Rust Cheatsheet for CP

Basics

- Declaring variables:

```
let x = 10;
let mut y = 20;
y = 30;
let z: i32 = 100;
```

- References and borrowing:

```
// Immutable reference
let a = 10;
let ra = &a; // ra is a reference to a
println!("{}", ra); // prints 10

// Mutable reference
let mut b = 10;
let rb = &mut b; // mutable reference
*rb += 5;
println!("{}", b); // prints 15
```

- Defining and calling functions:

```
// Define a function
fn add(a: i32, b: i32) -> i32 {
    a + b
}

// Call the function
let sum = add(3, 4);
println!("{}", sum); // prints 7
```

Handling stdin/stout

- Reading from stdin:

```
use std::io::*;
let stdin = stdin();
let mut input = stdin.lock().lines();
```

- Parsing inputs:

```
let line = input.next().unwrap().unwrap();
let mut iter = line.split_whitespace();
let n: usize =
↪  iter.next().unwrap().parse().unwrap();
```

- Printing output:

```
println!("{}", result);
```

Handling file input/output

- Reading from a file:

```
use std::fs::File;
use std::io::{BufRead, BufReader, Result};

let file = File::open("input.txt")?;
let reader = BufReader::new(file);

for line_result in reader.lines() {
    let line = line_result?;
    // process line...
}
```

- Writing to a file:

```
use std::fs::File;
use std::io::{Write, BufWriter, Result};

let file = File::create("output.txt")?;
let mut writer = BufWriter::new(file);
writeln!(writer, "{}", result)?;
```

Data Structures

- Arrays:

```
use std::io::*;

fn main() -> Result<()> {
    let stdin = stdin();
    let mut input = stdin.lock().lines();

    let line = input.next().unwrap().unwrap();
    let nums: Vec<usize> =
    ↪  line.split_whitespace()
        .map(|x| x.parse().unwrap()).collect();

    let (n, m, l) = (nums[0], nums[1],
    ↪  nums[2]);

    let mut vector = vec![0; n];
    let mut matrix = vec![vec![0; m]; n];
    let mut cube = vec![vec![vec![0; l]; m];
    ↪  n];
    Ok(())
}
```

- Vectors:

```
let mut v = vec![1, 2, 3];
v.push(x);
v.pop();
v.len();
v.sort();
v.sort_unstable();
```

- HashMap:

```
use std::collections::HashMap;
let mut map = HashMap::new();
map.insert(key, val);
map.get(&key);
map.contains_key(&key);
```

- HashSet:

```
use std::collections::HashSet;
let mut set = HashSet::new();
set.insert(x);
set.contains(&x);
```

- VecDeque:

```
use std::collections::VecDeque;
let mut dq = VecDeque::new();
dq.push_back(x);
dq.pop_front();
```

- Strings:

```
let s = String::from("abc");
s.push('x');
s.push_str("xyz");
s.len();
&s[i..j];
```

- Graph Adjacency List

```
let mut adj = vec![Vec::new(); n];
adj[u].push((v, w)); // weighted
adj[u].push(v);      // unweighted
```

Iteration & Functional Methods

- Basic loop:

```
for x in &v {
    println!("{}", x);
}
```

- Enumerate:

```
for (i, x) in v.iter().enumerate() {
    // ...
}
```

- Map, Filter:

```
let w: Vec<_> = v.iter().map(|x|
↪  x+1).collect();
let evens: Vec<_> = v.iter().filter(|&&x|
↪  x%2==0).collect();
```

- Fold:

```
let sum = v.iter().fold(0, |acc, &x| acc + x);
```

Control Flow

```
// if/else
if x > 0 { ... } else { ... }

// match
match x {
  0 => "zero",
  1 => "one",
  _ => "other"
}

// loops
while condition { ... }
loop { if condition { break; } }
for i in 0..n { ... }
```

Utilities & Techniques

- Sorting:

```
v.sort();
v.sort_unstable();
```

- Binary Search:

```
match v.binary_search(&x) {
    Ok(i) => i,
    Err(i) => i
}
```

- Conversions:

```
let x = num as i64;
```

- String to chars:

```
let chars: Vec<char> = s.chars().collect();
```

Ownership & Borrowing

- Borrowing:

```
fn process(v: &Vec<i32>) { ... }
```

- Mutable reference:

```
fn modify(v: &mut Vec<i32>) {
    v.push(5);
}
```

- Slices:

```
fn first_slice(s: &[i32]) -> i32 {
    s[0]
}
```

Common CP Patterns

- Two-pointer:

```
let (mut l, mut r) = (0, v.len()-1);
while l < r {
    // ...
}
```

- HashMap Counting:

```
*map.entry(x).or_insert(0) += 1;
```

- Prefix sums:

```
prefix[i] = prefix[i-1] + arr[i];
```

- BFS queue:

```
use std::collections::VecDeque;
let mut q = VecDeque::new();
q.push_back(start);
```

BFS Example (Unweighted Graph)

```
use std::io::*;
use std::collections::VecDeque;

fn main() -> Result<()> {
    let stdin = stdin();
    let mut input = stdin.lock().lines();

    let line = input.next().unwrap().unwrap();
    let mut iter = line.split_whitespace();
    let n: usize =
    ↪  iter.next().unwrap().parse().unwrap();
    let m: usize =
    ↪  iter.next().unwrap().parse().unwrap();

    let mut adj = vec![Vec::new(); n];
    for _ in 0..m {
        let line = input.next().unwrap().unwrap();
        let mut it = line.split_whitespace();
        let u: usize =
        ↪  it.next().unwrap().parse().unwrap();
        let v: usize =
        ↪  it.next().unwrap().parse().unwrap();
        adj[u].push(v);
        adj[v].push(u); // if undirected
    }

    let start_line = input.next().unwrap().unwrap();
    let start: usize = start_line.parse().unwrap();

    let mut dist = vec![-1; n];
    dist[start] = 0;
    let mut q = VecDeque::new();
    q.push_back(start);

    while let Some(u) = q.pop_front() {
        for &vv in &adj[u] {
            if dist[vv] == -1 {
                dist[vv] = dist[u] + 1;
                q.push_back(vv);
            }
        }
    }

    for (i, d) in dist.iter().enumerate() {
        println!("Node {}: {}", i, d);
    }

    Ok(())
}
```

Dijkstra Example (Weighted Graph)

```
use std::io::*;
use std::collections::BinaryHeap;
use std::cmp::Reverse;

fn main() -> Result<()> {
    let stdin = stdin();
    let mut input = stdin.lock().lines();

    let line = input.next().unwrap().unwrap();
    let mut it = line.split_whitespace();
    let n: usize =
    ↪  it.next().unwrap().parse().unwrap();
    let m: usize =
    ↪  it.next().unwrap().parse().unwrap();

    let mut adj = vec![Vec::new(); n];
    for _ in 0..m {
        let line = input.next().unwrap().unwrap();
        let mut it = line.split_whitespace();
        let u: usize =
        ↪  it.next().unwrap().parse().unwrap();
        let v: usize =
        ↪  it.next().unwrap().parse().unwrap();
        let w: i64 =
        ↪  it.next().unwrap().parse().unwrap();
        adj[u].push((v, w));
    }

    let start_line = input.next().unwrap().unwrap();
    let start: usize = start_line.parse().unwrap();

    let mut dist = vec![i64::MAX; n];
```

```
    dist[start] = 0;
    let mut heap = BinaryHeap::new();
    heap.push((Reverse(0), start));

    while let Some((Reverse(d), u)) = heap.pop() {
        if d > dist[u] { continue; }
        for &(v, w) in &adj[u] {
            let nd = d + w;
            if nd < dist[v] {
                dist[v] = nd;
                heap.push((Reverse(nd), v));
            }
        }
    }

    for (i, d) in dist.iter().enumerate() {
        println!("Node {}: {}", i, d);
    }

    Ok(())
}
```

Error Handling

- Panic:

  ```
  panic!("Error message");
  ```

- Expect on input:

  ```
  let x: i32 = line.parse().expect("parse
  ↪  error");
  ```