

Measuring Runtime Performance

Data Structures & Algorithms

Complexity Notation

- n = input size
- $f(n)$ = max number of steps when input has size n
- $O(f(n))$ = asymptotic upper bound

```
1 void f(int *out, const int *in, int size) {  
2     int product = 1;  
3     for (int i = 0; i < size; ++i)  
4         product *= in[i];  
5     for(int i = 0; i < size; ++i)  
6         out[i] = product / in[i];  
7 } // f()
```

$$f(n) = 1 + 1 + 1 + 3n + 1 + 1 + 3n = 6n + 5 = O(n)$$

Ways to measure complexity

- Analytically
 - Analysis of the code itself
 - Recognizing common algorithms/patterns
 - Based on a recurrence relation
- Empirically
 - Measure runtime programmatically
 - Measure runtime using external tools
 - Test on inputs of a variety of sizes

Measuring Runtime Programmatically

- "Programmatically" – measurements are taken from inside the code itself
- Varies greatly depending on the language
- Many different ways to do it even just in C/C++!

Measuring Time In C++11+

```
1  #include <chrono>
2
3  class Timer {
4      std::chrono::time_point<std::chrono::system_clock> cur;
5      std::chrono::duration<double> elap;
6  public:
7      Timer() : cur(), elap(std::chrono::duration<double>::zero()) {}
8
9      void start() {
10         cur = std::chrono::system_clock::now();
11     } // start()
12
13     void stop() {
14         elap += std::chrono::system_clock::now() - cur;
15     } // stop()
16
17     void reset() {
18         elap = std::chrono::duration<double>::zero();
19     } // reset()
20
21     double seconds() {
22         return elap.count();
23     } // seconds()
24 }; // Timer{}
```

Note: Checking time too often will slow down your program!

Example

```
25 int main() {
26     Timer t;
27     t.start();
28     doStuff1();
29     t.stop();
30     cout << "1: " << t.seconds()
31         << "s" << endl;
32
33     t.reset();
34     t.start();
35     doStuff2();
36     t.stop();
37     cout << "2: " << t.seconds()
38         << "s" << endl;
39     return 0;
40 } // main()
```

Let's try it!

Save the file to a folder you can access from a *NIX shell, and/or upload to CAEN

- Browser Download

<https://eecs281staff.github.io/search-demo/search.cpp>

- Command Line Download

```
$ mkdir search-demo && cd search-demo
```

```
$ wget https://eecs281staff.github.io/search-demo/search.cpp
```

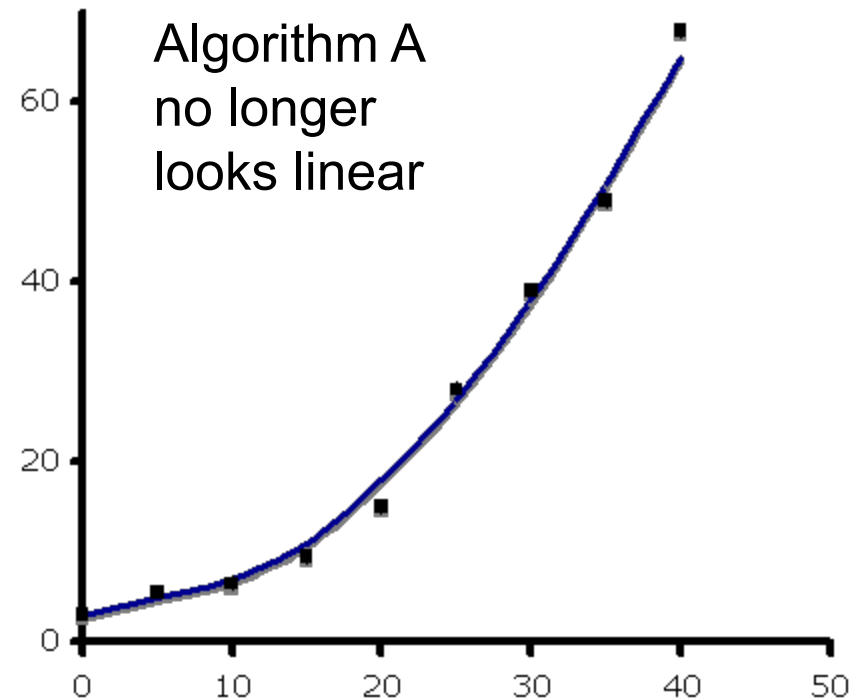
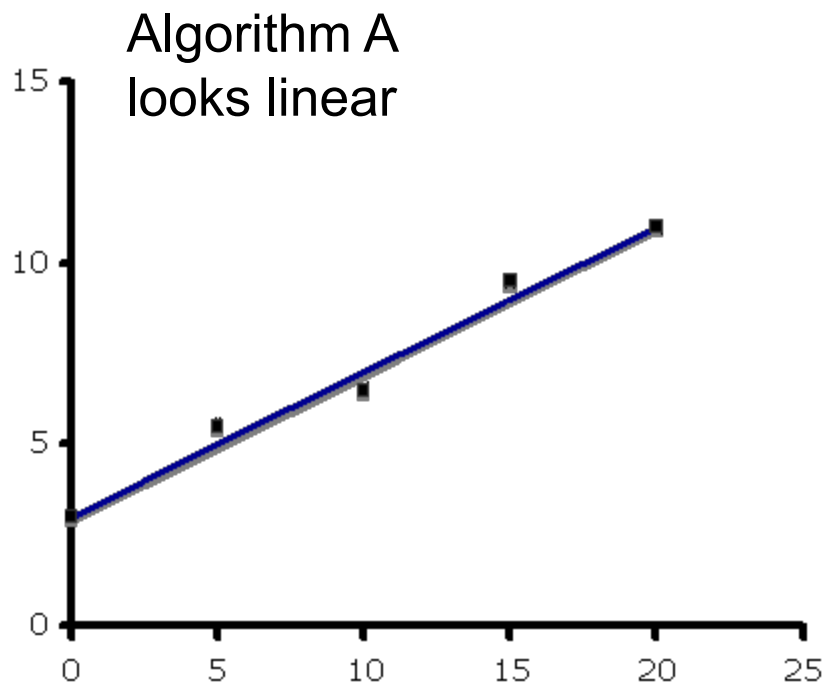


After Downloading

- Compile:
`$ g++ -std=c++17 -O3 search.cpp -o search`
- Run a binary search, 1M items:
`$./search b 1000000`
- Run a linear search, 1M items:
`$./search l 1000000`
- Try with larger numbers!

Empirical Results

- Plot actual run time versus varying input sizes
- Include a large range to accurately display trend



- Caveat: for small inputs, asymptotics may not play out yet

Prediction versus Experiment

- What if experimental results are *worse* than predictions?
 - Example: results are exponential when analysis is linear
 - Error in complexity analysis
 - Error in coding (check for extra loops, unintended operations, etc.)
- What if experimental results are *better* than predictions?
 - Example: results are linear when analysis is exponential
 - Experiment may not have fit worst case scenario
 - Error in complexity analysis
 - Error in analytical measurements
 - Incomplete algorithm implementation
 - Algorithm implemented is better than the one analyzed
- What if experimental data match asymptotic prediction but runs are too slow?
 - Performance bug?
 - Check compile options (e.g. use `-O3`)
 - Look for optimizations to improve the constant factors

Measuring Runtime Performance

Data Structures & Algorithms

Runtime Analysis Tools

Data Structures & Algorithms

Using a Profiling Tool

- This won't tell you the complexity of an algorithm, but it tells you where your program spends its time.
- Many different tools exist – you'll learn to use `perf` in lab.

Samples: 268 of event 'cpu-clock:uH', Event count (approx.): 67000000				
Children	Self	Command	Shared Object	Symbol
+ 84.70%	0.00%	processing_publ	processing_public_tests.exe	[.] test_all
+ 84.70%	0.00%	processing_publ	processing_public_tests.exe	[.] main
+ 84.70%	0.00%	processing_publ	libc-2.17.so	[.] __libc_start_main
+ 70.15%	0.00%	processing_publ	processing_public_tests.exe	[.] test_seam_carve
+ 69.78%	0.00%	processing_publ	processing_public_tests.exe	[.] seam_carve
+ 66.04%	0.00%	processing_publ	processing_public_tests.exe	[.] seam_carve_width
+ 42.16%	42.16%	processing_publ	libc-2.17.so	[.] __memcpy_ssse3_back
+ 20.90%	0.00%	processing_publ	processing_public_tests.exe	[.] seam_carve_height
+ 19.03%	12.31%	processing_publ	processing_public_tests.exe	[.] Image_get_pixel
+ 14.55%	1.12%	processing_publ	processing_public_tests.exe	[.] compute_energy_matrix
+ 11.57%	0.75%	processing_publ	processing_public_tests.exe	[.] compute_vertical_cost_matrix
+ 9.70%	9.70%	processing_publ	processing_public_tests.exe	[.] Matrix_at
+ 8.58%	3.36%	processing_publ	processing_public_tests.exe	[.] Matrix_column_of_min_value_in_row
+ 8.21%	0.75%	processing_publ	processing_public_tests.exe	[.] Matrix_min_value_in_row
+ 7.84%	0.00%	processing_publ	processing_public_tests.exe	[.] remove_vertical_seam
+ 5.60%	0.00%	processing_publ	libstdc++.so.6.0.21	[.] 0xffff80eca0abb340
+ 5.60%	0.00%	processing_publ	libstdc++.so.6.0.21	[.] std::basic_ofstream<char, std::char_traits<char>>::write<char, int> (this, const char*, int)
+ 5.60%	0.00%	processing_publ	[unknown]	[.] 0x48087f8d48fb8948
+ 4.48%	0.00%	processing_publ	processing_public_tests.exe	[.] test_rotate
+ 4.10%	4.10%	processing_publ	libstdc++.so.6.0.21	[.] std::num_get<char, std::istreambuf_iterator<char, std::char_traits<char>>> (this, std::istreambuf_iterator<char, std::char_traits<char>>, std::istreambuf_iterator<char, std::char_traits<char>>, int)
+ 4.10%	4.10%	processing_publ	processing_public_tests.exe	[.] Matrix_at

ip: Treat branches as callchains: perf report --branch-history

A snapshot of a `perf` report generated for EECS 280 Project 2.

Image credit: Alexandra Brown

Measuring Runtime on Linux

If you are launching a program using command

```
% progName -options args
```

Then

```
% /usr/bin/time progName -options args
```

will produce a runtime report

```
0.84user 0.00system 0:00.85elapsed 99%CPU
```

If you're timing a program in the current folder, use `./`

```
% /usr/bin/time ./progName -options args
```

Often, you can just type `time` rather than `/usr/bin/time`.

Measuring Runtime on Linux

- Example: this command just wastes time by copying zeros to the "bit bucket"

```
% time dd if=/dev/zero of=/dev/null
```

kill it with control-C

```
3151764+0 records in
```

```
3151764+0 records out
```

```
1613703168 bytes (1.6 GB) copied, 0.925958 s, 1.7 GB/s
```

```
Command terminated by signal 2
```

```
0.26user 0.65system 0:00.92elapsed 99%CPU
```

```
(0avgtext+0avgdata 3712maxresident)k
```

```
0inputs+0outputs (0major+285minor)pagefaults 0swaps
```

Measuring Runtime on Linux

`0.26user 0.65system 0:00.92elapsed 99%CPU`

- `user` time is spent by your program
- `system` time is spent by the OS on behalf of your program
- `elapsed` is wall clock time - time from start to finish of the call, including any time slices used by other processes
- `%CPU` Percentage of the CPU that this job got. This is just $(\text{user} + \text{system}) / \text{elapsed}$
- `man time` for more information

Using valgrind

- Suppose we want to check for memory leaks:
`valgrind ./search b 1000000`
- Force a leak!
 - Replace `return 0` with `exit(0)`, run valgrind using flags `--leak-check=full --show-leak-kinds=all`
- Who leaked that memory?
 - The memory address isn't very useful, we just know that `main()` called operator `new`
 - Recompile with `-g3` instead of `-O3` and run valgrind one more time

Runtime Analysis Tools

Data Structures & Algorithms

Analyzing Recursion

Data Structures & Algorithms

Job Interview Question

- Implement this function

// returns x^n

```
int power(int x, uint32_t n);
```

- The obvious solution using $n - 1$ multiplications is **$O(n)$**
 - $2^8 = 2 * 2 * \dots * 2$
- Less obvious: **$O(\log n)$** multiplications
 - Hint: $2^8 = ((2^2)^2)^2$
 - How does it work for 2^7 ?
- Write both solutions iteratively and recursively

Computing x^n

```
1  int power(int x, uint32_t n) {
2      if (n == 0) {
3          return 1;
4      } // if
5
6      int result = x;
7      for (int i = 1; i < n; ++i) {
8          result = result * x;
9      } // for
10
11     return result;
12 } // power()
```

Analyzing Solutions

- *Iterative* functions use loops
- A function is *recursive* if it calls itself
- What is the time complexity of each function?

Iterative

```
1  int power(int x, int n) {  
2      int result = 1;  
3      for (int i = 0; i < n; ++i) {  
4          result = result * x;  
5      } // for()  
6  
7      return result;  
8  } // power()
```

$\Theta(n)$

It's just a regular loop.

Recursive

```
9  int power(int x, int n) {  
10     if (n == 0) {  
11         return 1;  
12     } // if()  
13     return x * power(x, n - 1);  
14 } // power()
```

???

We need another
tool to analyze this.

Recurrence Relations

- A *recurrence relation* describes the way a problem depends on a subproblem.

- A recurrence can be written for a computation:

$$x^n = \begin{cases} 1 & n == 0 \\ x * x^{n-1} & n > 0 \end{cases}$$

- A recurrence can be written for the time taken:

$$T(n) = \begin{cases} c_0 & n == 0 \\ T(n-1) + c_1 & n > 0 \end{cases}$$

- A recurrence can be written for the amount of memory used*:

$$M(n) = \begin{cases} c_0 & n == 0 \\ M(n-1) + c_1 & n > 0 \end{cases}$$

*Non-tail recursive

Solving Recurrences

- Substitution method
 1. Write out $T(n)$, $T(n - 1)$, $T(n - 2)$
 2. Substitute $T(n - 1)$, $T(n - 2)$ into $T(n)$
 3. Look for a pattern
 4. Use a summation formula
- Another way to solve recurrence equations is the Master Method (AKA Master Theorem)

Solving Recurrences: Linear

$$T(n) = \begin{cases} c_0 & n == 0 \\ T(n-1) + c_1 & n > 0 \end{cases}$$

```
1  int power(int x, int n) {  
2      if (n == 0)  
3          return 1;  
4  
5      return x * power(x, n - 1);  
6  } // power()
```

Recurrence: $T(n) = T(n-1) + c$
Complexity: $\Theta(n)$

Solving Recurrences: Logarithmic

$$T(n) = \begin{cases} c_0 & n == 0 \\ T\left(\frac{n}{2}\right) + c_1 & n > 0 \end{cases}$$

```
1  int power(int x, int n) {  
2      if (n == 0)  
3          return 1;  
4  
5      int result = power(x, n / 2);  
6      result *= result;  
7      if (n % 2 != 0) // n is odd  
8          result *= x;  
9  
10     return result;  
11 } // power()
```

Recurrence: $T(n) = T(n / 2) + c$
Complexity: $\Theta(\log n)$

A Logarithmic Recurrence Relation

$$T(n) = \begin{cases} c_0 & n == 0 \\ T\left(\frac{n}{2}\right) + c_1 & n > 0 \end{cases} \rightarrow \mathbf{\Theta(\log n)}$$

- Fits the logarithmic recursive implementation of `power()`
 - The power to be calculated is divided into two halves and combined with a single multiplication
- Also fits Binary Search
 - The search space is cut in half each time, and the function recurses into only one half

Recurrence Thought Exercises

- What if a recurrence cuts a problem into two subproblems, and both subproblems were recursively processed?
- What if a recurrence cuts a problem into three subproblems and...
 - Processes one piece
 - Processes two pieces
 - Processes three pieces
- What if there was additional, non-constant work after the recursion?

Binomial Coefficient

$$\binom{n}{k} = \frac{n!}{k! (n - k)!}$$

- Binomial Coefficient – “ n choose k ”
- Write this function with pen and paper
- Compile and test what you’ve written
- Options
 - Iterative
 - Recursive
 - Tail recursive
- Analyze

Analyzing Recursion

Data Structures & Algorithms