https://simplifycpp.org

الخوارزميات الأساسية في لغة Rust

دليل هندسي عملي

إعداد: أيمن الحراكي

الخوارزميات الأساسية في لغة Rust

دليل هندسي عملي

تم دعم بعض أعمال الصياغة واستكشاف الأفكار باستخدام أدوات ذكاء اصطناعي حديثة،

وذلك تحت إشراف كامل، مع التحقق والمراجعة والتصحيح، وبمسؤولية وتأليف كاملين.

إعداد : أيمن الحراكي

simplifycpp.org

فبراير 2026

المحتويات

مقدمة المؤلف

الخوارزميات هي الأساس الحقيقي لهندسة البرمجيات. فهي التي تحدد جودة الحل، وكفاءته، وقابليته للتوسع، وموثوقيته على المدى الطويل. من دون فهم عميق للخوارزميات تصبح البرمجة مجرد كتابة تعليمات؛ أما معها فتتحول البرمجة إلى هندسة منضبطة.

اخترت **Rust** لهذا الكتاب لأنها تفرض إدارة صارمة للذاكرة وانضباطًا هيكليًا، مما يجعل تنفيذ الخوارزميات تجربة تعليمية قوية تجمع بين الأداء العالي وضمانات الأمان القوية. هذا الانضباط يشجع على التفكير الصحيح منذ البداية، وهو ما ينسجم تمامًا مع فلسفة هذا العمل.

صُمم هذا الكتاب ليكون مرجعًا عمليًا مختصرًا يركز على أنماط الخوارزميات الأساسية، مدعومًا بأمثلة واضحة وتحليل دقيق للتعقيد والأداء. الهدف ليس الحفظ، بل الفهم والتطبيق الهندسي السليم.

آمل أن يكون هذا الدليل رفيقًا موثوقًا يعزز أساسك الخوارزمي ويزودك بأدوات عملية لبناء حلول متينة باستخدام Rust.

أيمن الحراكي

التمهيد

لماذا وُجد هذا الكتاب

وُجد هذا الكتيب لسد فجوة عملية:

- كثير من كتب الخوارزميات تشرح الأفكار، لكنها لا تشرح القرارات الهندسية اللازمة لإنتاج تنفيذ صحيح، قابل للصيانة، وقابل للقياس.
- كثير من كتب Rust تشرح اللغة، لكنها لا توضح كيفية تحويل فكرة خوارزمية إلى مكوّن مختبر وقابل للاعتماد في بيئة إنتاج.

الخوارزميات ليست مجرد "حيلة لاجتياز مقابلة". في العمل الهندسي الحقيقي، الخوارزميات هي جوهر:

- ميزانيات الأداء وزمن الاستجابة،
- سلوك الذاكرة وقابليته للتنبؤ،
- الصحة تحت الحالات الحدّية والمدخلات العدائية،
- قابلية الصيانة وأمان إعادة الهيكلة.

لذلك يعامل هذا الدليل الخوارزميات على أنها مكوّنات:

- واجهة API واضحة،
- ثوابت (invariants) صريحة،
- اختبارات تثبت صحة هذه الثوابت،
- قياسات أداء توضح المقايضات،
- توثيق يمنع سوء الاستخدام.

مثال عملي لما تعنيه "هندسة الخوارزمية"

يمكنك تعلم البحث الثنائي في فقرة واحدة. لكن أداة بحث ثنائي موثوقة هي أثر هندسي صغير:

- يجب أن تحدد ماذا يحدث عند غياب المفتاح،
- يجب أن تتعامل مع التكرارات بشكل حتمي (أول/آخر ظهور)،
- يجب أن تثبت الانتهاء والصحة،
- يجب أن تتجنب تجاوز السعة في حساب الفهارس،
- يجب اختبارها بالحالات الحدّية واختبارات عشوائية.

```rust
/// Returns the leftmost index i such that a[i] >= key (a must be sorted).
/// This is "lower_bound".
pub fn lower_bound<T: Ord>(a: &[T], key: &T) -> usize {
    let mut lo: usize = 0;
    let mut hi: usize = a.len(); // invariant: answer is in [lo, hi]
    while lo < hi {
        // mid is safe from overflow: lo + (hi - lo)/2
        let mid = lo + (hi - lo) / 2;
        if &a[mid] < key {
```

```
            lo = mid + 1;
        } else {
            hi = mid;
        }
    }
    lo
}

#[cfg(test)]
mod tests {
    use super::lower_bound;

    #[test]
    fn lower_bound_basic() {
        let a = [1, 2, 2, 2, 5, 9];
        assert_eq!(lower_bound(&a, &0), 0);
        assert_eq!(lower_bound(&a, &1), 0);
        assert_eq!(lower_bound(&a, &2), 1);
        assert_eq!(lower_bound(&a, &3), 4);
        assert_eq!(lower_bound(&a, &10), a.len());
    }
}
```

هذا هو أسلوب هذا الكتاب: فكرة بسيطة ← عقد واضح ← تنفيذ ← اختبارات.

كيف تغيّر **Rust** طبيعة النقاش

تغيّر Rust هندسة الخوارزميات لأنها تفرض التصريح بالصحة والملكية وقت الترجمة.

1) الصحة تصبح بنيوية

تشجع Rust على بناء واجهات تمنع سوء الاستخدام:

- الملكية توضّح من يملك حق التعديل،
- الاستعارة (borrowing) توضّح أعمار المراجع،
- مطابقة الأنماط تفرض تغطية جميع الحالات،
- التعدادات (enums) تستبدل القيم الحارسة،
- النتائج (Result) تستبدل مسارات الخطأ الضمنية.

2) الأداء يبقى في المقدمة

تهدف Rust إلى أداء قابل للتنبؤ:

- لا يوجد جامع قمامة إجباري (GC),
- أنواع قيم فعّالة،
- تحكم واضح في التخصيص،
- يمكن للمكررات (iterators) أن تُترجم إلى حلقات فعالة،
- فحوص حدود صريحة يمكن للمُحسّن غالبًا إزالتها.

3) عزل **unsafe**

بعض الخوارزميات تحتاج تقنيات منخفضة المستوى. تسمح Rust بذلك، لكنها تشجع على العزل:

- إبقاء `unsafe` داخل وحدة صغيرة،

- تقديم واجهة آمنة مع توثيق الثوابت،
- اختبار مكثف عند الحدود.

مثال صغير: اختيار الأمان افتراضيًا

```rust
/// A safe "stack" with explicit operations and no invalid states.
pub struct Stack<T> {
    v: Vec<T>,
}

impl<T> Stack<T> {
    pub fn new() -> Self { Self { v: Vec::new() } }
    pub fn push(&mut self, x: T) { self.v.push(x); }
    pub fn pop(&mut self) -> Option<T> { self.v.pop() }
    pub fn peek(&self) -> Option<&T> { self.v.last() }
    pub fn len(&self) -> usize { self.v.len() }
    pub fn is_empty(&self) -> bool { self.v.is_empty() }
}
```

قد يبدو ذلك بسيطًا، لكن هذا الانضباط يتوسع ليشمل الرسوم البيانية، والجداول في البرمجة الديناميكية، والهياكل ذات الفهارس الكثيفة.

النطاق والفلسفة

هذا الكتيب هو دليل هندسي عملي.

النطاق

نركز على الخوارزميات المتكررة في الأنظمة الواقعية:

- البحث والاختيار،
- الفرز والترتيب،
- أساسيات التجزئة،
- المكدسات والطوابير،
- الأكوام وأولوية العناصر،
- بنية الاتحاد-الإيجاد،
- الرسوم البيانية،
- أنماط البرمجة الديناميكية،
- خوارزميات السلاسل النصية،
- الحيل العددية والبتّية الآمنة.

الفلسفة

- الوضوح أولاً ثم السرعة.
- اجعل الثوابت مرئية.
- قِس ولا تخمّن.
- فضّل **safe Rust**.
- هندس الواجهة جيداً.

ما ليس هذا الكتاب له

- ليس دورة كاملة في لغة Rust.
- ليس كتابًا أكاديميًا يبرهن كل نظرية رسميًا.
- ليس ملخصًا لمسابقات البرمجة.
- ليس فهرسًا لكل الخوارزميات المعروفة.
- ليس استعراضًا لكل مكتبات النظام البيئي.

كيفية استخدام هذا الكتاب

استخدم هذا الكتاب كدليل ورشة عمل.

إعداد **Rust** على ويندوز (سير عمل **cargo**)

```
# Create a new project
cargo new algo_lab
cd algo_lab

# Run tests
cargo test

# Run in release mode (important for benchmarks and performance)
cargo run --release

# Check formatting and linting discipline
cargo fmt
```

```
cargo clippy -- -D warnings
```

الجمهور المستهدف

هذا الكتيب موجّه إلى:

- مهندسي البرمجيات الراغبين في تقوية أساسهم الخوارزمي،
- مبرمجي الأنظمة الباحثين عن أداء متوقع وواجهات آمنة،
- مطوري Rust الذين يريدون منهجية منظمة،
- الطلاب الباحثين عن تنفيذات بجودة هندسية.

الانضباط الهندسي بدل الحفظ

الحفظ لا ينجح في الهندسة الواقعية لأن شكل المشكلة يتغير:

- قيود مختلفة،
- توزيعات مدخلات مختلفة،
- متطلبات صحة مختلفة،
- ميزانيات ذاكرة مختلفة،
- شروط تزامن وزمن استجابة مختلفة.

الانضباط هو ما يتوسع.

قائمة التحقق لهندسة الخوارزمية

لكل خوارزمية:

١. العقد: ماذا أقبل وماذا أضمن؟

٢. الثوابت: ماذا يجب أن يبقى صحيحًا؟

٣. الحالات الحدّية: فراغ، عنصر واحد، تكرارات.

٤. التعقيد: زمن وذاكرة.

٥. أنماط الفشل: تجاوز سعة، فهارس غير صالحة.

٦. الاختبارات: وحدات واختبارات عشوائية.

٧. القياس: ما المقياس المهم؟

لا تثق في تنفيذ خوارزمية حتى (1) تصرّح بالثوابت، (2) تختبر الحالات الحدّية، (3) تقيس ما يهم في وضع release.

الفصل ١: أساسيات Rust لتصميم الخوارزميات

١.١ إعداد المشروع باستخدام Cargo

يصبح العمل على الخوارزميات في Rust أكثر موثوقية عندما تتعامل معه كمشروع هندسي متكامل: تنظيم واضح للوحدات، اختبارات صارمة، وبناء قابل للإعادة التنفيذ على نظام ويندوز. يقدّم هذا القسم إعداداً موجهًا لويندوز يدعم:

- تنفيذات قابلة للإعادة الاستخدام داخل src/lib.rs,
- أمثلة صغيرة داخل src/main.rs,
- اختبارات وحدات واختبارات تكامل،
- تشغيل بوضع release لقياس الأداء بشكل واقعي.

إنشاء مشروع جديد (Windows PowerShell)

```
cargo new algo_handbook
cd algo_handbook
```

تحويله إلى مشروع يعتمد على المكتبة أولاً

حتى إن احتفظت بملف main.rs للأمثلة، ينبغي أن تعيش الخوارزميات داخل مكتبة حتى يمكن اختبارها وإعادة استخدامها.

```
# Ensure lib.rs exists (Cargo auto-detects a library when src/lib.rs exists)
New-Item -ItemType File -Path .\src\lib.rs -Force
```

التنظيم المقترح:

```
algo_handbook/
  Cargo.toml
  src/
    lib.rs
    main.rs
    search.rs
    sort.rs
    heap.rs
  tests/
    integration.rs
```

ملف **Cargo.toml** عملي وصارم

خيارات release تؤثر بشدة على قياسات أداء الخوارزميات. استخدم --release عند القياس.

```
[package]
name = "algo_handbook"
version = "0.1.0"
edition = "2021"
```

```
[profile.release]
opt-level = 3
lto = true
codegen-units = 1
panic = "abort"
```

أهم أوامر **cargo** (ويندوز)

```
cargo build
cargo test
cargo run --release
cargo test --release
cargo fmt
cargo clippy -- -D warnings
```

ربط الوحدات الأساسية

```
/* src/lib.rs */
pub mod search;
pub mod heap;
pub mod maps;

pub use search::*;
pub use heap::*;
pub use maps::*;
```

٢.١ قواعد الملكية وتأثيرها على الأداء

الملكية في Rust أداة أداء. فهي تجبرك على اتخاذ قرارات صريحة حول:

- متى تُنقل البيانات ومتى تُستعار،
- متى يحدث تخصيص على الكومة،
- متى يحدث نسخ عميق (clone)،
- هل يمكن تنفيذ الخوارزمية في مكانها (&mut [T]) أم تحتاج إلى تخصيص جديد.

استخدم الشرائح للقراءة فقط

إذا كانت الخوارزمية تقرأ فقط، ففضّل &[T] بدل &Vec<T>.

```
pub fn sum_i64(a: &[i64]) -> i64 {
    let mut s = 0i64;
    for &x in a {
        s += x;
    }
    s
}
```

استخدم &mut [T] للخوارزميات داخل المكان

```
pub fn reverse_in_place<T>(a: &mut [T]) {
    let mut i = 0usize;
    let mut j = a.len();
    while i < j {
```

```
        j -= 1;
        a.swap(i, j);
        i += 1;
    }
}
```

افهم الفرق بين Copy و Move وClone

```
pub fn copy_move_clone_demo() {
    let x: u64 = 7;
    let y = x;
    let _ = (x, y);

    let s1 = String::from("abc");
    let s2 = s1;
    let s3 = s2.clone();
    let _ = (s3,);
}
```

تجنب النسخ غير المقصود في المسارات الحساسة

```
pub fn contains_sorted<T: Ord>(a: &[T], key: &T) -> bool {
    let mut lo = 0usize;
    let mut hi = a.len();
    while lo < hi {
        let mid = lo + (hi - lo) / 2;
        if &a[mid] < key {
            lo = mid + 1;
```

```
        } else {
            hi = mid;
        }
    }
    lo < a.len() && &a[lo] == key
}
```

٣.١ الاستخدام الفعّال لـ Vec والشرائح

الحاوية Vec<T> هي اللبنة الأساسية لمعظم الخوارزميات.

- التخطيط للسعة،
- تقليل إعادة التخصيص،
- تقليل فحوص الحدود،
- استخدام الشرائح لدعم المقاطع دون نسخ.

حجز السعة مسبقًا

```
pub fn filter_non_negative(a: &[i32]) -> Vec<i32> {
    let mut out = Vec::with_capacity(a.len());
    for &x in a {
        if x >= 0 {
            out.push(x);
        }
    }
    out
}
```

فضّل واجهات الشرائح

```
pub fn rotate_left_by_one<T>(a: &mut [T]) {
    if a.len() <= 1 { return; }
    a.rotate_left(1);
}
```

استخدم دوال الشرائح لزيادة الوضوح

```
pub fn is_sorted_i32(a: &[i32]) -> bool {
    a.windows(2).all(|w| w[0] <= w[1])
}
```

نمط المؤشرين

```
pub fn merge_sorted(a: &[i32], b: &[i32]) -> Vec<i32> {
    let mut out = Vec::with_capacity(a.len() + b.len());
    let (mut i, mut j) = (0usize, 0usize);
    while i < a.len() && j < b.len() {
        if a[i] <= b[j] {
            out.push(a[i]); i += 1;
        } else {
            out.push(b[j]); j += 1;
        }
    }
    out.extend_from_slice(&a[i..]);
    out.extend_from_slice(&b[j..]);
    out
}
```

٤.١ مقدمة سريعة إلى HashMap و BinaryHeap

HashMap: العدّ والفهرسة

```
use std::collections::HashMap;

pub fn frequency(words: &[&str]) -> HashMap<String, usize> {
    let mut m: HashMap<String, usize> = HashMap::new();
    for &w in words {
        *m.entry(w.to_string()).or_insert(0) += 1;
    }
    m
}
```

HashMap: نمط الحفظ المؤقت

```
use std::collections::HashMap;

pub fn fib(n: u64) -> u64 {
    fn go(n: u64, memo: &mut HashMap<u64, u64>) -> u64 {
        if let Some(&v) = memo.get(&n) { return v; }
        let v = match n {
            0 => 0,
            1 => 1,
            _ => go(n - 1, memo) + go(n - 2, memo),
        };
        memo.insert(n, v);
        v
    }
```

```
    let mut memo = HashMap::new();
    go(n, &mut memo)
}
```

BinaryHeap: استخراج أكبر k

```
use std::collections::BinaryHeap;

pub fn top_k(a: &[i32], k: usize) -> Vec<i32> {
    let mut heap = BinaryHeap::new();
    for &x in a {
        heap.push(x);
    }
    let k = k.min(heap.len());
    (0..k).filter_map(|_| heap.pop()).collect()
}
```

سلوك **min-heap** باستخدام Reverse

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

pub fn k_smallest(a: &[i32], k: usize) -> Vec<i32> {
    let mut heap: BinaryHeap<Reverse<i32>> = BinaryHeap::new();
    for &x in a {
        heap.push(Reverse(x));
    }
    let k = k.min(heap.len());
```

```
    let mut out: Vec<i32> = (0..k)
        .filter_map(|_| heap.pop())
        .map(|r| r.0)
        .collect();
    out.sort();
    out
}
```

٥.١ كتابة اختبارات وحدات نظيفة

تنفيذ الخوارزميات يجب أن يُختبر كأثر هندسي من الدرجة الأولى.

اختبارات حتمية

```
pub fn lower_bound_i32(a: &[i32], key: i32) -> usize {
    let mut lo = 0usize;
    let mut hi = a.len();
    while lo < hi {
        let mid = lo + (hi - lo) / 2;
        if a[mid] < key {
            lo = mid + 1;
        } else {
            hi = mid;
        }
    }
    lo
}
```

التحقق العشوائي مقابل مرجع بسيط

```
fn baseline_lower_bound(a: &[i32], key: i32) -> usize {
    for i in 0..a.len() {
        if a[i] >= key { return i; }
    }
    a.len()
}
```

٦.١ قالب قابل لإعادة الاستخدام

الاتساق يجعل قاعدة الشيفرة قابلة للصيانة.

هيكل القالب

```
/* ============================
   Algorithm: <NAME>
   ============================

   Contract:
   - Inputs:
   - Outputs:
   - Preconditions:
   - Postconditions:

   Invariants:
   - (loop invariants)
```

```
   Complexity:
   - Time:
   - Memory:

   Failure modes:
   - Panics?
   - Overflow?
   - Recursion depth?

   Notes:
   - Implementation choices, tradeoffs
*/

pub fn algo_name() {
    unimplemented!()
}
```

١.٧ خاتمة الفصل

في هذا الفصل وضعت الأساس العملي لهندسة الخوارزميات في Rust:

- استخدام Cargo كنظام هندسي منضبط.
- جعل الملكية تقود التصميم.
- الاستخدام الفعّال لـ Vec<T> والشرائح.
- توظيف HashMap و BinaryHeap بثقة.
- كتابة اختبارات نظيفة ومنهجية.

- اعتماد قالب موحّد لكل تنفيذ خوارزمي.

بهذا الانضباط، يمكن الانتقال في الفصول التالية إلى بناء عائلة خوارزميات متكاملة بجودة إنتاجية عالية.

الفصل ٢: الانضباط في التعقيد والأداء

١.٢ التعقيد الزمني في الواقع العملي

التعقيد الزمني ليس مجرد صيغة رياضية مجردة. في الهندسة الواقعية هو مزيج من:

- الرتبة الخوارزمية ($O(n)$، $O(n \log n)$، $O(n^2)$، إلخ)
- العوامل الثابتة
- أنماط الوصول إلى الذاكرة
- قابلية التنبؤ بالتفرعات
- محلية الذاكرة المخبئية
- تحسينات المترجم في وضع release

في Rust هذه الأبعاد مرئية ويمكن التحكم بها.

السلوك الحدّي مقابل العوامل الثابتة

قد تكون خوارزميتان من نفس الرتبة $O(n)$، لكن تختلفان عمليًا بسبب:

- وجود فحوص حدود داخل حلقة ضيقة،
- إعادة تخصيص متكرر،
- ضعف محلية الذاكرة،
- استخدام نسخ غير ضروري (clone).

مثال: تجميع بسيط مقابل أسلوب محسّن.

```rust
/* Naive style: indexing inside loop */
pub fn sum_indexing(a: &[i64]) -> i64 {
    let mut s = 0;
    for i in 0..a.len() {
        s += a[i];
    }
    s
}

/* Iterator style: often cleaner and equally optimized */
pub fn sum_iterator(a: &[i64]) -> i64 {
    a.iter().copied().sum()
}
```

كلاهما $O(n)$. في البناء المحسّن، غالبًا ما يزيل LLVM الفحوص الزائدة، لكن البنية غير الجيدة قد تمنع هذه التحسينات.

الحلقات المتداخلة والتكلفة الحقيقية

كشف التكرار:

```rust
pub fn has_duplicate_naive(a: &[i32]) -> bool {
    for i in 0..a.len() {
        for j in (i + 1)..a.len() {
            if a[i] == a[j] {
                return true;
            }
        }
    }
    false
}
```

مقابل حل يعتمد على HashSet:

```rust
use std::collections::HashSet;

pub fn has_duplicate_hash(a: &[i32]) -> bool {
    let mut seen = HashSet::with_capacity(a.len());
    for &x in a {
        if !seen.insert(x) {
            return true;
        }
    }
    false
}
```

نظريًا $O(n)$ أفضل من $O(n^2)$، لكن للمدخلات الصغيرة قد يكون الحل البسيط أسرع. الانضباط الهندسي يعني القياس بدل الافتراض.

توضيح سلوك التوسع

```rust
pub fn quadratic_example(n: usize) -> usize {
```

```
    let mut c = 0;
    for i in 0..n {
        for j in 0..n {
            if i == j {
                c += 1;
            }
        }
    }
    c
}

pub fn linear_example(n: usize) -> usize {
    let mut c = 0;
    for i in 0..n {
        if i % 2 == 0 {
            c += 1;
        }
    }
    c
}
```

مع زيادة n تصبح الدالة التربيعية غير عملية بسرعة. هذا السلوك يجب أن يوجّه اختيار الخوارزمية.

٢.٢ التعقيد المكاني وسلوك الذاكرة

التعقيد المكاني لا يقل أهمية عن الزمني. سلوك الذاكرة يؤثر على:

- كفاءة الذاكرة المخبئية،
- أخطاء الصفحات،

- كلفة التخصيص،
- التجزئة،
- قابلية التوسع.

التخصيص على المكدس

- سريع جدًا،
- يُحرَّر تلقائيًا،
- محدود الحجم.

```
pub fn stack_array_example() -> i32 {
    let arr = [1, 2, 3, 4, 5];
    arr.iter().sum()
}
```

التخصيص على الكومة

- مرن في الحجم،
- يعتمد على المخصّص،
- له كلفة لكل عملية تخصيص.

```
pub fn heap_vector_example(n: usize) -> i64 {
    let v: Vec<i64> = vec![0; n];
    v.iter().sum()
}
```

محلية الذاكرة

الذاكرة المتجاورة مثل Vec تحسن أداء الذاكرة المخبئية.

```
pub fn contiguous_scan(v: &[i64]) -> i64 {
    let mut s = 0;
    for &x in v {
        s += x;
    }
    s
}
```

الهياكل المرتبطة تقلل المحلية:

```
pub struct Node {
    pub value: i64,
    pub next: Option<Box<Node>>,
}
```

التنقل عبر مؤشرات منفصلة يضعف أداء المخبأ.

٣.٢ المكدس مقابل الكومة

خصائص المكدس

- إطارات ثابتة الحجم،
- كلفة منخفضة جداً،
- خطر تجاوز المكدس مع الاستدعاء العميق.

```rust
pub fn factorial_recursive(n: u64) -> u64 {
    if n <= 1 {
        1
    } else {
        n * factorial_recursive(n - 1)
    }
}
```

البديل التكراري:

```rust
pub fn factorial_iterative(n: u64) -> u64 {
    let mut result = 1;
    for i in 2..=n {
        result *= i;
    }
    result
}
```

أنماط نمو الكومة

التخصيص المتكرر داخل الحلقات يضعف الأداء.

```rust
pub fn inefficient_concat(a: &[&str]) -> String {
    let mut s = String::new();
    for &part in a {
        s = s + part;
    }
    s
}
```

الأسلوب الفعّال:

```
pub fn efficient_concat(a: &[&str]) -> String {
    let total: usize = a.iter().map(|s| s.len()).sum();
    let mut out = String::with_capacity(total);
    for &part in a {
        out.push_str(part);
    }
    out
}
```

٤.٢ تجنب التخصيص غير الضروري

التخصيص من أكثر العمليات كلفة في الحلقات الضيقة.

إعادة استخدام المخازن

```
pub fn reuse_buffer(a: &[i32], buffer: &mut Vec<i32>) {
    buffer.clear();
    for &x in a {
        if x > 0 {
            buffer.push(x);
        }
    }
}
```

استخدم الشرائح بدل النسخ

```
pub fn print_slice(a: &[i32]) {
    for &x in a {
        println!("{}", x);
    }
}
```

فضّل المكررات التي لا تخصّص

```
pub fn count_even(a: &[i32]) -> usize {
    a.iter().filter(|&&x| x % 2 == 0).count()
}
```

٥.٢ قياس الأداء في Rust

لا تفترض الأداء؛ قم بقياسه.

وضع release إلزامي

```
cargo run --release
cargo test --release
```

قياس بسيط باستخدام Instant

```
use std::time::Instant;
```

```rust
pub fn measure_example() {
    let data: Vec<i32> = (0..1_000_000).collect();
    let start = Instant::now();
    let sum: i32 = data.iter().sum();
    let duration = start.elapsed();
    println!("Sum: {}, Time: {:?}", sum, duration);
}
```

نمط قياس مصغّر

```rust
use std::time::Instant;

pub fn benchmark<F: Fn()>(f: F, iterations: usize) {
    let start = Instant::now();
    for _ in 0..iterations {
        f();
    }
    let elapsed = start.elapsed();
    println!("Elapsed: {:?}", elapsed);
}
```

تجنب أخطاء القياس

- تأكد أن العمل لا يُزال بالتحسين،
- استخدم وضع release,
- كرر القياس عدة مرات،

- قم بتسخين الذاكرة المخبئية،
- قارن دائمًا مع مرجع أساسي.

٦.٢ خاتمة الفصل

الانضباط في الأداء ليس خيارًا في هندسة الخوارزميات.
هذا الفصل أسس لما يلي:

- فهم التعقيد الزمني نظريًا وعمليًا.
- إدراك أثر التعقيد المكاني على السلوك المخبئي وقابلية التوسع.
- اختيار واعٍ بين المكدس والكومة.
- تقليل التخصيصات لزيادة الإنتاجية.
- قياس الأداء في وضع release بشكل إلزامي.

من الآن فصاعدًا، كل خوارزمية في هذا الدليل ستلتزم بـ:

- تحليل تعقيد واضح،
- توصيف صريح لسلوك الذاكرة،
- وعي بالتخصيص،
- أداء مقاس في بناء واقعي.

الصحة إلزامية. الأداء قابل للقياس. والانضباط هو ما يجعل الاثنين مستدامين.

الفصل ٣: المؤشران وتقنية النافذة المنزلقة

١.٣ مسألة Two Sum (مصفوفة مرتبة)

عندما يكون الإدخال مصفوفة مرتبة، يمكن حل مسألة Two Sum بزمن خطي باستخدام مؤشرين. هذا مثال كلاسيكي على استغلال الترتيب لتجنب استخدام التجزئة أو الذاكرة الإضافية.

المسألة

معطى شريحة مرتبة a وعدد صحيح target، أوجد الفهرسين `(i, j)` بحيث:

$$a[i] + a[j] = target \quad \text{و} \quad i < j$$

أعد None إذا لم يوجد حل.

الفكرة الأساسية

- ابدأ بـ i = 0 (يسار) و j = n-1 (يمين).
- إذا كان المجموع صغيرًا جدًا، حرّك المؤشر الأيسر لزيادة المجموع.
- إذا كان المجموع كبيرًا جدًا، حرّك المؤشر الأيمن لتقليل المجموع.

التنفيذ

```rust
pub fn two_sum_sorted(a: &[i32], target: i32) -> Option<(usize, usize)> {
    if a.len() < 2 { return None; }
    let mut i: usize = 0;
    let mut j: usize = a.len() - 1;

    while i < j {
        let s = a[i] + a[j];
        if s == target {
            return Some((i, j));
        } else if s < target {
            i += 1;
        } else {
            j -= 1;
        }
    }
    None
}
```

التعقيد

- الزمن: $O(n)$
- الذاكرة: $O(1)$

٢.٣ إزالة التكرارات داخل المكان

نمط متكرر في المصفوفات المرتبة: ضغط المصفوفة المرتبة داخل المكان بحيث يظهر كل عنصر مرة واحدة فقط.

الثوابت

- المؤشر write يشير إلى الموضع التالي لكتابة عنصر فريد.
- جميع العناصر في a[0..write] فريدة.

التنفيذ

```
pub fn remove_duplicates_sorted(a: &mut [i32]) -> usize {
    if a.is_empty() { return 0; }
    let mut write: usize = 1;
    for read in 1..a.len() {
        if a[read] != a[read - 1] {
            a[write] = a[read];
            write += 1;
        }
    }
    write
}
```

التعقيد

- الزمن: $O(n)$
- الذاكرة: $O(1)$

٣.٣ تقنية النافذة المنزلقة

النافذة المنزلقة هي تعميم قوي لفكرة المؤشرين:

- نافذة [l, r] على البيانات.
- تحريك r لتوسيع النافذة.
- تحريك l لتقليصها عند انتهاك القيود.

متى نستخدمها؟

- إيجاد أطول/أقصر مقطع يحقق شرطًا.
- إيجاد أكبر/أصغر مجموع ضمن شرط.
- معالجة مقاطع نصية أو مصفوفات مع قيود.
- عدّ النوافذ التي تحقق معيارًا معينًا.

قالب عام قابل لإعادة الاستخدام

```rust
pub fn sliding_window_template(a: &[u8]) -> usize {
    let mut l: usize = 0;
    let mut best: usize = 0;

    let mut freq = [0u32; 256];
    let mut bad = 0u32;

    for r in 0..a.len() {
        let x = a[r] as usize;
```

```
        freq[x] += 1;
        if freq[x] == 2 {
            bad += 1;
        }

        while bad > 0 {
            let y = a[l] as usize;
            if freq[y] == 2 {
                bad -= 1;
            }
            freq[y] -= 1;
            l += 1;
        }

        let len = r + 1 - l;
        if len > best { best = len; }
    }

    best
}
```

٤.٣ أطول مقطع دون تكرار أحرف

هذه المسألة هي المثال القياسي لتقنية النافذة المنزلقة.

نسخة سريعة تعتمد على البايت

```
pub fn longest_unique_substring_bytes(s: &str) -> usize {
```

```
    let b = s.as_bytes();
    let mut last = [-1i32; 256];
    let mut l: usize = 0;
    let mut best: usize = 0;

    for (r, &ch) in b.iter().enumerate() {
        let idx = ch as usize;
        let p = last[idx];
        if p >= 0 {
            let p = p as usize;
            if p >= l {
                l = p + 1;
            }
        }
        last[idx] = r as i32;

        let len = r + 1 - l;
        if len > best { best = len; }
    }
    best
}
```

نسخة تعتمد على char

```
use std::collections::HashMap;

pub fn longest_unique_substring_chars(s: &str) -> usize {
    let mut last: HashMap<char, usize> = HashMap::new();
    let mut l: usize = 0;
```

```
    let mut best: usize = 0;

    for (r, ch) in s.chars().enumerate() {
        if let Some(&p) = last.get(&ch) {
            if p >= l {
                l = p + 1;
            }
        }
        last.insert(ch, r);
        let len = r + 1 - l;
        if len > best { best = len; }
    }
    best
}
```

٥.٣ أقصى مجموع مقطع

مجموع نافذة بحجم ثابت

```
pub fn max_sum_fixed_window(a: &[i64], k: usize) -> Option<i64> {
    if k == 0 || a.len() < k { return None; }
    let mut sum: i64 = a[..k].iter().sum();
    let mut best = sum;

    for i in k..a.len() {
        sum += a[i];
        sum -= a[i - k];
        if sum > best { best = sum; }
```

```
    }
    Some(best)
}
```

خوارزمية Kadane

```rust
pub fn max_subarray_kadane(a: &[i64]) -> Option<i64> {
    if a.is_empty() { return None; }
    let mut best = a[0];
    let mut cur = a[0];

    for &x in &a[1..] {
        cur = (cur + x).max(x);
        best = best.max(cur);
    }
    Some(best)
}
```

٦.٣ الحالات الحدّية واستراتيجيات التحسين

تفشل هذه الخوارزميات غالبًا بسبب:

1) المدخلات الفارغة أو الصغيرة

- شريحة فارغة،
- عنصر واحد،

- $k = 0$،
- $k < n$.

2) أخطاء الحدود

- استخدام شروط خاطئة في الحلقات،
- حساب طول النافذة بطريقة غير صحيحة،
- نسيان تحديث أفضل نتيجة.

3) تجاوز النقص مع usize

تأكد من عدم إنقاص مؤشر غير صفري.

4) اختيار تمثيل الحالة المناسب

- نطاق ثابت (ASCII) ← مصفوفة ثابتة.
- نطاق واسع (Unicode) ← HashMap.

5) تجنب التخصيص غير الضروري

اعمل بالشرائح وأعد الفهارس أو الأطوال كلما أمكن.

6) حركة أحادية الاتجاه

- المؤشر r يتحرك للأمام مرة واحدة،
- المؤشر l يتحرك أحاديًا،

- تحديثات الحالة تتم في $O(1)$.

اختبار عشوائي مرجعي

```
fn baseline_two_sum_sorted(a: &[i32], target: i32) -> Option<(usize, usize)>
↪  {
    for i in 0..a.len() {
        for j in (i + 1)..a.len() {
            if a[i] + a[j] == target {
                return Some((i, j));
            }
        }
    }
    None
}
```

٧.٣ خاتمة الفصل

المؤشران والنافذة المنزلقة ليستا حيلاً، بل منهجيات منضبطة لبناء خوارزميات خطية.
في هذا الفصل تناولنا:

- حل Two Sum على بيانات مرتبة بزمن $O(n)$ وذاكرة $O(1)$.
- إزالة التكرارات داخل المكان باستخدام ثوابت واضحة.
- قالب نافذة منزلقة قابل للإعادة الاستخدام.
- أطول مقطع دون تكرار (نسخة بايت ونسخة char).
- أنماط أقصى مقطع (نافذة ثابتة و Kadane).

- قائمة فحص عملية للحالات الحدّية والتحسين.

في الفصل التالي سنبني على هذه الأنماط لتطبيق نوافذ تعتمد على التردد، وطوابير أحادية الاتجاه، وقيود أكثر تعقيداً تظهر في الأنظمة الواقعية ومسائل المقابلات المتقدمة.

الفصل ٤: مجاميع البادئة وتقنيات المدى

١.٤ أساسيات مجموع البادئة

مجاميع البادئة (Prefix Sums) من أعلى الأدوات قيمة في هندسة الخوارزميات. فهي تحوّل استعلامات المدى المتكررة من $O(n)$ لكل استعلام إلى $O(1)$ لكل استعلام بعد مرحلة تمهيدية بتكلفة $O(n)$.

التعريف

معطى مصفوفة $a[0..n)$، نعرّف مصفوفة مجاميع البادئة $p[0..n]$ كما يلي:

$$p[0] = 0, \quad p[i+1] = p[i] + a[i]$$

وعندها يكون مجموع أي مدى $a[l] + \cdots + a[r-1]$ هو:

$$\mathsf{sum}(l, r) = p[r] - p[l]$$

انضباط هندسي

- استخدم طولاً قدره $n+1$ لتجنب الحالات الخاصة.

- استخدم نوعًا عدديًا أوسع لتقليل خطر تجاوز السعة.

- تعامل مع مجاميع البادئة كواجهة API: تُبنى مرة وتُستعلم مرات عديدة.

التنفيذ: البناء والاستعلام

```
pub fn prefix_sums_i64(a: &[i64]) -> Vec<i64> {
    let mut p = Vec::with_capacity(a.len() + 1);
    p.push(0);
    for &x in a {
        let next = p[p.len() - 1] + x;
        p.push(next);
    }
    p
}

pub fn range_sum(p: &[i64], l: usize, r: usize) -> i64 {
    /* caller guarantees 0 <= l <= r <= n */
    p[r] - p[l]
}
```

التعقيد

- البناء: $O(n)$ زمنًا و $O(n)$ ذاكرة

- كل استعلام: $O(1)$

٢.٤ عدد المقاطع التي مجموعها يساوي K

هذه المسألة توضح القوة الحقيقية لمجاميع البادئة عند دمجها مع HashMap.

الفكرة الأساسية

إذا كان $p[i]$ هو مجموع البادئة حتى الفهرس i، فإن المقطع $[l, r)$ مجموعُه يساوي k إذا:

$$p[r] - p[l] = k \quad \leftarrow \quad p[l] = p[r] - k$$

إذًا لكل r نحتاج عدد مرات ظهور القيمة $(p[r] - k)$ سابقًا.

التنفيذ

```rust
use std::collections::HashMap;

pub fn count_subarrays_sum_k(a: &[i64], k: i64) -> i64 {
    let mut freq: HashMap<i64, i64> = HashMap::new();
    freq.insert(0, 1);

    let mut p: i64 = 0;
    let mut count: i64 = 0;

    for &x in a {
        p += x;
        if let Some(&c) = freq.get(&(p - k)) {
            count += c;
        }
        *freq.entry(p).or_insert(0) += 1;
    }
```

```
    count
}
```

التعقيد

- الزمن: $O(n)$ في المتوسط
- الذاكرة: $O(n)$ في أسوأ الحالات

٣.٤ تقنية مصفوفة الفروق

مصفوفة الفروق (Difference Array) تحوّل عمليات التحديث على مدى إلى عمليات $O(1)$ لكل تحديث، يليها تمرير واحد لإعادة البناء.

الفكرة

لإضافة قيمة v إلى كل العناصر في المدى $[l, r]$:

$$d[l] += v, \quad d[r+1] -= v$$

ثم نعيد بناء المصفوفة الأصلية بجمع تراكمي.

التنفيذ

```
pub fn diff_range_add(n: usize, updates: &[(usize, usize, i64)]) -> Vec<i64>
↪  {
    let mut d = vec![0i64; n + 1];
    for &(l, r, v) in updates {
```

```
        d[l] += v;
        d[r + 1] -= v;
    }

    let mut a = vec![0i64; n];
    let mut cur = 0i64;
    for i in 0..n {
        cur += d[i];
        a[i] = cur;
    }
    a
}
```

التعقيد

- التحديثات: $O(m)$
- إعادة البناء: $O(n)$
- الذاكرة: $O(n)$

٤.٤ مجموع بادئة ثنائي الأبعاد

يمكّننا مجموع البادئة ثنائي الأبعاد من حساب مجموع أي مستطيل في زمن ثابت.

التعريف

$$p[y+1][x+1] = g[y][x] + p[y][x+1] + p[y+1][x] - p[y][x]$$

ومجموع مستطيل نصف-مفتوح:

$$S = p[y_1][x_1] - p[y_0][x_1] - p[y_1][x_0] + p[y_0][x_0]$$

تنفيذ بتخزين متجاور

```
pub struct Prefix2D {
    h: usize,
    w: usize,
    p: Vec<i64>,
}

impl Prefix2D {
    fn idx(&self, y: usize, x: usize) -> usize {
        y * (self.w + 1) + x
    }

    pub fn new(g: &[Vec<i64>]) -> Self {
        let h = g.len();
        let w = if h == 0 { 0 } else { g[0].len() };

        let mut p = vec![0i64; (h + 1) * (w + 1)];
        let mut obj = Self { h, w, p };

        for y in 0..h {
            for x in 0..w {
                let val = g[y][x];
                let a = obj.p[obj.idx(y, x + 1)];
                let b = obj.p[obj.idx(y + 1, x)];
```

```
                let c = obj.p[obj.idx(y, x)];
                obj.p[obj.idx(y + 1, x + 1)] = val + a + b - c;
            }
        }
        obj
    }

    pub fn rect_sum(&self, y0: usize, x0: usize, y1: usize, x1: usize) -> i64
    ↪ {
        let p11 = self.p[self.idx(y1, x1)];
        let p01 = self.p[self.idx(y0, x1)];
        let p10 = self.p[self.idx(y1, x0)];
        let p00 = self.p[self.idx(y0, x0)];
        p11 - p01 - p10 + p00
    }
}
```

التعقيد

- **البناء:** $O(hw)$
- **كل استعلام:** $O(1)$
- **الذاكرة:** $O(hw)$

٥.٤ تطبيقات عملية

1) المقاييس السريعة

عند تجميع أحداث زمنية، تتيح مجاميع البادئة:

- حساب مجموع آخر فترة زمنية بسرعة،
- دعم نوافذ متداخلة بكفاءة.

2) مشاكل التوازن

تحويل القيم (مثل 0 إلى -1) يسمح باستخدام مجموع البادئة لاكتشاف أطول مدى يحقق توازنًا معينًا.

مثال: أطول مقطع يحوي عددًا متساويًا من 0 و 1

```rust
use std::collections::HashMap;

pub fn longest_equal_0_1(bits: &[u8]) -> usize {
    let mut first: HashMap<i64, usize> = HashMap::new();
    first.insert(0, 0);

    let mut p: i64 = 0;
    let mut best: usize = 0;

    for (i, &b) in bits.iter().enumerate() {
        p += if b == 0 { -1 } else { 1 };
        let pos = i + 1;
```

```
        if let Some(&start) = first.get(&p) {
            let len = pos - start;
            if len > best { best = len; }
        } else {
            first.insert(p, pos);
        }
    }
    best
}
```

3) معالجة المدى غير المتزامنة

تُستخدم مصفوفة الفروق عند وجود عدد كبير من التحديثات على المدى، مع الحاجة للنتيجة النهائية فقط.

4) التحليلات ثنائية الأبعاد

تُستخدم مجاميع البادئة ثنائية الأبعاد في:

- الخرائط الحرارية،
- جداول الإشغال،
- حساب مجموع المستطيلات بسرعة.

٦.٤ خاتمة الفصل

قدّم هذا الفصل مجاميع البادئة وتقنيات المدى كأدوات هندسية عملية:

- مجاميع البادئة تجعل استعلامات المدى ثابتة الزمن بعد تمهيد خطي.
- دمجها مع HashMap ينتج حلولاً خطية لمسائل المقاطع حتى مع القيم السالبة.
- مصفوفة الفروق تحوّل تحديثات المدى إلى عمليات ثابتة الزمن.
- التعميم ثنائي الأبعاد يمكّن من استعلامات مستطيلات بزمن ثابت.
- التطبيقات الواقعية تشمل المقاييس، مشاكل التوازن، التحديثات غير المتزامنة، وتحليلات الشبكات.

في الفصول القادمة سنبني أدوات مدى أكثر تقدمًا (مثل الطوابير الأحادية، طرق الفواصل، والتحسينات المعتمدة على الأكوام) مع نفس الانضباط: ثوابت واضحة، سلوك ذاكرة مدروس، واختبارات قوية.

الفصل ٥: إتقان البحث الثنائي

١.٥ البحث الثنائي الكلاسيكي

يُعدّ البحث الثنائي أهم مثال على خوارزمية تعتمد صحتها على عقد دقيق وثابت حلقة واضح. في الواقع العملي، تنشأ الأخطاء عادة بسبب:

- عدم وضوح معنى lo و hi,
- الخلط بين الحدود الشاملة والحصرية،
- عدم إثبات الانتهاء،
- تجاوز السعة عند حساب المنتصف،
- عدم تحديد سلوك العناصر المكررة.

سنستخدم في هذا الفصل نموذجًا منضبطًا واحدًا:

$$\text{lo} \in [0, n], \quad \text{hi} \in [0, n], \qquad \text{ومدى البحث هو } [\text{lo}, \text{hi})$$

النطاق نصف-المفتوح يجعل البرهان والتنفيذ أبسط.

العقد (بحث الانتماء)

معطى شريحة مرتبة a ومفتاح key، أعد Some(index) إذا وُجد المفتاح، وإلا أعد None. في حالة التكرار، لا نضمن أي موضع محدد؛ للحصول على سلوك حتمي استخدم lower_bound أو upper_bound.

ثابت الحلقة

في كل تكرار:

- جميع الفهارس المرشحة ضمن `[lo, hi)`.
- المدى يتناقص حتى يتحقق `lo == hi`.

التنفيذ

```
pub fn binary_search<T: Ord>(a: &[T], key: &T) -> Option<usize> {
    let mut lo: usize = 0;
    let mut hi: usize = a.len(); /* exclusive */

    while lo < hi {
        let mid = lo + (hi - lo) / 2;
        if &a[mid] < key {
            lo = mid + 1;
        } else if &a[mid] > key {
            hi = mid;
        } else {
            return Some(mid);
        }
    }
```

```
    None
}
```

التعقيد

- **الزمن:** $O(\log n)$
- **الذاكرة:** $O(1)$

٢.٥ الحد الأدنى والحد الأعلى

في التطبيقات الواقعية، التكرار موجود. لذلك نحتاج تعريفات حتمية:

- **Lower Bound**: أول فهرس i بحيث a[i] >= key.
- **Upper Bound**: أول فهرس i بحيث a[i] > key.

Lower Bound

```
pub fn lower_bound<T: Ord>(a: &[T], key: &T) -> usize {
    let mut lo: usize = 0;
    let mut hi: usize = a.len();
    while lo < hi {
        let mid = lo + (hi - lo) / 2;
        if &a[mid] < key {
            lo = mid + 1;
        } else {
            hi = mid;
        }
```

```
    }
    lo
}
```

Upper Bound

```rust
pub fn upper_bound<T: Ord>(a: &[T], key: &T) -> usize {
    let mut lo: usize = 0;
    let mut hi: usize = a.len();
    while lo < hi {
        let mid = lo + (hi - lo) / 2;
        if &a[mid] <= key {
            lo = mid + 1;
        } else {
            hi = mid;
        }
    }
    lo
}
```

حساب عدد التكرارات

```rust
pub fn count_equal<T: Ord>(a: &[T], key: &T) -> usize {
    let lb = lower_bound(a, key);
    let ub = upper_bound(a, key);
    ub - lb
}
```

٣.٥ البحث الثنائي على الإجابة

كثير من المسائل لا تبحث عن موضع عنصر، بل عن أصغر قيمة تحقق خاصية رتيبة.
إذا كانت الدالة pred(x) رتيبة (خطأ ثم صحيح)، يمكن استخدام البحث الثنائي لإيجاد أول قيمة تحقق الشرط.

النمط العام: أول قيمة صحيحة

```
pub fn first_true(mut lo: i64, mut hi: i64, pred: impl Fn(i64) -> bool) ->
↪  i64 {
    while lo < hi {
        let mid = lo + (hi - lo) / 2;
        if pred(mid) {
            hi = mid;
        } else {
            lo = mid + 1;
        }
    }
    lo
}
```

مثال: أقل سعة شحن

```
pub fn can_ship(weights: &[i64], days: i64, cap: i64) -> bool {
    let mut used_days = 1;
    let mut cur = 0;

    for &w in weights {
        if w > cap { return false; }
```

```
        if cur + w <= cap {
            cur += w;
        } else {
            used_days += 1;
            cur = w;
            if used_days > days { return false; }
        }
    }
    true
}

pub fn min_ship_capacity(weights: &[i64], days: i64) -> i64 {
    let mut lo = 0;
    let mut hi = 0;
    for &w in weights {
        if w > lo { lo = w; }
        hi += w;
    }
    first_true(lo, hi, |cap| can_ship(weights, days, cap))
}
```

٤.٥ البحث في مصفوفة مدوّرة

في المصفوفة المرتبة والمدوّرة، يوجد نصفان مرتبان. نحدد في كل خطوة أي نصف مرتب ثم نقرر الاتجاه.

```
pub fn search_rotated_distinct(a: &[i32], key: i32) -> Option<usize> {
    if a.is_empty() { return None; }
```

```
    let mut lo: usize = 0;
    let mut hi: usize = a.len() - 1;

    while lo <= hi {
        let mid = lo + (hi - lo) / 2;
        if a[mid] == key {
            return Some(mid);
        }

        if a[lo] <= a[mid] {
            if a[lo] <= key && key < a[mid] {
                if mid == 0 { break; }
                hi = mid - 1;
            } else {
                lo = mid + 1;
            }
        } else {
            if a[mid] < key && key <= a[hi] {
                lo = mid + 1;
            } else {
                if mid == 0 { break; }
                hi = mid - 1;
            }
        }
    }
    None
}
```

٥.٥ تجنب أخطاء الحدود وتجاوز السعة

القاعدة 1: اختر نموذج حدود واحداً

إما:

- `[lo, hi)` (مفضل للبساطة)،
- أو `[lo, hi]` عند الضرورة.

القاعدة 2: احسب المنتصف بأمان

لا تستخدم `(lo + hi)/2`. استخدم:

$$\text{mid} = \text{lo} + \frac{\text{hi} - \text{lo}}{2}$$

القاعدة 3: ضمن التقدم

- نحو اليمين: `lo = mid + 1`
- نحو اليسار: `hi = mid`

القاعدة 4: عرّف سلوك التكرار

- البحث الكلاسيكي: أي تطابق.
- lower_bound: أول >=.
- upper_bound: أول >.

القاعدة 5: تجنب نقص usize

عند استخدام حدود شاملة وطرح 1، احذر من `mid == 0`. النطاق نصف-المفتوح يقلل هذا الخطر.

٦.٥ خاتمة الفصل

البحث الثنائي بسيط في فكرته وصارم في تنفيذه.
في هذا الفصل تعلمنا:

- البحث الكلاسيكي باستخدام نطاق نصف-مفتوح.
- Lower/Upper Bound لمعالجة التكرار والحساب.
- البحث على الإجابة باستخدام نمط `first_true`.
- البحث في مصفوفة مدوّرة.
- قائمة انضباط لتجنب أخطاء الحدود وتجاوز السعة.

في الفصل التالي سننتقل إلى خوارزميات الفرز والتقسيم والاختيار، حيث تكون الثوابت وضبط الحدود بنفس الأهمية.

الفصل ٦: الفرز والاختيار

٦.١ sort مقابل sort_unstable

توفر لغة Rust طريقتين أساسيتين لفرز الشرائح في المكان:

- sort() و sort_by(...): فرز مستقر
- sort_unstable() و sort_unstable_by(...): فرز غير مستقر

كلاهما يعتمد على المقارنة ولهما نفس ضمان التعقيد:

- الزمن: $O(n \log n)$ في المتوسط وأسوأ الحالات
- الذاكرة: بين $O(1)$ و $O(n)$ حسب استراتيجية الاستقرار

ماذا يعني الاستقرار عمليًا

الفرز المستقر يحافظ على الترتيب النسبي للعناصر ذات المفاتيح المتساوية.
إذا قمنا بفرز سجلات حسب مفتاح، وتكرر المفتاح:

- الفرز المستقر يحافظ على ترتيبها الأصلي،
- الفرز غير المستقر قد يعيد ترتيبها عشوائيًا.

قرار هندسي

- استخدم sort عند الحاجة إلى فرز متعدد المراحل (مفتاح ثانوي ثم رئيسي).
- استخدم sort_unstable عندما لا يهم الاستقرار وتريد أداءً أفضل عادة.

مثال توضيحي للاستقرار

```rust
#[derive(Clone, Debug, PartialEq, Eq)]
struct Item {
    key: i32,
    id: i32,
}

pub fn stable_vs_unstable_demo() -> (Vec<Item>, Vec<Item>) {
    let mut a = vec![
        Item { key: 1, id: 10 },
        Item { key: 2, id: 20 },
        Item { key: 1, id: 11 },
        Item { key: 2, id: 21 },
        Item { key: 1, id: 12 },
    ];

    let mut b = a.clone();

    a.sort_by_key(|x| x.key);
    b.sort_unstable_by_key(|x| x.key);

    (a, b)
}
```

٢.٦ المقارنات المخصصة

غالبًا لا يكون الفرز حسب الترتيب الطبيعي فقط، بل:

- حسب حقل معين،
- حسب عدة مفاتيح،
- حسب قيمة محسوبة،
- حسب ترتيب جزئي (بحذر).

عقد المقارن

يجب أن يحقق المقارن ترتيبًا كليًا:

- قابلية الانتقال،
- عدم التناقض،
- الاتساق.

الفرز بعدة مفاتيح

```
#[derive(Clone, Debug, PartialEq, Eq)]
struct Person {
    last: &'static str,
    first: &'static str,
    age: u32,
}

pub fn sort_people(a: &mut [Person]) {
```

```
    a.sort_by(|p, q| {
        p.last.cmp(q.last)
            .then(p.first.cmp(q.first))
            .then(p.age.cmp(&q.age))
    });
}
```

الفرز باستخراج مفتاح

```
pub fn sort_by_abs(a: &mut [i32]) {
    a.sort_by_key(|x| x.abs());
}
```

فرز الأعداد العشرية

f64 لا يطبق Ord بسبب NaN. إذا كان المجال يضمن عدم وجود NaN:

```
pub fn sort_f64_no_nan(a: &mut [f64]) {
    a.sort_by(|x, y| x.partial_cmp(y).unwrap());
}
```

٣.٦ Quickselect العنصر (k)

الفرز الكامل يكلف $O(n \log n)$. الاختيار يمكن أن يتم بمتوسط $O(n)$ باستخدام Quickselect.

العقد

بعد التنفيذ:

- العنصر عند الفهرس k هو العنصر الذي سيكون في هذا الموضع بعد الفرز،
- العناصر قبله $\leq$،
- العناصر بعده $\geq$،
- دون ضمان ترتيب كامل.

التنفيذ

```rust
pub fn quickselect_kth(a: &mut [i32], k: usize) -> i32 {
    assert!(k < a.len());

    let mut lo = 0usize;
    let mut hi = a.len();

    while hi - lo > 1 {
        let pivot_index = lo + (hi - lo) / 2;
        let p = partition_around(a, lo, hi, pivot_index);

        if k == p {
            return a[p];
        } else if k < p {
            hi = p;
        } else {
            lo = p + 1;
        }
```

```
    }
    a[lo]
}

fn partition_around(a: &mut [i32], lo: usize, hi: usize, pivot_index: usize)
↪   -> usize {
    a.swap(pivot_index, hi - 1);
    let pivot = a[hi - 1];

    let mut store = lo;
    for i in lo..(hi - 1) {
        if a[i] < pivot {
            a.swap(store, i);
            store += 1;
        }
    }
    a.swap(store, hi - 1);
    store
}
```

٤.٦ Top-K باستخدام BinaryHeap

نمط Top-K شائع جداً.
استراتيجيتان أساسيتان:

- دفع جميع العناصر ثم سحب k عناصر.
- استخدام كومة بحجم ثابت k (أفضل عندما k صغير مقارنة بـ n).

الطريقة البسيطة

```rust
use std::collections::BinaryHeap;

pub fn top_k_simple(a: &[i32], k: usize) -> Vec<i32> {
    let mut heap = BinaryHeap::new();
    for &x in a {
        heap.push(x);
    }
    let k = k.min(heap.len());
    (0..k).filter_map(|_| heap.pop()).collect()
}
```

كومة بحجم ثابت

```rust
use std::cmp::Reverse;
use std::collections::BinaryHeap;

pub fn top_k_streaming(a: &[i32], k: usize) -> Vec<i32> {
    if k == 0 { return vec![]; }

    let mut heap: BinaryHeap<Reverse<i32>> = BinaryHeap::new();

    for &x in a {
        if heap.len() < k {
            heap.push(Reverse(x));
        } else if let Some(&Reverse(min_k)) = heap.peek() {
            if x > min_k {
                heap.pop();
```

```
                heap.push(Reverse(x));
            }
        }
    }

    let mut out: Vec<i32> = heap.into_iter().map(|r| r.0).collect();
    out.sort_unstable_by(|x, y| y.cmp(x));
    out
}
```

٥.٦ الموازنة بين الاستقرار والأداء

متى تحتاج الاستقرار

- فرز متعدد المراحل.
- الحفاظ على ترتيب الإدخال بين العناصر المتساوية.
- مخرجات حتمية للاختبارات أو التدقيق.

متى تفضل عدم الاستقرار

- المفاتيح فريدة.
- لا يهم ترتيب العناصر المتساوية.
- الأداء أهم من الحفاظ على الترتيب.

الاختيار مقابل الفرز

- إذا كنت تحتاج وسيطًا أو حدًا فقط ← استخدم **Quickselect**.
- إذا كنت تحتاج أعلى k عناصر و k صغير ← استخدم كومة بحجم k.
- إذا كنت تحتاج ترتيبًا كاملاً ← استخدم الفرز.

٦.٦ خاتمة الفصل

في هذا الفصل:

- فهمت الفرق الهندسي بين sort المستقر و sort_unstable.
- بنيت مقارنات مخصصة آمنة باستخدام نمط cmp().then(...).
- نفذت Quickselect لاختيار العنصر k بمتوسط زمن خطي.
- طبقت Top-K باستخدام BinaryHeap.
- حددت الموازنة بين الاستقرار والأداء والترتيب الجزئي.

في الفصول القادمة سننتقل إلى خوارزميات تعتمد على البنى التجميعية والرسوم البيانية والبرمجة الديناميكية بنفس الانضباط: عقد واضح، إدارة دقيقة للذاكرة، واختبارات قوية.

الفصل ٧: الخوارزميات الجشعة & الكومة (Heap & Greedy Algorithms)

١.٧ الداخلية (Internals) لبنية BinaryHeap

الكومة الثنائية هي شجرة ثنائية كاملة يتم تخزينها داخل مصفوفة متجاورة في الذاكرة. في Rust يتم توفيرها عبر std::collections::BinaryHeap<T>.
وهي كومة عظمى (max-heap) بشكل افتراضي:

- peek() يعيد أكبر عنصر (وفق Ord),
- pop() يزيل أكبر عنصر،
- push(x) يضيف عنصرًا ويعيد ترتيب الكومة.

تمثيل المصفوفة

عند تخزين الكومة في مصفوفة تبدأ من الصفر v:

- فهرس الأب: `(i - 1) / 2`
- فهرس الابن الأيسر: `2*i + 1`

- فهرس الابن الأيمن: `2*i + 2`

ثابت الكومة (heap invariant) في الكومة العظمى:

$$v[\mathsf{parent}(i)] \geq v[i]$$

لكل أبناء صالحين.

تكلفة العمليات

- `peek`: بزمن $O(1)$
- `push`: بزمن $O(\log n)$ (sift-up)
- `pop`: بزمن $O(\log n)$ (تبديل الجذر مع الأخير ثم sift-down)
- الإنشاء من `Vec` (heapify): بزمن $O(n)$

لماذا هي سريعة عمليًا؟

- الذاكرة المتجاورة تحسن cache locality،
- المسار اللوغاريتمي قصير،
- غالبًا المقارنات هي الأعلى تكلفة؛ لذلك تجنب المقارنات الباهظة داخل كومات ساخنة.

تنفيذ تعليمي: كومة عظمى بسيطة

هذا التنفيذ يشرح الداخلية (وليس بديلاً عن `std`). وهو مفيد للفهم والتصحيح.

```rust
pub struct MaxHeap<T: Ord> {
    v: Vec<T>,
}

impl<T: Ord> MaxHeap<T> {
    pub fn new() -> Self { Self { v: Vec::new() } }
    pub fn with_capacity(cap: usize) -> Self { Self { v:
    ↪  Vec::with_capacity(cap) } }

    pub fn len(&self) -> usize { self.v.len() }
    pub fn is_empty(&self) -> bool { self.v.is_empty() }

    pub fn peek(&self) -> Option<&T> {
        self.v.get(0)
    }

    pub fn push(&mut self, x: T) {
        self.v.push(x);
        self.sift_up(self.v.len() - 1);
    }

    pub fn pop(&mut self) -> Option<T> {
        let n = self.v.len();
        if n == 0 { return None; }
        if n == 1 { return self.v.pop(); }

        self.v.swap(0, n - 1);
        let out = self.v.pop();
        self.sift_down(0);
```

```
        out
    }

    fn sift_up(&mut self, mut i: usize) {
        while i > 0 {
            let p = (i - 1) / 2;
            if self.v[p] >= self.v[i] { break; }
            self.v.swap(p, i);
            i = p;
        }
    }

    fn sift_down(&mut self, mut i: usize) {
        let n = self.v.len();
        loop {
            let l = 2 * i + 1;
            let r = 2 * i + 2;
            if l >= n { break; }

            let mut best = l;
            if r < n && self.v[r] > self.v[l] {
                best = r;
            }

            if self.v[i] >= self.v[best] { break; }
            self.v.swap(i, best);
            i = best;
        }
    }
```

```
}

#[cfg(test)]
mod tests_maxheap {
    use super::MaxHeap;

    #[test]
    fn pushes_and_pops_in_order() {
        let mut h = MaxHeap::new();
        h.push(3);
        h.push(1);
        h.push(5);
        h.push(2);

        assert_eq!(h.pop(), Some(5));
        assert_eq!(h.pop(), Some(3));
        assert_eq!(h.pop(), Some(2));
        assert_eq!(h.pop(), Some(1));
        assert_eq!(h.pop(), None);
    }
}
```

المساحة والملكية (Space and ownership)

الكومة تمتلك عناصرها. إذا أردت تجنب نقل (move) تراكيب كبيرة بشكل متكرر:

- خزّن مفاتيح خفيفة الوزن (lightweight keys),
- خزّن فهارس (indices) تشير إلى مصفوفة خارجية،

- استخدم Box<T> أو Arc<T> عندما تكون مشاركة الملكية ضرورية.

٢.٧ ترتيب مخصص (Custom Ordering)

تعتمد الكومة في Rust على تنفيذ Ord لنوع العنصر. لتخصيص الترتيب، غالبًا ستفعل أحد التالي:

- تغليف العنصر في struct وتطبيق Ord،
- استخدام std::cmp::Reverse<T> للحصول على سلوك min-heap،
- تخزين tuples بحيث يكون العنصر الأول هو مفتاح الأولوية.

كومة صغرى باستخدام Reverse

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

pub fn min_heap_demo(values: &[i32]) -> Vec<i32> {
    let mut h: BinaryHeap<Reverse<i32>> = BinaryHeap::new();
    for &x in values {
        h.push(Reverse(x));
    }
    let mut out = Vec::new();
    while let Some(Reverse(x)) = h.pop() {
        out.push(x);
    }
    out
}

#[cfg(test)]
```

```rust
mod tests_minheap {
    use super::min_heap_demo;

    #[test]
    fn ascending_output() {
        let out = min_heap_demo(&[3, 1, 4, 1, 5]);
        assert_eq!(out, vec![1, 1, 3, 4, 5]);
    }
}
```

عنصر بأولوية مخصصة (Custom priority item)

مثال: جدولة المهام وفق أولوية أعلى، مع كسر التعادل (tie-break) بالأقدمية حسب الطابع الزمني.

مهم: BinaryHeap هي max-heap، لذا يجب أن يكون Ord بحيث: bigger = higher priority.

```rust
use std::cmp::Ordering;

#[derive(Clone, Debug, Eq, PartialEq)]
pub struct Task {
    pub priority: i32,
    pub created_at: u64, /* smaller means older */
    pub id: u64,
}

impl Ord for Task {
    fn cmp(&self, other: &Self) -> Ordering {
        /* higher priority first */
        self.priority.cmp(&other.priority)
```

```
            /* older first for equal priority */
            .then_with(|| other.created_at.cmp(&self.created_at))
            /* stable-ish tie-break */
            .then_with(|| other.id.cmp(&self.id))
    }
}

impl PartialOrd for Task {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        Some(self.cmp(other))
    }
}

#[cfg(test)]
mod tests_task_order {
    use super::Task;
    use std::collections::BinaryHeap;

    #[test]
    fn higher_priority_first_then_older() {
        let mut h = BinaryHeap::new();
        h.push(Task { priority: 5, created_at: 100, id: 1 });
        h.push(Task { priority: 5, created_at:  90, id: 2 }); /* older */
        h.push(Task { priority: 9, created_at: 200, id: 3 });

        let a = h.pop().unwrap();
        assert_eq!(a.priority, 9);

        let b = h.pop().unwrap();
```

```
        assert_eq!(b.id, 2); /* older among priority=5 */
    }
}
```

تجنب أخطاء المقارنات (Avoiding comparator bugs)

يجب أن يكون ترتيبك ترتيبًا كليًا (total order):

- متسق (consistent),
- تعدّي (transitive),
- بدون قيم NaN في الأعداد العائمة إلا مع معالجة صريحة.

تنفيذ Ord غير صحيح قد يجعل الكومة تتصرف بشكل خاطئ.

٣.٧ جدولة المهام وتحديد الأولويات (Task Scheduling and Prioritization)

جدولة المهام من أكثر الاستخدامات العملية للكومات. أنت تختار باستمرار next best task وفق سياسة:

- أقرب موعد نهائي أولاً (earliest deadline first),
- أقصر مهمة أولاً (shortest job first),
- أعلى أولوية أولاً (highest priority first),
- أفضل نسبة (profit/cost)، وغيرها.

النمط A: المعالجة وفق أقرب موعد نهائي

نُمثل الموعد النهائي ككومة صغرى باستعمال `Reverse(deadline)`.
مثال: مهام لها deadline و duration، نتحقق هل يمكن تنفيذها ضمن المواعيد إذا رتبناها حسب الموعد النهائي.

```rust
use std::cmp::Reverse;
use std::collections::BinaryHeap;

#[derive(Clone, Debug)]
pub struct Job {
    pub deadline: i64,
    pub duration: i64,
}

/* simple feasibility: run jobs in earliest-deadline order */
pub fn feasible_by_deadline(jobs: &[Job]) -> bool {
    let mut h: BinaryHeap<Reverse<(i64, i64)>> = BinaryHeap::new();
    for j in jobs {
        h.push(Reverse((j.deadline, j.duration)));
    }

    let mut t = 0i64;
    while let Some(Reverse((d, dur))) = h.pop() {
        t += dur;
        if t > d { return false; }
    }
    true
}
```

```rust
#[cfg(test)]
mod tests_deadlines {
    use super::{feasible_by_deadline, Job};

    #[test]
    fn feasibility() {
        let ok = [
            Job { deadline: 3, duration: 1 },
            Job { deadline: 5, duration: 2 },
            Job { deadline: 6, duration: 2 },
        ];
        assert!(feasible_by_deadline(&ok));

        let bad = [
            Job { deadline: 3, duration: 2 },
            Job { deadline: 4, duration: 3 },
        ];
        assert!(!feasible_by_deadline(&bad));
    }
}
```

النمط B: جدولة ديناميكية باستخدام قائمة أولوية

نمط أنظمة واقعي: تصل المهام وتقوم دائمًا بإرسال الأعلى أولوية.

```rust
use std::collections::BinaryHeap;

pub fn dispatch_order(tasks: &[super::Task]) -> Vec<u64> {
    let mut h: BinaryHeap<super::Task> = BinaryHeap::new();
```

```
    for t in tasks {
        h.push(t.clone());
    }

    let mut out = Vec::with_capacity(tasks.len());
    while let Some(t) = h.pop() {
        out.push(t.id);
    }
    out
}
```

النمط C: الاحتفاظ بأفضل k مهام (Top-k)

شائع في القياسات (telemetry): احتفظ بأفضل k أحداث شوهدت حتى الآن.

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

pub fn keep_top_k(values: &[i32], k: usize) -> Vec<i32> {
    if k == 0 { return vec![]; }

    let mut h: BinaryHeap<Reverse<i32>> = BinaryHeap::new(); /* min-heap of
    ↪  size k */
    for &x in values {
        if h.len() < k {
            h.push(Reverse(x));
        } else if let Some(&Reverse(min_k)) = h.peek() {
            if x > min_k {
                h.pop();
```

```
                h.push(Reverse(x));
            }
        }
    }

    let mut out: Vec<i32> = h.into_iter().map(|r| r.0).collect();
    out.sort_unstable_by(|a, b| b.cmp(a));
    out
}

#[cfg(test)]
mod tests_keep_top_k {
    use super::keep_top_k;

    #[test]
    fn basic() {
        let a = [5, 1, 9, 2, 9, 3, 7];
        assert_eq!(keep_top_k(&a, 3), vec![9, 9, 7]);
    }
}
```

٤.٧ تصميم الاستراتيجيات الجشعة (Designing Greedy Strategies)

الخوارزميات الجشعة تختار أفضل قرار محلي في كل خطوة. هي سريعة وأنيقة غالبًا، لكنها تتطلب تبريرًا منطقيًا.

قائمة انضباط قبل اعتماد greedy

قبل الالتزام بالنهج الجشع، يجب تحديد:

- الهدف: تعظيم/تصغير ماذا؟
- قاعدة الاختيار: ما القرار المحلي؟
- فكرة البرهان: برهان تبديل (exchange argument) أو خاصية القطع (cut property).
- الحالة: ما أقل معلومات يجب الحفاظ عليها؟

مثال كلاسيكي: جدولة الفترات (Interval Scheduling)

المطلوب: اختيار أكبر عدد من الفترات غير المتداخلة. القاعدة الجشعة: اختر دائمًا الفترة ذات النهاية الأقرب.

```rust
#[derive(Clone, Debug)]
pub struct Interval {
    pub start: i64,
    pub end: i64,
}

pub fn max_non_overlapping(mut v: Vec<Interval>) -> usize {
    v.sort_unstable_by(|a, b| a.end.cmp(&b.end).then(a.start.cmp(&b.start)));

    let mut count = 0usize;
    let mut cur_end: Option<i64> = None;

    for it in v {
        if cur_end.is_none() || it.start >= cur_end.unwrap() {
```

```
            count += 1;
            cur_end = Some(it.end);
        }
    }
    count
}

#[cfg(test)]
mod tests_intervals {
    use super::{max_non_overlapping, Interval};

    #[test]
    fn classic() {
        let v = vec![
            Interval { start: 1, end: 3 },
            Interval { start: 2, end: 4 },
            Interval { start: 3, end: 5 },
            Interval { start: 0, end: 6 },
            Interval { start: 5, end: 7 },
            Interval { start: 8, end: 9 },
        ];
        assert_eq!(max_non_overlapping(v), 4);
    }
}
```

Greedy + Heap: جدولة أكبر عدد من الدورات (Courses)

نمط شائع: نرتب حسب الموعد النهائي، ثم نضع المدد في كومة عظمى. إذا تجاوزنا الموعد النهائي، نحذف أطول مدة (إصلاح جشع).

```rust
use std::collections::BinaryHeap;

#[derive(Clone, Debug)]
pub struct Course {
    pub duration: i64,
    pub deadline: i64,
}

pub fn max_courses(mut courses: Vec<Course>) -> usize {
    courses.sort_unstable_by(|a, b| a.deadline.cmp(&b.deadline));

    let mut total = 0i64;
    let mut heap: BinaryHeap<i64> = BinaryHeap::new(); /* durations */

    for c in courses {
        total += c.duration;
        heap.push(c.duration);

        if total > c.deadline {
            if let Some(longest) = heap.pop() {
                total -= longest;
            }
        }
    }

    heap.len()
}

#[cfg(test)]
```

```rust
mod tests_courses {
    use super::{max_courses, Course};

    #[test]
    fn example() {
        let v = vec![
            Course { duration: 100, deadline: 200 },
            Course { duration: 200, deadline: 1300 },
            Course { duration: 1000, deadline: 1250 },
            Course { duration: 2000, deadline: 3200 },
        ];
        assert_eq!(max_courses(v), 3);
    }
}
```

لماذا تعمل هذه الفكرة؟ (لمحة برهانية هندسية)

- الترتيب حسب الموعد النهائي يضمن التعامل أولاً مع القيود الأشد.
- عند تجاوز الموعد، حذف أطول مدة يقلل الزمن الإجمالي بأكبر قدر.
- خطوة الإصلاح هذه تحافظ على أكبر عدد ممكن من العناصر المختارة حتى تلك اللحظة.

٥.٧ خاتمة الفصل

هذا الفصل رسّخ الكومات والخوارزميات الجشعة كأدوات هندسية قابلة للاستخدام الإنتاجي:

- فهمت نموذج التخزين بالمصفوفة وكيف يتم تنفيذ sift-up و sift-down.

- نفذت كومة عظمى تعليمية لفهم الثوابت (invariants) والتصحيح.
- أنشأت ترتيبات مخصصة عبر Ord وكسور التعادل، واستعملت Reverse للحصول على min-heap.
- طبقت الكومات على أنماط جدولة واقعية: أقرب موعد، قوائم الإرسال ذات الأولوية، وأفضل k عناصر أثناء التدفق.
- تعلمت تصميم greedy بانضباط، ورأيت نجاحين كلاسيكيين: جدولة الفترات، وجدولة الدورات المعتمدة على heap.

في الفصل القادم سننقل هذه الأفكار إلى خوارزميات الرسوم البيانية حيث تدعم الكومات والاستدلال الجشع مسارات أقصر (shortest paths) وأشجار الامتداد الدنيا (minimum spanning trees).

الفصل ٨: الأنماط الأساسية في البرمجة الديناميكية (Core Dynamic Programming Patterns)

١.٨ تصميم انتقالات الحالة (Designing State Transitions)

البرمجة الديناميكية (DP) هي انضباط هندسي للتعامل مع المسائل ذات الأجزاء المتداخلة. أي حل DP صحيح يُبنى عبر خمس خطوات تصميمية واضحة.

1) تعريف الحالة (State)

الحالة هي أقل قدر من المعلومات يمثل مسألة فرعية.
أمثلة شائعة:

- dp[i]: أفضل نتيجة للمقدمة حتى الموضع i،
- dp[i][j]: أفضل نتيجة باستخدام i عناصر بسعة j،
- dp[pos][flag]: أفضل نتيجة عند موضع مع وضع قيد معين.

2) تعريف الانتقال (Transition)

الانتقال يربط حالة بحالات سابقة:

$$dp[i] = \min / \max / \text{choices over sum}(dp[\text{previous}] + \text{cost})$$

كل خوارزمية DP هي تعداد مضبوط للخيارات الممكنة.

3) تعريف الحالات الأساسية (Base Cases)

يجب أن تغطي الحالات الأساسية:

- أصغر حجم إدخال،
- الحالات الحدّية مثل مصفوفة فارغة،
- الحالات الابتدائية للقيود.

4) اختيار ترتيب التقييم

- Bottom-up: حساب الحالات بترتيب يعتمد على التبعيات.
- Top-down: استدعاء ذاتي + تخزين (memoization) مع الانتباه لعمق الاستدعاء.

5) التحقق من الثوابت والتعقيد

- تأكد أن كل حالة مطلوبة تم حسابها قبل استخدامها،
- تجنب تجاوز السعة العددية،
- افهم تكلفة الزمن والذاكرة.

قالب عام لـ DP أحادي البعد (Bottom-up)

```rust
/* ============================
   DP Template (1D)
   ============================
   State: dp[i] = best answer for prefix i
   Base: dp[0] = ...
   Transition: dp[i] = f(dp[i-1], dp[i-2], ...)
   Order: for i in 1..=n
*/
pub fn dp_template(n: usize) -> i64 {
    if n == 0 { return 0; }

    let mut dp = vec![0i64; n + 1];
    dp[0] = 0;
    dp[1] = 1;

    for i in 2..=n {
        dp[i] = dp[i - 1] + dp[i - 2]; /* example recurrence */
    }
    dp[n]
}
```

متى تكون DP الأداة المناسبة؟

DP مرشح قوي عندما:

- توجد مسائل فرعية متداخلة،
- يمكن التعبير عن الحل كمزيج أمثل لحلول أصغر،

- الحل الجشع غير واضح الصحة،
- الاستدعاء الذاتي ينفجر بدون تخزين النتائج.

٢.٨ تحسين فيبوناتشي (Fibonacci Optimization)

فيبوناتشي مثال تعليمي كلاسيكي، لكن الانضباط الهندسي يعني:

- تجنب الاستدعاء الذاتي الساذج،
- اختيار الأنواع العددية بعناية،
- تقليل الذاكرة عند الإمكان.

الاستدعاء الساذج (نمط خاطئ)

```
pub fn fib_naive(n: u64) -> u64 {
    if n <= 1 { return n; }
    fib_naive(n - 1) + fib_naive(n - 2)
}
```

هذا زمنه أسي ويكرر العمل كثيراً.

Memoization (Top-down)

```
pub fn fib_memo(n: usize) -> u64 {
    fn go(i: usize, memo: &mut Vec<Option<u64>>) -> u64 {
        if let Some(v) = memo[i] { return v; }
        let v = if i <= 1 { i as u64 } else { go(i - 1, memo) + go(i - 2,
        ↪  memo) };
```

```
        memo[i] = Some(v);
        v
    }

    let mut memo = vec![None; n + 1];
    go(n, &mut memo)
}
```

Bottom-up تكراري

```
pub fn fib_dp(n: usize) -> u64 {
    if n <= 1 { return n as u64; }
    let mut dp = vec![0u64; n + 1];
    dp[0] = 0;
    dp[1] = 1;
    for i in 2..=n {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
    dp[n]
}
```

تحسين إلى ذاكرة ثابتة

```
pub fn fib_optimized(n: usize) -> u64 {
    if n <= 1 { return n as u64; }
    let mut a: u64 = 0;
    let mut b: u64 = 1;
    for _ in 2..=n {
```

```
        let c = a + b;
        a = b;
        b = c;
    }
    b
}
```

٣.٨ Climbing Stairs

المسألة

لديك n درجات. يمكنك الصعود خطوة أو خطوتين. كم عدد الطرق المختلفة للوصول للقمة؟

الحالة

$$dp[i] = \text{عدد الطرق للوصول إلى الدرجة } i$$

الانتقال

$$dp[i] = dp[i-1] + dp[i-2]$$

تنفيذ مُحسَّن

```
pub fn climb_stairs(n: usize) -> u64 {
    if n == 0 { return 1; }
    if n == 1 { return 1; }
```

```
    let mut a: u64 = 1;
    let mut b: u64 = 1;

    for _ in 2..=n {
        let c = a + b;
        a = b;
        b = c;
    }
    b
}
```

هذه هي فيبوناتشي بإزاحة فهرس.

House Robber ٤.٨

المسألة

مصفوفة a تمثل أموالاً في منازل. لا يمكنك سرقة منزلين متجاورين. أوجد أكبر مجموع ممكن.

الحالة

$$dp[i] = \text{أفضل مجموع من } a[0..i)$$

الانتقال

$$dp[i] = \max(dp[i-1], dp[i-2] + a[i-1])$$

تنفيذ بذاكرة ثابتة

```rust
pub fn house_robber(a: &[i64]) -> i64 {
    let mut prev2: i64 = 0;
    let mut prev1: i64 = 0;

    for &x in a {
        let take = prev2 + x;
        let skip = prev1;
        let cur = if take > skip { take } else { skip };
        prev2 = prev1;
        prev1 = cur;
    }
    prev1
}
```

هذا نموذج آلة حالتين: prev1 و prev2.

٥.٨ تقنيات تحسين الذاكرة

1) متغيرات متدحرجة (Rolling Variables)

عندما تعتمد الحالة على قيمتين سابقتين فقط. استُخدمت في فيبوناتشي و Robber. House

2) مصفوفة دائرية (Rolling Array)

إذا اعتمدت dp[i] على آخر k حالات:

```rust
pub fn dp_k_rolling(n: usize, k: usize) -> i64 {
    if n == 0 { return 0; }
```

```
    let mut ring = vec![0i64; k.max(1)];
    ring[0] = 1;

    for i in 1..=n {
        let mut s = 0i64;
        for t in 1..=k {
            if i >= t {
                s += ring[(i - t) % k];
            }
        }
        ring[i % k] = s;
    }
    ring[n % k]
}
```

3) ضغط الأبعاد (1D ← 2D)

مثال Knapsack 0/1:

```
pub fn knapsack_01(values: &[i64], weights: &[usize], cap: usize) -> i64 {
    assert_eq!(values.len(), weights.len());
    let mut dp = vec![0i64; cap + 1];

    for i in 0..values.len() {
        let w = weights[i];
        let v = values[i];

        for c in (w..=cap).rev() {
            let candidate = dp[c - w] + v;
```

```
            if candidate > dp[c] {
                dp[c] = candidate;
            }
        }
    }
    dp[cap]
}
```

التكرار العكسي يمنع استخدام العنصر نفسه أكثر من مرة.

4) اختيار Top-down أو Bottom-up بوعي

- Top-down يحسب فقط الحالات المطلوبة.
- Bottom-up أكثر أمانًا ضد مشاكل عمق الاستدعاء وغالبًا أفضل للذاكرة المؤقتة.

5) خزّن فقط ما تحتاجه

إذا كنت تحتاج القيمة النهائية فقط، لا تخزن مسار الحل. إذا كنت تحتاج إعادة البناء، خزن مؤشرات الأب أو القرارات.
قاعدة هندسية: لا تخزن معلومات إضافية دون حاجة واضحة.

٦.٨ خاتمة الفصل

هذا الفصل رسخ عقلية DP الأساسية:

- التصميم يبدأ بتعريف الحالة والانتقال بدقة، لا بالكتابة العشوائية للكود.
- فيبوناتشي أظهر التطور الكامل: استدعاء ساذج ← bottom-up ← memoization ← ذاكرة ثابتة.

- Climbing Stairs تدريب مباشر على تحويل علاقة عودية إلى كود آمن.
- House Robber نموذج كلاسيكي للاختيار بين "أخذ" و"تخطي" باستخدام حالة متدحرجة.
- تحسين الذاكرة يتم منهجيًا: متغيرات متدحرجة، مصفوفات دائرية، ضغط أبعاد، واختيار ترتيب تقييم مناسب.

في الفصول القادمة سننتقل إلى عائلات أقوى من DP: DP على الشبكات (grid DP)، DP على السلاسل الفرعية (subsequence DP)، و DP مع تحسينات أحادية الرتابة، وكلها مبنية على نفس انضباط تصميم الحالة والانتقال.

الفصل ٩: نمط حقيبة الظهر (The Knapsack Pattern)

١.٩ حقيبة الظهر 0/1

نمط Knapsack 0/1 يُنمذج قرارات يمكن اتخاذ كل عنصر فيها مرة واحدة كحد أقصى. يظهر هذا النمط في مجالات هندسية عديدة:

- اختيار الميزات ضمن ميزانية ذاكرة محددة،
- تعبئة حمولات ضمن حد حجم معين،
- اختيار مهام ضمن ميزانية زمنية،
- اختيار وحدات ضمن قيود طاقة.

المسألة

لديك n عناصر. العنصر i يمتلك:

- وزن w_i (تكلفة)،

- قيمة v_i (منفعة).

مع سعة C، اختر مجموعة عناصر تعظم مجموع القيم بحيث مجموع الأوزان $\leq C$.

حالة DP

$dp[c] =$ أكبر قيمة يمكن تحقيقها بسعة c

نقوم بتحديث dp عنصرًا بعد عنصر.

الانتقال

لكل عنصر (w, v):

$$dp[c] = \max(dp[c],\ dp[c - w] + v)$$

لكن القاعدة الانضباطية الأساسية في 0/1 Knapsack هي:

مرّر السعات بالاتجاه العكسي حتى لا يُستخدم العنصر أكثر من مرة.

تنفيذ بذاكرة 1D محسّنة

```rust
pub fn knapsack_01(values: &[i64], weights: &[usize], cap: usize) -> i64 {
    assert_eq!(values.len(), weights.len());

    let mut dp = vec![0i64; cap + 1];

    for i in 0..values.len() {
        let w = weights[i];
        let v = values[i];
```

```
        if w > cap { continue; }

        for c in (w..=cap).rev() {
            let cand = dp[c - w] + v;
            if cand > dp[c] {
                dp[c] = cand;
            }
        }
    }

    dp[cap]
}
```

إعادة بناء الحل (اختياري)

إذا احتجت معرفة العناصر المختارة:

- استخدم جدولاً ثنائي الأبعاد مع مؤشرات قرار،
- أو خزّن معلومات السلف (بحذر بسبب الكتابة فوق القيم).

قاعدة هندسية: لا تخزّن معلومات إعادة البناء إلا إذا كانت مطلوبة فعلاً.

تنفيذ 2D مرجعي (أوضح، أثقل ذاكرة)

```
pub fn knapsack_01_2d(values: &[i64], weights: &[usize], cap: usize) -> i64 {
    assert_eq!(values.len(), weights.len());
    let n = values.len();
```

```
    let mut dp = vec![vec![0i64; cap + 1]; n + 1];

    for i in 1..=n {
        let w = weights[i - 1];
        let v = values[i - 1];
        for c in 0..=cap {
            dp[i][c] = dp[i - 1][c]; /* skip */
            if w <= c {
                let cand = dp[i - 1][c - w] + v;
                if cand > dp[i][c] {
                    dp[i][c] = cand; /* take */
                }
            }
        }
    }

    dp[n][cap]
}
```

التعقيد

- الزمن: $O(n \cdot C)$
- الذاكرة: $O(C)$ (نسخة 1D) أو $O(n \cdot C)$ (نسخة 2D)

٢.٩ حقيبة الظهر غير المحدودة (Unbounded Knapsack)

في هذا النمط يمكن أخذ كل عنصر عدداً غير محدود من المرات.

التغيير الوحيد هو:

مرّر السعات بالاتجاه الأمامي للسماح بإعادة استخدام العنصر نفسه.

الانتقال

$$dp[c] = \max(dp[c],\ dp[c-w] + v)$$

لكن التحديث يصبح:
`for c in w..=cap`

التنفيذ

```rust
pub fn knapsack_unbounded(values: &[i64], weights: &[usize], cap: usize) ->
↪  i64 {
    assert_eq!(values.len(), weights.len());
    let mut dp = vec![0i64; cap + 1];

    for i in 0..values.len() {
        let w = weights[i];
        let v = values[i];
        if w > cap { continue; }

        for c in w..=cap {
            let cand = dp[c - w] + v;
            if cand > dp[c] {
                dp[c] = cand;
            }
        }
```

```
    }

    dp[cap]
}
```

ملاحظة هندسية

مشاكل تغيير العملات (Coin Change) هي شكل من أشكال Unbounded Knapsack:

- تعظيم قيمة،
- أو تقليل عدد العناصر (باستخدام min بدل max).

٣.٩ تحسين الذاكرة

تحسين الذاكرة ليس حيلة، بل تحليل تبعيات.

لماذا 1D تعمل؟

في النسخة 2D:

$$dp[i][c] = \max(dp[i-1][c],\ dp[i-1][c-w_i] + v_i)$$

إذًا الصف i يعتمد فقط على الصف $i-1$. لذلك يمكن ضغط البعد.

- 0/1: تحديث عكسي لمنع إعادة الاستخدام.
- Unbounded: تحديث أمامي للسماح بإعادة الاستخدام.

خطأ شائع: اتجاه تحديث خاطئ

- 0/1 + تحديث أمامي ← يتحول إلى Unbounded بدون قصد.

- Unbounded + تحديث عكسي ← يمنع إعادة الاستخدام.

اختبار اتجاه التحديث

```
pub fn knapsack_01_wrong(values: &[i64], weights: &[usize], cap: usize) ->
↪  i64 {
    /* WRONG: forward update turns it into unbounded behavior */
    assert_eq!(values.len(), weights.len());
    let mut dp = vec![0i64; cap + 1];
    for i in 0..values.len() {
        let w = weights[i];
        let v = values[i];
        if w > cap { continue; }
        for c in w..=cap {
            let cand = dp[c - w] + v;
            if cand > dp[c] { dp[c] = cand; }
        }
    }
    dp[cap]
}
```

٤.٩ دراسة حالة: تخصيص الموارد

السيناريو

لديك خدمة Rust على Windows يمكنها تحميل وحدات اختيارية. كل وحدة:

- تستهلك ذاكرة، (MB)
- تعطي قيمة (أداء/أمان/ميزات).

تريد أكبر فائدة ضمن ميزانية ذاكرة. هذه مسألة Knapsack 0/1 مباشرة.

النموذج

- السعة C = ميزانية الذاكرة،
- الوزن w_i = استهلاك الوحدة،
- القيمة v_i = درجة الفائدة.

تنفيذ مع إعادة بناء

```rust
#[derive(Clone, Debug)]
pub struct Module {
    pub name: &'static str,
    pub mem_mb: usize,
    pub score: i64,
}

pub fn best_module_score(mods: &[Module], budget_mb: usize) -> i64 {
    let values: Vec<i64> = mods.iter().map(|m| m.score).collect();
```

```rust
    let weights: Vec<usize> = mods.iter().map(|m| m.mem_mb).collect();
    knapsack_01(&values, &weights, budget_mb)
}

pub fn choose_modules(mods: &[Module], budget_mb: usize) -> Vec<usize> {
    let n = mods.len();
    let cap = budget_mb;

    let mut dp = vec![vec![0i64; cap + 1]; n + 1];

    for i in 1..=n {
        let w = mods[i - 1].mem_mb;
        let v = mods[i - 1].score;

        for c in 0..=cap {
            dp[i][c] = dp[i - 1][c];
            if w <= c {
                let cand = dp[i - 1][c - w] + v;
                if cand > dp[i][c] {
                    dp[i][c] = cand;
                }
            }
        }
    }

    let mut c = cap;
    let mut picked: Vec<usize> = Vec::new();
    for i in (1..=n).rev() {
        if dp[i][c] != dp[i - 1][c] {
```

```
            picked.push(i - 1);
            c -= mods[i - 1].mem_mb;
        }
    }
    picked.reverse();
    picked
}
```

ملاحظة هندسية حول الاختبار

اختبار إعادة البناء هو المرجع النهائي:

- يتحقق من عدم تجاوز الميزانية،
- يتحقق من تطابق مجموع القيم مع الناتج.

٥.٩ خاتمة الفصل

هذا الفصل رسّخ نمط حقيبة الظهر كأداة هندسية عامة:

- 0/1 Knapsack لنمذجة اختيارات لمرة واحدة باستخدام تحديث عكسي.
- Unbounded Knapsack للاختيارات قابلة للتكرار باستخدام تحديث أمامي.
- ضغط الذاكرة يعتمد على تحليل التبعيات وليس على الحدس.
- اتجاه التحديث يحدد طبيعة المسألة فعليًا.
- دراسة حالة تخصيص الوحدات أظهرت كيف يتحول النمط النظري إلى أداة قرار عملية.

ستجد هذا النمط في كل مكان: ميزانيات، حصص، تخطيط سعات، اختيار ميزات، تعبئة موارد، وتحسينات مقيدة في الأنظمة البرمجية.

الفصل ١٠: تمثيل الرسوم البيانية في Rust

١.١٠ نمذجة قائمة المجاورات (Adjacency List)

في الأنظمة الواقعية، معظم الرسوم البيانية تكون متناثرة .(Sparse) لذلك فإن تمثيل قائمة المجاورات هو الخيار الافتراضي:

- الذاكرة تتناسب مع عدد الحواف وليس مع V^2،
- الاجتياز (Traversal) فعال وصديق للذاكرة المخبئية إذا خُزّن بشكل متجاور،
- من السهل إرفاق بيانات إضافية بكل حافة.

النموذج الأساسي

لنفرض أن الرؤوس مرقمة 0..n. قائمة المجاورات:

$$g[u] = \{v \mid (u \leftarrow v) \in E\}$$

في Rust التمثيل القياسي هو:

```
pub type Graph = Vec<Vec<usize>>;
```

بناء رسم غير موجه (Undirected)

```
pub fn build_undirected(n: usize, edges: &[(usize, usize)]) ->
↪  Vec<Vec<usize>> {
    let mut g = vec![Vec::<usize>::new(); n];
    for &(u, v) in edges {
        assert!(u < n && v < n);
        g[u].push(v);
        g[v].push(u);
    }
    g
}
```

تحسينات هندسية

في الإنتاج:

- احسب درجات الرؤوس مسبقًا لتخصيص السعة المناسبة،
- رتّب الجيران إذا كنت تحتاج ترتيبًا حتميًا،
- أزل التكرارات إن كان الإدخال قد يحتوي حواف مكررة.

تخصيص مسبق حسب الدرجة

```
pub fn build_undirected_prealloc(n: usize, edges: &[(usize, usize)]) ->
↪  Vec<Vec<usize>> {
    let mut deg = vec![0usize; n];
    for &(u, v) in edges {
        assert!(u < n && v < n);
```

```
        deg[u] += 1;
        deg[v] += 1;
    }

    let mut g: Vec<Vec<usize>> =
        (0..n).map(|i| Vec::with_capacity(deg[i])).collect();

    for &(u, v) in edges {
        g[u].push(v);
        g[v].push(u);
    }
    g
}
```

متى لا تكفي قائمة المجاورات؟

إذا كنت تحتاج:

- اختبار وجود حافة بين أي رأسين بزمن ثابت،
- رسمًا شديد الكثافة،

فقد تكون مصفوفة المجاورات أو Bitset أنسب. هذا الدليل يركز على الرسوم المتناثرة.

٢.١٠ الرسم الموجّه مقابل غير الموجّه

الرسم الموجّه يحتوي حواف باتجاه:

$$u \leftarrow v$$

الرسم غير الموجّه يعني:

$$u \leftarrow v$$

الفرق في التمثيل

- الموجّه: خزّن فقط u -> v في g[u]
- غير الموجّه: خزّن الاتجاهين

بناء رسم موجّه

```rust
pub fn build_directed(n: usize, edges: &[(usize, usize)]) -> Vec<Vec<usize>>
↪ {
    let mut g = vec![Vec::<usize>::new(); n];
    for &(u, v) in edges {
        assert!(u < n && v < n);
        g[u].push(v);
    }
    g
}
```

عمليات تتأثر بالاتجاه

- درجة الدخول (in-degree) تختلف عن درجة الخروج،
- إمكانية الوصول غير متناظرة،
- خوارزميات مثل Sort Topological و SCC تتطلب رسمًا موجّهًا.

حساب درجات الدخول

```
pub fn indegrees(g: &[Vec<usize>]) -> Vec<usize> {
    let n = g.len();
    let mut indeg = vec![0usize; n];
    for u in 0..n {
        for &v in &g[u] {
            indeg[v] += 1;
        }
    }
    indeg
}
```

٣.١٠ الرسوم الموزونة (Weighted Graphs)

في الرسم الموزون، كل حافة تحمل وزنًا (تكلفة، مسافة، سعة).

تمثيل قياسي

```
pub type WGraph = Vec<Vec<(usize, i64)>>;
```

بناء رسم موجّه موزون

```
pub fn build_weighted_directed(
    n: usize,
    edges: &[(usize, usize, i64)]
) -> Vec<Vec<(usize, i64)>> {
    let mut g = vec![Vec::<(usize, i64)>::new(); n];
```

```
    for &(u, v, w) in edges {
        assert!(u < n && v < n);
        g[u].push((v, w));
    }
    g
}
```

بناء رسم غير موجّه موزون

```
pub fn build_weighted_undirected(
    n: usize,
    edges: &[(usize, usize, i64)]
) -> Vec<Vec<(usize, i64)>> {
    let mut g = vec![Vec::<(usize, i64)>::new(); n];
    for &(u, v, w) in edges {
        assert!(u < n && v < n);
        g[u].push((v, w));
        g[v].push((u, w));
    }
    g
}
```

اختيار نوع الوزن

- i64 للتكاليف العامة،
- u64 للمسافات غير السالبة،
- تجنب الفاصلة العائمة في الخوارزميات الأساسية إن أمكن.

نمط Infinity آمن

```
pub const INF: i64 = i64::MAX / 4;

pub fn relax_example(dist_u: i64, w: i64, dist_v: &mut i64) {
    if dist_u >= INF { return; }
    let cand = dist_u + w;
    if cand < *dist_v {
        *dist_v = cand;
    }
}
```

٤.١٠ قائمة الحواف (Edge List)

بعض الخوارزميات تعمل طبيعيًا مع قائمة حواف:

```
#[derive(Clone, Copy, Debug)]
pub struct Edge {
    pub u: usize,
    pub v: usize,
    pub w: i64,
}
```

يمكن الاحتفاظ بالتمثيلين معًا:

- قائمة مجاورات للاجتياز،
- قائمة حواف للمعالجة الشاملة.

٥.١٠ تمثيل مضغوط CSR

التمثيل **Compressed Sparse Row** يقلل عدد التخصيصات ويحسّن التوضع في الذاكرة.

```rust
#[derive(Clone, Debug)]
pub struct CSRWeighted {
    pub n: usize,
    pub off: Vec<usize>,   /* length n+1 */
    pub to: Vec<usize>,    /* length m */
    pub w: Vec<i64>,       /* length m */
}

impl CSRWeighted {
    pub fn from_edges_directed(
        n: usize,
        edges: &[(usize, usize, i64)]
    ) -> Self {
        let mut deg = vec![0usize; n];
        for &(u, v, _) in edges {
            assert!(u < n && v < n);
            deg[u] += 1;
        }

        let mut off = vec![0usize; n + 1];
        for i in 0..n {
            off[i + 1] = off[i] + deg[i];
        }

        let m = edges.len();
        let mut to = vec![0usize; m];
```

```
        let mut w = vec![0i64; m];
        let mut cur = off[..n].to_vec();

        for &(u, v, ww) in edges {
            let idx = cur[u];
            to[idx] = v;
            w[idx] = ww;
            cur[u] += 1;
        }

        Self { n, off, to, w }
    }

    pub fn neighbors(
        &self,
        u: usize
    ) -> impl Iterator<Item = (usize, i64)> + '_ {
        let a = self.off[u];
        let b = self.off[u + 1];
        (a..b).map(move |i| (self.to[i], self.w[i]))
    }
}
```

٦.١٠ خاتمة الفصل

في هذا الفصل:

- تعرّفت على قائمة المجاورات كتمثيل افتراضي للرسوم المتناثرة.

- فرّقت بين الرسم الموجّه وغير الموجّه وتأثير ذلك على الدرجات وإمكانية الوصول.
- وسّعت النموذج إلى الرسوم الموزونة مع انضباط في اختيار نوع الوزن والتعامل مع اللانهاية.
- تعرّفت على تمثيل CSR لتحسين الأداء في الرسوم الكبيرة جدًا.

بهذه البنية الصلبة للتمثيل، يمكن الانتقال بثقة إلى خوارزميات الاجتياز والمسارات الأقصر والأشجار الممتدة الدنيا مع ضمان أداء ومنهجية واضحة.

الفصل ١١: البحث بعرض المستوى (BFS) والبحث بالعمق (DFS)

١١.١ البحث بعرض المستوى (Breadth-First Search)

يستكشف BFS الرسم البياني على شكل طبقات:

- يبدأ بالرأس الابتدائي (مسافة 0)،
- ثم جميع الجيران على مسافة 1،
- ثم مسافة 2، وهكذا.

في الرسوم غير الموزونة، يحسب BFS أقصر مسافة بعدد الحواف.

البنية الأساسية

يتطلب BFS:

- طابور (FIFO)،
- مصفوفة زيارة أو مسافات،

- قائمة مجاورات.

في Rust نستخدم VecDeque.

حساب المسافات من مصدر واحد

```
use std::collections::VecDeque;

pub fn bfs_distances(g: &[Vec<usize>], start: usize) -> Vec<i32> {
    let n = g.len();
    assert!(start < n);

    let mut dist = vec![-1i32; n];
    let mut q: VecDeque<usize> = VecDeque::new();

    dist[start] = 0;
    q.push_back(start);

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

    dist
}
```

إعادة بناء المسار الأقصر

```rust
use std::collections::VecDeque;

pub fn bfs_shortest_path(
    g: &[Vec<usize>],
    start: usize,
    goal: usize
) -> Option<Vec<usize>> {
    let n = g.len();
    assert!(start < n && goal < n);

    let mut dist = vec![-1i32; n];
    let mut parent: Vec<Option<usize>> = vec![None; n];
    let mut q: VecDeque<usize> = VecDeque::new();

    dist[start] = 0;
    q.push_back(start);

    while let Some(u) = q.pop_front() {
        if u == goal { break; }
        let du = dist[u];
        for &v in &g[u] {
            if dist[v] == -1 {
                dist[v] = du + 1;
                parent[v] = Some(u);
                q.push_back(v);
            }
        }
    }
```

```
    if dist[goal] == -1 { return None; }

    let mut path = Vec::new();
    let mut cur = goal;
    path.push(cur);
    while cur != start {
        cur = parent[cur].unwrap();
        path.push(cur);
    }
    path.reverse();
    Some(path)
}
```

ملاحظات هندسية

- استخدام -1 كعلامة عدم زيارة.
- BFS آمن من ناحية المكدس لأنه تكراري.
- في الرسوم الكبيرة، يُفضَّل تمثيل CSR لتحسين التوضع في الذاكرة.

٢.١١ البحث بالعمق (Depth-First Search)

DFS يستكشف أعمق مسار ممكن قبل الرجوع.
يستخدم في:

- التحقق من إمكانية الوصول،

- المكونات المتصلة،
- الفرز الطوبولوجي،
- كشف الدورات،
- الجسور ونقاط القطع (متقدم).

تنفيذ تكراري آمن

```
pub fn dfs_iterative(g: &[Vec<usize>], start: usize) -> Vec<usize> {
    let n = g.len();
    assert!(start < n);

    let mut seen = vec![false; n];
    let mut order = Vec::new();
    let mut st: Vec<usize> = Vec::new();
    st.push(start);

    while let Some(u) = st.pop() {
        if seen[u] { continue; }
        seen[u] = true;
        order.push(u);

        for &v in g[u].iter().rev() {
            if !seen[v] {
                st.push(v);
            }
        }
    }
```

```
    order
}
```

DFS مع تتبع الآباء

```
pub fn dfs_with_parent(
    g: &[Vec<usize>],
    start: usize
) -> (Vec<Option<usize>>, Vec<usize>) {
    let n = g.len();
    assert!(start < n);

    let mut parent = vec![None; n];
    let mut seen = vec![false; n];
    let mut disc = Vec::new();

    let mut st: Vec<(usize, usize)> = Vec::new();
    st.push((start, 0));

    while let Some((u, idx)) = st.pop() {
        if !seen[u] {
            seen[u] = true;
            disc.push(u);
        }

        if idx < g[u].len() {
            let v = g[u][idx];
            st.push((u, idx + 1));
```

```
            if !seen[v] {
                parent[v] = Some(u);
                st.push((v, 0));
            }
        }
    }

    (parent, disc)
}
```

٣.١١ المكونات المتصلة

في الرسم غير الموجّه، المكوّن المتصل هو مجموعة رؤوس يمكن لكل منها الوصول إلى الآخر.

التنفيذ باستخدام BFS

```
use std::collections::VecDeque;

pub fn connected_components(g: &[Vec<usize>]) -> Vec<Vec<usize>> {
    let n = g.len();
    let mut seen = vec![false; n];
    let mut comps = Vec::new();

    for s in 0..n {
        if seen[s] { continue; }

        let mut comp = Vec::new();
        let mut q = VecDeque::new();
```

```
        seen[s] = true;
        q.push_back(s);

        while let Some(u) = q.pop_front() {
            comp.push(u);
            for &v in &g[u] {
                if !seen[v] {
                    seen[v] = true;
                    q.push_back(v);
                }
            }
        }

        comps.push(comp);
    }

    comps
}
```

تمثيل أكثر قابلية للتوسع

```
use std::collections::VecDeque;

pub fn component_ids(g: &[Vec<usize>]) -> Vec<usize> {
    let n = g.len();
    let mut seen = vec![false; n];
    let mut id = vec![usize::MAX; n];
    let mut cur_id = 0usize;
```

```
    for s in 0..n {
        if seen[s] { continue; }

        let mut q = VecDeque::new();
        seen[s] = true;
        id[s] = cur_id;
        q.push_back(s);

        while let Some(u) = q.pop_front() {
            for &v in &g[u] {
                if !seen[v] {
                    seen[v] = true;
                    id[v] = cur_id;
                    q.push_back(v);
                }
            }
        }

        cur_id += 1;
    }

    id
}
```

٤.١١ كشف الدورات

في الرسم غير الموجّه

الدورة موجودة إذا وجدنا رأسًا مزورًا ليس هو الأب.

```rust
pub fn has_cycle_undirected(g: &[Vec<usize>]) -> bool {
    let n = g.len();
    let mut seen = vec![false; n];

    for s in 0..n {
        if seen[s] { continue; }

        let mut st: Vec<(usize, usize)> = Vec::new();
        st.push((s, usize::MAX));

        while let Some((u, p)) = st.pop() {
            if seen[u] { continue; }
            seen[u] = true;

            for &v in &g[u] {
                if !seen[v] {
                    st.push((v, u));
                } else if v != p {
                    return true;
                }
            }
        }
    }
```

```
    false
}
```

في الرسم الموجّه

نستخدم ألوانًا:

- 0 = غير مزور،
- 1 = قيد المعالجة،
- 2 = مكتمل.

```
pub fn has_cycle_directed(g: &[Vec<usize>]) -> bool {
    let n = g.len();
    let mut color = vec![0u8; n];

    for s in 0..n {
        if color[s] != 0 { continue; }

        let mut st: Vec<(usize, usize)> = Vec::new();
        st.push((s, 0));

        while let Some((u, idx)) = st.pop() {
            if color[u] == 0 {
                color[u] = 1;
            }

            if idx < g[u].len() {
```

```
                let v = g[u][idx];
                st.push((u, idx + 1));

                if color[v] == 0 {
                    st.push((v, 0));
                } else if color[v] == 1 {
                    return true;
                }
            } else {
                color[u] = 2;
            }
        }
    }

    false
}
```

١١.٥ خاتمة الفصل

في هذا الفصل:

- BFS يستكشف طبقيًا ويحسب أقصر المسافات في الرسوم غير الموزونة.
- DFS يُنفَّذ تكراريًا في Rust لضمان أمان المكدس.
- المكونات المتصلة تُستخرج بتكرار BFS/DFS على الرؤوس غير المزارة.
- كشف الدورات يعتمد على نوع الرسم: تحقق الأب في غير الموجّه، وتتبع الحالة في الموجّه.

بهذه اللبنات الأساسية، يمكن الانتقال إلى خوارزميات متقدمة مثل الفرز الطوبولوجي، أقصر المسارات، والأشجار الممتدة الدنيا.

الفصل ١٢: أقصر المسارات والاتصال

١.١٢ خوارزمية Dijkstra باستخدام BinaryHeap

تحسب خوارزمية Dijkstra أقصر المسارات من مصدر واحد في رسم بياني ذي أوزان غير سالبة. وهي خوارزمية جشعة تقوم في كل خطوة بتثبيت أقرب رأس غير محسوم بعد.

الشروط المسبقة

- جميع الأوزان يجب أن تكون ≥ 0.
- يمكن أن يكون الرسم موجهاً أو غير موجه.
- في حال وجود أوزان سالبة يجب استخدام Bellman-Ford أو خوارزميات متخصصة.

نموذج البيانات

نستخدم قائمة مجاورات موزونة:

```
pub type WGraph = Vec<Vec<(usize, i64)>>; /* (to, weight) */
```

التعامل مع BinaryHeap

`BinaryHeap` في Rust هو max-heap. Dijkstra يحتاج min-heap، لذلك نستخدم `Reverse((dist, node))`.

انضباط قيمة اللانهاية

```
pub const INF: i64 = i64::MAX / 4;
```

تنفيذ Dijkstra (مسافات فقط)

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

pub fn dijkstra(g: &WGraph, start: usize) -> Vec<i64> {
    let n = g.len();
    assert!(start < n);

    let mut dist = vec![INF; n];
    dist[start] = 0;

    let mut heap: BinaryHeap<Reverse<(i64, usize)>> = BinaryHeap::new();
    heap.push(Reverse((0, start)));

    while let Some(Reverse((d, u))) = heap.pop() {
        if d != dist[u] {
            continue; /*        */
        }
```

```
        for &(v, w) in &g[u] {
            debug_assert!(w >= 0);
            if dist[u] >= INF { continue; }

            let nd = d + w;
            if nd < dist[v] {
                dist[v] = nd;
                heap.push(Reverse((nd, v)));
            }
        }
    }

    dist
}
```

إعادة بناء المسار

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

pub fn dijkstra_with_parent(
    g: &WGraph,
    start: usize
) -> (Vec<i64>, Vec<Option<usize>>) {

    let n = g.len();
    assert!(start < n);

    let mut dist = vec![INF; n];
```

```
    let mut parent = vec![None; n];
    dist[start] = 0;

    let mut heap: BinaryHeap<Reverse<(i64, usize)>> = BinaryHeap::new();
    heap.push(Reverse((0, start)));

    while let Some(Reverse((d, u))) = heap.pop() {
        if d != dist[u] { continue; }

        for &(v, w) in &g[u] {
            debug_assert!(w >= 0);
            let nd = d + w;
            if nd < dist[v] {
                dist[v] = nd;
                parent[v] = Some(u);
                heap.push(Reverse((nd, v)));
            }
        }
    }

    (dist, parent)
}

pub fn reconstruct_path(
    parent: &[Option<usize>],
    start: usize,
    goal: usize
) -> Option<Vec<usize>> {
```

```
    if start == goal { return Some(vec![start]); }

    let mut path = Vec::new();
    let mut cur = goal;

    path.push(cur);
    while cur != start {
        cur = match parent[cur] {
            Some(p) => p,
            None => return None,
        };
        path.push(cur);
    }

    path.reverse();
    Some(path)
}
```

ملاحظات هندسية

- شرط d != dist[u] يمنع معالجة إدخالات قديمة.
- العقد غير القابلة للوصول تحقق dist[v] >= INF.
- يمكن دعم مصادر متعددة بإدخال عدة عقد في البداية بمسافة 0.

٢.١٢ بنية المجموعات المنفصلة (Union-Find / DSU)

تحافظ DSU على تقسيم العناصر إلى مجموعات منفصلة. العمليات:

- `find(x)`
- `union(a, b)`
- `same(a, b)`

مع ضغط المسار والدمج حسب الحجم تصبح العمليات شبه ثابتة الزمن.

تنفيذ عملي

```rust
#[derive(Clone, Debug)]
pub struct DSU {
    parent: Vec<usize>,
    size: Vec<usize>,
}

impl DSU {
    pub fn new(n: usize) -> Self {
        let mut parent = Vec::with_capacity(n);
        let mut size = Vec::with_capacity(n);
        for i in 0..n {
            parent.push(i);
            size.push(1);
        }
        Self { parent, size }
    }

    pub fn find(&mut self, x: usize) -> usize {
        let p = self.parent[x];
        if p == x {
```

```
            x
        } else {
            let r = self.find(p);
            self.parent[x] = r;
            r
        }
    }

    pub fn union(&mut self, a: usize, b: usize) -> bool {
        let mut ra = self.find(a);
        let mut rb = self.find(b);
        if ra == rb { return false; }

        if self.size[ra] < self.size[rb] {
            core::mem::swap(&mut ra, &mut rb);
        }

        self.parent[rb] = ra;
        self.size[ra] += self.size[rb];
        true
    }

    pub fn same(&mut self, a: usize, b: usize) -> bool {
        self.find(a) == self.find(b)
    }
}
```

ملاحظات هندسية

- ممتاز لمشاكل الاتصال الثابتة في الرسوم غير الموجهة.
- لا يدعم حذف الحواف بكفاءة.
- المفاتيح غير العددية يجب تحويلها إلى معرفات عددية.

٣.١٢ أنماط مشاكل الاتصال

هل عقدتان متصلتان؟

- DSU مناسب عند كثرة الاستعلامات.
- BFS/DFS مناسب عند الحاجة لاجتياز إضافي.

عدد المكونات المتصلة باستخدام DSU

```rust
use std::collections::HashSet;

pub fn num_components_dsu(
    n: usize,
    edges: &[(usize, usize)]
) -> usize {

    let mut d = DSU::new(n);
    for &(u, v) in edges {
        d.union(u, v);
    }
```

```
    let mut roots = HashSet::new();
    for i in 0..n {
        roots.insert(d.find(i));
    }

    roots.len()
}
```

أقصر مسار غير موزون

استخدم BFS.

أقصر مسار موزون

استخدم Dijkstra بشرط عدم وجود أوزان سالبة.

اتصال تدريجي مع الزمن

```
#[derive(Clone, Copy)]
pub struct TimedEdge {
    pub t: i64,
    pub u: usize,
    pub v: usize,
}

pub fn earliest_connection(
    n: usize,
    mut edges: Vec<TimedEdge>,
```

```
    a: usize,
    b: usize
) -> Option<i64> {

    edges.sort_unstable_by(|x, y| x.t.cmp(&y.t));

    let mut d = DSU::new(n);
    for e in edges {
        d.union(e.u, e.v);
        if d.same(a, b) {
            return Some(e.t);
        }
    }

    None
}
```

٤.١٢ تحليل التعقيد

Dijkstra

باستخدام قائمة مجاورات و heap:

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

الذاكرة:

- $O(V)$ للمسافات،
- $O(E)$ للحواف،

- heap قد يحتوي حتى $O(E)$ إدخالاً.

Union-Find

مع ضغط المسار والدمج حسب الحجم:

$$O(\alpha(n))$$

عملياً تُعامل كزمن ثابت تقريبا.

استعلامات الاتصال

- BFS لكل استعلام: $O(Q(V+E))$
- DSU: بناء $O(E\alpha(V))$ ثم $O(Q\alpha(V))$

٥.١٢ خاتمة الفصل

في هذا الفصل:

- طُبِّقت Dijkstra باستخدام BinaryHeap و Reverse وتقنية الإدخالات القديمة.
- نُفِّذت DSU احترافية بضغط المسار والدمج حسب الحجم.
- رُبطت مشاكل الاتصال بالأداة المناسبة: BFS/DFS أو Dijkstra أو DSU.
- أُجري تحليل تعقيد دقيق يوضح الفرق بين الأدوات.

بهذه الأسس يمكن الانتقال إلى خوارزميات متقدمة مثل الأشجار الممتدة الدنيا، الفرز الطوبولوجي، والمكونات القوية الاتصال.

الفصل ١٣: مطابقة أنماط السلاسل النصية

١.١٣ المطابقة البسيطة (Naive Pattern Matching)

مطابقة الأنماط تجيب عن السؤال الأساسي:

أين يظهر النمط p داخل النص t؟

تعريف المشكلة

لدينا:

- نص T بطول n،
- نمط P بطول m،

نبحث عن جميع الفهارس i بحيث:

$$T[i..i+m) = P[0..m)$$

بشرط أن $i + m \leq n$.

الفكرة البسيطة

نحاول كل محاذاة ممكنة للنمط داخل النص:

- لكل موضع بداية i،
- نقارن المحارف حتى يحدث اختلاف أو تطابق كامل.

التعقيد الزمني:

$$O(n \cdot m)$$

وأسوأ حالة تحدث عند تكرار بادئات النمط كثيراً داخل النص.

ملاحظة هندسية: البايتات مقابل Unicode

نوع &str في Rust مشفر UTF-8 ولا يمكن فهرسته مباشرة بأعداد صحيحة.
في الخوارزميات:

- استخدم as_bytes() للمطابقة الدقيقة على مستوى البايت.
- استخدم chars() فقط إذا كنت تحتاج فعلاً مطابقة على مستوى Unicode.

معظم الاستخدامات النظامية (سجلات، بروتوكولات، Tokens) تعتمد على البايتات.

تنفيذ المطابقة البسيطة على مستوى البايت

```
pub fn naive_find_all(text: &[u8], pat: &[u8]) -> Vec<usize> {
    let n = text.len();
    let m = pat.len();
    if m == 0 { return (0..=n).collect(); }
    if m > n { return Vec::new(); }
```

```
    let mut out = Vec::new();

    for i in 0..=(n - m) {
        let mut ok = true;
        for j in 0..m {
            if text[i + j] != pat[j] {
                ok = false;
                break;
            }
        }
        if ok {
            out.push(i);
        }
    }

    out
}
```

متى تكون الطريقة البسيطة كافية؟

- النمط قصير.
- النص قصير.
- عدد الاستعلامات قليل.
- قابلية القراءة أهم من أسوأ حالة.

عند الحاجة إلى ضمان تعقيد خطي استخدم KMP.

٢.١٣ خوارزمية KMP

خوارزمية Knuth--Morris--Pratt تتجنب المقارنات المكررة وتضمن:

$$O(n + m)$$

الفكرة

عند حدوث اختلاف عند الموضع j في النمط، لا نعود إلى 0، بل ننتقل إلى أطول بادئة صحيحة تساوي لاحقة من الجزء المطابق.
هذه المعلومات محفوظة في دالة البادئة pi.

تنفيذ KMP

```
pub fn kmp_find_all(text: &[u8], pat: &[u8]) -> Vec<usize> {
    let n = text.len();
    let m = pat.len();

    if m == 0 { return (0..=n).collect(); }
    if m > n { return Vec::new(); }

    let pi = prefix_function(pat);
    let mut out = Vec::new();
    let mut j = 0usize;

    for i in 0..n {
        while j > 0 && text[i] != pat[j] {
            j = pi[j - 1];
        }
```

```
        if text[i] == pat[j] {
            j += 1;
        }
        if j == m {
            out.push(i + 1 - m);
            j = pi[m - 1];
        }
    }

    out
}
```

٣.١٣ فهم دالة البادئة

$\pi[i] =$ تساوي لاحقة منه$P[0..i]$طول أطول بادئة صحيحة من

تنفيذ حسابها

```
pub fn prefix_function(pat: &[u8]) -> Vec<usize> {
    let m = pat.len();
    let mut pi = vec![0usize; m];

    for i in 1..m {
        let mut j = pi[i - 1];

        while j > 0 && pat[i] != pat[j] {
            j = pi[j - 1];
```

```
        }

        if pat[i] == pat[j] {
            j += 1;
        }

        pi[i] = j;
    }

    pi
}
```

النموذج الذهني

- احتفظ بطول التطابق الحالي j.
- عند اختلاف، عد إلى pi[j-1].
- عند تطابق، زد j.

هذا هو نفس المنطق المستخدم أثناء فحص النص.

٤.١٣ استخدامات عملية

1. فحص السجلات

البحث عن توقيع خطأ داخل سجل ضخم.

2 . تحليل البروتوكولات

- علامات البداية.
- رؤوس magic.
- فواصل.

3 . استعلامات متكررة لنفس النمط

احسب pi مرة واحدة، ثم افحص كل نص في زمن خطي.

4 . اكتشاف الدورية

إذا كان:

$$k = m - \pi[m-1]$$

وكان $m \bmod k = 0$، فالنمط دوري بطول k.

```rust
pub fn smallest_period_len(s: &[u8]) -> usize {
    if s.is_empty() { return 0; }
    let pi = prefix_function(s);
    let m = s.len();
    let k = m - pi[m - 1];
    if k != 0 && m % k == 0 { k } else { m }
}
```

5 . واجهة بحث بسيطة

```
pub fn find_all_str(text: &str, pat: &str) -> Vec<usize> {
    kmp_find_all(text.as_bytes(), pat.as_bytes())
}
```

تحذير هندسي

المؤشرات المعادة هي فهارس بايت. في حال الحاجة لفهارس Unicode يجب تحويلها بعناية.

٥.١٣ خاتمة الفصل

في هذا الفصل:

- المطابقة البسيطة سهلة ولكن أسوأ حالاتها $O(nm)$.
- KMP يضمن $O(n+m)$ ويعالج التداخلات.
- دالة البادئة تمثل انتقالات آلة النمط.
- التطبيقات تشمل تحليل السجلات، البروتوكولات، الاستعلامات المتكررة، واكتشاف الدورية.

بإتقان KMP يمكن الانتقال إلى خوارزميات أكثر تقدماً مثل Z-Function، التجزئة الدوارة، ومصفوفات اللواحق.

الفصل ١٤: مشروع التخرج المصغر (Capstone Mini Project)

١.١٤ تصميم محرك استعلامات صغير

يبني هذا المشروع الختامي محرك استعلامات صغير داخل الذاكرة (in-memory) يجمع عدة تقنيات خوارزمية من هذا الدليل.
الهدف ليس منافسة قواعد البيانات، بل ممارسة الانضباط الهندسي:

- تصميم تخطيط البيانات بوعي،
- بناء فهارس قابلة لإعادة الاستخدام،
- تنفيذ الاستعلامات بكفاءة،
- قياس الأداء،
- إعادة الهيكلة نحو معمارية نظيفة.

نطاق المشروع

سننفذ مجموعة بيانات من النوع Record داخل الذاكرة، وندعم استعلامات مثل:

- **مطابقة تامة:** country == "SA",
- **مجال عددي:** age in [30, 45],
- **بحث جزئي:** name contains "man",
- **دمج شروط:** name contains "man" AND age in [30,45] AND country == "SA".

قيود التصميم

- Rust مستقر فقط، متوافق مع Windows، دون استدعاءات نظام خاصة.
- **الاعتماد على مكتبة std فقط:** HashMap، BTreeMap، البحث الثنائي، و KMP.
- التركيز على الوضوح مع تحليل الأداء.

نموذج السجل

نستخدم `String` مملوكة للبيانات طويلة العمر، مع معرف عددي ثابت.

```
#[derive(Clone, Debug)]
pub struct Record {
    pub id: u32,
    pub name: String,
    pub country: String,
    pub age: u32,
    pub score: i64,
}

impl Record {
    pub fn new(id: u32, name: &str, country: &str, age: u32, score: i64) ->
    ↪  Self {
```

```
        Self {
            id,
            name: name.to_string(),
            country: country.to_string(),
            age,
            score,
        }
    }
}
```

قرارات تخطيط البيانات

- تخزين Vec<Record> كمصدر الحقيقة.
- تخزين الفهارس كمؤشرات إلى هذا المتجه.
- بناء خرائط من القيم إلى فهارس السجلات.

هذا يقلل النسخ ويجعل الفلاتر رخيصة.

٢.١٤ دمج التجزئة، المجالات، والبحث

محرك الاستعلام عبارة عن خط أنابيب:

١. اختيار مجموعة مرشحين أولية باستخدام أكثر فهرس انتقائية.
٢. تقاطع النتائج مع الفهارس الأخرى.
٣. تطبيق الشروط غير المفهرسة (مثل البحث الجزئي).
٤. إعادة النتائج.

تمثيل مجموعة المرشحين

نستخدم `Vec<u8>` كـ bitmap (0 أو 1) لسهولة التقاطع.

```
pub fn mark_bitmap(n: usize, indices: &[usize]) -> Vec<u8> {
    let mut b = vec![0u8; n];
    for &i in indices {
        b[i] = 1;
    }
    b
}

pub fn bitmap_and_inplace(a: &mut [u8], b: &[u8]) {
    for i in 0..a.len() {
        a[i] &= b[i];
    }
}
```

٣.١٤ مرجع الصحة: التنفيذ البسيط

```
impl Engine {
    pub fn query_naive(&self, q: &Query) -> Vec<usize> {
        let mut out = Vec::new();

        for i in 0..self.records.len() {
            let r = &self.records[i];

            if let Some(ref c) = q.country_eq {
                if &r.country != c { continue; }
```

```
            }

            if let Some((lo, hi)) = q.age_range {
                if r.age < lo || r.age > hi { continue; }
            }

            if let Some(mins) = q.min_score {
                if r.score < mins { continue; }
            }

            if let Some(ref needle) = q.name_contains {
                if !kmp_contains(r.name.as_bytes(), needle.as_bytes()) {
                ↪  continue; }
            }

            out.push(i);
        }

        out
    }
}
```

٤.١٤ التنفيذ المفهرس

الفكرة:

- ابدأ بأكثر شرط انتقائية.
- ابنِ bitmap أولية.

- طبّق التقاطعات.
- افحص الشروط المتبقية.

```
impl Engine {
    pub fn query_indexed(&self, q: &Query) -> Vec<usize> {
        let n = self.records.len();
        let mut cand: Vec<u8> = vec![1u8; n];
        let mut seeded = false;

        if let Some(ref c) = q.country_eq {
            if let Some(idxs) = self.idx_country.get(c) {
                cand = mark_bitmap(n, idxs);
            } else {
                return Vec::new();
            }
            seeded = true;
        }

        if let Some((lo, hi)) = q.age_range {
            let idxs = self.indices_age_range(lo, hi);
            let b = mark_bitmap(n, &idxs);
            if seeded {
                bitmap_and_inplace(&mut cand, &b);
            } else {
                cand = b;
                seeded = true;
            }
        }
```

```
        let mut out = Vec::new();
        for i in 0..n {
            if cand[i] == 0 { continue; }
            let r = &self.records[i];

            if let Some(mins) = q.min_score {
                if r.score < mins { continue; }
            }

            if let Some(ref needle) = q.name_contains {
                if !kmp_contains(r.name.as_bytes(), needle.as_bytes()) {
                ↪  continue; }
            }

            out.push(i);
        }

        out
    }
}
```

٥.١٤ اختبار الصحة

```
#[cfg(test)]
mod tests_engine_correctness {
    use super::{Engine, Query, Record};

    #[test]
```

```
    fn naive_equals_indexed() {
        let rows = vec![
            Record::new(1, "Ayman", "SA", 41, 95),
            Record::new(2, "Hassan", "SA", 33, 70),
            Record::new(3, "Mariam", "EG", 29, 88),
            Record::new(4, "Salman", "SA", 27, 60),
        ];

        let e = Engine::new(rows);

        let mut q = Query::default();
        q.country_eq = Some("SA".into());
        q.age_range = Some((30, 45));
        q.name_contains = Some("man".into());

        assert_eq!(e.query_naive(&q), e.query_indexed(&q));
    }
}
```

٦.١٤ تقييم الأداء

```
use std::time::Instant;

pub fn time_queries(e: &Engine, q: &Query, iters: usize) -> (u128, u128) {
    let mut sink: usize = 0;

    let t0 = Instant::now();
    for _ in 0..iters {
```

```
        let r = e.query_naive(q);
        sink ^= r.len();
    }
    let naive_ns = t0.elapsed().as_nanos();

    let t1 = Instant::now();
    for _ in 0..iters {
        let r = e.query_indexed(q);
        sink ^= r.len();
    }
    let indexed_ns = t1.elapsed().as_nanos();

    if sink == usize::MAX { eprintln!("{}", sink); }

    (naive_ns, indexed_ns)
}
```

تحليل النتائج

- النسخة المفهرسة تتفوق عندما تكون الفلاتر انتقائية.
- النسخة البسيطة قد تكون كافية لمجموعات بيانات صغيرة.
- تقليل مجموعة المرشحين مبكراً هو مفتاح السرعة.

٧.١٤ إعادة الهيكلة لمعمارية نظيفة

الأهداف:

- فصل الفهارس عن التنفيذ.
- إبقاء الخوارزميات نقية وقابلة للاختبار.
- استخدام Traits لمرونة التوسعة.

واجهة فهرس عامة

```
pub trait CandidateIndex {
    fn candidates(&self, q: &Query, n: usize) -> Option<Vec<u8>>;
}
```

بهذا تصبح الفهارس مكونات مستقلة يمكن تبديلها.

٨.١٤ الخاتمة النهائية للمشروع

هذا المشروع أثبت أن المعرفة الخوارزمية تتحول إلى قدرة هندسية عند:

- بناء الفهارس بوعي،
- تقليل المرشحين مبكراً،
- اختبار الصحة دائماً بمقارنة مع تنفيذ مرجعي،
- قياس الأداء قبل الحكم،
- إعادة الهيكلة نحو تصميم نظيف وقابل للنمو.

أهم نتيجة ليست الشيفرة، بل العقلية:

> صمّم الفهارس بوعي، قلّل المساحة البحثية مبكراً، قِس الأداء، وأعد الهيكلة نحو معمارية نظيفة.

وهذه هي روح هندسة الخوارزميات العملية في Rust.

الملاحق

الملحق A: ملخص التعقيد (Complexity Cheat Sheet)

هذا الملحق هو مرجع عملي للتعقيد الزمني والذاكري للأنماط المستخدمة في هذا الدليل. ليكن:

- n حجم الإدخال (عناصر، محارف، عقد)،
- m طول النمط،
- V عدد العقد،
- E عدد الحواف،
- C السعة،
- Q عدد الاستعلامات.

معدلات النمو الأساسية (حدس عملي)

- $O(1)$: عمل ثابت
- $O(\log n)$: عملية على شجرة متوازنة أو Heap

- $O(n)$: مرور واحد
- $O(n \log n)$: فرز أو تقسيم وغزو
- $O(n^2)$: حلقة مزدوجة
- $O(2^n)$: تعداد جميع المجموعات الجزئية (يُتجنب للقيم الكبيرة)

عمليات هياكل البيانات الشائعة

- Vec:
  - الفهرسة: $O(1)$
  - push (مُهَيأ): $O(1)$
  - pop: $O(1)$
  - إدراج/حذف في الوسط: $O(n)$
  - البحث الثنائي (إذا كان مرتباً): $O(\log n)$
- HashMap (متوسط):
  - إدراج/بحث/حذف: $O(1)$ متوسط، $O(n)$ في أسوأ حالة
- BTreeMap:
  - إدراج/بحث/حذف: $O(\log n)$
  - استعلام مجال: $O(\log n + k)$ حيث k عدد النتائج
- BinaryHeap:
  - pop/push: $O(\log n)$
  - peek: $O(1)$

  - البناء من Vec: $O(n)$

- الفرز:
  - `sort_unstable`: $O(n \log n)$
  - `sort`: $O(n \log n)$ (مستقر)

أنماط الخوارزميات في هذا الدليل

- مؤشرين / نافذة منزلقة: $O(n)$
- مجموعات مسبقة (Prefix Sum):
  - البناء: $O(n)$
  - استعلام المجال: $O(1)$
- البحث الثنائي: $O(\log n)$
- مكدس رتيب (Monotonic Stack): $O(n)$
- KMP: $O(n + m)$
- DFS: / BFS $O(V + E)$
- Dijkstra: $O((V + E) \log V)$
- DSU: مُهَيأ قريب من $O(1)$
- Knapsack:
  - 0/1: $O(n \cdot C)$
  - غير محدود: $O(n \cdot C)$

قائمة فحص هندسية

- إذا كانت الخوارزمية $O(n^2)$، تأكد من حدود n الفعلية.
- انتبه للثوابت المخفية: التجزئة، التخصيصات، فقدان التوطين (cache).
- خصص مرة واحدة ومرر خطياً أفضل من تخصيصات صغيرة متكررة.

الملحق B: أخطاء Rust الشائعة في تصميم الخوارزميات

الخطأ 1: فهرسة &str مباشرة

سلاسل Rust UTF-8 لا يمكن فهرستها مباشرة كمصفوفة محارف.
استخدم:

- as_bytes() للخوارزميات المعتمدة على البايت.
- chars() إذا احتجت Unicode scalar.

```
pub fn count_ascii_a(s: &str) -> usize {
    s.as_bytes().iter().filter(|&&b| b == b'a').count()
}
```

الخطأ 2: تعقيد تربيعي غير مقصود عند دمج String

كرر التخصيص داخل حلقة يؤدي إلى $O(n^2)$. استخدم with_capacity.

الخطأ 3: نسخ clone() غير ضروري

مرر &[T] بدلاً من نسخ Vec<T>.

الخطأ 4: اتجاه Heap خاطئ

`BinaryHeap` هو Max-Heap. استخدم `Reverse` للحصول على Min-Heap.

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

الخطأ 5: فيض عددي

استخدم:

$$mid = lo + (hi - lo)/2$$

واستخدم i64 للمسافات والمجاميع.

الخطأ 6: عمق الاستدعاء العودي

DFS العودي قد يسبب Stack Overflow. يفضل التنفيذ التكراري.

الملحق C: أفضل ممارسات الذاكرة والتخصيص

القاعدة 1: احجز السعة مسبقاً

```
let mut v = Vec::with_capacity(n);
```

القاعدة 2: فضّل التخزين المتجاور

`Vec<T>` أفضل توطيناً من هياكل متفرقة.

القاعدة 3: أعد استخدام المخازن

```
buf.clear();
buf.extend_from_slice(data);
```

القاعدة 4: اختر الهيكل حسب نمط الوصول

- بحث سريع: HashMap
- مجال مرتب: BTreeMap
- Top-K: BinaryHeap
- طرفي صف: VecDeque

القاعدة 5: قلل التخصيص لكل عنصر

لرسوم ضخمة:

- استخدم CSR.
- استخدم مصفوفة مسطحة + إزاحات.

الملحق D: تمارين مصنفة حسب النمط

مؤشرين / نافذة

- Two-sum على مصفوفة مرتبة
- أطول مقطع دون تكرار

Sum Prefix

- مجموع مقطع يساوي K
- 2D prefix sum

بحث ثنائي

- Lower/Upper bound
- بحث في مصفوفة مدوّرة

مكدس رتيب

- Next Greater Element
- Largest Rectangle

برمجة ديناميكية

- Fibonacci
- House Robber
- Knapsack

رسوم بيانية

- BFS
- Dijkstra
- DSU

الملحق E: جدول ملخص الأنماط

النمط	متى يستخدم	التعقيد
Pointers Two	مصفوفة مرتبة / إزالة داخلية	$O(n)$
Sliding Window	أطول/أقصر مقطع بشرط	$O(n)$
Hashing	عدّ التكرارات / عضوية	$O(1)$ متوسط
Search Binary	شرط رتيب	$O(\log n)$
Heap	أفضل اختيار متكرر	$O(\log n)$
DP	بنية مثلى متكررة	غالباً $O(n)$ أو $O(nC)$
BFS/DFS	الوصول والمكونات	$O(V+E)$
Dijkstra	أقصر مسار (أوزان موجبة)	$O((V+E)\log V)$
DSU	استعلامات اتصال كثيرة	قريب من $O(1)$
KMP	بحث جزئي بكفاءة	$O(n+m)$

قاعدة اختيار النمط

- شرط رتيب؟ جرّب البحث الثنائي.
- حدود أقرب عنصر؟ جرّب المكدس الرتيب.
- أفضل اختيار متكرر؟ Heap + Greedy.
- أفضل مجموع تحت قيود؟ Knapsack / DP.
- سؤال وصول؟ ابدأ بـ BFS/DFS.

قائمة اختبار قياسية

```
pub fn test_checklist() {
```

```
    /* empty */
    /* smallest */
    /* duplicates */
    /* worst-case */
    /* random + brute-force cross-check */
}
```

المراجع

المراجع التأسيسية في الخوارزميات

يتضمن هذا القسم المراجع الأكاديمية المعتمدة والواسعة الاستخدام التي تؤسس للأنماط الخوارزمية المستخدمة في هذا الدليل.

- Stein Clifford Rivest, L. Ronald Leiserson, E. Charles Cormen, H. Thomas
  Introduction to Algorithms (MIT Press).
  مرجع قياسي في التحليل التقاربي، الفرز، الأكوام، الجداول المبعثرة، خوارزميات الرسوم البيانية، أقصر المسارات، البرمجة الديناميكية، والتصميم الجشع.

- **Robert Sedgewick, Kevin Wayne**
  Algorithms (Addison-Wesley).
  يركز بقوة على الجانب الهندسي: التنفيذ العملي، الحدس حول الأداء، والأنماط الواقعية.

- Tardos va Kleinberg, Jon
  (Pearson). Design Algorithm
  ممتاز في بناء التفكير التصميمي الصارم: براهين الخوارزميات الجشعة، نمذجة البرمجة الديناميكية، وصحة خوارزميات الرسوم البيانية.

- Skiena S. Steven
  (Springer). Manual Design Algorithm The

جسر عملي بين النظرية والهندسة، مع تركيز على التعرف على الأنماط وربط المسائل بالخوارزميات المناسبة.

- Knuth E. Donald
(Addison-Wesley). Programming Computer of Art The
معالجة عميقة تأسيسية؛ مفيد لفهم الخوارزميات كنظام حسابي منضبط وليس مجرد وصفات برمجية.

- Vazirani Umesh Papadimitriou, Christos Dasgupta, Sanjoy
(McGraw-Hill). Algorithms
شروحات واضحة للنماذج الأساسية: تقسيم وغزو، الجشع، البرمجة الديناميكية، وخوارزميات الرسوم.

- Tamassia Roberto Goodrich, T. Michael
Applications. and Design Algorithm
مرجع قوي في أساسيات هياكل البيانات والتفكير الخوارزمي المنظم.

- مراجع البرمجة التنافسية (تدريب الأنماط)
Laaksonen). (Antti Handbook CP Halim)، & (Halim Programming Competitive
مفيدة كفهرس تدريبي منظم للأنماط، ويفضل إقرانها بالمراجع الأكاديمية السابقة لفهم البراهين والصحة.

توثيق مكتبة Rust القياسية

المراجع التالية تُعد مصادر أساسية وموثوقة لاستخدام Rust المستقر في هندسة الخوارزميات. وهي تنطبق على Windows وLinux وmacOS.

- **The Rust Programming Language (The Book)**
المرجع الأساسي لتعلم Rust: الملكية (Ownership)، الاستعارة (Borrowing)، الأعمار (Lifetimes)، السمات (Traits)، معالجة الأخطاء، ومجموعات البيانات القياسية.

- توثيق مكتبة Rust القياسية (std)
  مرجع رسمي لـ `BinaryHeap`، `BTreeMap`، `HashMap`، `VecDeque`، `Vec`، المقاطع (slices)، السلاسل النصية، وأنماط الفرز والبحث الثنائي.

- **The Rust Reference**
  تفصيل دقيق على مستوى اللغة: القواعد الدلالية للتعابير، الحركات (moves)، الاستعارات، مطابقة الأنماط، وحل السمات.

- **Rust by Example**
  أمثلة قصيرة عملية توضح أنماط الاستخدام الصحيحة مع الملكية والاستعارة.

- **The Cargo Book**
  تنظيم المشاريع، ملفات التعريف (profiles)، الميزات (features)، الاختبارات، وبناء المشاريع بطريقة قابلة لإعادة الإنتاج.

- **The Rustonomicon**
  مرجع للحدود المتقدمة: قواعد `unsafe`، التماثل (aliasing)، تخطيط الذاكرة، والتفاصيل منخفضة المستوى الحساسة للأداء.

- **Rust Style Guide** و **Rust API Guidelines**
  إرشادات التسمية، نماذج الأخطاء، وقابلية الصيانة عند تحويل الخوارزميات إلى مكونات مكتبية قابلة لإعادة الاستخدام.

مراجع هندسة الأنظمة والأداء

تشرح هذه المراجع كيفية تفاعل الخوارزميات مع العتاد الفعلي: الذاكرة المخبئية، التخصيص، التوطين، التنبؤ بالتفرع، ومنهجيات القياس.

- **Randal E. Bryant, David R. O'Hallaron**
  Computer Systems: A Programmer's Perspective.
  أساس عملي لفهم سلوك الذاكرة، المخابئ، ونماذج تكلفة المعالج.

- **Brendan Gregg**
  Systems Performance: Enterprise and the Cloud.
  مرجع منهجي لقياس الأداء، التحليل، وتحويل البيانات إلى استنتاجات صحيحة.

- **Ulrich Drepper**
  What Every Programmer Should Know About Memory.
  تحليل عميق للمخابئ، التوطين، وسبب سيطرة الذاكرة على الأداء في كثير من التطبيقات.

- **Agner Fog**
  أدلة تحسين البرمجيات وملاحظات المعمارية الدقيقة.
  مفيدة لفهم خطوط التنفيذ (pipeline)، الأداء على مستوى التعليمات، وسلوك المترجمات والمعالجات.

- **AMD** و **Intel** معمارية أدلة
  مراجع رسمية حول ترتيب الذاكرة، المخابئ، وعدادات الأداء.

- توثيق أدوات التحليل على **Windows** و **Linux** (منهجي)
  مفاهيم التوقيت، المحللات بالعينة (sampling profilers)، وأدوات التتبع مفاهيم عامة قابلة للتطبيق عبر الأنظمة.

مراجع نظرية الرسوم والبرمجة الديناميكية

نظرية الرسوم وخوارزمياتها

- **Reinhard Diestel**
  Theory. Graph
  مرجع نظري صارم: التعاريف، الاتصال، الدورات، والبنية الهيكلية.

- **Douglas B. West**
  Introduction to Graph Theory.

كتاب تدريسي قوي يربط المفاهيم مباشرة بتفكير BFS/DFS.

- **J. A. Bondy, U. S. R. Murty**
  Graph Theory.
  مرجع واسع للتأسيس النظري العميق.

- **Ravindra K. Ahuja, Thomas L. Magnanti, James B. Orlin**
  Network Flows: Theory, Algorithms, and Applications.
  مرجع ممتاز لأقصر المسارات، الشبكات الجريانية، والهندسة العملية حول الرسوم.

البرمجة الديناميكية والتحسين

- **Cormen et al. (CLRS)**
  فصول قوية في نمذجة الحالات، الانتقالات، وبراهين الصحة.

- **Kleinberg & Tardos**
  البرمجة الديناميكية كنظام تصميم منضبط.

- **Richard Bellman**
  المؤسس النظري لمبدأ المثالية (Principle of Optimality) كأساس للبرمجة الديناميكية.

- نمذجة **Knapsack** وتخصيص الموارد
  كتب الشبكات والتحسين تقدم منظوراً هندسياً واضحاً لمسائل القيود والموارد.

خوارزميات السلاسل النصية

- **Dan Gusfield**
  Algorithms on Strings, Trees, and Sequences.
  مرجع كلاسيكي لمطابقة السلاسل، دالة البادئة، وخوارزميات النصوص المتقدمة.

- **Knuth, Morris, Pratt**
  الصياغة الأصلية لخوارزمية KMP ودالة البادئة، لفهم سبب خطيتها وبنيتها الآلية.

ملاحظة انضباط هندسي

- ارجع إلى توثيق Rust لفهم الدلالات الدقيقة والأداء المتوقع لهياكل البيانات.
- ارجع إلى كتب الخوارزميات لبناء البراهين وتبرير الاختيار الصحيح للنمط.
- ارجع إلى مراجع الأنظمة لضمان أن يبقى تصميمك سريعاً على العتاد الحقيقي، لا فقط في التحليل التقاربي.