יעילות, סגנון ובדיקה של קוד

"איכות" של פיתרון קוד לבעייה

- נכונות
- פשטות (תכנות)
- יעילות

"איכות" של פיתרון קוד לבעייה

- נכונות
- פשטות (תכנות)
- יעילות

מה יותר חשוב?

"איכות" של פיתרון קוד לבעייה

- נכונות
- פשטות (תכנות)
- יעילות

מה יותר חשוב?

תלוי לשם מה...

לעיתים נסכים אפילו להקריב נכונות בשביל יעילות.

"איכות" של פיתרון קוד לבעייה

- נכונות – **אמצעים: סגנון (style) בדיקה (testing) ודיבוג (debugging) לפי סדר החשיבות**
- פשטות (תכנות) – **אמצעים: סגנון**
- יעילות – **אמצעים: שקפים הבאים**

יעילות של תוכנית

- כאשר תוכנית מחשב רצה פעולות לוקחות זמן (וגם משאבים אחרים כמו זיכרון)
- האם אנחנו צריכים לכתוב אותה כך שזמן הריצה יהיה קצר יותר?

יעילות של תוכנית

- כאשר תוכנית מחשב רצה פעולות לוקחות זמן (וגם משאבים אחרים כמו זיכרון)
- האם אנחנו צריכים לכתוב אותה כך שזמן הריצה יהיה קצר יותר?
  - לא תמיד – לפעמים אנחנו יכולים לדעת מראש שזמן הריצה לקלטים אפשריים הוא מספיק מהר על החומרה של היום.

יעילות של תוכנית

- כאשר תוכנית מחשב רצה פעולות לוקחות זמן (וגם משאבים אחרים כמו זיכרון)
- האם אנחנו צריכים לכתוב אותה כך שזמן הריצה יהיה קצר יותר?
  - לא תמיד – לפעמים אנחנו יכולים לדעת מראש שזמן הריצה לקלטים אפשריים הוא מספיק מהר על החומרה של היום.

מראש: לפני תכנות או אחרי תכנות?

סיבוכיות זמן ריצה

- רוצים דרך להשוות אלגוריתמים לפני שמתכנתים אותם ועל מנת שנוכל לקבל "סדר גודל" של זמן ריצה לפני התכנות.
  - חוסך זמן מימוש/תכנות לאלגוריתמים איטיים מדי
  - חוסך זמן מימוש/תכנות אם אין צורך לתכנת משהוא מהיר ומסובך שאין צורך בכך

חיפוש ערך מקסימלי במערך

C++:

```
int max_val(const vector<int>& vec) {
  int max = vec[0];
  for (size_t i = 1; i < vec.size(); ++i) {
    if (vec[i]>max) max = vec[i];
  }
  return max;
}
```

Matlab:

```
M=max(A)% behind the scenes a loop like in C++
```

- איך נבדוק זמן ריצה? נספור פעולות? נמדוד?

חיפוש ערך מקסימלי במערך

- איך נבדוק זמן ריצה? נספור פעולות?
  - בעייה 1: מספר הפעולות תלוי בקלט ולא רק בגודלו:

```
if (vec[i]>max) max = vec[i];
```

  - פיתרון: "worst case analysis": שם יפה ל "מה קורה שאין מזל"
  - במקרה שלנו – האיבר המקסימלי הוא האיבר האחרון.

חיפוש ערך מקסימלי במערך

- איך נבדוק זמן ריצה? נספור פעולות?
  - בעייה 2: מספר פעולות של ביטוי כלשהוא תלוי שפת תכנות, צורה בה מקמפלים, מחשב

```
vec[i];
```

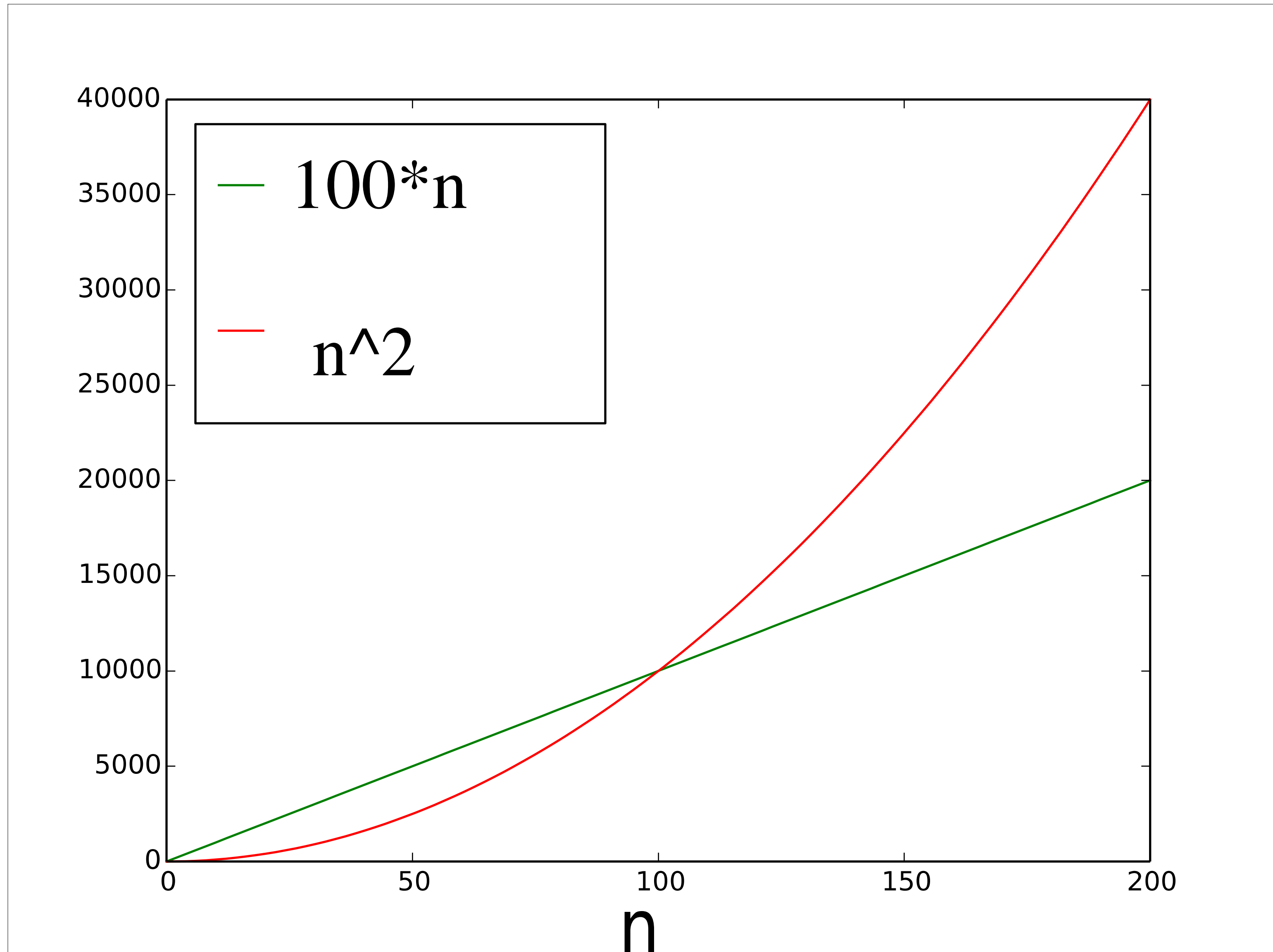
- פתרון: נספור רק "סדר גודל" של מספר פעולות.
- כלומר, נשאיר רק את הביטויים שגדלים הכי מהר שגודל הקלט גדל ונוריד את המקדם.

חיפוש ערך מקסימלי במערך

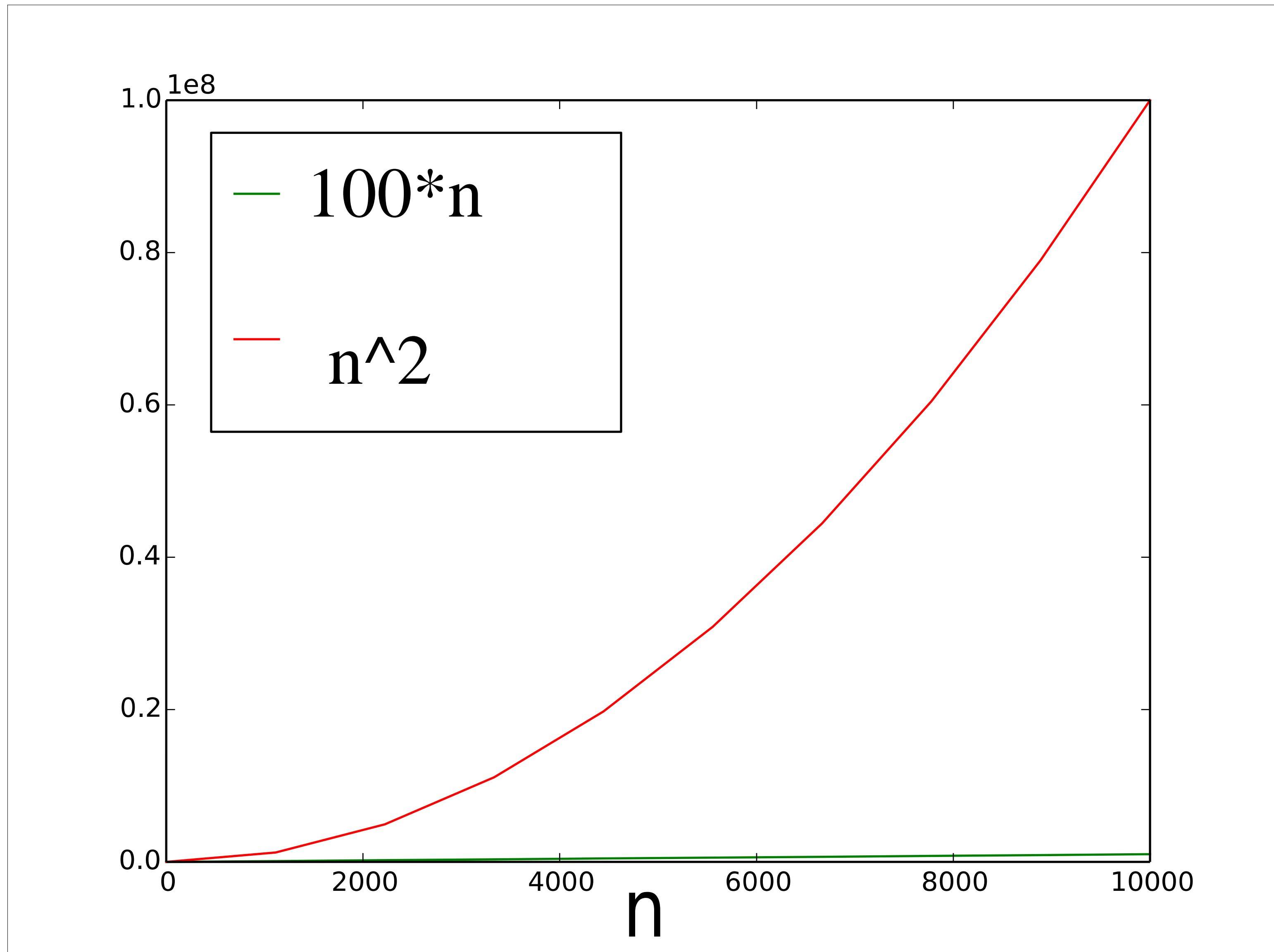
```
int max_val(const vector<int>& vec) {
  int max = vec[0];
  for (size_t i = 1; i < vec.size(); ++i) {
    if (vec[i]>max) max = vec[i];
  }
  return max;
}
```

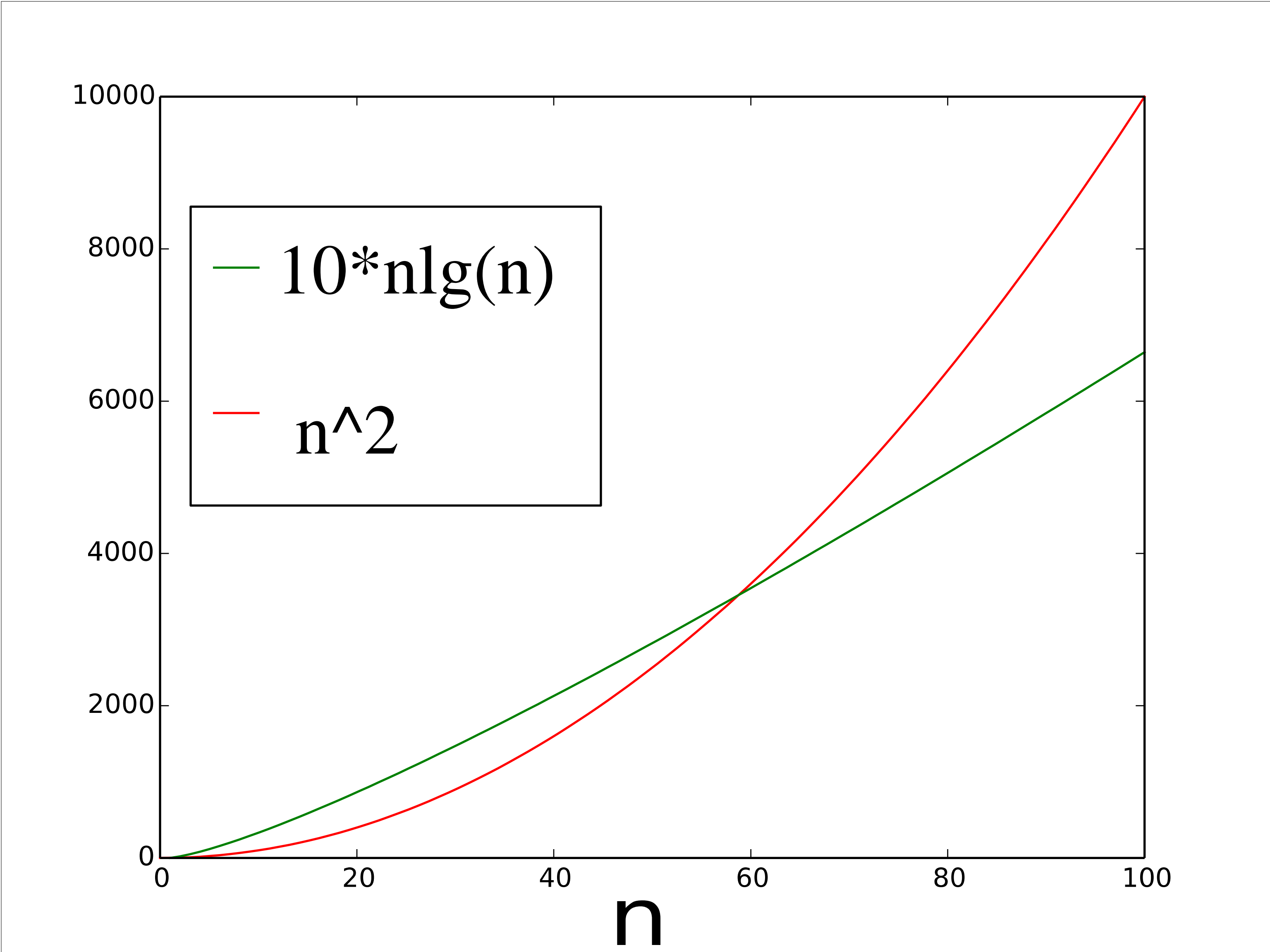
- סדר הגודל של מספר הפעולות: vec.size() פעולות

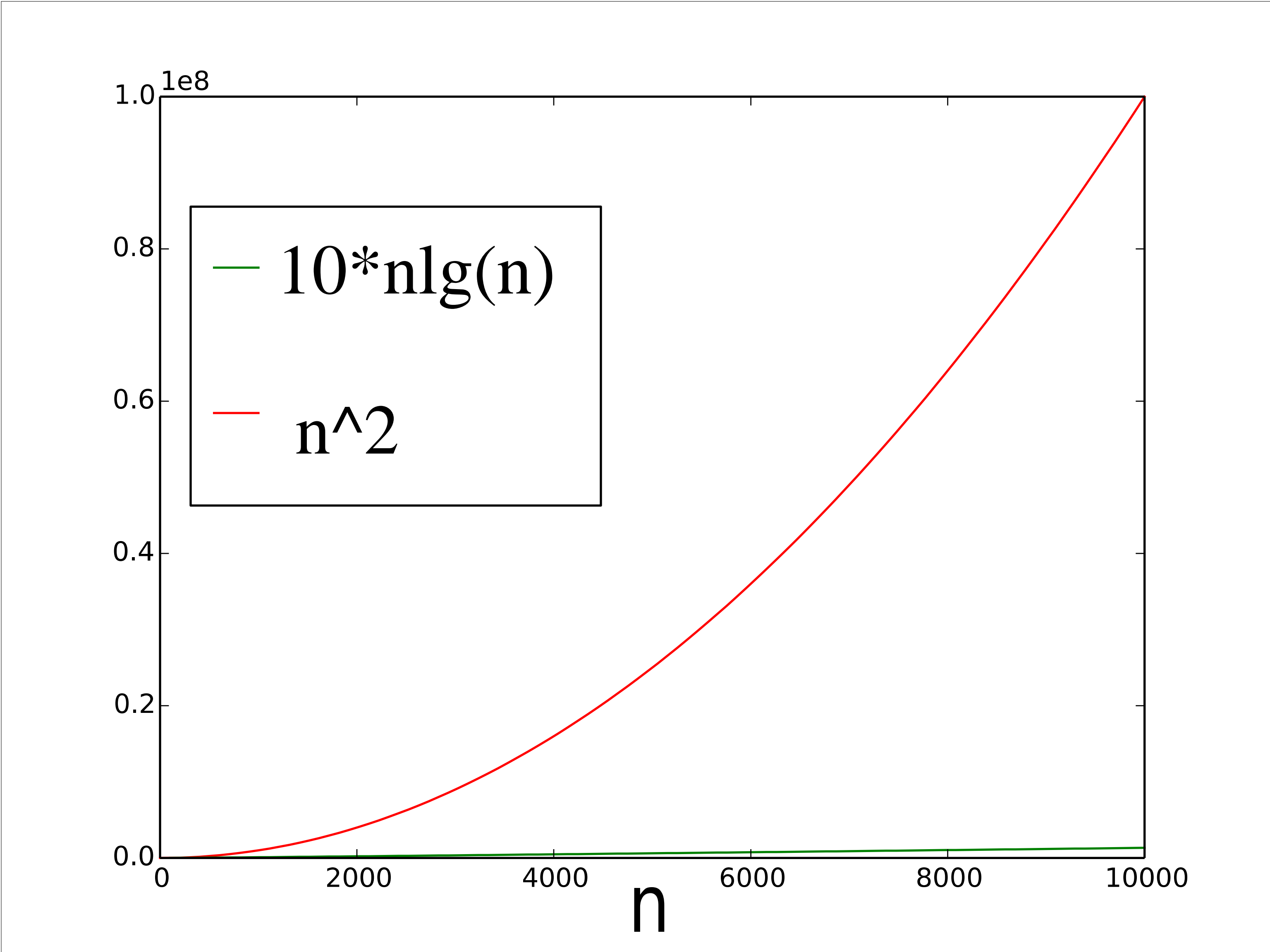
למה סדרי גודל?



למה סדרי גודל?




10*nlg(n)
n^2
10000
8000
6000
4000
2000
0
0
20
40
60
80
100
n

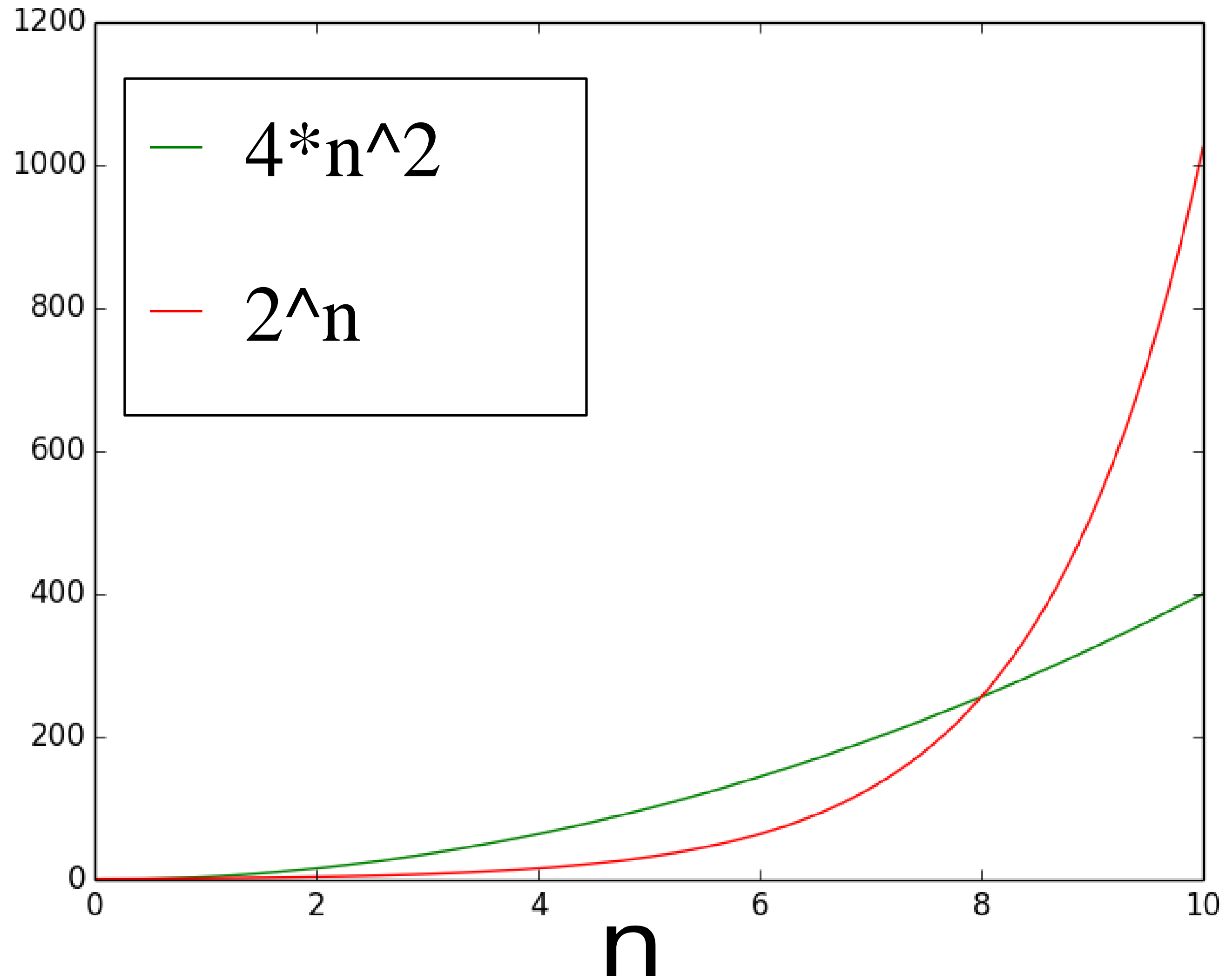


1e8
1.0
0.8
0.6
0.4
0.2
0.0
0
2000
4000
6000
8000
10000
n
10*nlg(n)
n^2

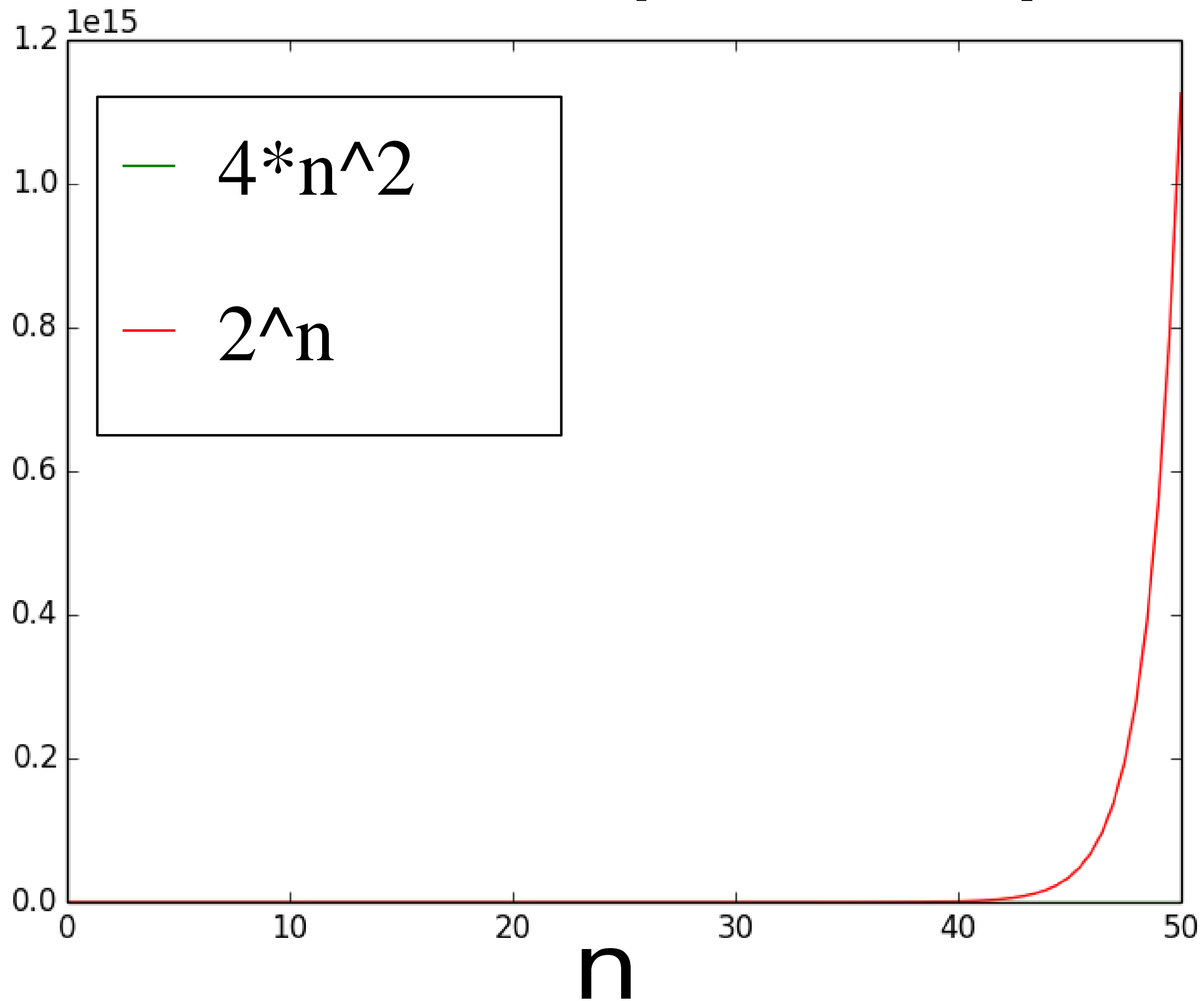
מיון מהיר יותר – merge sort

- נניח שאנחנו צריכים למיין מיליון מספרים
- מיון בחירה יצטרך כפולה של $(10^6)^2$ פעולות שווה ל: **1,000,000,000,000** פעולות
- מיון מיזוג יצטרך כפולה של $(10^6 \lg 10^6)$ פעולות שווה ל: **20,000,000** פעולות.

זמן ריצה אקספוניניציאלי



זמן ריצה אקספוננציאלי



טיפים למתקדמים

- לבדוק אם יש קוד שמממש כבר את מה שצריך (נדיר) או מממש פונקציות שאני צריך להשתמש בהן (מאד נפוץ)
- לבדוק אם בשביל גודל הקלט הצפוי יש בעייה בכלל להריץ את האלגוריתם.
- טכניקת ייעול מאד נפוצה: לזכור תוצאות חישוב ביניים.

טיפים למתקדמים

- אם עובדים אם מטלב: אפשר לכתוב איזורי צוואר בקבוק ב C++ ע"י טכניקת mex
- אם עובדים בשפה שאפשר לקמפל אותה אז לקמפל עם "דגלים" של אופטימיזציה
- אפשר גם לבדוק התאמת קימפול לחומרה הספיציפית: "דגלים" מיוחדים לקומפיילר.
- אם עדיין צריך להאיץ את התוכנית ולא מוצאים משהוא יעיל יותר בסדר גודל – צריך למדוד זמני ריצה ולחפש צוואר בקבוק (profiling)

דוגמא: median filtering

- אלגוריתם "נאיבי": מיין וקח את נקודת האמצע

זמן ריצה =$O(N^2 (K^2 \lg(K^2)))$ (N מספר שורות/עמודות תמונה,

$O(N^2 K^2 \lg(K))$ K גודל הפילטר)

- אלגוריתם "נאיבי 2": מצא median באופן ישיר ע"י אלגוריתם median of median

זמן ריצה = $O(N^2 K^2)$

- אלגוריתם "החזק היסטוגרמה ועדכן אותה"

זמן ריצה = $O(N^2(G+K))$ (G מספר דרגות אפור, בד"כ 255)

T. Huang, G. Yang, and G. Tang, “A Fast Two-Dimensional Median Filtering Algorithm,”, 1979.

דוגמא: median filtering

- אלגוריתם "החזק היסטוגרמות ועדכן אותן"

Median Filtering in Constant Time

Simon Perreault and Patrick Hebert, 2007

זמן ריצה = O(N^2 G)

במאמר עוד הרבה אופטימיזציות למחשבים מודרניים (לא משנים סדר גודל של זמן ריצה אבל משנים פרקטית).

יש גם קוד באתר: http://nomis80.org/ctmf.html

Program style

Program style

1. Take the time for it. **Very important**. If not followed your code will be “**write only**”.
2. Common sense
3. Read “real” coding guidelines document – will give you insight how important it is. e.g. good one from Microsoft or one from Google or for Matlab.

Principles:

1. Readability
2. Common Sense
3. Clarity
4. Right focus

What's in a name

```
Example
ONE= 1
TEN= 10
TWENTY= 20

More reasonable
INPUT_MODE= 1
INPUT_BUFSIZE= 10
OUTPUT_BUFSIZE= 20
```

What's in a name

Use descriptive names

```
npending = 0; // current length of input queue
```

Naming conventions vary (style)

- `numPending`
- `num_pending`
- `NumberOfPendingEvents`

- Be consistent, with yourself and peers.

What's in a name

Consider (wording)

```
noOfItemsInQ

frontOfTheQueue

queueCapacity

...
```

The word "queue" appears in 3 different ways

- Be consistent, with yourself and peers.

What's in a name

Compare

```
for theElementIndex= 1:numberOfElements
  elementArray(theElementIndex)= theElementIndex;
```

and

```
for i=1:N
  elem(i)= i;
```

Use short names for locals

What's in a name

Use active name for functions

```
now = getDate()
```

Compare

```
if( checkdigit(c) ) …
```

to

```
if( isdigit(c) ) …
```

Accurate active names makes bugs apparent

Indentation

Use indentation to show structure

Compare

```
 for n= 1:N   weightArr(n)= weight(n);
   heightArr(n)= height(n); end
```

To

```
for n= 1:N
  weightArr(n)= weight(n);
  heightArr(n)= height(n);
end
```

Expressions

Use parentheses to resolve ambiguity

Compare

```
leap_year = y % 4 == 0 && y %100 != 0
            || y % 400 == 0;
```

to

```
leap_year = ((y % 4 == 0) && (y %100 != 0))
             || (y % 400 == 0);
```

Statements (C, C++, Not Matlab)

Use braces to resolve ambiguity

Compare

```
if( i < 100 )
x = i;
i++;
```

To

```
if( i < 100 )
{
  x = i;
}
i++;
```

Idioms (C, C++, Not Matlab)

Do not try to make code “interesting”!

```
i = 0;
while( i <= n-1 )
{
    array(i++) = 1;
}
for( i = 0; i < n; )
{
    array[i++] = 1;
}
```

```
for( i = n; --i >= 0; )
{
    array[i] = 1;
}
for( i = 0; i < n; i++ )
{
    array[i] = 1;
}
```

This is the common “idiom” that any programmer will recognize

Idioms

Use “elseif” for multiway decisions

```
if ( cond_1 )
  statement_1
elseif ( cond_2 )
  statement_2
…
elseif ( cond_n )
  statement_n
else
  default-statement
end
```

Idioms

Compare:

```
if ( x > 0 )
   if ( y > 0 )
     if ( x+y < 100 )

        ...

     else
        fprintf('Too large!\n');
     end
   else
     fprintf('y too small!\n');
   end
else
   fprintf('x too small!\n');
```

```
if ( x <= 0 )
  fprintf('x too small!\n');
elseif ( y <= 0 )
  fprintf('y too small!\n');
elseif ( x+y >= 100 )
  printf('Sum too large!\n');
else
  …
end
```

Comments

Don’t write the obvious

```
% return SUCCESS
SUCCESS= true;
% Initialize total to number_received
total = number_received;
```

Test:

Does comment add something that is not evident from the code?

Comments

Don't comment bad code – rewrite it!

```
function [out] = ...
...
% If result == 0 a match was found so return
% true; otherwise return false;
out= ~result;
```

Instead

```
Function [foundMatch] = ...
...
```

Comments

Write comments that will help the story and

Write comments for each function that describes its legal contract:

- That is, the function promises to _________ if the input is _______
- Full description of its parameters

Style recap

- Descriptive names
- Clarity in expressions
- Straightforward flow
- Readability of code & comments
- Consistent conventions & idioms

Why Bother?

Good style:

- Easy to understand code
- Smaller & polished
- Makes errors apparent

Sloppy style → bad code

- Hard to read
- Broken flow
- Harder to find errors & correct them

Debugging

Debugging 101

1. “Define” the bug --- reproduce it
2. Use debugger or/and printouts
3. Don’t panic --- think!
4. Divide & Conquer
5. Test before instead of debugging after

Unit Tests

Some slides are from: Hong Qing Yu &
David Matuszek

The Testing Problems

Should write

Do

programmers

few

Why?

I am so busy

It is difficult

OK, I’ll write tests, but what should I use a framework ?

Disadvantages of writing a test suite

- I need to learn a new thing
  - *True*—but done once
- You don’t have time to do all that extra work
  - *False*—Experiments repeatedly show that test suites reduce debugging time more than the amount spent building the test suite

Advantages of having a test suite

- Your program will have many fewer bugs
- It will be easier to catch bugs if you’ll have ones
- It will be a ***lot*** easier to maintain and modify your program
  - This is a *huge* win for programs that, unlike class assignments, get actual use !

Example (Java): Old way vs. new way

```java
int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

void testMax() {
    int x = max(3, 7);
    if (x != 7) {
        System.out.println("max(3, 7) gives " + x);
    }
    x = max(3, -7);
    if (x != 3) {
        System.out.println("max(3, -7) gives " + x);
    }
}

public static void main(String[] args) {
    new MyClass().testMax();
}
```

```java
@Test
void testMax() {
    assertEquals(7, max(3, 7));
    assertEquals(3, max(3, -7));
}
```

XP approach to testing

- Write tests → Write code
- Code without test → Automatic fail
- Small code change → Tests run
- Bug found and fixed → Add test for it

Recommended approach

1. Write a test for some function you intend to write
2. Write a stub for the function
3. Run the test and make sure it fails
4. Replace the stub with code: just enough to pass the test
5. Run the test
6. If it fails, debug the function, or the test until the test passes
7. If the function needs to do more, or handle more complex situations: Do steps 3, 4, 5, 6 again for this functionality or input

Refactoring

- Refactoring is the process of code improvement where code is reorganized and rewritten to make it more efficient, easier to understand, etc.
- Refactoring is required because code tends to become messy.
- Refactoring should not change the functionality of the system.
- Automated testing simplifies refactoring as you can see if the changed code still runs the tests successfully.