1) تهيئة الإدخال/الإخراج السريع (I/O Fast)

الغرض: تقليل زمن القراءة/الكتابة من/إلى الطرفية.

iOS::sync\_with\_stdio(false);

// يفصل iostream عن stdio لزيادة السرعة

cin.tie(nullptr);

// يفصل cin عن cout لمنع التفريغ التلقائي

ملاحظات نقدية سريعة:

* لا تخلط بين scanf/printf وcin/cout بعد تعطيل المزامنة.

2) أنواع البيانات وأحجامها (على منصات 64-بت الشائعة)

الأحجام قد تختلف بين الأنظمة؛ القيم التالية هي الأكثر شيوعًا في بيئات المسابقات (LP64).

|  |  |  |  |
| --- | --- | --- | --- |
| **النوع** | **الحجم التقريبي** | **المدى التقريبي** | **ملاحظات عملية** |
| bool | 1 بايت | {0,1} | استخدمه للرايات فقط |
| char | 1 بايت | -128 … 127 أو 0 … 255 | انتبه للتوقيع |
| int | 4 بايت | −2.1e9 … 2.1e9 | لا يكفي لضرب قيم ~1e9 |
| long long (ll) | 8 بايت | −9e18 … 9e18 | الاختيار الآمن للأعداد |
| unsigned long long | 8 بايت | 0 … 1.8e19 | للأعداد الموجبة فقط |
| float | 4 بايت | ~7 أرقام دقة | تجنّبه للحسابات الدقيقة |
| double | 8 بايت | ~15 رقم دقة | الافتراضي للكسور |
| long double (ld) | 16/8 بايت | أعلى من double (حسب المترجم) | استخدمه عند الحاجة |
| string | — | نص ديناميكي | احذر من نسخ السلاسل الكبير |
| vector<T> | — | مصفوفة ديناميكية | نموّ مضاعف؛ O(1) إلحاقًا |

تحويلات مفيدة وآمنة:

long long a = 1e12;

// استخدم ll للأعداد الكبيرة

\_\_int128 big = (\_\_int128)a\*a;

// حساب وسيط آمن جداً

long long x = (long long)big;

// التحويل قد يفيض إن تعدّى المدى

3) الثوابت والحدود (Limits & Constants)

مصادر قياسية:

#include <climits> // INT\_MAX, LLONG\_MAX, ...

#include <cfloat> // DBL\_MAX, FLT\_EPSILON, ...

#include <limits> // numeric\_limits<T>

#include <cmath> // acos, sqrt, ...

أمثلة:

int imax = INT\_MAX; // 2147483647

long long llmax = LLONG\_MAX; // 9223372036854775807

double dmax = numeric\_limits<double>::max();

double eps = numeric\_limits<double>::epsilon();

// أصغر فرق مُمَيَّز

// ثابت π بطريقة مضمونة:

const long double PI = acosl(-1.0L);

// أو:

constexpr long double PIc = 3.141592653589793238L;

نصائح سريعة:

* لا تعتمد على M\_PI؛ غير مضمونة إلا مع تعريفات معينة.

4) Typedefs & Macros (اختصارات عملية وآمنة)

using / typedef (مُستحسن)

تجعل الشيفرة أقصر وأكثر وضوحًا:

using ll = long long;

using ull = unsigned long long;

using ld = long double;

using pii = pair<int,int>;

template<class T> using min\_heap = priority\_queue<T, vector<T>, greater<T>>;

ماكروز مفيدة (خليها قليلة وواضحة)

#define all(x) begin(x), end(x) // تمرير الحاويات إلى STL

#define rall(x) rbegin(x), rend(x)

#define sz(x) (int)(x).size()

// Debug سريع (يختفي في التحكيم)

#ifdef LOCAL

#define dbg(x) cerr << "[" << #x << "] = " << (x) << '\n'

#else

#define dbg(x) ((void)0)

#endif

تحذير نقدي:

* تجنّب الماكروز المعقّدة (توابع متعددة الأسطر، أسماء عامة) لأنها تصعّب التتبع وقت الضغط.
* فضّل inline وconstexpr وusing على الماكروز متى أمكن.

5) قِطع كود جاهزة قصيرة (من الأساسيات فقط)

جمع/طرح/ضرب مع MOD (آمن وسريع)

constexpr int MOD = 1'000'000'007;

inline long long addmod(long long a, long long b, long long m=MOD){

a%=m; b%=m; long long c=a+b; if(c>=m) c-=m; return c;

}

inline long long submod(long long a, long long b, long long m=MOD){

a%=m; b%=m; long long c=a-b; if(c<0) c+=m; return c;

}

inline long long mulmod(long long a, long long b, long long m=MOD){

return ( (\_\_int128)a \* b ) % m; // يمنع الفيض الوسيط

}

inline long long binpow(long long a, long long e, long long m=MOD){

a%=m; long long r=1%m;

while(e){ if(e&1) r=mulmod(r,a,m); a=mulmod(a,a,m); e>>=1; }

return r;

}

inline long long invmod(long long a, long long m=MOD){ // عندما m أولي

return binpow(a, m-2, m);

}

ما هو الـ mod في البرمجة التنافسية؟

الـ mod اختصار لـ modulus أو الموديلار، وهو ببساطة الباقي من القسمة.

📌 الفكرة الأساسية

عندما نقول:

x % m

فهذا يعني: قسمة x على m وأخذ الباقي فقط.

• 6 = 27%7.

📌 لماذا نستخدمه في المسابقات؟

1. منع الأعداد من أن تكبر جدًا (تجنّبالـ *overflow*):  
   عند الحسابات الكبيرة (مثل ضرب أعداد كثيرة)، الأرقام قد تصل لمليارات المليارات بسرعة، لكن أخذ الباقي يبقيها صغيرة.
2. مطابقة شرط المسألة:  
   كثير من مسائل العدّ تطلب الناتج بصيغة "modulo 1,000,000,007".
3. الحفاظ على خصائص رياضية مفيدة:

mm)) mod b mod) + (m a mod)) = m b) mod + a) •

m m)) mod b mod) x (m a mod)) = m b) mod x a) •

لماذا الرقم 1,000,000,007 بالذات؟

* هو عدد أولي (*prime number*).
* قريب من 10^9
* مناسب لحدود الزمن والذاكرة.
* كونه أوليًا يجعل بعض العمليات (مثل المعكوس الضربي – Modular Inverse) أسهل باستخدام نظرية فيرما الصغرى.

📌 في الكود:

constexpr int MOD = 1’000'000'007;

هذا يحدد قيمة الموديلار الافتراضية لتكون 1,000,000,007

• كل الدوال مثل addmod, submod, mulmod تعمل على الأعداد داخل هذا الموديلار بحيث يكون الناتج دائمًا بين 0 و 1 -MOD.

حدود الأنواع (طباعة سريعة للمرجع)

cout << INT\_MIN << " .. " << INT\_MAX << '\n';

cout << LLONG\_MIN << " .. " << LLONG\_MAX << '\n';

cout << numeric\_limits<double>::min() << " .. "

<< numeric\_limits<double>::max() << '\n';

|  |  |  |
| --- | --- | --- |
| **الثابت** | **القيمة التقريبية** | **الوصف والاستخدام** |
| INT\_MAX | 2,147,483,647 | أكبر قيمة يمكن أن يخزنها النوع int (32-بت). |
| INT\_MIN | -2,147,483,648 | أصغر قيمة يمكن أن يخزنها النوع int (32-بت). |
| LLONG\_MAX | 9,223,372,036,854,775,807 | أكبر قيمة يمكن أن يخزنها النوع long long (64-بت). |
| LLONG\_MIN | -9,223,372,036,854,775,808 | أصغر قيمة يمكن أن يخزنها النوع long long (64-بت). |
| DBL\_MAX | ≈ 1.7976931348623157 × 10^308 | أكبر قيمة يمكن أن يخزنها النوع double. |
| DBL\_MIN | ≈ 2.2250738585072014 × 10^-308 | أصغر قيمة موجبة يمكن أن يخزنها النوع double. |
| FLT\_EPSILON | ≈ 1.19209 × 10^-7 | أصغر فرق يمكن تمييزه بين عددين من نوع float. |
| DBL\_EPSILON | ≈ 2.22045 × 10^-16 | أصغر فرق يمكن تمييزه بين عددين من نوع double. |
| M\_PI *(*إنوجد*)* | 3.141592653589793 | قيمة π (باي) — غير مضمونة دائمًا، الأفضل تعريفها يدويًا. |
| acosl(-1.0L) *(*بديلآمنلـ *M\_PI)* | 3.14159265358979323846264338327950288... | طريقة مضمونة لحساب π باستخدام دالة acos. |

📦 الحاويات المتسلسلة (Sequence Containers)

|  |  |
| --- | --- |
| **الحاوية** | **متى تستخدمها في المسابقات** |
| **vector** | الحاوية الافتراضية لأغلب المسائل: مصفوفة ديناميكية، وصول عشوائي سريع O(1)، إدراج في النهاية سريع. مثالية لتخزين القوائم، الأوزان، الإحداثيات، نتائج الفرز، والبحث الثنائي. |
| **deque** | عند الحاجة إلى إضافة/حذف من البدايةوالنهاية بكفاءة O(1) في الحالتين. مثالية لمسائل **Sliding Window** أو المحاكاة التي تتحرك من طرفين. |
| **array** | مصفوفة ثابتة الحجم (تحديد الحجم وقت الترجمة)، أسرع من vector قليلًا. مناسبة عندما تعرف الحجم مسبقًا ولا تحتاج تغيير الحجم. |
| **string** | للتعامل مع النصوص: بحث، تعديل، تقسيم، مقارنة. مفيدة في مسائل النصوص، **pattern matching**، تحليل المدخلات النصية. |

📚 الحاويات المرتبطة (Associative Containers)

|  |  |
| --- | --- |
| **الحاوية** | **متى تستخدمها في المسابقات** |
| **set** | مجموعة مرتبة بدون تكرار، بحث/إدراج/حذف في O(log n). مثالية لمسائل تتطلب الحفاظ على مجموعة عناصر فريدة مرتبة، أو إيجاد أقربعنصرأكبر**/**أصغر باستخدام lower\_bound. |
| **multiset** | مثل set لكن يسمح بالتكرار، ممتازة لمسائل الحفاظ على بيانات مرتبة مع التكرارات، مثل تتبع القيم في نافذة منزلقة بوجود تكرار. |
| **map** | تخزين أزواج (مفتاح،قيمة) مرتبة حسب المفتاح، وصول في O(log n). مثالية للجداول التي تحتاج ترتيب المفاتيح مع البحث أو التنقل بينها. |
| **multimap** | مثل map لكن يسمح بمفاتيح متكررة، مفيد لمسائل تخزين عدة قيَم لنفس المفتاح مع ترتيبها. |

⚡ الحاويات غير المرتبة (Unordered Containers)

|  |  |
| --- | --- |
| **الحاوية** | **متى تستخدمها في المسابقات** |
| **unordered\_set** | مثل set لكن غير مرتبة، وصول/إدراج/حذف في متوسط **O(1)**. ممتازة لمسائل التحقق السريع من الوجود، أو إزالة العناصر المكررة بدون اهتمام بالترتيب. |
| **unordered\_multiset** | مثل multiset لكن غير مرتبة، مع التكرار. مفيد لمسائل العدّ السريع للقيم المتكررة. |
| **unordered\_map** | مثل map لكن غير مرتبة، وصول/إدراج في متوسط **O(1)**. ممتاز لمسائل العدّ (**frequency map**)، أو ربط القيم بالمفاتيح بدون الحاجة إلى ترتيب. |
| **unordered\_multimap** | مثل unordered\_map لكن يسمح بتكرار المفاتيح، مفيد لمسائل تحتاج تخزين عدة قيَم لنفس المفتاح بسرعة بدون ترتيب. |

🛠 الحاويات المهيكلة (Container Adapters)

|  |  |
| --- | --- |
| **الحاوية** | **متى تستخدمها في المسابقات** |
| **stack** | مكدس (LIFO) مثالي لمسائل الأقواس المتوازنة، التراجع (Undo)، وحل بعض مشاكل **DFS** بدون استدعاء عودي. |
| **queue** | طابور (FIFO) مثالي لمسائل **BFS**، والمحاكاة التي تعتمد على ترتيب وصول العناصر. |
| **priority\_queue** *(Max-Heap)* | طابور أولوية لإخراج أكبر عنصر بسرعة. مثالي لخوارزميات مثل **Dijkstra** أو اختيار أعلى/أكبر قيمة باستمرار. لتحويله لـ **Min-Heap**: |

priority\_queue<int, vector<int>, greater<int>> pq;

``` |

---

💡 \*\*قاعدة ذهبية للمسابقات:\*\*

ابدأ دائمًا بـ `vector` أو `unordered\_map` إلا إذا المسألة \*\*تتطلب ترتيب العناصر ديناميكيًا\*\* أو \*\*تحتاج إدخال/إزالة من طرفين\*\*، أو \*\*تحتاج خصائص معينة\*\* مثل أقرب عنصر أو تكرار مرتب.

---

لو تحب، أقدر أعمل لك \*\*خريطة قرار (Decision Tree)\*\* رسومية، بحيث تنظر إلى طبيعة المسألة وتعرف فورًا أي حاوية تختار.

هل تريدني أعملها لك؟

STL Containers

Vector

|  |  |
| --- | --- |
| **الدالة / العملية** | **الفائدة** |
| vector() | إنشاء vector فارغ. |
| vector(size\_t n) | إنشاء vector بحجم ثابت مبدئيًا. |
| vector(size\_t n, const T& val) | إنشاء vector بحجم ثابت وقيم مبدئية. |
| operator= | نسخ أو نقل محتويات vector إلى آخر. |
| assign(count, value) | ملء الحاوية بعدد معين من العناصر بنفس القيمة. |
| assign(first, last) | ملء الحاوية بنطاق عناصر من حاوية أخرى. |
| at(pos) | الوصول إلى عنصر مع فحص الحدود (يرمي استثناء إن كان خارج النطاق). |
| operator[](pos) | الوصول إلى عنصر بدون فحص الحدود (أسرع). |
| front() | الوصول لأول عنصر. |
| back() | الوصول لآخر عنصر. |
| data() | الحصول على مؤشر (Pointer) لأول عنصر في الذاكرة. |
| empty() | التحقق إن كانت الحاوية فارغة. |
| size() | إرجاع عدد العناصر الحالي. |
| capacity() | إرجاع عدد العناصر التي يمكن تخزينها قبل إعادة تخصيص الذاكرة. |
| max\_size() | أكبر عدد ممكن للعناصر حسب النظام. |
| reserve(n) | حجز سعة تخزين لعناصر مستقبلية لتجنب إعادة التخصيص المتكرر. |
| shrink\_to\_fit() | تقليص السعة لتساوي الحجم الفعلي. |
| clear() | حذف جميع العناصر مع بقاء السعة. |
| insert(pos, value) | إدراج عنصر قبل الموضع المحدد. |
| insert(pos, count, value) | إدراج عدد من العناصر في موقع محدد. |
| insert(pos, first, last) | إدراج عناصر من نطاق في موقع محدد. |
| emplace(pos, args...) | إنشاء عنصر في الموقع المحدد مباشرة بدون نسخ. |
| erase(pos) | حذف عنصر عند موقع معين. |
| erase(first, last) | حذف نطاق من العناصر. |
| push\_back(value) | إضافة عنصر في النهاية. |
| emplace\_back(args...) | إنشاء عنصر في النهاية مباشرة بدون نسخ. |
| pop\_back() | حذف آخر عنصر. |
| resize(count) | تغيير الحجم (قد يزيد أو ينقص العناصر). |
| resize(count, value) | تغيير الحجم وتعبئة القيم الجديدة بقيمة محددة. |
| swap(other) | تبادل المحتويات مع vector آخر. |
| begin() | إرجاع مكرر (Iterator) لأول عنصر. |
| end() | إرجاع مكرر لعنصر بعد الأخير (نهاية النطاق). |
| rbegin() | مكرر معكوس لأول عنصر معكوس. |
| rend() | مكرر معكوس لعنصر بعد الأخير في الاتجاه المعكوس. |
| cbegin() | مكرر ثابت لأول عنصر (قراءة فقط). |
| cend() | مكرر ثابت بعد الأخير (قراءة فقط). |
| crbegin() | مكرر ثابت معكوس. |
| crend() | مكرر ثابت معكوس لنهاية النطاق. |

Strings

|  |  |
| --- | --- |
| **الدالة / العملية** | **الفائدة** |
| جميع بُناة vector + مميزات النصوص | نفس خصائص vector<char> لكن مع دعم النصوص. |
| c\_str() | إرجاع نص بأسلوب C (منتهي بـ \0). |
| append(str) | إضافة نص في النهاية. |
| operator+= | إلحاق نص أو محرف في النهاية. |
| substr(pos, count) | إرجاع جزء من النص يبدأ من موضع محدد بطول معين. |
| find(str) | إيجاد أول ظهور للنص الفرعي. |
| rfind(str) | إيجاد آخر ظهور للنص الفرعي. |
| find\_first\_of(chars) | إيجاد أول ظهور لأي محرف من مجموعة. |
| find\_last\_of(chars) | إيجاد آخر ظهور لأي محرف من مجموعة. |
| find\_first\_not\_of(chars) | إيجاد أول محرف غير موجود في المجموعة. |
| find\_last\_not\_of(chars) | إيجاد آخر محرف غير موجود في المجموعة. |
| replace(pos, count, str) | استبدال جزء معين من النص. |
| insert(pos, str) | إدراج نص عند موقع محدد. |
| erase(pos, count) | حذف جزء من النص. |
| compare(str) | مقارنة النصوص (ترجع 0 إذا كانت متساوية). |

Stack

|  |  |
| --- | --- |
| **الدالة / العملية** | **الفائدة** |
| stack() | إنشاء مكدس فارغ. |
| stack(const stack &other) | إنشاء نسخة من مكدس آخر. |
| stack(stack &&other) | إنشاء مكدس بالنقل من مكدس آخر. |
| operator=(const stack &other) | نسخ محتويات مكدس آخر. |
| operator=(stack &&other) | نقل محتويات مكدس آخر. |
| empty() | التحقق إن كان المكدس فارغًا (يُرجع true إذا لا يحتوي على عناصر). |
| size() | إرجاع عدد العناصر في المكدس. |
| top() | الوصول إلى العنصر الموجود في أعلى المكدس (قراءة/تعديل). |
| push(const T &value) | إضافة عنصر جديد في أعلى المكدس (نسخ القيمة). |
| push(T &&value) | إضافة عنصر جديد في أعلى المكدس (نقل القيمة). |
| emplace(args...) | إنشاء عنصر مباشرة في أعلى المكدس بدون نسخ أو نقل. |
| pop() | إزالة العنصر الموجود في أعلى المكدس (بدون إرجاعه). |
| swap(stack &other) | تبادل محتويات المكدس مع مكدس آخر بنفس النوع. |

Queue

|  |  |
| --- | --- |
| **الدالة / العملية** | **الفائدة** |
| queue() | إنشاء طابور فارغ. |
| queue(const queue &other) | إنشاء نسخة من طابور آخر. |
| queue(queue &&other) | إنشاء طابور بالنقل من طابور آخر. |
| operator=(const queue &other) | نسخ محتويات طابور آخر. |
| operator=(queue &&other) | نقل محتويات طابور آخر. |
| empty() | التحقق إن كان الطابور فارغًا. |
| size() | إرجاع عدد العناصر في الطابور. |
| front() | الوصول لأول عنصر (قراءة/تعديل). |
| back() | الوصول لآخر عنصر (قراءة/تعديل). |
| push(const T &value) | إضافة عنصر في نهاية الطابور (نسخ القيمة). |
| push(T &&value) | إضافة عنصر في نهاية الطابور (نقل القيمة). |
| emplace(args...) | إنشاء عنصر مباشرة في نهاية الطابور بدون نسخ أو نقل. |
| pop() | إزالة العنصر الموجود في بداية الطابور. |
| swap(queue &other) | تبادل محتويات الطابور مع طابور آخر. |

Duque

|  |  |
| --- | --- |
| **الدالة / العملية** | **الفائدة** |
| deque() | إنشاء صف مزدوج الأطراف فارغ. |
| deque(size\_t n) | إنشاء بحجم ثابت مبدئيًا. |
| deque(size\_t n, const T &val) | إنشاء بحجم ثابت وقيم مبدئية. |
| operator= | نسخ أو نقل محتويات. |
| assign(count, value) | ملء الحاوية بعدد معين من العناصر. |
| assign(first, last) | ملء الحاوية من نطاق. |
| at(pos) | الوصول لعنصر مع فحص الحدود. |
| operator[](pos) | الوصول لعنصر بدون فحص الحدود. |
| front() | الوصول لأول عنصر. |
| back() | الوصول لآخر عنصر. |
| empty() | التحقق إن كانت فارغة. |
| size() | عدد العناصر الحالي. |
| max\_size() | أقصى حجم ممكن. |
| shrink\_to\_fit() | تقليص السعة للحجم الفعلي. |
| clear() | مسح جميع العناصر. |
| insert(pos, value) | إدراج عنصر في موقع محدد. |
| insert(pos, count, value) | إدراج عدد من العناصر. |
| insert(pos, first, last) | إدراج عناصر من نطاق. |
| emplace(pos, args...) | إنشاء عنصر في موقع محدد بدون نسخ. |
| erase(pos) | حذف عنصر بموقع محدد. |
| erase(first, last) | حذف نطاق عناصر. |
| push\_back(value) | إضافة عنصر في النهاية. |
| emplace\_back(args...) | إنشاء عنصر في النهاية بدون نسخ. |
| push\_front(value) | إضافة عنصر في البداية. |
| emplace\_front(args...) | إنشاء عنصر في البداية بدون نسخ. |
| pop\_back() | إزالة آخر عنصر. |
| pop\_front() | إزالة أول عنصر. |
| resize(count) | تغيير حجم الحاوية. |
| resize(count, value) | تغيير الحجم مع تعبئة القيم الجديدة. |
| swap(other) | تبادل المحتويات مع deque آخر. |
| begin() / end() | مكررات بداية/نهاية. |
| rbegin() / rend() | مكررات معكوسة. |
| cbegin() / cend() | مكررات ثابتة. |
| crbegin() / crend() | مكررات ثابتة معكوسة. |

Pair

|  |  |
| --- | --- |
| **الدالة / العملية** | **الفائدة** |
| pair() | إنشاء زوج افتراضيًا (القيم مهيّأة افتراضيًا). |
| pair(const T1& a, const T2& b) | إنشاء زوج بالقيمتين مباشرة. |
| pair(const pair& other) | نسخ زوج آخر. |
| pair(pair&& other) | نقل زوج آخر بكفاءة. |
| operator=(const pair& other) | إسناد بالنسخ. |
| operator=(pair&& other) | إسناد بالنقل. |
| first | العضو الأول من النوع T1. |
| second | العضو الثاني من النوع T2. |
| swap(pair& other) | تبادل زوجين بكفاءة. |
| operator==, operator!= | مقارنة مساواة/عدم مساواة عنصرًا بعنصر. |
| operator<, operator>, operator<=, operator>= | مقارنة معجمية (تبدأ بـ first ثم second). |
| operator<=> *(C++20)* | مقارنة مركّبة (ثلاثية) إن توفرت. |
| std::make\_pair(a, b) | إنشاء زوج باستنتاج الأنواع تلقائيًا. |
| std::get<0>(p) | وصول للعنصر الأول بأسلوب get. |
| std::get<1>(p) | وصول للعنصر الثاني بأسلوب get. |
| std::tuple\_size<pair<...>>::value | حجم الزوج (ثابتًا 2 — لآليات عامّة). |
| std::tuple\_element<I, pair<...>>::type | نوع العنصر عند الفهرس I. |

Tuble

|  |  |
| --- | --- |
| **الدالة / العملية** | **الفائدة** |
| tuple() | إنشاء صفّ (tuple) فارغ/مهيّأ افتراضيًا لكل عنصر. |
| tuple(const Ts&...) | إنشاء بقيم محددة لكل عنصر. |
| tuple(const tuple& other) | نسخ صفّ آخر. |
| tuple(tuple&& other) | نقل صفّ آخر بكفاءة. |
| operator=(const tuple& other) | إسناد بالنسخ. |
| operator=(tuple&& other) | إسناد بالنقل. |
| swap(tuple& other) | تبادل محتويات صفّين بكفاءة. |
| std::get<I>(t) | وصول للعنصر عند الفهرس I (وقت الترجمة). |
| std::get<T>(t) | وصول حسب النوع T إذا كان فريدًا داخل الصفّ. |
| std::tie(a, b, ...) | ربط مراجع متغيّرات بعناصر الصفّ لتفكيكه. |
| std::make\_tuple(args...) | إنشاء صفّ مع استنتاج الأنواع تلقائيًا. |
| std::forward\_as\_tuple(args&&...) | إنشاء صفّ من مراجع مع توجيه مثالي (للاستدعاءات القالبية). |
| std::tuple\_cat(t1, t2, ...) | دمج عدة صفوف في صفّ واحد. |
| std::apply(f, t) | استدعاء الدالة f وتمرير عناصر t كوسائط تلقائيًا. |
| operator==, operator!= | مقارنة مساواة/عدم مساواة عنصرًا بعنصر لكل موضع. |
| operator<, operator>, operator<=, operator>= | مقارنة معجمية لكل العناصر بالترتيب. |
| operator<=> *(C++20)* | مقارنة مركّبة إذا دعمت الأنواع الداخلية ذلك. |
| std::tuple\_size<tuple<...>>::value | عدد العناصر داخل الصفّ (قيمة ثابتة في وقت الترجمة). |
| std::tuple\_element<I, tuple<...>>::type | نوع العنصر عند الفهرس I. |

Priority Queues

|  |  |
| --- | --- |
| **الدالة / العملية** | **الفائدة** |
| priority\_queue() | إنشاء طابور أولوية فارغ (Max-Heap افتراضيًا). |
| priority\_queue(comp) | إنشاء بطابور أولوية مع مقارنة مخصصة. |
| priority\_queue(first, last) | إنشاء من نطاق عناصر. |
| empty() | التحقق إن كان فارغًا. |
| size() | عدد العناصر. |
| top() | الوصول لأعلى عنصر حسب المقارنة (الأكبر في Max-Heap). |
| push(const T &value) | إدراج عنصر (نسخ القيمة). |
| push(T &&value) | إدراج عنصر (نقل القيمة). |
| emplace(args...) | إنشاء عنصر مباشرة بدون نسخ. |
| pop() | إزالة أعلى عنصر. |
| swap(other) | تبادل المحتويات مع طابور أولوية آخر. |

SET

|  |  |
| --- | --- |
| **الدالة / العملية** | **الفائدة** |
| set() | إنشاء مجموعة فارغة. |
| operator= | نسخ أو نقل محتويات المجموعة. |
| begin() / end() | مكررات للاجتياز بالترتيب التصاعدي. |
| rbegin() / rend() | مكررات للاجتياز بالترتيب التنازلي. |
| empty() | التحقق إن كانت المجموعة فارغة. |
| size() | عدد العناصر في المجموعة. |
| max\_size() | أقصى عدد يمكن تخزينه. |
| insert(value) | إدراج عنصر (يتجاهل إذا كان موجودًا). |
| emplace(args...) | إنشاء عنصر مباشرة داخل المجموعة. |
| emplace\_hint(pos, args...) | إنشاء عنصر مع تلميح موضع للإدراج. |
| erase(value) | حذف عنصر بالقيمة. |
| erase(pos) | حذف عنصر بالمكرر. |
| erase(first, last) | حذف نطاق عناصر. |
| swap(other) | تبادل محتويات مع مجموعة أخرى. |
| clear() | مسح جميع العناصر. |
| find(value) | إيجاد عنصر وإرجاع مكرر له. |
| count(value) | إرجاع 1 إذا موجود، 0 إذا غير موجود. |
| contains(value) *(C++20)* | التحقق من وجود عنصر مباشرة. |
| lower\_bound(value) | إرجاع أول عنصر أكبر أو يساوي القيمة. |
| upper\_bound(value) | إرجاع أول عنصر أكبر من القيمة. |
| equal\_range(value) | إرجاع زوج مكررات يحدد النطاق الذي يساوي القيمة. |

Multi set

|  |  |
| --- | --- |
| **الدالة / العملية** | **الفائدة** |
| multiset() | إنشاء مجموعة متعددة القيم فارغة. |
| operator= | نسخ أو نقل محتويات. |
| begin() / end() | مكررات للاجتياز بالترتيب. |
| rbegin() / rend() | مكررات للاجتياز العكسي. |
| empty() | التحقق إن كانت فارغة. |
| size() | عدد العناصر. |
| max\_size() | أقصى حجم ممكن. |
| insert(value) | إدراج عنصر (يسمح بالتكرار). |
| emplace(args...) | إنشاء عنصر مباشرة داخل المجموعة. |
| emplace\_hint(pos, args...) | إنشاء عنصر مع تلميح موضع للإدراج. |
| erase(value) | حذف كل النسخ من القيمة. |
| erase(pos) | حذف عنصر بمكرر. |
| erase(first, last) | حذف نطاق عناصر. |
| swap(other) | تبادل المحتويات مع multiset آخر. |
| clear() | مسح جميع العناصر. |
| find(value) | إيجاد عنصر وإرجاع مكرر له. |
| count(value) | عدد النسخ الموجودة للقيمة. |
| lower\_bound(value) | أول عنصر ≥ القيمة. |
| upper\_bound(value) | أول عنصر > القيمة. |
| equal\_range(value) | إرجاع نطاق لكل العناصر التي تساوي القيمة. |

Map

|  |  |
| --- | --- |
| **الدالة / العملية** | **الفائدة** |
| map() | إنشاء خريطة فارغة. |
| operator= | نسخ أو نقل محتويات الخريطة. |
| begin() / end() | مكررات للاجتياز بالترتيب التصاعدي حسب المفتاح. |
| rbegin() / rend() | مكررات للاجتياز بالترتيب التنازلي. |
| empty() | التحقق إن كانت الخريطة فارغة. |
| size() | عدد العناصر في الخريطة. |
| max\_size() | أقصى عدد ممكن من العناصر. |
| operator[](key) | الوصول أو إنشاء قيمة مرتبطة بالمفتاح. |
| at(key) | الوصول للقيمة مع فحص وجود المفتاح. |
| insert({key, value}) | إدراج عنصر (يتجاهل إذا المفتاح موجود). |
| emplace(key, value) | إنشاء عنصر مباشرة داخل الخريطة. |
| emplace\_hint(pos, key, value) | إنشاء عنصر مع تلميح موضع للإدراج. |
| erase(key) | حذف عنصر حسب المفتاح. |
| erase(pos) | حذف عنصر بالمكرر. |
| erase(first, last) | حذف نطاق عناصر. |
| swap(other) | تبادل المحتويات مع خريطة أخرى. |
| clear() | مسح جميع العناصر. |
| find(key) | إيجاد عنصر وإرجاع مكرر له. |
| count(key) | 1 إذا المفتاح موجود، 0 إذا غير موجود. |
| contains(key) *(C++20)* | التحقق من وجود المفتاح مباشرة. |
| lower\_bound(key) | أول عنصر مفتاحه ≥ المفتاح المعطى. |
| upper\_bound(key) | أول عنصر مفتاحه > المفتاح المعطى. |
| equal\_range(key) | زوج مكررات يحدد النطاق الذي يحمل نفس المفتاح. |

Unordered Map

|  |  |
| --- | --- |
| **الدالة / العملية** | **الفائدة** |
| unordered\_map() | إنشاء خريطة تجزيئية فارغة. |
| unordered\_map(size\_type n) | إنشاء مع عدد دلاء ابتدائي لتحسين الأداء. |
| unordered\_map(first, last) | إنشاء من نطاق مفاتيح/قيم. |
| unordered\_map(const unordered\_map&) | نسخ المحتويات. |
| unordered\_map(unordered\_map&&) | نقل المحتويات دون نسخ. |
| operator=(const unordered\_map&) | إسناد بالنسخ. |
| operator=(unordered\_map&&) | إسناد بالنقل. |
| at(const Key& k) | وصول آمن للقيمة حسب المفتاح (يرمي استثناء إذا غير موجود). |
| operator[](const Key& k) | وصول/إنشاء عنصر للمفتاح إن لم يوجد. |
| begin() / end() | مكررات على العناصر (ترتيب غير مضمون). |
| cbegin() / cend() | مكررات ثابتة للقراءة فقط. |
| empty() | هل الخريطة فارغة؟ |
| size() | عدد العناصر الحالي. |
| max\_size() | أقصى عدد عناصر يمكن تخزينه. |
| clear() | مسح جميع العناصر. |
| insert(value\_type v) | إدراج زوج (مفتاح،قيمة) إن لم يكن المفتاح موجودًا. |
| insert(hint, value\_type v) | إدراج مع تلميح — قد يسرّع. |
| insert(first, last) | إدراج من نطاق. |
| insert\_or\_assign(key, obj) *(C++17)* | إدراج جديد أو إسناد إذا المفتاح موجود. |
| try\_emplace(key, args...) *(C++17)* | إنشاء القيمة في المكان فقط إذا لم يوجد المفتاح (بدون نسخ). |
| emplace(args...) | إنشاء عنصر مباشرة في الحاوية (بدون تهيئة مؤقتة). |
| erase(const Key& k) | حذف كل العناصر ذات المفتاح k (0 أو 1 عادة). |
| erase(iterator pos) | حذف عنصر بمكرّر. |
| erase(first, last) | حذف نطاق عناصر. |
| swap(unordered\_map& other) | تبادل المحتويات بكفاءة. |
| extract(const Key& k) | نزع عقدة مفتاح لإعادة إدراجها أو تعديلها خارج الحاوية. |
| extract(iterator pos) | نزع عقدة عند مكرّر محدد. |
| merge(unordered\_map& source) | دمج عناصر من خريطة أخرى دون نسخ غير الضروري. |
| find(const Key& k) | إيجاد عنصر بالمفتاح (يعيد end() إذا غير موجود). |
| count(const Key& k) | 0 أو 1 (وجود/عدم وجود المفتاح). |
| contains(const Key& k) *(C++20)* | فحص وجود المفتاح مباشرة. |
| equal\_range(const Key& k) | زوج مكررات لنطاق العناصر بالمفتاح (عادة عنصر واحد أو فارغ). |
| bucket\_count() | عدد الدلاء الحالي (لأغراض الضبط). |
| max\_bucket\_count() | أقصى عدد دلاء ممكن. |
| bucket\_size(n) | عدد العناصر داخل الدلو رقم n. |
| bucket(const Key& k) | رقم الدلو الذي سيحتوي k. |
| load\_factor() | نسبة التحميل الحالية = size / bucket\_count. |
| max\_load\_factor() | قراءة الحد الأقصى لنسبة التحميل. |
| max\_load\_factor(float z) | تعيين الحد الأقصى لنسبة التحميل (يضبط إعادة التجزئة). |
| rehash(size\_type n) | تغيير عدد الدلاء لإعادة توزيع العناصر. |
| reserve(size\_type n) | حجز سعة لإيواء n عنصرًا لتقليل إعادة التجزئة. |

الخوارزميات المهمة في حل المشاكل:

|  |  |
| --- | --- |
| **الخوارزمية** | **نوع المشاكل المناسب** |
| **البحث الخطي (Linear Search)** | عند صِغر حجم البيانات أو عند عدم وجود ترتيب في العناصر. |
| **البحث الثنائي (Binary Search)** | إيجاد عنصر أو جواب في مجال مرتب، أو البحث على الجواب في المسائل التي يمكن صياغتها على شكل "تحقق/لا". |
| **فرز (Sort)** | ترتيب العناصر لحل مشاكل المقارنة، إزالة التكرار، أو تحضير البيانات للبحث الثنائي. |
| **فرز عدّي (Counting Sort)** | فرز عند صِغر مجال القيم (0..K)، مفيد لمسائل التكرارات الكبيرة. |
| **BFS (Breadth-First Search)** | إيجاد أقصر مسار في رسم بياني غير موجه أو بأوزان = 1، أو تقسيم الرسم البياني إلى طبقات. |
| **DFS (Depth-First Search)** | اكتشاف الاتصال في الرسوم البيانية، التحقق من وجود دورة، الترتيب الطوبولوجي (مع DAG)، ومسائل Flood Fill. |
| **Dijkstra** | إيجاد أقصر مسار في رسوم بيانية بأوزان موجبة فقط. |
| **Bellman–Ford** | إيجاد أقصر مسار في رسوم بيانية تحتوي أوزان سالبة، وكشف الدورات السالبة. |
| **Floyd–Warshall** | إيجاد أقصر مسار بين كل زوج من العقد عندما يكون عدد العقد صغير (~≤500). |
| **Union–Find (DSU)** | تحديد اتصال المكونات، التحقق من اتحاد المجموعات، خوارزمية Kruskal لبناء MST. |
| **البرمجة الديناميكية (Dynamic Programming - DP)** | مسائل الأمثلية أو عدّ الطرق التي يمكن تقسيمها إلى حالات فرعية مع تداخل (Overlapping Subproblems). |
| **Segment Tree** | استعلامات مجال (مثل المجموع/القيمة العظمى/الصغرى) مع تحديثات متكررة على العناصر. |
| **Fenwick Tree (BIT)** | مجموع بادئة أو استعلام مجال بسيط مع تحديث نقطة، بكفاءة log n. |
| **KMP** | البحث عن نمط في نص بكفاءة O(n+m) دون إعادة فحص الحروف. |
| **Rabin–Karp** | البحث عن عدة أنماط في نص أو تسريع البحث الأولي مع التحقق لاحقًا. |
| **Merge Sort** | فرز مستقر O(n log n) مناسب للبيانات التي تحتاج الحفاظ على ترتيب العناصر المتساوية. |
| **Quick Sort** | فرز سريع عمليًا O(n log n) مع متوسط أداء ممتاز. |
| **Counting / Bucket / Radix Sort** | مسائل الفرز عندما القيم عددية ونطاقها صغير أو متوسط. |
| **Two Pointers** | إيجاد أزواج/مجاميع/شرائح تحقق شرطًا في بيانات مرتبة، أو ضغط النوافذ المنزلقة. |
| **Sliding Window** | إيجاد أفضل/أقصى/أدنى قيمة في نافذة متحركة على مصفوفة أو سلسلة. |
| **Topological Sort** | ترتيب العقد في رسم بياني DAG بحيث تأتي جميع الحواف من العقد السابقة إلى اللاحقة (جدولة المهام). |

1) البحث الخطي والبحث الثنائي

متى أستخدمه؟

* البحث الخطي: مدخلات صغيرة أو بنية غير قابلة للفرز سريعًا.
* البحث الثنائي: للبحث عن قيمة/جواب في مجالمرتب أو على الجوابنفسه مع دالة تحقّق ok(mid).

// بحث خطي

int linear\_search(const vector<int>& a, int x){

for(int i=0;i<(int)a.size();++i) if(a[i]==x) return i;

return -1;

}

// بحث ثنائي على مصفوفة مرتبة

int binary\_search\_idx(const vector<int>& a, int x){

int l=0, r=(int)a.size()-1, ans=-1;

while(l<=r){

int m = l + (r-l)/2;

if(a[m]>=x){ ans=m; r=m-1; } else l=m+1;

}

return ans; // أول موضع >= x (lower\_bound)

}

// بحث ثنائي على "الجواب"

long long binary\_search\_answer(long long low, long long high, auto ok){

long long ans=-1;

while(low<=hi){

long long mid=(low+high)/2;

if(ok(mid)) ans=mid, high=mid-1;

else low=mid+1;

}

return ans;

}

2) الفرز (Merge Sort / Quick Sort / Counting Sort)

متى أستخدمه؟

* sort الجاهزة غالبًا تكفي (O(n log n)).
* Merge: مستقر، جيد عندما تحتاج الاستقرار.
* Quick: أسرع عمليًا في العادة.
* Counting: عندما النطاقصغير (مثل 0..10^6) ⇒ O(n + K).

// Merge Sort (قالب)

void merge\_sort(vector<int>& a, int l, int r){

if(l>=r) return;

int m=(l+r)/2;

merge\_sort(a,l,m); merge\_sort(a,m+1,r);

vector<int> tmp; tmp.reserve(r-l+1);

int i=l,j=m+1;

while(i<=m || j<=r){

if(j>r || (i<=m && a[i]<=a[j])) tmp.push\_back(a[i++]);

else tmp.push\_back(a[j++]);

}

for(int k=0;k<(int)tmp.size();++k) a[l+k]=tmp[k];

}

// Quick Sort (قالب)

int part(vector<int>& a, int l, int r){

int p=a[r], i=l;

for(int j=l;j<r;++j) if(a[j]<=p) swap(a[i++],a[j]);

swap(a[i],a[r]); return i;

}

void quick\_sort(vector<int>& a, int l, int r){

if(l<r){ int p=part(a,l,r); quick\_sort(a,l,p-1); quick\_sort(a,p+1,r); }

}

// Counting Sort (عندما القيم 0..K)

void counting\_sort(vector<int>& a, int K){

vector<int> cnt(K+1,0);

for(int x:a) ++cnt[x];

int idx=0;

for(int v=0; v<=K; ++v) while(cnt[v]--) a[idx++]=v;

}

3) خوارزميات الرسم البياني

BFS / DFS

متى؟

* BFS: أقصر مسار في جراف غير موجه/أوزانه =1، واكتشاف الطبقات.
* DFS: الوصول/التحقق من الاتصال/اكتشاف الدورات/الترتيب الطوبولوجي (في DAG مع تعديلات).

// BFS على قائمة مجاورات

vector<int> bfs\_dist(int n, const vector<vector<int>>& g, int src){

vector<int> dist(n,-1); queue<int> q;

dist[src]=0; q.push(src);

while(!q.empty()){

int u=q.front(); q.pop();

for(int v:g[u]) if(dist[v]==-1){

dist[v]=dist[u]+1; q.push(v);

}

}

return dist;

}

// DFS تكراري (لتجنب stack overflow)

vector<int> order; // إن أردت ترتيب زيارة

void dfs\_iter(int n, const vector<vector<int>>& g, int s){

vector<int> vis(n,0); stack<int> st; st.push(s);

while(!st.empty()){

int u=st.top(); st.pop();

if(vis[u]) continue; vis[u]=1; order.push\_back(u);

for(int v: g[u]) if(!vis[v]) st.push(v);

}

}

Dijkstra

متى؟

* أقصر مسار في جراف بأوزان غيرسالبة.

// Dijkstra: O((n+m) log n)

vector<long long> dijkstra(int n, const vector<vector<pair<int,int>>>& g, int s){

const long long INF = (long long)4e18;

vector<long long> dist(n, INF);

priority\_queue<pair<long long,int>, vector<pair<long long,int>>, greater<>> pq;

dist[s]=0; pq.push({0,s});

while(!pq.empty()){

auto [d,u]=pq.top(); pq.pop();

if(d!=dist[u]) continue;

for(auto [v,w]: g[u]) if(dist[v]>d+w){

dist[v]=d+w; pq.push({dist[v],v});

}

}

return dist;

}

4) البرمجة الديناميكية (DP)

متى؟

* تركيب مسائل “اختيار أمثل/عدّ طرق” مع تقاطعحالات وبنيةتفكيك (optimal substructure + overlapping subproblems).

// Memoization (Top-Down)

const int MAXN = 1e5+5;

long long dp[MAXN]; // هيّئها بـ LLONG\_MIN أو -1

long long solve(int i){

if(i==0) return 1; // حالة أساس (مثال)

long long &ret = dp[i];

if(ret!=-1) return ret;

ret = 0;

// مثال: ret = solve(i-1) + solve(i-2);

return ret;

}

// Tabulation (Bottom-Up)

long long tabulation(int n){

vector<long long> f(n+1,0);

f[0]=1;

for(int i=1;i<=n;++i){

// f[i] = f[i-1] + f[i-2]; (مثال)

}

return f[n];

}

6) البحث في النصوص

KMP (Knuth–Morris–Pratt)

متى؟

* مطابقة نمط داخل نص بكفاءة O(n+m) دون إعادة فحص زائدة.

vector<int> kmp\_pi(const string& p){

int m=p.size(); vector<int> pi(m,0);

for(int i=1;i<m;++i){

int j=pi[i-1];

while(j>0 && p[i]!=p[j]) j=pi[j-1];

if(p[i]==p[j]) ++j;

pi[i]=j;

}

return pi;

}

vector<int> kmp\_search(const string& s, const string& p){

vector<int> pi=kmp\_pi(p), pos; int n=s.size(), m=p.size(), j=0;

for(int i=0;i<n;++i){

while(j>0 && s[i]!=p[j]) j=pi[j-1];

if(s[i]==p[j]) ++j;

if(j==m){ pos.push\_back(i-m+1); j=pi[j-1]; }

}

return pos;

}

Rabin–Karp

متى؟

* مطابقة متعددة الأنماط أو كشف سريع أولي (مع احتمال تصادم) — عملي لعدة أنماط.

vector<int> rabin\_karp(const string& s, const string& p){

const long long B=911382323, M=1'000'000'007;

int n=s.size(), m=p.size(); if(m>n) return {};

long long ph=0, h=0, powB=1;

for(int I=0;i<m;++i){

ph=(ph\*B + p[I])%M;

h=(h\*B + s[I])%M;

if(i<m-1) powB=(powB\*B)%M; }

vector<int> pos;

if(h==ph && s.substr(0,m)==p) pos.push\_back(0);

for(int i=m;i<n;++i){

h = ( (h - s[i-m]\*powB)%M + M )%M;

h = (h\*B + s[i])%M;

if(h==ph && s.substr(i-m+1, m)==p) pos.push\_back(i-m+1);

}

return pos;

}

STL algorithms

|  |  |
| --- | --- |
| **اسم الخوارزمية** | **متى تستخدم بالتفصيل** |
| **sort** | عند الحاجة إلى ترتيب عناصر حاوية (مثل vector أو array) تصاعديًا أو تنازليًا. ضروري قبل استخدام binary\_search أو عند إزالة التكرارات بـ unique. يمكن تخصيص ترتيب العناصر باستخدام دالة مقارنة (مثال: ترتيب حسب طول السلاسل أو باقي القسمة). |
| **binary\_search** | للتحقق من وجود عنصر معين في حاويةمرتبة. أسرع من البحث الخطي (O(log n)). مناسب لمسائل الوجود/عدم الوجود أو التحقق من قابلية الحل عند فرز البيانات مسبقًا. |
| **lower\_bound** | إيجاد أولعنصرأكبرأويساويقيمةمحددة في حاوية مرتبة. مثالي لإيجاد موقع إدراج عنصر مع الحفاظ على الترتيب، أو لحساب عدد العناصر الأقل من قيمة معينة. |
| **upper\_bound** | إيجاد أولعنصرأكبرمنقيمةمحددة في حاوية مرتبة. مناسب لتحديد نهاية مجموعة من العناصر المتساوية، أو لحساب عدد العناصر الأقل أو المساوية لقيمة معينة. |
| **next\_permutation** | توليد الترتيبالمعجميالتالي لعناصر الحاوية. يُستخدم في مسائل تجربة جميع الترتيبات الممكنة (Permutations) بشكل مرتب. |
| **prev\_permutation** | توليد الترتيبالمعجميالسابق لعناصر الحاوية. مفيد عند استعراض التباديل بالرجوع للخلف. |
| **reverse** | لعكس ترتيب عناصر الحاوية كليًا أو جزئيًا. يُستخدم في قلب النصوص أو قلب نطاق عناصر بعد عملية فرز أو تدوير. |
| **rotate** | لتدوير العناصر بحيث يبدأ النطاق من موضع معين. مثالي لمحاكاة الدوران الدائري أو إعادة ترتيب قائمة بحيث يبدأ العنصر المطلوب أولًا. |
| **accumulate** | لجمع أو دمج كل عناصر الحاوية باستخدام عملية محددة (الجمع الافتراضي أو الضرب أو أي عملية مخصصة). مناسب لمسائل المجموع أو جداء العناصر بسرعة. |

1) sort

متى أستخدمها؟

* لترتيب عناصر مصفوفة أو vector تصاعديًا أو تنازليًا.
* تجهيز البيانات للبحث الثنائي (binary\_search, lower\_bound).
* ترتيب عناصر قبل إزالة التكرارات (unique).

القالب:

#include <bits/stdc++.h>

using namespace std;

int main(){

vector<int> v = {5, 2, 9, 1};

// ترتيب تصاعدي (افتراضي)

sort(v.begin(), v.end());

// ترتيب تنازلي

sort(v.begin(), v.end(), greater<int>());

// ترتيب بمقارنة مخصصة (مثال: ترتيب حسب باقي القسمة على 3)

sort(v.begin(), v.end(), [](int a, int b){

return (a % 3) < (b % 3);

});

for(int x : v) cout << x << " ";

}

2) binary\_search

متى أستخدمها؟

* للبحث عن وجود عنصر في حاويةمرتبة.
* بديل أسرع من البحث الخطي، لكن يحتاج ترتيب مسبق.

القالب:

vector<int> v = {1, 3, 5, 7, 9};

if(binary\_search(v.begin(), v.end(), 5)){

cout << "Found\n";

} else {

cout << "Not Found\n";

}

3) lower\_bound و upper\_bound

متى أستخدمها؟

* lower\_bound: إيجاد أول عنصر **≥** القيمة.
* upper\_bound: إيجاد أول عنصر **>** القيمة.
* مهم جدًا في البحث عن أول/آخر موضع يحقق شرط في بيانات مرتبة.

القالب:

vector<int> v = {1, 3, 3, 5, 7};

auto it1 = lower\_bound(v.begin(), v.end(), 3); // أول عنصر >= 3

cout << "lower\_bound: index = " << (it1 - v.begin()) << "\n"; // 1

auto it2 = upper\_bound(v.begin(), v.end(), 3); // أول عنصر > 3

cout << "upper\_bound: index = " << (it2 - v.begin()) << "\n"; // 3

4) next\_permutation و prev\_permutation

متى أستخدمها؟

* توليد جميع التباديل (Permutations) لمجموعة عناصر بترتيب معجمي.
* حل مسائل تتطلب تجربة كل الترتيبات.

القالب:

string s = "123";

do {

cout << s << "\n";

} while(next\_permutation(s.begin(), s.end()));

// للرجوع للخلف:

prev\_permutation(s.begin(), s.end());

5) reverse

متى أستخدمها؟

* لعكس ترتيب العناصر في مجال محدد.
* مفيدة في عكس النصوص أو المصفوفات.

القالب:

vector<int> v = {1, 2, 3, 4};

reverse(v.begin(), v.end()); // الآن v = {4, 3, 2, 1}

6) rotate

متى أستخدمها؟

* لتدوير العناصر بحيث يبدأ المجال من نقطة جديدة.
* مفيد في محاكاة الدوران الدائري.

القالب:

vector<int> v = {1, 2, 3, 4, 5};

rotate(v.begin(), v.begin()+2, v.end());

// الآن v = {3, 4, 5, 1, 2}

7) accumulate

متى أستخدمها؟

* لجمع كل العناصر في حاوية بسهولة.
* يمكن تخصيص العملية (جمع، ضرب، إلخ).

القالب:

#include <numeric>

vector<int> v = {1, 2, 3, 4};

int sum = accumulate(v.begin(), v.end(), 0); // 10

// مثال على الضرب:

int product = accumulate(v.begin(), v.end(), 1, multiplies<int>()); // 24