Data Structures: Basic Concepts

Wei-Mei Chen

Department of Electronic and Computer Engineering
National Taiwan University of Science and Technology

System Life Cycle

1. Good programmers regard large-scale computer programs as systems that contain many complex interacting parts.
2. As systems, these programs undergo a development process called the **system life cycle**.
    - Requirements
    - Analysis: **bottom-up** vs. **top-down**
    - Design: **data objects** and **operations**
    - Refinement and Coding
    - Verification
        - Correctness proofs
        - Testing
        - Debugging

DMA

Pointers

- The two most important operators used with the pointer type:
  - &: the address operator
  - *: the dereferencing (or indirection) operator

```
int *pi, i;
```

- `if (pi==NULL)` $\equiv$ `if (!pi)`
- Heap: storage at run-time

Pointers

- The two most important operators used with the pointer type:
  - &: the address operator
  - *: the dereferencing (or indirection) operator

```
int *pi, i;
pi = &i;
```

- `if (pi==NULL)` $\equiv$ `if (!pi)`
- Heap: storage at run-time

Pointers

- The two most important operators used with the pointer type:
  - &: the address operator
  - *: the dereferencing (or indirection) operator

```
int *pi, i;
pi = &i;
i=10;
*pi = 10;
```

- `if (pi==NULL)` $\equiv$ `if (!pi)`
- Heap: storage at run-time

Allocation and Deallocation

- **malloc** and **free**
  - **void ***
  - type cast
- **After freeing any memory, reset the variable to NULL.**

Allocation and deallocation of memory

```
int i, *pi;
float f, *pf;
pi = (int *) malloc(sizeof(int));
pf = (float *) malloc(sizeof(float));
*pi = 1024;
*pf = 3.14;
printf("an integer = %d, a float = %f\n", *pi, *pf);
free(pi);
pi = NULL;
free(pf);
pf = NULL;
```

malloc

- If the memory is available, a pointer to the start of an area of memory of the required size is returned
- When the requested memory is not available, the pointer NULL is returned

```
if(!(pi = (int *) malloc(sizeof(int)))||
      !(pf = (float *) malloc(sizeof(float))))
{
      fprintf(stderr, "Insufficient memory");
      exit(EXIT_FAILURE);
}
```

```
#define MALLOC(p,s)\
if(!(p) = malloc(s)) {\
 fprintf(stderr, "Insufficient memory"); \
 exit(EXIT_FAILURE);\
}
```

```
MALLOC(pi,sizeof(int));
MALLOC(pf,sizeof(float));
```

Dangling Reference

Dangling pointers and wild pointers in computer programming are pointers that do not point to a valid object of the appropriate type.

Example

```
int i, *pi;
float f, *pf;
pi = (int *) malloc(sizeof(int));
pf = (float *) malloc(sizeof(float));
*pi = 1024;   *pf = 3.14;
printf("an integer = %d, a float = %f\n", *pi, *pf);
pf = (float *) malloc(sizeof(float));⟸
free(pi); pi = NULL;
free(pf); pf = NULL;
```

☞ The pointer to the storage used to hold the value 3.14 disappears.

Algorithm Specification

Algorithms

An **algorithm** is a finite set of instructions that, if followed, accomplishes a particular task. All algorithms must satisfy the following criteria.

1. **Input**: Zero or more quantities are externally supplied.
2. **Output**: At least one quantity is produced.
3. **Definiteness**: Each instruction is clear and unambiguous.
4. **Finiteness**: If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. **Effectiveness**: Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. ➠ **feasible**

Description of Algorithms

- Pseudocode: is an English-like representation of the algorithm logic.
  - English part
  - code part
- Algorithm Header
  - **Purpose**, **Condition**, and **Return**
- Statement Numbers
- Statement Constructs
  - It consists of an extended version of the basic algorithmic constructs: sequence, selection, and iteration.

Example: Chang-Robert's Algorithm

Input: All processors $P_1, P_2, \ldots, P_n$ form a unidirectional ring $\mathcal{R}$.

Output: The processor with the minimum identity declares itself the leader.

Elect-Leader($\mathcal{R}$)

for all P_i **do**

$m_i :=$ the identity of P_i

send m_i to the next processor

$t_i := \infty$

while $(t_i \neq m_i)$

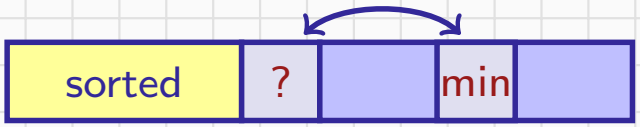
receive t_i from the previous processor

if $t_i < m_i$, **then** $m_i := t_i$; send m_i

End.

Selection Sort

- Consider $\{23, 78, 45, 8, 32, 56\}$
  From those integers that are currently unsorted, **find the smallest and place it next in the sorted list**.
- Sort list array by selecting smallest element in unsorted portion and exchanging it with element at the beginning of the unsorted list.




```
void swap ( int *x, int *y )
{
  int temp = *x;
  *x = *y;
  *y = temp;
}
```

swap(&a, &b);

https://www.geeksforgeeks.org/difference-between-call-by-value-and-call-by-reference/

macro!!!

```
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
```

Example: swap()

```
#include <stdio.h>
void swap(int *, int *); //prototype of the function
int main() {
    int a = 10;
    int b = 20;
    printf("Before swapping:  a = %d, b = %d\n", a, b);
    swap(&a, &b);
    printf("After swapping:  a = %d, b = %d\n", a, b);
}
void swap (int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
    printf("After swapping a = %d, b = %d\n",*x,*y);
}
```

```
#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
void sort(int [], int );
void main(void)
{  int i, n;
   int list[MAX_SIZE];
   printf("Enter the number of numbers to generate: ");
   scanf("%d", &n);
   if( n < 1 || n > MAX_SIZE){
      fprintf(stderr, "Improper value of n\n");
      exit(EXIT_FAILURE);
   }
   for(i = 0; i < n ; i++) {
      list[i] = rand( ) % 1000;
      printf("%d ",list[i]);
    }
    sort(list,n);
    printf("\n Sorted array:\n ");
    for(i = 0; i < n ; i++)
      printf("%d ",list[i]);
    printf("\n");
}

void sort(int list[], int n)
{  int i, j, min, temp;
   for(i = 0; i < n-1; i++)  {
     min = i;
     for(j = i+1 ; j < n ; j++)
     if(list[j] < list[min])
        min = j;
     SWAP(list[i],list[min],temp);
   }
}
```

```
#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
void sort(int [], int );
void main(void)
{  int i, n;
   int list[MAX_SIZE];
   printf("Enter the number of numbers to generate: ");
   scanf("%d", &n);
   if( n < 1 || n > MAX_SIZE){
      fprintf(stderr, "Improper value of n\n");
      exit(EXIT_FAILURE);
   }
   for(i = 0; i < n ; i++) {
      list[i] = rand( ) % 1000;
      printf("%d ",list[i]);
    }
    sort(list,n);
    printf("\n Sorted array:\n ");
    for(i = 0; i < n ; i++)
      printf("%d ",list[i]);
    printf("\n");
}

void sort(int list[], int n)
{  int i, j, min, temp;
   for(i = 0; i < n-1; i++)  {
     min = i;
     for(j = i+1 ; j < n ; j++)
     if(list[j] < list[min])
         min = j;
     SWAP(list[i],list[min],temp);
   }
}
```

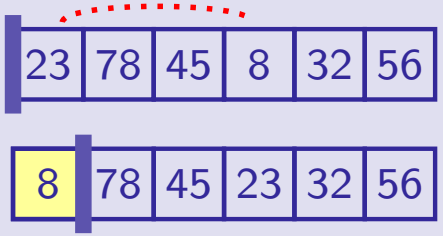
23	78	45	8	32	56

```
#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
void sort(int [], int );
void main(void)
{  int i, n;
   int list[MAX_SIZE];
   printf("Enter the number of numbers to generate: ");
   scanf("%d", &n);
   if( n < 1 || n > MAX_SIZE){
      fprintf(stderr, "Improper value of n\n");
      exit(EXIT_FAILURE);
   }
   for(i = 0; i < n ; i++) {
      list[i] = rand( ) % 1000;
      printf("%d ",list[i]);
    }
    sort(list,n);
    printf("\n Sorted array:\n ");
    for(i = 0; i < n ; i++)
      printf("%d ",list[i]);
    printf("\n");
}

void sort(int list[], int n)
{  int i, j, min, temp;
   for(i = 0; i < n-1; i++)  {
     min = i;
     for(j = i+1 ; j < n ; j++)
     if(list[j] < list[min])
         min = j;
     SWAP(list[i],list[min],temp);
   }
}
```

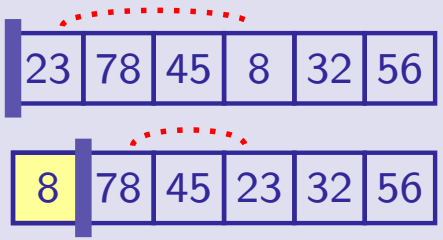
```
#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
void sort(int [], int );
void main(void)
{  int i, n;
   int list[MAX_SIZE];
   printf("Enter the number of numbers to generate: ");
   scanf("%d", &n);
   if( n < 1 || n > MAX_SIZE){
      fprintf(stderr, "Improper value of n\n");
      exit(EXIT_FAILURE);
   }
   for(i = 0; i < n ; i++) {
      list[i] = rand( ) % 1000;
      printf("%d ",list[i]);
    }
    sort(list,n);
    printf("\n Sorted array:\n ");
    for(i = 0; i < n ; i++)
      printf("%d ",list[i]);
    printf("\n");
}

void sort(int list[], int n)
{  int i, j, min, temp;
   for(i = 0; i < n-1; i++)  {
     min = i;
     for(j = i+1 ; j < n ; j++)
     if(list[j] < list[min])
         min = j;
     SWAP(list[i],list[min],temp);
   }
}
```



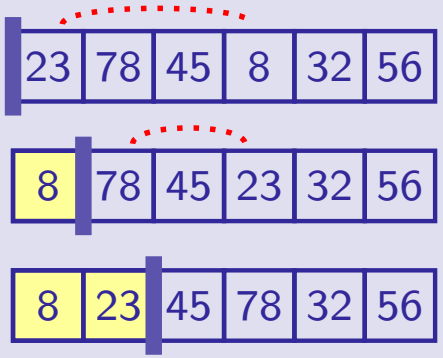
```
#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
void sort(int [], int );
void main(void)
{  int i, n;
   int list[MAX_SIZE];
   printf("Enter the number of numbers to generate: ");
   scanf("%d", &n);
   if( n < 1 || n > MAX_SIZE){
      fprintf(stderr, "Improper value of n\n");
      exit(EXIT_FAILURE);
   }
   for(i = 0; i < n ; i++) {
      list[i] = rand( ) % 1000;
      printf("%d ",list[i]);
    }
    sort(list,n);
    printf("\n Sorted array:\n ");
    for(i = 0; i < n ; i++)
      printf("%d ",list[i]);
    printf("\n");
}

void sort(int list[], int n)
{  int i, j, min, temp;
   for(i = 0; i < n-1; i++)  {
     min = i;
     for(j = i+1 ; j < n ; j++)
     if(list[j] < list[min])
         min = j;
     SWAP(list[i],list[min],temp);
   }
}
```



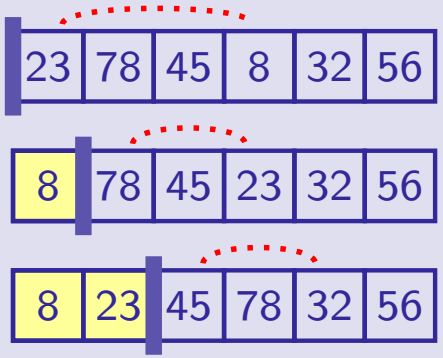
```
#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
void sort(int [], int );
void main(void)
{  int i, n;
   int list[MAX_SIZE];
   printf("Enter the number of numbers to generate: ");
   scanf("%d", &n);
   if( n < 1 || n > MAX_SIZE){
      fprintf(stderr, "Improper value of n\n");
      exit(EXIT_FAILURE);
   }
   for(i = 0; i < n ; i++) {
      list[i] = rand( ) % 1000;
      printf("%d ",list[i]);
    }
    sort(list,n);
    printf("\n Sorted array:\n ");
    for(i = 0; i < n ; i++)
      printf("%d ",list[i]);
    printf("\n");
}

void sort(int list[], int n)
{  int i, j, min, temp;
   for(i = 0; i < n-1; i++)  {
     min = i;
     for(j = i+1 ; j < n ; j++)
     if(list[j] < list[min])
         min = j;
     SWAP(list[i],list[min],temp);
   }
}
```



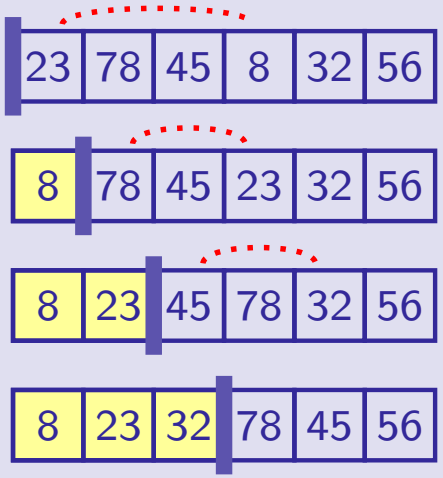
```
#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
void sort(int [], int );
void main(void)
{  int i, n;
   int list[MAX_SIZE];
   printf("Enter the number of numbers to generate: ");
   scanf("%d", &n);
   if( n < 1 || n > MAX_SIZE){
      fprintf(stderr, "Improper value of n\n");
      exit(EXIT_FAILURE);
   }
   for(i = 0; i < n ; i++) {
      list[i] = rand( ) % 1000;
      printf("%d ",list[i]);
    }
    sort(list,n);
    printf("\n Sorted array:\n ");
    for(i = 0; i < n ; i++)
      printf("%d ",list[i]);
    printf("\n");
}

void sort(int list[], int n)
{  int i, j, min, temp;
   for(i = 0; i < n-1; i++)  {
     min = i;
     for(j = i+1 ; j < n ; j++)
     if(list[j] < list[min])
         min = j;
     SWAP(list[i],list[min],temp);
   }
}
```



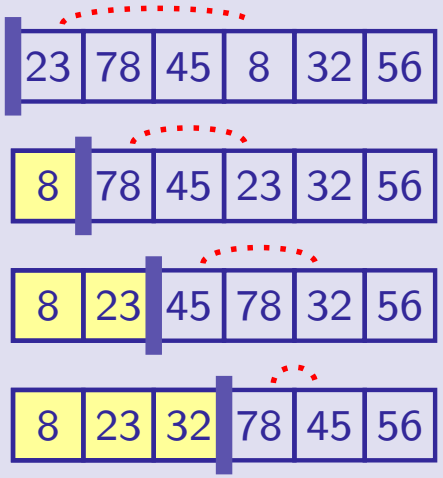
```
#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
void sort(int [], int );
void main(void)
{  int i, n;
   int list[MAX_SIZE];
   printf("Enter the number of numbers to generate: ");
   scanf("%d", &n);
   if( n < 1 || n > MAX_SIZE){
      fprintf(stderr, "Improper value of n\n");
      exit(EXIT_FAILURE);
   }
   for(i = 0; i < n ; i++) {
      list[i] = rand( ) % 1000;
      printf("%d ",list[i]);
    }
    sort(list,n);
    printf("\n Sorted array:\n ");
    for(i = 0; i < n ; i++)
      printf("%d ",list[i]);
    printf("\n");
}

void sort(int list[], int n)
{  int i, j, min, temp;
   for(i = 0; i < n-1; i++)  {
     min = i;
     for(j = i+1 ; j < n ; j++)
     if(list[j] < list[min])
         min = j;
     SWAP(list[i],list[min],temp);
   }
}
```



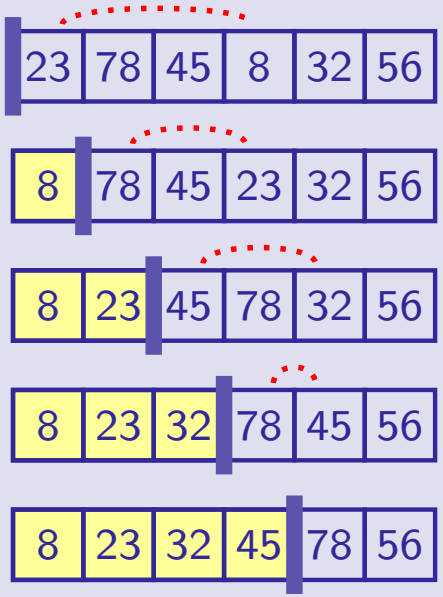
```
#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
void sort(int [], int );
void main(void)
{  int i, n;
   int list[MAX_SIZE];
   printf("Enter the number of numbers to generate: ");
   scanf("%d", &n);
   if( n < 1 || n > MAX_SIZE){
      fprintf(stderr, "Improper value of n\n");
      exit(EXIT_FAILURE);
   }
   for(i = 0; i < n ; i++) {
      list[i] = rand( ) % 1000;
      printf("%d ",list[i]);
    }
    sort(list,n);
    printf("\n Sorted array:\n ");
    for(i = 0; i < n ; i++)
      printf("%d ",list[i]);
    printf("\n");
}

void sort(int list[], int n)
{  int i, j, min, temp;
   for(i = 0; i < n-1; i++)  {
     min = i;
     for(j = i+1 ; j < n ; j++)
     if(list[j] < list[min])
         min = j;
     SWAP(list[i],list[min],temp);
   }
}
```



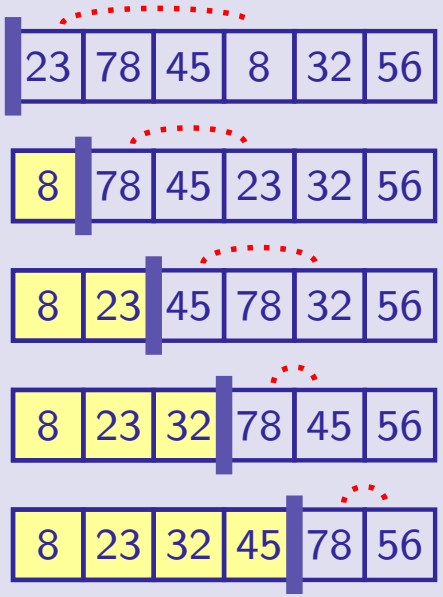
```
#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
void sort(int [], int );
void main(void)
{  int i, n;
   int list[MAX_SIZE];
   printf("Enter the number of numbers to generate: ");
   scanf("%d", &n);
   if( n < 1 || n > MAX_SIZE){
      fprintf(stderr, "Improper value of n\n");
      exit(EXIT_FAILURE);
   }
   for(i = 0; i < n ; i++) {
      list[i] = rand( ) % 1000;
      printf("%d ",list[i]);
    }
    sort(list,n);
    printf("\n Sorted array:\n ");
    for(i = 0; i < n ; i++)
      printf("%d ",list[i]);
    printf("\n");
}

void sort(int list[], int n)
{  int i, j, min, temp;
   for(i = 0; i < n-1; i++)  {
     min = i;
     for(j = i+1 ; j < n ; j++)
     if(list[j] < list[min])
         min = j;
     SWAP(list[i],list[min],temp);
   }
}
```



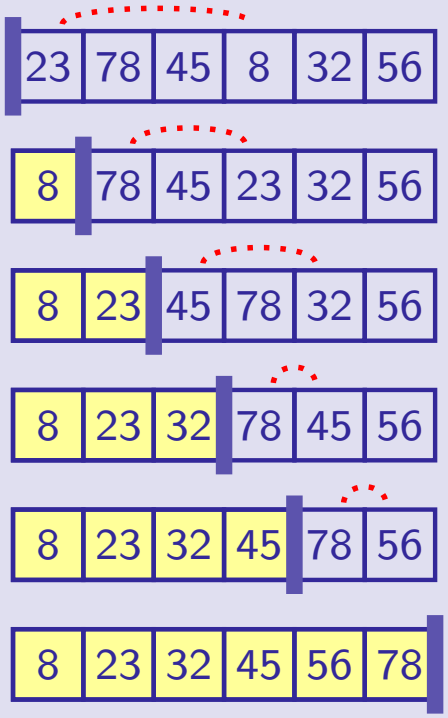
```
#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
void sort(int [], int );
void main(void)
{  int i, n;
   int list[MAX_SIZE];
   printf("Enter the number of numbers to generate: ");
   scanf("%d", &n);
   if( n < 1 || n > MAX_SIZE){
      fprintf(stderr, "Improper value of n\n");
      exit(EXIT_FAILURE);
   }
   for(i = 0; i < n ; i++) {
      list[i] = rand( ) % 1000;
      printf("%d ",list[i]);
    }
    sort(list,n);
    printf("\n Sorted array:\n ");
    for(i = 0; i < n ; i++)
      printf("%d ",list[i]);
    printf("\n");
}

void sort(int list[], int n)
{  int i, j, min, temp;
   for(i = 0; i < n-1; i++)  {
     min = i;
     for(j = i+1 ; j < n ; j++)
     if(list[j] < list[min])
         min = j;
     SWAP(list[i],list[min],temp);
   }
}
```



```
#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
void sort(int [], int );
void main(void)
{  int i, n;
   int list[MAX_SIZE];
   printf("Enter the number of numbers to generate: ");
   scanf("%d", &n);
   if( n < 1 || n > MAX_SIZE){
      fprintf(stderr, "Improper value of n\n");
      exit(EXIT_FAILURE);
   }
   for(i = 0; i < n ; i++) {
      list[i] = rand( ) % 1000;
      printf("%d ",list[i]);
    }
    sort(list,n);
    printf("\n Sorted array:\n ");
    for(i = 0; i < n ; i++)
      printf("%d ",list[i]);
    printf("\n");
}

void sort(int list[], int n)
{  int i, j, min, temp;
   for(i = 0; i < n-1; i++)  {
     min = i;
     for(j = i+1 ; j < n ; j++)
     if(list[j] < list[min])
         min = j;
     SWAP(list[i],list[min],temp);
   }
}
```



Binary Search

- Consider a **sorted** list: $\{8, 23, 32, 45, 56, 78\}$
  Figure out if *searchnum* is in the list.
  - YES ➠ i where list[i]=*searchnum*
  - NO ➠ -1

```
while (more integers to check) {
  middle = (left + right) / 2;
  if (searchnum < list[middle])
    right = middle - 1;
  else if (searchnum == list[middle])
    return middle;
  else left = middle + 1;
}
```

Binary Search

- Consider a **sorted** list: $\{8, 23, 32, 45, 56, 78\}$
  Figure out if *searchnum* is in the list.
  - YES ➠ i where list[i]=*searchnum*
  - NO ➠ -1

```
while (more integers to check) {
  middle = (left + right) / 2;
  if (searchnum < list[middle])
    right = middle - 1;
  else if (searchnum == list[middle])
    return middle;
  else left = middle + 1;
}
```

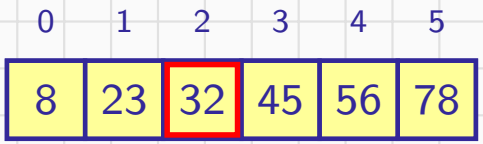
```
int compare ( int x, int y )
{
  if (x < y) return -1;
  else if (x == y) return 0;
  else return 1;
}
```

Search for 45

☞ left$\leftarrow 0$ right$\leftarrow n-1$

```
int binarysearch (int list[], int searchnum, int left, int right)
{
  int middle ;
  while (left <= right)
  {
     middle = (left + right)/2;
     switch(COMPARE(list[middle], searchnum)) {
       case -1: left = middle+1;
              break;
       case 0 : return middle;
       case 1 : right = middle−1
     }
  }
  return -1;
}
```

Search for 45

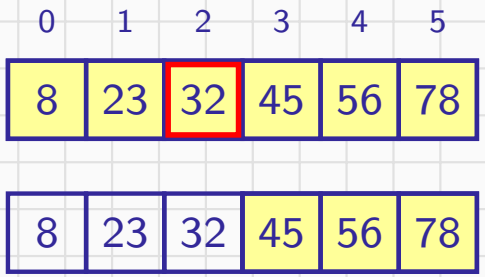


☞ left$\leftarrow 0$ right$\leftarrow n-1$

```
int binarysearch (int list[], int searchnum, int left, int right)
{
  int middle ;
  while (left <= right)
  {
     middle = (left + right)/2;
     switch(COMPARE(list[middle], searchnum)) {
       case -1: left = middle+1;
              break;
       case 0 : return middle;
       case 1 : right = middle−1
     }
  }
  return -1;
}
```

Search for 45

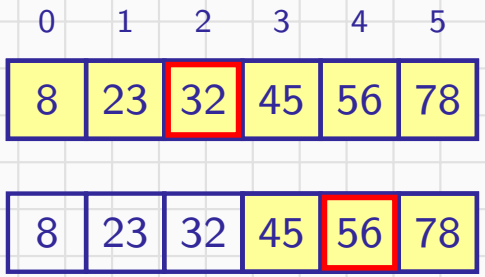
☞ left$\leftarrow 0$ right$\leftarrow n-1$




```
int binarysearch (int list[], int searchnum, int left, int right)
{
  int middle ;
  while (left <= right)
  {
     middle = (left + right)/2;
     switch(COMPARE(list[middle], searchnum)) {
       case -1: left = middle+1;
              break;
       case 0 : return middle;
       case 1 : right = middle−1
     }
  }
  return -1;
}
```

Search for 45

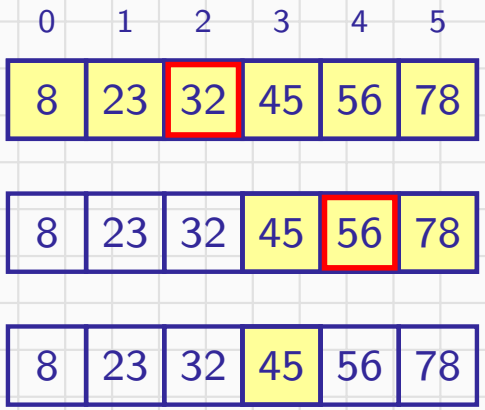
☞ left$\leftarrow 0$ right$\leftarrow n-1$




```
int binarysearch (int list[], int searchnum, int left, int right)
{
  int middle ;
  while (left <= right)
  {
     middle = (left + right)/2;
     switch(COMPARE(list[middle], searchnum)) {
       case -1: left = middle+1;
              break;
       case 0 : return middle;
       case 1 : right = middle−1
     }
  }
  return -1;
}
```

Search for 45

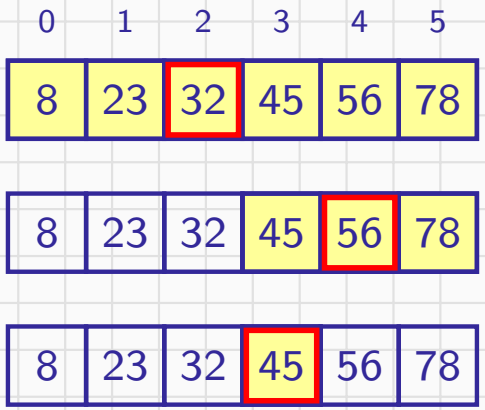
☞ left$\leftarrow 0$ right$\leftarrow n-1$




```
int binarysearch (int list[], int searchnum, int left, int right)
{
  int middle ;
  while (left <= right)
  {
     middle = (left + right)/2;
     switch(COMPARE(list[middle], searchnum)) {
       case -1: left = middle+1;
              break;
       case 0 : return middle;
       case 1 : right = middle−1
     }
  }
  return -1;
}
```

Search for 45

☞ left$\leftarrow 0$ right$\leftarrow n-1$




```
int binarysearch (int list[], int searchnum, int left, int right)
{
  int middle ;
  while (left <= right)
  {
     middle = (left + right)/2;
     switch(COMPARE(list[middle], searchnum)) {
       case -1: left = middle+1;
              break;
       case 0 : return middle;
       case 1 : right = middle−1
     }
  }
  return -1;
}
```

Recursive Algorithms

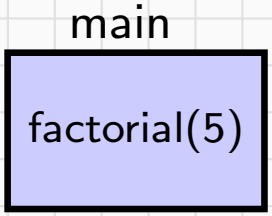
- A recursive algorithm is an algorithm which calls itself with "smaller" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller input.
- Example: factorial $n! = n \times (n-1)!$
- Binary search

```
int binarysearch (int list[], int searchnum, int left, int right)
{
 int middle ;
 while (left <= right)
 {
    middle = (left + right)/2;
    switch(COMPARE(list[middle], searchnum)) {
       case -1  : return binarysearch (list, searchnum, middle+1, right);
       case  0  : return middle;
       case  1  : return binarysearch (list, searchnum, left, middle-1);
 }
 return -1;
}
```

Calling a Recursive Algorithm

```
int factoral(int n)
{
   if (n!=0)
      return(n*factoral(n-1));
   else
      return(1);
}
```

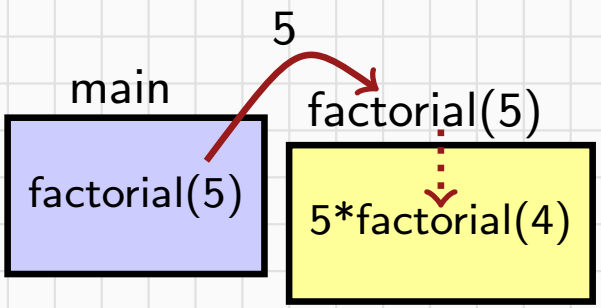
```
int main(void)
{
   int x;
   cout << "Input an integer:";
   cin >> x;
   cout << "x!="<<factorial(x);
   return(0);
}
```



Calling a Recursive Algorithm

```
int factoral(int n)
{
   if (n!=0)
      return(n*factoral(n-1));
   else
      return(1);
}
```

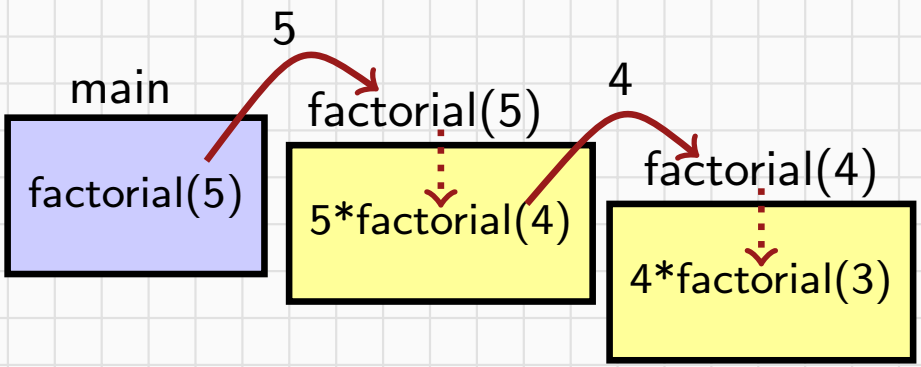
```
int main(void)
{
   int x;
   cout << "Input an integer:";
   cin >> x;
   cout << "x!="<<factorial(x);
   return(0);
}
```



Calling a Recursive Algorithm

```
int factoral(int n)
{
   if (n!=0)
      return(n*factoral(n-1));
   else
      return(1);
}
```

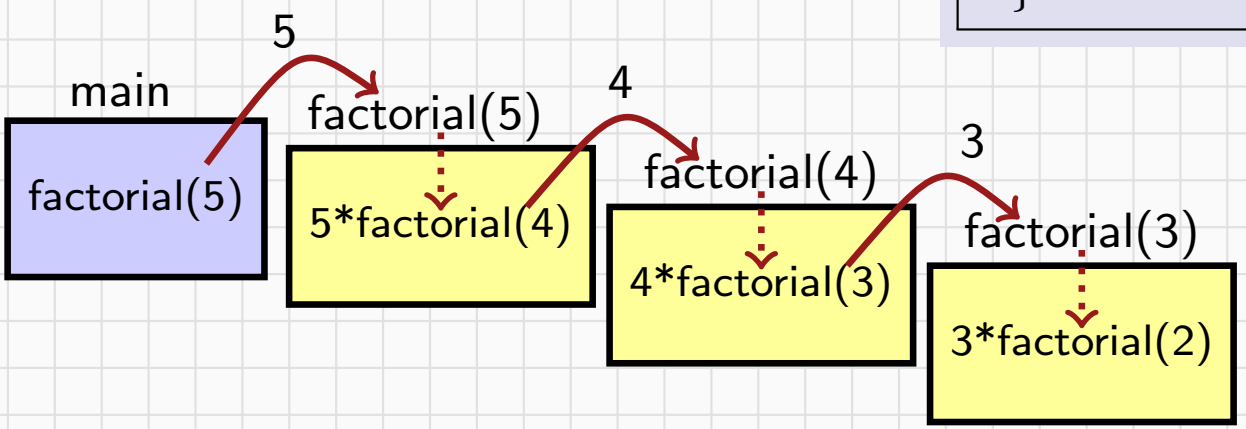
```
int main(void)
{
   int x;
   cout << "Input an integer:";
   cin >> x;
   cout << "x!="<<factorial(x);
   return(0);
}
```



Calling a Recursive Algorithm

```
int factoral(int n)
{
   if (n!=0)
      return(n*factoral(n-1));
   else
      return(1);
}
```

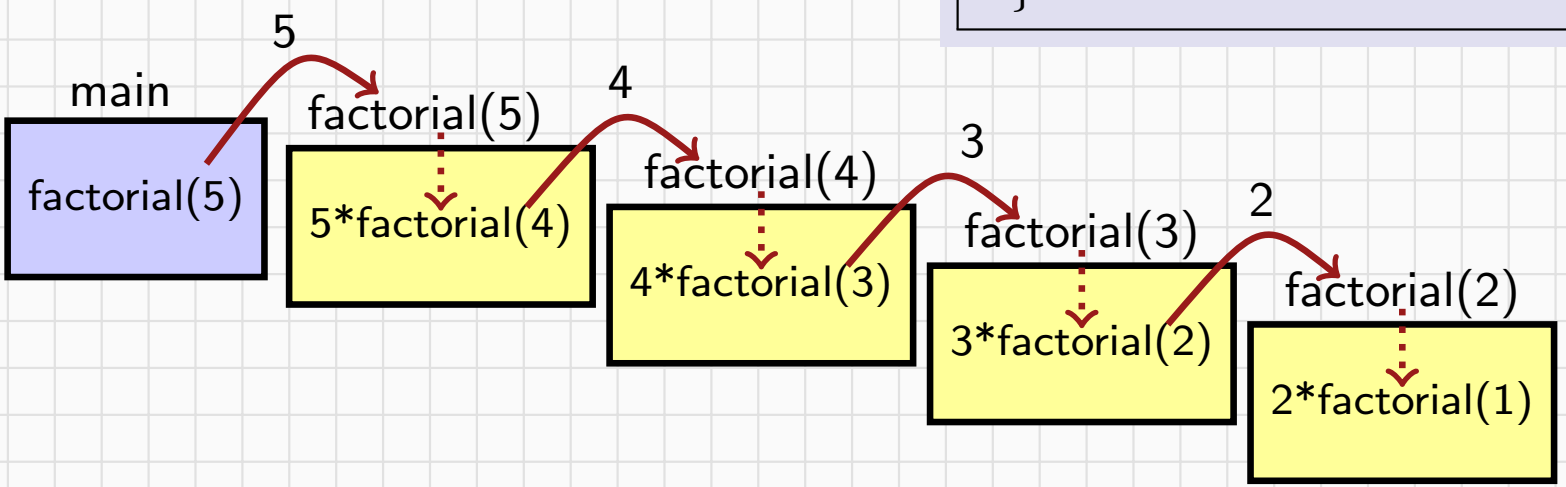
```
int main(void)
{
   int x;
   cout << "Input an integer:";
   cin >> x;
   cout << "x!="<<factorial(x);
   return(0);
}
```



Calling a Recursive Algorithm

```
int factoral(int n)
{
   if (n!=0)
      return(n*factoral(n-1));
   else
      return(1);
}
```

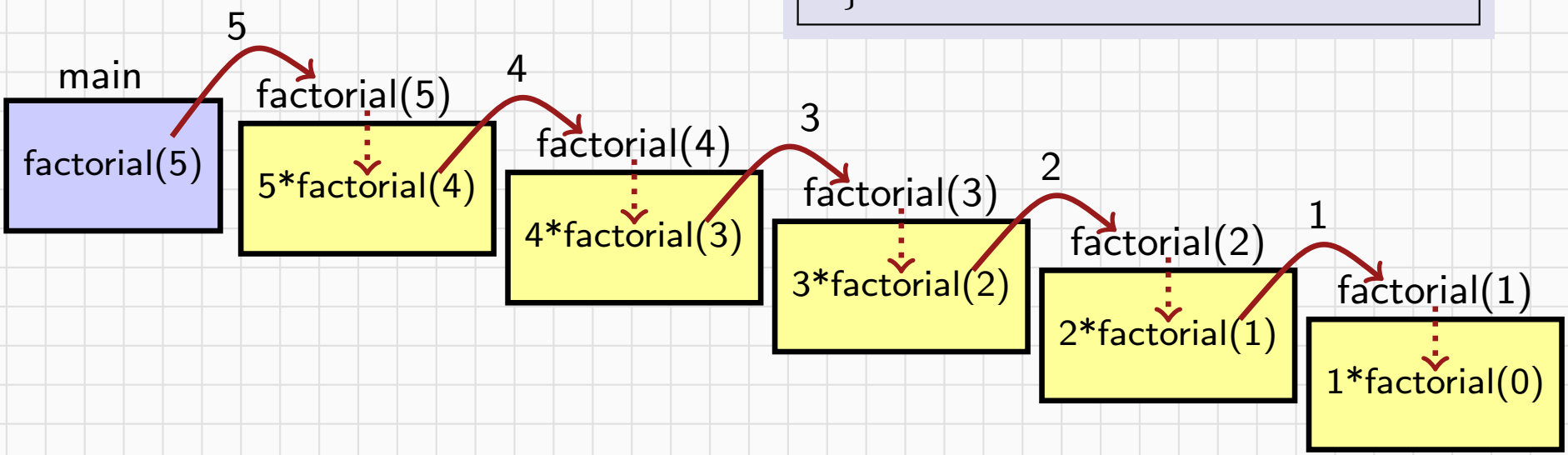
```
int main(void)
{
   int x;
   cout << "Input an integer:";
   cin >> x;
   cout << "x!="<<factorial(x);
   return(0);
}
```



Calling a Recursive Algorithm

```
int factoral(int n)
{
   if (n!=0)
      return(n*factoral(n-1));
   else
      return(1);
}
```

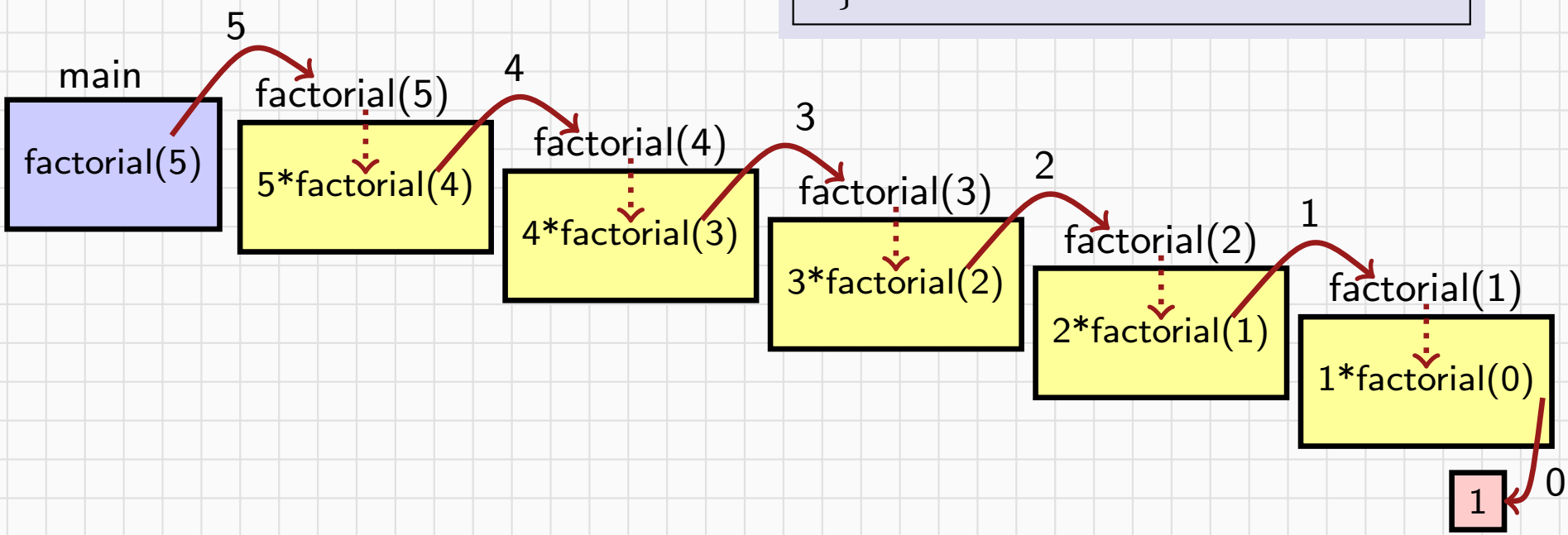
```
int main(void)
{
   int x;
   cout << "Input an integer:";
   cin >> x;
   cout << "x!="<<factorial(x);
   return(0);
}
```



Calling a Recursive Algorithm

```
int factoral(int n)
{
   if (n!=0)
      return(n*factoral(n-1));
   else
      return(1);
}
```

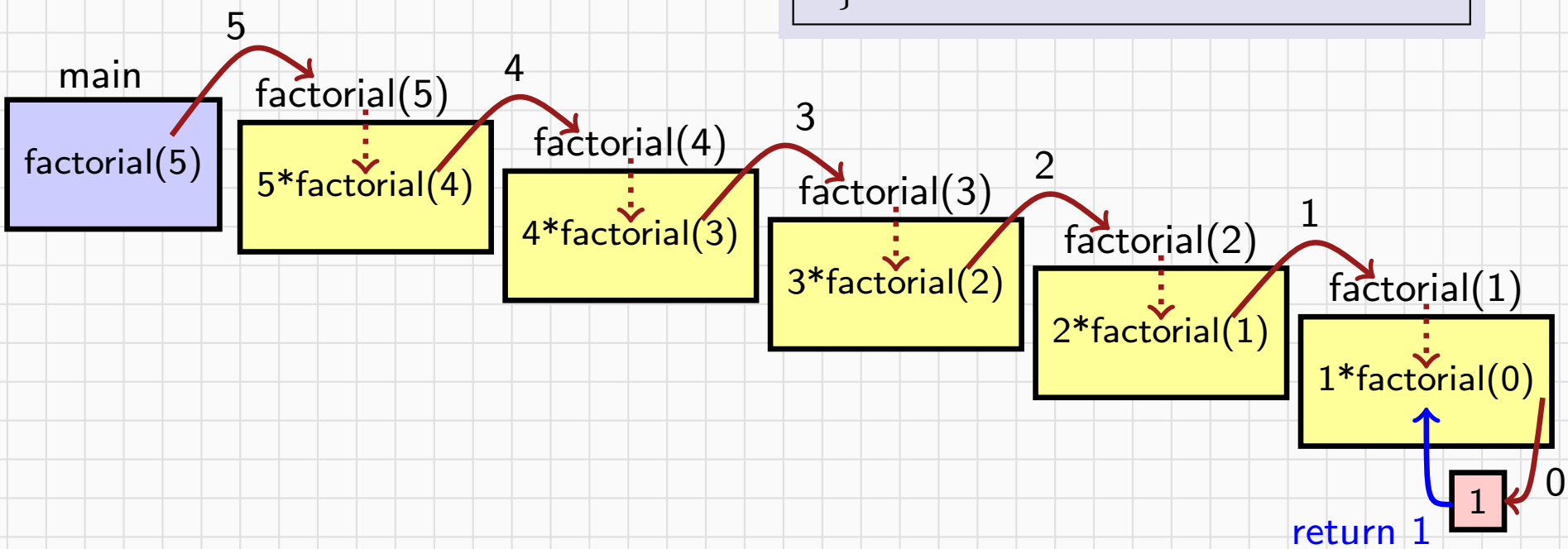
```
int main(void)
{
   int x;
   cout << "Input an integer:";
   cin >> x;
   cout << "x!="<<factorial(x);
   return(0);
}
```



Calling a Recursive Algorithm

```
int factoral(int n)
{
   if (n!=0)
      return(n*factoral(n-1));
   else
      return(1);
}
```

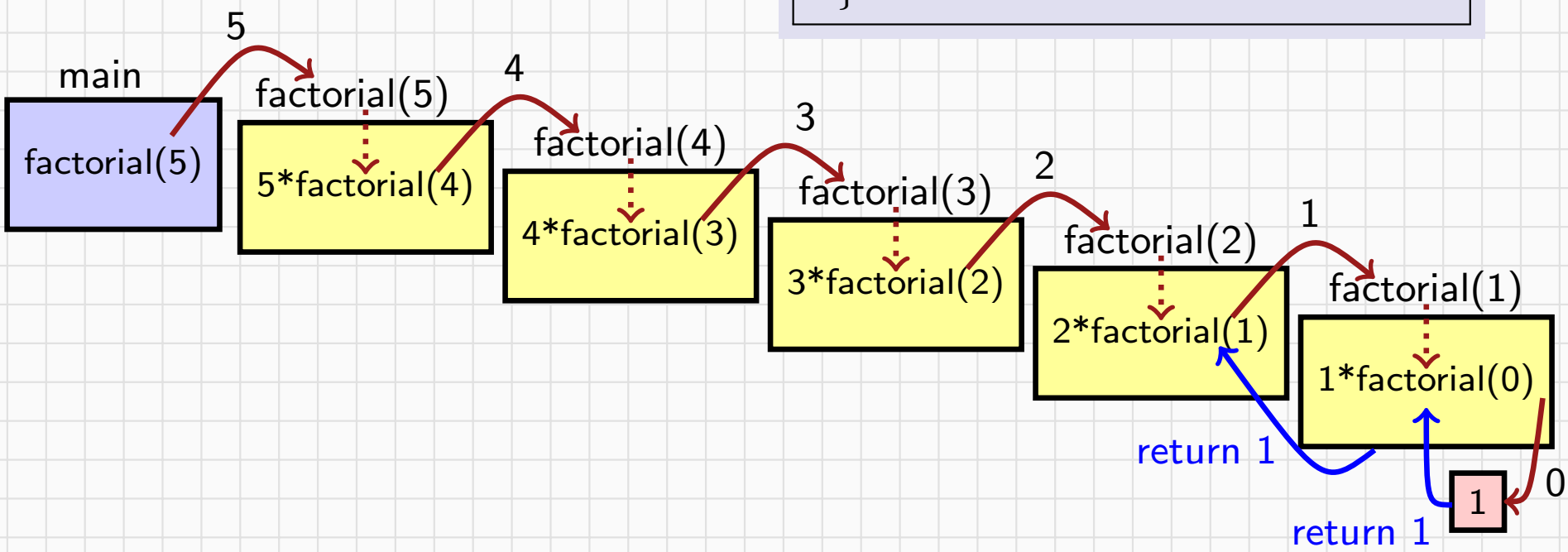
```
int main(void)
{
   int x;
   cout << "Input an integer:";
   cin >> x;
   cout << "x!="<<factorial(x);
   return(0);
}
```



Calling a Recursive Algorithm

```
int factoral(int n)
{
   if (n!=0)
      return(n*factoral(n-1));
   else
      return(1);
}
```

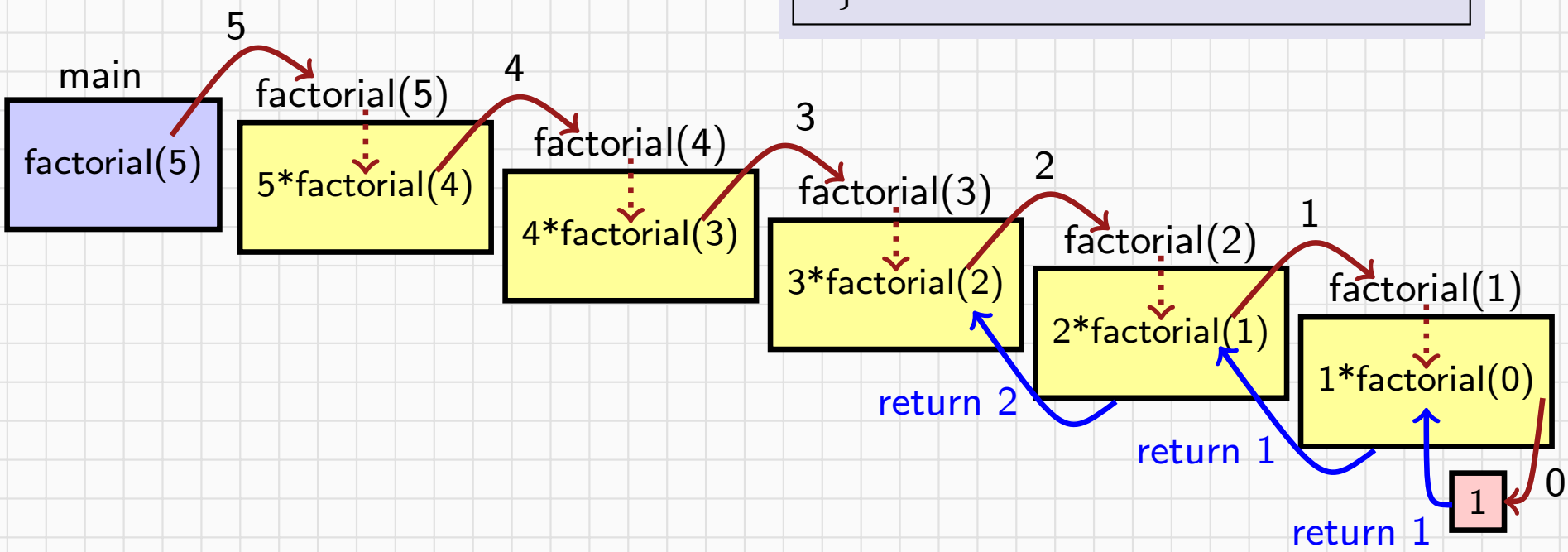
```
int main(void)
{
   int x;
   cout << "Input an integer:";
   cin >> x;
   cout << "x!="<<factorial(x);
   return(0);
}
```



Calling a Recursive Algorithm

```
int factoral(int n)
{
   if (n!=0)
      return(n*factoral(n-1));
   else
      return(1);
}
```

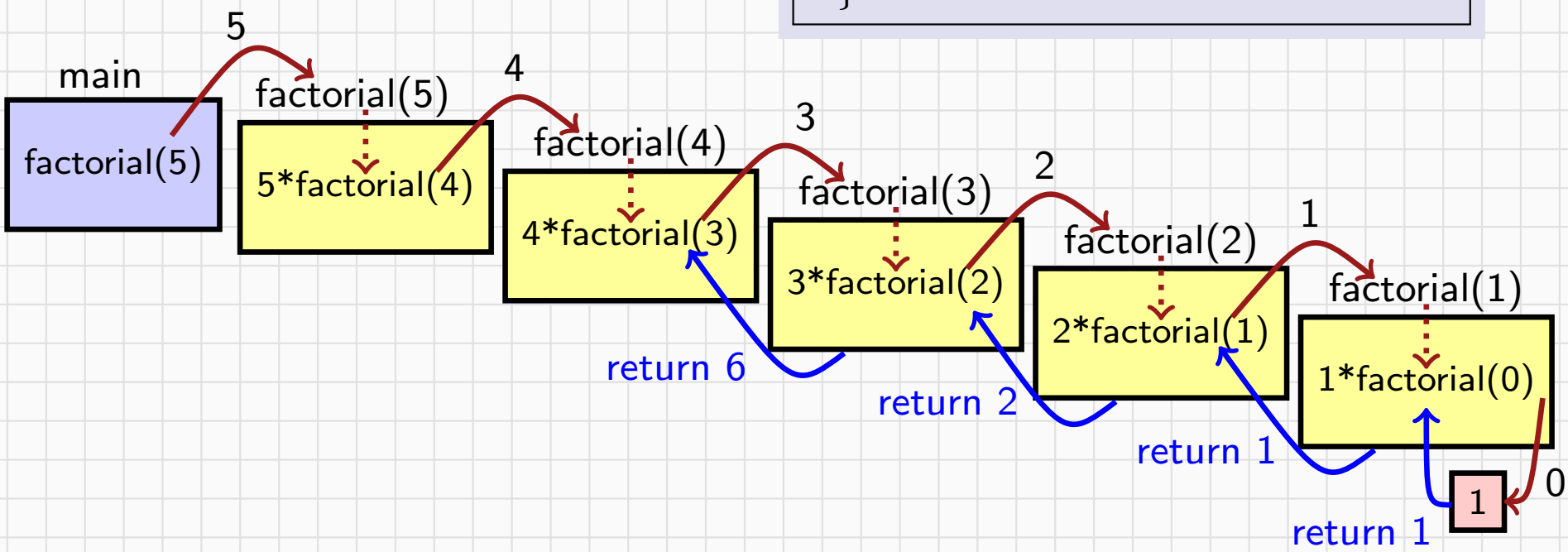
```
int main(void)
{
   int x;
   cout << "Input an integer:";
   cin >> x;
   cout << "x!="<<factorial(x);
   return(0);
}
```



Calling a Recursive Algorithm

```
int factoral(int n)
{
   if (n!=0)
      return(n*factoral(n-1));
   else
      return(1);
}
```

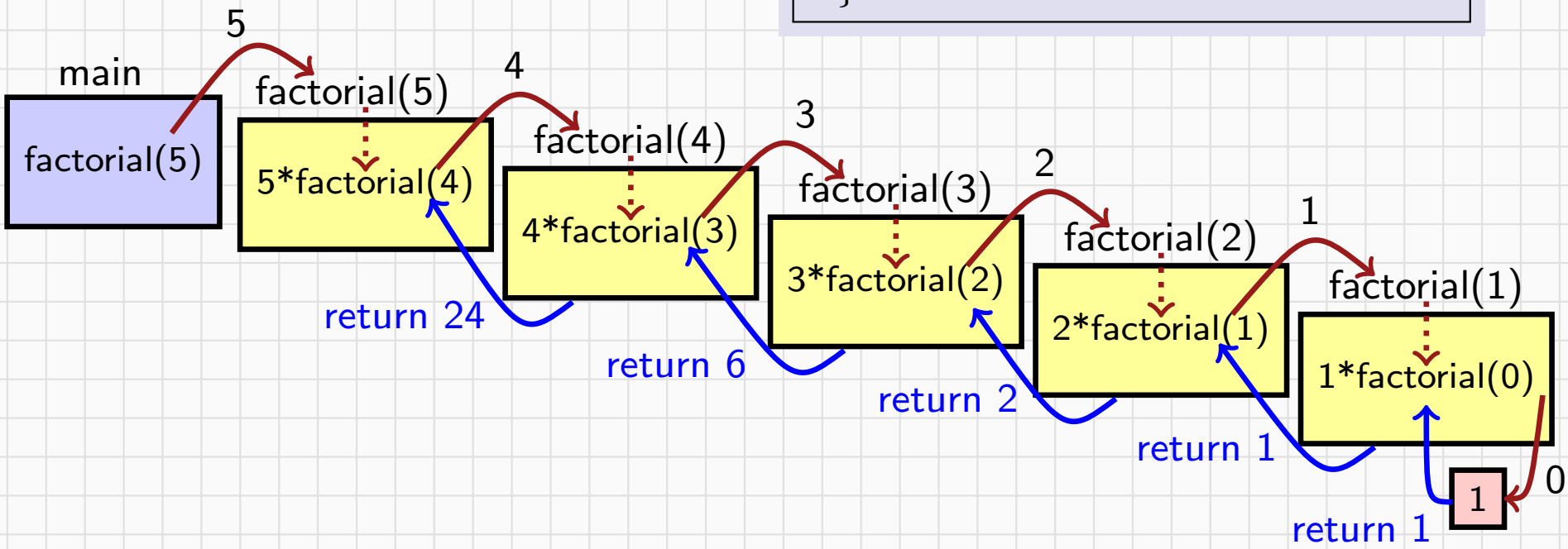
```
int main(void)
{
   int x;
   cout << "Input an integer:";
   cin >> x;
   cout << "x!="<<factorial(x);
   return(0);
}
```



Calling a Recursive Algorithm

```
int factoral(int n)
{
   if (n!=0)
      return(n*factoral(n-1));
   else
      return(1);
}
```

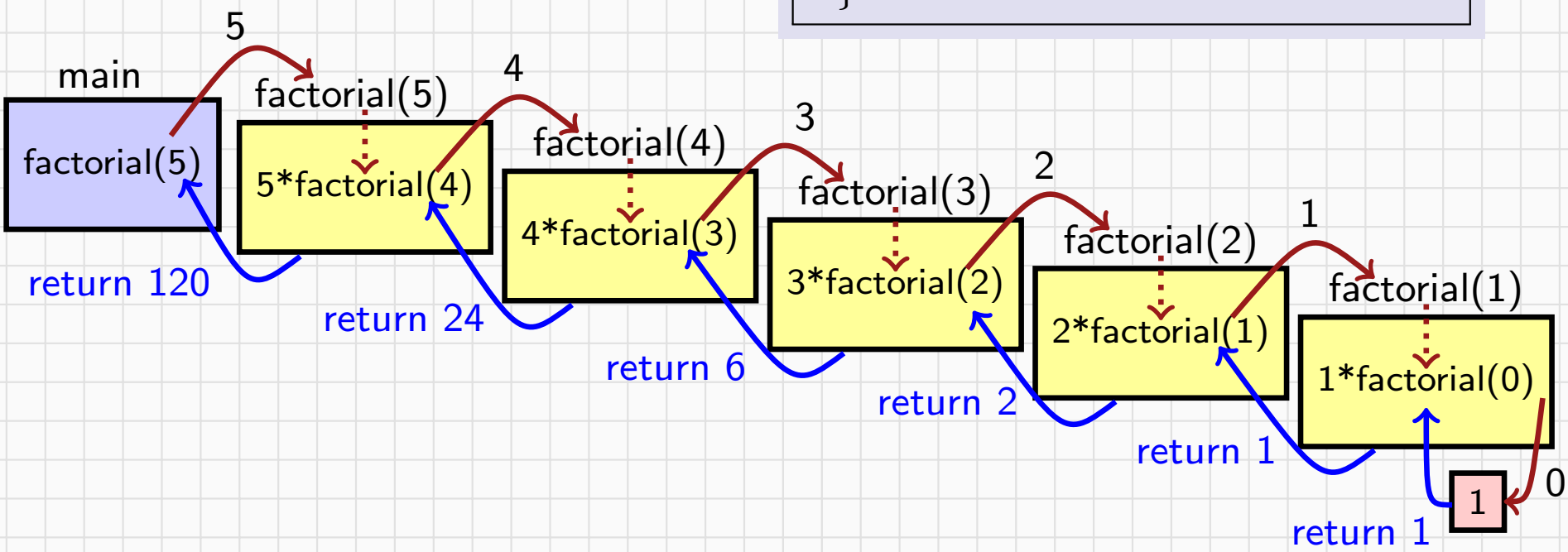
```
int main(void)
{
   int x;
   cout << "Input an integer:";
   cin >> x;
   cout << "x!="<<factorial(x);
   return(0);
}
```



Calling a Recursive Algorithm

```
int factoral(int n)
{
   if (n!=0)
      return(n*factoral(n-1));
   else
      return(1);
}
```

```
int main(void)
{
   int x;
   cout << "Input an integer:";
   cin >> x;
   cout << "x!="<<factorial(x);
   return(0);
}
```



Data Abstraction

Data Types

- A **data type** is a collection of **objects** and a set of **operations** that act on those objects.
- For example, the data type **`int`** consists of the objects $\{0, +1, -1, +2, -2, \ldots,$ INT_MIN, INT_MAX$\}$ and the operations $+, -, *, /$ and %.
- The data types of C
  - The basic data types: **`char`**, **`int`**, **`float`** and **`double`**
  - The group data types: **`array`** and **`struct`**
  - The pointer data type
  - The user-defined types

Abstract Data Type

- An abstract data type(ADT) is a data type that is organized in such a way that the specification of the **objects** and the **operations** on the objects is separated from the representation of the objects and the implementation of the operations.
- class(C++)

Natural_Number

***Structure 1.1:** Abstract data type *Natural_Number*

ADT *Natural_Number* is

objects: an ordered subrange of the integers starting at zero and ending at the maximum integer (*INT_MAX*) on the computer

functions:

for all x, y $\in$ *Nat_Number*; *TRUE*, *FALSE* $\in$ *Boolean*

and where +, -, <, and == are the usual integer operations.

Nat_No Zero ( )	::= 0
Boolean Is_Zero(x)	::= **if** (x) **return** *FALSE*
	else return *TRUE*
Nat_No Add(x, y)	::= **if** ((x+y) <= *INT_MAX*) **return** x+y
	else return *INT_MAX*
Boolean Equal(x,y)	::= **if** (x== y) **return** *TRUE*
	else return *FALSE*
Nat_No Successor(x)	::= **if** (x == *INT_MAX*) **return** x
	else return x+1
Nat_No Subtract(x,y)	::= **if** (x<y) **return** 0
	else return x-y

end *Natural_Number*

Specification vs. Implementation

- An ADT is implementation independent
- Operation specification
  - function name
  - the types of arguments
  - the type of the results
- The functions of a data type can be classify into several categories:
  - creator / constructor
  - transformers
  - observers / reporters

Performance Analysis

Measurements

- Criteria
  - Is it correct?
  - Is it efficient?
  - Is it readable?
- Performance Analysis (machine independent)
  - space complexity: storage requirement
  - time complexity: computing time
- Performance Measurement (machine dependent)

Definition

The **space complexity** of a program is the amount of memory that it needs to run to completion. The **time complexity** of a program is the amount of computer time that it needs to run to completion.

Space Complexity

- Fixed space requirements (c)
  Independent of the characteristics of the inputs and outputs
  - Instruction space
  - Space for simple variables, fixed-size structured variable, constants
- **Variable space requirements ($S_P(I)$)**
  Depend on the instance characteristic I
  - number, size, values of inputs and outputs associated with I
  - recursive stack space, formal parameters, local variables, return address

$$S(P) = c + S_P(I)$$

Example of abc

```
float abc (float a, float b, float c)
{
    return a + b + b * c
           + (a + b - c)/(a + b) + 4.00;
}
```

only fixed space requirement ➠ $S_{abc}(I) = 0$

Example of sum

```
float sum(float list[], int n)
{
    float tempsum = 0;
    int i;
    for( i = 0; i <n ; i++)
        tempsum += list[i];
    return tempsum;
}
```

$\Rightarrow$ $S_{sum}(I) = 0$

- C passes all parameters by value.
- When an array is passed as an argument to a function, C interprets it as passing the address of the first element of the array.

Example of rsum

```
float rsum(float list[], int n)
{
    if (n)
        return rsum(list,n-1)+list[n-1];
    return 0;
}
```

Space needed for one recursive call:

Type	Name	# of bytes
parameter: array pointer	**list** []	4
parameter: integer	**n**	4
return address:(used internally)		4
TOTAL per recursive call		12

☞ # of number is MAX ⇒ total space=12*MAX

Time Complexity

- The time, $T(P)$, taken by a program, P, is the sum of its compile time C and its run (or execution) time, $T_P(I)$

$$T(P) = C + T_P(I)$$

- **Fixed time requirements**
  Compile time (C), independent of instance characteristics
- **Variable time requirements**
  A simple program that adds and subtracts numbers

$$T_P(n) = c_a ADD(n) + c_s SUB(n) + c_l LDA(n) + c_{st} STA(n)$$

Machine-Independent Estimation

- A **program step** is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.
- Example: (Regard as the same unit machine independent)

  `a = 2;` or `a = x*y + b + c/d;`
- Methods to compute the step count
  - Introduce a global variable **count** into programs
  - Tabular method
    - Determine the step count for each statement: **s/e**
      Determine the number of times for each statement: **frequency**
    - Add up the contribution of all statements

Step Counter

```
float sum(float list[], int n)
{
    float tempsum = 0; count++;
    int i;
    for (i = 0; i < n; i++)
    {
        count++; /*for the loop */
        tempsum += list[i];
        count++;/* for the assignment
    }
    count++; /* last execution of for */
    count++; /* for return */
    return tempsum;
}
```

$2n + 3$ steps

Iterative Summing of a List of Numbers

Statement	s/e	Freqency	Subtotal
`float sum(float list[] , int n)`	0	0	0
{	0	0	0
`  float tempsum = 0;`	1	1	1
`  int i;`	0	0	0
`  for( i = 0; i < n; i++)`	1	$n+1$	$n+1$
`      tempsum += list[i];`	1	n	n
`  return tempsum;`	1	1	1
}	0	0	0
Total			$2n+3$

s/e: steps/exection

Recursive Summing of a List of Numbers

Statement	s/e	Frequency	Subtotal
`float rsum(float list[] , int n)`	0	0	0
{	0	0	0
`  if (n)`	1	$n+1$	$n+1$
`      return rsum(list, n-1) + list[n-1];`	1	n	n
`  return list[0];`	1	1	1
}	0	0	0
Total			$2n+2$

Matrix Addition

Statement	s/e	Frequency	Subtotal
`void add(int a[] [MAX_SIZE],...)`	0	0	0
{	0	0	0
`  int i, j;`	0	0	0
`  for (i = 0; i < rows; i++)`	1	$rows + 1$	$rows + 1$
`    for (j= 0; j < cols; j++)`	1	$rows(cols + 1)$	$rows \cdot cols + rows$
`      c[i][j] = a[i][j] +b[i][j];`	1	$rows \cdot cols$	$rows \cdot cols$
}	0	0	0
Total			$2rows \cdot cols$ $+2rows + 1$

Summary

- For a program,
  - The **best case** step count is the minimum number of steps that can be executed on input data of size n.
  - The **worst case** step count is the maximum number of steps that can be executed on input data of size n.
  - The **average case** step count is the average number of steps that can be executed on input data of size n.
- Motivation for step counts
  - Compare the time complexities of two programs.
  - Predict the growth in the rum time as the input I changes. Especially for large n.

Order

- We will show how algorithms can be grouped according to their eventual behavior (for large n).
- For sufficiently large of value, $c_1 n^2 + c_2 n$ is greater than $c_3 n$
- Break-even point:
  Any linear-time algorithm is eventually more efficient than any quadratic-time algorithm

n	$100n$	$0.01n^2 + 10n + 100$
10	1,000	201
20	2,000	304
50	5,000	625
100	10,000	1,200
1,000	100,000	20,100
10,000	1,000,000	1,100,100

Asymptotic Analysis

- To compare two algorithms with running times $f(n)$ and $g(n)$, we need a rough measure that characterizes how fast each function grows.

  ☞ Hint: use rate of growth

- Compare functions in the limit, that is, asymptotically! (i.e., for large values of n)

Θ-Notation

Definition

$\Theta(g(n)) = \{f(n)$: there exists positive constants $c_1, c_2,$ and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$

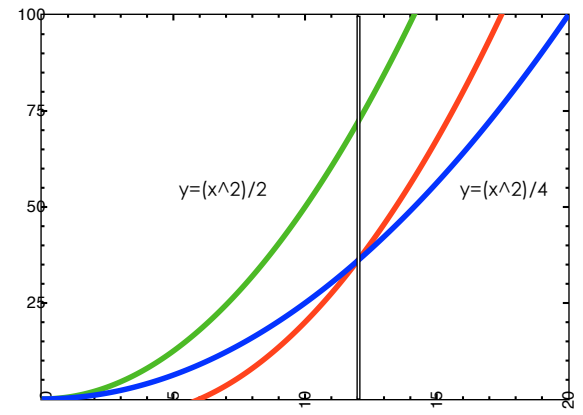
- $\Theta(g(n))$: a set of functions
- $f(n) = \Theta(g(n)) \rightarrow f(n)$ is a member of $\Theta(g(n))$ or $f(n) \in \Theta(g(n))$
- $g(n)$ is an **asymptotic tight bound** for $f(n$

  Ex: $\frac{1}{2}n^2 - 3n = \Theta(n^2)$

  Pf: Since $\frac{n^2}{4} \leq \frac{1}{2}n^2 - 3n \leq \frac{n^2}{2}$ if $n \geq 12$,

  choose $c_1 = \frac{1}{4}, c_2 = \frac{1}{2}, n_0 = 12$

  ☞ $6n^3 \neq \Theta(n^2)$



O-Notation

Definition

$$O(g(n)) = \{f(n) : \text{there exists positive constants } c, \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

- $O(g(n))$: a set of functions
- $f(n) = O(g(n)) \rightarrow f(n)$ is a member of $O(g(n))$ or $f(n) \in O(g(n))$
- $g(n)$ is an **asymptotic upper bound** for $f(n)$.
- Note that $f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$ (Θ-notation is a stronger notion than O-notation)
- Ex: $n^2 - 3n = O(n^2)$
  $10n + 5 = O(n^2)$
  $5 = O(1), 5 = O(n), 5 = O(n^2), \cdots$ (Which one is tight?)

Ω-Notation

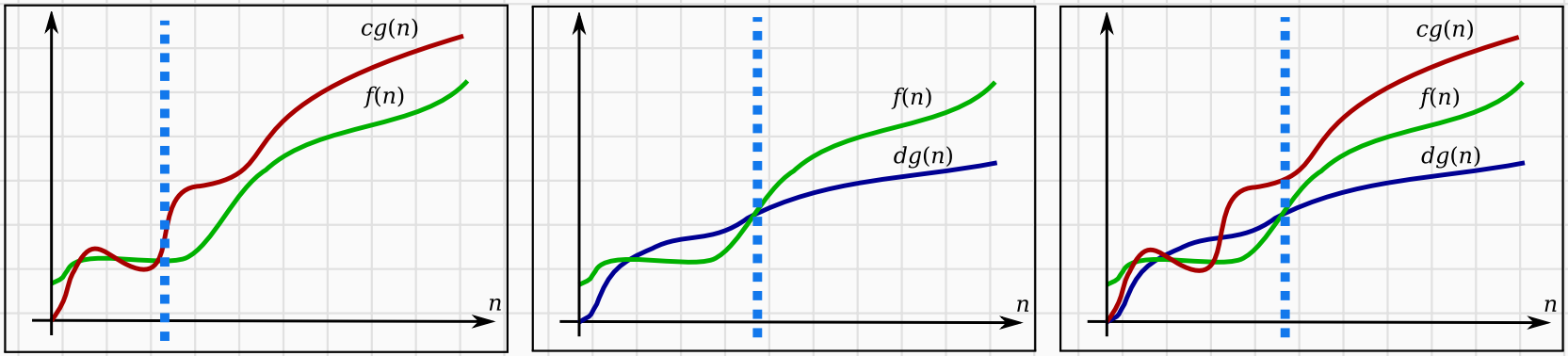
Definition

$$\Omega(g(n)) = \{f(n) : \text{there exists positive constants } c, \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

- $\Omega(g(n))$: a set of functions
- $f(n) = \Omega(g(n)) \rightarrow f(n)$ is a member of $\Omega(g(n))$ or $f(n) \in \Omega(g(n))$
- $g(n)$ is an **asymptotic lower bound** for $f(n)$.
- Ex: $5n^2 = \Omega(n)$
  $n = \Omega(\log n)$
  $n = \Omega(2n), n^3 = \Omega(n^2)$

Asymptotic Notation

- O-**notation**: asymptotic "less than":
  $f(n) = O(g(n))$ implies: $f(n)$ "$\leq$" $g(n)$
- Ω-**notation**: asymptotic "greater than":
  $f(n) = \Omega(g(n))$ implies: $f(n)$ "$\geq$" $g(n)$
- Θ-**notation**: asymptotic "equality":
  $f(n) = \Theta(g(n))$ implies: $f(n)$ "$=$" $g(n)$



Examples for $O, \Omega,$ and Θ

1. $f(n) = 3n + 2$
   - $3n + 2 \leq 4n$ for all $n \geq 2 \Rightarrow 3n + 2 = O(n)$
   - $3n + 2 \geq 3n$ for all $n \geq 1 \Rightarrow 3n + 2 = \Omega(n)$
   - $3n \leq 3n + 2 \leq 4n$ for all $n \geq 2 \Rightarrow 3n + 2 = \Theta(n)$

Examples for $O, \Omega,$ and Θ

1. $f(n) = 3n + 2$
   - $3n + 2 \leq 4n$ for all $n \geq 2 \Rightarrow 3n + 2 = O(n)$
   - $3n + 2 \geq 3n$ for all $n \geq 1 \Rightarrow 3n + 2 = \Omega(n)$
   - $3n \leq 3n + 2 \leq 4n$ for all $n \geq 2 \Rightarrow 3n + 2 = \Theta(n)$
2. $f(n) = 10n^2 + 4n + 2$
   - $10n^2 + 4n + 2 \leq 11n^2$ for all $n \geq 5$
     $\Rightarrow 10n^2 + 4n + 2 = O(n^2)$
   - $10n^2 + 4n + 2 \geq n^2$ for all $n \geq 1$
     $\Rightarrow 10n^2 + 4n + 2 = \Omega(n^2)$
   - $n^2 \leq 10n^2 + 4n + 2 \leq 11n^2$ for all $n \geq 5$
     $\Rightarrow 10n^2 + 4n + 2 = \Theta(n^2)$

Examples for $O, \Omega,$ and Θ

1. $f(n) = 3n + 2$
   - $3n + 2 \leq 4n$ for all $n \geq 2 \Rightarrow 3n + 2 = O(n)$
   - $3n + 2 \geq 3n$ for all $n \geq 1 \Rightarrow 3n + 2 = \Omega(n)$
   - $3n \leq 3n + 2 \leq 4n$ for all $n \geq 2 \Rightarrow 3n + 2 = \Theta(n)$
2. $f(n) = 10n^2 + 4n + 2$
   - $10n^2 + 4n + 2 \leq 11n^2$ for all $n \geq 5$
     $\Rightarrow 10n^2 + 4n + 2 = O(n^2)$
   - $10n^2 + 4n + 2 \geq n^2$ for all $n \geq 1$
     $\Rightarrow 10n^2 + 4n + 2 = \Omega(n^2)$
   - $n^2 \leq 10n^2 + 4n + 2 \leq 11n^2$ for all $n \geq 5$
     $\Rightarrow 10n^2 + 4n + 2 = \Theta(n^2)$
3. $100n + 6 = O(n)$ $\quad (100n + 6 \leq 101n$ for $n \geq 10)$

Examples for $O, \Omega,$ and Θ

1. $f(n) = 3n + 2$
   - $3n + 2 \leq 4n$ for all $n \geq 2 \Rightarrow 3n + 2 = O(n)$
   - $3n + 2 \geq 3n$ for all $n \geq 1 \Rightarrow 3n + 2 = \Omega(n)$
   - $3n \leq 3n + 2 \leq 4n$ for all $n \geq 2 \Rightarrow 3n + 2 = \Theta(n)$
2. $f(n) = 10n^2 + 4n + 2$
   - $10n^2 + 4n + 2 \leq 11n^2$ for all $n \geq 5$
     $\Rightarrow 10n^2 + 4n + 2 = O(n^2)$
   - $10n^2 + 4n + 2 \geq n^2$ for all $n \geq 1$
     $\Rightarrow 10n^2 + 4n + 2 = \Omega(n^2)$
   - $n^2 \leq 10n^2 + 4n + 2 \leq 11n^2$ for all $n \geq 5$
     $\Rightarrow 10n^2 + 4n + 2 = \Theta(n^2)$
3. $100n + 6 = O(n)$ $\quad (100n + 6 \leq 101n \text{ for } n \geq 10)$
4. $6 \cdot 2^n + n^2 = O(2^n)$ $\quad (6 \cdot 2^n \leq 6 \cdot 2^n + n^2 \leq 7 \cdot 2^n \text{ for } n \geq 4\ )$

Theorem 1.2

If $f(n) = a_m n^m + \cdots + a_1 n + a_0$, then $f(n) = O(n^m)$.

Theorem 1.2

If $f(n) = a_m n^m + \cdots + a_1 n + a_0$, then $f(n) = O(n^m)$.

Proof.

$$\begin{aligned} f(n) &\leq |a_m| n^m + |a_{m-1}| n^{m-1} \cdots + |a_1| n + |a_0| \\ &\leq \sum_{i=0}^{m} |a_i| n^i \\ &\leq n^m \sum_{i=0}^{m} |a_i| n^{i-m} \\ &\leq n^m \sum_{i=0}^{m} |a_i| \text{ if } n \geq 1 \end{aligned}$$

Theorem 1.2

If $f(n) = a_m n^m + \cdots + a_1 n + a_0$, then $f(n) = O(n^m)$.

Proof.

$$\begin{aligned} f(n) &\leq |a_m| n^m + |a_{m-1}| n^{m-1} \cdots + |a_1| n + |a_0| \\ &\leq \sum_{i=0}^{m} |a_i| n^i \\ &\leq n^m \sum_{i=0}^{m} |a_i| n^{i-m} \\ &\leq n^m \sum_{i=0}^{m} |a_i| \text{ if } n \geq 1 \end{aligned}$$

Theorem 1.3

If $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Omega(n^m)$.

Theorem 1.2

If $f(n) = a_m n^m + \cdots + a_1 n + a_0$, then $f(n) = O(n^m)$.

Proof.

$$\begin{aligned} f(n) &\leq |a_m| n^m + |a_{m-1}| n^{m-1} \cdots + |a_1| n + |a_0| \\ &\leq \sum_{i=0}^{m} |a_i| n^i \\ &\leq n^m \sum_{i=0}^{m} |a_i| n^{i-m} \\ &\leq n^m \sum_{i=0}^{m} |a_i| \text{ if } n \geq 1 \end{aligned}$$

Theorem 1.3

If $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Omega(n^m)$.

Theorem 1.4

If $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Theta(n^m)$.

Theorem 1.2

If $f(n) = a_m n^m + \cdots + a_1 n + a_0$, then $f(n) = O(n^m)$.

Proof.

$$\begin{aligned} f(n) &\leq |a_m| n^m + |a_{m-1}| n^{m-1} \cdots + |a_1| n + |a_0| \\ &\leq \sum_{i=0}^{m} |a_i| n^i \\ &\leq n^m \sum_{i=0}^{m} |a_i| n^{i-m} \\ &\leq n^m \sum_{i=0}^{m} |a_i| \text{ if } n \geq 1 \end{aligned}$$

Theorem 1.3

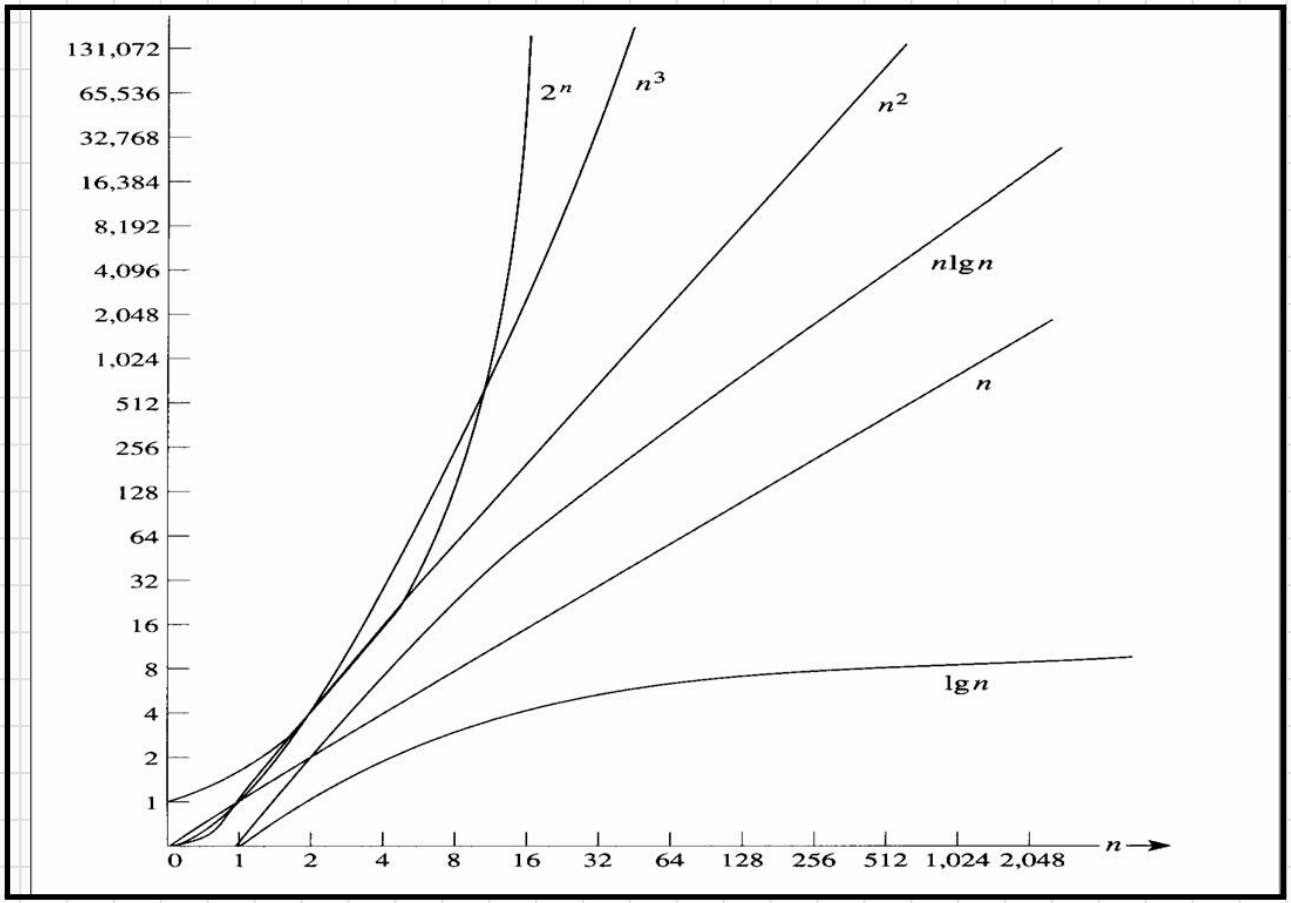
If $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Omega(n^m)$.

Theorem 1.4

If $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Theta(n^m)$.

Simple Rule: Drop **lower order** terms and **constant factors**.

Growth Rates of Complexity Function



$O(n)$: linear $O(n^2)$: quadratic $O(n^3)$: cubic $O(2^n)$: exponential

Performance Measurement

Comparison of Execution Time

The time needed by a 1 **billion instructions per second** computer to execute a program of complexity $f(n)$ instructions

	$f(n)$						
n	n	$n \log_2 n$	n^2	n^3	n^4	n^{10}	2^n
10	0.01μs	0.03μs	0.1μs	1μs	10μs	10s	1μs
20	0.02μs	0.09μs	0.4μs	8μs	160μs	2.84h	1ms
30	0.03μs	0.15μs	0.9μs	27μs	810μs	6.83d	1s
40	0.04μs	0.21μs	1.6μs	64μs	2.56ms	121d	18m
50	0.05μs	0.28μs	2.5μs	125μs	6.25ms	3.1y	13d
100	0.10μs	0.66μs	10μs	1ms	100ms	3171y	$4 \cdot 10^{13}$y
10^3	1μs	9.96μs	1ms	1s	16.67m	$3.17 \cdot 10^{13}$y	$32 * 10^{283}$y
10^4	10μs	130μs	100ms	16.67m	115.7d	$3.17 \cdot 10^{23}$y	
10^5	100μs	1.66ms	10s	11.57d	3171y	$3.17 \cdot 10^{33}$y	
10^6	1ms	19.92ms	16.67m	31.71y	$3.17 \cdot 10^7$y	$3.17 \cdot 10^{43}$y	

Event Timing in C

At some point we must consider how the algorithm executes on our machine.

	Method 1	Method 2
Start timing	start = clock();	start = time(NULL);
Stop timing	stop = clock();	stop = time(NULL);
Type returned	clock_t	time_t
Result in seconds	duration = ((double) (stop-start))/CLOCKS_PER_SEC;	duration = (double) difftime(stop,start);

```
#include <time.h>
```

Timing Program for Selection Sort

```
#include <stdio.h>
#include <time.h>
#include "selectionSort.h"
#define MAX_SIZE 1001
void main(void)
{
    int i, n, step = 10;
    int a[MAX_SIZE];
    double duration;
    clock_t start;
    printf(" n time\n");
    for (n = 0; n <= 1000; n += step)
    {
        for (i = 0; i < n; i++)
            a[i] = n - i;
        start = clock();
        sort(a, n);
        duration = ((double) (clock() - start))/CLOCKS_PER_SEC;
        printf("%6d %f\n", n, duration);
        if (n == 100) step = 100;
    }
}
```

Timing Program for Selection Sort

```
#include <stdio.h>
#include <time.h>
#include "selectionSort.h"
#define MAX_SIZE 1001
void main(void)
{
    int i, n, step = 10;
    int a[MAX_SIZE];
    double duration;
    clock_t start;
    printf(" n time\n");
    for (n = 0; n <= 1000; n += step)
    {
        for (i = 0; i < n; i++)
            a[i] = n - i;
        start = clock();
        sort(a, n);
        duration = ((double) (clock() - start))/CLOCKS_PER_SEC;
        printf("%6d %f\n", n, duration);
        if (n == 100) step = 100;
    }
}
```

duration $\Rightarrow$ 0!

```
#include <stdio.h>
#include <time.h>
#include "selectionSort.h"
#define MAX_SIZE 1001
void main(void)
{  int i, n, step = 10;
   int a[MAX_SIZE];
   double duration;
   clock_t start;
   printf(" n repetition time\n");
   for (n = 0; n <= 1000; n += step)
   {  long repetitions = 0;
      clock_t start = clock();
      do
      {  repetitions++;
         for (i=0; i<n ; i++)
            a[i] = n - i;
         sort(a, n);
      }while(clock() - start < 1000)
      duration = ((double) (clock() - start))/CLOCKS_PER_SEC;
      duration /= repetitions;
      printf("%6d %9d %f\n", n, repetitions, duration);
      if (n == 100) step = 100;
   }
}
```

More accurate???