خوارزميات البحث و الترتيب

```
pattern_tree *acbm_init(pattern_data
*patterns, int npattern);
int acbm_search(pattern_tree *ptree, unsigned
char *text, int text_len, unsigned int
matched_indexs[], int nmax_index);

int _tmain(int argc, char* argv[])
{
    pattern_data *patterns = NULL;
    int npattern = argc -2;
    ...
/* read app args, file length, and other
  initialization operati ons. */

    ptree = acbm_init(patterns, npattern);

    matched = acbm_search(ptree, (unsigned
char *)text, textLength, matched_indexs,
nmax_index);
    printf("total match is %d\n", gMatched);
    return 0;
}
```

إعداد
أحمد الشنقيطي

الحمد لله الذي بحمده يُستفتح كل كتاب و بذكره يُصدر كل خطاب وبفضله يتنعم أهل النعيم في دار الجزاء و الثواب والصلاة و السلام على سيد المرسلين و إمام المتقين المبعوث رحمة للعالمين محمد ابن عبد الله الصادق الأمين و على صحابته الأخيار و من تبعهم بإحسان إلى يوم الدين أما بعد :

في هذا الكتاب, سأضع بين أيديكم شرحا لأهم الخوارزميات المُستعملة في البحث و الترتيب و كلي أمل بأن يستفيد الجميع.

إذا رأيت أي خطأ أو تقصير في الكتاب فاعلم أن ذلك من نفسي و من الشيطان فالكمال لله و حده عز و جل.

إهداء :

أهدي هذا الكتاب إلى منتديات "**الفريق العربي للبرمجة**" من زوار وأعضاء ومشرفين و خبراء وأخص بالذكر منهم الأستاذ الفاضل **وجدي عصام**.

تم الانتهاء من الإصدار الأول من هذا الكتاب بتاريخ 23/07/2012

عـن الـمؤلـف:

الإسم: أحمد/محمد

اللقب: الشنقيطي

سنة الميلاد: 1992

الدولة: بلاد شنقيط و أرض المليون شاعر .. موريتانيا

الهواية: Security & programming

المستوى الأكاديمي: خريج كلية العلوم و التقنيات.

للتواصل: ahmed.ould_mohamed@yahoo.fr

جميع الحقوق محفوظة © All rights reserved

فهرس الكتاب

المقدمة 6

I - خوارزمية البحث الخطي (Linear search algorithm) 7

I.1 - الخوارزمية (الهدف, الفكرة, النتيجة, الإيجابيات و السلبيات) 7

I.2 - الخوارزمية بلغة السي++ 7

I.3 - التعقيد الزمني 10

I.4 - ملأ مصفوفة عشوائيا و اختبار سرعة الخوارزمية 11

I.5 - كم تحتاج هذه الخوارزمية من الوقت لتنفيذ مهمتها ؟ 12

I.6 - اختبر قدراتك ! 14

II - خوارزمية ترتيب الفقاعات (Bubble sort algorithm) 15

II.1 - الخوارزمية (الهدف, الفكرة, النتيجة, الإيجابيات و السلبيات) 15

II.2 - قبل أن نبدأ .. إليك 4 طرق لعمل Swap 16

II.2.1 - الطريقة الأولى 16

II.2.2 - الطريقة الثانية 17

II.2.3 - الطريقة الثالثة 17

II.2.4 - الطريقة الرابعة 18

II.2.5 - إبحث عن أفضل طريقة لعمل Swap 19

II.3 - كتابة دالة تقوم بعملية التبديل 19

II.4 - الخوارزمية بلغة السي++ 22

II.5 - التعقيد الزمني 27

II.6 - ملأ مصفوفة عشوائيا و اختبار سرعة الخوارزمية 27

III - خوارزمية البحث الثنائي (Binary search algorithm)............................29
III.1 - الخوارزمية (الهدف, الفكرة, النتيجة, الإيجابيات و السلبيات)............................29
III.2 - الخوارزمية بلغة السي++ 29
III.2.1 - الطريقة الأولى...30
III.2.2 - الطريقة الثانية 31
III.3 – أمثلة على الخوارزمية 31
III.3.1 – المثال الأول..32
III.3.2 – المثال الثاني...33
III.3.3 – المثال الثالث...34
III.4 – التعقيد الزمني 36
III.5 – ملأ مصفوفة عشوائيا و اختبار السرعة 36
III.6 – اختبر قدراتك !...38
الخاتمة 41

الـمـقـدمـة

تُعتبر خوارزمية البحث ركنا أساسيا من أركان علم الخوارزميات و تتخذ هذه الخوارزمية عدة أشكال, من أبسطها البحث عن عدد في مصفوفة مُحددة الحجم و يزداد الأمر تعقيدا عند الانتقال إلى البحث عن كلمة داخل نص, تماما كما ترى في محررات النصوص العادية والتي تحتوي على خاصية Find & Replace حيث أن أغلب المحررات الحالية تستخدم خوارزميه بحث تسمى Boyer-Moore Searching التي تُعد تقريبا من أسرع الخوارزميات في مجال البحث. هناك أيضا بحث من نوع آخر, فماذا لو كانت لدينا مجموعة حروف و نريد إيجاد جميع الكلمات التي تبدأ بهذه الحروف ؟؟ عادة ما يُسمى هذا النوع من الخوارزميات بـ prefix searching و لهذا النوع تطبيقات كثيرة خصوصا في محركات البحث و القواميس و المتصفحات التي تستخدم هذه الخوارزمية عند كتابتك لموقع يبدأ بحرف كنت قد زرته سابقا.

لا تقتصر خوارزمية البحث على ما ذكرناه آنفا, فهناك نوع آخر من الخوارزميات يُستخدم لإيجاد نص قريب من النص الذي كنت تبحث عنه, حيث تقوم الخوارزمية بالبحث عن 4 أو 5 كلمات قريبة من الكلمات الخاطئة, تماما كما يفعل Google عند الترجمة أو Office Word عند كتابة نص يحتوي على أخطاء إملائية و تعتمد أغلب هذه الخوارزميات على الوزن الصوتي للحرف و من أشهرها Soundex Searching.

ليس هذا فقط، فمضادات الفيروسات تستخدم خوارزميات بحث سريعة للبحث عن وجود توقيع مطابق لأحد بيانات الملف المراد فحصه، وبما أن قواعد بيانات التواقيع تكون ضخمة للغاية فالبحث عن كل توقيع سيكون بطيئا جدا وهناك الأفضل، حيث توجد خوارزميات بإمكانها البحث عن عده تواقيع في نفس اللحظة وتستخدم بنية شجرية للقيام بهذا الأمر، هناك أيضا خوارزميات أخرى تعتمد على مفاهيم مختلفة مثل Hash Table وعدة أمور أخرى، و يُسمى هذا النوع من الخوارزميات بـ Multiple pattern searching و هو من أصعب الخوارزميات، وهناك العديد من مضادات الفيروسات التي تستخدم مثل هذه الخوارزميات مثل ClamAV .

I – خوارزمية البحث الخطي (Linear search algorithm)

I.1 – الخوارزمية (الهدف, الفكرة, النتيجة, الإيجابيات و السلبيات)

الهدف : البحث عن قيمة المفتاح key داخل المصفوفة X.

الفكرة : تعتمد هذه الخوارزمية على البحث التسلسلي (Sequential Search) في المصفوفة X حيث يبدأ البحث من أول عنصر إلى أن تنتهي المصفوفة, و في كل مرة نقارن محتوى الخانة الحالية X[i] مع المفتاح Key, فإذا كانت القيمتان متساويتان (X[i] = Key) ستتم إعادة قيمة المتغير i الذي يُمثل مكان وجود المفتاح Key في المصفوفة X, أما إذا كانت القيمتان مختلفتان فسننتقل إلى الخانة الموالية X[i+1] للبحث من جديد و هكذا.

النتيجة : إذا كان Key موجود في X فسنحصل على رقم الخانة التي يوجد بها المفتاح و إلا فالقيمة المُعادة ستكون 1-.

الإيجابيات : الخوارزمية **بسيطة** و **تقليدية** أيضا كما لا يُشترط الترتيب عند البحث.

السلبيات : **بطيئة** و **غير عملية**, خصوصا عند معالجة المصفوفات الضخمة.

I.2 – الخوارزمية بلغة السي++

```
1  int linearSearch(int X[], int length, int Key){ //دالة البحث
2  /*نستعمل هذه الحلقة للمرور على جميع عناصر المصفوفة*/
3      for(int i = 0;i < length; i++)
4          if(X[i] == Key)//هل الخانة الحالية مساوية للمفتاح ؟
5             return i;//إعادة رقم الخانة التي يوجد بها المفتاح
6      return -1;//لم يتم العثور على المفتاح
7  }
```

الدالة linearSearch تستقبل 3 وسائط, الأول هو اسم المصفوفة المراد البحث داخلها و الوسيط الثاني هو طول المصفوفة أما الوسيط الثالث فهو المفتاح الذي نبحث عنه. بكل بساطة قمنا باستخدام حلقة for للمرور على جميع عناصر المصفوفة (لاحظ أن البحث خطي) واضعين بذلك شرط الحلقة

كما يلي : إذا كانت قيمة الخانة الحالية مُساوية لقيمة المفتاح فقم بإعادة رقم الخانة. (مفهوم إعادة القيمة يرتبط بالخروج من الدالة)

طيب ماذا عن الحالة العكسية ؟ أقصد إذا كان المفتاح غير موجود في المصفوفة ؟ هنا تأتي فائدة إعادة 1-. يُمكنك إبدال 1- بأي عدد سالب تماما, المهم أن لا يكون موجبا أو معدوما حتى لا يقع التباس بين القيمة التي تدل على عدم وجود المفتاح و رقم الخانة التي يوجد بها الأخير.

نأتي الآن إلى الدالة الرئيسية main :

```
#include<iostream>
using namespace std;
int linearSearch(int X[], int length, int Key){ //دالة البحث
    for(int i = 0;i < length; i++)//نستعمل هذه الحلقة للمرور على جميع عناصر المصفوفة
        if(X[i] == Key)//هل الخانة الحالية مساوية للمفتاح ؟
           return i;//إعادة رقم الخانة التي يوجد بها المفتاح
    return -1;//لم يتم العثور على المفتاح
}
int main(){
    int xInt[] = {3,-1,0,17,9};//االمصفوفة التي سيتم البحث داخلها
    int key;//هذا المتغير سيحمل قيمة المفتاح
    cout<<"Entrez la valeur de clé : ";//أدخل قيمة القمتاح
    cin>>key;//تخزين قيمة المفتاح في هذا المتغير
    int found = linearSearch(xInt,5,key);//البحث عن قيمة المفتاح داخل المصفوفة السابقة
    if( found == -1)//هل القيمة غير موجودة ؟
      cout<<"La valeur n'est pas trouvée !";//لم يتم العثور على المفتاح داخل المصفوفة
    else//هل القيمة موجودة ؟
      cout<<"La clé se trouve dans la case numéro "<<found + 1;//إظهار رقم الخانة التي يوجد المفتاح داخلها
    return 0;
}
```

في البداية قمنا بالإعلان عن مصفوفة باسم xInt عدد عناصرها 5 ثم قمنا أيضا بالإعلان عن متغير باسم key حيث طلبنا من المستخدم إدخال قيمة للمتغير. لننتقل إلى الجزء الأهم في الـ main و هو تطبيق الدالة linearSearch على المصفوفة xInt و المتغير key :

تم الإعلان عن متغير جديد باسم found الذي يحوي القيمة المُعادة من طرف الدالة. بعد ذلك قمنا بفحص قيمة found فإذا كانت قيمته مساوية لـ 1- فهذا يعني أن المفتاح غير موجود في المصفوفة و إلا فالمفتاح موجود في الخانة رقم found+1 لأن الترقيم يبدأ من صفر فالخانة رقم 0 هي الخانة الأولى و هكذا.

الوسيط الثاني للدالة يُمثل الطول الفعلي للمصفوفة لذا لا تحاول أن تُرسل إلى الدالة عدد أكبر من طول المصفوفة لأن هذا الفعل يؤدي إلى حدوث الـ buffer overflow الذي ينتج عنه انتهاك لتسيير الذاكرة و قد يتوقف برنامجك. يمكنك إرسال عدد أصغر من طول المصفوفة إذا كنت ترغب في أن يقتصر البحث على جزء من المصفوفة.

إذا وُجد المفتاح أكثر من مرة في المصفوفة X فإن الدالة ستعيد رقم أول خانة يظهر فيها المفتاح.عليك الآن بتغيير الدالة لتعيد رقم آخر خانة يُوجد بها المفتاح.

ماذا لو أردنا معرفة عدد المرات التي ظهر فيها المفتاح داخل المصفوفة ؟ لنفترض أن المصفوفة كبيرة جدا و سيتم البحث داخل مصفوفة جزئية من المصفوفة الرئيسية. إذا كل ما سنقوم به هو إضافة متغير جديد إلى جسم الدالة (أو لنقل عداد) و زيادة هذا العداد كلما وجدنا خانة مساوية للمفتاح هكذا :

```
1  int linearSearch(int X[], int inf, int sup, int Key){ //دالة البحث
2      int found = 0;//هذا المتغير سيحوي عدد المرات التي يوجد فيها المفتاح داخل المصفوفة
3      for(int i = inf;i < sup; i++)//نستعمل هذه الحلقة للمرور على جميع عناصر المصفوفة
4          if(X[i] == Key)//هل الخانة الحالية مساوية للمفتاح ؟
5              found++;;//زيادة العداد بواحد
6      if(found >= 1)//هل يوجد المفتاح على الأقل مرة واحدة ؟
7          return found;
8      else
9          return -1;//لم يتم العثور على المفتاح
10 }
```

في هذه الحالة ستستقبل الدالة أربع وسائط هي : اسم المصفوفة الجزئية و من أين تبدأ و أين تنتهي ؟ إضافة إلى المفتاح.

كالعادة نبدأ بالمرور على جميع عناصر المصفوفة و كل ما وجدنا خانة مُساوية للمفتاح أضفنا 1 إلى قيمة العداد. عند نهاية الحلقة, إذا كانت قيمة العداد أكبر أو تساوي واحد فإن المفتاح يوجد على الأقل مرة واحدة داخل المصفوفة و هنا ستتم إعادة قيمة المتغير found و في حالة العكس ستتم إعادة 1-.

```
#include<iostream>
using namespace std;
int linearSearch(int X[], int inf, int sup, int Key){ //دالة البحث
    int found = 0;//هذا المتغير سيحوي عدد المرات التي يوجد فيها المفتاح داخل المصفوفة
    for(int i = inf;i < sup; i++)//نستعمل هذه الحلقة للمرور على جميع عناصر المصفوفة
        if(X[i] == Key)//هل الخانة الحالية مساوية للمفتاح ؟
            found++;;//زيادة العداد بواحد
    if(found >= 1)//هل يوجد المفتاح على الأقل مرة واحدة ؟
        return found;
    else
        return -1;//لم يتم العثور على المفتاح
}
int main(){
    int xInt[] = {3,-1,0,17,9,20,17,75,17,28,71,4,17,5,-8,94,87,21};//االمصفوفة التي سيتم البحث داخلها
    int key;//هذا المتغير سيحمل قيمة المفتاح
    cout<<"Entrez la valeur de clé : ";//أدخل قيمة القمتاح
    cin>>key;//تخزين قيمة المفتاح في هذا المتغير
    int found = linearSearch(xInt,5,11,key);//البحث عن قيمة المفتاح داخل المصفوفة السابقة
    if( found == -1)//هل القيمة غير موجودة ؟
        cout<<"La valeur n'est pas trouvée !";//لم يتم العثور على المفتاح داخل المصفوفة
    else//هل القيمة موجودة ؟
        cout<<"la valeur se trouve dans le tableau "<<found<<" foi(s).";//عدد المرات التي يوجد فيها المفتاح داخل المصفوفة
    return 0;
}
```

في الدالة الرئيسية قمنا بالإعلان عن مصفوفة تحتوي على 18 عنصر بينما قمنا بالبحث عن القيمة 17 داخل مصفوفة جزئية تتكون من 6 عناصر فقط (المصفوفة الجزئية تبدأ من الخانة السادسة و حتى الثانية عشر). برأيك ما مُخرجات البرنامج ؟

I.3 – التعقيد الزمني

كما ذكرنا آنفا فإن هذه الخوارزمية تعتمد على البحث الخطي أو المتسلسل حيث يبدأ البحث من أول خانة منتقلا إلى الخانة المجاورة إن لم يجد المفتاح و هكذا حتى الوصول إلى الخانة الأخيرة (مما يعني أن الزمن المستغرق لتنفيذ هذه الخوارزمية يزيد كلما زاد حجم المصفوفة), إذا خوارزمية البحث الخطي لها تعقيد زمني $T(n) = n = O(n)$.

في أفضل الحالات يمكن أن نجد المفتاح في أول خانة $T(n) = 1 = O(1)$, و في المتوسط $T(n) = \frac{n}{2} = O\left(\frac{n}{2}\right)$ و في أسوء الحالات يمكن أن نجد المفتاح في الخانة الأخيرة كما يمكن أن يكون هذا الأخير غير موجود.

كمثال للتوضيح لنفترض الجدول الآتي :

41	5	-1	18	7	62	39	0	-6	28

إذا قمنا بتطبيق خوارزمية البحث الخطي على الجدول فسنمر بالخطوات التالية :

41	5	-1	18	7	62	39	0	-6	28
41	**5**	-1	18	7	62	39	0	-6	28
41	5	**-1**	18	7	62	39	0	-6	28
41	5	-1	**18**	7	62	39	0	-6	28
41	5	-1	18	**7**	62	39	0	-6	28
41	5	-1	18	7	**62**	39	0	-6	28
41	5	-1	18	7	62	**39**	0	-6	28
41	5	-1	18	7	62	39	**0**	-6	28
41	5	-1	18	7	62	39	0	**-6**	28

يوجد المفتاح في الخانة ذات اللون الأبيض و الخلفية السوداء و في كل مرة تتم مقارنة الخانة ذات اللون الأحمر و الخلفية البيضاء مع المفتاح. لاحظ أنه من خلال البحث عن المفتاح تم تشكيل قطر لمثلث قاعدته عبارة عن المصفوفة التي تبدأ بأول عنصر و تنتهي بالمفتاح.

I.4 – ملأ مصفوفة عشوائيا و اختبار سرعة الخوارزمية

قمت بإدراج هذه الفقرة لكي ألفت انتباهك أخي القارئ إلى مفهوم التعقيد الزمني من الناحية العملية. لذا سنقوم بالإعلان عن مصفوفة (كبيرة الحجم نسبيا), تحتوي على مائة ألف خانة ثم نقوم بملأ هذه المصفوفة بشكل عشوائي و من ثم نستدعي دالة البحث للبحث عن المفتاح.

بكل بساطة, توجد حالتان, إما أن يكون المفتاح غير موجود داخل المصفوفة أو العكس. الحالة الأولى تُمثل أسوأ حالات الخوارزمية أما الحالة الثانية فقد تمثل أفضل الحالات أو الحالة المتوسطة. ما يهمنا في هذه الفقرة هو عدد الخانات التي مرت بها الدالة قبل الوصول إلى المفتاح أو لنقل عدد المقارنات اللازمة للحصول على الخانة المساوية لقيمة المفتاح.

إذا لم يتم العثور على المفتاح فهذا يعني أن عدد المقارنات يساوي عدد عناصر المصفوفة أما إذا تم العثور على المفتاح فهذا يعني أن عدد المقارنات يساوي رقم الخانة التي يوجد بها المفتاح + 1 لأن ترقيم عناصر المصفوفة يبدأ من صفر.

```cpp
#include<iostream>
#include<ctime>
#include<cstdlib>
using namespace std;

int linearSearch(int X[], int length, int Key) {
    for (int i = 0; i < length; i++)
        if (X[i] == Key)
            return i;
    return -1;
}

int main() {
    const int SIZE = 100000;
    int xInt[SIZE];
    int key = 7;
    srand(time(NULL));
    for (int i = 0; i < SIZE; i++)
        xInt[i] = rand() % 100000;
    int found = linearSearch(xInt, SIZE, key);
    if (found == -1)
        cout << "La valeur n'est pas trouvée !" << endl << "nombre de comparaisons : " << SIZE << endl;
    else
        cout << "La valeur a été trouvé" << endl << "nombre de comparaisons : " << found + 1 << endl;
    system("pause");
    return 0;
}
```

- تكرار الأعداد العشوائية المُولدة يعتمد على عدة عوامل منها سعة المجال الذي تم توليد الأعداد العشوائية بداخله.
- سبق و أن كتبتُ موضوع يتحدث عن توليد الأعداد العشوائية بشيء من التفصيل, يمكنك مراجعة الموضوع من هنا : http://www.arabteam2000-forum.com/index.php?showtopic=217773

I.5 – كم تحتاج هذه الخوارزمية من الوقت لتنفيذ مهمتها ؟

قد يدور في ذهنك السؤال التالي : هل نستطيع رصد الوقت الذي تستغرقه هذه الخوارزمية ؟

و الإجابة هي نعم, فهناك دوال API تمكننا من الحصول على فترات توقيت أقل من 1 ميلي ثانية،

لكن ولسوء الحظ فإن هذه الدوال تعتمد بشكل كامل على النظام و خصوصا المعالج المركزي. يجب أن يدعم النظام ما يسمى بـ عداد الأداء عالي الدقة (High Resolution Performance Counter) حيث تعتمد دقة هذا العداد على المعالج المركزي .توجد دالتان من دوال الــ **API** تستخدمان لهذا الغرض :

الدالة الأولى هي QueryPerformanceFrequency حيث تعطينا تردد العداد أو لنقل عدد الدورات في الثانية الواحدة و تعيد صفراً إذا كان النظام لا يدعم العداد العالي الدقة. أما الدالة الثانية فهي QueryPerformanceCounter التي تعيد القيمة الحالية للعداد. بهذا يمكننا تقييم المدة التي يأخذها تنفيذ جزء معين من الكود و ذلك باستدعاء الدالة QueryPerformanceCounter مباشرة قبل الجزء الذي نريد تقييم مدته الزمنية ثم استدعائها مرة أخرى مباشرة بعده، ثم نحسب الفرق و نقسمه على التردد فنحصل على الوقت الذي استغرقه تنفيذ ذلك الجزء من الكود. لنفرض مثلاً أن التردد كان 50000 دورة في الثانية، و أن القيمة التي حصلنا عليها قبل الكود هي 1500 و القيمة بعد الكود هي 3500, الفرق هو 2000، و بالقسمة على 50000 نحصل على 0.04 ثانية.

```
void mesure() {
    LARGE_INTEGER t0, t1, freq;
    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&t0);
    found = linearSearch(xInt, key);
    QueryPerformanceCounter(&t1);
    cout << "Temps pris par la fonction " << double(t1.QuadPart - t0.QuadPart) / freq.QuadPart << "s" << endl;
}
```

يمكنك أيضا استخدام timeGetTime أو GetTickCount إذا أردت معرفة التوقيت بدقة أقل. يمكنك بهذه الإجراءات قياس الفروق الزمنية بدقة ميللي ثانية واحدة على الأكثر، وهذه الدقة قد تكون كافية لقياس عمل الخوارزمية التي نتحدث عنها. بهذه الطريقة ستريح نفسك من الصداع الذي تسببه لك المؤقتات الأخرى مثل QPC وRDTSC. المؤقتة QPC يستخدم ما يحلو له في الجهاز ليقدم أكبر دقة ممكنة. قد يعتمد في بعض الأجهزة على تعليمات مباشرة للـ BIOS، وفي بعض الحالات قد يقرأ القيم من أماكن أخرى غريبة. لذا ليست له وحدة ثابتة تستطيع استخدامها، فعندما تنادي الدالة QueryPerformanceCounter ستستقبل قيمة .. الله أعلم ماذا تعني !! قد تكون الدورة هي معدل نبض ساعة المعالج Processor Clock وقد تعني أي شيء آخر. لذلك أنت بحاجة إلى شيء ما يساعدك على تحويل هذه القيمة إلى وحدة زمنية كالثواني, وهنا تأتي الإجراء فائدة الدالة

QueryPerformenceFrequency حيث تعيد عدد "الوحدات" في الثانية الواحدة.

مثلاً : أعطتنا QueryPerformanceFrequency القيمة 15000000 (15 مليون وحدة في الثانية)، وسنسميها التردد. وقمنا باستدعاء QueryPerformanceCounter مرتين على فترتين متباعدتين وطرحنا الفرق لنحصل على 45000000. الآن يمكننا أن نقسم هذه القيمة على التردد لنحصل على 3 ثواني. وهذا هو مبدأ عمل المؤقت QPC الذي تصل دقته إلى النانوثانية، ويستخدم في تزمين الصوت مع الصورة في مشغلات الأفلام كما يُستخدم في قياس الزمن المستغرق لتنفيذ الإجراءات السريعة في سي++.

إلا أن هناك بعض الأمور التي تمنع QPC من أن يكون مؤقتاً مثاليا, تتخلص هذه الأمور في شيئين : تعدد المعالجات، والمعالجات التي تغير سرعتها بشكل مستمر. في حالة تعدد المعالجات، لو اعتمد QPCعلى عداد التعليمات في المعالج (وهي الحالة الأكثر شيوعاً) فإنك قد تحصل على نتائج غريبة بين النداءات المختلفة للإجراء، وذلك بحسب أي المعالجات قام بتنفيذ الطلب. فمثلاً المعالج الأول أنجز مئة مليون تعليمة حتى الآن، بينما الثاني أنجز 90 مليون تعليمة فقط، ولذلك فقد تحصل أحياناً على نتائج مضحكة ! كأن يعود الزمن إلى الوراء (يكون الفرق سالبا) أو يقفز قفزات كبيرة مفاجئة للأمام.

في الحالة الثانية فإن بعض المعالجات تغير سرعتها بشكل مستمر على حسب الطاقة المتوفرة، وهذه المعالجات تتواجد بشكل رئيسي في الأجهزة المحمولة laptops حيث يخفف المعالج من سرعته عندما يدخل في نظام توفير طاقة البطارية.. وهنا فإن قيمة التردد التي حسبناها مسبقاً تصبح خاطئة لأن التردد قد تغير ويجب قراءته مرة أخرى ...!

I.6 – اختبر قدراتك !

لدينا مؤسسة صغيرة تحتوي على مجموعة من العمال, كل عامل مُعرف برقم ID واسم و مرتب الشهر, عند تشغيل البرنامج سيُدخل المستخدم اسم العامل و كلمة المرور ثم يقوم البرنامج بالبحث داخل بيانات العمال, فإذاكان اسم العامل موجود و كلمة المرور صحيحة فسيتم إظهار رسالة ترحيبية بالإضافة إلى المرتب الشهري للعامل و إلا فسيُظهر البرنامج رسالة تفيد بأن هذا العامل غير مُسجل.

II – خوارزمية ترتيب الفقاعات (Bubble sort algorithm)

كان من المفترض أن نتحدث في هذه الوحدة عن خوارزمية الترتيب الثنائي و لكن أحببت أن أبدأ بإحدى خوارزميات الترتيب نظرا لأن الـ Binary search algorithm يعتمد على فكرة الترتيب.

في هذه الوحدة (أو لنقل بقية أجزاء الكتاب) ستجدني أتحدث أحيانا مع شخص افتراضي اسمه عمرو, ألجأ إلى الحديث مع هذا الشخص عندما أصل إلى فقرة تحتاج إلى توضيح أكثر.

II.1 – الخوارزمية (الهدف, الفكرة, النتيجة, الإيجابيات و السلبيات)

الهدف : ترتيب عناصر المصفوفة X تصاعديا أو تنازليا. (في بقية الدرس سنعتبر أن الترتيب تصاعدي و n هو عدد عناصر المصفوفة X)

الفكرة : تقارن هذه الخوارزمية بين قيم الخانات المتجاورة, تبدأ بالمقارنة بين أول خانتين من المصفوفة, إذا كان محتوى الخانة الأولى أكبر من محتوى الخانة الثانية سيتم تبادل محتوى الخانتين و هكذا مع بقية الخانات. عند انتهاء الدورة الأولى ستكون الخوارزمية قد أنجزت n-1 مقارنة و بهذا ستتم إزاحة أكبر عناصر المصفوفة إلى الخانة الأخيرة. بقي الآن ترتيب الـ n-1 عنصر. و هكذا الأمر مع بقية الدورات ...

النتيجة : عملية الترتيب تختلف عن عملية البحث في هذه النقطة, فالترتيب لا يقبل وجود عدة احتمالات في النتيجة, فعند تطبيق الخوارزمية سنحصل على مصفوفة مرتبة تصاعديا و بالتالي لا توجد احتمالات للمناقشة.

الإيجابيات : الخوارزمية سهلة التصور و بسيطة المفهوم.

السلبيات : **بطيئة** شيئا ما و**غير عملية**, خصوصا عند معالجة المصفوفات الضخمة.

II.2 – قبل أن نبدأ .. إليك 4 طرق لعمل Swap

من خلال دراستك لخوارزميات الترتيب المختلفة ستجد أن مفهوم **Swap** أو **تبادل** مُحتوى متغيرين يتكرر باستمرار .. فأغلب خوارزميات الترتيب (إن لم تكن كلها !) تعتمد في مرحلة ما على تبديل محتوى خانتين لوضعهما في الترتيب الصحيح, لذا سنوضح كيفية تبديل محتوى متغيرين قبل أن نبدأ بكتابة الخوارزمية بلغة السي++.

سنتناول فيما يلي 4 طرق لعمل **Swap**, لكل منها إيجابيات كما لها سلبيات أيضا.

II.2.1 – الطريقة الأولى

و هي الأكثر شهرة, تكمن في استخدام وسيط مساعد يُساعدنا على تبديل محتوى المتغيرين, فنضع قيمة المتغير الأول في الوسيط ثم نضع قيمة المتغير الثاني في المتغير الأول ثم نضع قيمة الوسيط في المتغير الثاني, و بهذا نكون قد بادلنا بين قيمة المتغير الأول و الثاني. لنفرض أن x=5 و y=-1 :

```
t = x //t = 5
x = y //x = -1
y = t //y = 5
```

في النهاية تكون x=-1 و y=5 . و بالتالي تم تبديل قيمتي x و y.

يمكنك تخيل الفكرة كما يلي .. لدينا كأسين x و y, الكأس x يحتوي على العصير و الكأس y يحتوي على الماء, لو أردنا أن نُبادل محتوى الكأسين, (أي نجعل الماء في الكأس x و العصير في الكأس y) سنقوم باستخدام كأس ثالث فارغ (وهو الوسيط المساعد t) و تكون عملية التبادل كما يلي :

1. نجعل العصير في الكأس الفارغ.

2. نجعل الماء في الكأس x.

3. نجعل العصير في الكأس y.

الآن أصبح الكأس x يحتوي على الماء و الكأس y يحتوي على العصير. هذه الطريقة بسيطة و تقليدية أيضا, إلا أنها تأخذ مساحة أكثر من الذاكرة بالإعلان عن الوسيط الثالث.

يمكننا عمل Swap بدون حاجة إلى وسيط مساعد عن طريق إجراء بعض العمليات الحسابية البسيطة على المتغيرين.

II.2.2 – الطريقة الثانية

تصلح للمتغيرات العددية فقط

```
x=x+y;
y=x-y;
x=x-y;
```

II.2.3 – الطريقة الثالثة

تصلح للمتغيرات العددية فقط و يجب أن تختلف قيمة y عن الصفر

```
x=x*y;
y=x/y;
x=x/y;
```

الطريقتان السابقتان يمكنهما التسبب في حدوث Overflow إذا كانت قيمة x+y أو x*y أكبر من القيمة العظمى لنوع المتغيرين x,y.

II.2.4 – الطريقة الرابعة

توجد أيضا طريقة رابعة لا تحتاج إلى وسيط مساعد و لا يمكن أيضا أن تكون سببا في حدوث Overflow. تعتمد هذه الطريقة على أحد مؤثرات الـ Bit (Bitwise) وهو المؤثر XOR (Exclusive OR) و الذي يُرمز له بـ ^ , هذا المؤثر يعطينا True إذا و فقط إذا كان المدخلين مختلفين, أما إذا كانا متشابهين فالنتيجة ستكون False و يعمل هذا المؤثر كالتالي :

```
/*
 True = 1
 False = 0
*/
1 ^ 1 = 0
0 ^ 0 = 0
1 ^ 0 = 1
0 ^ 1 = 1
```

و كمثال على ذلك :

```
/*
 True = 1
 False = 0
*/
    15 ^ 8 = 7
    00001111  // 15 in Binary
^
    00001000//8 in Binary
=   00000111// 7 in Binary
```

سنقوم بتجربة هذه الطريقة على المتغيرين x و y حيث x = 15 و y=7, لذلك سنقوم بالخطوات الثلاثة التالية :

```
x = x^y // x = 15 XOR 7 = 8
y = x^y//y = 8 XOR 15 = 7
x = x^y//x = 7 XOR 8 = 15
```

و بهذا تم تبديل محتوى المتغيرين x و y.

II.2.5 – إبحث عن أفضل طريقة لعمل Swap

يمكننا الآن كتابة دالة تبادل بين مُحتوى مُتغيرين باستخدام إحدى الطرق الأربعة الموضحة أعلاه, مع الأخذ في الاعتبار سلبيات كل طريقة. إذا أدرت التوسع أكثر في هذه الجزئية, فيمكنك البحث عن أفضل طريقة لتبديل محتوى متغيرين بحيث تحقق النقاط الآتية في آن واحد :

- ✓ لا تحتاج إلى وسيط مساعد. (المستوى : بسيط)
- ✓ لا تسبب Overflow. (المستوى : صعب)
- ✓ لا تضع أي شرط على قيم أو نوع المتغيرين. (المستوى : أكثر صعوبة, إن لم يكن مستحيلا !)

II.3 – كتابة دالة تقوم بعملية التبديل

بعد أن تطرقنا إلى الطرق الأربعة التي يمكننا من خلالها تبديل محتوى متغيرين. سنقوم الآن بكتابة دالة تُبادل بين محتوى خانتين من المصفوفة X باستخدام الطريقة الأولى (لأنها الأكثر شعبية .. رغم احتوائها على انتهاك صارخ لحقوق الـ Overflow !) :

```
1  void Swap(int *X, int i, int j)
2  {
3      int temp;
4      temp = *(X + i);
5      *(X + i) = *(X + j);
6      *(X + j) = temp;
7  }
```

عمرو : لم أفهم استخدام المؤشرات !

أحمد : لا بأس !, سترتاح قليلا عندما تعلم أن الكتابة السابقة مُطابقة تماما للكتابة الآتية :

```
1  void Swap(int X[], int i, int j)
2  {
3      int temp;
4      temp = X[i];
5      X[i] = X[j];
6      X[j] = temp;
7  }
```

عمرو : كيف !؟

أحمد : الدالة السابقة تستقبل 3 وسائط, الأول عبارة عن اسم المصفوفة و الوسيطين, الثاني و الثالث عبارة عن أرقام الخانات المُراد تبديلها.

عمرو : لكن .. لماذا تُعيد الدالة void ؟

أحمد : ببساطة .. لأن هدف الدالة يكمن في **تبديل** محتوى الخانتين, فقط ! لا أكثر و لا أقل, لذا ليست هناك قيمة مُعادة. أغلب الدوال التي تستقبل "بالمرجع" تعيد void لأن التغيير سيحصل في المتغير نفسه. لذا لسنا بحاجة إلى إعادة قيمة المتغير الجديدة.

عمرو : طيب, لكنني لم أفهم الطريقة التي تستخدم المؤشرات !

أحمد : هل فهمتَ الطريقة الثانية ؟

عمرو : نعم

أحمد : إذا أنت بالتأكيد قد فهمت الطريقة الأولى !, دعني أسألك !

عمرو : تفضل

أحمد : ألم نقل أن الدالة Swap تستقبل مصفوفة ؟

عمرو : بلى !

أحمد : و هل المصفوفة عبارة عن كتلة واحدة ؟

عمرو : عفوا, لم أفهم السؤال !

أحمد : ...كنت أقصد, هل تُخَزَّن المصفوفة في خانة واحدة من الذاكرة ؟ كما يحدث مع الأنواع البدائية للغة كـ int مثلا ؟

عمرو : لا! طبعا, فالمصفوفة عبارة عن مجموعة من الخانات المتجاورة فيما بينها و ليست خانة واحدة.

أحمد : جميل جدا, إذا .. كيف تستقبل الدالة هذه الخانات في آن واحد و تميزها عن غيرها ؟

عمرو: بصراحة .. لا أدري !

أحمد : الأمر بسيط جدا .. كما قلت أنت بنفسك قبل قليل " المصفوفة عبارة عن مجموعة من الخانات المتجاورة فيما بينها " و لكي يكون الكلام أكثر دقة يمكننا القول بأن المصفوفة عبارة عن مجموعة من العناصر المتجانسة(من نفس النوع) مُسجلة تحت اسم واحد ,حيث يمكن تمييز كل عنصر بترتيبه (دليله) في المصفوفة.

عمرو : ...متجانسة ؟!

أحمد : نعم, متجانسة ! و إلا فسنحصل على structure أو class ...عموما, دعنا في الموضوع !, قل لي .. كيف تُفرق بين مصفوفتين ؟

عمرو : بالإسم طبعا .. لأنه لا يمكن وجود أكثر من مصفوفة بنفس الإسم !

أحمد : طيب, لكن اسم المصفوفة ما هو إلا مؤشر يشير إلى أول عنصر فيها.

عمرو : أممممم .. يبدو أن مفتاح الفكرة يكمن في العنصر الأول !

أحمد : بالضبط, فانطلاقا من العنصر الأول يمكننا الوصول إلى بقية عناصر المصفوفة عن طريق إزاحة المؤشر إلى الأمام, لأننا قلنا سابقا أن خانات المصفوفة مُتجاورة !

عمرو: و ماذا لو حصلنا على مصفوفة خاناتها غير مُتجاورة ؟

أحمد: مُستحيل .. المصفوفة بالتعريف هي مجموعة من الخانات المتجاورة فيـ...

عمرو: نعم .. نعم !, تذكرت ! ...طيب, هل لك أن تُلخص لي فكرة الكود السابق ؟ فأفكاري حوله لم تنضج بعد !

أحمد: الدالة Swap تستقبل مؤشر من نوع int بالإضافة إلى رقمان يدلان على مكان وجود القيم التي نريد إبدالها فيما بينها. واضح ؟

عمرو: مفهوم !

أحمد: في جسم الدالة قمنا باستخدام وسيط مساعد من شأنه مُساعدتنا في عملية Swap. الكتابة *(X + i) يمكننا قراءتها "قم بإزاحة المؤشر i إلى الخانة الموالية" و بالتالي سيشير المؤشر إلى الخانة رقم

i+1 لأن الترقيم يبدأ من صفر. بنفس الطريقة قمنا بإزاحة المؤشر للحصول على محتوى الخانة رقم j.و بهذا تمت عملية التبادل.

عمرو: اتضحت الفكرة. طيب, هل من تطبيق لما سبق ذكره ؟

أحمد: على مهلك, جواب سؤالك سيكون في الفقرة الموالية.

II.4 – الخوارزمية بلغة السي++

كما ذكرنا سابقا فإن هذه الخوارزمية تقوم في الدورة الأولى بإزاحة أكبر عناصر المصفوفة إلى الخانة الأخيرة ثم تقوم في الدورة الثانية بإزاحة العدد الذي يلي أكبر العناصر إلى الخانة القبل الأخيرة و هكذا .. آخر دورة سيتم فيها وضع أصغر العناصر في الخانة الأولى. كل دورة ينتج عنها بعض التبديلات بين خانات المصفوفة مما يُساعد على تحسين ترتيب المصفوفة.

```
void bubbleSort(int array[], int size) {
    for (int i = 0; i <= size - 2; i++)
        for (int j = 0; j <= size - i - 2; j++)
            if (array[j] > array[j + 1])
                Swap(array, j, j + 1);
}
```

تستقبل الدالة bubbleSort وسيطين: اسم المصفوفة و عدد عناصرها. تبدأ الحلقة الأولى ذات العداد i من أول خانة في المصفوفة و حتى الخانة القبل الأخيرة.

عمرو : هل أفهم من كلامك أن الترتيب لن يشمل الخانة الأخيرة ؟

أحمد : كلا, لكن إذا وصل عداد الحلقة الأولى إلى القيمة size-1 سيؤدي هذا إلى حصول Underflow !, ستفهم أكثر بعد قليل ... هناك حلقة ثانية, تبدأ في كل مرة من الخانة الأولى و حتى الخانة رقم size - i – 2.

عمرو : .. size - i – 2 !!؟, من أين أتيت بهذا العدد ؟؟

أحمد : الحلقة الثانية تعمل على ترتيب المصفوفة و في أسوأ الأحوال سيتم وضع عنصر واحد في الترتيب الصحيح. منطقيا .. إذا عادت الخوارزمية من جديد للترتيب فيجب أن تستثني العناصر التي تم ترتيبها. من أجل i=0 سيتم وضع أكبر الأعداد في الخانة الأخيرة (حصلنا على خانة واحدة مرتبة) و من أجل i=1 سيتم وضع العدد الذي يلي أكبر الأعداد في الخانة القبل أخيرة (حصلنا على خانتين مرتبتين). إذا في كل مرة سنحصل على size-i-1 من العناصر الغير مرتبة.

عمرو : و بعد ؟

أحمد: حتى لا نُعيد اختراع العجلة .. لكي يتم ترتيب العناصر الغير مرتبة فقط يجب أن تبدأ الحلقة الثانية من الخانة الأولى (رقم صفر) و تتوقف عند الخانة رقم size-i-1.. و لكي لا يحدث Overflow يجب أن يكون j+1 = size-i-1 مما يعني أن j=size-i-2 و هي القيمة التي تتوقف عندها الحلقة الثانية.

أحمد :نعود الآن إلى قضية الـ Underflow, عدد عناصر المصفوفة يساوي size, إذا رقم الخانة الأخيرة هو size-1 و رقم الخانة القبل أخيرة هو size-2, لاحظ أنه إذا كانت i=size-2 ستصبح j=0 وهنا ستتم مقارنة قيمة الخانة الأولى مع الثانية و إبدال محتوى الخانتين إذا كان الترتيب معكوسا و هذه هي آخر خطوات الخوارزمية.

أما إذا كانت i=size-1 فستصبح j=-1 و هذا خطأ منطقي لأن ترقيم المصفوفات يبدأ من صفر.

نأتي الآن إلى كتابة كود متكامل يحتوي على الدالة bubbleSort :

```cpp
#include <iostream>
using namespace std;

void Swap(int *X, int i, int j) {
     int temp;
     temp = *(X + i);
     *(X + i) = *(X + j);
     *(X + j) = temp;
}

void bubbleSort(int array[], int size) {
     for (int i = 0; i <= size - 2; i++)
         for (int j = 0; j <= size - i - 2; j++)
             if (array[j] > array[j + 1])
                 Swap(array, j, j + 1);
}

int main() {
     const int LENGTH = 10;
     int x[LENGTH] = {3,9,2,15,4,11,0,-5,8,-2};
     cout<<"Before : ";
     for (int i = 0; i < LENGTH; i++)
         cout<<x[i]<<" ";
     bubbleSort(x, LENGTH);
     cout<<endl<<"After : ";
     for (int i = 0; i < LENGTH; i++)
         cout<<x[i]<<" ";
     return 0;
}
```

المتغير LENGTH يمثل طول المصفوفة x. قمنا بطباعة عناصر المصفوفة قبل و بعد استدعاء دالة الترتيب. لاحظ الفرق.

مثال للتوضيح :

لنفترض أننا نريد ترتيب الجدول التالي :

8	5	0	11	6	2

عند تطبيق الخوارزمية على الجدول سنحصل على النتائج التالية :

إذا كانت i=0 :

5	8	0	11	6	2
5	0	8	11	6	2
5	0	8	11	6	2
5	0	8	6	11	2

5	0	8	6	2	11

في كل مرة يتم تبديل محتوى الخانتين ذوات اللون الأزرق الفاتح, الخانة ذات اللون الأخضر تدل على أن ترتيب الخانتين في الوضع الصحيح لذا لم تتم مبادلتهما. لاحظ أنه من خلال الدورة الأولى تمت إزاحة أكبر عناصر الجدول إلى الخانة الأخيرة (في هذا المثال تمت إزاحة العدد 11 إلى الخانة الأخيرة).

i=1 :

0	5	8	6	2	11
0	5	8	6	2	11
0	5	6	8	2	11
0	5	6	2	8	11
0	5	6	2	8	11

إذا كانت i = 1 فإن القيمة العظمى لـ j ستكون 3 و بالتالي لن تتم المقارنة بين محتوى الخانة الأخيرة و الخانة الموجودة قبلها. لاحظ أنه سيتم ترتيب عناصر الجدول انطلاقا من نهايته.

i = 2 :

0	5	6	2	8	11
0	5	6	2	8	11
0	5	2	6	8	11
0	5	2	6	8	11
0	5	2	6	8	11

إذا كانت i=2 فإن $J_{max}=2$ و هذا يعني أن الخانات رقم 3,4,5 مُرتبة بشكل صحيح.

i = 3 :

0	5	2	6	8	11
0	2	5	6	8	11
0	2	5	6	8	11
0	2	5	6	8	11
0	2	5	6	8	11

لاحظ معي أنه تم ترتيب عناصر الجدول مع أن الحلقة لم تنتهي بعد فما زالت دورتان i=4, i=5.

ما رأيك لو أضفنا شرطا بسيطا يتعلق بالخروج من الدالة إذا وصلت الخوارزمية إلى ترتيب الجدول قبل انتهاء الحلقة ؟

```
void bubbleSort(int array[], int size) {
    bool sort;
    for (int i = 0; i <= size - 2; i++) {
        sort = false;
        for (int j = 0; j <= size - i - 2; j++)
            if (array[j] > array[j + 1]) {
                Swap(array, j, j + 1);
                sort = true;
            }
        cout << "i = " << i << " : " << endl;
        for (int k = 0; k < size; k++)
            cout << array[k] << " ";
        cout << endl;
        if (!sort && i != size - 2) {
            cout << "the table has been sorted to i = " << i << endl;
            break;
        }
    }
}
```

الفكرة ببساطة تكمن في استخدام متغير منطقي وتغيير قيمته عندما يتم تبادل محتوى خانتين. عندما نجد أنه لم تتغير قيمة المتغير منطقي في إحدى الدورات فهذا يعني أن خانات الجدول أصحبت مُرتبة بشكل صحيح.

مُحتوى الدالة الرئيسية سيكون هذا :

```
int main() {
    const int LENGTH = 6;
    int x[] = {8, 5, 0, 11, 6, 2};
    bubbleSort(x, LENGTH);
    return 0;
}
```

II.5 – التعقيد الزمني

تُعد خوارزمية ترتيب الفقاعات من أبطئ خوارزميات الترتيب حيث يعود السبب إلى كثرة عمليات المقارنة و التبديل. من أجل ترتيب جدول يحتوي على n عنصر سنحتاج إلى n-1 مقارنة (الأول و الثاني, الثاني و الثالث, ... , ما قبل الأخير والأخير) ثم نكرر نفس العملية من جديد على جدول يحتوي على n-1 عنصر و هكذا دواليك. إذا المرور رقم i سيحوي n-i مقارنة و بالتالي سيكون التعقيد الزمني للخوارزمية من حيث عدد المقارنات يُساوي :

$$T(n) = (n-1) + (n-2) + (n-3) + \dots 1 = \sum_{i=1}^{n-1} i \; = \frac{n(n-1)}{2}$$

بالنسبة لعدد التبديلات : في أسوأ الحالات سنحتاج إلى n-i في المرور رقم i. في هذه الحالة نقول أن الجدول مُرتب بشكل مقلوب.

إذا نخلص إلى أنه في المرور رقم i سنحتاج بالضبط إلى n-i مقارنة و n-i تبديل في أسوأ الحالات.

في أسوأ الحالات سيكون تعقيد الخوارزمية يساوي $O(n^2)$ و في المتوسط سيكون عدد التبديلات يُساوي $\frac{n(n-1)}{4}$ أما في أفضل الحالات فسيكون التعقيد خطيا $O(n)$.

II.6 – ملأ مصفوفة عشوائيا و اختبار سرعة الخوارزمية

سبق و أن تعرضنا في الحلقة السابقة إلى ملأ مصفوفة عشوائيا, الجديد هنا هو استخدام عداد و زيادته كل ما تمت عملية تبديل, لا أكثر :

```cpp
void bubbleSort(int array[], int size) {
    bool sort;
    int nSort = 0;
    for (int i = 0; i <= size - 2; i++) {
        sort = false;
        for (int j = 0; j <= size - i - 2; j++)
            if (array[j] > array[j + 1]) {
                Swap(array, j, j + 1);
                sort = true;
                nSort++;
            }
        if (!sort && i != size - 2) {
            cout << "the table has been sorted to i = " << i << endl;
            break;
        }
    }
    cout<<"number of comparisons : "<<nSort<<endl;
}
```

عند استدعاء الدالة bubbleSort سيتم إظهار عدد المرات التي تمت فيها عملية التبديل بالإضافة طبعا إلى رقم المرور الذي تمت فيه عملية الترتيب.

ملأ المصفوفة x عشوائيا و استدعاء الدالة bubbleSort داخل main سيكون هكذا :

```cpp
int main() {
    const int LENGTH = 1000;
    int x[LENGTH];
    srand(time(NULL));
    for(int i = 0;i < LENGTH;i++)
        x[i] = rand()%2000;
    bubbleSort(x, LENGTH);
    system("PAUSE");
    return 0;
}
```

III – خوارزمية البحث الثنائي (Binary search algorithm)

III.1 – الخوارزمية (الهدف, الفكرة, النتيجة, الإيجابيات و السلبيات)

الهدف : البحث عن قيمة المفتاح key داخل المصفوفة X.

الفكرة : تعتمد هذه الخوارزمية على البحث الثنائي (Binary search) في المصفوفة X حيث يبدأ البحث من العنصر الذي يقع في وسط المصفوفة, و في كل مرة نقارنه مع المفتاح Key, إذا كانت القيمتان متساويتان, فهذا يدل على أنه تم إيجاد قيمة المفتاح في المصفوفة, أما إذا كانت القيمتان مختلفتان ستقوم الخوارزمية بإجراء فحص جديد, إذا كانت قيمة المفتاح أصغر من قيمة العنصر الأوسط سيتم البحث في الجزء الأيسر من المصفوفة و في الحالة المعاكسة سيتم البحث في الجزء الأيمن من المصفوفة, و هكذا .. حتى نحصل على مصفوفة تتكون من خانة واحدة قيمتها مساوية للمفتاح أو مختلفة عنه.

النتيجة : إذا كانت X تحتوي على Key فسنحصل على رقم الخانة التي يوجد بها الأخير و إلا فالقيمة المُعادة ستكون 1-.

الإيجابيات: الـ **Binary search** تُنصف (تقلص للنصف) عدد عناصر المصفوفة في كل تكرار, لذا تستغرق عملية البحث زمنا لوغاريتميا. تنتمي هذه الخوارزمية إلى عائلة "**فرق تسد**".
السلبيات: خوارزمية البحث الثنائي أكثر تعقيدا من خوارزمية البحث الخطي التقليدية, كما أن الأولى **تشترط الترتيب عند البحث**.

III.2 – الخوارزمية بلغة السي++

يمكننا كتابة الخوارزمية باستخدام الـ while loop أو الـ Function recersive.

III.2.1 – الطريقة الأولى

```cpp
#include <iostream>
using namespace std;

int binarySearch(int array[], const int length, int Key) {
    int low = 0, mid, high = length - 1;
    while (low <= high) {
        mid = (low + high) / 2; // العنصر الذي يقع في وسط المصفوفة
        if (Key == array[mid])
            return mid; // تم إيجاد المفتاح
        else if (Key < array[mid])
            high = mid - 1; // البحث في الجزء الأيسر من المصفوفة
        else
            low = mid + 1; // البحث في الجزء الأيمن من المصفوفة
    }
    return -1; // لم يتم العثور على المفتاح
}

int main() {
    const int LENGTH = 1000, centre = LENGTH / 2;
    int result, X[LENGTH];
    for (int i = 0, j = -centre; j <= centre; i++, j++)
        X[i] = j;
    result = binarySearch(X, LENGTH, -73);
    if (result != -1)
        cout << "la clef se trouve dans la case numéro " << result << endl;
    else
        cout << "Valeur introuvable" << endl;
    return 0;
}
```

الدالة binarySearch تستقبل 3 وسائط, المصفوفة المُراد البحث داخلها و عدد عناصرها و قيمة المفتاح.يبدأ ترقيم عناصر المصفوفة بالقيمة low و ينتهي عند high, ويمثل المتغير mid رقم الخانة الوسطى من المصفوفة. في كل مرة نقارن قيمة المفتاح بقيمة أوسط عناصر المصفوفة, إذا كانتا متساويتين نُعيد رقم الخانة و إذا كانتا مختلفتين نتقدم بخطوة إلى الأمام أو نرجع بخطوة إلى الوراء حسب وضعية المفتاح. نكرر الخطوات السابقة ما دام low أقل أو يساوي high (هذا يعني أن المصفوفة تحتوي على خانة أو أكثر). إذا تم الخروج من الحلقة while دون إعادة قيمة فهذا يعني أن المفتاح غير موجود, في هذه الحالة ستعيد الدالة 1-.

في الدالة الرئيسية قمنا بالإعلان عن مصفوفة تحوي ألف عنصر ثم ملأناها بالقيم الواقعة في المجال -[500,500] لاحظ أن عناصر المصفوف يجب أن تكون مرتبة. بعد ذلك قمنا بالبحث عن القيمة -73 داخل المصفوفة و قد تم إيجادها في الخانة رقم 427 وهذا شيء طبيعي لأن الصفر يتواجد في الخانة رقم 500 و بالتالي القيمة 73- ستتواجد في الخانة رقم 73-500.

يمكننا استدعاء الدالة binarySearch على جزء من المصفوفة, في هذه الحالة نضع low و high كوسائط للدالة و ليس كمتغيرات محلية.

III.2.2 – الطريقة الثانية

```
1  #include <iostream>
2  using namespace std;
3
4  int binarySearch(const int array[], int key, int low, int high) {
5      int mid;
6      if (high < low) return -1;
7      else {
8          mid = (low + high) / 2;
9          if (array[mid] > key)
10             return binarySearch(array, key, low, mid - 1);
11         else if (array[mid] < key)
12             return binarySearch(array, key, mid + 1, high);
13         else
14             return mid;
15     }
16 }
17
18 int main() {
19     const int LENGTH = 1000, centre = LENGTH / 2;
20     int result, X[LENGTH];
21     for (int i = 0, j = -centre; j <= centre; i++, j++)
22         X[i] = j;
23     result = binarySearch(X, -500, 0, LENGTH - 1);
24     if (result != -1)
25         cout << "la clef se trouve dans la case numéro " << result << endl;
26     else
27         cout << "Valeur introuvable" << endl;
28     return 0;
29 }
```

تستقبل الدالة binarySearch أربع وسائط, هم على التوالي : اسم المصفوفة و المفتاح و من أين يبدأ البحث ؟ و أين ينتهي ؟

المتغير mid يحتوي على رقم الخانة الوسطى من المصفوفة. إذا كانت قيمة المفتاح أكبر (أصغر) من mid سيتم تطبيق الدالة على الجانب الأيسر (الأيمن) من المصفوفة. إذا كانت قيمة high أقل تماما من low فهذا يعني أن المصفوفة أصبحت فارغة.

III.3 – أمثلة على الخوارزمية

نبدأ مع مثال بسيط يُوضح فكرة البحث الثنائي:

III.3.1 – المثال الأول

أحمد: ما رأيك بلعبة الكاهن ؟

عمرو: الكاهن ؟؟

أحمد: نعم, أختارُ عددا عشوائيا يقع في المجال [0,100] و عليك معرفة العدد !

عمرو: هذا مستحيل !!

أحمد: لا تستعجل .. في كل مرة تختار فيها عددا, سأخبرك ما إذا كان أكبر أو أصغر من العدد الذي تبحث عنه.

عمرو: لا بأس, سأحاول .. سأبدأ بالعدد الذي يقع في منتصف المجال, هل عددك أكبر من 50 ؟

أحمد: نعم !

عمرو : سأختار الآن أوسط أعداد المجال الجديد, هل العدد أكبر من 75 ؟

أحمد: لا !

عمرو: إذا العدد يقع بين 51 و 75, هل هو أكبر من 63 ؟

أحمد: نعم !

عمرو: جيد, أصبح المجال ضيقا الآن, هل العدد أصغر من 69 ؟

أحمد : نعم !

عمرو: أكبر من 66 ؟

أحمد: لا !

عمرو: أصغر من 65 ؟

أحمد: لا !

عمرو: إذا فالعدد المختار هو 66 ؟

أحمد: نعم, أحسنت !

- لو وضع عمرو في الحسبان أن العدد الذي يبحث عنه يمكن أن يتواجد في منتصف المجال, لاستغنى عن آخر سؤالين !
- للفائدة, سبق و أن كتبتُ برنامجا صغيرا يُحاكي لعبة الكاهن, يمكنك تجربته من هنا : http://www.arabteam2000-forum.com/index.php?showtopic=231557

III.3.2 – المثال الثاني

نريد البحث عن العدد 8 في مصفوفة مُكونة من 10 خانات, قيمها كالآتي:

0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

الخانات الملونة باللون الأخضر تُمثل الجزء الذي سيتم البحث فيه من المصفوفة و الخانة ذات اللون الأزرق تمثل أوسط العناصر.

III.3.3 – المثال الثالث

ابحث عن القيمة 33 في المصفوفة التالية :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
↑ low														↑ high

الحل:

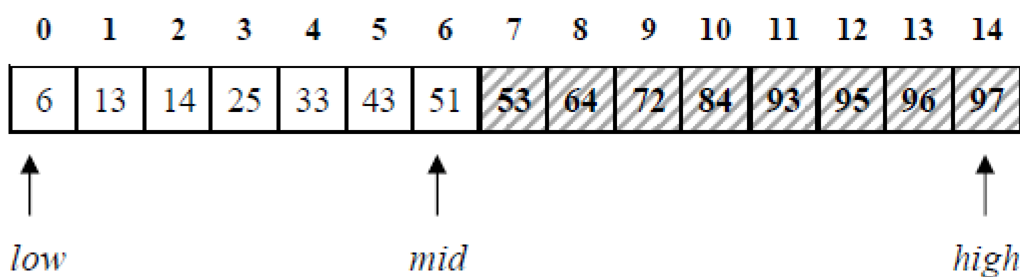
1. نحدد منتصف المصفوفة:

$$Mid = \frac{low + high}{2} = \frac{(0 + 14)}{2} = 7$$

2. تقسيم المصفوفة إلى قسمين وفقا لموقع mid:

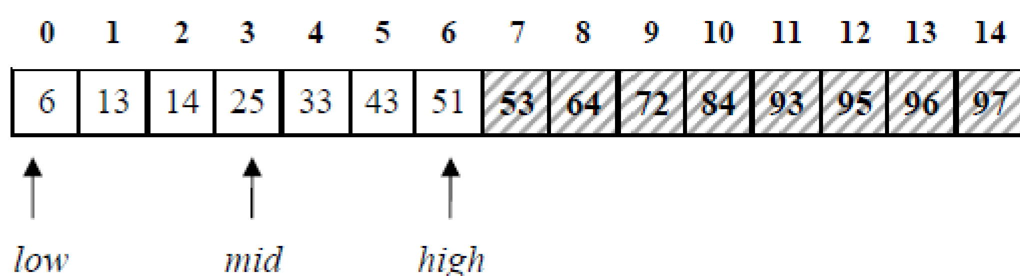
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
↑ low							↑ mid							↑ high

3. بما أن 33 أصغر من 53, فعملية البحث ستنفذ على النصف الأول:

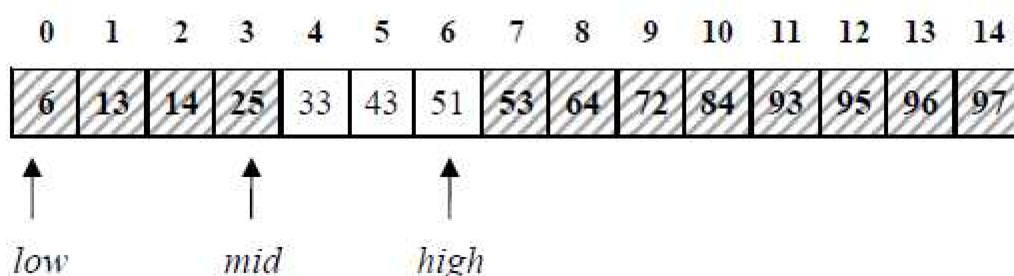


4. نعيد عملة التقسيم على النصف الأول و تصبح قيمة mid تساوي:

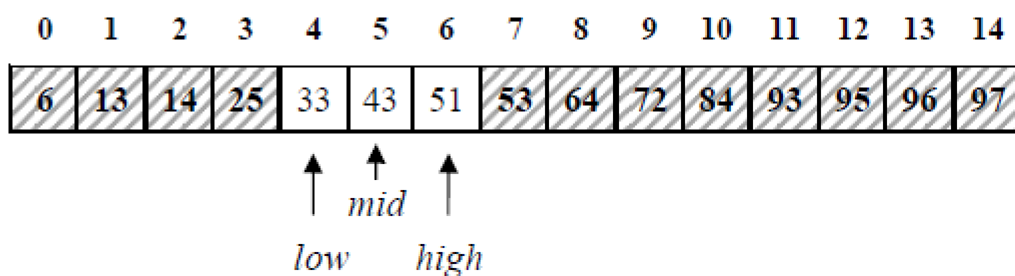
$$Mid = \frac{(0+7)}{2} = 3$$



5. بما أن 33 أكبر من 25, سيتم استبعاد النصف الأول و تستمر عملية البحث في النصف الثاني:



6. من جديد, نعيد عملية التقسيم حيث تصبح قيمة mid تساوي 5:



7. و أخيرا نجد أن القيمة 33 موجودة في الموقع low و الذي يساوي 4.

III.4 – التعقيد الزمني

خوارزمية البحث الثنائي سريعة جدا و فعالة أيضا, خصوصا عند التعامل مع المصفوفات الكبيرة, لأنها تُلغي في كل مرة نصف المصفوفة و تبحث في النصف الآخر (طبعا, دون أن ننسى أن المصفوفة يجب أن تكون مُرتبة و هذه ضريبة السرعة ^_^)

لو أخذنا على سبيل المثال مصفوفة تتكون من 1024 خانة, في أسوأ الحالات ستقوم الخوارزمية بـ 10 مقارنات لمعرفة ما إذاكان المفتاح موجود أم لا ! (لاحظ أن $1024 = 2^{10}$). و لو زدنا عدد خانات المصفوفة ليصبح 576 048 1, ستقوم الخوارزمية بـ 20 مقارنة في أسوأ الحالات !

في **أفضل الحالات**, يكون يتواجد المفتاح في منتصف المصفوفة و بالتالي يكون التعقيد الزمني $T(n) = 1$ و في **أسوأ الحالات** يكون المفتاح في بداية أو نهاية المصفوفة أو غير موجود. في كل مرة يتم إلغاء نصف المصفوفة و العمل على النصف الآخر, لذا سيكون طول المصفوفة يساوي $\frac{}{2^k}$ في الخطوة رقم k.

نتوقف عندما تصبح المصفوفة عبارة عن خانة واحدة:

$$\frac{n}{2^k} = 1 \Rightarrow n = 2^k = e^{k.\log_2} \Rightarrow k = \frac{\log_n}{\log_2}$$

إذا, في هذه الحالة سيكون التعقيد يساوي $O(n) = \log_n$

الحالة المتوسطة: عندما تكون هناك احتمالية وجود المفتاح في أي جزء من المصفوفة, و عليه سيكون التعقيد:

$$T(n) = \sum_{i=1}^{n}(i * \frac{1}{n}) = \frac{1}{n} * \sum_{i=1}^{n} i = \frac{1}{n} * \frac{n(n+1)}{2} = \frac{(n+1)}{2}$$

III.5 – ملأ مصفوفة عشوائيا و اختبار السرعة

في هذه الفقرة, سنملأ عشوائيا مصفوفة تحتوي على 1000 خانة ثم نرتب المصفوفة لنبحث عن داخلها عن قيمة معينة, سنحتاج إلى الدوال التالية لتنفيذ ما سبق:

```
void Swap(int *X, int i, int j) {
    int temp = *(X + i);
    *(X + i) = *(X + j);
    *(X + j) = temp;
}

void bubbleSort(int array[], int size) {
    for (int i = 0; i <= size - 2; i++)
        for (int j = 0; j <= size - i - 2; j++)
            if (array[j] > array[j + 1])
                Swap(array, j, j + 1);
}

int binarySearch(const int array[], const int length, int Key) {
    int low = 0, mid, high = length - 1;
    while (low <= high) {
        mid = (low + high) / 2;
        if (Key == array[mid])
            return mid;
        else if (Key < array[mid])
            high = mid - 1;
        else
            low = mid + 1;
    }
    return -1;
}
```

لا جديد, فقط قمتُ بتجميع الدوال التي تعرفنا عليها سابقا !

محتوى الدالة الرئيسية سيكون هكذا:

```
#include <ctime>
#include <cstdlib>
#include <iostream>
using namespace std;

int main() {
    const int LENGTH = 1000;
    int result, Random, X[LENGTH];
    srand(time(NULL));
    for (int i = 0; i < LENGTH; i++)
        X[i] = (rand() % 10000) + i;
    bubbleSort(X, LENGTH);
    Random = X[rand() % LENGTH];
    result = binarySearch(X, LENGTH, Random);
    if (result != -1)
        cout << "la clef " << Random << " se trouve dans la case numero " <<
            result << endl;
    else
        cout << "valuer intruovabe !" << endl;
    return 0;
}
```

قمنا بملأ المصفوفة عشوائيا ثم رتبنا القيم تصاعديا, بعد ذلك قمنا بتخزين قيمة إحدى خانات المصفوفة في المتغير Random بشكل عشوائي ثم بحثنا عن مكان تلك القيمة داخل المصفوفة X.

لاحظ سرعة الخوارزمية مُقارنة مع خوارزمية البحث الخطي التي تعرفنا عليها في الدرس الأول.

III.6 – اختبر قدراتك !

لتكن M مصفوفة ثنائية البعد, عدد صفوفها L وعدد أعمدتها C. المصفوفة مُرتبة تصاعديا حسب الأعمدة و حسب الصفوف أيضا .

- اكتب دالة تبحث داخل M عن قيمة معينة (تستقبلها كوسيط) ثم تُعيد true إذا كانت القيمة موجودة و false في الحالة المعاكسة.
- أعط التعقيد الزمني للدالة السابقة بدلالة L و C ؟

الحل:

لإيجاد مكان المفتاح في المصفوفة, نبحث عن رقم السطر الذي يحوي قيمة المفتاح ثم نُطبق عليه خوارزمية البحث الثنائي, مع الأخذ في الاعتبار أن المصفوفة مُرتبة حسب الأعمدة و الصفوف أيضا:

```cpp
#include <iostream>
using namespace std;

#define numberOfColumns 4

bool searchTwoDimensional(int array[][numberOfColumns], int rows, int columns, int value) {
    int i = 0;
    while (i <= rows - 1 && array[i++][0] <= value);
    if (i == 0) return false;
    else {
        i--;
        int low = 0, high = columns - 1, mid = (low + high) / 2;
        while (array[i][mid] != value && low <= high) {
            array[i][mid] < value ? low = mid + 1 : high = mid - 1;
            mid = (high + low) / 2;
        }
        if (high < low) return false;
        else return true;
    }
}

int main() {
    int newArray[3][numberOfColumns] = {
        {1, 3, 5, 7},
        {10, 15, 27, 28},
        {30, 39, 52, 100}
    };
    cout << searchTwoDimensional(newArray, 3, numberOfColumns, 52);
    return 0;
}
```

بعد الخروج من الحلقة الأولى, لدينا ثلاث حالات:

- i=0 و هذا يعني أنه لم يتم الدخول إلى الحلقة ! لأن قيمة أول عنصر في المصفوفة أكبر تماما من قيمة المفتاح. و بما أن المصفوفة مرتبة حسب الأعمدة و الصفوف, فيمكننا القول بأن المفتاح غير موجود في المصفوفة. في هذه الحالة ستعيد الدالة false.
- i أكبر أو يساوي 2 و أقل أو يساوي rows, في هذه الحالة سيمثل i رقم أول سطر يبدأ بقيمة أكبر تماما من قيمة المفتاح, لذا فإن المفتاح سيتواجد في السطر رقم i-1.
- i = rows + 1, هذه الحالة تعني أن جميع الأسطر تبدأ بقيمة أكبر تماما من قيمة المفتاح , قد يتواجد المفتاح في السطر الأخير من المصفوفة.

بعد الخروج من الحلقة الأولى (في حالة i يختلف عن الصفر), سيتنقل التنفيذ إلى الجزء المخصص للبحث عن قيمة المفتاح باستخدام خوارزمية البحث الثنائي.

في الدالة الرئيسية, قمنا بالإعلان عن مصفوفة ثنائية البعد, تحتوي على 3 أسطر و أربعة أعمدة. ثم قمنا بالبحث عن القيمة 52 التي تتواجد في آخر أسطر المصفوفة.

بالنسبة للتعقيد الزمني:

في أسوأ الحالات, ستمر الحلقة الأولى على جميع أسطر المصفوفة و بالتالي سيكون تعقيدها الزمني من النوع $\theta(rows)$, الحلقة الثانية لها تعقيد لوغاريتمي $\theta(\log(columns))$.

إذا, التعقيد الكلي سيكون: $\theta(rows + \log(columns))$

لا يمكننا تبسيط هذه العبارة لأننا لا نعرف العلاقة ما بين rows و columns.

الـخـاتـمـة

إلى هنا أصل بكم إلى نهاية هذا الكتاب الذي تحدثنا فيه عن بعض الخوارزميات المستعملة بكثرة في الترتيب و البحث, إذا أردت التعمق أكثر فيمكنك دراسة ما تبقى من الخوارزميات البدائية مثل خوارزمية Selection sort و الـ Insertion sort و Creation sort و Merge sort و Quicksort. بعد ذلك يمكنك الانتقال إلى تطبيق الخوارزميات السابقة على بنى و تراكيب أخرى أكثر تطور مثل الـ Binary tree.