https://simplifycpp.org

خوارزميات C++ الحديثة

النظرية، التصميم، والتطبيق عالي الأداء

إعداد: أيمن الحراكي

خوارزميات C++ الحديثة

النظرية، التصميم، والتطبيق عالي الأداء

تمت الاستعانة بأدوات ذكاء اصطناعي حديثة في بعض مراحل الصياغة واستكشاف الأفكار، وذلك تحت إشراف كامل، مع المراجعة والتحقق والتصحيح، مع احتفاظ المؤلف الكامل بحقوق التأليف والمسؤولية العلمية.

إعداد : أيمن الحراكي

simplifycpp.org

يناير 2026

المحتويات

تمهيد المؤلف

تُشكِّل الخوارزميات الجوهر الفكري لكل نظام برمجي فعّال، قابل للتوسّع، وموثوق. فإلى جانب إنتاج نتائج صحيحة، تحدد جودة الخوارزمية بشكل مباشر خصائص الأداء، واستهلاك الموارد، وقابلية الصيانة على المدى الطويل. إن الإتقان الحقيقي للخوارزميات لا يقتصر على معرفة الإجراءات، بل يتطلب صرامة تحليلية، واستدلالاً رياضياً، وحكماً هندسياً منضبطاً.

كُتب هذا الكتاب *Modern C++ Algorithms: A Graduate-Level Companion* لطلاب الدراسات العليا، والباحثين، ومهندسي البرمجيات المحترفين الذين يمتلكون أساساً قوياً في **C++**، ويسعون إلى تعميق فهمهم لتصميم الخوارزميات، وتحليلها، وتنفيذها باستخدام **Modern C++**. هذا ليس كتاباً تمهيدياً؛ إذ يفترض معرفة مسبقة بهياكل البيانات الأساسية، وتعقيد الخوارزميات، وبنى لغة C++ الجوهرية.

الهدف المركزي لهذا الكتاب هو ربط النظرية الخوارزمية الكلاسيكية بالتنفيذ العملي عالي الجودة المخصص للإنتاج. ويجري التركيز على التعبير عن الخوارزميات بوضوح وكفاءة عبر الاستفادة من ميزات C++ الحديثة مثل البرمجة العامة (Generic Programming)، والقوالب (Templates)، ومكتبة القوالب القياسية (STL)، مع الالتزام الصارم بالصحة، والأداء، وقابلية الصيانة.

يسعى هذا العمل إلى الارتقاء بالقارئ من مجرد تطبيق الخوارزميات الجاهزة إلى التفكير العميق في تصميمها، ومفاضلاتها، واستراتيجيات تنفيذها في الأنظمة الحقيقية. ولا يزال المخطوط قيد المراجعة النشطة، وتُشارك هذه النسخة الأولية بهدف دعوة القرّاء لتقديم ملاحظات نقدية، وتصحيحات، واقتراحات تُسهم في تعزيز الدقة التقنية والوضوح التعليمي.

أدعوكم لمتابعتي على **LinkedIn**:

https://linkedin.com/in/aymanalheraki

كما يمكنكم زيارة موقع الشركة:

https://simplifycpp.org

مع تمنياتي للجميع بالتوفيق والازدهار.

أيمن الحراكي

الباب ا

الأسس (مرتكزة على C++)

الفصل ١: مقدمة وكيفية استخدام هذا الكتاب

١.١ الفئة المستهدفة والمتطلبات المسبقة (C++17/20/23)

يستهدف هذا الكتاب القرّاء المتقدمين الذين يطمحون إلى إتقان التفكير الخوارزمي وتنفيذه الصارم باستخدام **Modern C++**. وهو ليس كتاباً تمهيدياً في البرمجة، بل رفيقاً أكاديمياً متقدماً لطلاب الدراسات العليا والمحترفين الذين يمتلكون خلفية متينة في C++ ويرغبون في تعميق فهمهم لتصميم الخوارزميات، وتحليلها، وتطبيقها العملي.
تشمل الفئة المستهدفة ما يلي:

- طلاب الدراسات العليا وما بعدها في علوم الحاسوب، أو الرياضيات، أو الهندسة، حيث يُكمّل هذا الكتاب المقررات المتقدمة، ومشاريع البحث، وأعمال الرسائل العلمية.
- الباحثون والأكاديميون العاملون في مجالات مثل الحوسبة عالية الأداء، والتحسين، وخوارزميات الرسوم البيانية، والطرق العددية، أو تعلم الآلة الموجّه للأنظمة.
- مهندسو البرمجيات المحترفون ومطورو الأنظمة الذين يبنون برمجيات حسّاسة للأداء في مجالات مثل التمويل، والحوسبة العلمية، والأنظمة المضمنة، أو البنى التحتية واسعة النطاق.
- ممارسو **C++** ذوو الخبرة الذين ينتقلون من المعايير القديمة إلى C++17 وC++20 وC++23، ويبحثون عن أنماط خوارزمية حديثة واصطلاحية.

١.١.١ المتطلبات المسبقة

للاستفادة الكاملة من هذا الكتاب، يُتوقع من القارئ امتلاك ما يلي:

- معرفة قوية بلغة **C++** حتى C++17 على الأقل، بما في ذلك القوالب، ودلالات النقل (Move Semantics)، والتعابير اللامبدا، والمؤشرات الذكية. كما يُنصح بشدة بالإلمام بميزات C++20/23 مثل Concepts وRanges و Coroutines.
- أسس رياضية في الرياضيات المتقطعة، والجبر الخطي الأساسي، والاحتمالات، وتحليل التعقيد التقاربي.
- خلفية أساسية في علوم الحاسوب تشمل هياكل البيانات (المصفوفات، الأشجار، الرسوم البيانية، جداول التجزئة) وأنماط الخوارزميات مثل العودية، والتقسيم والتغلب، والبرمجة الديناميكية.
- أدوات تطوير حديثة تشمل مترجمات تدعم C++20/23 ومعرفة أساسية بأنظمة البناء المعتمدة على CMake.

كما تُعد الخبرة في نماذج البرمجة المتوازية، وأدوات القياس، والاختبار، وأنظمة التحكم بالإصدارات إضافة مفيدة ولكنها غير إلزامية.

٢.١.١ موضع هذا الكتاب

على خلاف كتب الخوارزميات التمهيدية التي تركز على الشيفرة الوهمية أو العرض الحيادي للغات، يركز هذا الكتاب على تنفيذات **C++** عالية الجودة ومطابقة للمعايير الحديثة. تُعرض كل خوارزمية بأساسها النظري، وتُنفذ بأسلوب اصطلاحي في C++17/20/23، مع التركيز على الصحة، والأداء، وقابلية الصيانة.
ومن خلال وضع متطلبات عالية، يتجنب الكتاب تكرار المواد الأولية، ويفتح المجال للتعامل المباشر مع موضوعات متقدمة مثل تدفقات الرسوم البيانية، وتقنيات التحسين، والخوارزميات الواعية للتوازي بعمق مهني وبحثي.

٢.١ معايير الترميز المستخدمة في الأمثلة

تتبع جميع أمثلة الشيفرة معايير صارمة ومتسقة لضمان الوضوح، وسهولة القراءة، والتوافق مع أفضل ممارسات C++ الحديثة.

١.٢.١ التنسيق والأسلوب

- تنسيق آلي متوافق مع اصطلاحات LLVM/Clang-Format.
- أربع مسافات لكل مستوى إزاحة؛ ولا تُستخدم علامات الجدولة.
- طول السطر غالباً محدود بين 100 و120 محرفاً.
- وضع الأقواس في السطر نفسه مع التصريحات أو عبارات التحكم.
- استخدام واضح للمسافات البيضاء دون مسافات زائدة في نهاية السطر.
- استخدام // للتعليقات القصيرة وتعليقات بأسلوب Doxygen للواجهات العامة.

٢.٢.١ اصطلاحات التسمية

- الدوال والمتغيرات: camelCase.
- الأنواع (الأصناف والبُنى): PascalCase.
- الثوابت: ALL_CAPS_WITH_UNDERSCORES.
- نطاقات الأسماء: موجزة، بحروف صغيرة، وذات بنية هرمية.

١.٢.٣ تنظيم ملفات الترويس والمصدر

تتبع الأمثلة فصلاً نظيفاً بين الواجهة والتنفيذ:

- تستخدم ملفات الترويس `#pragma once` وتحتوي على التصريحات.
- تحتوي ملفات المصدر على التعريفات وتفاصيل التنفيذ.
- ترتيب التضمين: ملف الترويس المقابل، ثم المكتبة القياسية، ثم مكتبات الطرف الثالث، ثم ترويسات المشروع.

٤.٢.١ ممارسات C++ الحديثة

تُطبَّق أفضل الممارسات الحديثة باستمرار في جميع الأمثلة:

- تفضيل `constexpr` و`noexcept` و`[[nodiscard]]` و`auto` حيثما كان ذلك مناسباً.
- فرض مبدأ RAII لإدارة الموارد.
- تفضيل المؤشرات الذكية على الملكية الخام.
- استخدام ميزات C++20/23 مثل Ranges و Concepts و Structured Bindings.
- تجنب البنى المهجورة أو غير الآمنة.

٣.١ البناء والتشغيل: سير عمل قابل للإعادة الإنتاج

يركز هذا الكتاب على سير عمل مهني وقابل للإعادة الإنتاج. تُبنى جميع الأمثلة باستخدام **CMake**، مع تحديد صريح لمعيار اللغة، وتحذيرات صارمة من المترجم، وخيارات اختيارية لأدوات الفحص أثناء التشغيل.

يتكون سير العمل الأدنى من تهيئة المشروع، وبنائه، وتشغيله، واختباره في مجلدات بناء معزولة، مما يضمن القابلية للنقل والكشف المبكر عن الأخطاء. وتُستخدم أدوات الفحص والاختبارات الخفيفة للتحقق من الصحة، لا سيما في الخوارزميات الحساسة للأداء أو التوازي.

يضمن هذا النهج أن تكون كل خوارزمية معروضة ليست صحيحة نظرياً فحسب، بل قوية وقابلة للاختبار ومناسبة لبيئات البحث أو الإنتاج الحقيقية.

الباب ٢

البُنى الخطية والأساسية (مع تطبيقات C++)

الفصل ٢: المصفوفات والمتجهات

٢.١ المصفوفات الثابتة مقابل std::vector: الذاكرة والأداء

تُعدّ المصفوفات الثابتة و std::vector من الحاويات الخطية الأساسية في C++. ويتطلب الاختيار بينهما فهماً دقيقاً لتخطيط الذاكرة، وخصائص الأداء، وقيود المرونة.

٢.١.١ المصفوفات الثابتة

تمثل المصفوفات الثابتة كتل ذاكرة متجاورة ذات حجم ثابت، تُخصَّص في وقت الترجمة (على المكدس) أو كذاكرة ثابتة/عالمية.

- حجم ثابت معروف وقت الترجمة.
- تخطيط متجاور مع أفضلية عالية للاستفادة من الذاكرة المخبئية.
- دون بيانات وصفية أو كلفة تخصيص ديناميكي.
- وصول سريع جداً ولكن دون فحص للحدود.

تُناسب هذه المصفوفات الأحمال الحساسة للأداء وذات الحجم المستقر، مثل الروتينات العددية منخفضة المستوى أو الأنظمة المضمنة. وتتمثل قيودها الرئيسية في عدم القابلية لتغيير الحجم والسلوك غير الآمن عند تجاوز الحدود.

٢.١.٢ std::vector

يُعدّ std::vector حاوية متجاورة ديناميكية قابلة لتغيير الحجم، تُدار بواسطة المكتبة القياسية.

- نمو ديناميكي مع إدراج بزمن O(1) مُعدَّل في نهاية الحاوية.
- ذاكرة متجاورة مع أداء ممتاز للذاكرة المخبئية.
- إدارة تلقائية للذاكرة وضمانات قوية للسلامة في وجود الاستثناءات.
- كلفة إضافية بسيطة بسبب بيانات الحجم والسعة والمُخصِّص.

تُعد المتجهات الخيار الافتراضي لمعظم الخوارزميات التي تتطلب أحجاماً مرنة، وتكاملاً مع STL، وسهولة في الصيانة.

٣.١.٢ اعتبارات الذاكرة والمخبأ

تستفيد كل من المصفوفات والمتجهات من الوصول المتسلسل وآليات الجلب المسبق العتادية. إلا أن إعادة التخصيص المتكررة في المتجهات قد تؤثر مؤقتاً على سلوك المخبأ. ويمكن التخفيف من ذلك بحجز السعة مسبقاً.

٤.١.٢ إرشادات

- استخدم المصفوفات الثابتة عندما يكون الحجم ثابتاً والكلفة الدنيا حرجة.
- استخدم std::vector عندما يتغير الحجم أو يكون التكامل مع STL مطلوباً.
- فضّل std::array للحاويات ثابتة الحجم مع دلالات STL.

٢.٢ تقنيات الخوارزميات داخل المكان (In-Place)

تعمل الخوارزميات داخل المكان مباشرة على المصفوفات أو المتجهات دون استخدام ذاكرة إضافية كبيرة، مما يحسن الاستفادة من المخبأ والأداء.

١.٢.٢ النافذة المنزلقة

تعالج تقنية النافذة المنزلقة المقاطع المتجاورة بكفاءة عبر تحديث الحالة تدريجياً. وتحقق زمناً O(n) ومساحة O(1)، وتُستخدم على نطاق واسع في حساب المجاميع، والقيم الصغرى، والعظمى ضمن النطاقات.

٢.٢.٢ طريقة المؤشرين

تجتاز هذه الطريقة السلاسل من اتجاهات متعددة، وغالباً ما تُطبّق على البيانات المرتبة لحل مسائل الأزواج والفواصل. وتوفّر حلولاً بزمن خطي ومساحة إضافية ثابتة.

٣.٢.٢ التقسيم

يعيد التقسيم ترتيب العناصر حول محور أو شرط، ويُعدّ أساسياً في خوارزمية Quicksort ومسائل التصنيف. وتوفر المكتبة القياسية أدوات مثل std::partition بتنفيذات محسّنة وآمنة.

٤.٢.٢ أفضل الممارسات

- تقليل تخصيصات الذاكرة المساعدة.
- التحقق الدقيق من حدود الفهارس.
- تفضيل خوارزميات STL لضمان الصحة والوضوح.
- دمج التقنيات لحل المسائل المعقدة بكفاءة.

٣.٢ تمارين وأنماط

١.٣.٢ الأنماط الأساسية

- تدوير المصفوفة داخل المكان باستخدام العكس.
- حساب مجموع المقاطع باستخدام المجاميع السابقة.
- مصفوفات البادئة واللاحقة للاستعلامات ذات الزمن الثابت.

٢.٣.٢ نواتج التعلّم

تعزز هذه التمارين التصميم الموفر للذاكرة، والتحكم بالفهارس، والتفكير الخوارزمي. ويُعدّ إتقان هذه الأنماط ضرورياً لتطوير خوارزميات عالية الأداء على البُنى الخطية.

٤.٢ الخلاصة

- توفر المصفوفات الثابتة أقصى أداء مع أقل مرونة.
- يوازن std::vector بين الأداء والسلامة وسهولة الاستخدام.
- تقلل التقنيات داخل المكان من استهلاك الذاكرة وتحسن سلوك المخبأ.
- تشكل المصفوفات والمتجهات أساساً للعديد من الخوارزميات المتقدمة في Modern C++.

الفصل ٣: القوائم المرتبطة

١.٣ تنفيذ القوائم المرتبطة في Modern C++

القوائم المرتبطة بُنى ديناميكية مُحسّنة للإدراج والحذف المتكرر. وعلى عكس الحاويات المتجاورة، فإنها تُضحّي بالاستفادة من المخبأ مقابل مرونة بنيوية. تتيح C++ الحديثة تنفيذها باستخدام المؤشرات الخام أو المؤشرات الذكية، ولكلٍ مفاضلاته.

١.١.٣ القوائم الأحادية

تتكون القائمة الأحادية من عُقد تحتوي على بيانات ومؤشر إلى العقدة التالية.

- التنفيذ بالمؤشرات الخام: كلفة دنيا وأداء عالٍ، لكنه يتطلب إدارة صريحة للذاكرة وانتباهًا صارماً للملكية.
- التنفيذ بالمؤشرات الذكية (غالباً std::unique_ptr): يفرض ملكية أحادية، ويضمن التحرير التلقائي، ويوفر سلامة في وجود الاستثناءات.

في Modern C++ يُعدّ std::unique_ptr الخيار المفضل للقوائم الأحادية، لتوافقه الطبيعي مع نموذج الملكية ومنعه لتسرب الذاكرة.

٢.١.٣ القوائم الثنائية

تدعم القوائم الثنائية الاجتياز في الاتجاهين عبر تخزين رابطَي next وprev.

- تنفيذات المؤشرات الخام بسيطة لكنها عرضة للأخطاء.
- نمط حديث شائع يستخدم std::unique_ptr للرابط next ومؤشراً خاماً (أو std::weak_ptr) للرابط prev لتجنب دورات الملكية.

يوازن هذا النهج الهجين بين السلامة والعملية مع الحفاظ على الإدارة التلقائية للذاكرة.

٣.١.٣ المؤشرات الخام مقابل المؤشرات الذكية

- المؤشرات الخام: أقصى أداء وتحكم صريح، مع مخاطر عالية للتسرب والمراجع المتدلية.
- المؤشرات الذكية: كلفة أعلى قليلاً، دلالات ملكية قوية، تنظيف تلقائي، وسلامة في وجود الاستثناءات.

يوصى في معظم شيفرات C++ الحديثة باستخدام المؤشرات الذكية، مع حصر المؤشرات الخام في السياقات منخفضة المستوى أو الحرجة للأداء مع ملكية واضحة.

٢.٣ الخوارزميات الأساسية على القوائم المرتبطة

تعمل العديد من خوارزميات القوائم المرتبطة بكفاءة داخل المكان وبمساحة إضافية ثابتة.

١.٢.٣ العكس

يعيد عكس قائمة أحادية ربط العقد تكرارياً بزمن O(n) ومساحة ،O(1) ويُعد أساسياً لآليات التراجع والسلوك الشبيه بالمكدس.

٢.٢.٣ كشف الدورات

تكتشف خوارزمية السلحفاة والأرنب لفلويد الدورات باستخدام مؤشرين بسرعات مختلفة، بزمن خطي ومساحة ثابتة، وهي أساسية للتحقق من سلامة القوائم.

٣.٢.٣ دمج القوائم المرتبة

يمكن دمج قائمتين مرتبتين بزمن خطي عبر إعادة ربط العقد الموجودة دون تخصيص ذاكرة جديدة، مما يجعلها مثالية لفرز الدمج على القوائم ودمج التدفّقات.

٤.٢.٣ إزالة العقدة رقم N من النهاية

تتيح تقنية المؤشرين إزالة العقدة رقم N من النهاية بمرور واحد فقط، محققةً زمناً $O(n)$ ومساحة $O(1)$.

٣.٣ التصميم وأفضل الممارسات

- تفضيل الخوارزميات داخل المكان لتقليل كلفة الذاكرة.
- معالجة الحالات الحدّية صراحةً (قائمة فارغة، عقدة واحدة).
- استخدام أنماط المؤشرين لمسائل الاجتياز.
- تفضيل واجهات شبيهة بـ STL (المكررات) بدلاً من كشف العقد الخام.

٤.٣ المكررات والاختبار

يتيح دعم المكررات تكامل القوائم المرتبطة بسلاسة مع حلقات for المعتمدة على النطاق والخوارزميات القياسية.

- تحسن المكررات سهولة الاستخدام وقابلية التركيب.
- يجب أن تتحقق اختبارات الوحدة من الصحة والاجتياز والحالات الحدّية.
- ينبغي استخدام أدوات الفحص مثل Valgrind والمُعقِّمات للتأكد من سلامة الذاكرة.

٥.٣ الخلاصة

- القوائم الأحادية بسيطة وتنسجم طبيعياً مع std::unique_ptr.
- تتطلب القوائم الثنائية معالجة حذرة للروابط الخلفية لتجنب دورات الملكية.
- توفر المؤشرات الذكية السلامة والصحة، بينما تتطلب المؤشرات الخام انضباطاً عالياً.
- تعمل العديد من خوارزميات القوائم المرتبطة بزمن خطي ومساحة ثابتة باستخدام تقنيات قائمة على المؤشرات.
- تجمع التنفيذات المتينة بين إدارة ذاكرة آمنة، وملكية واضحة، ودعم المكررات، واختبارات شاملة.

الفصل ٤: المكدسات والطوابير والمزدوجات وطوابير الأولوية

١.٤ مهايئات STL مقابل التنفيذات المخصصة

تُعدّ المكدسات والطوابير والمزدوجات وطوابير الأولوية بُنى خطية أساسية تتحكم في ترتيب التنفيذ والاجتياز وتحديد الأولويات. وتوفر C++ الحديثة مهايئات حاويات STL آمنة ومحسّنة وموحّدة، إلى جانب إمكانية بناء هياكل مخصصة لاحتياجات خاصة.

١.١.٤ مهايئات الحاويات في STL

تُقدَّم هذه البُنى في المكتبة القياسية أساساً على شكل مهايئات فوق حاويات داخلية:

- std::stack: وصول بأسلوب LIFO، وغالباً ما يُبنى فوق std::deque.
- std::queue: وصول بأسلوب FIFO، ويُبنى أيضاً فوق std::deque.
- std::deque: طابور مزدوج الأطراف مع إدراج وحذف فعّالين من الطرفين.
- std::priority_queue: بنية قائمة على الكومة (افتراضياً كومة عظمى) ومبنية فوق std::vector.

خصائص التصميم:

- واجهات مقيّدة تفرض الاستخدام الصحيح.
- سلامة قوية في وجود الاستثناءات وتعقيد متوقّع.
- تنفيذات محسّنة ومجرّبة جيداً.

٢.١.٤ التنفيذات المخصصة

تُنفَّذ المكدسات أو الطوابير المخصصة عادة باستخدام القوائم المرتبطة أو المصفوفات.

- تتيح تحكماً دقيقاً في تخطيط الذاكرة.
- تمكّن من سلوكيات خاصة (مكدسات محدودة، طوابير بلا أقفال).
- تزيد من تعقيد التنفيذ ومخاطر أخطاء الذاكرة.

في C++ الحديثة، تُبرَّر التنفيذات المخصصة أساساً في السيناريوهات الآنية، أو المضمنة، أو التجريبية.

٣.١.٤ متى نستخدم كل خيار

- فضّل مهايئات **STL** للصحة وسرعة التطوير وسهولة الصيانة.
- استخدم الهياكل المخصصة فقط عندما تفرض القيود أو القياس تحكماً أدق.

٢.٤ الأنماط الخوارزمية وحالات الاستخدام

تُصبح هذه البُنى قوية عند تطبيقها على أنماط خوارزمية معروفة.

١.٣.٤ المكدسات: تحليل التعابير

تمثل المكدسات السلوك المتداخل وآخر داخل أول خارج.

- تقييم التعابير (اللاحقة والسابقة).
- التحقق من الأقواس والبنية النحوية.
- الاجتياز بعمق ومحاكاة العودية.

يعمل التحليل القائم على المكدس غالباً بزمن $O(n)$ ومساحة مساعدة خطية.

٢.٣.٤ الطوابير: البحث بعرض الشجرة

تمكّن الطوابير من الاستكشاف بالمستويات في الرسوم البيانية والأشجار.

- البحث بعرض الشجرة.
- أقصر مسار في الرسوم غير الموزونة.
- جدولة المهام ومعالجة الأحداث.

يحقق BFS زمناً $O(V + E)$ ومساحة مساعدة $O(V)$.

٣.٣.٤ المزدوجات: تحسين النافذة المنزلقة

تدعم المزدوجات عمليات فعّالة من الأمام والخلف، مما يجعلها مثالية للخوارزميات المعتمدة على النوافذ.

- القيم العظمى/الصغرى في النافذة المنزلقة.
- الطوابير الرتيبة.
- الحسابات المتدفقة والآنية.

تضمن تقنيات المزدوج الرتيب زمناً $O(n)$ عبر إدراج وحذف كل عنصر مرة واحدة كحد أقصى.

٤.٢.٤ طوابير الأولوية: الترتيب الديناميكي

تحافظ طوابير الأولوية على ترتيب جزئي ديناميكي باستخدام الكومات.

- خوارزمية ديكسترا لأقصر مسار.
- استعلامات أعلى k عناصر.
- جدولة المهام والخوارزميات الجشعة.

تكلف العمليات عادة زمناً O(log n)، مع وصول بزمن ثابت لأعلى أولوية.

٣.٤ تمارين وأنماط أساسية

١.٣.٤ الطابور الرتيب

يحافظ الطابور الرتيب على العناصر مرتبة لدعم القيم القصوى ضمن النوافذ بزمن خطي.

- القيم العظمى/الصغرى في النافذة المنزلقة.
- زمن O(n)، ومساحة O(k).
- شائع في التحليلات المتدفقة والآنية.

٢.٣.٤ أكبر K عناصر باستخدام الكومات

تتبع كومة ثابتة الحجم أكبر (أو أصغر) k عناصر بكفاءة.

- كومة صغرى بحجم k لمسائل أكبر k.
- زمن O(n log k)، ومساحة O(k).
- مستخدمة على نطاق واسع في الترتيب والتوصية ومعالجة التدفّقات.

٤.٤ الخلاصة

- توفر مهايئات STL بُنى خطية آمنة ومحسّنة وموحّدة.
- تمنح التنفيذات المخصصة تحكماً أكبر لكنها تزيد التعقيد والمخاطر.
- تتحكم المكدسات في ترتيب التقييم، وتدير الطوابير الاجتياز، وتحسّن المزدوجات النوافذ، وتعالج طوابير الأولوية الترتيب الديناميكي.
- يُعدّ إتقان هذه الأنماط أساسياً لتصميم الخوارزميات الحديثة في C++.

الفصل ٥: التجزئة والحاويات غير المرتبة

١.٥ الحاويات غير المرتبة في Modern C++

توفر المكتبة القياسية في C++ الحاويتين std::unordered_map وstd::unordered_set كبُنى ترابطية قائمة على جداول التجزئة. وتقدم زمناً ثابتاً في المتوسط للإدراج والبحث والحذف، مما يجعلها لا غنى عنها في الخوارزميات الحساسة للأداء المعتمدة على الوصول السريع بالمفتاح.

١.١.٥ البنية الداخلية

تُنفَّذ الحاويات غير المرتبة كجداول تجزئة تضم المكونات الأساسية التالية:

- السلال: مصفوفة خانات تُفهرس بقيم التجزئة.
- دالة التجزئة: تربط المفاتيح بفهارس السلال (افتراضياً std::hash<Key>).
- معالجة التصادم: غالباً عبر السلاسل المنفصلة؛ وقد تستخدم بعض التنفيذات أشكالاً من العنونة المفتوحة.
- عامل التحميل: نسبة العناصر المخزنة إلى عدد السلال؛ تؤدي القيم العالية إلى زيادة احتمالية التصادم.

في الظروف الطبيعية تعمل العمليات بزمن O(1) في المتوسط، بينما يتدهور أسوأ حال إلى O(n) عند كثرة التصادمات.

٢.١.٥ سلوك التصادم

تحدث تصادمات التجزئة عندما تُعيَّن مفاتيح مختلفة إلى السلة نفسها.

- تخزن السلاسل المنفصلة العناصر المتصادمة في قوائم لكل سلة.
- تزيد إعادة التجزئة عدد السلال عند تجاوز عامل التحميل عتبة محددة.
- ترتيب الاجتياز غير محدد وقد يتغير بعد إعادة التجزئة.

يعتمد أداء الحاويات غير المرتبة بشكل كبير على جودة دالة التجزئة.

٣.١.٥ دوال التجزئة المخصصة

تُعد دوال التجزئة المخصصة ضرورية للأنواع المعرفة من قبل المستخدم أو المفاتيح المركبة. ويجب أن يوفّر نوع المفتاح دالة تجزئة ومقارنة مساواة.

- تنفيذ operator== بما يتوافق مع دالة التجزئة.
- دمج تجزئات الأعضاء لتقليل التصادمات.
- تفضيل التركيبات البسيطة والسريعة.

تتيح التجزئة المخصصة تخزيناً فعالاً للبُنى، والـ tuples، والمفاتيح الخاصة بالتطبيق.

٤.١.٥ التحكم في عامل التحميل

يوازن عامل التحميل بين استهلاك الذاكرة وأداء البحث.

- القيمة الافتراضية القصوى غالباً 0.1.
- تخفيضها يقلل التصادمات على حساب الذاكرة.
- تتيح reserve() وrehash() الحجز الاستباقي للسلال.

يُنصح بالحجز المسبق عندما يكون الحجم المتوقع للحاوية معروفاً.

٢.٥ أنماط خوارزمية قائمة على التجزئة

تُعد جداول التجزئة محورية في العديد من الحلول الخوارزمية الشائعة بسبب سرعة الوصول في المتوسط.

١.٢.٥ حساب التكرار

يُعدّ عدّ تكرار العناصر حالة استخدام نموذجية.

- مرور واحد على البيانات.
- زمن O(n) في المتوسط، ومساحة O(n).
- مناسب للتدفّقات والسجلات والتحليلات.

يمتد هذا النمط طبيعياً إلى السلاسل والمفاتيح المخصصة.

٢.٢.٥ مسائل المجموع الزوجي

تمكّن الخرائط التجزئية من حلول خطية لمسائل مجموع الأزواج.

- تخزين القيم التي شوهدت سابقاً.
- التحقق من القيم المكملة بزمن O(1) في المتوسط.
- تقليل الحلول التربيعية إلى تعقيد خطي.

ويمتد المفهوم إلى مسائل k-sum مع بنى إضافية.

٣.٢.٥ التخزين المؤقت والحفظ (Memoization)

تشكل الحاويات القائمة على التجزئة أساساً للتخزين المؤقت والحفظ.

- تخزين النتائج المحسوبة لإعادة استخدامها.
- تقليل الحسابات المتكررة بشكل كبير في الخوارزميات العودية.
- أساسية في البرمجة الديناميكية والأنظمة المتصلة.

ومن الأنماط المتقدمة الشائعة مخبأ LRU، الذي يجمع بين خريطة تجزئة وحاوية خطية للحفاظ على ترتيب الحداثة.

٣.٥ تمارين متقدمة وأنماط تصميم

١.٣.٥ تصميم مخبأ LRU

يحقق مخبأ LRU وصولاً وتحديثاً بزمن O(1) عبر الجمع بين:

- std::unordered_map للوصول السريع بالمفتاح.
- std::list (أو مزدوج) للحفاظ على ترتيب الوصول.

يوضح هذا النمط تكامل التجزئة مع بُنى أخرى لحل مشكلات أداء واقعية.

٢.٣.٥ استراتيجيات العنونة المفتوحة

إلى جانب تنفيذات STL، توفر العنونة المفتوحة فهماً أعمق لآليات جداول التجزئة.

- الفحص الخطي: البحث المتسلسل عن سلة فارغة.
- تجزئة روبن-هود: موازنة أطوال الفحص لتقليل التباين.

تستبدل هذه الاستراتيجيات مؤشرات الارتباط بتحسين في محلية الذاكرة، لكنها تتطلب معالجة دقيقة للحذف وإعادة التجزئة.

٤.٥ أفضل الممارسات

- استخدم الحاويات غير المرتبة للأحمال كثيفة البحث.
- عرّف دوال تجزئة قوية للمفاتيح المخصصة.
- احجز السلال مسبقاً متى أمكن.
- راقب عامل التحميل لتجنب الأداء المرضي.
- ادمج التجزئة مع البُنى الخطية للأنماط المتقدمة.

٥.٥ الخلاصة

- توفر الحاويات غير المرتبة وصولاً متوسطه O(1) عبر التجزئة.
- يعتمد الأداء على جودة دالة التجزئة وإدارة عامل التحميل.
- تشكّل الخوارزميات القائمة على التجزئة أساس تحليل التكرار ومسائل الأزواج والتخزين المؤقت.
- تعمّق الأنماط المتقدمة مثل مخابئ LRU والعنونة المفتوحة فهم تصميم جداول التجزئة في الواقع العملي.
- يُعدّ إتقان التجزئة أساسياً لتطوير خوارزميات عالية الأداء في Modern C++.

الباب ٣

الأشجار والأشجار المتوازنة

الفصل ٦: الأشجار الثنائية واجتياز الأشجار

١.٦ البنية، استراتيجيات الاجتياز، ونماذج التكرار

تُعدّ الأشجار الثنائية بنية هرمية أساسية تقوم عليها الأكوام (Heaps)، وأشجار البحث، وأشجار التعابير، والعديد من هياكل الفهرسة. ولا تكمن أهميتها في التخزين فقط، بل في الأنماط الخوارزمية التي يتيحها الاجتياز المنظم. يركّز هذا الفصل على تمثيل العقد، ويُقارن بين الاجتياز العودي والتكراري، ويقدّم الوصول بأسلوب المكررات بما يتوافق مع مبادئ التصميم في Modern C++.

١.١.٦ نماذج تمثيل العقد

يمكن تمثيل عقد الشجرة الثنائية بعدة طرق، يعكس كل منها مفاضلات مختلفة.

- المؤشرات الخام: كلفة دنيا وتحكم صريح، لكنها عرضة للأخطاء.
- المؤشرات الذكية: وضوح في الملكية وإدارة تلقائية للعمر.
- التخزين المعتمد على الفهارس: تمثيل مضغوط مناسب للحالات الثابتة أو الحساسة للمخبأ.

في Modern C++ يُفضَّل غالباً استخدام std::unique_ptr لامتلاك بنى الأشجار، لما يوفره من أمان دون كلفة زمنية إضافية.

٢.١.٦ الاجتياز العودي

يعكس الاجتياز العودي التعريف الاستقرائي للأشجار، وغالباً ما يكون الأكثر وضوحاً وقراءة.

- Preorder: معالجة الجذر أولاً.
- Inorder: زيارة مرتبة في أشجار البحث الثنائية.
- Postorder: تقييم الأبناء قبل العقدة.

المزايا:

- مختصر ومعبر.
- يعكس دلالات الشجرة بشكل طبيعي.

القيود:

- عمق المكدس يتناسب مع ارتفاع الشجرة.
- خطر تجاوز المكدس في الأشجار المائلة.

٣.١.٦ الاجتياز التكراري

يستبدل الاجتياز التكراري العودية الضمنية بهياكل بيانات صريحة.

- استخدام المكدسات للاجتياز بعمق.
- استخدام الطوابير للاجتياز بعرض الشجرة.
- إتاحة الإنهاء المبكر والتحكم الدقيق.

يُفضَّل هذا الأسلوب في الأشجار العميقة أو العدائية حيث تكون سلامة المكدس حاسمة.

٤.١.٦ مهايئات المكررات

تُغلّف مهايئات المكررات منطق الاجتياز خلف واجهات تكرار قياسية.

- تمكّن حلقات for المعتمدة على النطاق.
- تتكامل مع أدوات <algorithm>.
- تفصل منطق الاجتياز عن الخوارزميات.

يتماشى الاجتياز القائم على المكررات مع أسلوب استخدام الحاويات الاصطلاحي في Modern C++.

٢.٦ الخوارزميات الأساسية على الأشجار

١.٢.٦ الاجتيازات بعمق

تستكشف اجتيازات العمق الفروع كاملة قبل الرجوع.

- Preorder: نسخ الأشجار والتسلسل.
- Inorder: اجتياز مرتب لأشجار البحث الثنائية.
- Postorder: التدمير وتقييم التعابير.

تعمل جميع اجتيازات العمق بزمن O(n)، مع مساحة مساعدة تتناسب مع ارتفاع الشجرة.

٢.٢.٦ الاجتياز بعرض الشجرة

يعالج الاجتياز بالمستويات العقد حسب العمق.

- يُنفَّذ باستخدام طابور.

- أساسي لمسائل أقصر مسار والحسابات الطبقية.
- يتوافق طبيعياً مع التسلسل بالمستويات.

٣.٢.٦ التسلسل وإعادة البناء

يحوّل التسلسل الشجرة إلى تمثيل خطي مناسب للتخزين أو النقل.

- Preorder مع مؤشرات null: بسيط وغير ملتبس.
- Level-order: مضغوط للأشجار الكاملة أو المتوازنة.

تعيد عملية فك التسلسل بناء البنية بشكل حتمي وفق الترميز المختار.

٣.٦ إعادة البناء والمقارنة البنيوية

١.٣.٦ إعادة بناء الشجرة من الاجتيازات

تُعمّق مسائل إعادة البناء فهم دلالات الاجتياز.

- Inorder + Preorder: تحديد الجذر أولاً.
- Inorder + Postorder: تحديد الجذر أخيراً.
- Postorder + Preorder: يتطلب افتراض شجرة ثنائية كاملة.

يتيح استخدام خرائط التجزئة لفهارس المواقع إعادة البناء بزمن $O(n)$.

٦.٣.٢ كشف الأشجار الفرعية

تتحقق اختبارات الشجرة الفرعية مما إذا كانت شجرة ما جزءاً بنيوياً من أخرى.

- العودية المباشرة: مقارنة بنيوية عند كل عقدة.
- القائمة على التسلسل: اختزال المطابقة إلى بحث نصي.

قد تستخدم الأساليب المحسّنة تجزئات متدحرجة أو خوارزميات مطابقة خطية.

٦.٤ تمارين وتطبيق

- إعادة بناء الأشجار من أزواج الاجتياز.
- تنفيذ الاجتيازات العودية والتكرارية.
- كشف الأشجار الفرعية بالأسلوبين البنيوي والمتسلسل.
- مقارنة الأداء بين الأشجار المتوازنة والمائلة.

٦.٥ الخلاصة

- تتيح الأشجار الثنائية تصميم خوارزميات هرمية.
- تحدد استراتيجيات الاجتياز ترتيب الوصول وسلوك الخوارزمية.
- يخدم الأسلوبان العودي والتكراري أدواراً متكاملة.
- يُعدّ التسلسل وإعادة البناء أساسياً للحفظ والمقارنة.
- يُعدّ إتقان اجتياز الأشجار حجر أساس لهياكل البيانات والخوارزميات المتقدمة في Modern C++.

الفصل ٧: أشجار البحث الثنائية والأشجار المُعزَّزة

١.٧ ثوابت أشجار البحث الثنائية والعمليات الأساسية

تنظم أشجار البحث الثنائية (Binary Search Trees -- BSTs) المفاتيح وفق قاعدة ترتيب صارمة تُمكّن من تنفيذ عمليات البحث بكفاءة. وتُعد الأساس المفاهيمي للأشجار المتوازنة، وأشجار الإحصاء الرتبي، والعديد من البُنى المعتمدة على الفواصل.

١.١.٧ ثابت الترتيب في BST

لكل عقدة ذات مفتاح k:

- جميع المفاتيح في الشجرة الفرعية اليسرى أصغر strictly من k.
- جميع المفاتيح في الشجرة الفرعية اليمنى أكبر strictly من k.

وبناءً على ذلك، ينتج الاجتياز Inorder لمخطط BST مفاتيح مرتبة.

٢.١.٧ العمليات الأساسية

تدعم أشجار البحث الثنائية ثلاث عمليات محورية:

- البحث: النزول يساراً أو يميناً بناءً على المقارنة.

- الإدراج: الإدراج عند أول موضع فارغ يحافظ على الترتيب.
- الحذف: معالجة حالات الورقة، والابن الواحد، واللابنين (باستخدام الخلف أو السلف الترتيبي).

في الأشجار المتوازنة تعمل هذه العمليات بزمن O(log n)؛ أما في الأشجار المنحرفة فتتدهور إلى O(n).

٣.١.٧ حالات الأداء الحرجة

- إدراج مفاتيح مرتبة في BST ساذجة ينتج شجرة مائلة.
- قد يصل عمق العودية إلى O(n) في أسوأ الحالات.
- تخصيص العقد المرتبط يؤدي إلى محلية مخبأ ضعيفة.

عملياً، تُستخدم متغيرات ذات توازن ذاتي مثل أشجار AVL وRed-Black لضمان ارتفاع لوغاريتمي.

٢.٧ الأشجار المُعزَّزة

تُوسّع الأشجار المُعزَّزة أشجار البحث الثنائية بتخزين بيانات إضافية في كل عقدة مع الحفاظ على ثابت الترتيب، مما يمكّن من استعلامات متقدمة دون تغيير التعقيد التقاربي.

١.٢.٧ تعزيز حجم الشجرة الفرعية

تعزيز شائع يتمثل في تخزين حجم كل شجرة فرعية.

- تخزّن كل عقدة size = 1 + size(left) + size(right).
- تُحدّث القيم أثناء الإدراج والحذف.

يتيح هذا التعزيز تنفيذ استعلامات الإحصاء الرتبي بكفاءة.

٢.٣.٧ الإحصاء الرتبي

مع أحجام الأشجار الفرعية تصبح استعلامات أساسية فعّالة:

- العنصر رقم k الأصغر: التوجيه يساراً أو يميناً وفق حجم الشجرة الفرعية اليسرى.
- order_of_key(x): عدّ عدد المفاتيح الأصغر strictly من x.

تعمل العمليتان بزمن O(log n) على الأشجار المتوازنة.

٣.٣.٧ استعلامات النطاق

ينتج العد ضمن نطاق مباشرة من استعلامات الرتبة:

- عدد العناصر في [L, R] يساوي order_of_key(R+1) - order_of_key(L).
- يتيح حساب النسب المئوية، والكوانتيلات، ونوافذ الانزلاق.

٤.٣.٧ التعامل مع القيم المكررة

توجد استراتيجيتان قياسيتان:

- تخزين عدّاد تكرار لكل مفتاح وإدراجه ضمن أحجام الأشجار الفرعية.
- إدراج القيم المكررة باستمرار في شجرة فرعية واحدة (غالباً اليمنى).

يحافظ التعزيز القائم على التكرار على صحة استعلامات الرتبة.

٣.٧ أشجار الفواصل

تُعزّز أشجار الفواصل عقد BST ببيانات نطاق لدعم استعلامات التداخل.

- تخزّن كل عقدة فاصلاً [l, r).
- تُعزَّز العقدة بقيمة maxEnd، وهي أكبر نهاية في شجرتها الفرعية.
- تُقصى الأشجار الفرعية أثناء الاستعلام اعتماداً على maxEnd.

يتيح ذلك كشف التداخل بزمن O(log n) على الأشجار المتوازنة.

٤.٧ تمارين وتطبيق

- تنفيذ استعلام العنصر رقم k باستخدام أحجام الأشجار الفرعية.
- توسيع BST لدعم order_of_key.
- بناء شجرة فواصل ودعم استعلامات التداخل.
- مقارنة الأشجار المُعزَّزة بحلول الاجتياز الساذجة.

٥.٧ الخلاصة

- تعتمد BSTs على ثابت ترتيب صارم لتمكين العمليات الفعّالة.
- يرتبط الأداء بشكل حاسم بارتفاع الشجرة.
- تُمكّن التعزيزات مثل حجم الشجرة أو أقصى نهاية من استعلامات قوية دون تغيير التعقيد التقاربي.

- تُظهر الإحصاءات الرتبية وأشجار الفواصل كيف تحوّل إضافات صغيرة BSTs إلى بُنى متخصصة.
- تُعدّ الأشجار المُعزَّزة المتوازنة أساسية للبيانات المرتبة، واستعلامات النطاق، والإحصاءات الآنية في Modern C++.

الفصل ٨: الأشجار ذاتية التوازن: AVL وRed-Black

١.٨ أشجار AVL والدورانات

تُعدّ أشجار AVL أقدم أشجار البحث الثنائية ذاتية التوازن. وتفرض ثابت توازن صارم: فرق الارتفاع بين الشجرتين الفرعيتين اليسرى واليمنى لأي عقدة لا يتجاوز واحداً. يضمن ذلك زمناً O(log n) للبحث والإدراج والحذف.

١.١.٨ عامل التوازن وأنواع الاختلال

لعقدة N يُعرَّف عامل التوازن كما يلي:

$$\text{BF}(N) = \text{height(left)} - \text{height(right)}$$

- متوازن إذا كان BF ∈ {-1, 0, 1}.
- مائل لليسار إذا BF > 1.
- مائل لليمين إذا BF < -1.

تنشأ أربع حالات اختلال:

- LL: دوران يميني أحادي.
- RR: دوران يساري أحادي.
- LR: دوران يساري على الابن الأيسر ثم دوران يميني.
- RL: دوران يميني على الابن الأيمن ثم دوران يساري.

٢.١.٨ تمثيل عقد AVL

تعتمد التنفيذات الحديثة على تخزين الارتفاعات واستخدام std::unique_ptr لتوضيح الملكية.

- تُحدَّث الارتفاعات من الأسفل إلى الأعلى.
- تعيد الدورانات ترتيب المؤشرات فقط مع الحفاظ على ترتيب BST.
- تتطلب المؤشرات الذكية استخدام std::move صراحة أثناء الدوران.

٣.١.٨ الدورانات كآلية أساسية

- تُصلح الدورانات الأحادية الاختلالات الرتيبة (LL، RR).
- تختزل الدورانات المزدوجة إلى دورانين أحاديين (LR، RL).
- بعد كل دوران يجب إعادة حساب الارتفاعات محلياً.

الدورانات عمليات بزمن ثابت وتشكل القلب الميكانيكي لتوازن AVL.

٤.١.٨ الإدراج وإعادة التوازن

يتبع الإدراج منطق BST القياسي، ثم يُتحقق من التوازن أثناء فك العودية.

- كشف الاختلال باستخدام عامل التوازن.

- اختيار حالة الدوران وفق بنية الشجرة الفرعية.
- استعادة التوازن مع الحفاظ على ترتيب .Inorder

٥.١.٨ خصائص أداء AVL

- الارتفاع لوغاريتمي صارم O(log n).
- البحث أسرع من أشجار Red-Black بسبب التوازن الأشد.
- قد تتسبب التحديثات بعدة دورانات.

تُناسب أشجار AVL الأحمال الكثيفة بالبحث.

٢.٨ أشجار Red-Black وحاويات المكتبة القياسية

تُعدّ أشجار Red-Black شكلاً أكثر تساهلاً من التوازن الذاتي، وتستبدل التوازن الصارم بقواعد أبسط لإعادة التوازن، مع ضمان ارتفاع لوغاريتمي وتقليل التغييرات البنيوية.

١.٢.٨ خصائص Red-Black

تحقق شجرة Red-Black خمسة ثوابت:

- كل عقدة إما حمراء أو سوداء.
- الجذر أسود.
- جميع الأوراق (العقد الفارغة/الحارسة) سوداء.
- لا تملك عقدة حمراء ابناً أحمر.
- جميع المسارات من الجذر إلى الأوراق تحوي العدد نفسه من العقد السوداء.

تحد هذه الخصائص الارتفاع إلى 2·log(n) كحد أقصى.

٢.٢.٨ استراتيجية إعادة التوازن

تُستعاد الثوابت عبر:

- إعادة التلوين عند المخالفات المحلية.
- دورانات مصحوبة بإعادة تلوين عند الحاجة لتغيير البنية.

يبدأ الإدراج عادة بعقدة حمراء؛ وقد يُدخل الحذف حالات double-black مؤقتة تنتشر للأعلى.

٣.٢.٨ العلاقة مع std::map وstd::set

تُنفَّذ جميع الحاويات الترابطية المرتبة في المكتبة القياسية باستخدام أشجار Red-Black:

- std::set, std::multiset
- std::map, std::multimap

تضمن هذه الحاويات تعقيد O(log n) بغض النظر عن ترتيب الإدخال، وتُخفي منطق إعادة التوازن عن المستخدم.

٤.٢.٨ AVL مقابل Red-Black

- أشجار AVL أكثر توازناً وتقدم بحثاً أسرع.
- أشجار Red-Black تُجري دورانات أقل عند التحديث.
- تُفضَّل Red-Black للمكتبات العامة.

٣.٨ تمارين وتقييم عملي

- تنفيذ شجرة AVL كاملة مع الإدراج والحذف.
- التحقق من عوامل التوازن بعد إدراجات عشوائية وعدائية.
- مقارنة الأداء مع std::set لبيانات كبيرة.
- تعزيز شجرة AVL بأحجام الأشجار الفرعية للإحصاء الرتبي.

٤.٨ الخلاصة

- تحافظ الأشجار ذاتية التوازن على ارتفاع لوغاريتمي تلقائياً.
- تفرض AVL توازناً صارماً عبر الدورانات وتخزين الارتفاعات.
- تفرض Red-Black توازناً متساهلاً عبر قواعد التلوين.
- تُعدّ الدورانات آليات بنيوية مشتركة بين العائلتين.
- يفسّر فهم هذه الأشجار سلوك std::map وstd::set في الأنظمة الحقيقية.

الفصل ٩: أشجار B وبُنى البيانات للذاكرة الخارجية

١.٩ تصميم أشجار B للتخزين الخارجي

أشجار B-Trees هي أشجار بحث متعددة الفروع صُممت لأنظمة الذاكرة الخارجية مثل الأقراص الصلبة وSSDs. وهدفها الجوهري تقليل عمليات الإدخال/الإخراج عبر زيادة التفرّع، مما يخفض ارتفاع الشجرة وعدد الوصولات إلى القرص لكل عملية.

١.١.٩ الثوابت البنيوية

تحقق شجرة B بدرجة دنيا t ما يلي:

- تخزّن كل عقدة بين t-1 و2t-1 مفاتيح (عدا الجذر).
- للعقد الداخلية (#keys + 1) أبناء.
- تظهر جميع الأوراق على العمق نفسه.
- تُخزَّن المفاتيح داخل العقدة بترتيب تصاعدي.

تضمن هذه الثوابت ارتفاعاً لوغاريتمياً مقاساً بكتل القرص.

٢.١.٩ تخطيط العقدة ومحاذاة الكتل

لتحقيق كفاءة إدخال/إخراج عالية، تُصمَّم العقدة لتشغل كتلة قرص واحدة تماماً.

- تتيح المصفوفات ثابتة الحجم قراءات/كتابات مباشرة للكتل.
- يقلل التفرّع الكبير ارتفاع الشجرة بشكل ملحوظ.
- تُخزَّن مراجع الأبناء كإزاحات قرصية لا كمؤشرات.

٣.١.٩ التمثيل داخل الذاكرة مقابل على القرص

- داخل الذاكرة: يمكن استخدام std::unique_ptr لسلامة الملكية.
- على القرص: تُعرَّف الأبناء بمعرّفات صفحات أو إزاحات بايتية.
- يحوّل التسلسل/فك التسلسل العقد إلى كتل بايتية خام.

يضمن التحكم الدقيق في المحاذاة والحشو حجماً متوقعاً للعقدة وتخزيناً قابلاً للنقل.

٢.٩ الخصائص التشغيلية

١.٢.٩ البحث

- تنفيذ بحث ثنائي داخل العقدة.
- النزول إلى كتلة الابن المناسبة.
- يكلف كل مستوى قراءة قرص واحدة.

٢.٢.٩ الإدراج

- إدراج المفاتيح في الأوراق.
- شطر العقدة المتجاوزة للسعة مع ترقية المفتاح الوسيط.
- قد تنتشر عمليات الشطر للأعلى وتُنشئ جذراً جديداً.

٣.٢.٩ استعلامات النطاق

- اجتياز العقد بترتيب المفاتيح.
- تحميل الكتل المتقاطعة مع النطاق المطلوب فقط.
- أكثر كفاءة بكثير من المسح الكامل للملفات.

٣.٩ اعتبارات الأداء

- يُضبط حجم العقدة ليتوافق مع حجم الكتلة الفيزيائية (عادة 4--8 كيلوبايت).
- يهيمن زمن الوصول للقرص على زمن التنفيذ؛ وكلفة المعالج ثانوية.
- يحسّن التخزين المؤقت للعقد الساخنة معدل النقل بشكل كبير.
- تُبسّط التخطيطات ثابتة الحجم الاستمرارية والتعافي.

٤.٩ التطبيقات

- فهارس قواعد البيانات (أساسية وثانوية).
- هياكل أدلة أنظمة الملفات.

- مخازن القيم-المفاتيح على القرص.
- مجموعات بيانات مرتبة ضخمة تتجاوز سعة الذاكرة.

٥.٩ تمرين: مكتبة B-Tree مصغّرة

١.٥.٩ الأهداف

- تنفيذ شجرة B عامة داخل الذاكرة.
- دعم الإدراج والبحث.
- الحفاظ على جميع ثوابت .B-Tree
- التحقق من الصحة باختبارات وحدة.

٢.٥.٩ الواجهة الأساسية

- التعميم بنوع المفتاح والقيمة والدرجة الدنيا.
- استخدام مصفوفات ثابتة الحجم للمفاتيح والأبناء.
- تغليف منطق الشطر والترقية.

٣.٥.٩ أهداف الاختبار

- التحقق من الاجتياز المرتب.
- تأكيد النمو اللوغاريتمي للارتفاع.
- اختبارات ضغط بإدخالات عشوائية كبيرة.

٤.٥.٩ امتدادات اختيارية

- استبدال المؤشرات بإزاحات قرصية.
- إضافة دعم استعلامات النطاق.
- إدخال التخزين المؤقت للعقد.
- المقارنة مع std::map وقواعد بيانات مدمجة.

٦.٩ الخلاصة

- تُحسَّن أشجار B للذاكرة الخارجية لا لمخبأ المعالج.
- يُعدّ تخطيط العقدة ومحاذاة الكتل عاملين حاسمين للأداء.
- يقلل التفرّع الكبير عمليات الإدخال/الإخراج وارتفاع الشجرة.
- تظل أشجار B أساس قواعد البيانات وأنظمة الملفات.
- تكشف النماذج الأولية البسيطة مبادئ بُنى البيانات الموجّهة للقرص.

الباب ٤

الرسوم البيانية (مُنفَّذة باستخدام C++)

الفصل ١٠: تمثيلات الرسوم البيانية في C++

١.١٠ التمثيلات الأساسية للرسم البياني

تعتمد كفاءة خوارزميات الرسوم البيانية اعتماداً حاسماً على كيفية تمثيل الرسم في الذاكرة. يؤثر اختيار التمثيل في استهلاك المساحة، وسلوك الذاكرة المخبئية، وسرعة الاجتياز، والتعقيد الخوارزمي. توفر Modern C++ تجريدات مرنة، إلا أن التصميم الموجّه للأداء يظل أمراً أساسياً.

١.١.١٠ قائمة المجاورات

تُعد قائمة المجاورات أكثر تمثيل شيوعاً للرسوم البيانية المتناثرة.

- تخزّن كل قمة قائمة بجيرانها الخارجين.
- تعقيد المساحة O(V + E).
- اجتياز الجيران فعّال وطبيعي.

نقاط القوة:

- فعّالة للرسوم المتناثرة.
- مثالية لخوارزميات BFS وDFS وأقصر المسارات.

القيود:

- فحص وجود حافة يتطلب زمناً خطياً في درجة القمة.
- الذاكرة غير متجاورة تماماً، مما يقلل من محلية المخبأ.

٢.١.١٠ قائمة الحواف

تخزّن قائمة الحواف جميع الحواف كسجلات مستقلة.

- تعقيد المساحة O(E).
- بنية بسيطة ومضغوطة.

نقاط القوة:

- أقل كلفة تخزين.
- مناسبة للخوارزميات المتمحورة حول الحواف.

القيود:

- تتطلب استعلامات الجيران مسح جميع الحواف.
- غير فعّالة للخوارزميات كثيفة الاجتياز.

تُستخدم قوائم الحواف عادة في خوارزميات Kruskal وBellman--Ford.

٣.١.١٠ الصف المضغوط المتناثر (CSR)

يُعد تمثيل CSR تمثيلاً عالي الأداء مُحسّناً للرسوم الكبيرة المتناثرة.

- تُخزَّن جميع معلومات المجاورات في مصفوفات متجاورة.

- يستخدم row_ptr لتحديد حدود المجاورات.
- تعقيد المساحة O(V + E).

نقاط القوة:

- محلية مخبأ ممتازة.
- اجتياز تسلسلي سريع.
- مثالي للرسوم الساكنة.

القيود:

- مكلف عند التعديل الديناميكي.
- بناء التمثيل أكثر تعقيداً.

يُعد CSR الصيغة المفضلة لتحليلات الأداء العالي ومحركات معالجة الرسوم والأعباء العددية.

٢.١٠ الرسوم الموزونة والموجّهة

١.٢.١٠ الرسوم الموزونة

تربط الرسوم الموزونة كلفة عددية بكل حافة.

- ضرورية لخوارزميات أقصر مسار وشجرة الامتداد الدنيا.
- تُنفَّذ عادة باستخدام قوائم مجاورات من أزواج (neighbor, weight).

يمكن لمصفوفات المجاورات تخزين الأوزان أيضاً، لكنها تتطلب مساحة $O(V^2)$، ولا تناسب إلا الرسوم الكثيفة.

٢.٢.١٠ الرسوم الموجّهة مقابل غير الموجّهة

- الرسوم الموجّهة: للحواف اتجاه ($u \to v$).
- الرسوم غير الموجّهة: الحواف ثنائية الاتجاه وتُخزَّن في قائمتي المجاورات.

يؤثر الاختيار في دلالات الخوارزميات وقواعد الوصول، لا في مبادئ التخزين الأساسية.

٣.١٠ تصميم الرسم البياني الموجّه للذاكرة

تتطلب الرسوم واسعة النطاق اهتماماً خاصاً بتخطيط الذاكرة.

١.٣.١٠ التخزين المتجاور

- تفضيل std::vector على الحاويات كثيفة المؤشرات.
- تحسين محلية المخبأ وكفاءة الجلب المسبق.

٢.٣.١٠ CSR للأداء

- يحاذي بيانات المجاورات تسلسلياً في الذاكرة.
- يتيح اجتيازاً سريعاً لـ BFS وDFS وخوارزميات أقصر مسار.
- يقلل تتبع المؤشرات.

٣.٣.١٠ هياكل حواف مُحزَّمة

- تخزين الحواف في مصفوفات مسطّحة من بُنى مدمجة.
- مثالي للخوارزميات التي تعالج الحواف تسلسلياً.

٤.٣.١٠ التخصيص المخصص

- تقلل مجمّعات الذاكرة التجزئة في الرسوم الضخمة.
- مفيدة عندما تصل أعداد القمم والحواف إلى الملايين.

٤.١٠ مفاضلات الأداء

- توازن قوائم المجاورات بين المرونة والكفاءة.
- تفضّل مصفوفات المجاورات فحص الحواف بزمن ثابت مقابل كلفة ذاكرة عالية.
- يحقق CSR أقصى سرعة اجتياز للرسوم الساكنة الكبيرة.

لا يوجد تمثيل واحد يتفوق في جميع السيناريوهات؛ فالاختيار الأمثل يعتمد على كثافة الرسم وقابليته للتغيير وأنماط الوصول الخوارزمية.

٥.١٠ الخلاصة

- يؤثر تمثيل الرسم مباشرة في أداء الخوارزميات.
- تفضّل الرسوم المتناثرة قوائم المجاورات أو CSR.
- تناسب قوائم الحواف المعالجة المتمحورة حول الحواف.
- يوفّر CSR أعلى كفاءة مخبأ للرسوم الساكنة.
- تشجّع Modern C++ على فصل التمثيل عن الخوارزميات للوضوح وإعادة الاستخدام والأداء.

الفصل ١١: الاجتياز والبحث

١.١١ البحث بعمق (DFS) والبحث بعرض الشجرة (BFS)

تستكشف خوارزميات اجتياز الرسوم القمم والحواف بشكل منظّم. يُعدّ DFS وBFS تقنيتين أساسيتين بزمن خطي O(V + E) عند استخدام قوائم المجاورات. تتيح Modern C++ تنفيذات نظيفة وعامة باستخدام المكررات وحاويات STL وواجهات تحترم الثبات.

١.١.١١ البحث بعمق (DFS)

يستكشف DFS الفروع إلى أقصى عمق قبل الرجوع.

DFS العودي

- يستخدم مكدس الاستدعاءات لتتبع المسار الحالي.
- مختصر ومعبر، لكنه محدود بعمق العودية.
- مناسب للأحجام المتوسطة.

DFS التكراري

- يستخدم مكدساً صريحاً لتجنب حدود العودية.
- أكثر متانة للرسوم الكبيرة أو العدائية.
- يمكن مطابقة ترتيب DFS العودي باستخدام مكررات عكسية.

خلاصة مهمة: DFS العودي والتكراري متكافئان خوارزمياً؛ ويُحدَّد الاختيار وفق سلامة المكدس والتحكم في التنفيذ.

٢.١.١١ البحث بعرض الشجرة (BFS)

يستكشف BFS الرسم مستوىً بعد مستوى باستخدام طابور.

- يضمن أقصر المسارات في الرسوم غير الموزونة.
- يكتشف الطبقات والمسافات من المصدر.
- قد يكبر حجم الطابور مع تفرّع عالٍ.

يُعد BFS أساسياً لاكتشاف أقصر المسارات، والتحقق من الثنائية، والاستكشاف بالمستويات.

٣.١.١١ تصميم الاجتياز القائم على المكررات

تُفضّل Modern C++ واجهات اجتياز تكون:

- قائمة على المكررات: تعمل مع أي حاوية مجاورات.
- تحترم الثبات: تمنع التعديل غير المقصود أثناء الاجتياز.
- عامة: قابلة لإعادة الاستخدام عبر تمثيلات متعددة.

يفصل التصميم القائم على القوالب الخوارزميات عن هياكل البيانات، مما يحسّن الصيانة وقابلية التوسعة.

٤.١.١١ مقارنة الخوارزميات

- DFS (عودي): بسيط ومعبر، محدود بالمكدس.
- DFS (تكراري): تحكم صريح وآمن للمكدس.
- :BFS أقصر مسارات في الرسوم غير الموزونة واكتشاف الطبقات.

تعمل جميعها بزمن O(V + E).

٢.١١ تطبيقات اجتياز الرسوم

تُعد خوارزميات الاجتياز لبنات بناء لتحليلات أعلى مستوى.

١.٢.١١ المكوّنات المتصلة

- استخدام DFS أو BFS من كل قمة غير مُزارة.
- يحدد كل اجتياز مكوّناً متصلاً واحداً.
- حل بزمن خطي للرسوم غير الموجّهة.

تشمل التطبيقات تحليل الشبكات والتجميع وكشف العزل.

٢.٢.١١ المكوّنات المتصلة بقوة (الرسوم الموجّهة)

- تحديد مجموعات القمم القابلة للوصول المتبادل.
- خوارزميات كلاسيكية: Tarjan وKosaraju.
- التعقيد: O(V + E).

تُعد SCCs أساسية في تحليل المترجمات ورسوم الاعتماد والأنظمة المعيارية.

١١.٣.٢ كشف الدورات

الرسوم غير الموجّهة

- DFS مع تتبع الأب.
- زيارة قمة مُزارة ليست الأب تشير إلى دورة.

الرسوم الموجّهة

- DFS مع التلوين (أبيض، رمادي، أسود).
- حافة خلفية إلى عقدة رمادية تدل على دورة.

يدعم كشف الدورات اكتشاف حالات الإقفال المتبادل والتحقق من الاعتماديات.

١١.٣.٤ الترتيب الطوبولوجي

لا يوجد ترتيب طوبولوجي إلا للرسوم الموجّهة الخالية من الدورات .(DAGs)

- قائم على :DFS دفع القمم بعد استكشاف التوابع.
- قائم على BFS (خوارزمية Kahn): إزالة القمم ذات الدرجة الداخلة الصفرية.
- كلاهما بزمن O(V + E).

الترتيب الطوبولوجي أساسي للجدولة وأنظمة البناء وحل الاعتماديات.

١١.٣ أفضل الممارسات في Modern C++

- فصل بنية الرسم عن خوارزميات الاجتياز.

- تفضيل الاجتياز القائم على المكررات والنطاقات.
- استخدام الخوارزميات التكرارية للرسوم الكبيرة.
- إدارة مصفوفات مساعدة (visited, color, parent) صراحة.
- تفضيل القوالب العامة لدعم حاويات مجاورات متعددة.

٤.١١ الخلاصة

- يُعد DFS وBFS بدائيتين خطيتين للاجتياز.
- DFS العودي والتكراري متكافئان نظرياً.
- يضمن BFS أقصر المسارات في الرسوم غير الموزونة.
- تمكّن خوارزميات الاجتياز من كشف الاتصال والدورات وSCCs والترتيب الطوبولوجي.
- يركّز تصميم Modern C++ على واجهات عامة وآمنة وقائمة على المكررات.

الفصل ١٢: أقصر المسارات

١.١٢ خوارزمية ديكسترا

تحسب خوارزمية ديكسترا أقصر المسارات من مصدر واحد في الرسوم ذات الأوزان غير السالبة. تعتمد صحتها على الإرخاء الجشع، وتعتمد كفاءتها على إدارة طوابير الأولوية.

١.١.١٢ الفكرة الأساسية

- تهيئة جميع المسافات بـ INF والمصدر بـ 0.
- استخراج القمة ذات أصغر مسافة مؤقتة.
- إرخاء الحواف الخارجة.

التعقيد الزمني:

- اختيار قائم على مصفوفة: $O(V^2)$ (للرسوم الكثيفة).
- كومة صغرى (طابور أولوية): $O((V+E)\log V)$ (القياسي).

٢.١.١٢ تنفيذ Modern C++ (كومة صغرى كسولة)

```cpp
class Graph {
    int V;
    std::vector<std::vector<std::pair<int,int>>> adj;
public:
    Graph(int V) : V(V), adj(V) {}

    void addEdge(int u, int v, int w) {
        adj[u].push_back({v, w});
    }

    std::vector<int> dijkstra(int src) const {
        const int INF = std::numeric_limits<int>::max();
        std::vector<int> dist(V, INF);
        dist[src] = 0;

        using Node = std::pair<int,int>; // {dist, vertex}
        std::priority_queue<Node, std::vector<Node>, std::greater<Node>> pq;
        pq.push({0, src});

        while (!pq.empty()) {
            auto [d, u] = pq.top(); pq.pop();
            if (d > dist[u]) continue;

            for (auto &[v, w] : adj[u]) {
                if (dist[u] + w < dist[v]) {
                    dist[v] = dist[u] + w;
                    pq.push({dist[v], v});
                }
```

```
            }
        }
        return dist;
    }
};
```

ممارسة أساسية:

- السماح بإدخالات مكررة في الكومة وتجاوز القديمة (إستراتيجية كسولة).
- تفضيل البساطة على تنفيذات decrease-key.

٣.١.١٢ ملخص الأداء

- مثالية للأوزان غير السالبة.
- صديقة للمخبأ مع قوائم المجاورات.
- غير صالحة للأوزان السالبة.

٢.١٢ SPFAg Bellman--Ford

١.٢.١٢ Bellman--Ford

تتعامل Bellman--Ford مع الأوزان السالبة وتكشف الدورات السالبة.

- إرخاء جميع الحواف V-1 مرة.
- تمريرة إضافية تكشف الدورات السالبة.
- التعقيد: $O(V \times E)$.

```cpp
struct Edge { int u, v, w; };

std::vector<int> bellmanFord(int V, const std::vector<Edge>& edges, int src)
↪  {
    const int INF = std::numeric_limits<int>::max();
    std::vector<int> dist(V, INF);
    dist[src] = 0;

    for (int i = 0; i < V - 1; ++i)
        for (auto &e : edges)
            if (dist[e.u] != INF && dist[e.u] + e.w < dist[e.v])
                dist[e.v] = dist[e.u] + e.w;

    for (auto &e : edges)
        if (dist[e.u] != INF && dist[e.u] + e.w < dist[e.v])
            throw std::runtime_error("Negative cycle");

    return dist;
}
```

٢.٢.١٢ ملاحظات حول SPFA

- تحسين قائم على الطابور لـ Bellman--Ford.
- أسرع في كثير من الرسوم المتناثرة مع نفس أسوأ حالة.
- يتطلب إدارة دقيقة للطابور.

٣.١٢ خوارزمية البحث A*

تحسّن A* ديكسترا بتوجيه البحث باستخدام دالة تخمين.

- g(n): الكلفة من البداية.
- h(n): كلفة تقديرية إلى الهدف.
- f(n) = g(n) + h(n).

١.٣.١٢ متطلبات دالة التخمين

- مقبولة: لا تبالغ في التقدير.
- متسقة: تضمن المثالية والكفاءة.

٢.٣.١٢ نمط A* مضغوط في C++

```
struct Node {
    int v;
    double g, f;
    bool operator>(const Node& o) const { return f > o.f; }
};
```

- استخدام std::priority_queue مع مقارنة مخصصة.
- تتبع السوابق لإعادة بناء المسار.
- يجب أن تكون دالة التخمين سريعة ومستقرة.

٤.١٢ أقصر المسارات متعددة المصادر

١.٤.١٢ المفهوم

- تهيئة عدة مصادر بمسافة 0.
- دفع جميع المصادر إلى طابور BFS أو ديكسترا.
- حساب أقل مسافة من أي مصدر.

٢.٤.١٢ BFS متعدد المصادر (غير موزون)

```
std::vector<int> multiSourceBFS(
    int V,
    const std::vector<std::vector<int>>& adj,
    const std::vector<int>& sources)
{
    const int INF = std::numeric_limits<int>::max();
    std::vector<int> dist(V, INF);
    std::queue<int> q;

    for (int s : sources) {
        dist[s] = 0;
        q.push(s);
    }

    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int v : adj[u])
            if (dist[v] == INF) {
```

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

١٢.٥ قوالب إعادة بناء المسار

١٢.٥.١ إعادة بناء عامة

```
template <typename V>
std::vector<V> reconstructPath(V target, const std::vector<V>& parent) {
    std::vector<V> path;
    for (V v = target; v != V(-1); v = parent[v])
        path.push_back(v);
    std::reverse(path.begin(), path.end());
    return path;
}
```

- قابلة للإعادة الاستخدام عبر BFS وديكسترا وA*.
- تعمل مع مسائل المصدر الواحد أو المتعددة.

١٢.٦ أفضل الممارسات

- اختيار الخوارزمية وفق خصائص الأوزان.

- تفضيل طوابير الأولوية الكسولة للبساطة.
- الحذر من فيض الأعداد الصحيحة.
- فصل تخزين الرسم عن الخوارزميات.
- الحجز المسبق للحاويات في الرسوم الكبيرة.
- استخدام القوالب لإعادة بناء المسارات القابلة لإعادة الاستخدام.

الفصل ١٣: شجرة الامتداد الدنيا واتحاد المجموعات

١.١٣ خوارزمية كروسكال مع اتحاد المجموعات

تبني خوارزمية Kruskal شجرة امتداد دنيا (MST) باختيار أخف حافة لا تُكوّن دورة. تعتمد كفاءتها العملية على بنية اتحاد المجموعات (DSU).

١.١.١٣ جوهر الخوارزمية

- فرز جميع الحواف تصاعدياً حسب الوزن.
- تهيئة DSU بحيث تكون كل قمة في مجموعة مستقلة.
- إضافة الحواف التي تصل مجموعتين مختلفتين.
- التوقف بعد اختيار V-1 حافة.

التعقيد:

- فرز الحواف: O(E log E).

- عمليات DSU: $O(\alpha(V))$ مُعدَّلة.
- الإجمالي: $O(E \log E)$ عملياً.

٢.١.١٣ تصميم DSU فعّال

يجمع DSU الحديث تحسينين:

- ضغط المسار: تسطيح الأشجار أثناء find.
- الاتحاد بالرتبة: ربط الأشجار الأصغر بالأكبر.

```
template <typename T>
class DSU {
    std::vector<T> parent, rank;
public:
    DSU(T n) : parent(n), rank(n, 0) {
        for (T i = 0; i < n; ++i) parent[i] = i;
    }

    T find(T x) {
        if (parent[x] != x)
            parent[x] = find(parent[x]);
        return parent[x];
    }

    bool unite(T a, T b) {
        a = find(a); b = find(b);
        if (a == b) return false;
        if (rank[a] < rank[b]) std::swap(a, b);
```

```
        parent[b] = a;
        if (rank[a] == rank[b]) ++rank[a];
        return true;
    }
};
```

٣.١.١٣ تنفيذ كروسكال

```
template <typename T>
struct Edge {
    T u, v;
    int w;
    bool operator<(const Edge& e) const { return w < e.w; }
};

template <typename T>
std::vector<Edge<T>> kruskal(T V, std::vector<Edge<T>>& edges) {
    std::sort(edges.begin(), edges.end());
    DSU<T> dsu(V);
    std::vector<Edge<T>> mst;

    for (auto &e : edges) {
        if (dsu.unite(e.u, e.v))
            mst.push_back(e);
        if (mst.size() == V - 1) break;
    }
    return mst;
}
```

ملاحظة: تتفوق كروسكال عندما تكون الحواف مخزنة بشكل مدمج أو مفروزة مسبقاً.

٢.١٣ خوارزمية بريم

تنمّي خوارزمية Prim شجرة الامتداد بدءاً من قمة، وتوسّعها بأرخص حافة متاحة.

١.٢.١٣ نسخة الكومة الثنائية

- تستخدم طابور أولوية مُفهرساً بالوزن.
- التعقيد: $O(E \log V)$.
- مفضّلة لمعظم الرسوم العملية.

```cpp
struct Edge { int to, w; };

std::vector<int> prim(int V, const std::vector<std::vector<Edge>>& adj) {
    const int INF = std::numeric_limits<int>::max();
    std::vector<int> key(V, INF), parent(V, -1);
    std::vector<bool> inMST(V, false);

    using Node = std::pair<int,int>; // {key, vertex}
    std::priority_queue<Node, std::vector<Node>, std::greater<Node>> pq;

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

    while (!pq.empty()) {
        auto [_, u] = pq.top(); pq.pop();
        if (inMST[u]) continue;
        inMST[u] = true;
```

```
        for (auto &e : adj[u]) {
            if (!inMST[e.to] && e.w < key[e.to]) {
                key[e.to] = e.w;
                parent[e.to] = u;
                pq.push({key[e.to], e.to});
            }
        }
    }
    return parent;
}
```

٢.٢.١٣ الكومة الثنائية مقابل فيبوناتشي

- الكومة الثنائية: بسيطة وصديقة للمخبأ وسريعة عملياً.
- كومة فيبوناتشي: حدود تقاربية أفضل لكنها معقدة ونادراً ما تُستخدم إنتاجياً.

توصية: استخدم الكومة الثنائية في C++ إلا لأغراض نظرية بحتة.

٣.١٣ الاتصال الديناميكي ومتغيرات MST

١.٣.١٣ اتحاد المجموعات للاتصال

يدعم DSU استعلامات الاتصال التزايدية:

```
template <typename T>
class Connectivity {
    DSU<T> dsu;
public:
```

```
    Connectivity(T n) : dsu(n) {}
    void addEdge(T u, T v) { dsu.unite(u, v); }
    bool connected(T u, T v) { return dsu.find(u) == dsu.find(v); }
};
```

٢.٣.١٣ متغيرات MST

- شجرة امتداد عظمى: عكس ترتيب الأوزان.
- MST مقيّدة: معالجة الحواف المفروضة أو الممنوعة مسبقاً.
- الرسوم متعددة الحواف: تتعامل كروسكال معها طبيعياً.

٤.١٣ أفضل الممارسات

- تفضيل كروسكال للرسوم المتناثرة والمتمحورة حول الحواف.
- تفضيل بريم للرسوم الكثيفة مع قوائم المجاورات.
- الحجز المسبق لبُنى DSU والمجاورات.
- استخدام تحديثات كسولة لطوابير الأولوية.
- فصل خوارزميات MST عن تخزين الرسم.

٥.١٣ الخلاصة

- يحقق كروسكال + DSU أداءً قريباً من الأمثل لـ MST.
- يُعد ضغط المسار والاتحاد بالرتبة عنصرين حاسمين.

- تُكمل بريم كروسكال بنمو متمحور حول القمم.
- يمتد DSU طبيعياً لمسائل الاتصال الديناميكي.
- تمكّن Modern C++ من تنفيذات نظيفة وعامة وعالية الأداء لـ MST.

الفصل ١٤: تدفّق الشبكات والرسوم المتقدمة

١.١٤ خوارزميات أقصى تدفّق

تحسب خوارزميات تدفّق الشبكات أقصى تدفّق ممكن من مصدر إلى مصب في رسم موجّه ذي سعات. تختلف الخوارزميات في اختيار المسارات المُعزِّزة وكفاءة معالجة الرسوم المتبقية.

١.١.١٤ طريقة Ford--Fulkerson

- البحث المتكرر عن مسار مُعزِّز في الرسم المتبقي.
- زيادة التدفّق حتى لا يوجد مسار من المصدر إلى المصب.

الخصائص:

- التعقيد: $O(E \times F)$ (يعتمد على قيمة التدفّق).
- بسيطة وبديهية.
- قد تكون غير فعّالة أو غير منتهية مع السعات غير العقلانية.

الاستخدام: لأغراض تعليمية وشبكات صغيرة بسعات صحيحة.

٢.١.١٤ خوارزمية Edmonds--Karp

هي تخصيص حتمي لـ Ford--Fulkerson.

- تستخدم BFS لاختيار أقصر مسار مُعزِّز (بعدد الحواف).
- تضمن زمناً متعدد الحدود.

التعقيد:

- $O(V \times E^2)$.

المفاضلة:

- أكثر قابلية للتنبؤ من نسخة DFS.
- أبطأ للرسوم الكبيرة أو الكثيفة.

٣.١.١٤ خوارزمية Dinic

تحسّن Dinic الحساب باستخدام رسوم طبقية.

- بناء رسم طبقي باستخدام BFS.
- إرسال تدفّقات حاجزة باستخدام DFS.
- التكرار حتى انعدام المسارات المُعزِّزة.

التعقيد:

- الرسوم العامة: $O(E \times V^2)$.
- سعات وحدية: $O(E\sqrt{V})$.

رؤية عملية:

- الأسرع لمعظم مسائل التدفّق الواقعية.
- أعقد قليلاً لكنه عالي القابلية للتوسّع.

٤.١.١٤ مقارنة الخوارزميات

- Ford--Fulkerson: الأبسط والأقل موثوقية للمدخلات الكبيرة.
- Edmonds--Karp: صحيح ومعتدل الأداء.
- Dinic: أفضل توازن بين الصحة والأداء.

٢.١٤ خوارزميات المطابقة

١.٢.١٤ المطابقة الثنائية وHopcroft--Karp

تحسب Hopcroft--Karp المطابقة العظمى في الرسوم الثنائية.

- تستخدم DFS + BFS بطبقات.
- تعثر على عدة مسارات مُعزِّزة في كل مرحلة.

التعقيد:

- $O(\sqrt{V} \times E)$.

التطبيقات:

- مسائل الإسناد.
- تخصيص الموارد.
- الجدولة وأنظمة الإقران.

ميزة أساسية: أسرع بكثير من المطابقة الساذجة القائمة على DFS.

٣.١٤ التدفّق مع الكلفة: Max-Flow Min-Cost

١.٣.١٤ تعريف المشكلة

تمدّد Max-Flow Min-Cost (MCMF) أقصى تدفّق بإسناد كلفة لكل حافة.

- لكل حافة سعة وكلفة.
- الهدف: تعظيم التدفّق مع تقليل الكلفة الكلية.

٢.٣.١٤ الأساليب الأساسية

- أقصر مسار متتابع:
  - إيجاد أرخص مسار مُعزِّز تكرارياً.
  - استخدام Bellman--Ford أو ديكسترا مع الجهود.
- إلغاء الدورات:
  - إزالة الدورات سالبة الكلفة في الرسم المتبقي.
  - ذات اهتمام نظري بالأساس.

المفاضلات:

- :Bellman--Ford بسيط لكنه بطيء.
- ديكسترا + الجهود: فعّال للكلف غير السالبة.

٤.١٤ إرشادات الأداء

- استخدام Dinic لمسائل أقصى التدفّق واسعة النطاق.
- استخدام Hopcroft--Karp للمطابقة الثنائية.
- استخدام MCMF فقط عند الحاجة للكلفة.
- تفضيل قوائم المجاورات مع حواف عكسية صريحة.
- الحجز المسبق للرسوم المتبقية لتجنب كلفة وقت التشغيل.

٥.١٤ التطبيقات

- النقل واللوجستيات.
- الاتصالات وتخصيص السعة.
- إسناد الوظائف وتوزيع المشاريع.
- الجدولة مع القيود.

٦.١٤ تمارين وامتدادات

- مقارنة Edmonds--Karp وDinic على الرسوم الكثيفة والمتناثرة.
- تنفيذ المطابقة الثنائية باستخدام Hopcroft--Karp والتدفّق.
- حل مسائل الإسناد باستخدام Max-Flow. Min-Cost
- استكشاف المطابقة الموزونة عبر تحويلات التدفّق.
- قياس الذاكرة والزمن مع زيادة حجم الرسم.

٧.١٤ الخلاصة

- يُعد تدفّق الشبكات تجريداً موحِّداً لمسائل التحسين.
- يؤثر اختيار الخوارزمية بقوة في القابلية للتوسّع.
- تتصدر Dinic وHopcroft--Karp التنفيذات العملية.
- يوازن Max-Flow Min-Cost بين الأداء وقوة التعبير.
- تمكّن Modern C++ من تنفيذات آمنة ومعيارية وعالية الأداء لخوارزميات التدفّق.

الباب ٥

نماذج التصميم والتقنيات الخوارزمية

الفصل ١٥: قسِّم تسُد (Divide and Conquer)

١.١٥ أنماط الفرز والاستدعاء الذاتي في Modern C++

يُعد مبدأ «قسِّم تسُد» نموذجاً خوارزمياً أساسياً يقوم على تفكيك المشكلة إلى مسائل فرعية مستقلة أصغر، وحلّها استدعائياً، ثم دمج نتائجها. يوفّر هذا النموذج ضمانات نظرية قوية وكفاءة عملية عالية، لا سيما عند اقترانه بتجريدات Modern C++ وآليات التنفيذ المتوازي.

١.١.١٥ فرز الدمج (Merge Sort)

يمثّل فرز الدمج مثالاً كلاسيكياً على «قسِّم تسُد» المتوازن ذي الأداء المتوقَّع.

- سلوك حتمي بزمن مضمون $O(n \log n)$.
- ترتيب مستقر للعناصر المتساوية.
- يتطلب ذاكرة مساعدة متناسبة مع حجم الإدخال.

التعقيد:

- الزمن: $O(n \log n)$ (في جميع الحالات)

- المساحة: $O(n)$

رؤية عملية:

- يُفضَّل عند الحاجة إلى الاستقرار.
- يشكّل الأساس المفاهيمي لـ std::stable_sort.

٢.١.١٥ الفرز السريع (Quicksort)

يُظهر Quicksort «قسِّم تسُد» المُحسَّن للتنفيذ الموضعي .(in-place)

- تقسيم البيانات حول محور .(pivot)
- فرز المقاطع الفرعية استدعائياً.
- لا توجد مرحلة دمج صريحة.

التعقيد:

- المتوسط: $O(n \log n)$
- أسوأ حالة: $O(n^2)$ (اختيار محور سيئ)
- المساحة: $O(\log n)$ لمكدس الاستدعاء

رؤية عملية:

- محلية مخبأ ممتازة.
- غير مستقر بطبيعته.
- يُستخدم نواةً لـ std::sort عبر Quicksort introsort) مع احتياط .(Heapsort

٣.١.١٥ استراتيجيات اختيار المحور

- محور ساذج: بسيط لكنه هش.
- محور عشوائي: توازن احتمالي.
- متوسط الثلاثة: متين وشائع عملياً.

يُعد الاختيار الجيد للمحور أساسياً للحفاظ على عمق استدعاء لوغاريتمي.

٢.١٥ أنماط الاستدعاء الذاتي والتحويلات

١.٢.١٥ الاستدعاء الذاتي القياسي

تعكس الاستدعاءات الذاتية بنية تفكيك المشكلة بشكل طبيعي، لكنها قد تؤدي إلى مكدسات عميقة مع مُدخلات كبيرة.

٢.٢.١٥ إزالة الاستدعاء الذاتي الذّيلي

يسمح الاستدعاء الذاتي الذّيلي بإعادة استخدام إطارات المكدس.

- تقليل عمق المكدس الأقصى.
- ضروري لتنفيذات Quicksort المتينة.
- يمكن تحويله يدوياً إلى تكرار.

٣.٢.١٥ التحويلات التكرارية

يمكن استبدال الاستدعاء الذاتي بمكدسات صريحة:

- تحكّم كامل في استخدام الذاكرة.
- مناسب للأنظمة ذات المكدس المحدود.
- يتيح جدولة دقيقة وأدوات قياس.

٣.١٥ ملخص مقارن

- Sort: Merge مستقر، متوقّع، كثيف الذاكرة.
- Quicksort: موضعي، سريع غالباً، حساس لجودة المحور.
- كلاهما يُظهر أشجار الاستدعاء القياسية لـ «قسِّم تسُد».

٤.١٥ قسِّم تسُد المتوازي

تتسم خوارزميات «قسِّم تسُد» بطبيعتها المتوازية بسبب استقلال المسائل الفرعية.

١.٤.١٥ سياسات التنفيذ

توفّر Modern C++ توازياً معيارياً عبر <execution>:

- seq: تنفيذ تسلسلي.
- par: تنفيذ متوازي.
- par_unseq: تنفيذ متوازي ومُتجه.

تمكّن هذه السياسات من قابلية التوسّع دون كشف تفاصيل الخيوط منخفضة المستوى.

٢.٤.١٥ مجمّعات الخيوط

توفّر مجمّعات الخيوط تحكّماً صريحاً في:

- حجم المهام.
- عمق الاستدعاء.
- موازنة الحمل والجدولة.

تُعد ضرورية عندما يفرض هيكل الخوارزمية نمط التنفيذ المتوازي.

٣.٤.١٥ الاستراتيجية الهجينة

- استخدام سياسات التنفيذ للتوازي الخشن.
- استخدام مجمّعات الخيوط للتحكم الدقيق.
- تطبيق حدود قطع لتجنب إنشاء مهام مفرط.

تُحقق هذه المقاربة قابلية توسّع متينة على الأنظمة متعددة الأنوية.

٥.١٥ الاختيار الحتمي: Medians of Median

تطبّق Medians of Median «قسِّم تسُد» على إحصاءات الترتيب.

- زمن حتمي $O(n)$.
- يتجنب أسوأ حالات الاختيار العشوائي.
- يستخدم التجميع والاختيار التكراري للوسيط لضمان التوازن.

فكرة أساسية: غالباً ما تُقايض الضمانات الخوارزمية البساطةَ بقابلية التنبؤ والمتانة.

٦.١٥ فرز الدمج المتوازي

يربط فرز الدمج المتوازي الاستدعاء مباشرةً بالمهام المتوازية.

- فرز المقاطع المستقلة بالتوازي.
- التحكم في عمق الاستدعاء يحد من الكلفة.
- فعّال على المعماريات متعددة الأنوية.

مبدأ التصميم: يجب أن يكون التوازي مضبوطاً ومراعياً للمحلية لتحقيق تسريع حقيقي.

٧.١٥ الخلاصة

- يوفّر «قسِّم تسُد» صرامة نظرية وكفاءة عملية.
- يبرز Sort Merge وQuicksort مفاضلات تصميم متباينة.
- تُحسّن إزالة الاستدعاء الذّيلي والتحويلات التكرارية المتانة.
- تمكّن Modern C++ من توازي قابل للتوسّع عبر السياسات ومجمّعات الخيوط.
- يعزّز الاختيار الحتمي والفرز المتوازي إتقان تصميم الخوارزميات الاستدعائية.

الفصل ١٦: البرمجة الديناميكية

١.١٦ الحفظ مقابل الجدولة في Modern C++

تعالج البرمجة الديناميكية (DP) المشكلات ذات المسائل الفرعية المتداخلة والبنية المثلى. عملياً، تُعبَّر DP باستراتيجيتين متكاملتين:

- الحفظ (Memoization): استدعاء علوي مع تخزين النتائج.
- الجدولة (Tabulation): بناء جدولي سفلي تكراري.

توفّر Modern C++ حاويات معبّرة وفعّالة لكلا النهجين.

١.١.١٦ الحفظ (من الأعلى إلى الأسفل)

يعزّز الحفظ الحلّ العودي بتخزين نتائج المسائل الفرعية عند ظهورها.

- تماثل طبيعي لتعريفات المسائل العودية.
- يحسب فقط الحالات المطلوبة فعلياً.
- مناسب لفضاءات الحالات المتناثرة أو غير المنتظمة.

حاويات شائعة:

- std::unordered_map للمفاتيح المتناثرة أو المركّبة.
- std::vector أو std::optional للمؤشرات المحدودة.

مفاضلات:

- قد يكون عمق الاستدعاء كبيراً.
- تُضيف الذاكرات المعتمدة على التجزئة كلفة وصول.

٢.١.١٦ الجدولة (من الأسفل إلى الأعلى)

تبني الجدولة الحلّ تكرارياً انطلاقاً من الحالات الأساسية.

- تُلغي الاستدعاء الذاتي واستخدام المكدس.
- وصول متوقّع وصديق للمخبأ.
- تتطلب معرفة فضاء الحالات كاملاً مسبقاً.

حاويات شائعة:

- std::vector للجداول الكثيفة.
- مصفوفات متدحرجة لتقليل المساحة.

مفاضلات:

- قد تُخصّص ذاكرة أكثر من اللازم.
- أقل حدسية لمسائل تُعبَّر طبيعياً بشكل عودي.

٣.١.١٦ ملخص المقارنة

- الحفظ: معبّر ومرن وحساب عند الطلب.
- الجدولة: ثوابت أسرع، آمن للمكدس، ذاكرة متوقّعة.
- يعتمد الاختيار على كثافة الحالات وعمق الاستدعاء وقيود الذاكرة.

٢.١٦ DP على البنى الشائعة

١.٢.١٦ السلاسل

تعمل DP المعتمدة على السلاسل على البوادئ أو الفهارس.

- مسائل شائعة: LCS، مسافة التحرير، الحقيبة.
- انتقالات الحالات تعتمد على فهارس مجاورة.

تحسين:

- استخدام مصفوفات متدحرجة لتقليل المساحة من O(nm) إلى $O(\min(n, m))$

٢.٢.١٦ الأشجار

تعرّف DP على الأشجار حالات لكل عقدة وتُجمِّع عبر الأبناء.

- عودية بطبيعتها.
- غالباً ما تنقسم الحالات إلى تضمين/استبعاد.

تقنيات أساسية:

- اجتياز بعدي .(Post-order)

- إعادة الجذر (Re-rooting) لاستعلامات جميع الجذور.
- DFS تكراري لتجنب العودية العميقة.

٣.٢.١٦ الرسوم

تكون DP على الرسوم أسهل على الرسوم الموجّهة الخالية من الدورات .(DAGs)

- تتطلب ترتيباً طوبولوجياً.
- تُستخدم لمسائل أطول مسار والجدولة.

الرسوم الدورية:

- تتطلب حفظاً مع كشف الدورات.
- غالباً ما تمتزج بتقنيات أقصر مسار.

٣.١٦ تقنيات تقليل المساحة

تعتمد كفاءة DP بشدة على تحسين الذاكرة.

- مصفوفات متدحرجة: تخزين المقاطع اللازمة فقط.
- DP موضعي: إعادة استخدام تخزين الإدخال عند الأمان.
- ضغط Bitset: فعّال للحالات الثنائية.
- خرائط متناثرة: تجنب التخصيص الكثيف لفضاءات كبيرة.

١٦.٤ تمارين DP كلاسيكية

١٦.٤.١ متغيرات الحقيبة

- حقيبة 0/1: تكرار عكسي، مساحة O(W).
- حقيبة غير محدودة: تكرار أمامي لإعادة الاستخدام.
- حقيبة محدودة: تحسين بالتقسيم الثنائي.
- حقيبة متعددة الأبعاد: قيود موارد متعددة.

١٦.٤.٢ أطول تسلسل متزايد (LIS)

- DP تربيعي: بسيط، $O(n^2)$.
- فرز الصبر: تحسين بالبحث الثنائي، $O(n \log n)$.

فكرة:

- يبيّن LIS كيف تُدمج DP مع الجشع والبحث الثنائي.

١٦.٥ اصطلاحات C++ لأداء DP عالٍ

- تفضيل std::vector للذاكرة المتجاورة.
- استخدام reserve() لتجنب إعادة التخصيص.
- تطبيق std::lower_bound للانتقالات المُحسَّنة.
- استغلال دلالات النقل والعروض (std::span, std::ranges) عند الملاءمة.

٦.١٦ الخلاصة

- تُعرَّف DP بتصميم الحالات بقدر ما تُعرَّف باستراتيجية التنفيذ.
- الحفظ والجدولة تقنيتان متكاملتان لا متنافستان.
- تفرض السلاسل والأشجار والرسوم قوالب DP مميّزة.
- تحسين المساحة أساسي لقابلية التوسّع.
- تمكّن Modern C++ حلول DP موجزة وفعّالة ومتينة.

الفصل ١٧: الخوارزميات الجشعة ومفاهيم الماتروید

١.١٧ براهين صحة الجشع وأنماط C++ الاصطلاحية

تبني الخوارزميات الجشعة حلولها عبر اتخاذ اختيارات محلية مثلى متكررة. تكمن جاذبيتها في البساطة والكفاءة، لكن صحتها تعتمد على خصائص بنيوية قوية للمسألة. إن فهم لماذا ينجح الجشع أهم من فهم كيف نبرمجه.
يركّز هذا القسم على:

- تقنيات رسمية لإثبات صحة الجشع.
- اصطلاحات C++ القياسية لتنفيذ الاستراتيجيات الجشعة.

١.١.١٧ صحة الخوارزميات الجشعة

لا تكون الخوارزميات الجشعة صحيحة إلا إذا تحققت شروط محددة. تهيمن ثلاث تقنيات إثبات على تحليل الجشع.

خاصية الاختيار الجشع

تتحقق هذه الخاصية إذا أمكن بناء حل أمثل عبر اختيار محلي أمثل أولاً ثم حلّ الباقي جشعاً.
مثال: اختيار الأنشطة

- الهدف: تعظيم عدد الأنشطة غير المتداخلة.
- القاعدة الجشعة: اختيار النشاط ذي أقرب زمن انتهاء.

فكرة البرهان:

- انظر إلى حل أمثل.
- إن لم يتضمن النشاط الأسرع انتهاءً، استبدل نشاطه الأول به.
- لا يقلّل الاستبدال من القابلية أو حجم الحل.

يُظهر هذا التبادل أمان الاختيار الجشع.

حجة التبادل

تُظهر حجة التبادل أن أي حل أمثل يمكن تحويله إلى الحل الجشع دون الإضرار بالأمثلية.
مثال: ترميز هوفمان

- الخطوة الجشعة: دمج الرمزين الأقل تكراراً.
- في أي شجرة بادئة مثلى، يجب أن تظهر الرموز الأقل تكراراً في أقصى عمق.
- إن لم يحدث ذلك، فإن تبديل العقد يحفظ الكلفة ويستعيد البنية الجشعة.

منظور الماترويد

تنجح كثير من الخوارزميات الجشعة لأن المسألة تُشكّل ماترويداً.

- يجرّد الماترويد مفهوم الاستقلالية.
- في الماترويدات، يكون الاختيار الجشع بالوزن أمثلياً دائماً.
- مثال: خوارزمية كروسكال على الماترويد الرسومي.

٢.١.١٧ أنماط جشعة اصطلاحية في C++

تتطابق الخوارزميات الجشعة بسلاسة مع مكوّنات مكتبة C++ القياسية.

الفرز كتمهيد

يحدّد الفرز ترتيب النظر في الاختيارات الجشعة.

```
sort(items.begin(), items.end(),
     [](const Item& a, const Item& b) {
         return a.finish < b.finish;
     });
```

طوابير الأولوية

يُعبَّر عن اختيار أفضل مرشح متاح بشكل طبيعي باستخدام الأكوام.

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

المجموعات المرتبة ومتعددة العناصر

عند الإدراج والحذف الديناميكيين، توفّر الحاويات المرتبة ضمانات لوغاريتمية.

```
multiset<int> active;
```

مقارنات مخصّصة وLambdas

تتيح C++ ترميز القواعد الجشعة بإيجاز.

```
auto cmp = [](const auto& a, const auto& b) {
    return a.second > b.second;
};
priority_queue<pair<int,int>, vector<pair<int,int>>, decltype(cmp)> pq(cmp);
```

النطاقات والعروض (C++20)

تمكّن النطاقات من سلاسل تمهيد معبّرة.

```
auto view = activities
          | std::views::filter([](auto a){ return a.start >= 0; })
          | std::views::transform([](auto a){
                return pair{a.start, a.finish};
            });
```

٣.١.١٧ تطبيقات جشعة قياسية

- جدولة الفترات واختيار الأنشطة.

- أشجار الامتداد الدنيا (كروسكال).
- ترميز هوفمان.
- الحقيبة الكسرية.
- جدولة الوظائف مع المواعيد النهائية.

٢.١٧ ترميز هوفمان باستخدام std::priority_queue

يبني ترميز هوفمان شفرة بادئة مثلى تُقلّل طول المسار المُرجَّح. وهو مثال جشع نموذجي ببرهان تبادلي وتنفيذ مباشر بالأكوام.

١.٢.١٧ الرؤية الجشعة

- دمج الرمزين الأقل تكراراً مراراً.
- معاملة العقدة المدمجة كرمز جديد بتكرار مركّب.

٢.٢.١٧ البناء بالأكوام

```
struct Node {
    char ch;
    int freq;
    Node *left, *right;
};

struct Cmp {
    bool operator()(Node* a, Node* b) const {
        return a->freq > b->freq;
```

```
    }
};
```

```
priority_queue<Node*, vector<Node*>, Cmp> pq;
```

٣.٢.١٧ ملاحظات تنفيذية

- تفضيل lambdas لمقارنات موجزة.
- استخدام المؤشرات الذكية لتجنب الإدارة اليدوية للذاكرة.
- استخراج الشفرات عبر اجتياز الشجرة بزمن خطي.

٣.١٧ تمارين: اختيار الأنشطة وجدولة الفترات

١.٣.١٧ اختيار الأنشطة

- الفرز حسب زمن الانتهاء.
- اختيار الأنشطة المتوافقة جشعاً.
- التعقيد الكلي: $O(n \log n)$.

٢.٣.١٧ جدولة الفترات

```
sort(intervals.begin(), intervals.end(),
     [](auto& a, auto& b){ return a.end < b.end; });

int count = 0, lastEnd = -1;
```

```
for (auto& iv : intervals) {
    if (iv.start >= lastEnd) {
        ++count;
        lastEnd = iv.end;
    }
}
```

٣.٣.١٧ ملاحظات أساسية

- ينجح الجشع لأن الفترات الأسرع انتهاءً تهيمن.
- تفشل المتغيرات الموزونة وتتطلب DP.
- تدعم STL أنماط التصميم الجشع مباشرة.

٤.١٧ الخلاصة

- تعتمد صحة الجشع على خاصية الاختيار أو حجة التبادل.
- تُنمذج الماترويدات حالات الأمثلية المضمونة للجشع.
- يُعد الفرز والأكوام والمجموعات المرتبة أدوات C++ الأساسية.
- يُجسّد ترميز هوفمان اتحاد النظرية والتنفيذ.
- تُبرز مسائل الجدولة قوة الجشع وحدوده.

الفصل ١٨: الخوارزميات العشوائية والأساليب الاحتمالية

١.١٨ توليد الأعداد العشوائية في Modern C++

تُعد العشوائية أداة قوية في تصميم الخوارزميات، إذ تُبسّط التنفيذ وتحسّن الأداء المتوقع وتمكّن التقريب الفعّال. توفّر Modern C++ إطاراً صارماً عبر <random> يفصل بين: المولّدات، والتوزيعات، والبذور.

١.١.١٨ المولّدات

```
#include <random>

std::mt19937 rng(42); //
auto x = rng();       //
```

٢.١.١٨ التوزيعات

```
std::uniform_real_distribution<double> dist(0.0, 1.0);
double u = dist(rng);
```

٣.١.١٨ البذور وإعادة الإنتاج

```
std::mt19937 rng(std::random_device{}());
```

٤.١.١٨ نمط اصطلاحي

```
inline double random01() {
    static thread_local std::mt19937 rng(123);
    static std::uniform_real_distribution<double> dist(0.0, 1.0);
    return dist(rng);
}
```

٢.١٨ الخوارزميات العشوائية عملياً

١.٢.١٨ QuickSelect

```
int quickSelect(vector<int>& a, int l, int r, int k, mt19937& rng) {
    uniform_int_distribution<int> dist(l, r);
    int p = dist(rng);
    swap(a[p], a[r]);

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

    if (i == k) return a[i];
    if (k < i) return quickSelect(a, l, i - 1, k, rng);
```

```
    return quickSelect(a, i + 1, r, k, rng);
}
```

٢.٢.١٨ التجزئة العشوائية

```
struct RandomHash {
    size_t operator()(uint64_t x) const {
        static mt19937_64 rng(0xdeadbeef);
        static uint64_t seed = rng() | 1;
        x ^= x >> 33;
        x *= seed;
        x ^= x >> 33;
        return x;
    }
};
```

٣.٢.١٨ تقدير مونتِ كارلو

```
double estimatePi(int n) {
    mt19937 rng(42);
    uniform_real_distribution<double> d(0.0, 1.0);
    int inside = 0;
    for (int i = 0; i < n; ++i) {
        double x = d(rng), y = d(rng);
        if (x*x + y*y <= 1.0) ++inside;
    }
    return 4.0 * inside / n;
}
```

٣.١٨ هياكل احتمالية: مرشح بلوم

```
class BloomFilter {
    vector<bool> bits;
    int m, k;
    mt19937 rng;

    size_t hash(const string& s, int i) const {
        return std::hash<string>{}(s) ^ (i * 0x9e3779b97f4a7c15ULL);
    }

public:
    BloomFilter(int size, int hashes)
        : bits(size), m(size), k(hashes), rng(42) {}

    void insert(const string& s) {
        for (int i = 0; i < k; ++i)
            bits[hash(s, i) % m] = true;
    }

    bool contains(const string& s) const {
        for (int i = 0; i < k; ++i)
            if (!bits[hash(s, i) % m]) return false;
        return true;
    }
};
```

٤.١٨ الخلاصة

- تمكّن العشوائية خوارزميات بزمن متوقّع فعّال.
- يوفّر <random> عشوائية عالية الجودة وقابلة لإعادة الإنتاج.
- تمنع الخوارزميات العشوائية التدهور العدائي.
- تقدّم مونتِ كارلو تقريبات قابلة للتوسّع.
- تُبرز الهياكل الاحتمالية مفاضلة المساحة والدقة.

الفصل ١٩: خوارزميات التقريب والمسائل الصعبة حسابياً (NP-Hard)

١.١٩ استراتيجيات التقريب عملياً

تُعد كثير من مسائل التحسين الأساسية NP-Hard، ما يجعل الحلول الدقيقة غير عملية. تُقايض خوارزميات التقريب الأمثلية بالكفاءة مع ضمانات قرب من الأمثل ضمن زمن متعدد الحدود.

١.١.١٩ التقريب الجشع

2-أضعاف) (تقريب الرؤوس غطاء

```
vector<int> vertexCoverApprox(int n, vector<pair<int,int>>& edges) {
    vector<bool> used(n, false);
    vector<int> cover;
    for (auto& [u, v] : edges) {
        if (!used[u] && !used[v]) {
            cover.push_back(u);
            cover.push_back(v);
            used[u] = used[v] = true;
```

```
        }
    }
    return cover;
}
```

٢.١.١٩ استرخاء البرمجة الخطية

- حل LP المُسترخى.
- التقريب (التدوير) الحتمي أو العشوائي.
- نسب تقريب محدودة.

٣.١.١٩ البحث المحلي

TSP لـ 2-Opt مُحسِّن

```
void twoOpt(vector<int>& tour, int i, int k) {
    reverse(tour.begin() + i, tour.begin() + k + 1);
}
```

٤.١.١٩ PTAS وFPTAS

- PTAS: تقريب $(1+\varepsilon)$.
- FPTAS: كثير حدود في n و$1/\varepsilon$.

٢.١٩ الخلاصة

- تجعل خوارزميات التقريب مسائل NP-Hard قابلة للتعامل.
- يوفّر الجشع بساطة وضمانات.
- يقدّم البحث المحلي حلولاً عملية عالية الجودة.
- تمكّن PTAS/FPTAS مقايضات مضبوطة بين الدقة والأداء.
- تدعم اصطلاحات Modern C++ تنفيذات فعّالة وقابلة لإعادة الاستخدام.

الباب ٦

الأداء، التوازي، والاعتبارات منخفضة المستوى (مُركّز على C++)

الفصل ٢٠: تصميم الخوارزميات المراعية للذاكرة والمخبأ

١.٢٠ تنظيم البيانات، المحلية، وAoS مقابل SoA

تُصمَّم المعالجات الحديثة حول هياكل مخابئ عميقة ،(L1/L2/L3) ولذلك فإن أداء الخوارزميات في C++ غالباً لا تحكمه عدد التعليمات بقدر ما تحكمه أنماط الوصول إلى الذاكرة. يركّز التصميم المراعي للمخبأ على تعظيم المحلية المكانية والمحلية الزمنية عبر اختيار تنظيم بيانات مناسب.

١.١.٢٠ أساسيات محلية المخبأ

- المحلية المكانية: الوصول المتتالي للذاكرة يعيد استخدام نفس سطر المخبأ.
- المحلية الزمنية: البيانات التي استُخدمت مؤخراً يُعاد استخدامها قريباً.
- أسطر المخبأ: غالباً 64 بايت؛ تحميل بيانات غير مستخدمة يهدر عرض الحزمة.

الخوارزميات التي تتجاهل المحلية تتكبّد غرامات قاسية بسبب إخفاقات المخبأ.

٢.١.٢٠ مصفوفة البُنى (AoS)

التعريف تخزّن AoS الكائنات كاملةً بشكل متجاور.

```
struct Particle {
    float x, y, z;
    float vx, vy, vz;
};
std::vector<Particle> particles;
```

الخصائص

- تمثيل كائني طبيعي.
- فعّال عندما تُستخدم جميع الحقول معاً.
- غير فعّال لعمليات الحقل الواحد (تلويث المخبأ).

٣.١.٢٠ بنية المصفوفات (SoA)

التعريف تخزّن SoA كل حقل في مصفوفة متجاورة مستقلة.

```
struct ParticlesSoA {
    std::vector<float> x, y, z;
    std::vector<float> vx, vy, vz;
};
```

المزايا

- محلية مكانية ممتازة عند التكرار على حقل واحد.
- تمكين المتجهة .(SIMD)
- تجنّب تحميل بيانات غير مستخدمة إلى المخبأ.

```
void update(ParticlesSoA& p, float dt) {
    for (size_t i = 0; i < p.x.size(); ++i)
        p.x[i] += p.vx[i] * dt;
}
```

٤.١.٣٠ AoS مقابل SoA: إرشادات عملية

- استخدم AoS عندما تستهلك العمليات الكائن كاملاً.
- استخدم SoA للنوى العددية والمحاكاة واسعة النطاق.

٥.١.٣٠ تنظيمات هجينة (AoSoA)

يوازن تجميع الكائنات في كتل SoA صغيرة بين التجريد والمحلية.

```
struct ParticleBlock {
    float x[32], y[32], z[32];
    float vx[32], vy[32], vz[32];
};
```

يتوافق هذا التنظيم جيداً مع أسطر المخبأ ومسجلات .SIMD

٦.١.٢٠ تقنيات C++ مراعية للمخبأ

- المحاذاة: alignas(64) لمحاذاة سطر المخبأ.
- التخصيص المتجاور: تفضيل كتل std::vector الكبيرة.
- تجنّب المشاركة الكاذبة: فصل الحقول كثيرة الكتابة عبر أسطر مخبأ مختلفة.
- تنظيمات صديقة لـ SIMD: SoA للحساب المتجه.

٢.٢٠ خوارزميات مُحسّنة للمخبأ: التقسيم والتبليط

١.٢.٢٠ لماذا ينجح التقسيم

يعيد التقسيم تنظيم الحلقات بحيث تلائم البيانات العاملة المخبأ، ما يقلّل إعادة التحميل والطرد.

- تحسين المحلية الزمنية.
- تقليل الضغط على عرض حزمة الذاكرة.
- تمكين المتجهة داخل الحلقة الداخلية.

٢.٢.٢٠ مثال: ضرب المصفوفات

الساذجة النسخة

```
void matMulNaive(const vector<vector<double>>& A,
                 const vector<vector<double>>& B,
                 vector<vector<double>>& C, int n) {
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            for (int k = 0; k < n; ++k)
```

```
                C[i][j] += A[i][k] * B[k][j];
}
```

محلية ضعيفة عند الوصول إلى أعمدة B.

المُقسّمة النسخة

```
void matMulBlocked(const vector<vector<double>>& A,
                   const vector<vector<double>>& B,
                   vector<vector<double>>& C,
                   int n, int bs) {
    for (int ii = 0; ii < n; ii += bs)
        for (int jj = 0; jj < n; jj += bs)
            for (int kk = 0; kk < n; kk += bs)
                for (int i = ii; i < min(ii+bs, n); ++i)
                    for (int j = jj; j < min(jj+bs, n); ++j)
                        for (int k = kk; k < min(kk+bs, n); ++k)
                            C[i][j] += A[i][k] * B[k][j];
}
```

الفوائد

- إعادة استخدام عالية داخل الكتل.
- عدد أقل بكثير من إخفاقات المخبأ.
- متجهة تلقائية أفضل من المُصرّف.

٣.٢.٢٠ التبليط للبيانات متعددة الأبعاد

```
void tiled2D(vector<vector<double>>& a, int n, int ts) {
    for (int i0 = 0; i0 < n; i0 += ts)
        for (int j0 = 0; j0 < n; j0 += ts)
            for (int i = i0; i < min(i0+ts, n); ++i)
                for (int j = j0; j < min(j0+ts, n); ++j)
                    a[i][j] *= 2.0;
}
```

تلائم كل بلاطة المخبأ، ما يقلّل حركة الذاكرة.

٤.٢.٢٠ ممارسات C++ اصطلاحية

- استخدام تخزين 1D مسطّح للمصفوفات لضمان التجاور.
- تفضيل أحجام كتل constexpr للضبط.
- الجمع بين التقسيم والتنفيذ المتوازي لقابلية التوسّع.

٣.٢٠ تمارين

- مقارنة تحديث الجسيمات AoS مقابل SoA على مجموعات كبيرة.
- تنفيذ ضرب مصفوفات مُقسّم مقابل ساذج وقياس التسريع.
- ضبط أحجام الكتل لمستويات مخبأ مختلفة.
- دمج التبليط مع التنفيذ المتوازي وتقييم القابلية للتوسّع.

٤.٢٠ خلاصة

- تنظيم البيانات لا يقل أهمية عن التعقيد الخوارزمي.
- تنظيم SoA يحسّن كفاءة المخبأ واستغلال SIMD بشكل كبير.
- التقسيم والتبليط أساسيان لأحمال البيانات الكبيرة.
- تمكّن C++ الحديثة من تصميم مراعي للمخبأ دون التضحية بالوضوح.

الفصل ٢١: الخوارزميات المتوازية والمتزامنة

١.٢١ بدائيات التزامن الأساسية في C++ الحديثة

توفّر C++ الحديثة بدائيات تزامن منخفضة المستوى ومُعيارية تمكّن تصميم خوارزميات متوازية آمنة وقابلة للتوسّع على الأنظمة متعددة الأنوية. يتطلب الاستخدام الصحيح فهماً لـ دلالات التزامن ومفاضلات الأداء.

١.١.٢١ الخيوط كوحدات تنفيذ متوازٍ

تمثّل std::thread خيط تنفيذ بمستوى نظام التشغيل يتشارك فضاء العناوين.

```
#include <thread>
#include <iostream>

void worker(int id) {
    std::cout << "Worker " << id << "\n";
}

int main() {
    std::thread t1(worker, 1);
```

```
    std::thread t2(worker, 2);
    t1.join();
    t2.join();
}
```

تمكّن الخيوط توازياً على مستوى المهام، لكنها تتطلب تزامناً صريحاً عند مشاركة البيانات.

٢.١.٢١ آليات التزامن

الأقفال (Mutexes)

تحمي الأقفال المقاطع الحرجة وتضمن الإقصاء المتبادل.

```
#include <mutex>
#include <thread>

int counter = 0;
std::mutex m;

void increment() {
    for (int i = 0; i < 1000; ++i) {
        std::lock_guard<std::mutex> lock(m);
        ++counter;
    }
}
```

الخصائص:

- ضمانات صحة قوية.
- احتمال تنازع وحجب.
- عرضة للجمود إن أسيء استخدامها.

الذرّيات (Atomics)

توفّر عمليات بلا قفل لحالات مشاركة بسيطة.

```
#include <atomic>

std::atomic<int> counter{0};

void increment() {
    counter.fetch_add(1, std::memory_order_relaxed);
}
```

الخصائص:

- كلفة منخفضة.
- محدودة بعمليات بسيطة.
- تتطلب وعياً بترتيب الذاكرة.

متغيرات الشرط

تمكّن التنسيق بين الخيوط.

```
#include <condition_variable>
#include <mutex>

std::mutex m;
std::condition_variable cv;
bool ready = false;

void wait_task() {
```

```
    std::unique_lock<std::mutex> lock(m);
    cv.wait(lock, [] { return ready; });
}
```

تُستخدم لأنماط المنتج-المستهلك والتنفيذ المرحلي.

٣.١.٢١ مفاهيم التصميم بلا أقفال

تضمن الخوارزميات بلا أقفال تقدّم خيط واحد على الأقل دون حجب.

- مبنية على عمليات CAS الذرّية.
- تتجنب الجمود وانعكاس الأولوية.
- صعبة التصميم والتحليل.

```
std::atomic<Node*> head;

while (!head.compare_exchange_weak(old, new_node)) {}
```

ملاحظة: تتطلب الخوارزميات العملية بلا أقفال استراتيجيات آمنة للاسترجاع الذاكرة.

٢.٢١ الخوارزميات المتوازية مع std::execution

قدّمت C++17 سياسات تنفيذ تعبّر عن نية التوازي دون إدارة يدوية للخيوط.

١.٢.٢١ سياسات التنفيذ

- std::execution::seq — تسلسلي
- std::execution::par — متوازي
- std::execution::par_unseq — متوازي ومُتجه

٢.٢.٢١ مثال: فرز متوازي

```
#include <algorithm>
#include <execution>

std::sort(std::execution::par, data.begin(), data.end());
```

المزايا:

- تغييرات شفرة طفيفة.
- قابلية نقل عبر المنصات.
- موازنة حمل تلقائية.

٣.٢.٢١ القيود

- ترتيب تنفيذ غير حتمي.
- يتطلب عمليات خالية من سباقات البيانات.
- قد تهيمن الكلفة على المدخلات الصغيرة.

٣.٢١ جدولة سرقة العمل

سرقة العمل استراتيجية جدولة ديناميكية تعتمدها كثير من البيئات التنفيذية.

١.٣.٢١ المفهوم

- يمتلك كل عامل طابور مهام محلياً.
- يسرق العمال الخاملون مهاماً من غيرهم.

- تعظيم الاستغلال والمحلية.

٢.٣.٢١ الأهمية

- ممتازة للأحمال غير المنتظمة.
- تتوسّع جيداً مع عدد الأنوية.
- تقلّل التنازع المركزي.

تعتمد كثير من تطبيقات std::execution::par داخلياً على سرقة العمل.

٤.٢١ المجموع التراكمي المتوازي (Scan)

المجموع التراكمي بدائية متوازية أساسية.

$$P[i] = \sum_{k=0}^{i} A[k]$$

١.٤.٢١ حل متوازي معياري

```
#include <numeric>
#include <execution>

std::inclusive_scan(std::execution::par,
                    a.begin(), a.end(),
                    out.begin());
```

الخصائص:

- عمل $O(n)$ وعمق $O(\log n)$.

- يُستخدم في الفرز، والضغط، ونوى GPU.

٥.٢١ الطوابير المتزامنة

تدعم الطوابير المتزامنة تواصلاً آمناً بين المنتجين والمستهلكين.

١.٥.٢١ طابور بأقفال

```
template <typename T>
class ThreadSafeQueue {
    std::queue<T> q;
    std::mutex m;
public:
    void push(T v) {
        std::lock_guard<std::mutex> lock(m);
        q.push(v);
    }
};
```

بسيط وصحيح، لكنه قد لا يتوسّع جيداً.

٢.٥.٢١ طابور بلا أقفال (تصوري)

```
std::atomic<Node*> head;
std::atomic<Node*> tail;
```

الخصائص:

- تقدّم بلا حجب.

- إنتاجية أعلى تحت التنازع.
- تعقيد عالٍ في الصحة وإدارة الذاكرة.

٦.٢١ إرشادات التصميم

- تفضيل STL المتوازي للمهام ذات التوازي البياني.
- استخدام الذرّيات للحالات المشتركة البسيطة.
- تقليل المقاطع الحرجة.
- تجنّب التصاميم بلا أقفال إلا عند الضرورة.
- التصميم للمحلية وقابلية التوسّع.

٧.٢١ خلاصة

- التزامن مسألة صحة أولاً ثم أداء.
- يبسّط STL المتوازي التوازي الآمن.
- تمكّن سرقة العمل تنفيذاً قابلاً للتوسّع.
- المجموعات التراكمية والطوابير أنماط متوازية أساسية.
- توفّر C++ الحديثة أدوات تزامن عالية ومنخفضة المستوى.

الفصل ٢٢: الميتابرمجة وخوارزميات وقت الترجمة

١.٢٢ أسس تصميم الخوارزميات في وقت الترجمة

تمكّن الميتابرمجة بالقوالب (TMP) برامج C++ من إجراء حسابات أثناء الترجمة. تطوّرت هذه التقنية—عبر constexpr، والقوالب المتغايرة، والمفاهيم—إلى أسلوب منضبط للتعبير عن خوارزميات ساكنة ومنطق على مستوى الأنواع وحسابات مسبقة.
تُفضّل C++ (20/23) الوضوح: يُعبَّر عن الحسابات العددية بـ constexpr، بينما تبقى القوالب أساسية لمعالجة الأنواع والبنى.

١.١.٢٢ الحساب العددي في وقت الترجمة

```
template <int N>
struct Factorial {
    static constexpr int value = N * Factorial<N - 1>::value;
};

template <>
struct Factorial<0> {
```

```
    static constexpr int value = 1;
};
```

تستبدل C++ الحديثة ذلك بدوال constexpr الأبسط:

```
constexpr int factorial(int n) {
    return n <= 1 ? 1 : n * factorial(n - 1);
}

constexpr int f6 = factorial(6);
```

إرشاد: استخدم constexpr للقيم، والقوالب لمنطق الأنواع.

٢.١.٢٢ قوائم الأنواع وخوارزميات على مستوى النوع

```
template <typename... Ts>
struct TypeList {};
```

٢.٢٢ المفاهيم وconstexpr (C++20/23)

```
template <std::integral T>
constexpr T sum_to(T n) {
    return (n * (n + 1)) / 2;
}
```

٣.٢٢ الخلاصة

- تنقل خوارزميات وقت الترجمة العمل من التنفيذ إلى الترجمة.
- تفرض المفاهيم صحة دلالية مبكّرة.
- تمكّن constexpr حسابات حقيقية مسبقة.
- تستبدل المتتاليات الساكنة الحلقات في وقت الترجمة.
- يفرض consteval ثوابت لا تقبل التفاوض.

الفصل ٢٣: التوصيف، القياس، وسير عمل التحسين

١.٢٣ التحسين المعتمد على القياس

يبدأ التحسين الفعّال بالأدلة لا بالحدس. يعتمد سير العمل المنضبط على: الموصّفات، والمعقّمات، وأعلام المُصرّف.

٢.٢٣ الخلاصة

- التحسين بلا قياس تخمين.
- الصحة شرط مسبق للأداء.
- التحسين الخوارزمي يحقق أكبر المكاسب.
- يوجّه التوصيف مواضع التحسين.
- التحسينات الدقيقة هي المرحلة الأخيرة.

هندسة الأداء في C++ عملية منهجية تجمع بين الأدوات والنظرية والتجريب المنضبط.

الباب ٧

مشاريع التتويج (Capstone Projects)

الفصل ٢٤: المشروع A — مكتبة رسوم بيانية عالية الأداء

١.٢٤ أهداف التصميم والمبادئ المعمارية

يدمج هذا المشروع التتويجي بين النظرية الخوارزمية، والتصميم المراعي للذاكرة، وهندسة C++ الحديثة لبناء مكتبة رسوم بيانية عالية الأداء. التركيز هنا ليس على كثرة الميزات، بل على أداء متوقّع، وقابلية التوسّع، وتجريد نظيف.

١.١.٢٤ أهداف التصميم الأساسية

١. الأداء أولاً

- تنظيمات ذاكرة متجاورة وصديقة للمخبأ
- اجتياز فعّال لرسوم بيانية تضم ملايين الحواف
- أقل قدر ممكن من كلفة التجريد

٢. عمومية وقابلية التوسعة

- رسوم بيانية موجهة وغير موجهة

- حواف موزونة وغير موزونة
- أنواع فهارس وأوزان قابلة للتهيئة

٣. سهولة استخدام بأسلوب STL

- اجتياز يعتمد على for range-based
- اجتياز مدفوع بالمُكرِّرات
- فصل واضح بين الواجهة والتخزين الداخلي

٤. قابلية التوسّع

- اجتياز متوازٍ فعّال للقراءة فقط
- تصميم موجّه للرسوم البيانية الثابتة أو قليلة التعديل

٢.٢٤ نظرة عامة على الواجهة البرمجية

تتبع الواجهة العامة اتفاقيات STL مع إخفاء تفاصيل التنظيم الداخلي.

```
Graph<int> g(vertices);
g.add_edge(0, 1, 5);
g.add_edge(2, 3);

for (auto v : g.vertices()) { /* ... */ }
for (auto n : g.neighbors(0)) { /* ... */ }
```

تعمل الخوارزميات مباشرةً على تجريد الرسم البياني:

```
auto dist = dijkstra(g, 0);
auto mst  = prim(g);
```

٣.٢٤ المُكرِّرات والاجتياز

توفّر المُكرِّرات وصولاً بلا كلفة إضافية إلى بنية الرسم البياني:

- مُكرِّرات الرؤوس: لاجتياز جميع الرؤوس
- مُكرِّرات التجاور: لاجتياز جيران رأس معيّن
- مُكرِّرات الحواف: لاجتياز الحواف على مستوى الرسم كاملاً

تُنفَّذ هذه المُكرِّرات كعروض خفيفة الوزن فوق نطاقات فهارس داخلية، ما يضمن اجتيازاً بزمن ثابت دون نسخ.

٤.٢٤ تنظيم الذاكرة: الصفوف المتناثرة المضغوطة (CSR)

تعتمد المكتبة على Row Sparse Compressed (CSR) لتخزين التجاور.

١.٤.٢٤ بنية CSR

- row_ptr: إزاحات لكل رأس
- col_idx: رؤوس الوجهة
- weights (اختياري): أوزان الحواف

```
row_ptr = [0, 2, 3, 3]
col_idx = [1, 2, 2]
```

٢.٤.٢٤ مبررات الاختيار

- تمثيل ذاكرة مدمج
- وصول متجاور للجيران
- محلية مخبأ ممتازة
- تنسيق صناعي قياسي للبيانات المتناثرة

٥.٢٤ تنفيذ الخوارزميات الأساسية

١.٥.٢٤ أقصر مسار من مصدر واحد (SSSP)

- Dijkstra: أوزان غير سالبة، قائمة أولوية
- Bellman--Ford: دعم الأوزان السالبة

يتيح CSR اجتياز الجيران بسرعة ووصولاً متوقّعاً للذاكرة.

٢.٥.٢٤ شجرة الامتداد الصغرى (MST)

- Kruskal: فرز الحواف مع DSU
- Prim: قائمة أولوية مع اجتياز CSR

٣.٥.٢٤ مقاييس المركزية

- مركزية الدرجة عبر إزاحات الصفوف في CSR
- مركزية القرب عبر تكرار SSSP

- مركزية التوسّط باستخدام خوارزمية Brandes

تسمح الاجتيازات المستقلة بتوازي اختياري.

٦.٢٤ اعتبارات الأداء

- يقلّل CSR من إخفاقات المخبأ
- يحافظ الوصول عبر المُكرِّرات على التجريد
- تتيح القوالب مقايضات بين الدقة والذاكرة
- يوجّه التوصيف مواضع التحسين

٧.٢٤ استراتيجية الاختبار

- اختبارات وحدات للمُكرِّرات وسلامة CSR
- اختبارات تكامل لـ SSSP وMST والمركزية
- تحقق حتمي على رسوم صغيرة

٨.٢٤ منهجية القياس

١.٨.٢٤ البيانات

- اصطناعية: عشوائية، شبكية، عديمة المقياس
- واقعية: اجتماعية، طرق، شبكات اقتباس

٢.٨.٢٤ المقاييس

- زمن التنفيذ
- بصمة الذاكرة
- إخفاقات المخبأ (عدادات عتادية)
- قابلية التوسّع مع حجم الرسم

٣.٨.٢٤ سير العمل

١. تحميل مجموعة البيانات
٢. تشغيل الخوارزمية
٣. قياس الزمن والذاكرة
٤. المقارنة مع تطبيقات مرجعية
٥. تسجيل نتائج قابلة للإعادة الإنتاج

٩.٢٤ ملاحظات أساسية

- يوفّر CSR تسريعات متعددة الأضعاف على الرسوم المتناثرة
- اختيار الخوارزمية يتفوّق على التحسينات الدقيقة
- تتوسّع مركزية التوسّط مع عدد الأنوية

١٠.٢٤ الخلاصة النهائية

يُظهر هذا المشروع كيف تتكامل الصرامة الخوارزمية، والتصميم المراعي للذاكرة، وتجريدات C++ الحديثة لإنتاج مكتبة رسوم بيانية قابلة للتوسّع وعالية الأداء. عبر توحيد تخزين CSR، وواجهات مدفوعة بالمُكرِّرات، وخوارزميات رسومية مثبتة، يجسر المشروع بين نظرية معالجة الرسوم والهندسة العملية للأنظمة.

الفصل ٢٥: المشروع B — مُصرِّف / مُفسِّر مصغّر

١.٢٥ تصميم الواجهة الأمامية: التحليل المعجمي والنحوي

يطوّر هذا المشروع مُصرِّفاً/مُفسِّراً مصغّراً بهندسة تعليمية واضحة ومراعية للأداء. تتكوّن الواجهة الأمامية من التحليل المعجمي والتحليل النحوي باستخدام تقنيات C++ الحديثة التي تؤكّد سلامة الأنواع والوضوح وقابلية الصيانة.

١.١.٢٥ التحليل المعجمي (Lexer)

يحوّل المُحلِّل المعجمي النص الخام إلى تدفّق منظّم من الرموز .(Tokens)

أهداف التصميم

- تجزئة حتمية
- تتبّع دقيق للأسطر والأعمدة
- أقل تخصيص ونسخ ممكن
- كشف مبكّر للإدخال غير الصالح

التمثيل الأساسي

```
enum class TokenType {
    Identifier, Number,
    Plus, Minus, Star, Slash,
    LParen, RParen,
    End
};

struct Token {
    TokenType type;
    std::string_view lexeme;
    int line;
    int column;
};
```

هيكل المُحلِّل المعجمي

```
class Lexer {
public:
    explicit Lexer(std::string_view src)
        : source(src), pos(0), line(1), column(1) {}

    Token next() {
        skip_ws();
        if (pos >= source.size())
            return {TokenType::End, {}, line, column};

        char c = source[pos];
```

```
        if (std::isdigit(c)) return number();
        if (std::isalpha(c)) return identifier();

        switch (c) {
            case '+': return simple(TokenType::Plus);
            case '-': return simple(TokenType::Minus);
            case '*': return simple(TokenType::Star);
            case '/': return simple(TokenType::Slash);
            case '(': return simple(TokenType::LParen);
            case ')': return simple(TokenType::RParen);
        }
        throw std::runtime_error("Invalid character");
    }

private:
    std::string_view source;
    size_t pos;
    int line, column;
};
```

٢.١.٢٥ التحليل النحوي (Parser)

يحوّل التحليل النحوي الرموز إلى شجرة تركيب مجرّدة (AST) تمثّل بنية البرنامج.

التحليل التنازلي التكراري

يربط هذا الأسلوب قواعد النحو مباشرةً بدوال، ما يوفّر قابلية قراءة وتقارير أخطاء دقيقة.

```
struct Expr { virtual ~Expr() = default; };
```

```
struct BinaryExpr : Expr {
    char op;
    std::unique_ptr<Expr> lhs, rhs;
};

class Parser {
public:
    explicit Parser(const std::vector<Token>& t)
        : tokens(t), pos(0) {}

    std::unique_ptr<Expr> expression() {
        return term();
    }

private:
    const std::vector<Token>& tokens;
    size_t pos;
};
```

مركِّبات المحلِّلات (بديل)

تؤلّف مركِّبات المحلِّلات محلِّلات صغيرة في أخرى أكبر عبر دوال عالية الرتبة، مع تفضيل الوحدوية وإعادة الاستخدام على السرعة القصوى.

```
template<typename T>
struct ParseResult {
    T value;
    bool ok;
};
```

٣.١.٢٥ تقنيات C++ الحديثة

- std::string_view لرموز بلا نسخ
- std::unique_ptr لملكية AST
- std::variant لعُقد AST الجبرية
- RAII لسلامة الذاكرة

٢.٢٥ تحويلات AST وتدفّق التحكّم

بعد التحليل، يُنقّي المُصرِّف AST ويهيّئها للتنفيذ أو التخفيض اللاحق.

١.٢.٢٥ تمثيل AST

```
struct Literal : Expr { int value; };
struct Variable : Expr { std::string name; };
struct Binary : Expr {
    char op;
    std::unique_ptr<Expr> lhs, rhs;
};
```

٢.٢.٢٥ التحويلات الأساسية

- طيّ الثوابت
- تبسيط جبري
- إزالة الشيفرة الميتة

```
std::unique_ptr<Expr> fold(Binary* b) {
    auto* l = dynamic_cast<Literal*>(b->lhs.get());
    auto* r = dynamic_cast<Literal*>(b->rhs.get());
    if (l && r) {
        return std::make_unique<Literal>(
            b->op == '+' ? l->value + r->value
                         : l->value - r->value
        );
    }
    return nullptr;
}
```

٣.٢.٢٥ تحليل تدفّق التحكّم

- بناء مخطط تدفّق التحكّم (CFG)
- تحليل المُسيطرات
- تتبّع أساسي لتدفّق البيانات

تمكّن هذه التحليلات تحسينات آمنة وتفسيراً منظّماً.

٣.٢٥ تمارين: الشيفرة ثلاثية العناوين وتخصيص المسجلات

١.٣.٢٥ الشيفرة ثلاثية العناوين (TAC)

```
b + a = t0
c * t0 = t1
t1 = x
```

تمثيل TAC

```
enum class OpCode { Add, Sub, Mul, Div };

struct TAC {
    OpCode op;
    std::string dst, lhs, rhs;
};
```

مثال التوليد

```
std::string gen(Expr* e) {
    if (auto* b = dynamic_cast<Binary*>(e)) {
        auto l = gen(b->lhs.get());
        auto r = gen(b->rhs.get());
        auto t = new_temp();
        code.push_back({OpCode::Add, t, l, r});
        return t;
    }
    return literal_or_var(e);
}
```

٢.٣.٢٥ تخصيص المسجلات (Linear Scan)

```
std::unordered_map<std::string, int> reg;
std::vector<bool> free_reg(8, true);

int alloc(const std::string& t) {
    for (int i = 0; i < free_reg.size(); ++i)
```

```
        if (free_reg[i]) {
            free_reg[i] = false;
            reg[t] = i;
            return i;
        }
    throw std::runtime_error("Spill required");
}
```

٣.٣.٢٥ نواتج التعلّم

- فهم تصميم IR عبر TAC
- تطبيق منطق مجال الحياة
- ممارسة التخصيص تحت قيود الموارد
- ربط منطق AST بالتنفيذ

الخلاصة النهائية

يُظهر المشروع B كيف ينبثق مسار مُصرِّف كامل من مكوّنات منظّمة: التحليل المعجمي، والتحليل النحوي، وتحويلات AST، وتوليد الشيفرة الوسيطة، وتخصيص المسجلات. بمزج نظرية المُصرِّفات الكلاسيكية مع ممارسات C++ الحديثة، يوفّر المشروع أساساً متيناً للمُفسِّرات، ومحركات البايت كود، والتوليد الأصلي لاحقاً.

الفصل ٢٦: المشروع C — مُختبِر استراتيجيات تداول خوارزمية

١.٢٦ خوارزميات السلاسل الزمنية: البث، النوافذ المنزلقة، والملخّصات الآنية

يطبّق هذا المشروع التفكير الخوارزمي على بيانات السلاسل الزمنية المالية المستمرة وعالية التردد وغير المحدودة غالباً. الهدف تصميم نظام اختبار خلفي يحسب المؤشرات ويقيّم الاستراتيجيات تدريجياً مع ذاكرة محدودة وأداء متوقّع. تمكّن C++ الحديثة تنفيذات فعّالة وآمنة وقابلة للتركيب.

١.١.٢٦ خوارزميات البث

تحدّث خوارزميات البث الإحصاءات مع وصول بيانات جديدة دون الرجوع للتاريخ.

- معالجة بمرور واحد
- تحديث O(1) لكل نبضة
- دون تخزين تاريخ غير محدود

مثال: المتوسط المتحرك الأُسّي (EMA)

```
class EMA {
    double alpha;
    double value = 0.0;
    bool init = false;

public:
    explicit EMA(double a) : alpha(a) {}

    double update(double price) {
        if (!init) { value = price; init = true; }
        else { value = alpha * price + (1 - alpha) * value; }
        return value;
    }
};
```

يضمن هذا التصميم تحديثاً بزمن ثابت وذاكرة ضئيلة.

٢.١.٢٦ خوارزميات النوافذ المنزلقة

تحسب النوافذ المنزلقة إحصاءات على عدد ثابت من أحدث القيم.

- المتوسط المتحرك البسيط
- التباين المتدحرج
- القيم الصغرى/الكبرى المتدحرجة

مثال: المتوسط المتحرك البسيط (SMA)

```
class SMA {
    std::deque<double> buf;
```

```
    size_t window;
    double sum = 0.0;

public:
    explicit SMA(size_t w) : window(w) {}

    double update(double price) {
        buf.push_back(price);
        sum += price;
        if (buf.size() > window) {
            sum -= buf.front();
            buf.pop_front();
        }
        return sum / buf.size();
    }
};
```

لا تُحتفظ إلا بأحدث القيم، ما يضمن ذاكرة محدودة.

٣.١.٢٦ ملخّصات التعلّم الآني

تلخّص الخوارزميات التقريبية تدفّقات كبيرة أو غير محدودة.
Sketch Count-Min (مفهومياً)

- تتبّع تقريبي لتكرارات الأحداث
- بصمة ذاكرة ثابتة
- حدود خطأ احتمالية

```
class CountMinSketch {
    std::vector<std::vector<int>> table;
    size_t depth, width;

public:
    CountMinSketch(size_t d, size_t w)
        : depth(d), width(w), table(d, std::vector<int>(w, 0)) {}

    void update(int x) {
        for (size_t i = 0; i < depth; ++i)
            table[i][hash(i, x) % width]++;
    }

    int query(int x) const {
        int r = INT_MAX;
        for (size_t i = 0; i < depth; ++i)
            r = std::min(r, table[i][hash(i, x) % width]);
        return r;
    }
};
```

تفيد هذه الملخّصات في كشف الشذوذ ضمن بيانات عالية التردد.

٢.٢٦ معمارية محرّك الاختبار الخلفي

١.٢.٢٦ أهداف التصميم

١. فصل واضح بين البيانات والمؤشرات والاستراتيجيات والتنفيذ

٢. تحديثات تدريجية O(1) لكل نبضة

٣. هياكل بيانات صديقة للمخبأ

٤. تشغيلات حتمية قابلة لإعادة الإنتاج

٢.٢.٢٦ معمارية طبقية

- طبقة البيانات: تبث الأسعار التاريخية أو الحيّة
- طبقة المؤشرات: EMA وSMA والتقلّب
- طبقة الاستراتيجية: توليد إشارات التداول
- طبقة التنفيذ: محاكاة الصفقات وP&L

```
for (const auto& bar : data) {
    double e1 = ema_fast.update(bar.close);
    double e2 = ema_slow.update(bar.close);
    strategy(bar, e1, e2);
    executor.process(bar);
}
```

٣.٢.٢٦ قيود الأداء

- تحديثات مؤشرات O(1)
- استخدام ذاكرة محدود
- توازٍ اختياري على مستوى الرموز
- تحسين موجّه بالتوصيف

٣.٢٦ تمارين: تقاطع EMA وتحليل الكمون

١.٣.٢٦ استراتيجية تقاطع المتوسطات

```
class EMACrossover {
    EMA fast, slow;
    bool long_pos = false;

public:
    EMACrossover(double a, double b)
        : fast(a), slow(b) {}

    void update(double price) {
        double f = fast.update(price);
        double s = slow.update(price);

        if (!long_pos && f > s) { long_pos = true; buy(price); }
        else if (long_pos && f < s) { long_pos = false; sell(price); }
    }
};
```

٢.٣.٢٦ قياس الكمون

```
auto t0 = std::chrono::high_resolution_clock::now();
for (auto& bar : stream) strategy.update(bar.close);
auto t1 = std::chrono::high_resolution_clock::now();

double avg =
    std::chrono::duration<double, std::milli>(t1 - t0).count()
```

```
    / stream.size();
```

٣.٣.٢٦ نواتج التعلّم

- تصميم خوارزميات البث
- تحسين النوافذ المنزلقة
- تقييم الأنظمة الحسّاسة للكمون
- سير عمل اختبار خلفي متكامل

الخلاصة النهائية

يُظهر المشروع C كيف تمكّن خوارزميات البث والنوافذ المنزلقة والملخّصات الآنية من معالجة فعّالة لبيانات سلاسل زمنية مالية. عبر الجمع بين الحساب التدريجي والذاكرة المحدودة واصطلاحات C++ الحديثة، يحقّق نظام الاختبار الخلفي قابلية التوسّع وإعادة الإنتاج وأداءً مناسباً لتجارب تداول خوارزمية واقعية.

الباب ٨

الاختبار، قابلية إعادة الإنتاج & الممارسات البحثية

الفصل ٢٧: اختبار صحة الخوارزميات في C++

١.٢٧ الاختبار القائم على الخصائص، التشويش (Fuzzing)، والتحقق الحتمي

لا يقتصر اختبار صحة الخوارزميات على أزواج دخل--خرج ثابتة. تتيح C++ الحديثة الاختبار القائم على الخصائص، وتشويش المدخلات، والتنفيذ الحتمي، بما يضمن الصحة عبر مساحات دخل واسعة، وحالات حدّية، وتطبيقات محسّنة.

١.١.٢٧ الاختبار القائم على الخصائص

يتحقق هذا الأسلوب من ثوابت خوارزمية يجب أن تصح لكل دخل صالح.

- الفرز: الناتج مرتّب ويحافظ على العناصر
- اجتياز الرسوم البيانية: زيارة جميع العقد القابلة للوصول
- الدوال الرياضية: عدم خرق قيود المجال

```
template<typename SortFn>
void test_sort(SortFn sort_fn) {
    std::mt19937 rng(42);
```

```
    {
        {
            auto ref = v;
            sort_fn(v);
            assert(v == ref);
            assert(std::is_sorted(v.begin(), v.end()));
            std::sort(ref.begin(), ref.end());
        }
    }

    std::vector<int> v(100);
    for (int i = 0; i < 100; ++i) {
        v[i] = dist(rng);
    }
    for (auto &x : v) std::cout << x << ' ';
    std::cout << '\n';
    std::sort(v.begin(), v.end());
    for (int i = 1; i < 100; ++i) {
        assert(v[i - 1] <= v[i]);
    }
    std::cout << "sorted ok\n";
}
```

تستكشف هذه الاختبارات العديد من المدخلات تلقائياً مع الحفاظ على قابلية الإعادة الكاملة.

٢.١.٢٧ تشويش المدخلات (Fuzzing)

يولّد التشويش مدخلات عشوائية وعدائية لكشف الأعطال، وخرق الثوابت، أو السلوك غير المعرّف.

```
void fuzz_bfs() {
    std::mt19937 rng(123);
    int n = 1 + rng() % 50;

    std::vector<std::vector<int>> g(n);
    for (int i = 0; i < 2 * n; ++i) {
        g[rng() % n].push_back(rng() % n);
    }

    std::vector<bool> visited(n, false);
```

```
;{0q{ <int>vector::std
;true = [0visited[


} q.empty())!) while
q.pop_back(); q.back(); = u int
g[u]) : v int) for
{ q.push_back(v); ;true = visited[v] } visited[v])!) if
{


n); => (true ,visited.end() ,count(visited.begin()::assert(std
{
```

يكمل التشويش الاختبار القائم على الخصائص عبر الضغط على الحالات الحدّية والأوضاع غير الصالحة.

٣.١.٢٧ الحتمية في الاختبارات

الحتمية أساسية للتصحيح، وقابلية إعادة الإنتاج البحثية، واستقرار التكامل المستمر .(CI)

- تهيئة مولدات الأرقام العشوائية دائماً
- تجنّب الحالة العالمية القابلة للتغيير
- تسجيل المدخلات الفاشلة لإعادة التشغيل
- تمرير مولدات RNG صراحةً

```
} rng) &mt19937::(stddeterministic_test void
;(100 ,0d( <int>uniform_int_distribution::std
d(rng); = x int
;(100 => x && 0 =< assert(x
{
```

٢.٢٧ الاختبار المعتمد على الأطر والتكامل مع CI

١.٢.٢٧ اختبارات الوحدات باستخدام GoogleTest

يوفّر GoogleTest اختبارات وحدات وتكامل منظّمة.

```
} BasicCases) ,TEST(SortTest
;{2 ,1 ,3} = v <int>vector::std
v.end()); ,sort(v.begin()::std
v.end())); ,is_sorted(v.begin()::ASSERT_TRUE(std
{
```

تتحقق اختبارات الوحدات من السلوكيات المعروفة والحالات الحدّية بشكل حتمي.

٢.٢.٢٧ اختبار الخصائص بأسلوب QuickCheck

تؤتمت مكتبات اختبار الخصائص التحقق العشوائي من الثوابت.

```
,elements" and order preserves "sortingcheck(::rc
} v) &<int>vector::std const)[]
v; = s auto
s.end()); ,sort(s.begin()::std
s.end())); ,is_sorted(s.begin()::RC_ASSERT(std
v; = r auto
r.end()); ,sort(r.begin()::std
r); == RC_ASSERT(s
;({
```

٣.٢.٢٧ التكامل المستمر

يضمن CI تشغيل الاختبارات عند كل التزام وعلى كل منصة.

```
cmake -S . -B build
cmake --build build
ctest --test-dir build
```

تفرض مسارات CI السلامة من الانحدار، وقابلية الإعادة، والصحة عبر المنصات.

الخلاصة

- الاختبار القائم على الخصائص يتحقق من الثوابت الخوارزمية
- التشويش يكشف الأعطال والحالات الحدّية والسلوك غير المعرّف
- التنفيذ الحتمي يضمن قابلية إعادة الإنتاج
- اختبارات الوحدات والخصائص وCI تشكّل استراتيجية صحة متكاملة

تُمكّن هذه التقنيات معاً تطوير خوارزميات C++ عالية الاعتمادية مناسبة للبحث العلمي والأنظمة الحرجة إنتاجياً.

الفصل ٢٨: التجارب القابلة لإعادة الإنتاج & مجموعات البيانات

١.٢٨ إدارة البيانات، المُولِّدات الاصطناعية، الحتمية، وإعداد التقارير

تعتمد قابلية إعادة إنتاج التجارب في C++ على انضباط إدارة مجموعات البيانات، وتوليد بيانات حتمي، وإعداد تقارير موحّد. تضمن هذه الممارسات تكرار النتائج والتحقق منها ومقارنتها عبر الأجهزة والزمن.

١.١.٢٨ إدارة مجموعات البيانات

تفصل الإدارة الفعّالة للبيانات بين البيانات والخوارزميات.

- إبقاء البيانات الخام غير قابلة للتغيير وتخزين المعالجة بشكل منفصل
- ترقيم الإصدارات وتوثيق التحويلات
- تسجيل البيانات الوصفية: الحجم، التوزيع، النطاقات، خطوات المعالجة
- استخدام std::vector أو std::array للذاكرة؛ واستخدام الذاكرة المعيّنة للملفات للبيانات الضخمة

```
} file) &string::std constload_dataset( <int>vector::std
    (file);in ifstream::std
    data; <int>vector::std
    x; int
    data.push_back(x); x) >> (in while
    data; return
{
```

٢.١.٢٨ المُولِّدات الاصطناعية للبيانات

تتيح البيانات الاصطناعية تجارب مضبوطة وقابلة للتوسّع ومركّزة على الإجهاد.

```
} (42 = seed unsigned ,hi int ,lo int ,n size_tgenerate_uniform( <int>vector::std
    (seed);rng mt19937::std
    hi); ,dist(lo <int>uniform_int_distribution::std
    v(n); <int>vector::std
    dist(rng); = x v) : x &auto) for
    v; return
{
```

- تضمن البذور الحتمية قابلية الإعادة
- تدعم توزيعات منتظمة وطبيعية وعدائية
- تندمج بسلاسة مع الاختبارات والقياسات

٣.١.٢٨ التهيئة والحتمية

الحتمية إلزامية للعلم القابل للإعادة الإنتاج ولتصحيح الأخطاء.

- تهيئة RNG صراحةً دائماً

- تسجيل البذور والمعاملات لكل تجربة
- استخدام RNG محليّة للخيط في الشيفرات المتوازية

```
} out) &<int>vector::std ,seed unsigned)generate_block void
    rng(seed); mt19937::std
    ;(100 ,0d( <int>uniform_int_distribution::std
    d(rng); = x out) : x &auto) for
{
```

٤.١.٢٨ معايير إعداد التقارير

يجب أن تنتج التجارب مخرجات قابلة للقراءة الآلية والتدقيق.

- تسجيل اسم مجموعة البيانات وإصدارها وحجمها والمعالجة
- تسجيل البذور والمعاملات والمترجم ونظام التشغيل والأعلام
- تخزين النتائج بصيغة CSV أو JSON للتحليل الآلي

```
} time_ms) double ,seed unsigned ,data &string::std const)log_run void
    app);::ios::std ,"runs.csv"log( ofstream::std
    ;"\n" >> time_ms >> "," >> seed >> "," >> data >> log
{
```

٢.٢٨ نشر التجارب: CMake وDocker وقائمة تحقق للإعادة الإنتاج

١.٢.٢٨ التغليف باستخدام CMake

يضمن CMake بناءً متسقاً ومتعدد المنصات.

```
cmake_minimum_required(VERSION 3.22)
project(Experiments LANGUAGES CXX)
set(CMAKE_CXX_STANDARD 23)
add_executable(experiment src/main.cpp)
```

- نظام بناء واحد لكل المنصات
- تكامل طبيعي مع مسارات CI

٢.٢.٢٨ الحاويات باستخدام Docker

يجمّد Docker بيئة التنفيذ.

```
FROM ubuntu:24.04
RUN apt-get update && apt-get install -y build-essential cmake
COPY . /work
WORKDIR /work
RUN cmake -S . -B build && cmake --build build
CMD ["./build/experiment"]
```

٣.٢.٢٨ قائمة تحقق دنيا لإعادة الإنتاج

١. توثيق نظام البناء وإصدارات المترجم

٢. توفير مجموعات البيانات أو المُولِّدات مع بذور ثابتة

٣. تعليمات تنفيذ واضحة

٤. اختبارات مضمّنة وقابلة للتشغيل

٥. سجلات تلتقط المعاملات والبيئة

٤.٢.٢٨ الخلاصة

- تحافظ إدارة البيانات المنظّمة على سلامة التجارب
- تتيح المُولِّدات الاصطناعية والبذور الثابتة دراسات حتمية
- يدعم التسجيل الموحّد التحقق والمقارنة
- تجعل CMake وDocker التجارب محمولة وقابلة للمشاركة

تمكّن هذه الممارسات معاً تجارب C++ عالية الجودة وقابلة لإعادة الإنتاج مناسبة للنشر البحثي والتحقق طويل الأمد.

الملاحق

الملحق A --- دليل C++ السريع لمطوري الخوارزميات

يوفر هذا الملحق مرجعًا مكثفًا وعالي الإشارة لمطوري الخوارزميات باستخدام C++ الحديثة. يلخص أهم حاويات STL والمكررات، والأنماط الخوارزمية، وإرشادات الأداء اللازمة لكتابة كود فعال، واضح، وقابل للصيانة.

A.1 اختيار حاويات STL

اختيار الحاوية الصحيحة هو قرار أداء أساسي.

- std::vector — الخيار الافتراضي، ذاكرة متجاورة، صديق للـ cache.
- std::deque — إدخال وإزالة فعالة من الطرفين.
- std::array — حجم ثابت دون كلفة إضافية.
- std::forward_list / std::list — فقط عند الحاجة إلى ثبات المكررات.
- std::set / std::map — تخزين مرتب بزمن لوغاريتمي.
- std::unordered_set / std::unordered_map — تجزئة بزمن وسطي O(1).

- Adapters: stack, queue, priority_queue.

قاعدة عامة: استخدم std::vector ما لم يوجد سبب خوارزمي واضح لغيره.

A.2 المكررات (Iterators) والنطاقات

تصنيفات المكررات (من الأضعف إلى الأقوى):

- Input / Output
- Forward
- Bidirectional
- Random Access

```
auto it = c.begin();
std::advance(it, n);
*it;
```

الأسلوب الحديث المفضل:

```
for (auto& x : container) {
    // process x
}
```

A.3 الأنماط الخوارزمية الأساسية

البحث

```
std::find(begin(c), end(c), v);
std::find_if(begin(c), end(c), pred);
```

الفرز

```
std::sort(begin(c), end(c));
std::stable_sort(begin(c), end(c));
```

التحويل والتجميع

```
std::transform(begin(c), end(c), begin(c), f);
auto sum = std::accumulate(begin(c), end(c), 0);
```

التقسيم

```
auto mid = std::partition(begin(c), end(c), pred);
```

خوارزميات المجموعات (لنطاقات مرتبة)

```
std::set_intersection(a.begin(), a.end(),
                      b.begin(), b.end(),
                      std::back_inserter(out));
```

A.4 Lambdas والمُرشِّحات

```
auto is_even = [](int x){ return x % 2 == 0; };
std::count_if(begin(c), end(c), is_even);
```

تُستخدم مع: find_if, count_if, any_of, all_of, none_of.

A.5 إرشادات الأداء

١. فضّل std::vector للخوارزميات كثيفة العبور.

٢. استخدم reserve() عند معرفة الحجم النهائي.

٣. مرّر الحاويات كـ const&.

٤. استخدم emplace_* بدل push.

٥. فضّل الحاويات غير المرتبة عند عدم الحاجة للترتيب.

٦. استغل move semantics لتجنب النسخ.

٧. استخدم ranges في C++20+.

```
auto result = vec
    | std::views::filter([](int x){ return x % 2 == 0; })
    | std::views::transform([](int x){ return x * x; });
```

A.6 خوارزميات STL عالية القيمة

- std::sort, std::stable_sort
- std::nth_element, std::partial_sort
- std::find, std::find_if
- std::accumulate, std::transform
- idiom erase-remove
- std::lower_bound, std::upper_bound
- std::min_element, std::max_element

الخلاصة

هذا الملحق يلخص المعرفة الأساسية لمطوري الخوارزميات في C++ الحديثة. إتقان الحاويات، والمكررات، وخوارزميات STL، وأنماط الأداء يتيح كتابة كود سريع، واضح، وقابل للصيانة في البحث العلمي والأنظمة عالية الأداء.

الملحق B --- قوالب كود شائعة

يوفر هذا الملحق مجموعة مدمجة وجاهزة للإنتاج من قوالب C++ المستخدمة بكثرة في الخوارزميات، مع ضمانات تعقيد واضحة وقابلة لإعادة الاستخدام.

B.1 اتحاد المجموعات (Disjoint Set) Union

```
struct DSU {
    std::vector<int> parent, size;

    explicit DSU(int n) : parent(n), size(n, 1) {
        for (int i = 0; i < n; ++i) parent[i] = i;
    }

    int find(int x) {
        return parent[x] == x ? x : parent[x] = find(parent[x]);
    }

    bool unite(int a, int b) {
        a = find(a); b = find(b);
        if (a == b) return false;
        if (size[a] < size[b]) std::swap(a, b);
        parent[b] = a;
```

```
        size[a] += size[b];
        return true;
    }

    bool connected(int a, int b) {
        return find(a) == find(b);
    }
};
```

التعقيد: $O(\alpha(n))$ وسطيًا.

B.2 شجرة المقاطع (Segment Tree)

```
template<typename T, typename Merge>
struct SegmentTree {
    int n;
    std::vector<T> tree;
    T id;
    Merge merge;

    SegmentTree(int n_, T identity, Merge m)
        : n(n_), tree(2*n_, identity), id(identity), merge(m) {}

    void build(const std::vector<T>& a) {
        for (int i = 0; i < n; ++i) tree[n + i] = a[i];
        for (int i = n - 1; i > 0; --i)
            tree[i] = merge(tree[i<<1], tree[i<<1|1]);
    }

    void update(int p, T v) {
```

```
        for (tree[p += n] = v; p > 1; p >>= 1)
            tree[p>>1] = merge(tree[p], tree[p^1]);
    }

    T query(int l, int r) {
        T resL = id, resR = id;
        for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
            if (l & 1) resL = merge(resL, tree[l++]);
            if (r & 1) resR = merge(tree[--r], resR);
        }
        return merge(resL, resR);
    }
};
```

الملحق C --- هياكل بيانات متقدمة

C.1 شجرة مقاطع مع Lazy Propagation

```
struct LazySegTree {
    int n;
    std::vector<long long> t, lazy;

    explicit LazySegTree(int n_)
        : n(n_), t(4*n_,0), lazy(4*n_,0) {}

    void push(int v) {
        if (!lazy[v]) return;
        for (int u : {v<<1, v<<1|1}) {
```

```
            t[u] += lazy[v];
            lazy[u] += lazy[v];
        }
        lazy[v] = 0;
    }

    void add(int v, int l, int r,
             int ql, int qr, long long val) {
        if (ql > r || qr < l) return;
        if (ql <= l && r <= qr) {
            t[v] += val;
            lazy[v] += val;
            return;
        }
        push(v);
        int m = (l+r)/2;
        add(v<<1, l, m, ql, qr, val);
        add(v<<1|1, m+1, r, ql, qr, val);
        t[v] = std::max(t[v<<1], t[v<<1|1]);
    }
};
```

C.2 Tree Fenwick موسعة

```
struct FenwickTree {
    int n;
    std::vector<long long> bit;

    explicit FenwickTree(int n_) : n(n_), bit(n_+1,0) {}
```

```
    void update(int i, long long v) {
        for (++i; i <= n; i += i & -i)
            bit[i] += v;
    }

    long long query(int i) const {
        long long s = 0;
        for (++i; i > 0; i -= i & -i)
            s += bit[i];
        return s;
    }
};
```

الملحق D --- قالب CMake و CI

```
cmake_minimum_required(VERSION 3.25)
project(ModernCppAlgorithms LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

add_executable(app src/main.cpp)
target_include_directories(app PRIVATE include)
```

الملحق E --- مراجع مقترحة

- CLRS — Algorithms to Introduction
- Skiena — Manual Design Algorithm The
- Knuth — Programming Computer of Art The
- Williams — Action in Concurrency C++
- Josuttis — Library Standard C++ The

الملحق F --- مخططات حلول ونماذج مخرجات

F.1 مثال DSU

```
DSU d(5);
d.unite(0,1);
d.unite(1,2);
std::cout << d.connected(0,2) << "\n";
```

```
1
```

الخلاصة النهائية

تجمع هذه الملاحق بين قوالب عملية، وهياكل بيانات متقدمة، وأدوات بناء واختبار، ومراجع بحثية، لتشكّل مرجعًا مهنيًا متكاملاً لمطوري الخوارزميات باستخدام C++ الحديثة.

المراجع المعتمدة

تم إعداد هذا الكتاب اعتماداً على مجموعة منتقاة من المراجع الأكاديمية والمهنية الموثوقة التي تمثل الأساس النظري والتطبيقي لعلوم الخوارزميات، وهياكل البيانات، والأداء، والتزامن، والتطبيقات العملية في C++ الحديثة. تم اختيار هذه المراجع بناءً على قيمتها العلمية طويلة الأمد وكونها مصادر أصلية وليست محتوىً مشتقًا أو مبسطًا.

أولاً: مراجع تصميم وتحليل الخوارزميات

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein *Introduction to Algorithms*, MIT Press.
- Steven S. Skiena *The Algorithm Design Manual*, Springer.
- Robert Sedgewick, Kevin Wayne *Algorithms*, Addison-Wesley.
- Donald E. Knuth *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley.

ثانيًا: هياكل البيانات المتقدمة

- Peter Brass *Advanced Data Structures*, Cambridge University Press.

- Sartaj Sahni *Data Structures, Algorithms, and Applications in C++*, McGraw-Hill.
- Mehta, Sahni (Editors) *Handbook of Data Structures and Applications*, Chapman & Hall/CRC.
- Robert Tarjan *Data Structures and Network Algorithms*, SIAM.

ثالثًا: الخوارزميات العشوائية والاحتمالية

- Rajeev Motwani, Prabhakar Raghavan *Randomized Algorithms*, Cambridge University Press.
- Michael Mitzenmacher, Eli Upfal *Probability and Computing*, Cambridge University Press.

رابعًا: خوارزميات الرسوم البيانية

- Ahuja, Magnanti, Orlin *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall.
- Gross, Yellen *Graph Theory and Its Applications*, Chapman & Hall.
- Robert Tarjan Papers on DFS, SCCs, and linear-time graph algorithms.

خامسًا: الأداء، الذاكرة، والمعالجات

- Ulrich Drepper *What Every Programmer Should Know About Memory*.
- John L. Hennessy, David A. Patterson *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann.
- Agner Fog *Optimizing Software in C++*.

سادسًا: التزامن والبرمجة المتوازية

- Anthony Williams *C++ Concurrency in Action*, Manning.
- Maurice Herlihy, Nir Shavit *The Art of Multiprocessor Programming*, Morgan Kaufmann.
- Herb Sutter *The Free Lunch Is Over*.

سابعًا: C++ الحديثة والمكتبة القياسية

- Nicolai M. Josuttis *The C++ Standard Library*, Addison-Wesley.
- Bjarne Stroustrup *A Tour of C++*, Addison-Wesley.
- ISO/IEC JTC1/SC22/WG21 *ISO C++ Standard (C++17, C++20, C++23 Drafts)*.
- cppreference.com C++ Standard Library Reference.

ثامنًا: الاختبار، القياس، وإعادة الإنتاج

- Google Benchmark Documentation
- GoogleTest Documentation
- LLVM Sanitizers Documentation
- McCool, Reinders, Robison *Structured Parallel Programming*.

ملاحظة ختامية

تم بناء محتوى هذا الكتاب اعتمادًا على هذه المراجع بشكل مباشر أو غير مباشر، سواء من حيث:

- الأسس النظرية للخوارزميات
- أنماط هياكل البيانات
- نماذج الأداء والذاكرة
- ممارسات C++ الحديثة
- منهجية البحث والتجربة القابلة لإعادة الإنتاج

وقد تمت صياغة المادة بأسلوب تعليمي وتطبيقي مستقل، مع الحفاظ على الأمانة العلمية واحترام المصادر الأصلية.