Pairwise summation

Объявление функции: без указателей

Если у нас нет указателей, то делать нечего, приходится передавать исходный массив, хотя нам, быть может, нужна только его часть.
И задавать эту часть:

```
double pair_sum(double m[], int first_idx, int numb);    // массив, индекс начального,
                                                         //     количество элементов
```

или

```
double pair_sum(double m[], int first_idx, int last_idx); // массив, индексы
                                                          //   начального и конечного эл-ов
```

или

```
double pair_sum(double m[], int first_idx, int end_idx); // массив, индекс начального,
                                                         //    индекс ПОСЛЕ конечного
```

Pairwise summation

Объявление функции: есть указатели!!

Но указатели есть, поэтому вместо пары "массив, first_idx" передаём сразу указатель на нужную часть массива:

```
double pair_sum(double m[], int numb);        // адрес начального элемента,
                                              //     количество элементов
```

или

```
double pair_sum(double *pfirst, double *plast);    // адреса начального и конечного
                                                   //     элементов
```

или

```
double pair_sum(double *pfirst, double *pend);    // адрес начального элемента,
                                                  //     адрес ПОСЛЕ конечного эл-та
```

Конечно же, **m** и **pfirst** – одно и то же.
Конечно же, из **m** и **numb** легко получить **pend**, если надо.

Pairwise summation

Объявление функции: есть указатели!!

Конечно же, **m** и **pfirst** – одно и то же.
Конечно же, из **m** и **numb**, если надо, легко получить **pend**:

```
double pair_sum(double m[], int numb)        // адрес начального элемента,
{                                            //     количество элементов
      double *pend = m + numb;
}
```

И наоборот, из **pfirst** и **pend** – получить **numb**:

```
double pair_sum(double *pfirst, double *pend)    // адрес начального элемента,
{                                                //     адрес ПОСЛЕ конечного эл-та
      int numb = pend - pfirst;
}
```

Ну и, понятно:

```
      plast = m + numb - 1;          //     plast = m + (numb - 1);
      numb = plast - pfirst + 1;     //     numb = (plast - pfirst) + 1;
```

Pairwise summation

Вызов функции

В данном случае удобнее использовать 1-й вариант:

```
double pair_sum(double m[], int numb);      // адрес начального элемента,
                                            //     количество элементов
```

Тогда вызовы могли бы выглядеть так:

```
int main(void)
{
    double a[100], *p, res, res1, resmdl;
    ...
    res = pair_sum(a, 100);
    ...
    res1 = pair_sum(a + 10, 70);
    res1 = pair_sum(&a[10], 70);
    ...
    p = a + 33;
    resmdl = pair_sum(p, 34);
    ...
    return 0;
}
```

Pairwise summation

Вызов функции

В данном случае удобнее использовать 1-й вариант:

```
double pair_sum(double m[], int numb);        // адрес начального элемента,
                                              //     количество элементов
```

Тогда вызовы могли бы выглядеть так:

```
int main(void)
{
    double a[100], *p, res, res1, resmdl;
    ...
    res = pair_sum(a, 100);     // сумма всех 100 эл-ов массива (с индексами 0...99)
    ...
    res1 = pair_sum(a + 10, 70);    // одно
    res1 = pair_sum(&a[10], 70);    //     и то же: сумма 70 эл-ов с индексами 10...79
    ...
    p = a + 33;                     // сумма
    resmdl = pair_sum(p, 34);       //     34 эл-ов с индексами 33...66
    ...
    return 0;
}
```

Pairwise summation

Тело функции

Рекурсия: разбить массив на 2 части и каждую просуммировать, результаты сложить:

```
double pair_sum(double m[], int numb)        // адрес начального элемента,
{                                            //     количество элементов
      …
      return pair_sum(m, numb/2) + pair_sum(m + numb/2, numb - numb/2);
      ...
}
```

Для ясности (и для некоторой оптимизации) лучше так:

```
double pair_sum(double m[], int numb)        // адрес начального элемента,
{                                            //     количество элементов
      int n1;                 // размер 1й половины
      …
      n1 = numb / 2;
      return pair_sum(m, n1) + pair_sum(m + n1, numb - n1);
      ...
}
```

Pairwise summation

Тело функции

```
double pair_sum(double m[], int numb)        // адрес начального элемента,
{                                            //     количество элементов
      int n1;                  // размер 1й половины
      …
      n1 = numb / 2;
      return pair_sum(m, n1) + pair_sum(m + n1, numb - n1);
      ...
}
```

Почему хуже было бы использовать **pfirst** и **pend** ?
А как бы мы вычислили середину массива?

```
double *pmdl = (pfirst + pend) / 2;          // НЕЛЬЗЯ (указатели нельзя складывать)
```

Всё равно пришлось бы сначала вычислить **numb**.

Следующий вопрос: как будем останавливать рекурсию?

Pairwise summation

Тело функции

Как будем останавливать рекурсию?

```
double pair_sum(double m[], int numb)        // адрес начального элемента,
{                                            //     количество элементов
      int n1;                  // размер 1й половины

      if (numb == 1) return m[0];       //     return *m;     – то же самое

      n1 = numb / 2;
      return pair_sum(m, n1) + pair_sum(m + n1, numb - n1);
}
```

Уже будет работать.

А если эту функцию сразу вызовут с **numb** <= 0 ?

Pairwise summation

Тело функции

А если эту функцию сразу вызовут с **numb** <= 0 ?

```
double pair_sum(double m[], int numb)        // адрес начального элемента,
{                                            //     количество элементов
      int n1;                  // размер 1й половины

      if (numb <= 0) return 0.;          // sanity
      if (numb == 1) return m[0];        //     return *m;     – то же самое

      n1 = numb / 2;
      return pair_sum(m, n1) + pair_sum(m + n1, numb - n1);
}
```

А сколько раз будет выполнена операция + для чисел double ?

А сколько будет выполнено вызовов **pair_sum** ? (и, соответственно, проверок и доп.вычислений?)

Pairwise summation

Тело функции: оптимизация

```
double pair_sum(double m[], int numb)        // адрес начального элемента,
{                                            //     количество элементов
      int n1;                 // размер 1й половины

      if (numb <= 0) return 0.;          // sanity
      if (numb == 1) return m[0];        //     return *m;     – то же самое

      n1 = numb / 2;
      return pair_sum(m, n1) + pair_sum(m + n1, numb - n1);
}
```

А сколько раз будет выполнена операция + для чисел double ?
Ровно (**numb** - 1). Вроде, порядок.

А сколько будет выполнено вызовов **pair_sum** ?
Пусть для простоты **numb** = 2^N, тогда вызовов будет 1 + 2 + 4 + …
… + 2^N = 2 * **numb** - 1 (доказать, что так будет при любом **numb**).
Плохо!!

Pairwise summation

Тело функции: оптимизация

Поэтому делают, например, так:

```
double pair_sum(double m[], int numb)        // адрес начального элемента,
{                                            //     количество элементов
    if (numb > 8) {              // рекурсия:
            int n1 = numb / 2;                                          // размер 1й половины
            return pair_sum(m, n1) + pair_sum(m + n1, numb - n1); // сумма двух половин
    }
    if (numb == 8)  return ((m[0] + m[1]) + (m[2] + m[3])) + ((m[4] + m[5]) + (m[6] + m[7]));
    if (numb == 7)  return ((m[0] + m[1]) + (m[2] + m[3])) + (m[4] + m[5] + m[6]);
    if (numb == 6)  return (m[0] + m[1] + m[2]) + (m[3] + m[4] + m[5]);
    if (numb == 5)  return (m[0] + m[1] + m[2]) + (m[3] + m[4]);
    if (numb == 4)  return (m[0] + m[1]) + (m[2] + m[3]);
    if (numb == 3)  return m[0] + m[1] + m[2];
    if (numb == 2)  return m[0] + m[1];
    if (numb == 1)  return m[0];

    return 0.;          // вызов с numb <= 0
}
```

Теперь вызовов **pair_sum** будет примерно **numb** / 4.

Вопрос: почему не нужны else в многочисленных if ?

Pairwise summation

Тело функции: оптимизация

Конечно, нерекурсивная часть (а это - половина всех вызовов) может быть тоже в разных вариантах. Например, с использованием switch. Или с такой оптимизацией:

```
if (numb > 4) {
  if (numb > 6) {
    if (numb == 8)  return ((m[0] + m[1]) + (m[2] + m[3])) + ((m[4] + m[5]) + (m[6] + m[7]));
       /* else 7 */    return ((m[0] + m[1]) + (m[2] + m[3])) + (m[4] + m[5] + m[6]);
  }
  if (numb == 6)    return (m[0] + m[1] + m[2]) + (m[3] + m[4] + m[5]);
       /* else 5 */    return (m[0] + m[1] + m[2]) + (m[3] + m[4]);
}
if (numb > 2) {
  if (numb == 4)    return (m[0] + m[1]) + (m[2] + m[3]);
       /* else 3 */    return m[0] + m[1] + m[2];
}
if (numb == 2)      return m[0] + m[1];
if (numb == 1)      return m[0];

return 0.;          // вызов с numb <= 0
```

Вопрос: в чём тут оптимизация? (а при использовании switch ?)

Pairwise summation

Такое в самом деле применяется для уменьшения ошибок округления, например, в NumPy (и ещё много где).
Только рекурсия останавливается при **numb** <= 128.
И нерекурсивная часть – чуток похитрее.

Вопрос на подумать (или – поискать): как зависит ошибка вычисления суммы (её верхняя граница) от числа слагаемых при обычном суммировании и при pairwise ?

Не совсем простая задача:
Написать нерекурсивный вариант **pair_sum**.

Не совсем простой она становится при двух требованиях:
- не изменять исходный массив
- не использовать дополнительную память размером порядка исходного массива (конечно же, доп.память нужна, но, как и при рекурсии, она должна быть порядка log(**numb**)).

Вопрос: почему (и где) при рекурсии используется доп.память именно такого размера?