

CISS350: Data Structures and Advanced Algorithms Assignment 4

OBJECTIVES

1. Computation of big- O of the worst case runtime performance a given algorithm.
2. Computation of big- O of the best case runtime performance a given algorithm.
3. Computation of big- O of the average case runtime performance a given algorithm.

In your Fedora virtual machine, as **student** user, execute the following in your bash shell:
pip install scipy --user.

For this assignment, you will need to modify q02.tex, q03.tex, etc. q01.tex has a complete solution for your reference.

Given an algorithm, $T_w(n)$ denotes the worst runtime, $T_a(n)$ the average runtime, and $T_b(n)$ the best runtime. $T(n)$ denotes the worst runtime of the algorithm.

Recall that

$$An^2 + Bn + C = O(n^2)$$

and

$$Bn + C = O(n)$$

and

$$C = O(1)$$

where A, B , and C are constants. In general

$$a_d n^d + a_{d-1} n^{d-1} + \cdots + a_0 = O(n^d)$$

where a_i 's are constants. Of course it's also true that

$$3n^2 - 5n + 10 = O(n^3)$$

and

$$3n^2 - 5n + 10 = O(n^{3.5})$$

and in fact

$$3n^2 - 5n + 10 = O(n^k)$$

for any real number $k \geq 2$. But the practice is to provide the best (i.e. tightest) bound.

For grading purposes, you must follow the following instructions:

1. When assigning times t_i , start with t_1 (not t_0) and use t_1, t_2, \dots without skipping any index.
2. When writing down exact runtimes, list the term with the highest growth rate. For instance, do not write

$$T(n) = t_1 + (t_2 + t_6)n^{100} + t_3n^{15.25}$$

Instead, write this:

$$T(n) = (t_2 + t_6)n^{100} + t_3n^{15.25} + t_1$$

3. When writing the constants for each term in the runtime, arrange the constants in ascending index values. For instance, do not write this:

$$T(n) = (t_3 + t_1 + t_2)n^{100} + (t_7 + t_2 + t_1)n^{15.25}$$

Instead, write this:

$$T(n) = (t_1 + t_2 + t_3)n^{100} + (t_1 + t_2 + t_7)n^{15.25}$$

4. Write legibly and clearly. If there's any ambiguity, then I reserve the right to interpret what you are trying to convey. And I am usually very good at picking the wrong interpretation.
5. Good writing means clear and unambiguous writing. " $x = 42$ " is clearer than "42". It does not even take that much time (nor waste ink) to write the former. If I see any ambiguity like the above, you will get a 0 for the whole question.
6. There's a big difference between " $x = 42$ " and " x 42". No one would say a "John tall" when it should be "John is tall". If I see something similar to " x 42", you will get a 0 for whole question.
7. In general, if I see any obviously bad/improper mathematical writing, I will give you a zero.

Q1. SOLUTION PROVIDED.

This is a solved problem. All the big- O computations in this assignment are somewhat similar to this question. Therefore study the solution very carefully. It will help you present your solutions in L^AT_EX.

The goal is to compute the big- O of the runtime performance of the following algorithm. Here's the algorithm:

```
INPUT:  x - array of doubles
        n - size of x
OUTPUT: result is stored in z
ALGORITHM:

    for i = 0, 1, 2, ..., n - 1:
        z = z + x[i] * x[i]
```

This is rewritten with timings as follows:

```
INPUT:  x - array of doubles
        n - size of x
OUTPUT: result is stored in z
ALGORITHM:
    i = 0                                time t1
LOOP:   if i >= n:                        time t2
        goto ENDLOOP                    time t3
        z = z + x[i] * x[i]              time t4
        i = i + 1                        time t5
        goto LOOP                        time t6
ENDLOOP:
```

The only relevant timing of the statements are given.

(a) Write down $T(n)$ in terms of n and the t_1, t_2, t_3, \dots . You should write it as a polynomial of n from the highest degree term to the lowest.

(b) Write down the big- O of $T(n)$ as $O(n^k)$ where k is the smallest possible positive integer.

SOLUTION

(a) The timings with the number of times a statement is executed is as follows:

	i = 0	time t1	1
LOOP:	if i >= n:	time t2	n + 1
	goto ENDLOOP	time t3	1
	z = z + x[i] * x[i]	time t4	n
	i = i + 1	time t5	n
	goto LOOP	time t6	n
ENDLOOP:			

Therefore

$$T(n) = (t_2 + t_4 + t_5 + t_6)n + (t_1 + t_2 + t_3)$$

(b) We have

$$T(n) = O(n)$$

□

Q2. The goal is to compute the big- O of the runtime performance of the following algorithm. Here's the algorithm:

```
INPUT:  x - array of doubles
        n - size of x
        a - double
        b - int
OUTPUT: result is stored in z
ALGORITHM:
    z = a + b * b

    for i = 0, 1, 2, ..., n-1:
        a = a + 1
        x[i] = a + z + b;
        z = z + 1

    for i = 0, 1, 2, ..., (n-1)/4:
        x[i] = x[i] * z

    z = z + n
```

This is rewritten with timings as follows:

```
INPUT:  x - array of doubles
        n - size of x
        a - double
        b - int
OUTPUT: z
ALGORITHM:
    z = a + b * b                                time t1

    i = 0                                         time t2
LOOP1:  if i >= n:                               time t3
        goto ENDLOOP1                           time t4
    a = a + 1                                    time t5
    x[i] = a + z + b;                            time t6
    z = z + 1                                    time t7
    i = i + 1                                    time t8
    goto LOOP1                                   time t9
ENDLOOP1:
```

	i = 0	time t10
LOOP2:	if i > (n - 1) / 4:	time t11
	goto ENDLOOP2	time t12
	x[i] = x[i] * z	time t13
	i = i + 1	time t14
	goto LOOP2	time t15
ENDLOOP2:		
	z = z + n	time t16

(You can pretend that $n - 1$ is divisible by 4 so that $(n - 1)/4$ is an integer.)

(a) Write down $T(n)$ in terms of n and the t_1, t_2, t_3, \dots . You should write it as a polynomial of n from the highest degree term to the lowest.

(b) Write down the big- O of $T(n)$ as $O(n^k)$ where k is the smallest possible positive integer.

SOLUTION

(a) The timings with the number of times a statement is executed is as follows (complete it like the solution given earlier):

	<code>z = a + b * b</code>	<code>time t1</code>
	<code>i = 0</code>	<code>time t2</code>
<code>LOOP1:</code>	<code>if i >= n:</code>	<code>time t3</code>
	<code>goto ENDLLOOP1</code>	<code>time t4</code>
	<code>a = a + 1</code>	<code>time t5</code>
	<code>x[i] = a + z + b;</code>	<code>time t6</code>
	<code>z = z + 1</code>	<code>time t7</code>
	<code>i = i + 1</code>	<code>time t8</code>
	<code>goto LOOP1</code>	<code>time t9</code>
<code>ENDLOOP1:</code>		
	<code>i = 0</code>	<code>time t10</code>
<code>LOOP2:</code>	<code>if i > (n - 1) / 4:</code>	<code>time t11</code>
	<code>goto ENDLLOOP2</code>	<code>time t12</code>
	<code>x[i] = x[i] * z</code>	<code>time t13</code>
	<code>i = i + 1</code>	<code>time t14</code>
	<code>goto LOOP2</code>	<code>time t15</code>
<code>ENDLOOP2:</code>		
	<code>z = z + n</code>	<code>time t16</code>

Q3. The goal is to compute the big- O of the runtime performance of the following algorithm. (The best, worst, and average runtimes are the same.) Here's the algorithm:

```
INPUT:  x - array of doubles
        n - size of x
        a - int
OUTPUT: None
ALGORITHM:

    a = a * a

    for i = 0, 1, 2, ..., n-1:
        x[i] = x[i] + a

        for j = 0, 1, 2, ..., i:
            x[j] = x[i] * a
```

- (a) Write down $T(n)$ in terms of n and the t_1, t_2, \dots . You should write it as a polynomial of n from the highest degree term to the lowest.
- (b) Write down the big- O of $T(n)$ as $O(n^k)$ where k is the smallest possible positive integer.

SOLUTION

(a) The timings with the number of times a statement is executed is as follows (complete it like the solution given earlier):

	a = a * a	time t1
	i = 0	time t2
LOOP1:	if i >= n:	time t3
	goto ENDLOOP1	time t4
	x[i] = x[i] + a	time t5
	j = 0	time t6
LOOP2:	if j > i:	time t7
	goto ENDLOOP2	time t8
	x[j] = x[i] * a	time t9
	j = j + 1	time t10
	goto LOOP2	time t11
ENDLOOP2:	i = i + 1	time t12
	goto LOOP1	time t13
ENDLOOP1:		

Q4. The goal is to compute the big- O of the best and worst runtime performance of the following algorithm. Here's the algorithm:

```

INPUT:  x - array of doubles
        n - size of x
        a - int
OUTPUT: None
ALGORITHM:

    a = x[0] / 2

    for i = 0, 1, 2, ..., 100000:

        for j = 0, 1, 2, ..., n-1:
            x[j] = x[j] + a

        for j = 0, 1, 2, ..., (n - 1)/2:
            if x[j] < 5:
                x[j] = x[j] * x[0]

```

```

INPUT:  x - array of doubles
        n - size of x
        a - int
OUTPUT: None
ALGORITHM:

        a = x[0] / 2                time t1

        i = 0                        time t2
LOOP1:   if i > 100000:              time t3
        goto ENDLOOP                time t4

        j = 0                        time t5
LOOP2:   if j > n - 1:              time t6
        goto ENDLOOP2              time t7
        x[j] = x[j] + a             time t8
        j = j + 1                   time t9
        goto LOOP2                  time t10
ENDLOOP2:

        j = 0                        time t11
LOOP3:   if j > (n - 1)/2:          time t12

```

	goto ENDL00P3	time t13
	if x[j] >= 5:	time t14
	goto ELSE	time t15
	x[j] = x[j] * x[0]	time t16
ELSE:	j = j + 1	time t17
	goto L00P3	time t18
ENDL00P3:		
	i = i + 1	time t19
	goto L00P1	time t20
ENDL00P1:		

- (a) Write down $T_w(n)$ in terms of n and the t_1, t_2, \dots . You should write it as a polynomial of n from the highest degree term to the lowest.
- (b) Write down the big- O of $T_w(n)$ as $O(n^k)$ where k is the smallest possible positive integer.
- (c) Write down the big- O of $T_b(n)$ as $O(n^k)$ where k is the smallest possible positive integer. (You are strongly advised to compute the precise $T_b(n)$ on your own of course. For grading purposes, you just have to write down the big- O .)

SOLUTION

(a) The timings with the number of times a statement is executed is as follows (complete it like the solution given earlier):

	a = x[0] / 2	time t1
	i = 0	time t2
LOOP1:	if i > 100000:	time t3
	goto ENDLLOOP	time t4
	j = 0	time t5
LOOP2:	if j > n - 1:	time t6
	goto ENDLLOOP2	time t7
	x[j] = x[j] + a	time t8
	j = j + 1	time t9
	goto LOOP2	time t10
ENDLOOP2:		
	j = 0	time t11
LOOP3:	if j > (n - 1)/2:	time t12
	goto ENDLLOOP3	time t13
	if x[j] >= 5:	time t14
	goto ELSE	time t15
	x[j] = x[j] * x[0]	time t16
ELSE:	j = j + 1	time t17
	goto LOOP3	time t18
ENDLOOP3:		
	i = i + 1	time t19
	goto LOOP1	time t20
ENDLOOP1:		

Q5. The goal is to compute the big- O of the best and worst runtime performance of the following algorithm. Here's the algorithm:

```
INPUT:  x - array of doubles
        n - size of x
        a - int
OUTPUT: None
ALGORITHM:

    a = x[0] / 2

    for i = 0, 1, 2, ..., 100000:

        if a < 100:
            for j = 0, 1, 2, ..., n-1:
                x[i] = x[i] + a

            for j = 0, 1, 2, ..., (n-1)/2:
                if x[j] < 5:
                    x[j] = x[j] * x[0]
```

(During your analysis, just treat $n - 1$ as though it's even so that $(n - 1)/2$ is an integer.)

(a) Write down the big- O of $T_b(n)$ as $O(n^k)$ where k is the smallest possible positive integer. (You are strongly advised to write down the precise expression for $T_b(n)$ on your own. For grading purposes, you just have to write down the big- O of $T_b(n)$.)

(b) Write down the big- O of $T_w(n)$ as $O(n^k)$ where k is the smallest possible positive integer. (You are strongly advised to write down the precise expression for $T_w(n)$ on your own. For grading purposes, you just have to write down the big- O of $T_w(n)$.)

SOLUTION

Q6. State the big-O of the runtime of the following sorting algorithm

```
INPUT:  x - array
        n - size of array x

ALGORITHM:

for i = 0, 1, 2, ..., n - 2:
    for j = 0, 1, 2, ..., n - 2:
        if x[j] < x[j + 1]:
            t = x[j]
            x[j] = x[j + 1]
            x[j + 1] = t
```

(You are strongly advised to translate the above to code that allows you compute the runtime and then write down the precise $T(n)$. For grading purposes you just have to write down the big-O of $T(n)$.)

SOLUTION

Q7. The purpose of this question is to compute the big-O runtime of an algorithm using experimental and brute force curve-fitting.

First implement the bubblesorting algorithm. Use it collect runtimes for arrays of size $n = 2000, 4000, 6000$, etc. until you have enough points to plot a graph of the runtime. You should have at least 10 data points. (For each n , you should do about 10 experiments and then take the average.) Specifically, when you run your program, it prompts the user for n and k . It then performs bubblesort on $\mathbf{x}[0], \dots, \mathbf{x}[\mathbf{n}-1]$ k times and reports on the average time. Here's a test run:

```
10000 10
8.5
```

This gives you an experimental runtime

$$T(10000)$$

of the bubblesort.

Next, plot the data points on a graph. See next page on how to do it in L^AT_EX.

Finally, find an n^d (i.e., find d with d as small as possible) such that

$$T(n) = O(n^d)$$

You do that by finding an N and a C such that the curve of Cn^d is above $T(n)$ for all $n \geq N$, i.e., such that for $n \geq N$,

$$T(n) \leq Cn^d$$

Include the following for this question:

- (a) C++ source code.
- (b) Bash shell session with collection of runtimes.
- (c) Graph showing the plot of $T(n)$ and Cn^d . State clearly the C and the d . Also, indicate clearly N on the n -axis.

SOLUTION

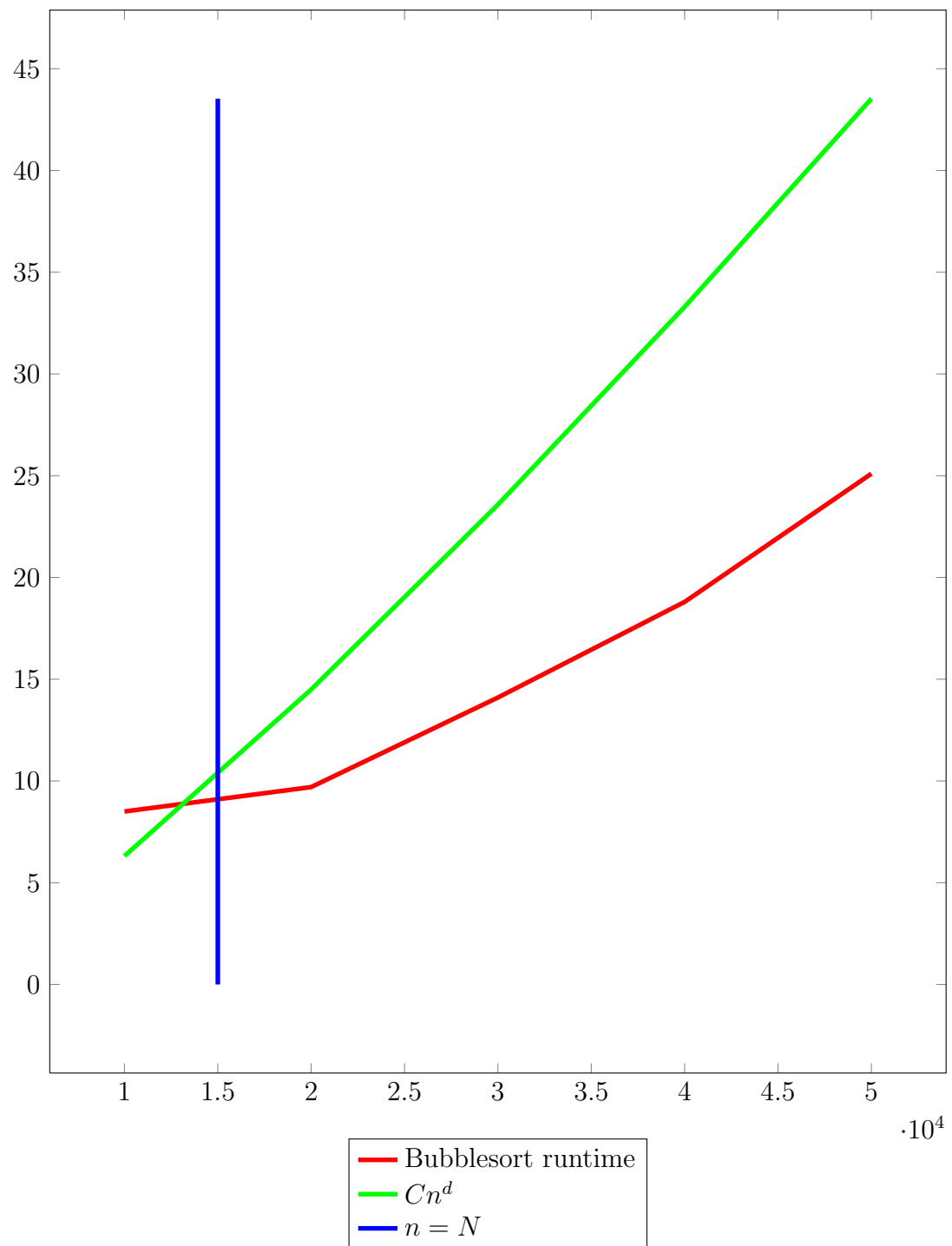
(a)

REPLACE THIS WITH YOUR C++ SOURCE FILE

(b)

```
REPLACE THIS WITH YOUR BASH SHELL SESSION WITH RUNTIMES COLLECTION
[student@localhost bubblesort]$ g++ *.cpp
[student@localhost bubblesort]$ ./a.out
10000 10
8.5
[student@localhost answer]$ ./a.out
20000 10
9.7
```

(c) In the follow change, modify C, N, d and the data in the python code for drawing the graph.



$$N = 15000, \quad C = 0.0001, \quad d = 1.2$$