

# Proofs of Correctness Complexity, Cost and Efficiency

## Proofs of Correctness

the `assert` macro  
enforcing preconditions and postconditions  
reasoning about loops

## Efficiency of Algorithms

complexity, cost, and efficiency  
counting the number of operations  
the big-O notation

### 1 Proofs of Correctness

the `assert` macro

enforcing preconditions and postconditions

reasoning about loops

### 2 Efficiency of Algorithms

complexity, cost, and efficiency

counting the number of operations

the big-O notation

MCS 360 Lecture 7  
Introduction to Data Structures  
Jan Vershelde, 8 September 2010

# Proofs of Correctness

## Complexity, Cost and Efficiency

### Proofs of Correctness

#### the `assert` macro

enforcing preconditions and postconditions  
reasoning about loops

### Efficiency of Algorithms

complexity, cost, and efficiency  
counting the number of operations  
the big-O notation

## 1 Proofs of Correctness

### the `assert` macro

enforcing preconditions and postconditions  
reasoning about loops

## 2 Efficiency of Algorithms

complexity, cost, and efficiency  
counting the number of operations  
the big-O notation

# Ensuring Correctness

## Proofs of Correctness

### the `assert` macro

enforcing preconditions and postconditions  
reasoning about loops

## Efficiency of Algorithms

complexity, cost, and efficiency  
counting the number of operations  
the big-O notation

We want to **prevent** errors by catching errors at compile time, not at run time.

For documentation purposes we specify

- preconditions: conditions on input parameters,
- postconditions: properties expected on return.

With `assert`, we **enforce** the conditions.

Recall the opening of a file: if the string cannot be associated with a file, then we want an exception thrown.

# Opening a File

## Proofs of Correctness

### the `assert` macro

enforcing preconditions and postconditions  
reasoning about loops

## Efficiency of Algorithms

complexity, cost, and efficiency

counting the number of operations

the big-O notation

```
#include <iostream>
#include <fstream>
#include <string>
#include <cassert>
using namespace std;

int main()
{
    string input_file_name;

    cout << "Give name of input file : ";
    cin >> input_file_name;

    ifstream ins(input_file_name.c_str());

    assert(ins);
```

Proofs of  
Correctnessthe `assert` macro

enforcing  
preconditions and  
postconditions  
reasoning about  
loops

Efficiency of  
Algorithms

complexity, cost, and  
efficiency  
counting the number  
of operations  
the big-O notation

There is no such file hello ...

```
$ open_file
```

```
Give name of input file : hello
```

```
Assertion failed: (ins), function main, \  
file open_file.cpp, line 32.
```

```
Abort trap
```

```
$
```

Toggle off the effect of the `assert`:

```
$ g++ -DNDEBUG -o open_file open_file.cpp
```

We may want to ignore assertions, for efficiency,  
or when the code has an interface that checks.

Proofs of  
Correctnessthe `assert` macro

enforcing  
preconditions and  
postconditions  
reasoning about  
loops

Efficiency of  
Algorithms

complexity, cost, and  
efficiency  
counting the number  
of operations  
the big-O notation

There is no such file hello ...

```
$ open_file
```

```
Give name of input file : hello
```

```
Assertion failed: (ins), function main, \  
file open_file.cpp, line 32.
```

```
Abort trap
```

```
$
```

Toggling off the effect of the `assert`:

```
$ g++ -DNDEBUG -o open_file open_file.cpp
```

We may want to ignore assertions, for efficiency,  
or when the code has an interface that checks.

# Proofs of Correctness

## Complexity, Cost and Efficiency

### Proofs of Correctness

the `assert` macro

enforcing preconditions and postconditions

reasoning about loops

### Efficiency of Algorithms

complexity, cost, and efficiency

counting the number of operations

the big-O notation

## 1 Proofs of Correctness

the `assert` macro

enforcing preconditions and postconditions

reasoning about loops

## 2 Efficiency of Algorithms

complexity, cost, and efficiency

counting the number of operations

the big-O notation

# Searching for a Character

## Proofs of Correctness

the `assert` macro  
 enforcing preconditions and postconditions  
 reasoning about loops

## Efficiency of Algorithms

complexity, cost, and efficiency  
 counting the number of operations  
 the big-O notation

Consider the following tasks:

- 1 Generate at random a string of 10 letters.  
Every letter is in the range from 'a' to 'z'.
- 2 Prompt the user for a character.  
Search for the given character in the string.

```
$ guess_character
generated "abiuemkary"
Make a guess : m
m is at position 5 in string "abiuemkary"
$
```

# Searching for a Character

## Proofs of Correctness

the `assert` macro  
enforcing preconditions and postconditions  
reasoning about loops

## Efficiency of Algorithms

complexity, cost, and efficiency  
counting the number of operations  
the big-O notation

Consider the following tasks:

- 1 Generate at random a string of 10 letters.  
Every letter is in the range from 'a' to 'z'.
- 2 Prompt the user for a character.  
Search for the given character in the string.

```
$ guess_character  
generated "abiuemkary"  
Make a guess : m  
m is at position 5 in string "abiuemkary"  
$
```

## the main program

```
int main()  
{  
    const int L = 10;  
    string s = random_string(L);  
    cout << "generated \"" << s << "\" << endl;  
  
    char guess;  
    cout << "Make a guess : ";  
    cin >> guess; cout << guess;  
  
    int index = search(s,guess);  
    if(index == -1)  
        cout << " does not occur";  
    else  
        cout << " is at position " << index;  
    cout << " in string \"" << s << "\" << endl;  
  
    return 0;  
}
```

## a useful function

Proofs of  
Correctnessthe `assert` macroenforcing  
preconditions and  
postconditionsreasoning about  
loopsEfficiency of  
Algorithmscomplexity, cost, and  
efficiencycounting the number  
of operations

the big-O notation

```
char random_char();
```

```
/*
```

```
Generates a lower case alphanumeric character,  
uniformly distributed at random.
```

```
Postcondition:
```

```
96 < int(random_char()) < 123. */
```

Lower case letters are encoded with consecutive numbers  
in the ASCII table and

$\text{int}('a') = 97$  and  $\text{int}('z') = 122$ .

## a useful function

Proofs of  
Correctness

the `assert` macro  
enforcing  
preconditions and  
postconditions  
reasoning about  
loops

Efficiency of  
Algorithms

complexity, cost, and  
efficiency  
counting the number  
of operations  
the big-O notation

```
char random_char();
```

```
/*
```

```
Generates a lower case alphanumeric character,  
uniformly distributed at random.
```

```
Postcondition:
```

```
96 < int(random_char()) < 123. */
```

Lower case letters are encoded with consecutive numbers  
in the ASCII table and

$\text{int}('a') = 97$  and  $\text{int}('z') = 122$ .

## asserting a postcondition

Proofs of  
Correctness

the `assert` macro  
enforcing  
preconditions and  
postconditions  
reasoning about  
loops

Efficiency of  
Algorithms

complexity, cost, and  
efficiency  
counting the number  
of operations  
the big-O notation

We compute with characters:

```
char random_char()  
{  
    const int m = int('z') - int('a') + 1;  
    const int r = int('a') + rand() % m;  
    const char c = char(r);  
  
    assert(int(c) > 96 && int(c) < 123);  
  
    return c;  
}
```

The postcondition does not guard against a too small  $m$  ...

## asserting a postcondition

Proofs of  
Correctness

the `assert` macro  
enforcing  
preconditions and  
postconditions  
reasoning about  
loops

Efficiency of  
Algorithms

complexity, cost, and  
efficiency  
counting the number  
of operations  
the big-O notation

We compute with characters:

```
char random_char()  
{  
    const int m = int('z') - int('a') + 1;  
    const int r = int('a') + rand() % m;  
    const char c = char(r);  
  
    assert(int(c) > 96 && int(c) < 123);  
  
    return c;  
}
```

The postcondition does not guard against a too small  $m$  ...

## a random string

Proofs of  
Correctness

the `assert` macro  
enforcing  
preconditions and  
postconditions  
reasoning about  
loops

Efficiency of  
Algorithms

complexity, cost, and  
efficiency  
counting the number  
of operations  
the big-O notation

```
string random_string ( int n );  
/*  
    Returns a random string of n  
    lower case alphanumeric characters.  
  
    Precondition: n >= 0.  
    Postcondition: if s = random_string(n),  
                   then s.length() == n and for all k  
                   in 0..n-1: 96 < int(s[k]) < 123. */
```

The assertions appear in loops.

## a random string

Proofs of  
Correctness

the `assert` macro  
enforcing  
preconditions and  
postconditions  
reasoning about  
loops

Efficiency of  
Algorithms

complexity, cost, and  
efficiency  
counting the number  
of operations  
the big-O notation

```
string random_string ( int n );  
/*  
    Returns a random string of n  
    lower case alphanumeric characters.  
  
    Precondition: n >= 0.  
    Postcondition: if s = random_string(n),  
                   then s.length() == n and for all k  
                   in 0..n-1: 96 < int(s[k]) < 123. */
```

The assertions appear in loops.

## extra separate loops

Proofs of  
Correctness

the `assert` macro  
enforcing  
preconditions and  
postconditions  
reasoning about  
loops

Efficiency of  
Algorithms

complexity, cost, and  
efficiency  
counting the number  
of operations  
the big-O notation

```
string random_string ( int n )
{
    string result = "";

    assert(n>=0);

    srand(time(0));
    for(int i=0; i<n; i++)
        result = result + random_char();

    assert(result.length() == n);
    for(int i=0; i<n; i++)
        assert(int(result[i]) > 96
            && int(result[i]) < 123);

    return result;
}
```

## extra separate loops

Proofs of  
Correctness

the `assert` macro  
enforcing  
preconditions and  
postconditions  
reasoning about  
loops

Efficiency of  
Algorithms

complexity, cost, and  
efficiency  
counting the number  
of operations  
the big-O notation

```
string random_string ( int n )
{
    string result = "";

    assert(n>=0);

    srand(time(0));
    for(int i=0; i<n; i++)
        result = result + random_char();

    assert(result.length() == n);
    for(int i=0; i<n; i++)
        assert(int(result[i]) > 96
            && int(result[i]) < 123);

    return result;
}
```

## specifying the search

Proofs of  
Correctness

the `assert` macro  
enforcing  
preconditions and  
postconditions  
reasoning about  
loops

Efficiency of  
Algorithms

complexity, cost, and  
efficiency  
counting the number  
of operations  
the big-O notation

After prompting the user for a character,  
we call `search`:

```
int search ( string s, char c );
/*
```

Searches the string `s` for the character `c`.

Postcondition: if `r == search(s,c)`, then  
either `r == -1` if

for all `k` in `0..s.length()-1`: `s[k] != c`,  
or `0 <= r < s.length()` and `s[r] == c`. \*/

## code for search

```
int search ( string s, char c )
{
    int result = -1;
    for(int i=0; i<s.length(); i++)
        if(s[i] == c)
            {
                result = i; break;
            }

    assert(result >= -1 ||
           result < s.length());

    if(result > -1)
        assert(s[result] == c);
    else
        for(int i=0; i<s.length(); i++)
            assert(s[i] != c);

    return result;
}
```

Proofs of  
Correctnessthe `assert` macroenforcing  
preconditions and  
postconditionsreasoning about  
loopsEfficiency of  
Algorithmscomplexity, cost, and  
efficiencycounting the number  
of operations

the big-O notation

## code for search

```
int search ( string s, char c )
{
    int result = -1;
    for(int i=0; i<s.length(); i++)
        if(s[i] == c)
            {
                result = i; break;
            }

    assert(result >= -1 ||
           result < s.length());

    if(result > -1)
        assert(s[result] == c);
    else
        for(int i=0; i<s.length(); i++)
            assert(s[i] != c);

    return result;
}
```

Proofs of  
Correctnessthe `assert` macroenforcing  
preconditions and  
postconditionsreasoning about  
loopsEfficiency of  
Algorithmscomplexity, cost, and  
efficiencycounting the number  
of operations

the big-O notation

## code for search

```
int search ( string s, char c )
{
    int result = -1;
    for(int i=0; i<s.length(); i++)
        if(s[i] == c)
            {
                result = i; break;
            }

    assert(result >= -1 ||
           result < s.length());

    if(result > -1)
        assert(s[result] == c);
    else
        for(int i=0; i<s.length(); i++)
            assert(s[i] != c);

    return result;
}
```

Proofs of  
Correctnessthe `assert` macroenforcing  
preconditions and  
postconditionsreasoning about  
loopsEfficiency of  
Algorithmscomplexity, cost, and  
efficiencycounting the number  
of operations

the big-O notation

# Proofs of Correctness

## Complexity, Cost and Efficiency

### Proofs of Correctness

the `assert` macro  
enforcing preconditions and postconditions  
reasoning about loops

### Efficiency of Algorithms

complexity, cost, and efficiency  
counting the number of operations  
the big-O notation

## 1 Proofs of Correctness

the `assert` macro  
enforcing preconditions and postconditions  
reasoning about loops

## 2 Efficiency of Algorithms

complexity, cost, and efficiency  
counting the number of operations  
the big-O notation

# Reasoning about Loops

## Proofs of Correctness

the `assert` macro  
enforcing preconditions and postconditions  
reasoning about loops

## Efficiency of Algorithms

complexity, cost, and efficiency  
counting the number of operations  
the big-O notation

A **loop invariant** is a relationship between variables true

- 1 at initialization, before the loop starts,
- 2 during the execution of the loop,
- 3 at the end, when the stop condition is true.

For example:

```
int result = -1;
for(int i=0; i<s.length(); i++)
    if(s[i] == c)
    {
        result = i; break;
    }
```

Loop invariant: for all  $k$ ,  $0 \leq k < i$ :  $s[k] \neq c$ .

# Reasoning about Loops

## Proofs of Correctness

the `assert` macro  
enforcing preconditions and postconditions  
reasoning about loops

## Efficiency of Algorithms

complexity, cost, and efficiency  
counting the number of operations  
the big-O notation

A **loop invariant** is a relationship between variables true

- 1 at initialization, before the loop starts,
- 2 during the execution of the loop,
- 3 at the end, when the stop condition is true.

For example:

```
int result = -1;
for(int i=0; i<s.length(); i++)
    if(s[i] == c)
    {
        result = i; break;
    }
```

Loop invariant: for all  $k$ ,  $0 \leq k < i$ :  $s[k] \neq c$ .

## using loop invariants

Proofs of  
Correctness

the `assert` macro  
enforcing  
preconditions and  
postconditions  
reasoning about  
loops

Efficiency of  
Algorithms

complexity, cost, and  
efficiency  
counting the number  
of operations  
the big-O notation

Given condition: for all  $k$ ,  $0 \leq k < i$ :  $s[k] \neq c$ ,  
verify that this is indeed a **loop invariant**.

Initially:  $\text{result} = -1$  and  $i = 0$ .

Loop stops if: not ( $i < s.\text{length}()$  and  $s[i] \neq c$ ):

- 1  $i == s.\text{length}()$ : test ( $s[i] == c$ ) was never true  
 $\Rightarrow \text{result} == -1$   
 $\Rightarrow$  for all  $k$ ,  $0 \leq k < s.\text{length}()$ :  $s[k] \neq c$
- 2  $s[i] == c$ : execute statements after `if( $s[i] == c$ )`,  
 $\Rightarrow \text{result} == i$   
 $\Rightarrow$  for all  $k$ ,  $0 \leq k < \text{result}$ :  $s[k] \neq c$

Showing that for all  $k$ ,  $0 \leq k < i$ :  $s[k] \neq c$  is a loop invariant  
leads to a proof of correctness.

## using loop invariants

Proofs of  
Correctness

the `assert` macro  
enforcing  
preconditions and  
postconditions  
reasoning about  
loops

Efficiency of  
Algorithms

complexity, cost, and  
efficiency  
counting the number  
of operations  
the big-O notation

Given condition: for all  $k$ ,  $0 \leq k < i$ :  $s[k] \neq c$ ,  
verify that this is indeed a **loop invariant**.

Initially:  $\text{result} = -1$  and  $i = 0$ .

Loop stops if: not ( $i < s.\text{length}()$  and  $s[i] \neq c$ ):

- 1  $i == s.\text{length}()$ : test ( $s[i] == c$ ) was never true  
 $\Rightarrow \text{result} == -1$   
 $\Rightarrow$  for all  $k$ ,  $0 \leq k < s.\text{length}()$ :  $s[k] \neq c$
- 2  $s[i] == c$ : execute statements after `if( $s[i] == c$ )`,  
 $\Rightarrow \text{result} == i$   
 $\Rightarrow$  for all  $k$ ,  $0 \leq k < \text{result}$ :  $s[k] \neq c$

Showing that for all  $k$ ,  $0 \leq k < i$ :  $s[k] \neq c$  is a loop invariant  
leads to a proof of correctness.

## using loop invariants

Proofs of  
Correctness

the `assert` macro  
enforcing  
preconditions and  
postconditions  
reasoning about  
loops

Efficiency of  
Algorithms

complexity, cost, and  
efficiency  
counting the number  
of operations  
the big-O notation

Given condition: for all  $k$ ,  $0 \leq k < i$ :  $s[k] \neq c$ ,  
verify that this is indeed a **loop invariant**.

Initially:  $\text{result} = -1$  and  $i = 0$ .

Loop stops if: not ( $i < s.\text{length}()$  and  $s[i] \neq c$ ):

- 1  $i == s.\text{length}()$ : test ( $s[i] == c$ ) was never true  
 $\Rightarrow \text{result} == -1$   
 $\Rightarrow$  for all  $k$ ,  $0 \leq k < s.\text{length}()$ :  $s[k] \neq c$
- 2  $s[i] == c$ : execute statements after `if( $s[i] == c$ )`,  
 $\Rightarrow \text{result} == i$   
 $\Rightarrow$  for all  $k$ ,  $0 \leq k < \text{result}$ :  $s[k] \neq c$

Showing that for all  $k$ ,  $0 \leq k < i$ :  $s[i] \neq c$  is a loop invariant  
leads to a proof of correctness.

## using loop invariants

Proofs of  
Correctness

the `assert` macro  
enforcing  
preconditions and  
postconditions  
reasoning about  
loops

Efficiency of  
Algorithms

complexity, cost, and  
efficiency  
counting the number  
of operations  
the big-O notation

Given condition: for all  $k$ ,  $0 \leq k < i$ :  $s[k] \neq c$ ,  
verify that this is indeed a **loop invariant**.

Initially:  $\text{result} = -1$  and  $i = 0$ .

Loop stops if:  $\text{not } (i < s.\text{length}()) \text{ and } s[i] \neq c$ :

- 1  $i == s.\text{length}()$ : test ( $s[i] == c$ ) was never true  
 $\Rightarrow \text{result} == -1$   
 $\Rightarrow$  for all  $k$ ,  $0 \leq k < s.\text{length}()$ :  $s[k] \neq c$
- 2  $s[i] == c$ : execute statements after  $\text{if}(s[i] == c)$ ,  
 $\Rightarrow \text{result} == i$   
 $\Rightarrow$  for all  $k$ ,  $0 \leq k < \text{result}$ :  $s[k] \neq c$

Showing that for all  $k$ ,  $0 \leq k < i$ :  $s[k] \neq c$  is a loop invariant  
leads to a proof of correctness.

## using loop invariants

Proofs of  
Correctness

the `assert` macro  
enforcing  
preconditions and  
postconditions  
reasoning about  
loops

Efficiency of  
Algorithms

complexity, cost, and  
efficiency  
counting the number  
of operations  
the big-O notation

Given condition: for all  $k$ ,  $0 \leq k < i$ :  $s[k] \neq c$ ,  
verify that this is indeed a **loop invariant**.

Initially:  $\text{result} = -1$  and  $i = 0$ .

Loop stops if: not ( $i < s.\text{length}()$  and  $s[i] \neq c$ ):

- 1  $i == s.\text{length}()$ : test ( $s[i] == c$ ) was never true  
 $\Rightarrow \text{result} == -1$   
 $\Rightarrow$  for all  $k$ ,  $0 \leq k < s.\text{length}()$ :  $s[k] \neq c$
- 2  $s[i] == c$ : execute statements after `if( $s[i] == c$ )`,  
 $\Rightarrow \text{result} == i$   
 $\Rightarrow$  for all  $k$ ,  $0 \leq k < \text{result}$ :  $s[k] \neq c$

Showing that for all  $k$ ,  $0 \leq k < i$ :  $s[k] \neq c$  is a loop invariant  
leads to a proof of correctness.

# Proofs of Correctness

## Complexity, Cost and Efficiency

### Proofs of Correctness

the `assert` macro  
enforcing preconditions and postconditions  
reasoning about loops

### Efficiency of Algorithms

**complexity, cost, and efficiency**  
counting the number of operations  
the big-O notation

## 1 Proofs of Correctness

the `assert` macro  
enforcing preconditions and postconditions  
reasoning about loops

## 2 Efficiency of Algorithms

**complexity, cost, and efficiency**  
counting the number of operations  
the big-O notation

# Complexity, cost, and efficiency

## Proofs of Correctness

the `assert` macro  
enforcing  
preconditions and  
postconditions  
reasoning about  
loops

## Efficiency of Algorithms

complexity, cost, and  
efficiency  
counting the number  
of operations  
the big-O notation

Complexity measures how hard a problem is.  
Cost is a property of an algorithm to solve a problem.

Given a particular algorithm, efficiency deals with  
**space** for intermediate and final results;  
**time** for arithmetical operations.

Counting the number of arithmetical operations  
allows to compare algorithms.

# Complexity, cost, and efficiency

## Proofs of Correctness

the `assert` macro  
enforcing preconditions and postconditions  
reasoning about loops

## Efficiency of Algorithms

complexity, cost, and efficiency  
counting the number of operations  
the big-O notation

Complexity measures how hard a problem is.

Cost is a property of an algorithm to solve a problem.

Given a particular algorithm, efficiency deals with

**space** for intermediate and final results;

**time** for arithmetical operations.

Counting the number of arithmetical operations allows to compare algorithms.

# Complexity, cost, and efficiency

## Proofs of Correctness

the `assert` macro  
enforcing preconditions and postconditions  
reasoning about loops

## Efficiency of Algorithms

complexity, cost, and efficiency  
counting the number of operations  
the big-O notation

Complexity measures how hard a problem is.

Cost is a property of an algorithm to solve a problem.

Given a particular algorithm, efficiency deals with

**space** for intermediate and final results;

**time** for arithmetical operations.

Counting the number of arithmetical operations allows to compare algorithms.

# Proofs of Correctness

## Complexity, Cost and Efficiency

### Proofs of Correctness

the `assert` macro  
enforcing preconditions and postconditions  
reasoning about loops

### Efficiency of Algorithms

complexity, cost, and efficiency  
**counting the number of operations**  
the big-O notation

## 1 Proofs of Correctness

the `assert` macro  
enforcing preconditions and postconditions  
reasoning about loops

## 2 Efficiency of Algorithms

complexity, cost, and efficiency  
**counting the number of operations**  
the big-O notation

## inner product

Proofs of  
Correctness

the `assert` macro  
enforcing  
preconditions and  
postconditions  
reasoning about  
loops

Efficiency of  
Algorithms

complexity, cost, and  
efficiency  
counting the number  
of operations  
the big-O notation

Given arrays  $A$  and  $B$  of length  $n$ , compute  $\sum_{k=1}^n A[k]B[k]$ .

```
int inner_product ( int n, int *A, int *B )
{
    int sum = 0;
    for(int i=0; i<n; i++)
        sum = sum + A[i]*B[i];
    return sum;
}
```

How many operations?

The body of the loop gets executed  $n$  times:  
 $n$  additions and  $n$  multiplications.

## inner product

Proofs of  
Correctness

the `assert` macro  
enforcing  
preconditions and  
postconditions  
reasoning about  
loops

Efficiency of  
Algorithms

complexity, cost, and  
efficiency  
counting the number  
of operations  
the big-O notation

Given arrays  $A$  and  $B$  of length  $n$ , compute  $\sum_{k=1}^n A[k]B[k]$ .

```
int inner_product ( int n, int *A, int *B )
{
    int sum = 0;
    for(int i=0; i<n; i++)
        sum = sum + A[i]*B[i];
    return sum;
}
```

How many operations?

The body of the loop gets executed  $n$  times:  
 $n$  additions and  $n$  multiplications.

# Counting Duplicates

## Proofs of Correctness

the `assert` macro  
enforcing preconditions and postconditions  
reasoning about loops

## Efficiency of Algorithms

complexity, cost, and efficiency  
counting the number of operations  
the big-O notation

```
int find_duplicates ( int n, int *A, int *B )
{
    int cnt = 0;
    for(int i=0; i<n; i++)
        for(int j=0; j<n; j++)
            if(A[i] == B[j])
                cnt++;
    return cnt;
}
```

## How many comparisons?

The body of the outer loop runs  $n$  times.

For every  $i$  in the outer loop, the body of the inner loop gets executed  $n$  times  $\Rightarrow n^2$  times.

# Counting Duplicates

## Proofs of Correctness

the `assert` macro  
enforcing preconditions and postconditions  
reasoning about loops

## Efficiency of Algorithms

complexity, cost, and efficiency  
counting the number of operations  
the big-O notation

```
int find_duplicates ( int n, int *A, int *B )
{
    int cnt = 0;
    for(int i=0; i<n; i++)
        for(int j=0; j<n; j++)
            if(A[i] == B[j])
                cnt++;
    return cnt;
}
```

How many comparisons?

The body of the outer loop runs  $n$  times.

For every  $i$  in the outer loop, the body of the inner loop gets executed  $n$  times  $\Rightarrow n^2$  times.

# Linear versus Quadratic Time

## Proofs of Correctness

the `assert` macro  
enforcing  
preconditions and  
postconditions  
reasoning about  
loops

## Efficiency of Algorithms

complexity, cost, and  
efficiency  
counting the number  
of operations  
the big-O notation

Let  $n$  be the dimension of a problem

$n$	cost	
	$2n$	$n^2$
1	2	1
2	4	4
4	8	16
8	16	64

Linear time: double  $n$ , double cost.

Quadratic time: double  $n$ , cost quadruples.

# Proofs of Correctness

## Complexity, Cost and Efficiency

### Proofs of Correctness

the `assert` macro  
enforcing preconditions and postconditions  
reasoning about loops

### Efficiency of Algorithms

complexity, cost, and efficiency  
counting the number of operations  
the big-O notation

## 1 Proofs of Correctness

the `assert` macro  
enforcing preconditions and postconditions  
reasoning about loops

## 2 Efficiency of Algorithms

complexity, cost, and efficiency  
counting the number of operations  
the big-O notation

# The big-O Notation

## Proofs of Correctness

the `assert` macro  
enforcing preconditions and postconditions  
reasoning about loops

## Efficiency of Algorithms

complexity, cost, and efficiency  
counting the number of operations  
the big-O notation

Let  $n$  be the dimension of the problem.

A function  $f(n)$  is  $O(g(n))$  (we say:  $f$  is of order  $g$ ) if there exists a positive constant  $c$  (*independent of  $n$* ):  
 $f(n) \leq cg(n)$ , for sufficiently large  $n$ .

Big-O defines the order of complexity, some examples:

- $f$  is  $O(1)$ : constant
- $f$  is  $O(\log(n))$ : logarithmic in  $n$
- $f$  is  $O(n)$ : linear in  $n$
- $f$  is  $O(n \log(n))$ : log-linear in  $n$
- $f$  is  $O(n^2)$ : quadratic in  $n$
- $f$  is  $O(2^n)$ : exponential in  $n$
- $f$  is  $O(n!)$ : factorial in  $n$

# The big-O Notation

## Proofs of Correctness

the `assert` macro  
enforcing preconditions and postconditions  
reasoning about loops

## Efficiency of Algorithms

complexity, cost, and efficiency  
counting the number of operations  
the big-O notation

Let  $n$  be the dimension of the problem.

A function  $f(n)$  is  $O(g(n))$  (we say:  $f$  is of order  $g$ ) if there exists a positive constant  $c$  (*independent of  $n$* ):  $f(n) \leq cg(n)$ , for sufficiently large  $n$ .

Big-O defines the order of complexity, some examples:

- $f$  is  $O(1)$ : constant
- $f$  is  $O(\log(n))$ : logarithmic in  $n$
- $f$  is  $O(n)$ : linear in  $n$
- $f$  is  $O(n \log(n))$ : log-linear in  $n$
- $f$  is  $O(n^2)$ : quadratic in  $n$
- $f$  is  $O(2^n)$ : exponential in  $n$
- $f$  is  $O(n!)$ : factorial in  $n$

# The Permanent of a Matrix

## Proofs of Correctness

the `assert` macro  
enforcing preconditions and postconditions  
reasoning about loops

## Efficiency of Algorithms

complexity, cost, and efficiency  
counting the number of operations  
the big-O notation

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \quad \text{per}(A) =$$

$$\begin{aligned} & a_{1,1}a_{2,2}a_{3,3} + a_{1,1}a_{2,3}a_{3,2} \\ & + a_{1,2}a_{2,1}a_{3,3} + a_{1,2}a_{2,3}a_{3,1} \\ & + a_{1,3}a_{2,1}a_{3,2} + a_{1,3}a_{2,2}a_{3,1} \end{aligned}$$

For a general  $n$ -by- $n$  matrix  $A$ , with entries  $a_{i,j}$ :

$$\text{per}(A) = \sum_{\sigma} \prod_{i=1}^n a_{i,\sigma(i)}$$

where  $\sigma$  is a permutation of  $(1, 2, \dots, n)$ .

The sum  $\Sigma$  runs of all permutations  $\sigma$ . As there are  $n!$  permutations, the cost for  $\text{per}(A)$  is  $O(n!)$ .

# Timing Code

## Proofs of Correctness

the `assert` macro  
enforcing preconditions and postconditions  
reasoning about loops

## Efficiency of Algorithms

complexity, cost, and efficiency  
counting the number of operations  
the big-O notation

```
#include <ctime>
```

```
int main()
```

```
{
```

```
    ...
```

```
    long int tstart = clock();
```

```
    // statements to be timed
```

```
    long int tstop = clock();
```

```
    long int e; // elapsed time
```

```
    e = 1000*(tstop - tstart)/CLOCKS_PER_SEC;
```

```
    cout << "elapsed time : " << e
```

```
         << " milliseconds" << endl;
```

# Summary + Assignments

## Proofs of Correctness

the `assert` macro  
enforcing preconditions and postconditions  
reasoning about loops

## Efficiency of Algorithms

complexity, cost, and efficiency  
counting the number of operations  
the big-O notation

Ended Chapter 2: *Program Correctness and Efficiency*,  
We enforce preconditions and postconditions with `assert`  
and prove correctness with loop invariants.

## Assignments:

- 1 Write an exception handler in case `assert(ins)` fails, where `ins` an input file stream. The error message written by the handler involves the file name used in the construction of `ins`.
- 2 Time the execution for the counting of the duplicates for sufficiently large dimension. Double the dimension and verify whether the time quadruples.
- 3 Use a triple loop to search for duplicates in three arrays. Verify experimentally whether the execution time is multiplied by 8 if the dimension of the arrays doubles.