

CMPS 2200

Introduction to Algorithms

Lecture 1: Overview

Today's agenda:

- Introductions
- Motivation for course
- Formalisms used throughout the course
- Navigating the course

What is an algorithm?

*an explicit, precise, unambiguous, mechanically-executable
sequence of elementary instructions, usually intended to
accomplish a specific purpose.*

-- Jeff Erickson (<https://jeffe.cs.illinois.edu/teaching/algorithms/>).

Examples?

BOB(n):

- for i n down to 1
 - Sing “ i bottles of beer on the wall, i bottles of beer,”
 - Sing “Take one down, pass it around, $i - 1$ bottles of beer on the wall.”
- Sing “No bottles of beer on the wall, no bottles of beer,”
- Sing “Go to the store, buy some more, n bottles of beer on the wall.”

Examples of algorithm-like things that are not algorithms?

BeAMillionaireAndNeverPayTaxes():

- Get a million dollars.
- If the tax man comes to your door and says, “You have never paid taxes!”
 - Say “I forgot.”

What makes a good algorithm?

- correct
- user-friendly
- many features
- robust
- simple
- secure
- low programmer cost
- **efficient**
 - runs quickly
 - requires little memory

Then, why study efficiency?

- separates feasible from infeasible
- correlates with user-friendliness

What if it took Google took 2 minutes to return results?

Simple warmup: **What does this do?**

```
In [2]: def my_function(a, b):  
        for i,v in enumerate(a):  
            if v == b:  
                return i  
        return -1
```



```
In [3]: def linear_search(mylist, key):  
        """  
        Args:  
        mylist...a list  
        key.....a search key  
        Returns:  
        index of key in mylist; -1 if not present  
        """  
        for i,v in enumerate(mylist):  
            if v == key:  
                return i  
        return -1  
  
linear_search([5,1,10,7,12,4,2], 12)
```

Out[3]: 4

What factors affect the running time of this algorithm?

- Input **size**
- Input **values**: is key at start or end?
- Hardware!
 - TI-85 vs. Supercomputer

We need a way to compare the efficiency of algorithms that abstracts away details of hardware and input.

Analysis of Linear Search, the long way

- Assign a time cost c_i to each line i .
- Figure out how often each line is run n_i
- total cost is the cost of each line multiplied by the number of times it is run

$$\text{Cost}(\text{linear-search}, \text{mylist}, \text{key}) = \sum_i c_i * n_i$$

```
In [4]: def linear_search(mylist, key):           # cost      number of times run
        for i,v in enumerate(mylist):          # c1         ?
            if v == key:                        # c2         ?
                return i                       # c3         ?
        return -1                             # c4         ?
```

Best/Average/Worst case

To deal with the effects of the input values on performance, we can consider three types of analysis:

- **Worst-case:** maximum time for any input of size n

```
linear_search([5,1,10,7,12,4,2], 9999)
```

- **Best case:** minimum time of any input of size n

```
linear_search([5,1,10,7,12,4,2], 5)
```

- **Average case:** expected time over all inputs of size n
 - Need some probability distribution over inputs

```
for (mylist, key) in ???:  
    linear_search(mylist, key)
```

Worst-case analysis of linear search

Assume $n \leftarrow \text{len}(\text{mylist})$

```
In [5]: def linear_search(mylist, key):           # cost           number of times run
        for i,v in enumerate(mylist):           # c1              ?
            if v == key:                         # c2              ?
                return i                         # c3              ?
        return -1                               # c4              ?
```

$$\text{Cost}(\text{linear-search}, n) = c_1 n + c_2 n + c_4$$

Cost is now just a function of:

- input size n
- constants c (depend on machine, compiler, etc)

How granular should we get?

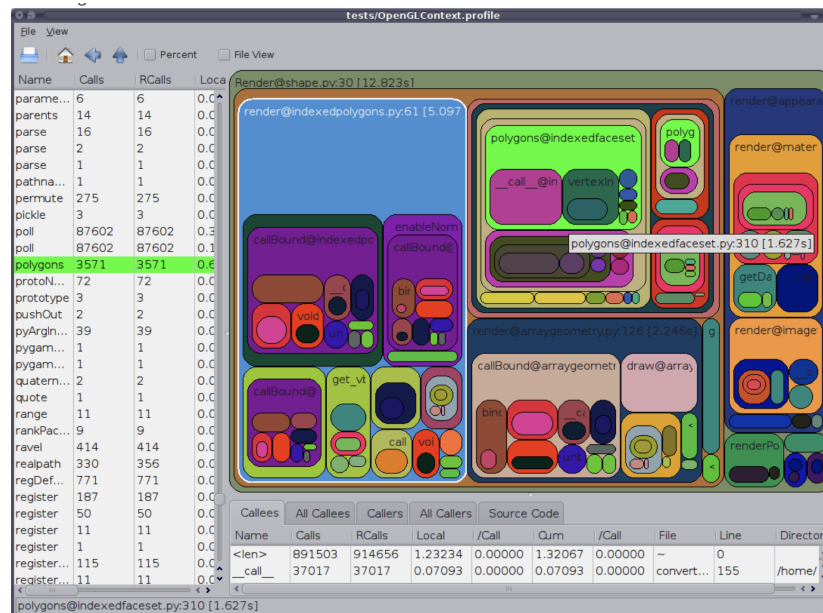
Consider this slightly different implementation:

```
In [6]: def new_linear_search(mylist, key):      # cost          number of times run
        for i in range(len(mylist)):          # c5              n
            if mylist[i] == key:                # c6              n
                return i                        # c3              0
        return -1                              # c4              1
```

$$\text{Cost}(\text{new-linear-search}, n) = c_5 n + c_6 n + c_4$$

$$\text{Cost}(\text{linear-search}, n) = c_1 n + c_2 n + c_4$$

There is a line between analyzing algorithms and optimizing a specific implementation (e.g., through profiling):



Big Idea: Asymptotic Analysis

- Ignore machine-dependent constants
- Focus on **growth** of running time
 - What happens in the limit as $n \rightarrow \infty$

$$c_1 n + c_2 n + c_4 \approx c_5 n + c_6 n + c_4$$

e.g., consider two algorithms with running times:

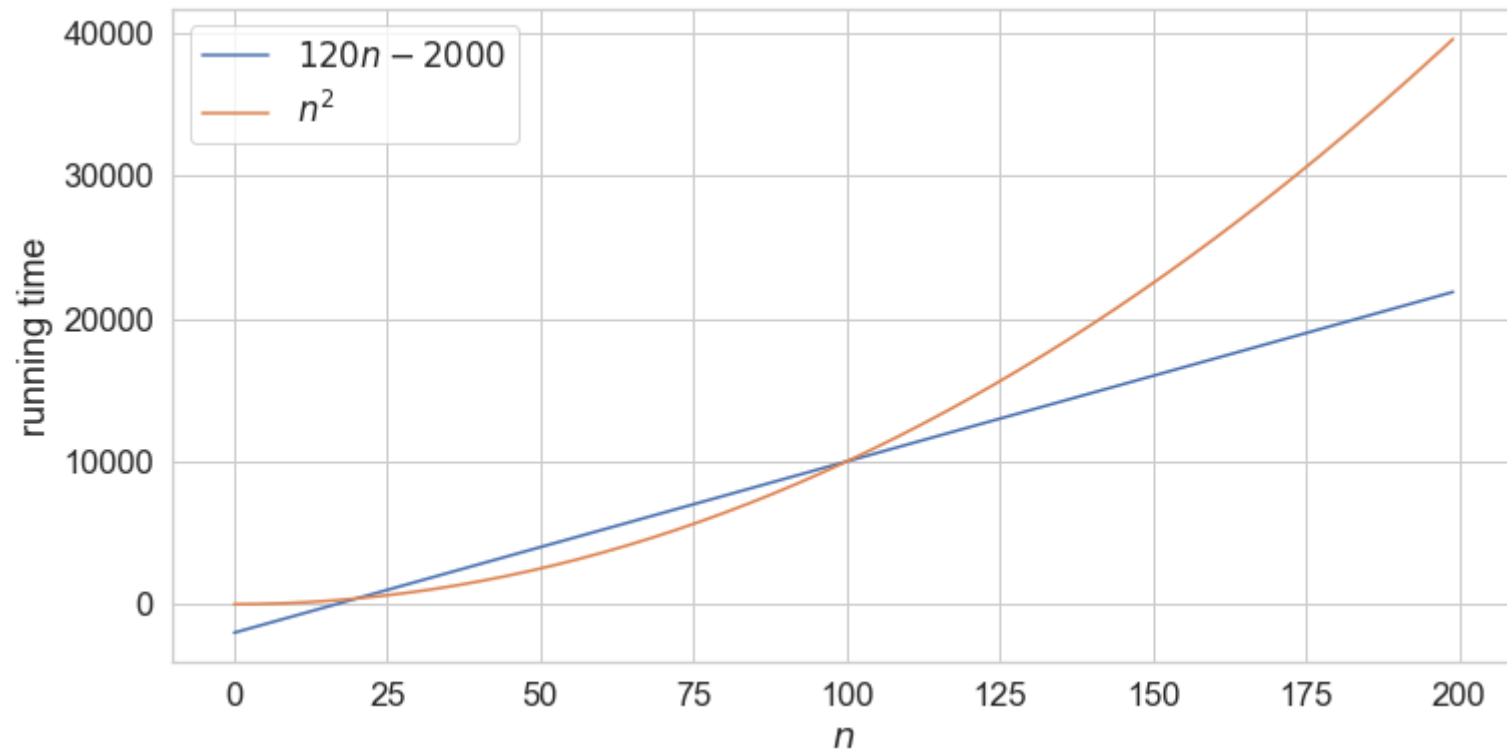
- algorithm 1: $c_1 n + c_2$
- algorithm 2: $c_3 n^2 + c_4 n + c_5$

Depending on the machine-dependent constants, algorithm 2 may sometimes be faster than algorithm 1:

- algorithm 1: $120n - 2000$
- algorithm 2: n^2


```
In [7]: n = np.arange(200)
time1 = 120*n + - 2000
time2 = n*n

# plot
plt.figure()
plt.plot(n, time1, label='$120 n - 2000$')
plt.plot(n, time2, label='$n^2$')
plt.xlabel('$n$')
plt.ylabel('running time')
plt.legend()
plt.show()
```



Definition: Asymptotic dominance

Function $f(n)$ **asymptotically dominates** function $g(n)$ if **there exist** constants c and n_0 such that

$$g(n) \leq c \cdot f(n) \text{ for all } n \geq n_0$$

e.g., n^2 asymptotically dominates $120n - 2000$

Proof:

Find c and n_0 such that

$$120n - 2000 \leq c * n^2 \text{ for all } n > n_0$$

Let $c = 1$. Find an n_0 such that

$$120n - 2000 \leq n^2 \text{ for all } n \geq n_0$$

$$120n - 2000 \leq n^2$$

$$0 \leq n^2 - 120n + 2000$$

$$0 \leq (n - 100)(n - 20)$$

$$\text{When } n = 100, 120n - 2000 = n^2$$

$$\text{For all } n \geq 100, 120n - 2000 \leq n^2$$

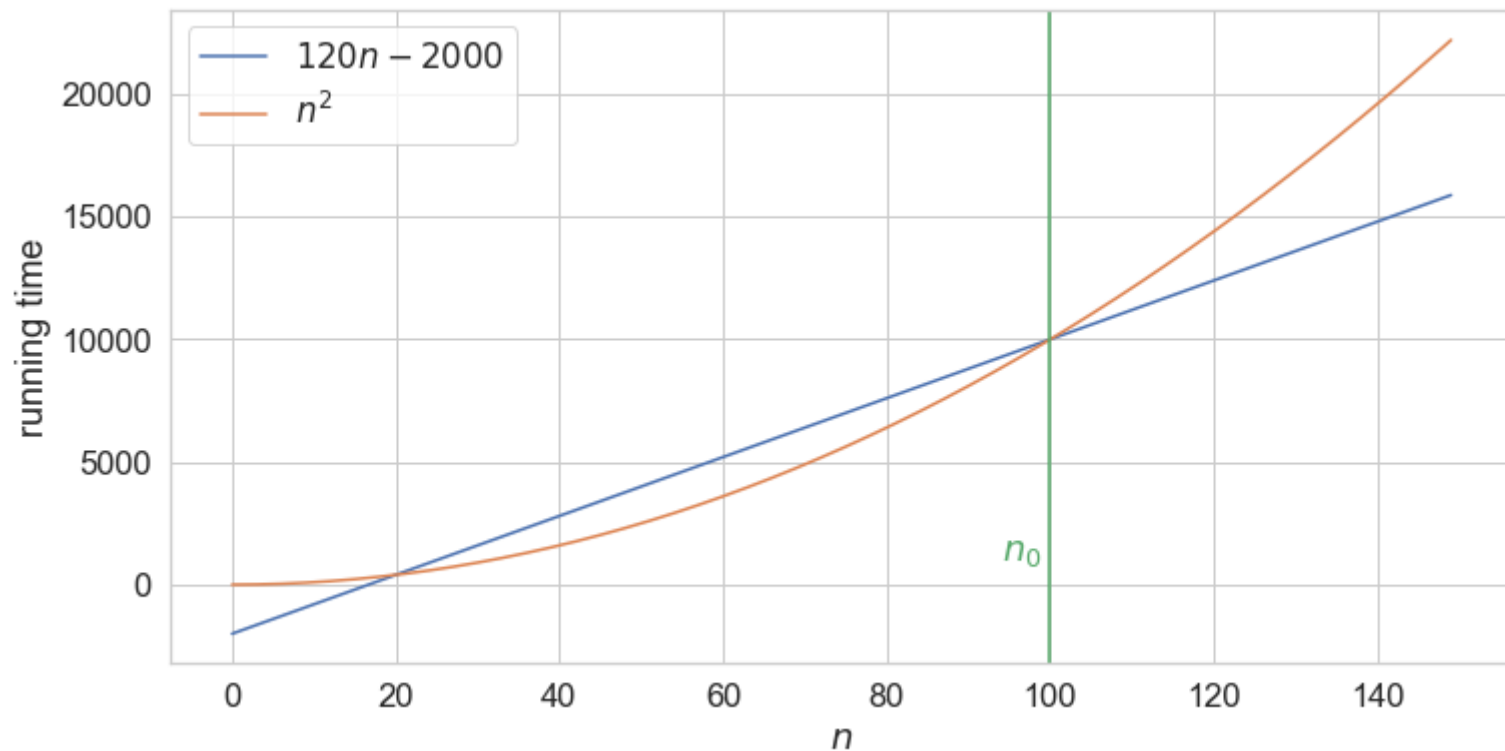
So, $c = 1$, $n_0 = 100$ satisfies the definition of asymptotic dominance.

```

In [8]: # show  $n_0$ 
n = np.arange(150)
time1 = 120*n + - 2000
time2 = n*n

# plot
plt.figure()
plt.plot(n, time1, label='$120 n - 2000$')
plt.plot(n, time2, label='$n^2$')
plt.axvline(100, color='g')
plt.text(94,1000,'$n_0$', fontsize=18, color='g')
plt.xlabel("$n$")
plt.ylabel('running time')
plt.legend()
plt.show()

```



Asymptotic Notation

$$\mathcal{O}(f(n)) = \{g(n) \mid f(n) \text{ asymptotically dominates } g(n)\}$$

$$\Omega(f(n)) = \{g(n) \mid g(n) \text{ asymptotically dominates } f(n)\}$$

$$\Theta(f(n)) = \mathcal{O}(f(n)) \cap \Omega(f(n))$$

e.g.

$$120n - 2000 \in \mathcal{O}(n^2)$$

$$10n^3 + 2n^2 - 100 \in \Omega(n^2)$$

$$14n^2 - 5n + 50 \in \Theta(n^2)$$

We often abuse notation such as

$$120n - 2000 = \mathcal{O}(n^2)$$

or

$$120n - 2000 \text{ is } \mathcal{O}(n^2)$$

Analogy:

| \square | Ω | Θ | \mathfrak{o} | ω |
|-----------|----------|----------|----------------|----------|
| \leq | \geq | $=$ | $<$ | $>$ |

Course Overview

- Analyzing algorithms: methods to compute tight bounds on running time
- Designing algorithms: various approaches to designing efficient algorithms
 - lists, sequences, trees, graphs,...
- Distinct from typical courses like this, we will emphasize **parallel** algorithms from the start (next lecture)

Navigating the course

- Canvas: syllabus, dates, grades
- Diderot: interactive textbook
- Github: assignments, slides