Programming Projects

CSE 5211 Analysis of Algorithms

Spring 2017 (January 12, 2017)

The purpose of these projects is for you to have the opportunity to exercise your ability to:

- Reason about problems
- Write algorithms that show your problem solving skills
- Explain your work to others
- Empirically test the performance of your programs
- Compare empirical data and theoretical results

Each project requires you to deliver a report* containing

- A description of a problem
- A description of least one algorithm that solves the problem
- Mathematical analysis of the best, average, and worst case run time of the described algorithm(s)
- At least one well-commented program that implements an algorithm for the problem
- A description of the steps necessary to compile and execute your program collecting run time and space usage data
- Visualization of your collected run time data over sample inputs
- Comparison of your empirical results with theoretical formulas

* Your report must be in Portable Data Format (pdf)

You may use any computing system that you have rights to access and you may write in any programming language you choose. Whichever programming language you choose, you must be able to collect profile data from executing your code.

Individual Programming Projects

Project 1

- Write a program that implements insertion sort (on a set of integers, if your programming language does not support polymorphic data).
- Write a program that implements quicksort.
- Compile your programs so that execution-time profile data is collected.

Please be certain not to plagiarize your assignments. Moss will be used to detect software plagiarism. TurnItIn will be used to detect report plagiarism. If you turn in plagiarized work, you will receive a grade of zero on your project and be reported to the department. On a second offense you will be reported to the university.

- Determine a sequence of input sizes that exercise your code.
- For each input size generate sample sequences of random integers and feed them to your insertion sort and quick sort routines.
- For each program, compute averages of execution time and memory space usage for each input size and produce a scatter plot the results.
- Compare your experimental results with theoretical complexities. Two complexity measures for sorting are: the number of comparisons, and the number of swaps. These theoretical results are known although leading coefficients and lower-order terms may differ based on implementations.

  Look up these results in (Knuth, 1998).

  - Insertion sort
    - Comparisons:
      - Best case $n-1=\Omega(n)$
      - Average case $\frac{1}{2}\binom{n}{2}+n-H_n=O(n^2)$
      - Worst case $\binom{n}{2}=O(n^2)$
    - Swaps:
      - Best case $0=\Omega(1)$
      - Average case $\frac{1}{2}\binom{n}{2}=O(n^2)$
      - Worst case $\binom{n}{2}=O(n^2)$
  - Quicksort
    - Comparisons:
      - Best case $O(n\lg n)$
      - Average case $2(n+1)H_{n+1}=O(n\lg n)$
      - Worst case $O(\frac{n^2}{2})$
    - Swaps:
      - Best case $0=O(1)$
      - Average case $O(n\lg n)$
      - Worst case $O(\frac{n^2}{2})$

Project 2: Student's Choice

By Monday of week 8 (February 27, 2017) inform your instructor of the project you propose to complete. Rather than assign a project to graduate-level students, let me suggest some algorithms you could explore.

- Aho–Corasick Algorithm
- Boyer–Moore Algorithm
- B-Trees
- Fast Fourier Transform
- Floyd-Warshall Algorithm
- Huffman Coding

- Knuth–Morris-Pratt Algorithm
- RSA Public Key Encryption
- Skiplists
- Stable matching

If you have another project idea, check for your instructor's approval before proceeding.

Algorithmics 2017

Participation in Algorithmics 2017 is required.

This is a call for participation in Algorithmics 2017, a workshop on algorithms that runs from April 14 to 21, 2017. It is sponsored by the School of Computing at the Florida Institute of Technology.

Research groups select a problem and report on algorithms that solve it. Team size is three (if enrollment is not a multiple of three, some teams will of size two). Groups submit a research report, as described above for individual projects, and make a presentation to the class. The schedule of steps in this assignment are listed in the course calendar. Briefly:

1. Week 2: Groups formed
2. Week 4: Project proposal and acceptance
3. Week 11: Draft submission
4. Week 14-15: Presentations
5. Week 16: Final submission and evaluations

Here is an alphabetized, non-exhaustive list of problem classes from which research groups may select a specific problem to analyze.

- Computational geometry
- Cryptography
- Graph
- Machine learning
- Medical
- Numerical
- Parsing
- Search
- Sort
- String

Rubrics

A rubric is a scoring guide used to evaluate the quality of work. Rubrics can help you understand how you will be evaluated, and this can improve your work. There are three rubrics for the projects described above:

1. A rubric for individual work
2. A rubric for teamwork

3. A rubric for presentation skills

Submit all completed rubrics by the due date listed in the course calendar.

Individual Projects Rubric

The following rubric will be used evaluate student work on individual projects.

Student Name: _____

Category	Beginning 1	Developing 2	Accomplished 3	Exemplary 4	Score
Compilation & Tests	The submitted code does not compile.	The code compiles but passes too few test cases.	The code compiles but passes most test cases.	The code compiles but passes all test cases.	
Documentation	The code is has no or very little documentation.	The code has some documentation, but appears as an afterthought.	The code contains useful documentation.	The code is well documented, perhaps even in a literate style.	
Tool Usage	Little evidence that software tools were effectively used.	Some evidence of minimal tool usage.	Evidence that several software tools were used	Documented use of a wide array of tools: configuration management, build, test, debug, profile, etc.	
Individual Project Average of Scores					

Team Project Rubric

Use the following rubric to evaluate your teammates.

Teammate Name:_____ **Your Name:** _____

Category	Beginning 1	Developing 2	Accomplished 3	Exemplary 4	Score
Conflict	Participated in regular conflict that interfered with group progress. The conflict was discussed outside of the group.	Was the source of conflict within the group. The group sought assistance in resolution from the instructor.	Was minimally involved in either starting or solving conflicts.	Worked to minimize conflict and was effective at solving personal issues within the group.	
Assistance	Contributions were insignificant or nonexistent.	Contributed some toward the project.	Contributed significantly but other members clearly contributed more.	Completed an equal share of work and strove to maintain equity.	
Effectiveness	Work performed was ineffective and mostly useless toward the final project.	Work performed was incomplete and contributions were less than expected.	Work performed was useful and contributed to the final project.	Work performed was very useful and contributed significantly to the final project.	
Attitude	Rarely had a positive attitude toward the group and project.	Usually had a positive attitude toward the group and project.	Often had a positive attitude toward the group and the project.	Always had a positive attitude toward the group and the project.	
Attendance & Readiness	Rarely attended group meetings.	Sometimes attended group meetings, brought needed materials, and was ready to work.	Almost always attended group meetings, brought needed materials, and was ready to work.	Always attended group meetings, brought needed materials, and was ready to work.	
Task Focus	Rarely focused on the task and what needed to be done. Let others do the work.	Other group members sometimes had to nag, prod, and remind to keep this member on task.	Focused on the task most of the time, what needed to be done.	Consistently stayed focused on the task, what needed to be done.	
Teammate Average of Scores					

Presentation Rubric

Use the following rubric to evaluate presentations by other groups.

Presentation Title:_____ **Your Name:** _____

Category	Beginning 1	Developing 2	Accomplished 3	Exemplary 4	Score
Participation	Too few group members participate.	Some group members participate.	Most group members participate.	All group members participate equally.	
Presence	Most group members do not make eye contact and have poor body language.	Some group members do not make eye contact or have poor body language.	Most group members do make eye contact and have good body language.	All group members do make eye contact and have good body language.	
Delivery	Too few group members speak and can be understood.	Some group are difficult to understand.	Most group members speak clearly and are easy to understand.	All group members speak clearly and are easy to understand.	
Organization	Information is disorganized.	Information may be only partially organized.	Most information is presented in an organized way.	All information is presented in an organized way.	
Visual Aids	Presentation is incomplete and disorganized.	Presentation is complete but disorganized.	Presentation is organized but incomplete.	Presentation is visually organized and complete.	
Presentation Average of Scores					

Sample Report

Almost ready for publication.

I don't expect you'll write at this level. It has taken me a lifetime to learn.

The Maximum Subsequence Sum Problem (MSSP)

Consider the Dow Jones Industrial Average: It goes up and down daily. What contiguous run of days has the highest gain? Consider your weight: It goes up and down daily. What contiguous run of days has the largest weight loss? These are just two examples of sequences over which sub-sequences with the largest gain or loss could be sought. "Algorithm Design Techniques," (Bentley, 1984) and §4.1 The Maximum Sub-array Problem, in (Corman et al., 2009) describe this problem in detail. Let's design some algorithms that solve the maximum subsequence sum problem.

Problem 1: Maximum Subsequence Sum (MSSP)

Given a list of integers $A[k]$, $k = 0, \ldots, (n-1)$, $n \geq 0$, *find the maximal value of*

$$\sum_{k=s}^{e} A[k] \quad \textit{for } 0 \leq s \leq e \leq (n-1).$$

In the case where all values in A *are negative, the empty subsequence* $[]$ *gives* 0 *as the maximum subsequence sum.*

Two algorithms for MSSP are presented below: A brute force cubic time algorithm and a more sophisticated linear time functional algorithm.

Brute Force MSSP Algorithm Here is a small example to show how the brute force algorithm runs. Pretend you are given sequence of $n = 11$ integers, say

$$\langle 31, -41, 59, 26, -53, 58, 97, -93, -23, 84, -32 \rangle$$

By inspection you may notice the largest gain is 187 over the contiguous subsequence

$$\langle 59, 26, -53, 58, 97 \rangle$$

To reason about this example, notice there are 11 sums with 1 term each requiring no additions; 10 sums with 2 terms each requiring 1 addition; 9 sums with 3 terms each requiring 2 additions, ..., and there is 1 sum with 11 terms requiring 10 additions. Computing each of these sums takes

$$11 \cdot 0 + 10 \cdot 1 + 9 \cdot 2 + 8 \cdot 3 + \cdots + 1 \cdot 10 + 0 \cdot 11 = \sum (\#\text{sums})(\#\text{adds per sum}) = 220 \text{ additions.}$$

In general, for a list of length n, the number of additions is

$$\sum_{k=0}^{n} (n-k)k = \frac{(n+1)(n)(n-1)}{6} = O(n^3)$$

Prove that this summation formula is correct.

The brute-force approach computes each sum for every possible subsequence and remembers the largest. The pseudocode

8a ⟨*Cubic time maximum subsequence sum* 8a⟩≡
```
int maxSubseqSum(int A[], int n) {
  ⟨Local State 8b⟩
  ⟨For each subsequence start 8c⟩ {
    ⟨For each subseqence ending at or after start 8d⟩ {
      ⟨Set sum to A[start] 8e⟩
      ⟨Compute sum from start+1 to end 8f⟩
      ⟨Check and update maxSoFar if necessary 9a⟩
    }
  }
  ⟨Return the maximum subsequence sum 9b⟩
}
```

Each of these pseudocode stubs can be replaced by simple executable statements.

Only two local values are needed: The `sum` of the current subsequence and `maxSoFar`, the largest subsequence sum computed up to now.

8b ⟨*Local State* 8b⟩≡
```
int sum =  0;
int maxSoFar =  0;
```

The `start` of a subsequence ranges from the head of the list at index 0 to the last value at index $n-1$.

8c ⟨*For each subsequence* `start` 8c⟩≡
```
for (int start = 0; start < n; start++)
```

Likewise, the end of a subsequence ranges from `start` to index $n-1$: A subsequence end's at or before its `start`

8d ⟨*For each subseqence* `end`*ing at or after* `start` 8d⟩≡
```
for (int end = start; end < n; end++)
```

To start, the current sum can be initialized to the value of `A[start]`.

8e ⟨*Set* `sum` *to A[start]* 8e⟩≡
```
sum = A[start];
```

Now compute the sum from `start+1` to `end`. If this sum is larger than any previous sum, update the `maxSoFar` value.

8f ⟨*Compute* `sum` *from* `start+1` *to* `end` 8f⟩≡
```
for (int k = start+1; k < end; k++) {
  sum = sum + A[k];
}
```

Check if a new maximum has been found. Update `maxSoFar` if necessary.

9a ⟨*Check and update* `maxSoFar` *if necessary* 9a⟩≡
```
maxSoFar = (sum > maxSoFar) ? sum : maxSoFar;
```

Finally, after all the loops complete, return the maximum subsequence sum.

9b ⟨*Return the maximum subsequence sum* 9b⟩≡
```
return maxSoFar;
```

Let's wrap this brute force code in a `main` program so that it can be compiled and exercised it over a range of data sets.

The code first reads the length `n` of the sequence and allocates memory for it. It then fills `seq` with values. Finally, `maxSubseqSum` is called and its return value printed.

9c ⟨*C cubic time* `main` *program* 9c⟩≡
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
   int n;
   scanf("%d\n", &n);
   int* seq = (int *)malloc(n * sizeof(int));
   for (int i = 0; i < n; i++) {
      scanf("%d", &seq[i]);
   }
   printf("%d\n", maxSubseqSum(seq, n));
}
```

Brute Force MSSP Analysis

The theoretical time complexity of this brute force algorithm is $\Theta(n^3)$. This can be seen by computing the expression

$$T(n) = \sum_{s=0}^{n-1} \sum_{e=s}^{n-1} \sum_{k=s+1}^{e} c$$

which corresponds to the loop over `s=start` which encloses the loop over `e=end` which encloses the loop summing the values from `k=start+1` to `k=end`. Each iteration of this innermost loop over k takes time bounded by some constant, call it c.

The innermost sum adds c to itself $e - s - 1$ times:

$$\sum_{k=s+1}^{e} c = (e-s)c$$

Then, the middle sum from `e=start` to $n-1$ gives

$$\sum_{e=s}^{n-1} (e-s)c = c \sum_{k=0}^{n-s-1} k = c\frac{(n-s)(n-s-1)}{2} = c\binom{n-s}{2}$$

Lastly, the time complexity of the first `for` loop on `start` can be computed by

$$\begin{aligned} \sum_{s=0}^{n} c\binom{n-s}{2} = \sum_{s=0}^{n} c\binom{s}{2} &= c\binom{n+1}{3} \\ &= c\frac{(n+1)(n)(n-1)}{6} \\ &= c\left(\frac{n^3-n}{6}\right) \end{aligned}$$

Bentley's article (Bentley, 1984) describes algorithmic design improvements ultimately resulting in a linear time algorithm. In particular, for a fixed `start`, there is no reason to compute the sum each time `end` changes: Just add the new ending value to the previous sum. This eliminates the inner loop on k and reduces the time complexity to $O(n^2)$.

As another algorithm design improvement, there is no reason to compute the maximum subsequence sum each time a new value is considered in the list. Recording the maximum subsequence sum so far and the maximum to the current value allows incremental changes and results in a linear time algorithm.

A Functional Linear Time Algorithm: Functional programming offers superior expressiveness for a large classes of problems. Let's describe a functional maximum subseqence algorithm.

The algorithm takes a list `[a]` and returns the maximum subsequence sum and the subsequence that *witnesses* it. Element in the list are of type `a`: They can be ordered and added.

Using the same small example from the brute force algorithm, consider the list

$$\langle 31, -41, 59, 26, -53, 58, 97, -93, -23, 84, 32\rangle$$

Keep two ordered pairs: `(sumToHere, listToHere)` and `(maxSoFar, maxListSoFar)` Each is initiallize to `(0, [])`. When a new value is read from the list, each ordered pair may change. Here is a trace showing how the two ordered pair change as each value is read.

The trace needs a proofreader.

x : max at here	max so far
: $(0, [])$	$(0, [])$
$31 : (31, [31])$	$(31, [31])$
$-41 : (0, [])$	$(31, [31])$
$59 : (59, [59])$	$(59, [59])$
$26 : (85, [59, 26])$	$(85, [59, 26])$
$-53 : (32, [59, 26, -53])$	$(85, [59, 26])$
$58 : (90, [59, 26, -53, 58])$	$(90, [59, 26, -53, 58])$
$97 : (187, [59, 26, -53, 58, 97])$	$(187, [59, 26, -53, 58, 97])$
$-93 : (94, [59, 26, -53, 58, 97, -93])$	$(187, [59, 26, -53, 58, 97])$
$-23 : (71, [59, 26, -53, 58, 97, -93, -23])$	$(187, [59, 26, -53, 58, 97])$
$84 : (155, [59, 26, -53, 58, 97, -93, -23, 84])$	$(187, [59, 26, -53, 58, 97])$
$32 : (187, [59, 26, -53, 58, 97, -93, -23, 84, 32])$	$(187, [59, 26, -53, 58, 97])$

Let's define a function `f` that acts on two ordered pairs and a value `x`. It returns an order pair. Each ordered pair contains a value and a list. The first input pair is the value and the list of the maximum to here. The second input ordered pair is the value and the list of the maximum so far. The value `x` is the next value in the sequence. For efficiency the returned list is built in reverse order.

11 ⟨ *Functional (Haskell) Linear Time Algorithm* 11 ⟩≡

```
maxsubseq :: (Ord a, Num a) => [a] -> (a, [a])
maxsubseq = snd . foldl f ((0,[]), (0,[])) where
  f ((maxToHere, listToHere), (maxSoFar, listSoFar)) x = (here, soFar) where
    here  = max (0,[]) (maxToHere + x, x:listToHere)
    soFar = max (maxSoFar, listSoFar) here
```

Assume the input file format is the same as the input for the brute-force cubic algorithm:

- A first line representing the integer n: Haskell will read this as a string `"n"`.
- Followed by n more lines, each a string `"v"` representing a integer value v, and forming a sequence of length n.

The `main` entry point for the HASKELL program reads the contents from `stdin` into a `String` called `contents`. It maps `contents` to a newline separated list of strings, and ignores the first line, the length of the sequence, by taking the `tail` of the list. All of this is stored in a list of strings, called `sequence`, representing the numerical sequence.

The `read` function performs magic. It's type is:

```
read :: Read a => String -> a
```

which says `a` is a type that can be read, and `read` turns a string magically into a value of that type. Well, it is not magic. A static analysis can determine the type needed down the line, and `read` turn a string into that type.

12 ⟨*Haskell linear time* `main` *program* 12⟩≡

```
main = do
    contents <- getContents
    let sequence = tail (lines contents)
    print $ maxsubseq (map read sequence) %$
```

Generating Test Data

Test data can be generated statically and stored in files or it can be generated dynamically and fed directly to the maximum subsequence sum algorithm. The code below allow a user to generate a sequence of random integers of length `n` and diameter `d`. Compile the code, assume it is in file `random.c`, to create an executable `random`.

Working all in one directory gets messy. I should establish a directory structure for code, data, reports, etc.

The random values lie in the range $[-d/2, d/2)$.

```
gcc random.c -o random
```

Run the code like this

```
./random > data-n-d-v.txt
```

where the `n`, `d`, and `v` describe the length, diameter, and version of the sample data.

Some code has been commented out, but it was useful in development. The `rand()` function is sufficient to exercise the algorithms.

13 ⟨*C random sequence generator* 13⟩≡

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

int main(int argc, char **argv)
{
  int n, d;
  /* printf("Enter sequence length: "); */
  scanf("%d", &n);
  /* printf("Enter sequence diameter: "); */
  scanf("%d", &d);
  printf ("%d\n", n);
  /*  printf ("%d\n", d); */
  srand (time(NULL));  /* seed the sequence */
  for (int i = 0; i < n; i++) {
     printf ("%d\n", -d/2 + rand() % d);
  }
  return 0;
}
```

I have an interest in learning to use the functional programming paradigm in problem solving. Here is code I've found online that generates and prints a list of random integers. Use `randomR (min, max)` in place of `random` to bound the numbers.

14 ⟨*Haskell random sequence generator* 14⟩≡

```
import System.Random
import Data.List

main = do
    length <- getLine
    seed  <- newStdGen
    let rs = randomlist (read length) seed
    print $ rs  %$

randomlist :: Int -> StdGen -> [Int]
randomlist n = take n . unfoldr (Just . random)
```

Compiling & Profiling

To exercise these algorithms they must be wrapped in `main` programs, compiled with appropriate options, and executed to gather run time profiling data. I used a simple `Makefile` to help simplify the build of these codes.

I did test both algorithms on small datasets where results could be checked by hand. Both algorithms appear to work correctly, but on a *real world* project much more testing must be done.

GCC: The Gnu C Compiler

The C code was compiled and execute to collect time and space profile data. The documentation for profiling with `gcc` can be found at binutils To summarize:

If I were doing this for real, I'd write scripts that automate most of the process in a few keystrokes.

1. Compile the code for profiling with the `-pg` option. Include the `-Wall` options to turn on warnings, and `-o` to name the executable.

   ```
   gcc -Wall -pg msss.c -o cmsss
   ```

2. Execute the program with run to generate `gmon.out` a file containing profile data.

   ```
   ./cmsss < datafile.txt
   ```

It seems `gprof` is not supported on Mac OS X. Other tools such as `gcov` and `lcov` could to be explored. Or, `gcc` could be replaced with `Xcode`. Or, Another operating system could be used. Using Occam's razor, I have access to Linux system and can run my code there.

3. Run the `gprof` too using the executable and profile files

   gprof cmsss gmon.out > analysis.txt

Here is a truncated profile generated by running `cmsss` on a dataset of 1000 values.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.99      0.80     0.80        1   797.82   797.82  maxSubseqSum
```

GHC: The Glasgow Haskell Compiler

The Haskell code was compiled and executed to collect time and space profile data. The (The GHC Team, 2014) describes the steps to complete these tasks. To summarize:

1. Compile the code for profiling with the -prof option. Include the the -fprof-auto option for adding automatic annotations. And, use the -rtsops option to support passing of run time flags when executing the program from command line.

   ```
   ghc -prof -auto-prof -rtsops msss.hs
   ```

2. Execute the program with run time options to generate a file, `msss.prof` containing profile data.

```
./hmsss +RTS -p < datafile.txt
```

There are many kinds of profiles that can be generated. See the documentation (The GHC Team, 2014).

Here is an truncated profile generated by running `hmsss` on a dataset of 1000 values. It shows the `maxsumseq` function took 11.1% of 0.01 seconds: Too little data to exercise the code. The largest percentage of time was spend in the `main` function: `IO` and preprocessing the sequence for input to `maxsubseq`. The book (O'Sullivan et al., 2008) gives a nice description of how to read the information in the profile.

```
        Tue Jan 10 13:23 2017 Time and Allocation Profiling Report  (Final)

           hmsss +RTS -p -RTS

        total time  =        0.01 secs   (9 ticks @ 1000 us, 1 processor)
        total alloc =  10,111,232 bytes  (excludes profiling overheads)

COST CENTRE      MODULE           %time %alloc

main             Main              66.7   82.4
CAF              GHC.IO.Handle.FD  11.1    0.5
main.sequence    Main              11.1    5.6
maxsubseq.f      Main              11.1    9.3
maxsubseq.f.here Main               0.0    1.5


                                                 individual     inherited
COST CENTRE            MODULE                  no.     entries  %time %alloc   %time %alloc

MAIN                   MAIN                     49           0    0.0    0.0   100.0  100.0
 CAF                   Main                     97           0    0.0    0.0    88.9   99.4
  main                 Main                     98           1   66.7   82.4    88.9   99.4
   main.sequence       Main                    100           1   11.1    5.6    11.1    5.6
   maxsubseq           Main                     99           1    0.0    0.4    11.1   11.4
    maxsubseq.f        Main                    101        1000   11.1    9.3    11.1   11.0
     maxsubseq.f.soFar Main                    103        1000    0.0    0.2     0.0    0.2
     maxsubseq.f.here  Main                    102        1000    0.0    1.5     0.0    1.5
```

Visualization of Run time Data

This part of the project can be time consuming, but it is important. Theory and practice may not always agree and that leads *science* in computing.

Generate and plot data. Overlay with theoretical results. Analyze variations between empirical and theoretical data.

References

Bentley, J. (1984). Programming pearls: Algorithm design techniques. *Commun. ACM*, 27(9):865–873. [page 7], [page 10]

Corman, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, third edition. [page 7]

Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA. [page 2]

O'Sullivan, B., Stewart, D., and Goerzen, J. (2008). *Real World Haskell*. O'Reilly Media. [page 16]

The GHC Team (2014). *The Glorious Glasgow Haskell Compilation System User's Guide*. [page 15], [page 16]