# Data Structures I

## COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Australia

## Table of Contents

## How does `<vector>` work?

- Vectors are dynamic arrays
- Random access is $O(1)$, like arrays
- A vector is stored *contiguously* in a single block of memory
- Supports an extra operation `push_back()`, which adds an element to the end of the array

## How does `push_back()` allocate memory?

- Do we have enough space allocated to store this new element? If so, we're done: $O(1)$.
- Otherwise, we need to allocate a new block of memory that is big enough to fit the new vector, and copy all of the existing elements to it. This is an $O(N)$ operation when the vector has $N$ elements. How can we improve?
- If we double the size of the vector each reallocation, we perform $O(N)$ work once, and then $O(1)$ work for the next $N - 1$ operations, an average of $O(1)$ per operation.
- We call this time complexity *amortised* $O(1)$.
- How is this different from average case complexity? When we quote 'average case' complexity (e.g. for a hash table), it is usually possible to construct a case where $N$ consecutive operations will each take $O(N)$ time, for a total time of $O(N^2)$. This is not possible with amortised complexity.

# Table of Contents

## Sets

- STL's `<set>` is a set with $O(\log n)$ random access
- Internally implemented as a red/black tree of set elements
- Unfortunately doesn't give you easy access to the underlying tree - iterator traverses it by infix order
- C++11 adds `<unordered_set>`, which uses hashing for $O(1)$ average case ($O(n)$ worst case) random access
- `<multiset>` and (C++11) `<unordered_multiset>` are also available

## Maps

- STL's `<map>` is a dictionary with $O(\log n)$ random access
- Internally implemented with a red/black tree of (key,value) pairs
- Unfortunately doesn't give you access to the underlying tree - iterator traverses it by infix order
- C++11 adds `<unordered_map>`, which uses hashing for $O(1)$ average case ($O(n)$ worst case) random access
- `<multimap>` and (C++11) `<unordered_multimap>` are also available

## Table of Contents

## Heaps

- Supports `push()` and `top()` operations in $O(\log n)$
- `top()` returns the value with highest priority
- Is usually used to implement a priority queue data structure
- STL implements a templated priority queue in `<queue>`
- The default is a max heap - often we want a min heap, so we declare it as follows:

```
priority_queue <T, vector<T>, greater<T>> pq;
```

- It's significantly more code to write a heap yourself, as compared to writing a stack or a queue, so it's usually not worthwhile to implement it yourself

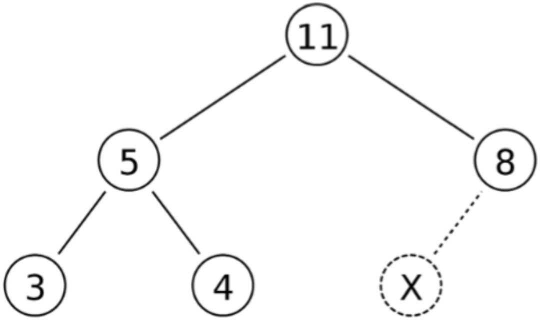
## Heaps

- The type of heaps usually used is more accurately called a *binary array heap* which is a binary heap stored in an array.
- It is a binary tree with two important properties:
  - **Heap property:** the value stored in every node is greater than the values in its children
  - **Shape property:** the tree is as close in shape to a complete binary tree as possible

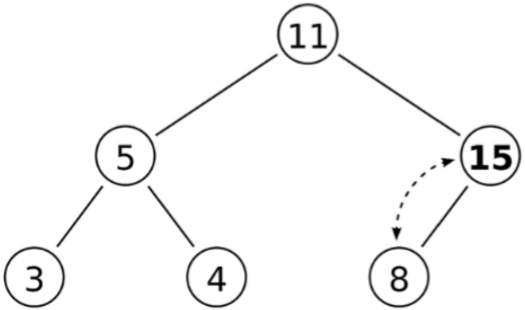
## Heaps

- Operation implementation
  - `push(v)`: add a new node with the value *v* in the first available position in the tree. Then, while the heap property is violated, swap with parent until it's valid again.
  - `pop()`: the same idea (left as an exercise)

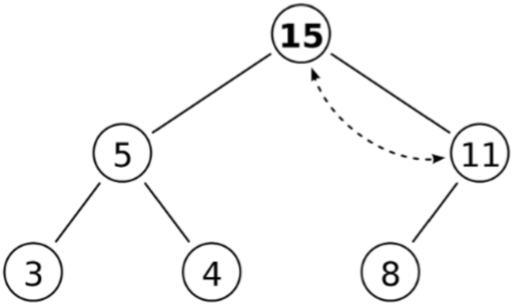
## Heaps



## Heaps



## Heaps



## Heaps

- **Implementation**

```
#include <queue>
```

## Table of Contents

## Subsequence Sum

- **Problem statement** You are given an array of $n$ numbers, say $a_0, a_1, \ldots, a_{n-1}$. Find the number of pairs $(i, j)$ with $0 \le i < j \le n$ such that the corresponding contiguous subsequence satisfies

$$a_i + a_{i+1} + \ldots + a_{j-1} = S$$

  for some specified sum $S$.
- **Input** The size $n$ of the array ($1 \le n \le 100,000$), and the $n$ numbers, each of absolute value up to 20,000, followed by the sum $S$, of absolute value up to 2,000,000,000.
- **Output** The number of such pairs

## Subsequence Sum

- **Algorithm 1** Evaluate the sum of each contiguous subsequence, and if it equals $S$, increment the answer.
- **Complexity** There are $O(n^2)$ contiguous subsequences, and each takes $O(n)$ time to add, so the time complexity is $O(n^3)$.
- **Algorithm 2** Compute the prefix sums

$$b_i = a_0 + a_1 + \ldots + a_{i-1}.$$

Then each subsequence can be summed in constant time:

$$a_i + a_{i+1} + \ldots + a_{j-1} = b_j - b_i.$$

- **Complexity** This solution takes $O(n^2)$ time, which is an improvement but still too slow.

## Subsequence Sum

- We need to avoid counting the subsequences individually.
- For each $1 \leq j \leq n$, we ask: how many $i < j$ have the property that $b_i = b_j - S$?
- If we know the frequency of each value among the $b_i$, we can add all the answers involving $j$ at once.
- The values could be very large, so a simple frequency table isn't viable - use a map!

## Subsequence Sum

- **Algorithm 3** Compute the prefix sums as above. Then construct a map, and for each $b_j$, add the frequency of $b_j - S$ to our answer and finally increment the frequency of $b_j$.
- **Complexity** The prefix sums take $O(n)$ to calculate, since

$$b_{i+1} = b_i + a_i.$$

Since map operations are $O(\log n)$, and each $b_j$ requires a constant number of map operations, the overall time complexity is $O(n \log n)$.

## Subsequence Sum

- **Implementation**

```cpp
const int N = 100100;
int a[N];
int b[N];

int main() {
  int n, S;
  cin >> n;
  for (int i = 0; i < n; i++) {
    cin >> a[i];
    b[i+1] = b[i] + a[i];
  }
  cin >> S;

  long long ret = 0;
  map<int,int> freq;
  for (int i = 0; i <= n; i++) {
    ret += freq[b[i]-S];
    freq[b[i]]++;
  }
  cout << ret << endl;
}
```

## Running Median

- **Problem statement** You are given an array of $n$ numbers. For each odd $k \leq n$, report the median of the first $k$ numbers in the array.
- **Input** The size $n$ of the array $(1 \leq n \leq 100,000)$, and the $n$ numbers, each of absolute value up to 2,000,000,000.
- **Output** The list of medians

## Running Median

- **Algorithm 1** Sort each odd length prefix of the array, and read off the median.
- **Complexity** This requires $O(n)$ sorts, each taking $O(n \log n)$ time, for an overall complexity of $O(n^2 \log n)$, which is too slow.
- **Algorithm 2** Sort the array using insertion sort, so that we don't have to sort from scratch each time.
- **Complexity** Insertion sort is $O(n^2)$, which is too slow.

## Running Median

- We don’t care about the entire result of sorting, just the median
- However, it’s not enough to maintain the median, because adding two numbers on the same side of the median would change the answer that we are to report
- We need to keep the numbers partly ordered, in some sense
- Heaps do this!
- Note: there is an alternative solution using a binary search tree.

## Running Median

- **Algorithm 3** Form a max heap $H_1$ and a min heap $H_2$. $H_1$ will have the lower half of the array, and $H_2$ will have the upper half of the array. We add every number to the relevant heap, making sure to rebalance the two heaps so that $H_1$ has the same or one more number than $H_2$. After each odd number of numbers, we report the number at the top of $H_1$.
- **Complexity** Heap operations take $O(\log n)$ time, and for each pair of numbers we read, we do at most a constant number of push(), pop() and top() operations. There are $O(n)$ pairs, so the overall time complexity is $O(n \log n)$.

## Running Median

- **Implementation**

```
#include<iostream>
#include<queue>

int main() {
  priority_queue<int, vector<int>, greater<int>> h1;
  priority_queue<int> h2;
  int n;
  cin >> n;
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    if (x > h1.top()) { // upper half
      h2.push(x);
      if (h1.size() < h2.size()) { // rebalance
        h1.push(h2.top());
        h2.pop();
      }
    } else { // lower half
      h1.push(x);
      if (h1.size() > h2.size() + 1) { // rebalance
        h2.push(h1.top());
        h1.pop();
      }
    }
    if (i % 2 == 0) // report median
      cout << h1.top() << endl;
  }
}
```

# Table of Contents

## Union-Find

- Also called a *system of disjoint sets*
- Given some set of elements, support the following operations:
  - union$(A, B)$: union the disjoint sets that contain $A$ and $B$
  - find$(A)$: return a canonical representative for the set that $A$ is in
    - More specifically, we must have find$(A) =$ find$(B)$ whenever $A$ and $B$ are in the same set.
    - It is okay for this answer to change as new elements are joined to a set. It just has to remain consistent across all elements in each disjoint set at a given moment in time.

## Union-Find: Straightforward Implementation

- Represent the unions as edges between the elements $A$ and $B$ in a tree
- A union is just adding the edge $AB$ to the tree
- A find is just walking up parent edges in the tree until the root is found
    - The number of edges traversed is equal to the height of the tree, so this is $O(h)$, where $h$ is the maximum height of the tree, which is $O(n)$

## Union-Find: Size Heuristic

- When adding an edge for a union, find the representative for both sets first
- Then, set the parent of the representative for the smaller set to the representative of the larger set
- The maximum height of this tree is now $O(\log n)$
    - When we traverse the edges from any particular element to its parent, we know that the subtree rooted at our current element must at least double in size, and we can double in size at most $O(\log n)$ times

## Union-Find: Size Heuristic

- We can now perform a find in $O(\log n)$ time, since the height is now $O(\log n)$
- Our union becomes slightly slower, because we need to perform two find operations at each union, but it's still only $O(\log n)$

## Union-Find: Path Compression

- When performing a find operation on some element $A$, instead of just returning the representative, we change the parent edge of $A$ to whatever the representative was, flattening that part of the tree
- Combined with the size heuristic, we get a time complexity of amortised $O(\alpha(n))$ per operation, but the proof is very complicated.

## Union-Find: Complexity

- What is $\alpha(n)$? $\alpha(n)$ is the inverse Ackermann function, a very slow growing function which is less than 5 for $n < 2^{2^{2^{2^{16}}}}$.
- As mentioned, the above two optimisations together bring the time complexity down to amortised $O(\alpha(n))$.
- **Warning:** due to a low-level detail, the path compression optimisation actually significantly slows down the find function, because we lose the tail recursion optimisation, now having to return to each element to update it.
- The time complexity improvement from $O(\log n)$ to $O(\alpha(n))$ is small enough that it can sometimes be overshadowed by this extra constant factor — whether or not this is the case depends on the bounds of the problem.

## Implementation

- **Implementation**

```
int parent[N];
int subtree_size[N];

void init(int n) {
  for (int i = 0; i <= n; i++) {
    parent[i] = i; subtree_size[i] = 1;
  }
}

int root(int x) {
  // only roots are their own parents
  // otherwise apply path compression
  return parent[x] == x ? x : parent[x] = root(parent[x]);
}

void join(int x, int y) {
  // size heuristic
  // hang smaller subtree under root of larger subtree
  x = root(x); y = root(y);
  if (x == y) return;
  if (subtree_size[x] < subtree_size[y]) {
    parent[x] = y;
    subtree_size[y] += subtree_size[x];
  } else {
    parent[y] = x;
    subtree_size[x] += subtree_size[y];
  }
}
```

## Table of Contents

## Kruskal's Algorithm

- Kruskal's algorithm is an alternative to Prim's algorithm for finding Minimum Spanning Trees.
- It has the same time complexity, but typically runs faster on sparse graphs, while Prim's typically runs faster on dense graphs.
- Kruskal's algorithm:
    - For each edge $e$ in increasing order of weight, add $e$ to the MST if the vertices it connects are not already in the same connected component.
    - Maintain connectedness with union-find.
    - This takes $O(|E| \log |E|)$ time to run, with the complexity dominated by the time needed to sort the edges in increasing order.

## Kruskal's Algorithm

- **Implementation**

```
struct edge {
  int u, v, w;
};
bool operator< (const edge& a, const edge& b) {
  return a.w < b.w;
}

edge edges[N];
int p[N];
int root (int u); // union-find root with path compression
void join (int u, int v); // union-find join with size heuristic

int mst() {
  sort(edges, edges+m); // sort by increasing weight
  int total_weight = 0;
  for (int i = 0; i < m; i++) {
    edge& e = edges[i];
    // if the endpoints are in different trees, join them
    if (root(e.u) != root(e.v)) {
      total_weight += e.w;
      join(e.u, e.v);
    }
  }
  return total_weight;
}
```

## Example problem: Masking

- **Problem statement** A graffiti artist wants to paint on a rectangular gridded canvas. First she applies strips of masking tape, which are rectangles aligned with the edges of the canvas, and covering only whole grid squares. Then she proceeds to paint contiguous areas ('holes') with block colours. Given the canvas size and the tape placements, what are the sizes of the holes that can be painted?
- **Input** The dimensions of the canvas, $m$ and $n$ $(1 \leq m, n \leq 500)$, the number of rectangles $k$ $(1 \leq k \leq 50)$ and the top left and bottom right coordinates of each rectangle.
- **Output** The sizes of the holes, in increasing order.

## Example problem: Masking

- Two squares are part of the same hole if they are adjacent and not part of any rectangle.
- This relation is transitive: if A and B are part of the same hole, and B and C are part of the same hole, then A and C are also part of the same hole
- We can solve this problem with a simple flood-fill, but let's do it with union-find instead (it'll be the same complexity!)

## Example problem: Masking

- **Algorithm** Mark every square covered by each piece of tape. Then set up union-find on the unmarked grid squares, and for every pair of adjacent unmarked grid squares, perform a join operation. Finally, count the frequency of each remaining root to determine the sizes of each hole, and sort them for printing.
- **Complexity** Marking squares takes $O(mn)$ for each of $k$ rectangles. The union-find step takes $O(mn \cdot \alpha(mn))$ times, and finally counting and sorting frequencies is $O(mn \log(mn))$. The overall time complexity is therefore $O(mn(k + \log(mn))$.

## Example problem: Masking

- **Implementation**

```
const int N = 505;
bool masked[N][N];
int freq[N*N];
const int di[4] = {-1,0,1,0};
const int dj[4] = {0,-1,0,1};

int main() {
  int m, n, k;
  cin >> m >> n >> k;
  // rectangle coordinates (top, left, bottom, right)
  vector<int> ilo, jlo, ihi, jhi;
  for (int t = 0; t < k; t++) {
    cin >> ilo >> jlo >> ihi >> jhi;
    // mark squares
    for (int i = ilo; i <= ihi; i++) {
      for (int j = jlo; j <= jhi; j++) {
        masked[i][j] = true;
      }
    }
  }
```

## Example problem: Masking

- **Implementation (continued)**

```
  // use union-find code from earlier
  init(m * n);
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
      // try neighbours in all directions
      for (int d = 0; d < 4; d++)
        // test if neighbour is on the canvas
        if (i + di[d] >= 0 && i + di[d] < m &&
            j + dj[d] >= 0 && j + dj[d] < n)
          if (!masked[i][j] && !masked[i+di[d]][j+dj[d]])
            join(i * n + j, (i + di[d]) * n + (j + dj[d]));

  // count frequencies
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
      if (!masked[i][j]) freq[root(i*n+j)]++;

  // sort and print out answers
  sort(freq, freq + m * n);
  for (int s = 0; s < m * n; s++) if (freq[s]) cout << freq[s] << ' ';
  cout << endl;
}
```