第三讲
递归算法
Lecture 3
Recursive Algorithms

明玉瑞 Yurui Ming

yrming@gmail.com

声明
Disclaimer

- 本讲义在准备过程中由于时间所限，所用材料来源并未规范标示引用来源。所引材料仅用于教学所用，作者无意侵犯原著者之知识版权，所引材料之知识版权均归原著者所有；若原著者介意之，请联系作者更正及删除。

  The time limit during the preparation of these slides incurs the situation that not all the sources of the used materials (texts or images) are properly referenced or clearly manifested. However, all materials in these slides are solely for teaching and the author is with no intention to infringe the copyright bestowed on the original authors or manufacturers. All credits go to corresponding IP holders. Please address the author for any concern for remedy including deletion.

分而治之
Divide and Conquer

- 在求解大规模问题中，一个常用的思想是分而治之。其是将原问题分解为若干个子问题，例如两个，而这些子问题的规模大体相当；基于分别求出的这些子问题的解，得到初始问题的解。

  Among the ideas of solving large-scale problems, one is divide-and-conquer. It is usually to decompose the original problem into several sub-problems, such as two, to find a solution. The scales of these sub-problems are roughly the same; and based on the separate solutions of these sub-problems, the solution of the initial problem is obtained.

- 根据分的方式，有时会做一些区分。当分成的两部分的量几乎等同的时候，就是传统的分而治之，当分的两部分的量很不对等时，可能会称为减而治之。

  There might exist different terminologies in describing the division strategy. Consider the case of dividing into two parts. When the scales of these two parts are almost equal, this is the traditional divide and conquer. If these two parts are obviously unbalanced, we call it reduce and conquer.

分而治之
Divide and Conquer

- 分而治之算法通常包括一个或者多个递归方法的调用，当这些调用将数据分隔成为独立的集合从而处理较小集合的时候，分而治之的策略将会有很高的效率。不过，在数据进行分解的时候，分而治之的策略可能会产生大量的重复计算，从而导致性能的降低。

  The divide-and-conquer methods usually include one or more recursive calls. When the data is separated into smaller independent sets to be processed by recursively invoking these calls, the divide-and-conquer strategy will be very efficient. However, when the data is decomposed, The divide-and-conquer strategy may generate a lot of repetitive calculations, resulting in performance degradation.

- 同时，函数调用需要栈变量维护调用和返回时的相关信息，不恰当的递归调用可能会导致栈溢出，存在潜在的安全隐患。

  At the same time, calling a function needs stack variables to maintain the information prior to calling and upon returning, and improper recursion might incur the possibility of stack overflow, and consequently introduce security risks.

递归
Recursion

- 那么什么是递归呢？

```
void func()
{
    printf("从前有座山，山上有座庙，庙里有个老和尚，老和尚在给小和尚讲故事：
    ");
    func();
}
```

- 输出：

从前有座山，山上有座庙，庙里有个老和尚，老和尚在给小和尚讲故事：从前有座山，山上有座庙，庙里有个老和尚，老和尚在给小和尚讲故事：从前有座山，山上有座庙，庙里有个老和尚，老和尚在给小和尚讲故事：从前有座山，山上有座庙，庙里有个老和尚，老和尚在给小和尚讲故事：...

递归
Recursion

- What's recursion?

```
void func()
{
    printf("Long long time ago, in a temple built on a hill, there lived two monks, and the old monk was telling a story to the little monk: ";
    func();
}
```

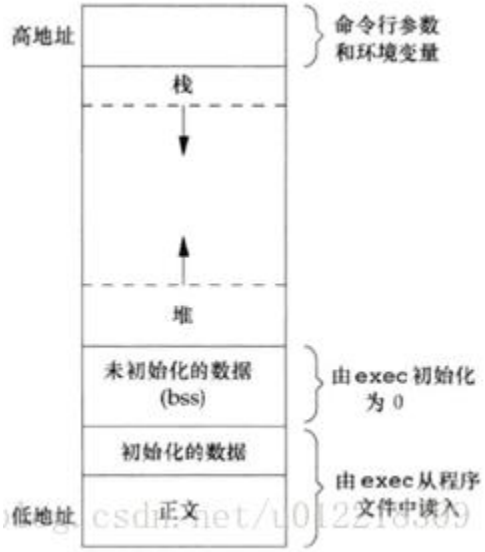
- Output:

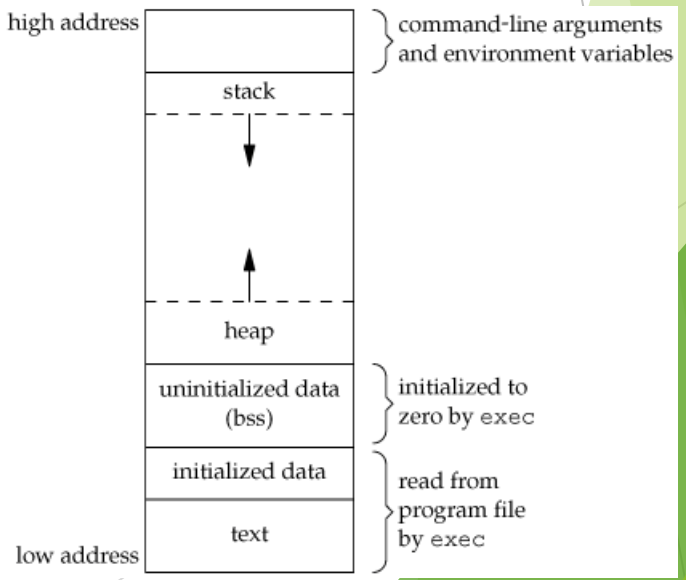
Long long time ago, in a temple built on a hill, there lived two monks, and the old monk was telling a story to the little monk: Long long time ago, in a temple built on a hill, there lived two monks, and the old monk was telling a story to the little monk: Long long time ago, in a temple built on a hill, there lived two monks, and the old monk was telling a story to the little monk: …

函数调用约定
Convention of Function Calling

- 当源代码被编译链接之后，会形成可执行程序，当被加载程序加载时，会根据可执行程序内容，加载进内存的不同地址，或内存段，如下图所示：

  After compiling and linking, the source code is converted into executable. Upon loading into the memory by linker and loader, the content of the executable will be loaded into different memory address or memory segment according to the property of the content, as illustrated below:





函数调用约定
Convention of Function Calling

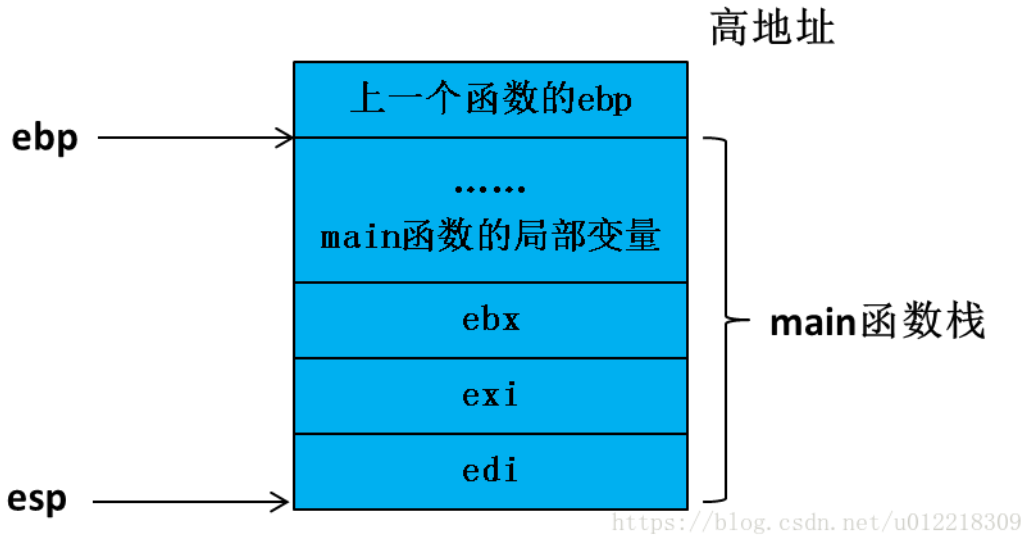
```
int sum(int a, int b)
{
    int c = a + b;
    return c;
}

int main()
{
    int a = 10, b = 20;
    int ret = sum(a, b);
    return 0;
}
```

函数调用约定
Convention of Function Calling

- main函数中在sum调用之前的栈如图：

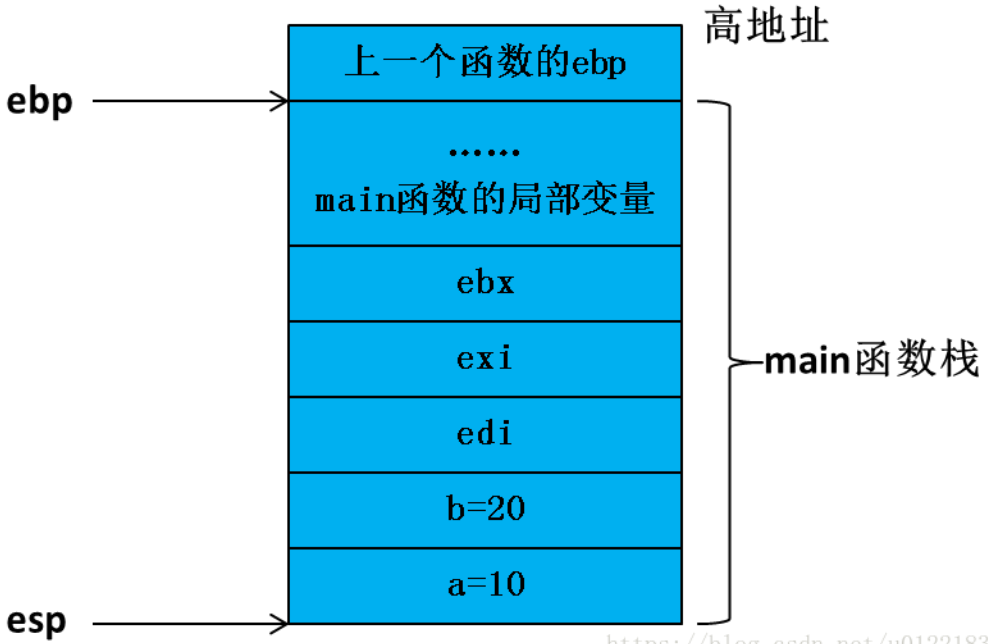
  Stack before calling the sum function in main:



函数调用约定
Convention of Function Calling

- 当准备sum调用时，第一步，将参数压入栈：

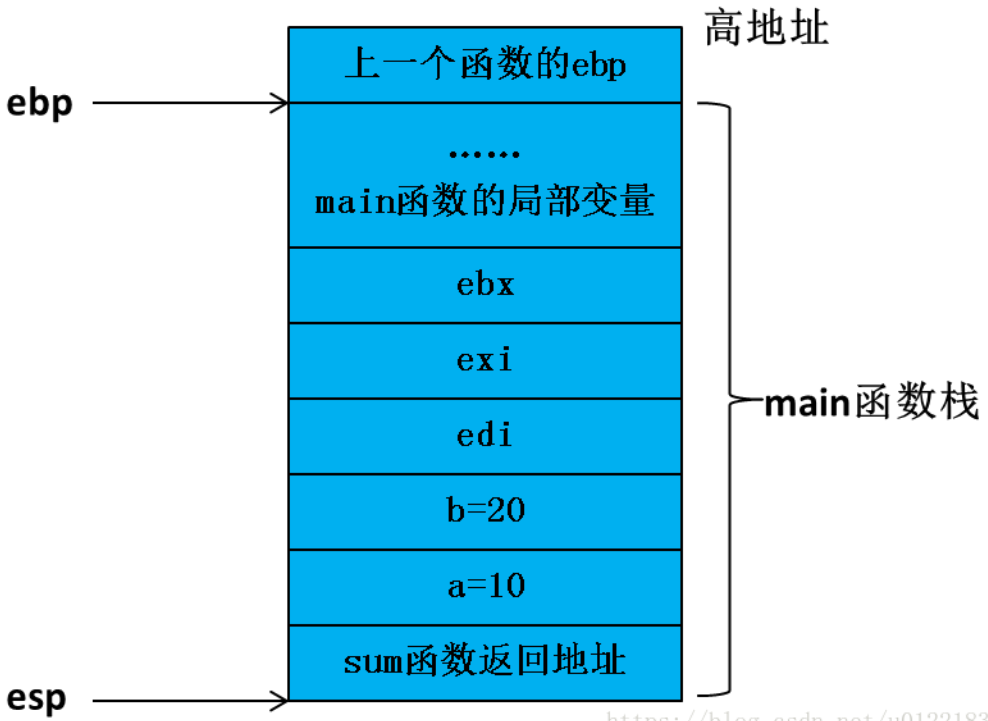
  Upon calling calling the sum function, step 1, to push the function parameters into stack:



函数调用约定
Convention of Function Calling

- 第二步，将返回地址压入栈：

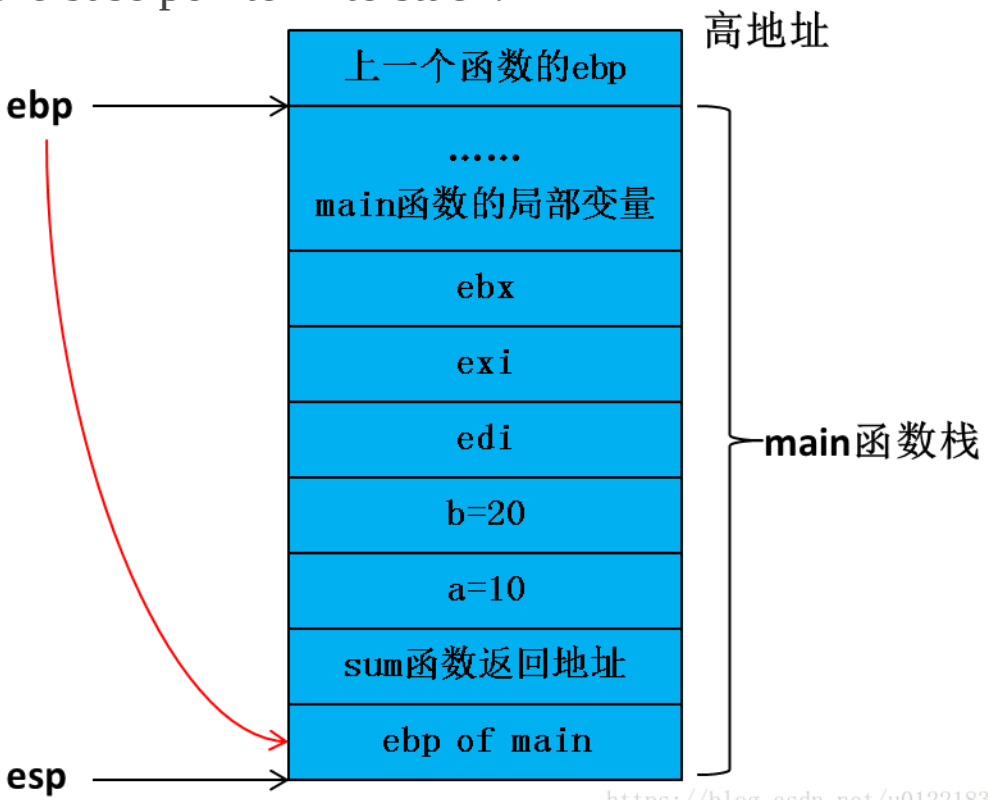
  Step 2, to push the return address into stack:



函数调用约定
Convention of Function Calling

- 第三步，将基指针压入栈：

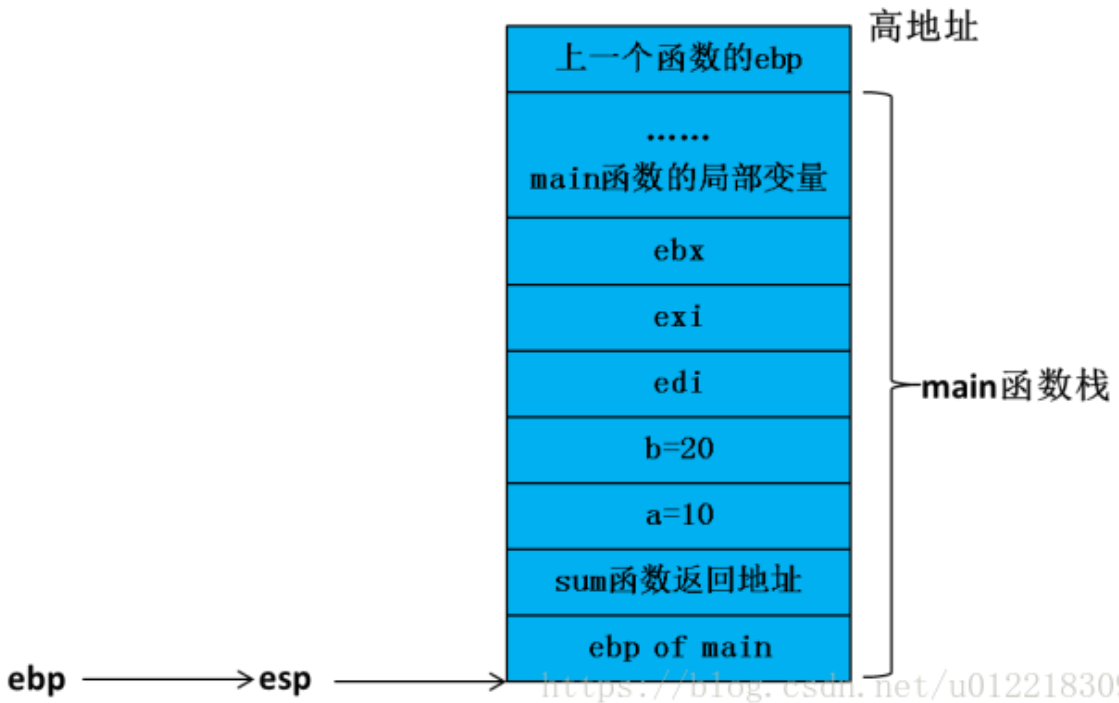
  Step 3, to push the base pointer into stack:



函数调用约定
Convention of Function Calling

- 第四步，将EBP的指针赋给ESP：

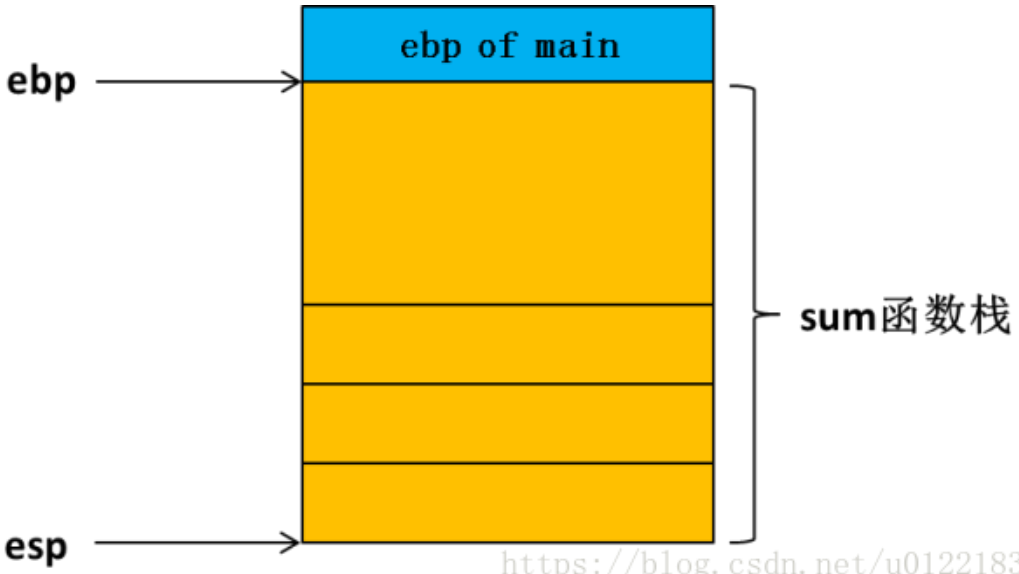
  Step 4, assign the value of EBP to ESP:



函数调用约定
Convention of Function Calling

- 第五步，将ESP下移动一段空间创建sum函数的栈的栈帧：

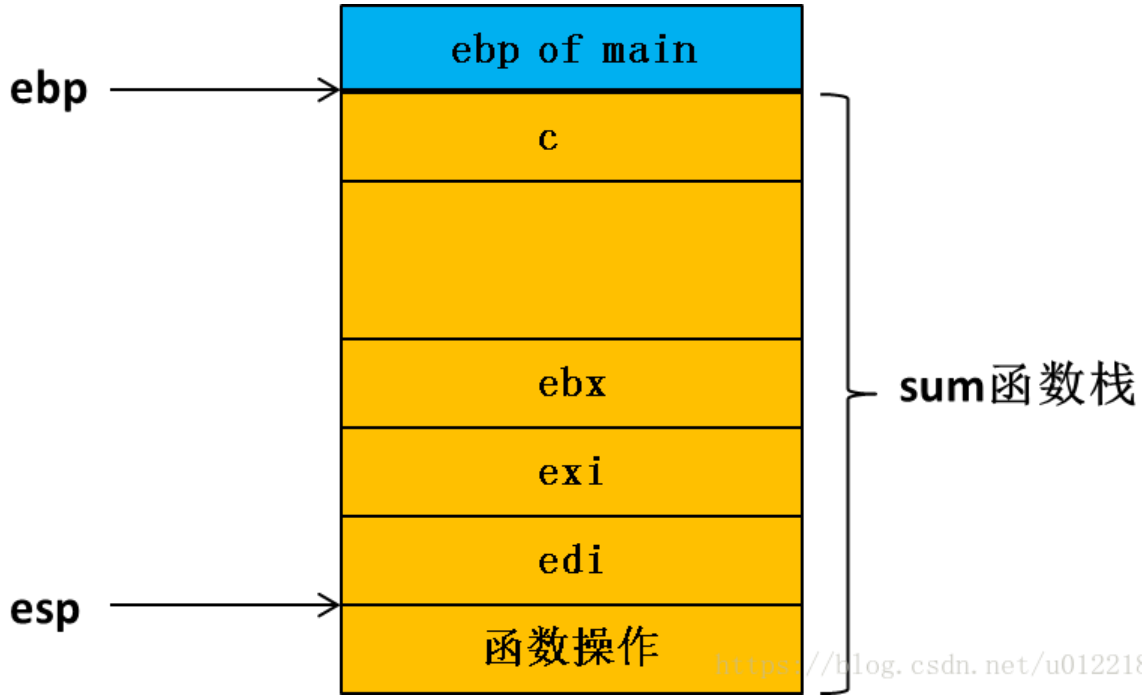
  Step 5, decrease ESP by some displacement to reserve space for stack frame:



函数调用约定
Convention of Function Calling

- 第六步，将进行加法操作，并将结果c放入[ebp-4]位置：

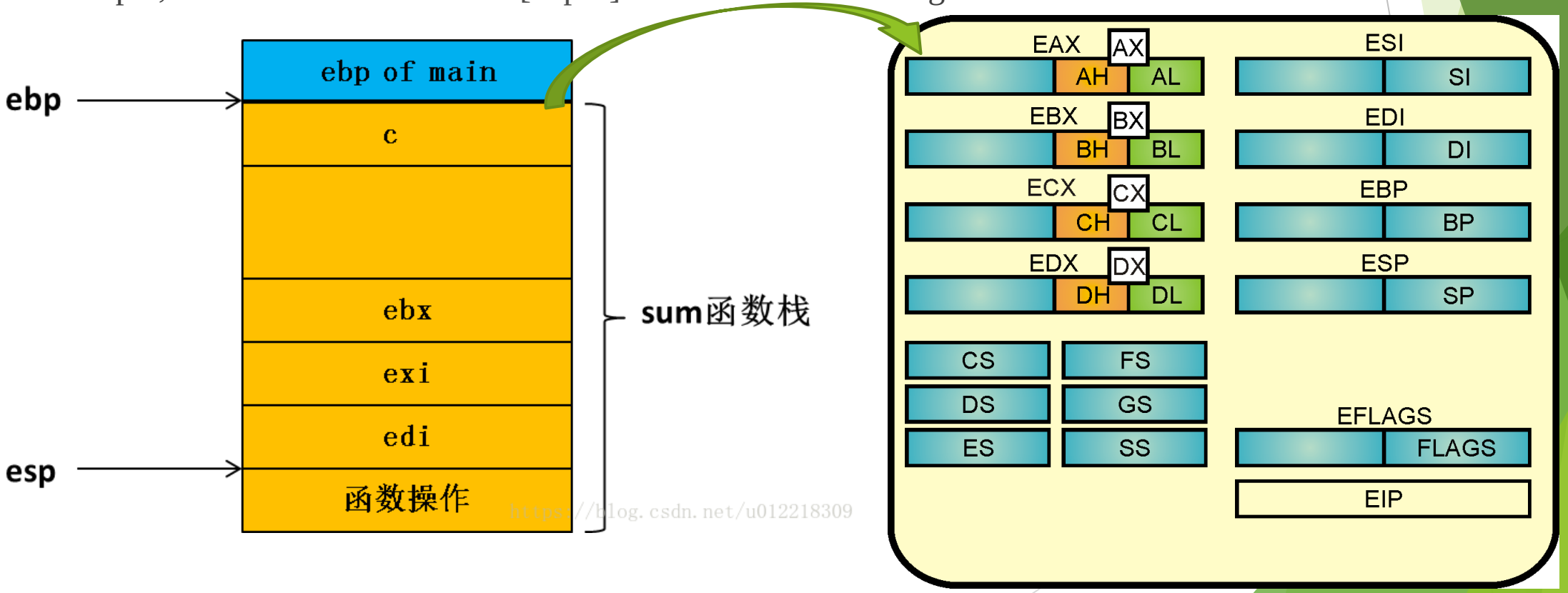
  Step 6, perform the addition and store the result to position [ebp-4]:



函数调用约定
Convention of Function Calling

- 第七步，将 [ebp-4]位置所存储之值放入EAX之中，准备返回：

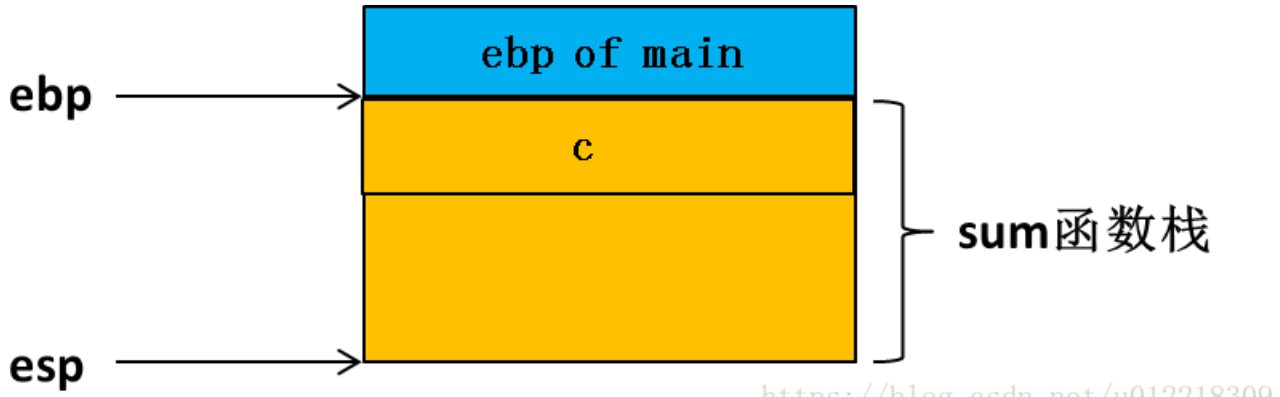
  Step 7, move the value stored at [ebp-4] into EAX for returning:



函数调用约定
Convention of Function Calling

- 第八步，依次弹出之前所压入之其它寄存器之值：

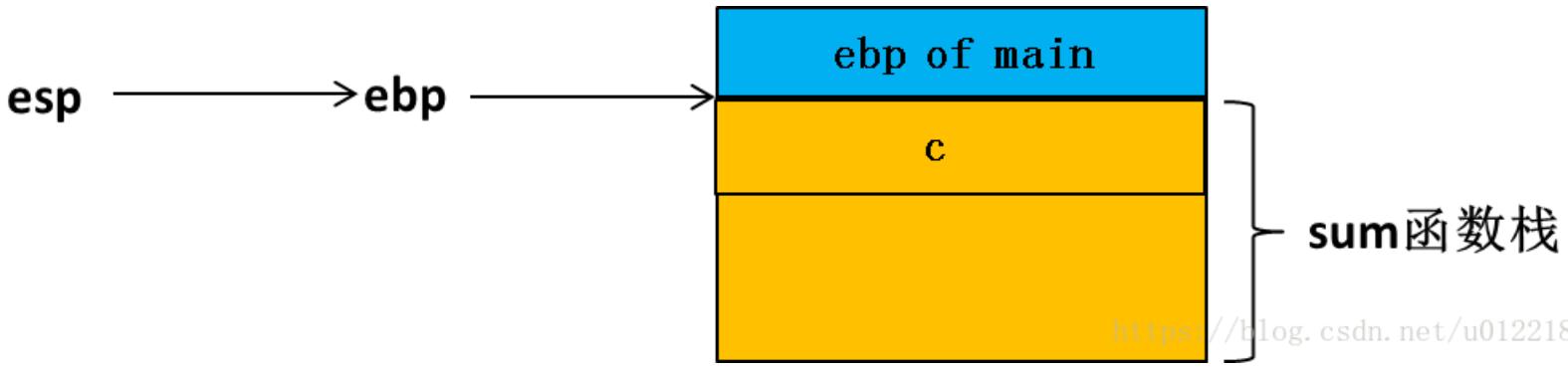
  Step 8, pop other register values that pushed previously:



函数调用约定
Convention of Function Calling

- 第九步，将ebp的值赋给esp，也就等于将esp指向ebp，销毁sum函数栈帧：

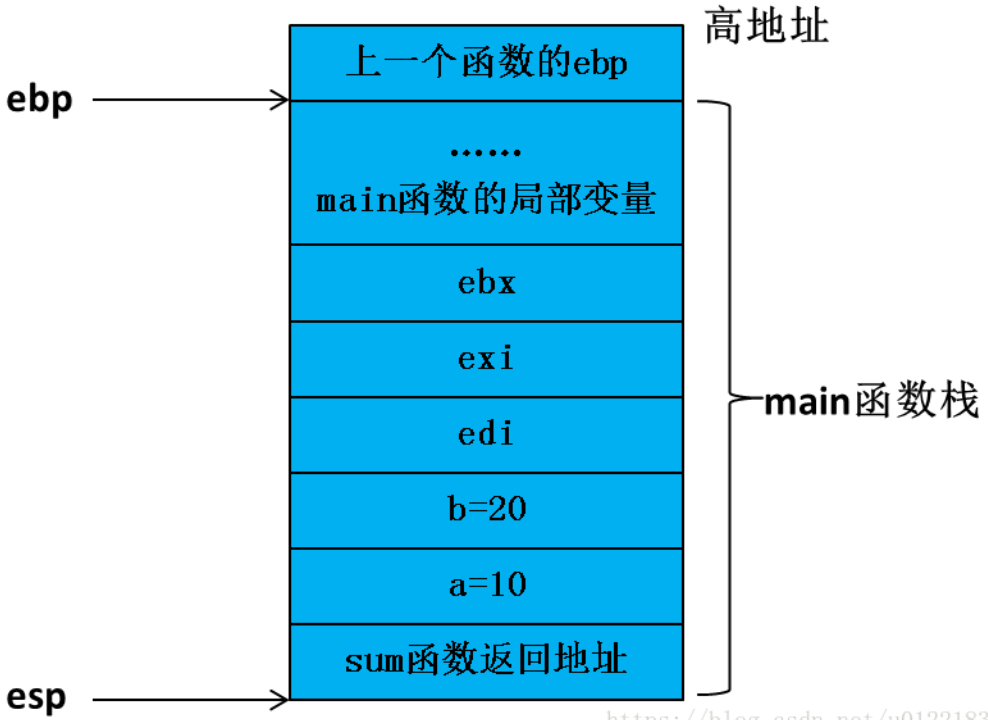
  Step 9, pop other register values that pushed previously:



函数调用约定
Convention of Function Calling

- 第十步，返回到main函数中，接着执行下一条语句：

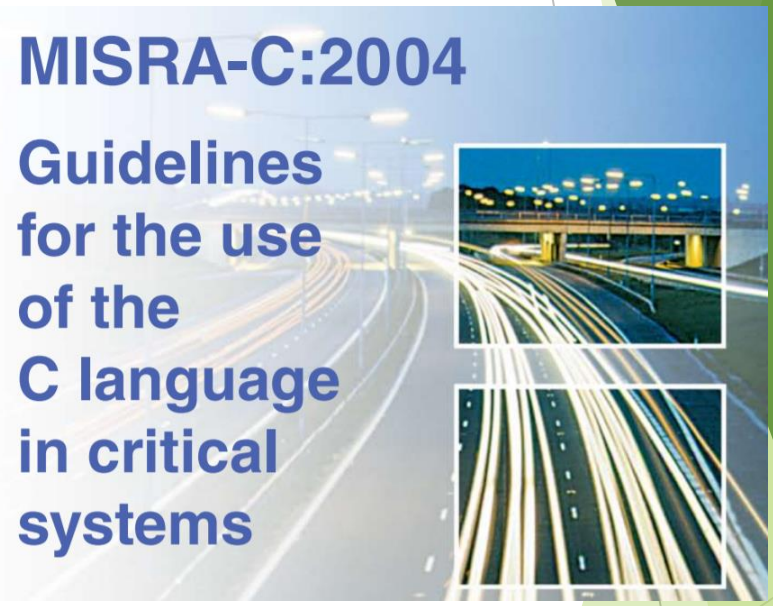
  Step 10, return to main function to continue execute the next instruction:

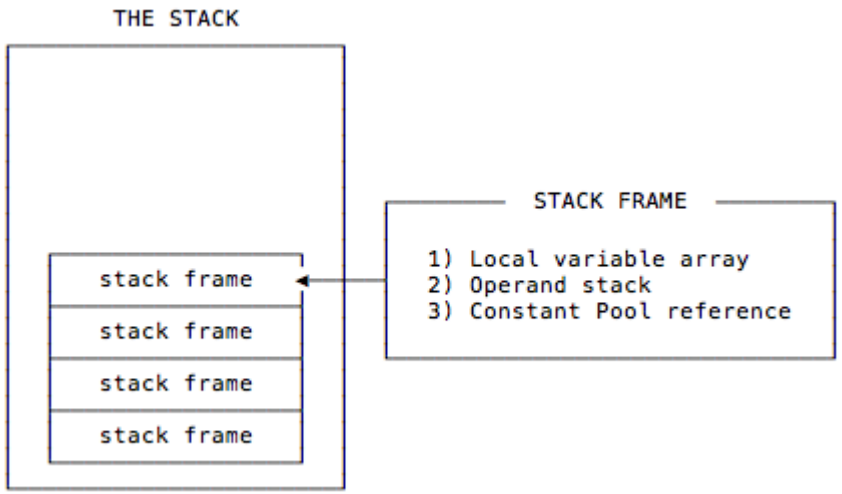


函数调用约定
Convention of Function Calling

- 如果递归调用，有可能导致很深的栈；因此在一些领域，比如汽车软件，考虑到安全性，会禁用递归。

  Recursion might incur exceeding depth of stacks, so in some domains, such as automotive software, recursion is prohibited from the security perspective.





Rule 16.2 (required): **Functions shall not call themselves, either directly or indirectly.**

This means that recursive function calls cannot be used in safety-related systems. Recursion carries with it the danger of exceeding available stack space, which can be a serious error. Unless recursion is very tightly controlled, it is not possible to determine before execution what the worst-case stack usage could be.

递归的类型
Types of Recursion

- 递归有两种类型，一种是常见的递归，一种是尾递归

  There are two types of recursion, the first is the normal recursion, and the second is called the tail recursion.

- 普通递归的例子（Example of normal recursion）

```
def recsum(x):
  if x == 1:
    return x
  else:
    return x + recsum(x - 1)
```

```
recsum(5)
5 + recsum(4)
5 + (4 + recsum(3))
5 + (4 + (3 + recsum(2)))
5 + (4 + (3 + (2 + recsum(1))))
5 + (4 + (3 + (2 + 1)))
5 + (4 + (3 + 3))
5 + (4 + 6)
5 + 10
15
```

递归的类型
Types of Recursion

- 尾递归的例子（Tail-recursion example）
- 尾递归的栈帧大小恒定，所以有时候可以考虑把普通递归改为尾巴递归。

  The stack frame of tail-recursion is fixed, so sometimes, it is good to rewrite the normal recursion as tail-recursion.

```
def tailrecsum(x, running_total=0):
  if x == 0:
    return running_total
  else:
    return tailrecsum(x - 1, running_total + x)
```

```
tailrecsum(5, 0)
tailrecsum(4, 5)
tailrecsum(3, 9)
tailrecsum(2, 12)
tailrecsum(1, 14)
tailrecsum(0, 15)
15
```

分治与递归
Divide-and-Conquer and Recursion

- 在有了对递归的了解之后，我们概述一下利用分治思想解决问题的步骤：
  - 将问题划分为多个子问题，这些子问题是同一问题的较小实例。
  - 通过递归解决子问题来解决原问题。特别地，如果子问题的大小足够小，只需以简单的方式解决子问题。
  - 将子问题的解组合成原问题的解。

After a preliminary understanding of recursion, the following summarizes the procedures by taking advantage of divide-and-conquer philosophy:

- Divide the problem into a number of sub-problems that are of smaller scale of the original problem.
- Conquer the sub-problems by solving them recursively. If the sizes of the sub-problems are small enough, just solve the subproblems in a straightforward manner.
- Combine the solutions to the subproblems into the solution for the original problem.

分治与递归
Divide-and-Conquer and Recursion

- 我们以合并排序为例子，来说明按照分治思想，利用递归方法，排序一个数组：
  - 划分：将待排序的n个元素序列划分为两个子序列，每个子序列有n/2个元素。
  - 征服：使用归并排序递归地对两个子序列进行排序。
  - 合并：合并两个排序的子序列以产生排序的答案。

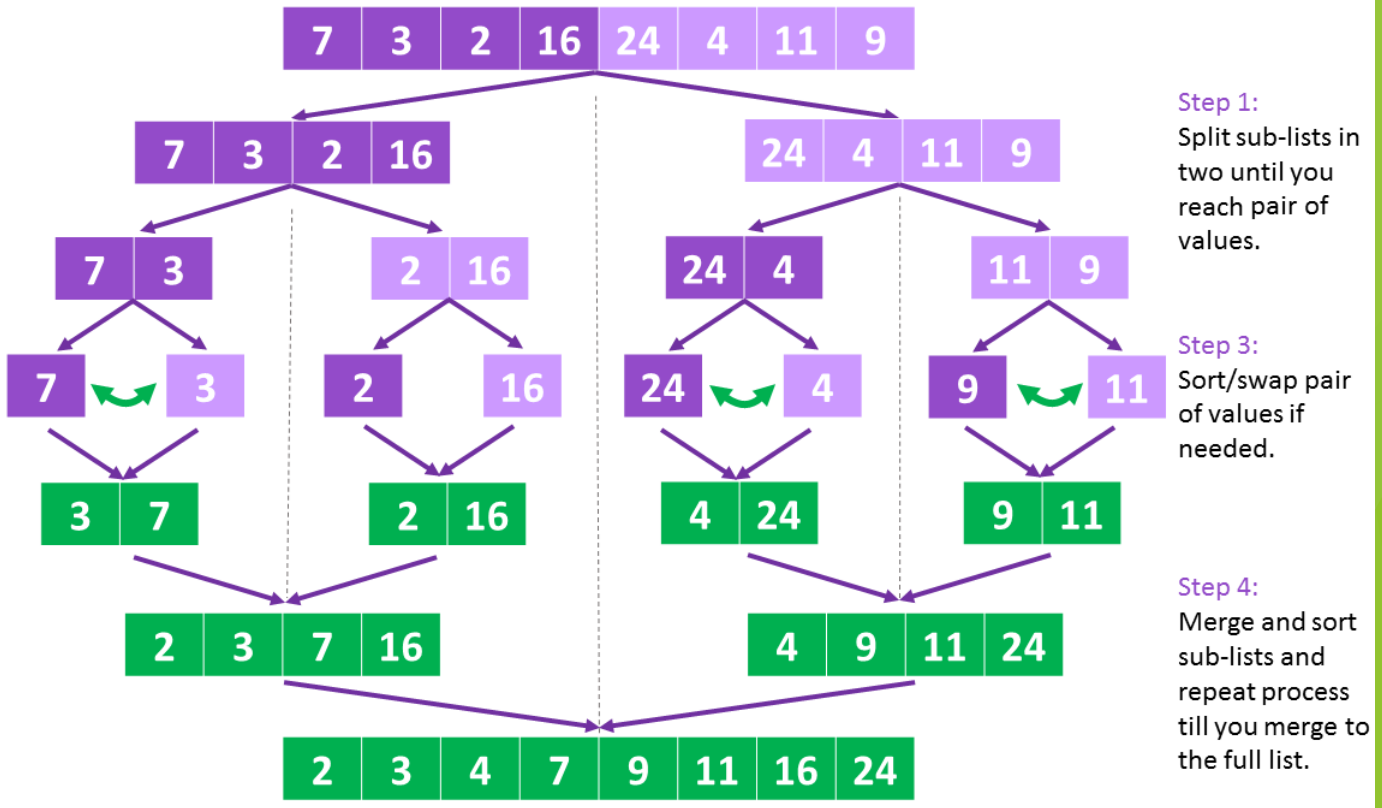
We demonstrate how to use recursion by manifestation of divide-and-conquer by considering the merge-sort to re-order an array:

- Divide: Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.
- Conquer: Sort the two subsequences recursively using merge sort.
- Combine: Merge the two sorted subsequences to produce the sorted answer.

分治与递归
Divide-and-Conquer and Recursion

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

Merge Sort



递归算法
Recursion

- 如前所述，特定算法的规模不仅体现在输入的数量上，对单个输入变量，也体现在数字的大小上。对于输入的数量，可以用分而治之的方法，通过递归算法解决，同样，对于涉及数值非常大的问题，也可以递归算法，将数值降为较小的方便处理的值进行处理。例如，求两个数的最大公约数就是此种情况。

  As aforementioned, the scale for a specific algorithm is not only reflected in the number of the input, but also the magnitude of the input value. For input of exceedingly large elements, we can try to solve it by dividing and conquering in recursive method, however, for input of big number, the recursion can also reduce the input value into small value for easily processing. For example, to find the greatest common divisor belongs to the second case.

```
int GCD(int no1, int no2)
{
    if (no2 != 0)
        return GCD(no2, no1 % no2);
    else
        return no1;
}
```

递归算法
Recursion

- 下面简要说明下面方法的正确性。令$g = \gcd(x, y)$且$x > y$，若$r_1 = x\%y$，则$g = \gcd(y, r_1)$。这是因为$x - qy = r_1$，由于$g|x - qy$，则$g|r_1$，即$g \leq \gcd(x, y)$。若存在$d > g$，满足$d|r_1$且$d|y$，则$d|qy + r_1$，即$d|x$，但这与$g = \gcd(x, y)$矛盾。持续递归，会有序列$\gcd(x, y) = \gcd(y, r_1) = \gcd(r_1, r_2) = \cdots = \gcd(r_n, 0)$，且$r_n = d$。从这里可以看出，序列的长度在一定程度上取决于$x$与$y$的大小，当$x$与$y$值越大，越有可能引发较深的递归层次。

  We simply show the correctness of above algorithm. Let $g = \gcd(x, y)$ and $x > y$, if $r_1 = x\%y$, we have $g = \gcd(y, r_1)$. This is because $x - qy = r_1$, the fact $g|x - qy$ infers that $g|r_1$, so at least we have $g \leq \gcd(x, y)$. If further there exists d greater than g and satisfying $d|r_1$ and $d|y$, then $d|qy + r_1$, which means $d|x$, contradictory to the fact $g = \gcd(x, y)$. Consecutive recursion leads to the sequence $\gcd(x, y) = \gcd(y, r_1) = \gcd(r_1, r_2) = \cdots = \gcd(r_n, 0)$, and $r_n = d$. We just mention here that the length of the sequence to some extent depends on the magnitude of x and y, the greater the values, the potential of higher depth of recursion.

时间复杂度
Time Complexity

- 正如第一章所讲，上面的例子说明算法的执行时间与输入的规模存在一定的关系，我们需要正式的术语来表达这种关系。对于一个给定的算法，假设输入的规模为n，其对应的时间复杂度用$g(n)$表示，则我们用$\Theta\big(g(n)\big)$表示这样一类函数集合：$\Theta\big(g(n)\big) = \{f(n): \exists c_1, c_2, n \geq n_0 > 0, s.t. 0 < c_1 g(x) \leq f(n) \leq c_2 g(x)\}$。一般说来，当$n \to +\infty$时，$g(n) \to +\infty$，则$\Theta\big(g(n)\big)$表示与$g(n)$同阶无穷大的函数的集合。

  As aforementioned in the first lecture, the above example demonstrates that the time spent on executing an algorithm depends on the scale of input, and it needs terminologies to formally describe such relations. For a given algorithm, suppose the input scale is n and the corresponding time complexity measurement $g(n)$, $\Theta\big(g(n)\big)$ denotes the set as below: $\Theta\big(g(n)\big) = \{f(n): \exists c_1, c_2, n \geq n_0 > 0, s.t. 0 < c_1 g(x) \leq f(n) \leq c_2 g(x)\}$. Generally speaking, when n approaches infinite, so does $g(n)$, hence, $\Theta\big(g(n)\big)$ represents the set of functions which are all infinities of the same of order as $g(n)$.

时间复杂度
Time Complexity

- 如果$h(n) \in \Theta(g(n))$，则除了称$h(n)$与$g(n)$为同阶无穷大之外，也称$g(n)$为$h(n)$的渐进确界。除了同阶无穷大，还有高阶无穷大、低阶无穷大等概念。高阶无穷大与低阶无穷大有时也称渐进上界与渐进下界，这些概念一定程度上是为了学科的完整性，实践中用的最多的还是同阶无穷大。由于常数项系数并不重要，则通常会用最简形式，如$\Theta(n^2)$表示与平方增长同阶的时间复杂度算法的集合。其中高阶无穷大集合用$O(g(n))$表示，高阶无穷小集合用$\Omega(g(n))$表示。

  Suppose $h(n) \in \Theta(g(n))$, beside treating $h(n)$ and $g(n)$ as same-order infinities, it is also said $g(n)$ is the asymptotically tight bound of $h(n)$. In addition, there are also concepts such as lower-order infinities, higher-order infinities, etc. lower-order infinities and higher-order infinities are related to asymptotic lower bound and asymptotic upper bound, respectively. These terminologies sometimes are coined just for the completeness of the discipline, in practice the most used term is still same-order infinities. Due to the insignificance of the constant coefficient, some term is generally used in the simplest form, such as $\Theta(n^2)$, which means time complexity increases as a squared relation with input scale n. Given $h(n)$, its lower-order infinities are collected as $O(g(n))$, and lower-order infinities are represented as $\Omega(g(n))$.

时间复杂度
Time Complexity

- 一般说来，$\Theta(n^2)$已经是一个比较高的时间复杂度，通过利用适当的数据结构，如树结构，解决同类问题的另外一种算法有可能将时间复杂度降为$\Theta\left(n\log_2(n)\right)$。下面我们说明，$\Theta\left(n\log_2(n)\right)$确实是比$\Theta(n^2)$要来得小的时间复杂度：

  Generally, time complexity measured by $\Theta(n^2)$ is highly enough for practice. By utilizing some proper data structure, for example, tree structure, another algorithm for solving the same problem might only be endowed with a time complexity of $\Theta\left(n\log_2(n)\right)$. Now we show that $\Theta\left(n\log_2(n)\right)$ is indeed much better than $\Theta(n^2)$.

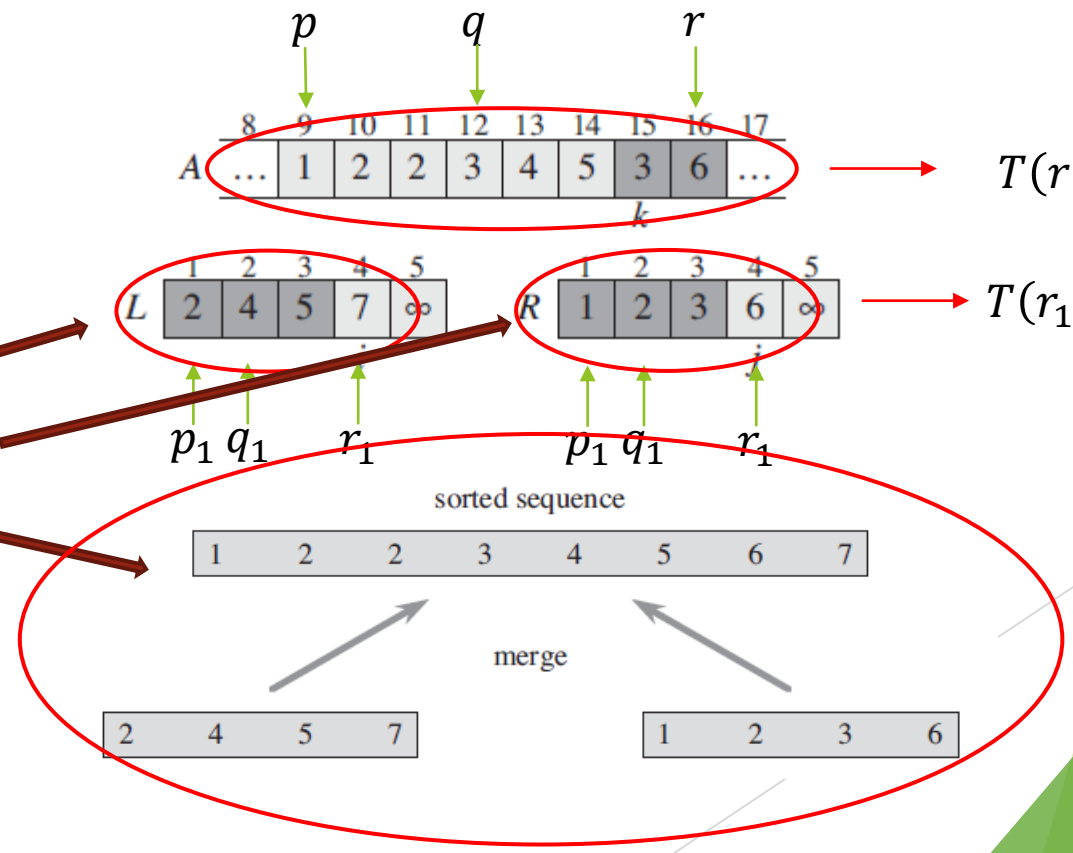
$$\lim_{n\to\infty}\frac{n\log_2(n)}{n^2}=\lim_{n\to\infty}\left(\frac{\log_2(n)}{n}\right)'=\lim_{n\to\infty}\frac{1}{n\ln 2}=0$$

  如上面推导所示，n^2趋近于正无穷的速度要比$n\log_2(n)$快得多，其导致$n\log_2(n)$比上n^2时，当$n\to\infty$时，比值为0。

时间复杂度
Time Complexity

- 下面，我们证明对第3讲所讲的合并排序，其时间复杂度为$\Theta(n\log_2(n))$

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```



$T(r-p+1)$

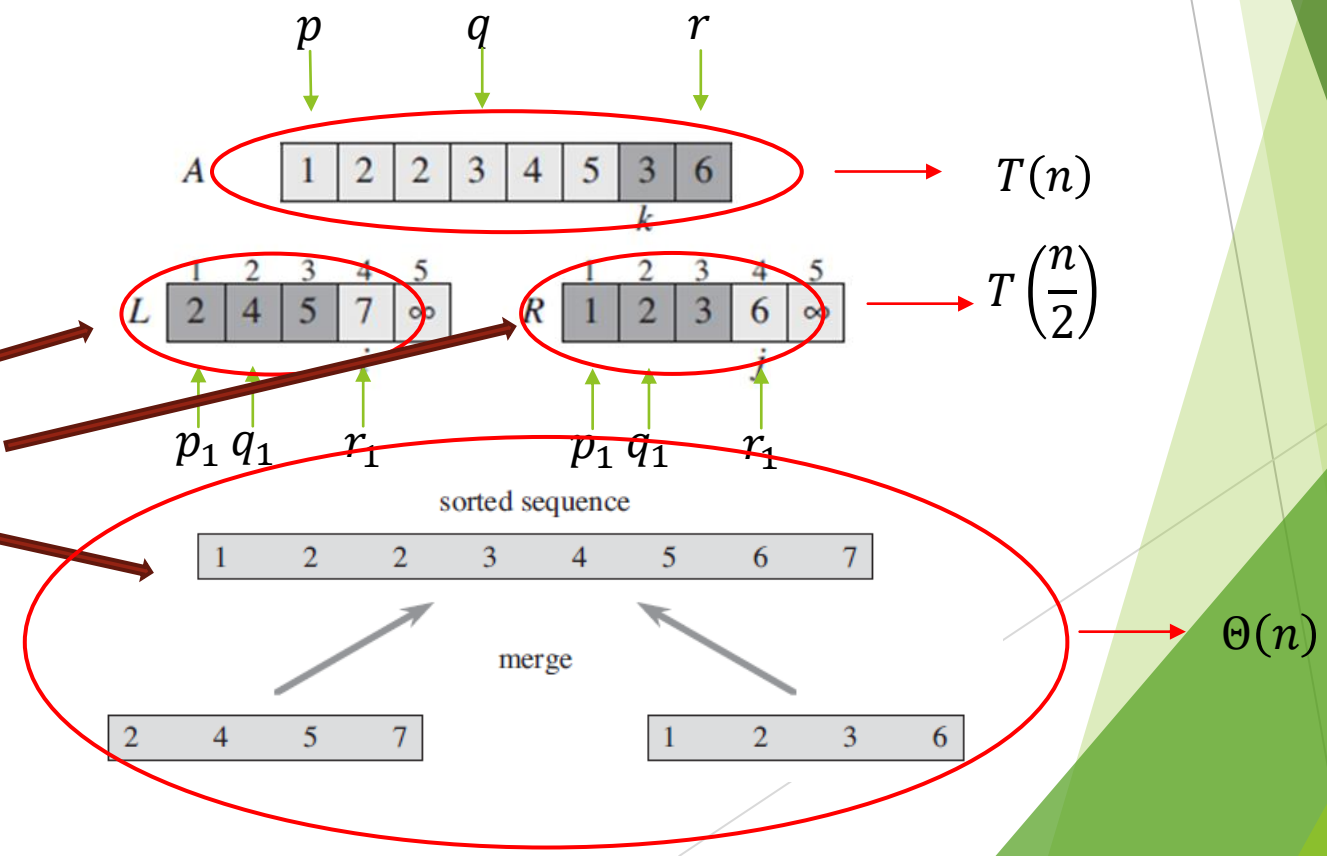
$T(r_1-p_1+1)=T\left(\frac{r-p+1}{2}\right)$

$\Theta(r-p+1)$

时间复杂度
Time Complexity

- 下面，我们证明对第3讲所讲的合并排序，其时间复杂度为$\Theta(n\log_2(n))$

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```



时间复杂度
Time Complexity

- 一般地，我们有如下式子成立：

  Generally, we have the following formula holds:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

- 我们只就一种特例用数学归纳法证明：$T(n) = \Theta\left(n\log_2(n)\right)$。考虑如下情况，假设$n$正好是2的$k$次方，即$n = 2^k$。当$k = 1$即$n = 2$时，因为仅有两个元素，对一个元素进行递归时，其总是排好顺序的，因此两个元素时，仅涉及合并，其复杂度仅涉及对元素作一次比较，然后排在一起，其复杂度不超过2。由于$2T\left(\frac{n}{2}\right) = 2T(1) = 0$，所以$T(2) = \Theta(2)$。

时间复杂度
Time Complexity

- 假设$k=l$时成立，即$T\left(n=2^k\right)=\Theta\left(2^k\log_2\left(2^k\right)\right)$，考虑$k=l+1$；此时

$$T\left(n=2^{k+1}\right)=2T\left(\frac{2^{k+1}}{2}\right)+\Theta\left(2^{k+1}\right)=2\cdot\Theta\left(2^k\log_2\left(2^k\right)\right)+\Theta\left(2^{k+1}\right)$$
$$=\Theta\left(2^{k+1}\left(\log_2\left(2^k\right)+\log_2 2\right)\right)=\Theta\left(2^{k+1}\left(\log_2 2^k\cdot 2\right)\right)=\Theta\left(2^{k+1}\left(\log_2 2^{k+1}\right)\right)$$
$$=\Theta\left(n\left(\log_2 n\right)\right)$$

- 当n不为2的k次方时，时间复杂度夹在$\Theta\left(2^{\lfloor\log_2 n\rfloor}\left(\log_2 2^{\lfloor\log_2 n\rfloor}\right)\right)$与$\Theta\left(2^{\lceil\log_2 n\rceil}\left(\log_2 2^{\lceil\log_2 n\rceil}\right)\right)$之间，可以认为即$\Theta\left(n\left(\log_2 n\right)\right)$

问题
Problems

1. 合并排序的伪代码如下图，请转化为基于STL的C++代码

   The pseudo-code of merge sort is as follows, please implement it in C++ based on STL

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
1   n1 = q − p + 1
2   n2 = r − q
3   let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
4   for i = 1 to n1
5       L[i] = A[p + i − 1]
6   for j = 1 to n2
7       R[j] = A[q + j]
8   L[n1 + 1] = ∞
9   R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

问题
Problems

2. 对给定的数列，利用递归算法，求数列中的最大值。Python代码如下，请转化为C++代码。

   For given array, find the maximal element based on recursive method. The Python script is already given, please re-implement it in C++

```python
def findmax(arr):
    if len(arr) == 2: # base case
        return arr[0] if arr[0]>arr[1] else arr[1]
    # recursive case
    return arr[0] if arr[0] > findmax(arr[1:]) else findmax(arr[1:])
```