort while joining groups e.g. mergesort

Quicksort
Splits an array into three sections by partitioning the elements relative to a pivot value, then recursively sorting two of the partitions.

- Partition array:
    - Choose a pivot (simple choice: first item in array)
    - Step through array, items less than pivot go to start, items more than pivot go to end
    - Pivot goes in between those sets
- Recursion:
    - Partition left-half recursively
    - Partition right-half recursively
    - Base case: singletons are already sorted

```
void quicksort(item A[], int l, int r){
      int i;
      if (r <= l) return;
      i = partition(A,l,r);
      quicksort(A,l,i-1);
      quicksort(A,i+1,r);
}

int partition(item A[], int left, int right){
      int i = left, j = right+1;
      int v = A[left];
      while( 1 ){
            while(A[++i] < v) /* do nothing */;
            while(A[--j] > v) /* do nothing */;

            if(i >= j) break;

            swap(A[i], A[j]);
      }
      swap(A[left], A[j]);
      return(i);
}
```

Advantages	Disadvantages
• Average case complexity of $O(n \log n)$ • In-place sort, no extra space required	• Worst case complexity $O(n^2)$ • Requires random access

For even partitioning - $\Theta(n)$ for n items

Even partioning, halfed input each step $\Theta(n \log n)$	Unbalanced partition, n-1 items remain $\Theta(n^2)$

Mergesort

Break the data into small data sets, sort the smaller sorts and merge the resulting sorted lists together.

1. If list has one element, return
2. Split list into two equal-sized pieces (recurisvely)
3. Sort each half
4. Merge two sorted halves

Implementation

```
/* n is size of A, m size of B */
merge(item C[], item A[], item B[], int n, int m){
      int i,j,k;
      for( i=0,j=0,k=0; k < n+m; k++ ){
            /* shortcut at the end of A or B*/
            if(i == n){
                  C[k] = B[j];
                  j++;
                  continue;
            }
            if(j == m){
                  C[k] = A[i];
                  i++;
                  continue;
            }
            /* copy smaller value to new array */
            if(A[i] <= B[j]){
                  C[k] = A[i];
                  i++;
            } else{
                  C[k] = B[j];
                  j++;
            }
      }
}
```

Top-down recursive
Break up until we get down to base case (stack).

```
int *mergesort(A, first, last){
      if(first < last){
            int i;
            item B[], item C[];
            mid = (int)(last-first+1) / 2;
            /* take original array, put half in A, put half in C */
            for(i=0;i<mid;i++){
                  B[i] = A[i];
            }
            for(i=mid;i<=last;i++){
                  C[i-mid] = A[i];
            }
            B = mergesort(B,0,mid-1);
            C = mergesort(C,0,mid-1);
            A = merge(B,C);
      }
      return A;
}
```

Bottom-up mergesort

Start from single itmes and join back together (queue).

1. Break list into n singleton lists
2. Insert single lists into a queue
3. deQueue first two items, merge them and enQueue them

Analysis

Cost of sorting n items = 2 * cost of sorting n/2 items + merge n items.
Approximate n as power of 2:

$$\begin{aligned} C(n) &= 2 \times C\left(\frac{n}{2}\right) + n - 1 \\ &= 2 \times \left[2 \times C\left(\frac{n}{4}\right) + \left(\frac{n}{2} - 1\right)\right] + (n-1) \\ &= 4 \times C\left(\frac{n}{4}\right) + (n-2) + (n-1) \\ &= 8 \times C\left(\frac{n}{8}\right) + (n-4) + (n-2) + (n-1) \\ &= \log_2 n \\ &= 2^{\log n} + n + \cdots < \log n \; times > \cdots + n \\ &\approx n \log n \end{aligned}$$

Properties

- Worst and average case $\Theta(n \log n)$
- Stable
- Can sort huge files on disk
- Slower than quicksort in practice

Space complexity

Bottom-up array (space for joining list)	$\Theta(n)$
Bottom-up linked list (lists in queue)	$\Theta(n)$
Top-down array (space for joining list)	$\Theta(n)$
Top-down linked list (other lists in stack)	$\Theta(\log n)$

Sorting Complexity Summary

	Best	**Average**	**Worst**
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Mergesort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$

Master Theorem

The master theorem (for divide-and-conquer recurrences) provides an asymptotic analysis for recurrence relations of types that use divde and conquer.

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- $f(n) \in \Theta\left(n^d\right)$
- $a \geq 1$: number of sub problems after a division
- $b > 1$: decrease of size of subproblems
- d: number of operations per element

Cases

if $d > \log_b a$	$T(n) \in \Theta(n^d)$
if $d = \log_b a$	$T(n) \in \Theta(n^d \log n)$
if $d < \log_b a$	$T(n) \in \Theta(n^{\log_b a})$

Derivation

Size of subproblems decreases by b

- Base case reached after $\log_b n$ levels
- Recursion tree $\log_b n$ levels

Branch factor is a

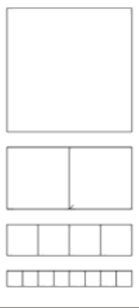
- At kth level, have a^k subproblems

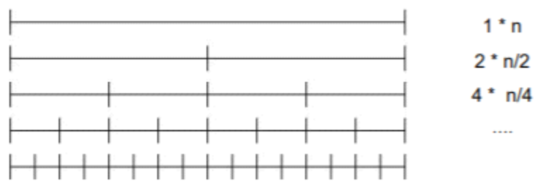
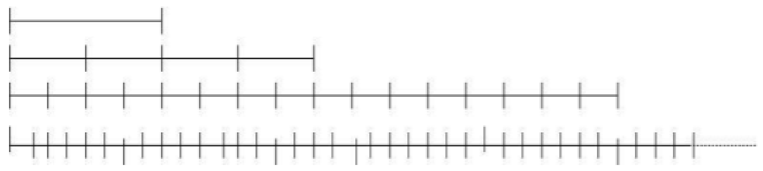
At level k, total work is:

- $a^k \times O\left(\frac{n}{b^k}\right)^d$
- # subproblems x cost of solving one

  $O\left(n^d\right) \times \left(\frac{a}{b^d}\right)^k$
    - As k (levels) goes from 0 to $\log_b n$, becomes a geometric series

Cases

$\frac{a}{b^d} < 1$ or $d > \log_b a$	$\left(\frac{a}{b^d}\right)^k$ gets smaller as k goes from 1 to $\log n$ • First term is largest amongst series, and is < 1 • Therefore $O\left(n^d\right)$ 
$\frac{a}{b^d} = 1$ or $d = \log_b a$	Ratio remains constant, so series is $O\left(n^d\right) + \cdots + O\left(n^d\right)$ • For $\log_b n$ levels • Therefore $O\left(n^d \log n\right)$

	
$\frac{a}{b^d} > 1$ or $d < \log_b a$	Series is increasing. • Sum dominated by last term $O(n^d)\left(\frac{a}{b^d}\right)^{\log_b n}$ • Therefore $O(n^{\log_b a})$ 

Priority Queue

A data type similar to a regular queue in which each element additionally has a priority associated with it - element with highest priority is served first.

Operations

Queues	**Priority Queues**
`makeQ();`	`makePQ();`
`enQ(Q,item);`	`enQ(PQ,item);`
`deQ(Q,first);`	`deletemax(PQ);`
`emptyQ(Q);`	`emptyPQ(PQ);`
	`changeWeight(PQ,item);`

Simple implementation complexities

	Construct	Get highest priority
Unsorted array	$O(n)$	$O(n)$
Sorted array	$O(n^2)$	$O(1)$

Applications
- Bandwidth management
- Shortest path algorithms
- Job scheduling
- Minimum spanning tree algorithm
- Huffman code (entropy encoding, compression, mp3)

The Heap

A specialized tree-based data structure (but an array) which is a complete tree where every node satisifes the 'heap condition':

$parent\ key \geq child\ key, \forall children$

There are two types of heaps:

- minheap - root is smallest value
- maxheap - root is largest value

Implementation

Arithmetic is given to assign children:

- Parent given $A[i]$ (from an array starting from i=1)
- Children of $A[i]$:
    - $i = parent \times 2, \qquad parent \times 2 + 1$

Heap Operations

`deletemax()`	1. Return highest priority item (return root)
	2. Fix heap:
	    a. Put last item into root position
	    b. Reduce size of PQ by one
	    c. Fix heap condition for root using downheap()
		`int deletemax(int* PQ, int* PQsize)   {`
		`        int v = PQ[1];`
		`        *(PQsize) -= 1;`
		`        PQ[1] = PQ[*PQsize];`
		`        downheap(PQ, 1, *PQsize);`
		`        return(v);`
		`}`

`downheap()`	Replace root in heap.
	1. Starting from the root, determine if it is a heap
	2. If not, swap with highest child
	3. Repeat until it is a heap
		`void downheap(int* PQ, int k, int PQsize){`
		`     int j,v;`
		`     v = PQ[k]; /* value, or priority */`
		
		`     /* while A[k] has children */`
		`     while(k <= PQsize/2){`
		`          /* point to children*/`
		`          j= k*2;`
		
		`          /* j set to highest child*/`
		`          if((j < PQsize) && (PQ[j] < PQ[j+1])){`
		`               j++;`
		`          }`
		
		`          /* check if it is already heap */`
		`          if (v >= PQ[j]){`
		`               break;`
		`          }`
		
		`          /* swap and continue */`
		`          PQ[k] = PQ[j];`
		`          k = j;`
		`     }`
		`     /* final position of original A[k] value*/`
		`     PQ[k] = v;`
		`}`

`upheap()`	Insertion a new item into an already-formed heap.
	1. Add new element to the end of the array
	2. Traverse up while heap property is broken
	3. If new elemenet is smaller than its parent, then swap
		`void upheap(int* PQ, int k){`
		`        int v;`
		`        v = PQ[k];`
		`        PQ[0] = INT_MAX;`
		`        /* if parent is less than the new integer, swap */`
		`        while(PQ[k/2] <= v){`
		`               PQ[k] = PQ[k/2];`
		`               k = k/2;`
		`        }`
		`        PQ[k] = v;`
		`}`

Bottom-up heap construction	Verify heap condition for smaller subheaps and join them together with their parent repeatedly	

Complexity
Upheap, downheap and deletemax are all $O(\log n)$

Heap Sort

Implementation

1. Construct heap
2. Swap root (maximum) with last element
3. Remove last element from further consideration (decrease size by 1)
4. Fix heap using downheap()
5. Repeat until finished

Complexity

1. Construct heap $O(n)$
2. Successively move maximum to end and fix:
    a. $n \times deletemax(\square)$;
    b. $n \times O(\log n)$

Strategies

Strategy 1 (top-down)	1. Insert items one-by-one into the array 2. upheap() as each new item is inserted	Insert n items into heap of size n: • Each insertion: $O(\log n)$ • Insertions: $O(n)$ Overall: $O(n \log n)$
Strategy 2 (bottom-up)	1. Insert items into unordered array 2. Once all items are in, downheap() for each subheap with roots from $A[1]$ to $A\left[\frac{n}{2}\right]$	On first glace it is $O(n \log n)$ but: • Only the root needs log(n) operations • The n/2 leaves have 0 work for downheap() • n/4 leaves at level h-1 have max 1 downheap() Analysis 1. At most $\frac{n}{2^{h+1}}$ nodes exist at height h 2. Total cost $\sum_{h=0}^{\log(n)} \frac{n}{2^{h+1}} * O(h)$ 3. Same as $O(n \sum \frac{h}{2^h})$ (converging geometric series) Overall: $\Theta(n)$

Graphs

A graph is a representation of a set of objects, wih some pairs of ojects connected by links.

A graph $G = \{V, E\}$ contains a set of:

- Vertices - can contain information
- Edges (links between vertices) - direction and/or weight

e.g. in trees and linked lists, vertices = nodes, edges = links.

Types of graphs

Undirected

Edges have no direction specified

Connected undirected graph Every pair of vertices is connected.
Unconnected undirected graph Has at least one unconnected pair of vertices.

Directed graph

Edge direction is specified and links are not symmetrical.

Acyclic, unconnected directed graph
Weakly connected directed graph If not every vertex is reachable from every other vertex.
Strongly connected directed graph If every vertex is reachable from every other vertex.

Special Graph Types

Strongly connected components in a digraph
Some weakly connected directed graphs will have sections of strongly connected graphs.

- Some applications include finding communities in networks

Bipartite graphs
U and V are disjoint sets of vertexes i.e. every vertex in U connects to a vertex in V and viceversa.

Complete graph
Each pair of graph vertices is connected by an edge but not to itself.

- Has $\frac{V(V-1)}{2}$ edges

Trees and graphs
Trees are an undirected graph that is connected and acyclic.

- Any two vertices are connected by exactly one simple path - a path that doesn't traverse a vertex more than once
- All vertices are connected

Representing Graph Vertices

Array/matrix representation
A matrix of source vertices (rows) to destination vertices (columns) with each element corresponding to the weight.

- Note: 0 can be a weight, so sometimes the value for marking a non-existent connection should be clear

Weighted directed graph
These will not be symmetrical in their matrix representation.

Sparse graphs
Graphs where there are few edges - these should be represented by an adjacency list.

- The size of each list (vertex) is its degree
- Each subnode in each node may carry a weight

Complexity of matrix vs list

	Space	Retrieve edge
Matrix	$\Theta(\lvert V\rvert^2)$	$\mathrm{O}(1)$
Adjacency list	$\Theta(\lvert V\rvert + \lvert E\rvert)$	$\mathrm{O}(\deg(\lvert \mathrm{V}\rvert))$

- Adjacency list better when number of vertices similar to number of edges

Traversal

Tree traversal: the process of visiting each node exactly once, in a systematic way.

- Assumes every node is reachable from the root
- Assumes every node has only one parent, can only be visited once

Graph traversal: the process of visiting all the nodes in a graph.

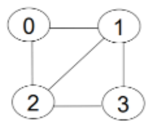
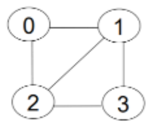
- Tree traversal is a special case of graph traversal
- Needs to ensure every node is reached and visited only once

Traversing an unconnected graph (depth first)

- Need to traverse each connected component
- Still need to mark nodes as visited, this can be done by a visited[] array:

```
/* invoke an array to track whether or not a
node has already been visited */
int visited[V];
listdfs(){
      int k;
      /* initialize - no nodes yet visited */
      for(k = 0; k < V; k++)
            visited[k]= 0;
}

/* adjacency list is an array of pointers to
nodes; node is struct with value (nodeID)
and next ptr*/
struct node{
      int value;
      struct node *next;
};
struct node* adj[V];


int visited[V];
int order=0; // keeps track of the order in which nodes are visited
void visitDFS(int k){
      struct node* t;
      visited[k] = ++order;
      for(t = adj[k]; t != NULL; t = t->next){
            if(!visited[t->v])
                  visitDFS( t->v );
      }
}
```

Simple Graph Searching

Depth-first search • Might not be shortest path • Works on connected, unweighted, directed and undirected graphs • Works on unconnected if start and end in same section • If visited nodes is kept track of, works on cyclic graphs • Usually uses a stack *Complexity* • Fill in visited[] array: |V| • Examine each edge twice: |E| • Overall: |V| + |E|	 0->1->2->3	`visited[k] = ++order;` `for(t=adj[k]; t != NULL; t=t->next){` `if(!visited[t->v]){` `visitDFS( t->v );` `}` `}`
Breadth-first search • Works one depth at a time • Will find shortest path • Works on connected, unweighted, directed and undirected graphs • Works on unconnected by checking visited array if there's an empty index • Works on cyclic graphs • Usually uses a queue *Complexity* • While loop runs V times • For loop runs out_degree times • Therefore O(V+E)	 2->0 2->1 2->3	`int visited[V]; int order=0;` `void visitBFS(int k){` `struct node* t;` `enQ(Q,k);` `while(!Qempty(Q)){` `k = deQ(Q);` `if(!visited[k]){` `visited[k] = ++order;` `for(t=adj[k]; t!=NULL; t=t->next){` `if(!visited[t->num]){` `enQ(Q,t->num);` `}` `}` `}` `}` `}`

Dijkstra's Algorithm

Store shortest distance from a particular vertex to every other vertex and update priority if shorter distance to vertex is found.

- Based on idea that any subpath along a shortest path is also a shortest path
- e.g. if shortest path from A to Y is through X, then path from A to X is also a shortest path
- Assumes no negative edges

Algorithm gives a shortest path tree, where the root is the source node and every node is connected to the root through its shortest path.

Overview

1. Let distance of start vertex from start vertex = 0
2. Let distance of all other vertices from start = infinity
3. Repeat following until all vertices visited (empty priority queue):
    a. Visit unvisited vertex with smallest known distance from start vertex
    b. For current vertex, examine its unvisited neighbours
    c. For current vertex, calculated distance of each neighbour from start vertex
    d. If calculated distance of a vertex is less than known distance, update shortest distance
    e. Update previous vertex for each updated distances

Relaxation

- Estimate the solution by answering an easier problem
- dist[] keeps updating the relaxed estimate until it is the solution to the original problem
- For shortest paths:
    - Estimate - known distance of best path so far
    - Solution - shortest possible distance
- Process vertices in order of a estimated closeness to source, value of dist[v]
- Priority queue to store vertex v and dist[v] value

```
void update_relax(u,v){
      if(dist[u] + edgeweight(u,v) < dist[v]){
            dist[v] = dist[u] + edgeweight(u,v);
            pred[v] = u;
      }
}
```

Pseudocode

```
void dijkstra(int** G, int s){
      int dist[Vsize], pred[Vsize];
      initialize(G, s, pred, dist);
      run(G, s, pred, dist);

      reconstruct(s, pred, dist);
}

void initialize(int** G, int Vsize, int s, int* pred, int* dist){
      int i;
      for(i=0; i<Vsize; i++){
```

```
                dist[i] = MAX_INT;
        }

        dist[s] = 0;
        for(i=0; i<V; i++){
                pred[i] = NULL;
        }
}

void run(int** G, int Vsize, int s, int* pred, int* dist){
        pq_node_t* pq;
        int u, v;
        pq = makePQ(G); /* vertices into min PQ, dist as priority */

        while(!emptyPQ(pq)){
                u = deletemin(pq);
                /* At this point vertex u has been processed,
                        i.e. dist[u] = delta(s,u) = shortest path to u found
*/

                for(/*each v conneted to u */){
                        if(dist[u] + edgeweight(u,v) < dist[v]){
                                update(v, pred, dist, pq);
                        }
                }
        }
}

void update(int v, int* pred, int* dist, pq_node_t* pq){
        dist[v] = dist[u] + edgeweight(u,v);
        pred[v] = u;
        decreaseweight(pq, v, dist[v]);
}
```

Dijkstra's Algorithm Analysis

```
void run(int** G, int Vsize, int s, int* pred, int* dist){
      pq_node_t* pq; // Assuming PQ is a minheap
      int u, v;
      pq = makePQ(G); // O(V)

      while(!emptyPQ(pq)){ // runs V times
            u = deletemin(pq); // O(log(V))

            for(/*each v conneted to u */){ // runs E times
                  if(dist[u] + edgeweight(u,v) < dist[v]){
                        update(v, pred, dist, pq); // O(log(V))
                  }
            }
      }
}
```

- makePQ() - O(V)
- V x deletemin() operations @ O(log(V))
- E x decreaseweight() operations @ O(log(V))

Total: $O((V + E)\log V)$

Limitations
- Assumes no negative edges
- Good for physical distances
- Distances are static
- If negative edges:
  - Use Bellman-Ford algorithm
  - $O(V * E)$
- Cannot deal with negative cycles

Transitive Closure

Transitive closure captures the reachability information.

Warshall Algorithm

1. On adjacency matrix, look at corresponding column and row for current intermediate
2. Source are all non-false elements in the column
3. Target are all non-false elements in the row

```
for( i=0; i < V; i++){ // intermediate vertex
      for( s=0; s < V; s++){ // source
            for( t=0; t < V; t++){ // tool/target
                  if( A[s][i] && A[i][t])
                        A[s][t] = TRUE;
```

- $\Theta(V^3)$ time complexity
- Boolean matrix, no self-loops

Floyd-Warshall algorithm

Same circumstance as Warshall but with weights; A[i][i] = 0; for each edge A[u][v] = weight(u, v); no path = infinity.

- Same algorithm as Warshall but detecting non-infinite and non-zero elements

```
for( i=0; i < V; i++){ // intermediate vertex
      for( s=0; s < V; s++){ // source vertex
            for( t=0; t < V; t++){ // target vertex
                  if(A[s][i] + A[i][t] < A[s][t])
                        A[s][t] = A[s][i] + A[i][t];
```

- $\Theta(V^3)$ time complexity
- No shortest path has length (number of segments) greather than V-1
- Gives distance of shortest path but doesn't establish the actual paths
  - Can be obtained through keeping another 2D path array
  - For each update to distance array, update path array to save node that made the path shorter
- Assumes graph representation is matrix

Implicit Graphs

Specify how to generate possible states - common in AI and problems where full graph is very large.

Optimisation

- Usually don't generate the whole graph
- Common efficiency improvement is to keep track of changes and reverse when traversing instead of copying all data

Greedy Algorithms & MSTs

Greedy algorithms keep taking the next best step repeatedly, until the best solution is reached e.g. Dijkstra's algorithm takes the next best edge to add to the path tree.

MSTs are undirected weighted graphs. They are a subgraph:

- A tree (no cycles)
- Graph must be connected
- Contains every vertex (spans)
- MST must have exactly V-1 edges
- No cycles
- Minimum sum of edge weights

General approach

- Start with isolated vertices, no edges
- Begin with any vertex (Prim's) or the least cost edge (Kruskal's)
- Keep adding vertices/edges to extend this subtree
    - Shortest connections
    - No cycles

MST Basic Concepts

Cut	A cut (V, V-S) of G is a partition of V
Cross	An edge (u, v) in E with one endpoint in S and the other in V-S
Light edge	The minimum weight edge crossing the cut
Respect	A cut respects a set A of edges if no edge in A crosses the cut

Cut during MST

- Fringe - part of V-S one step away from the MST
- Vertices in V-S have a cost (distance) from the Mst subtree so far constructured
- Distances between non-MST vertices and MST vertices are updated as vertices are added to MST

Prim's MST Algorithm

Preferred method for dense graphs and easiest with matrix representation.

Adds the next closest vertex.

Prim's MST construction
Pick lightest edge crossing the cut:

- Crossing edge (u, v) has u in S and v in V-S
  - Add v to S
  - Keep track of path (pred[])
  - Update distances between non-MST and MST vertices using w[]
- Repeat until V-S = {0}
- Reconstruct connections and distances from pred[] and w[]

Pseudocode

Draw a table to keep track of algorithm

	a	b	c	…
Dist[]	0	Inf	Inf	…
pred[]				…
inMST[]	F	F	F	…

PQ = {a, b, c, …}

Analysis/complexity

$O((V + E) \log V)$
$O(E \log V)$ for dense graphs with heap PQ

Proof

- Initialise arrays pred[] dist[] - O(v)
- Make PQ of vertices - if heap O(v)
- Loop while PQ not empty - V * …
  - Deletemin - if heap O(logV)
  - Update adjacent weights - O(degree of u * logV)
  - = O(V*logV + V*deg(u)*logV)
- Since V*deg(u) = E:
  - O(V*logV + E*logV)
  - = O((V+E)*logV)

Kruskal's MST Algorithm

Adds the next lowest weight edge that doesn't form a cycle (edge-based approach).

Pseudocode

```
E1: edges in MST so far
E2: remaining edges

E1=EMPTYSET, E2=E
Sort edges in E2 by weight
/* while E1 is not an MST and E2 is not empty */
while (|E1| < |V|-1) edges and (E2 not EMPTYSET)
      Pick min cost edge e(i,j) from E2 // pick beginning since sorted
      E2 = E2 \ e(i,j) // remove e(i, j) from E2
      if V(i),V(j) are not in same MST-so far, then
            E1 = E1 Union e(I,j)
            unite MSTs with V(i) and V(j)
```

Uniting MST requires new data structures and algorithm:

- Disjoint-set data structure
- Union-find algorithm

Analysis with best union-find (next page)

$$O(E \log E)$$

- Sort edges - O(E*logE)
- E finds and E unions - O(E+V)
- Therefore O(E*logE) - time dominated by sorting edges

When sorting dominates performance, partial sorting can help, only needing the smallest V-1 edges e.g. quicksort-like partition but:

- Works if graph is connected
- Doesn't work if longest edge needs to be in MST

Possible optimisations

- Attach new subtrees directly to root
- When moving up a tree - connect to root
- Union-by-rank - always join smaller tree to larger

Prim's vs Kruskal

	Prim	Kruskal	
General	$O((E+V)\log V)$	$O(E \log E)$	
Dense graph (E = V^ 2)	$O(E \log V)$	$O(E \log E)$	Prim's faster due to V<<E
Sparse graph (V=E)	$O(V \log V)$	$O(V \log E)$	Kruskal's faster due to data structures

Union-find Algorithm

Have disjoint subsets.

- Find - which subset is an element in (find if vertex is in same MST as another)
- Union - join two subsets into a single subset (join two subsets with the new edge into a single subset)

Naïve union-find (array)
Using an array id[] with indicies as the edges and elements as sets.

- Start with singleton sets

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	1	2	3	4	5	6	7

e.g. Union U(0, 1) to the same set, choose one to override:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	2	3	4	5	6	7

- Find:
  - id[p] == id[q]
  - O(1)
- Union:
  - id[p and all in same subset] = id[q]
  - O(V)

Speeding up union in union-find
Using a tree-based approach where id[] is a parent array, root is the representative of the subset.

- To union two subsets - make the root of one, the parent of the root of the other
- Start with all initialised to -1

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
-1	-1	-1	-1	-1	-1	-1	-1

e.g. Union U(0, 1) to the same set, assign one as the other's parent.

0	**1**	**2**	**3**	**4**	**5**	**6**
-1	0	-1	-1	-1	-1	-1

- Find
  - Traverse back through parent array to root
  - Nodes are in the same subset if they have the same root
  - Time for trace depends on depth of tree
  - Can be optimized by path compression
- Union
  - Weighted merge - merge smaller tree into larger

    - Keeps tree broader

Analysis
E union-finds on V vertices.

- Naïve - $O(EV)$
- Weighted or path compress - $O(V + E \log V)$
- Weighted and path compress - $O(E + V)\alpha(V) \approx O(E + V)$
    - $\alpha(n)$ is the inverse Ackermann function, a small constant

Topological Sort

A partial ordering that fulfils certain constraints.

- All edges e(i, j) go in horizontal i->j direction
- For directed acyclic graphs (DAG)

Useful for modelling may problems e.g. causalities, hierarchies etc.

Algorithm

1. Draw a table keeping track in-degree of each vertex (number of edges coming into the vertex)

e.g. graph above

Vertex	0	1	2	3	4	5	6
in-degree	0	2	2	2	1	2	0

2. Identify a source (in-degree = 0)
    a. Put that node in the topsort output
    b. Remove node from DAG
    c. Update in-degree matrix to account for removed node
3. Identify another source and repeat

Assumptions

- Must be at least one source and one sink for algorithm to work
- Valid assumption as acyclic graph - must have at least once source and one sink

Complexity

$$O(E)$$

Proof

- Build in-degree array by traversing edges - O(E)
- Remove edges - O(V * in-degree(V)) = O(E)
- Therefore O(E) + O(E) = O(E)

Topological Sort Uniqueness

If a Hamiltonian path (path where each vertex visited at most once) exists in the DAG, then the sort is unqiue.

- Finding a Hamiltonian Path is NP-Hard
- Have to solve problem: given a DAG, does a Hamiltonian Path exists?

Hamiltonian Path
Has a property called polynomial verifiability - verifying the existence of a Hamiltonian path can be easier than deteremining its existence.

- Given a topolgical sort, if two consectuvie vertices are not connected, then you can swap them - implies non unique and no Ham. path (can be done in linear time)

Path-finding in Graphs

Search algorithms over directed graphs:

- Search nodes (vertex) of the graph represent some information
- The edges capture transitions

Classification of search algorithms

Blind search	Only use the basic ingredients for general search algorithms e.g. DFS, BrFS, Dijkstra
Heuristic/informed search	Additionally use heuristic functions which estimate the distance (or remaining cost) to reach a target vertex e.g. A*
Systematic search	Consider a large number of search nodes simultaneously
Local search	Work with one (or a few) candidate solutions (search nodes) at a time e.g. hill-climbing

Blind vs heuristic search

	Pros	Cons
Blind search - does not requireany input beyond graph	No additional work for programmer	Same expansion order regardless what problem is
Heurestic search - requires additional heuristic function that maps nodes to estimates of their distances	Typically more effective in practice	Heuristic function has to be implemented

Blind search strategies

	Advantages
Breadth-first search	Time complexity
Depth-first search	Space complexity
Iterative deepening search	Combines advantages of both using depth-limited search as a sub-procedure

Heuristic Functions & Relaxing

Heuristic searches require a heuristic function to estimate remaining cost (cost of a shortest path from s to a goal state g).

- For many algorithms, h does not need to have any properties for the algorithm to work
- h is any function from states to numbers
- Successful heuristic search requires a good trade-off between h's informedness and the computational overhead of computing it

Relaxing
Simplifying the problem, and take the solution to the simpler problem as the heuristic estimate for the soltuion to the actual problem.

- Define a transformation that simplifies instance from the problem into instaces of a simpler problem