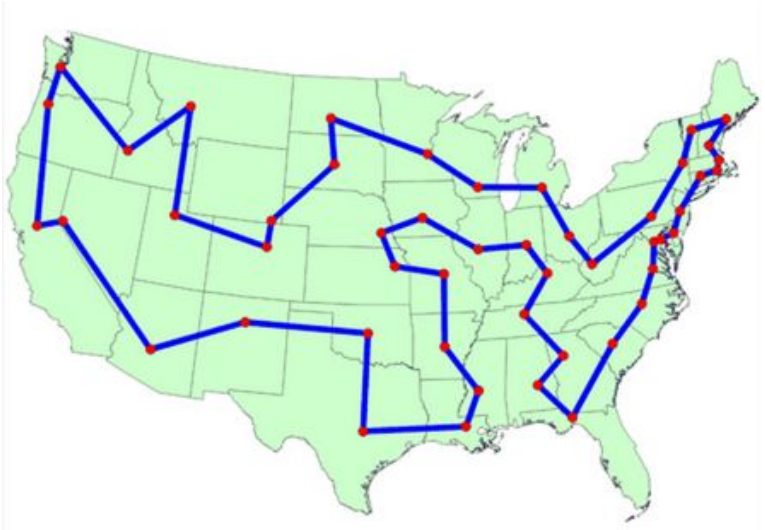


Lecture 11

Introduction to Asymptotic Analysis

~~CS~~ Math 61B, Spring 2025 @ UC Berkeley

Review Lecture Topics

- Class Inheritance, Interfaces
- Comparators vs Comparables
- Golden Rule of Equals
- Linked Lists
- Projects
- Sentinels
- Generics
- Static Classes/Methods
- Compiler vs Runtime Errors
- DLists

Goal: Measuring Code Efficiency

Lecture 11, CS61B, Fall 2024

Goal: Measuring Code Efficiency

Intuitive Runtime Metrics

- Clock Time
- Exact Operation Counting
- Exact Count Exercise

Asymptotic Analysis

- Why Scaling Matters
- Computing Worst Case Order of Growth (Tedious Approach)
- Computing Worst Case Order of Growth (Simplified Approach)

Asymptotic Notation

- Big Theta (a.k.a. Order of Growth)
- Big O and Big Omega

An engineer will do for a dime what any fool will do for a dollar.

Efficiency comes in two flavors:

- Programming cost (course to date. Will also revisit later).
  - How long does it take to develop your programs?
  - How easy is it to read, modify, and maintain your code?
    - More important than you might think!
    - Majority of cost is in maintenance, not development!
- Execution cost (from today until end of course).
  - How much time does your program take to execute?
  - How much memory does your program require?

Example of Algorithm Cost

Objective: Determine if a sorted array contains any duplicates.

- Given sorted array A, are there indices i != j where A[i] == A[j]?

-3	-1	2	4	4	8	10	12

Example of Algorithm Cost

Objective: Determine if a sorted array contains any duplicates.

- Given sorted array A, are there indices i != j where A[i] == A[j]?

-3	-1	2	4	4	8	10	12

Silly algorithm: Consider every possible pair, returning true if any match.

- Are (-3, -1) the same? Are (-3, 2) the same? ...

Better algorithm?

Example of Algorithm Cost

Objective: Determine if a sorted array contains any duplicates.

- Given sorted array A, are there indices i != j where A[i] == A[j]?

-3	-1	2	4	4	8	10	12

Silly algorithm: Consider every possible pair, returning true if any match.

- Are (-3, -1) the same? Are (-3, 2) the same? ...

Today's goal: Introduce formal technique for comparing algorithmic efficiency.

Better algorithm?

- For each number A[i], look at A[i+1], and return true the first time you see a match. If you run out of items, return false.

Measuring a Program's Performance

Expressing a program's efficiency boils down to two main steps:

1. Define some **metric** by which the program's efficiency will be evaluated, and measure the program's efficiency according to that metric.
    - Example metrics
        - Number of seconds it takes to run the program on a given input
        - Number of bytes of memory used by the program
        - Average error for randomized/approximation algorithms
2. This gives us a (mathematical) function that we can then **classify**.
    - Classification must be:
        - Simple
        - Mathematically rigorous
        - Model the program's efficiency "well enough"

Note that for step 2, we need a function in the mathematical sense (not a series of instructions that gets "run" by a computer, but a map that associates inputs with specific outputs)

Clock Time

Lecture 11, CS61B, Fall 2024

Goal: Measuring Code Efficiency

Intuitive Runtime Metrics

- **Clock Time**
- Exact Operation Counting
- Exact Count Exercise

Asymptotic Analysis

- Why Scaling Matters
- Computing Worst Case Order of Growth (Tedious Approach)
- Computing Worst Case Order of Growth (Simplified Approach)

Asymptotic Notation

- Big Theta (a.k.a. Order of Growth)
- Big O and Big Omega

What do we mean when we say that dup2 is better?

Intuitively, why is dup2 better than dup 1?

```
public static boolean dup1(int[] A) {
  for (int i = 0; i < A.length; i += 1) {
    for (int j = i + 1; j < A.length; j += 1) {
      if (A[i] == A[j]) {
         return true;
      }
    }
  }
  return false;
}
```

dup1

dup2

```
public static boolean dup2(int[] A) {
  for (int i = 0; i < A.length - 1; i += 1) {
    if (A[i] == A[i + 1]) {
      return true;
    }
  }
  return false;
}
```

What do we mean when we say that dup2 is better?

dup1 does more work (takes longer to run) to get the same result

For the rest of today's lecture, we'll measure a program's performance in terms of **runtime**. (Other metrics exist)

```
public static boolean dup1(int[] A) {
  for (int i = 0; i < A.length; i += 1) {
    for (int j = i + 1; j < A.length; j += 1) {
      if (A[i] == A[j]) {
         return true;
      }
    }
  }
  return false;
}
```

dup1

dup2

```
public static boolean dup2(int[] A) {
  for (int i = 0; i < A.length - 1; i += 1) {
    if (A[i] == A[i + 1]) {
      return true;
    }
  }
  return false;
}
```

How Do I Runtime Characterization?

Our goal is to somehow **characterize the runtimes** of the functions below.

- Characterization should be **simple** and **mathematically rigorous**.
- Characterization should **demonstrate superiority** of dup2 over dup1.

```
public static boolean dup1(int[] A) {
  for (int i = 0; i < A.length; i += 1) {
    for (int j = i + 1; j < A.length; j += 1) {
      if (A[i] == A[j]) {
         return true;
      }
    }
  }
  return false;
}
```

dup1

dup2

```
public static boolean dup2(int[] A) {
  for (int i = 0; i < A.length - 1; i += 1) {
    if (A[i] == A[i + 1]) {
      return true;
    }
  }
  return false;
}
```

Techniques for Measuring Computational Cost

Technique 1: Measure execution time in seconds using a client program.

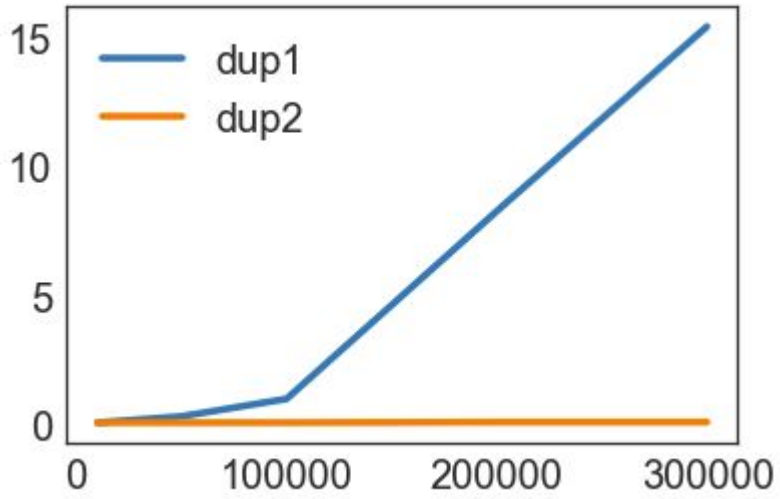
- Tools:
  - Physical stopwatch.
  - Unix has a built in `time` command that measures execution time.
  - Princeton Standard library has a `Stopwatch` class.


```
public static void main(String[] args) {
  int N = Integer.parseInt(args[0]);
  int[] A = makeArray(N);
  dup1(A);
}
```

Time Measurements for dup1 and dup2

N	dup1	dup2
10000	0.08	0.08
50000	0.32	0.08
100000	1.00	0.08
200000	8.26	0.1
400000	15.4	0.1

Time to complete (in seconds)

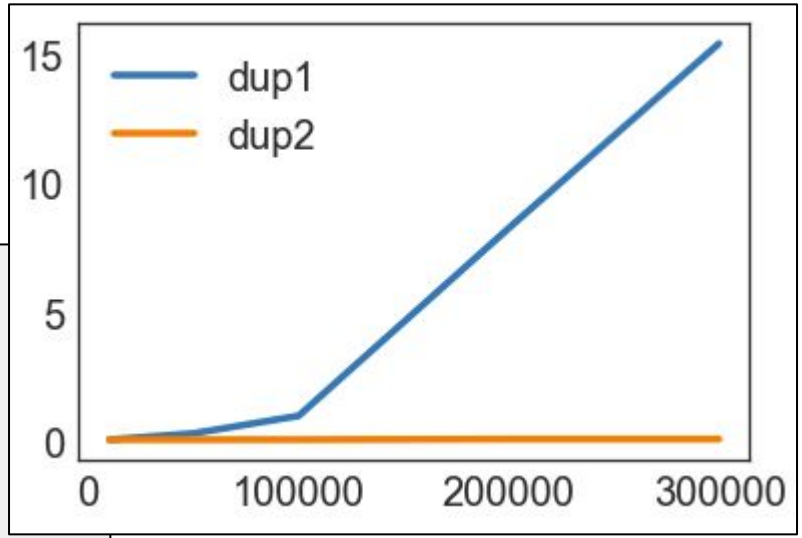


Techniques for Measuring Computational Cost

Technique 1: Measure execution time in seconds using a client program.

- Good: Easy to measure, meaning is obvious.
- Bad: May require large amounts of computation time. Result varies with machine, compiler, input data, programming language, etc.

Interesting observation: If you double the size of the input, dup1 takes ~4x longer, while dup2 takes ~2x longer. True regardless of language and machine.




```
public static void main(String[] args)
  int N = Integer.parseInt(args[0]);
  int[] A = makeArray(N);
  dup1(A);
}
```

Exact Operation Counting

Lecture 11, CS61B, Fall 2024

Goal: Measuring Code Efficiency

Intuitive Runtime Metrics

- Clock Time
- **Exact Operation Counting**
- Exact Count Exercise

Asymptotic Analysis

- Why Scaling Matters
- Computing Worst Case Order of Growth (Tedious Approach)
- Computing Worst Case Order of Growth (Simplified Approach)

Asymptotic Notation

- Big Theta (a.k.a. Order of Growth)
- Big O and Big Omega

Technique 2A: Count possible operations for an array of size N = 10,000.

- Good: Machine independent. Input dependence captured in model.
- Bad: Tedious to compute. Array size was arbitrary. Doesn't tell you actual time. Different arrays of size 10,000 take different number of operations

```
for (int i = 0; i < A.length; i += 1) {
  for (int j = i+1; j < A.length; j += 1) {
    if (A[i] == A[j]) {
       return true;
    }
  }
}
return false;
```

operation	count, N=10000
i = 0	1
j = i + 1	1 to 10000
less than (<)	2 to 50,015,001
increment (+=1)	0 to 50,005,000
equals (==)	1 to 49,995,000
array accesses	2 to 99,990,000

The counts are tricky to compute. Work not shown.

Technique 2B: Count possible operations in terms of input array size N.

- Good: Machine independent. Input dependence captured in model. Tells you how algorithm **scales**.
- Bad: Even more tedious to compute. Not a function (f(N) associated with multiple possible values).

```
for (int i = 0; i < A.length; i += 1) {
  for (int j = i+1; j<A.length; j += 1) {
    if (A[i] == A[j]) {
       return true;
    }
  }
}
return false;
```

operation	symbolic count	count, N=10000
i = 0	1	1
j = i + 1	1 to N	1 to 10000
less than (<)	2 to $(N^2+3N+2)/2$	2 to 50,015,001
increment (+=1)	0 to $(N^2+N)/2$	0 to 50,005,000
equals (==)	1 to $(N^2-N)/2$	1 to 49,995,000
array accesses	2 to N^2-N	2 to 99,990,000

Techniques for Measuring Computational Cost

Technique 3: For each input size N, select one specific input to "represent" that input size. Count the number of operations for that input.

Most common choices are:

- Worst case input (the input that takes the most operations)

-3	-1	2	4	5	8	10	12

- Best case input (the input that takes the fewest operations)

4	4	4	4	4	4	4	4

Techniques for Measuring Computational Cost

Technique 3: For each input size N, select one specific input to "represent" that input size. Count the number of operations for that input in terms of N.

- Good: Machine independent. Input dependence captured in model. Tells you how algorithm **scales**. Now a function!
- Bad: Still tedious to compute. Result is a table of values, so not really easy to compare.

```
for (int i = 0; i < A.length; i += 1) {
  for (int j = i+1; j<A.length; j += 1) {
    if (A[i] == A[j]) {
       return true;
    }
  }
}
return false;
```

operation	Best Case Input	Worst Case Input
i = 0	1	1
j = i + 1	1	N
less than (<)	2	$(N^2+3N+2)/2$
increment (+=1)	0	$(N^2+N)/2$
equals (==)	1	$(N^2-N)/2$
array accesses	2	N^2-N

Exact Count Exercise

Lecture 11, CS61B, Fall 2024

Goal: Measuring Code Efficiency

Intuitive Runtime Metrics

- Clock Time
- Exact Operation Counting
- **Exact Count Exercise**

Asymptotic Analysis

- Why Scaling Matters
- Computing Worst Case Order of Growth (Tedious Approach)
- Computing Worst Case Order of Growth (Simplified Approach)

Asymptotic Notation

- Big Theta (a.k.a. Order of Growth)
- Big O and Big Omega

Techniques for Measuring Computational Cost [dup2]

Your turn: Try to come up with rough estimates for the best and worst case counts for at least one of the operations.

- Tip: Don't worry about being off by one. Just try to predict the rough magnitudes of each.

```
for (int i = 0; i < A.length - 1; i += 1){
  if (A[i] == A[i + 1]) {
    return true;
  }
}
return false;
```

operation	Best Case	Worst Case
i = 0	1	1
less than (<)		
increment (+=1)		
equals (==)		
array accesses		

Techniques for Measuring Computational Cost [dup2]

Your turn: Try to come up with rough estimates for the symbolic and exact counts for at least one of the operations.

```
for (int i = 0; i < A.length - 1; i += 1) {
  if (A[i] == A[i + 1]) {
    return true;
  }
}
return false;
```

operation	Best Case	Worst Case
i = 0	1	1
less than (<)	1	N
increment (+=1)	0	N-1
equals (==)	1	N-1
array accesses	2	2N-2

Especially observant folks may notice we didn't count everything, e.g. "- 1" and "+ 1" operations. We'll see why this omission is not a problem very shortly.

If you did this exercise but were off by one, that's fine. The exact numbers aren't that important.

Why Scaling Matters

Lecture 11, CS61B, Fall 2024

Goal: Measuring Code Efficiency

Intuitive Runtime Metrics

- Clock Time
- Exact Operation Counting
- Exact Count Exercise

Asymptotic Analysis

- **Why Scaling Matters**
- Computing Worst Case Order of Growth (Tedious Approach)
- Computing Worst Case Order of Growth (Simplified Approach)

Asymptotic Notation

- Big Theta (a.k.a. Order of Growth)
- Big O and Big Omega

Measuring a Program's Performance

Expressing a program's efficiency boils down to two main steps:

1. Define some **metric** by which the program's efficiency will be evaluated, and measure the program's efficiency according to that metric.
    - Example metrics
        - Number of seconds it takes to run the program on a given input
        - Number of bytes of memory used by the program
        - Average error for randomized/approximation algorithms
2. This gives us a (mathematical) function that we can then **classify**.
    - Classification must be:
        - Simple
        - Mathematically rigorous
        - Model the program's efficiency "well enough"

Note that for step 2, we need a function in the mathematical sense (not a series of instructions that gets "run" by a computer, but a map that associates inputs with specific outputs)

Comparing Algorithms

Which algorithm is better? Why?

operation	Worst Case Input
i = 0	1
j = i + 1	N
less than (<)	$(N^2+3N+2)/2$
increment (+=1)	$(N^2+N)/2$
equals (==)	$(N^2-N)/2$
array accesses	N^2-N

dup1

operation	Worst Case Input
i = 0	1
less than (<)	N
increment (+=1)	N-1
equals (==)	N-1
array accesses	2N-2

dup2

Comparing Algorithms

Which algorithm is better? dup2. Why?

- Fewer operations to do the same work.
- Better answer: Algorithm **scales better** in the worst case. $(N^2+3N+2)/2$ vs. N.
- Even better answer: Parabolas (N^2) grow faster than lines (N).

operation	dup1 Worst Case Input	dup2 Worst Case Input
i = 0	1	1
j = i + 1	N	0
less than (<)	$(N^2+3N+2)/2$	N
increment (+=1)	$(N^2+N)/2$	N-1
equals (==)	$(N^2-N)/2$	N-1
array accesses	N^2-N	2N-2

Asymptotic Behavior

In most cases, we care only about asymptotic behavior, i.e. what happens for very large N.

- Simulation of billions of interacting particles.
- Social network with billions of users.
- Logging of billions of transactions.
- Encoding of billions of bytes of video data.

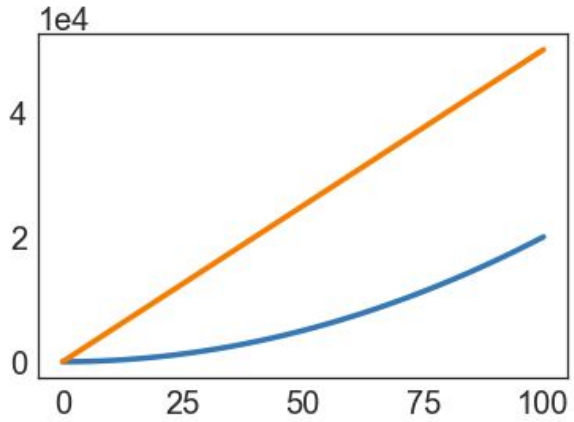
Algorithms which scale well (e.g. look like lines) have better asymptotic runtime behavior than algorithms that scale relatively poorly (e.g. look like parabolas).

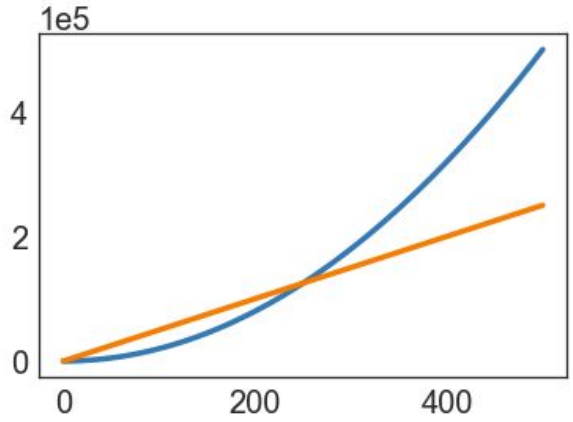
Parabolas vs. Lines

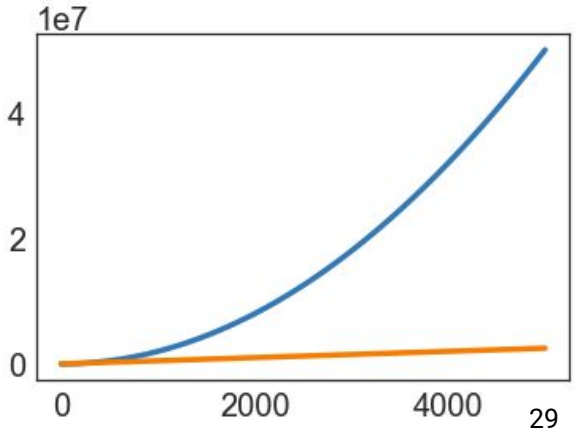
Suppose we have two algorithms that zerpify a collection of N items.

- `zerp1` takes $2N^2$ operations.
- `zerp2` takes 500N operations.

For small N, `zerp1` might be faster, but as dataset size grows, the parabolic algorithm is going to fall farther and farther behind (in time it takes to complete).







What matters in modern computing?

Modern computers are really fast and getting faster every day. At high processing speeds, scaling matters far more than lower order terms and constant factors.

	Maximum N for feasible computation (~ 1 minute runtime)			
Runtime function (in number of operations)	1 op/sec (human)	1M ops/sec (Slow Python)	1B ops/sec (modern CPU)	1T ops/sec (GPUs)
N^2	8	7746	244949	7745967
N^2+N	7	7745	244948	7745966
$2N^2$	5	5477	173205	5477226
$500N$	0	120000	120000000	120000000000
$0.001N^3$	39	3915	39149	391487

Scaling Across Many Domains

We'll informally refer to the "shape" of a runtime function as its ***order of growth*** (will formalize soon).

- Effect is dramatic! Often determines whether a problem can be solved at all.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

(from Algorithm Design: Tardos, Kleinberg)

Duplicate Finding

Our goal is to somehow **characterize the runtimes** of the functions below.

- ✗ Characterization should be **simple** and **mathematically rigorous**.
- ✓ Characterization should **demonstrate superiority** of `dup2` over `dup1`.

operation	dup1 Worst Case Input	dup2 Worst Case Input
i = 0	1	1
j = i + 1	N	0
less than (<)	$(N^2+3N+2)/2$	N
increment (+=1)	$(N^2+N)/2$	N-1
equals (==)	$(N^2-N)/2$	N-1
array accesses	N^2-N	2N-2
characterization	Quadratic	Linear

Computing Worst Case Order of Growth (Tedious Approach)

Lecture 11, CS61B, Fall 2024

Duplicate Finding

Our goal is to somehow **characterize the runtimes** of the functions below.

- Characterization should be **simple** and **mathematically rigorous**.
- We can reduce a function to a count of operations, but that's still not really a runtime. Let's try to convert this table into a proper runtime function

operation	dup1 Worst Case Input	dup2 Worst Case Input
i = 0	1	1
j = i + 1	N	0
less than (<)	$(N^2+3N+2)/2$	N
increment (+=1)	$(N^2+N)/2$	N-1
equals (==)	$(N^2-N)/2$	N-1
array accesses	N^2-N	2N-2

Intuitive Order of Growth Identification

Consider the algorithm below. What do you expect will be the **order of growth** of the runtime for the algorithm? (You don't need to know the exact amount of time it takes to do each operation!)

A. N [linear]
B. N^2 [quadratic]
C. N^3 [cubic]
D. N^6 [sextic]

operation	count
array access	$100N^2 + 3N$
greater than (>)	$2N^3 + 1$
print statement	5,000

In other words, if we plotted total runtime vs. N, what shape would we expect?

Intuitive Order of Growth Identification

Consider the algorithm below. What do you expect will be the **order of growth** of the runtime for the algorithm?

A. N [linear]
B. N^2 [quadratic]
C. N^3 [cubic]
D. N^6 [sextic]

operation	count
array access	$100N^2 + 3N$
greater than (>)	$2N^3 + 1$
print statement	5,000

Argument:

- Suppose array access takes α nanoseconds, > takes β nanoseconds, and print statements take γ nanoseconds.
- Total time is $\alpha(100N^2 + 3N) + \beta(2N^3 + 1) + 5000\gamma$ nanoseconds.
- For very large N, the $2\beta N^3$ term is much larger than the others **regardless of the values of α, β, and γ**.

Extremely important point.
Make sure you understand it!

Simplifying our runtime equation

What should the order of growth of dup1 be?

```
for (int i = 0; i < A.length; i += 1) {
  for (int j = i+1; j < A.length; j += 1) {
    if (A[i] == A[j]) {
       return true;
    }
  }
}
return false;
```

operation	Worst Case Input
i = 0	1
j = i + 1	N
less than (<)	$(N^2+3N+2)/2$
increment (+=1)	$(N^2+N)/2$
equals (==)	$(N^2-N)/2$
array accesses	N^2-N

Simplifying our runtime equation

Ideally, we want to be able to classify the behavior of the runtime of dup1 as N^2.

Let's look at what simplifications are necessary to achieve this, to make sure that we don't lose any important information along the way.

Later, we'll express those simplifications mathematically so we can formally state that this runtime is N^2.

```
for (int i = 0; i < A.length; i += 1) {
  for (int j = i+1; j < A.length; j += 1) {
    if (A[i] == A[j]) {
       return true;
    }
  }
}
return false;
```

operation	Worst Case Input
i = 0	1
j = i + 1	N
less than (<)	$(N^2+3N+2)/2$
increment (+=1)	$(N^2+N)/2$
equals (==)	$(N^2-N)/2$
array accesses	N^2-N

Convert count table into runtime

To make things easier, we should rewrite our table as a runtime.

Let's assume that each operation takes some constant amount of time, and let's use variables to represent each of those times.

Worst-case Runtime = $\alpha+\beta N+\gamma(N^2+3N+2)/2+\delta(N^2+N)/2+\varepsilon(N^2-N)/2+\zeta(N^2-N)$

$=(\gamma/2+\delta/2+\varepsilon/2+\zeta)N^2+(\beta+3\gamma/2+\delta/2-\varepsilon/2-\zeta)N+(\alpha+\gamma)$

```
for (int i = 0; i < A.length; i += 1) {
  for (int j = i+1; j < A.length; j += 1) {
    if (A[i] == A[j]) {
       return true;
    }
  }
}
return false;
```

operation	Variable	Worst Case Input
i = 0	α	1
j = i + 1	β	N
less than (<)	γ	$(N^2+3N+2)/2$
increment (+=1)	δ	$(N^2+N)/2$
equals (==)	ε	$(N^2-N)/2$
array accesses	ζ	N^2-N

Remove lower-order terms

Lower-order terms will eventually be much smaller than the highest-order term, **regardless of the coefficient**. For sufficiently high N, those terms are negligible

Still need to formally prove:

- Lower-order terms are actually negligible for sufficiently high N
- N^2 "dominates" over N and constants

But these seem reasonable enough assumptions to make for now.

```
for (int i = 0; i < A.length; i += 1) {
  for (int j = i+1; j < A.length; j += 1) {
    if (A[i] == A[j]) {
       return true;
    }
  }
}
return false;
```

Worst-case Runtime =

$(\gamma/2+\delta/2+\varepsilon/2+\zeta)N^2+(\beta+3\gamma/2+\ldots)N+(\alpha+\gamma)$

Combining coefficient of the leading term

The values of γ, δ, ε, and ζ are unknown durations, and will likely change depending on the specific system you use.

Depending on these exact values, we'll end up with a different leading coefficient. But in any case, that coefficient will be some constant value that depends on the system. Let's combine them into a single variable C.

```
for (int i = 0; i < A.length; i += 1) {
  for (int j = i+1; j < A.length; j += 1) {
    if (A[i] == A[j]) {
       return true;
    }
  }
}
return false;
```

Worst-case Runtime =

CN^2 $(C = (\gamma/2+\delta/2+\varepsilon/2+\zeta))$

Removing coefficient of the leading term

Since the constant C changes depending on the machine running the code, it doesn't convey any information about the code itself. In theory, any positive C could occur depending on the system.

We still need to note that the constant exists, though, so let's "group" together all functions of the form CN^2 (along with anything that simplifies to CN^2). We'll name this group "$\Theta(N^2)$" (Theta N squared).

```
for (int i = 0; i < A.length; i += 1) {
  for (int j = i+1; j < A.length; j += 1) {
    if (A[i] == A[j]) {
       return true;
    }
  }
}
return false;
```

Worst-case Runtime is a function within the Theta class $\Theta(N^2)$.

Or, using math notation,

Worst-case Runtime $\in \Theta(N^2)$

Simplification Summary

Simplifications:

1. Convert table to runtime
2. Ignore lower order terms.
3. Ignore any coefficients.

These three simplifications are OK because we only care about the "**order of growth**" of the runtime.

operation	Worst Case Count
i = 0	1
j = i + 1	N
less than (<)	$(N^2+3N+2)/2$
increment (+=1)	$(N^2+N)/2$
equals (==)	$(N^2-N)/2$
array accesses	N^2-N

→ Worst-case Runtime $\in \Theta(N^2)$

Simplification Summary: Repeating the Process for dup2

Simplifications:

1. Convert table to runtime
2. Ignore lower order terms.
3. Ignore any coefficients.

These three simplifications are OK because we only care about the "**order of growth**" of the runtime.

operation	Worst Case Input
i = 0	1
less than (<)	N
increment (+=1)	N-1
equals (==)	N-1
array accesses	2N-2

→ What Theta class will contain dup2's worst-case runtime?

Simplification Summary: Repeating the Process for dup2

Simplifications:

1. Convert table to runtime
2. Ignore lower order terms.
3. Ignore any coefficients.

These three simplifications are OK because we only care about the "**order of growth**" of the runtime.

operation	Worst Case Input
i = 0	1
less than (<)	N
increment (+=1)	N-1
equals (==)	N-1
array accesses	2N-2

→ Worst-case Runtime $\in \Theta(N)$

Summary of Our (Painful) Analysis Process

One thing to note: If N -> N^2, then 2N -> $(2N)^2 = 4N^2$; doubling the size of the input means 4x longer runtime

This is what we observed earlier! So despite all our simplifications, we're still capturing the "essence" of the program's runtime.

operation	Worst Case Input
i = 0	1
j = i + 1	N
less than (<)	$(N^2+3N+2)/2$
increment (+=1)	$(N^2+N)/2$
equals (==)	$(N^2-N)/2$
array accesses	N^2-N

Worst-case Runtime $\in \Theta(N^2)$

Summary of Our (Painful) Analysis Process

Our process:

- Construct a table of exact counts of all possible operations.
- Convert table into a runtime function, then simplify the function until it's a Theta class.

operation	Worst Case Input
i = 0	1
j = i + 1	N
less than (<)	$(N^2+3N+2)/2$
increment (+=1)	$(N^2+N)/2$
equals (==)	$(N^2-N)/2$
array accesses	N^2-N

→ Worst-case Runtime $\in \Theta(N^2)$

By using our simplifications from the outset, we can avoid building the table at all!

Computing Worst Case Order of Growth (Simplified Approach)

Lecture 11, CS61B, Fall 2024

Simplified Analysis Process

Rather than building the entire table/runtime function, we can instead:

- Treat anything that takes constant time (relative to N) as a single operation
  - Why? Any coefficients will be ignored by our Theta class anyway, so 5 operations is the same as 1 operation.
- Figure out the order of growth for the operation count by either:
  - Making an exact count, then discarding the unnecessary pieces.
  - Using intuition and inspection to determine order of growth (only possible with lots of practice).

Let's redo our analysis of `dup1` with this new process.

- This time, we'll show all our work.

Analysis of Nested For Loops (Based on Exact Count)

Find the order of growth of the worst case runtime of dup1.

```
int N = A.length;
for (int i = 0; i < N; i += 1)
   for (int j = i + 1; j < N; j += 1)
       if (A[i] == A[j])
           return true;
return false;
```

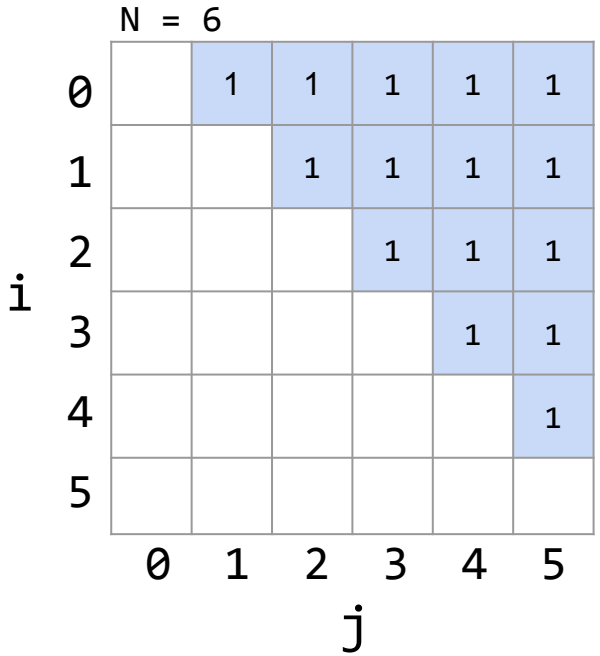
Analysis of Nested For Loops (Based on Exact Count)

Find the order of growth of the worst case runtime of dup1.

```
int N = A.length;
for (int i = 0; i < N; i += 1)
   for (int j = i + 1; j < N; j += 1)
            //1 unit of time
return false;
```

Analysis of Nested For Loops (Based on Exact Count)

Find the order of growth of the worst case runtime of dup1.




```
int N = A.length;
for (int i = 0; i < N; i += 1)
   for (int j = i + 1; j < N; j += 1)
            //1 unit of time
return false;
```

Worst case runtime:

$f(N) = 1 + 2 + 3 + \ldots + (N - 3) + (N - 2) + (N - 1)$

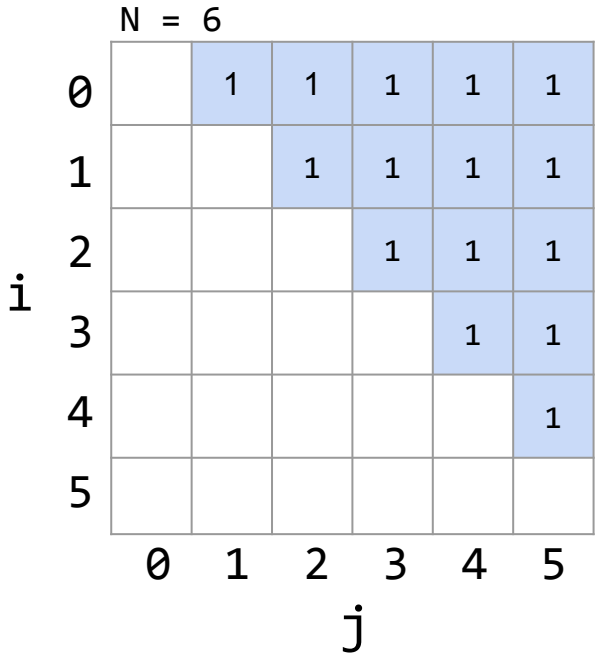
$f(N) = (N - 1) + (N - 2) + (N - 3) + \ldots + 3 + 2 + 1$

$2f(N) = \underbrace{N + N + \ldots + N}_{\text{N-1 of these}} = N(N - 1)$

$\therefore f(N) = N(N - 1)/2$

Analysis of Nested For Loops (Based on Exact Count)

Find the order of growth of the worst case runtime of dup1.




```
int N = A.length;
for (int i = 0; i < N; i += 1)
   for (int j = i + 1; j < N; j += 1)
            //1 unit of time
return false;
```

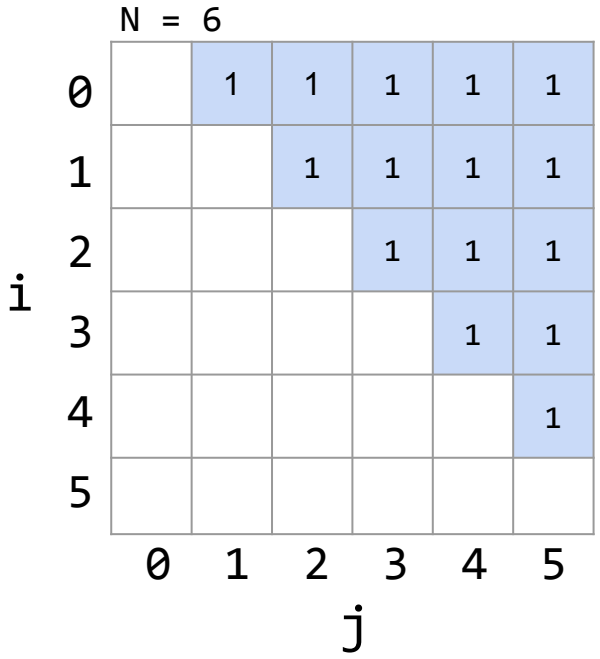
Worst case number of steps:

$f(N) = 1 + 2 + 3 + \ldots + (N - 3) + (N - 2) + (N - 1) = N(N-1)/2$

Worst case runtime = $f(N) \in \Theta(N^2)$

Analysis of Nested For Loops (Simpler Geometric Argument)

Find the order of growth of the worst case runtime of dup1.




```
int N = A.length;
for (int i = 0; i < N; i += 1)
   for (int j = i + 1; j < N; j += 1)
            //1 unit of time
return false;
```

Worst case number of steps:

- Given by area of right triangle of side length N-1.
- Area of triangle is length x height / 2 = (approximately) NxN/2

Worst case runtime $\in \Theta(N^2)$

Big Theta (a.k.a. Order of Growth)

Lecture 11, CS61B, Fall 2024

Goal: Measuring Code Efficiency

Intuitive Runtime Metrics

- Clock Time
- Exact Operation Counting
- Exact Count Exercise

Asymptotic Analysis

- Why Scaling Matters
- Computing Worst Case Order of Growth (Tedious Approach)
- Computing Worst Case Order of Growth (Simplified Approach)

Asymptotic Notation

- **Big Theta (a.k.a. Order of Growth)**
- Big O and Big Omega

Formalizing Order of Growth

Given a function Q(N), we can apply our last two simplifications (ignore low orders terms and multiplicative constants) to yield the order of growth of Q(N).

- Example: $Q(N) = 3N^3 + N^2$
- Order of growth: N^3

Let's finish out this lecture by formally defining a Theta class.

- The math might seem daunting at first.
- … but the idea is exactly the same! We're effectively formalizing our simplifications into a mathematical equation.
  - We don't care about small cases
  - We don't care about being off by a constant factor

Big-Theta: Formal Definition (Visualization)

$$R(N) \in \Theta(f(N))$$

means there exist positive constants k_1 and k_2 such that:

$$k_1 \cdot f(N) \leq R(N) \leq k_2 \cdot f(N)$$

for all values of N greater than some N_0.

i.e. very large N

Example: $4N^2+N \in \Theta(N^2)$

- $R(N) = 4N^2+N$
- $f(N) = N^2$
- k1 = 3
- k2 = 5

Order of Growth Exercise

Consider the functions below.

- Informally, what is the “shape” of each function for very large N?
- In other words, what is the order of growth of each function?

function	order of growth
$N^3 + 3N^4$	
$1/N + N^3$	
$1/N + 5$	
$Ne^N + N$	
$40 \sin(N) + 4N^2$	

Order of Growth Exercise

Consider the functions below.

- Informally, what is the “shape” of each function for very large N?
- In other words, what is the order of growth of each function?

function	order of growth
$N^3 + 3N^4$	N^4
$1/N + N^3$	N^3
$1/N + 5$	1
$Ne^N + N$	Ne^N
$40 \sin(N) + 4N^2$	N^2

Big-Theta

Suppose we have a function R(N) with order of growth f(N).

- In “Big-Theta” notation we write this as $R(N) \in \Theta(f(N))$.
- Examples:
  - $N^3 + 3N^4 \in \Theta(N^4)$
  - $1/N + N^3 \in \Theta(N^3)$
  - $1/N + 5 \in \Theta(1)$
  - $Ne^N + N \in \Theta(Ne^N)$
  - $40 \sin(N) + 4N^2 \in \Theta(N^2)$

function R(N)	order of growth
$N^3 + 3N^4$	N^4
$1/N + N^3$	N^3
$1/N + 5$	1
$Ne^N + N$	Ne^N
$40 \sin(N) + 4N^2$	N^2

Big-Theta Challenge (Visualization)

Suppose R(N) = $(4N^2+3N*\ln(N))/2$.

- Find a simple f(N) and corresponding k_1 and k_2.

$$R(N) \in \Theta(f(N))$$

means there exist positive constants k_1 and k_2 such that:

$$k_1 \cdot f(N) \leq R(N) \leq k_2 \cdot f(N)$$

for all values of N greater than some N_0.

i.e. very large N

Big-Theta Challenge (Visualization)

Suppose R(N) = $(4N^2+3N*\ln(N))/2$.

- $f(N) = N^2$
- $k_1 = 1$
- $k_2 = 3$
- Why?

$$R(N) \in \Theta(f(N))$$

means there exist positive constants k_1 and k_2 such that:

$$k_1 \cdot f(N) \leq R(N) \leq k_2 \cdot f(N)$$

for all values of N greater than some N_0.

i.e. very large N

Big-Theta Challenge (Visualization)

Suppose R(N) = $(4N^2+3N*\ln(N))/2$.

- $(4N^2+3N*\ln(N))/2 = 2N^2+3/2N*\ln(N)$
- $N^2 \leq 2N^2 \leq 2N^2+3/2N*\ln(N)$
- $N^2 \leq 2N^2+3/2N*\ln(N)$
- $f(N) \leq R(N)$

$$R(N) \in \Theta(f(N))$$

means there exist positive constants k_1 and k_2 such that:

$$k_1 \cdot f(N) \leq R(N) \leq k_2 \cdot f(N)$$

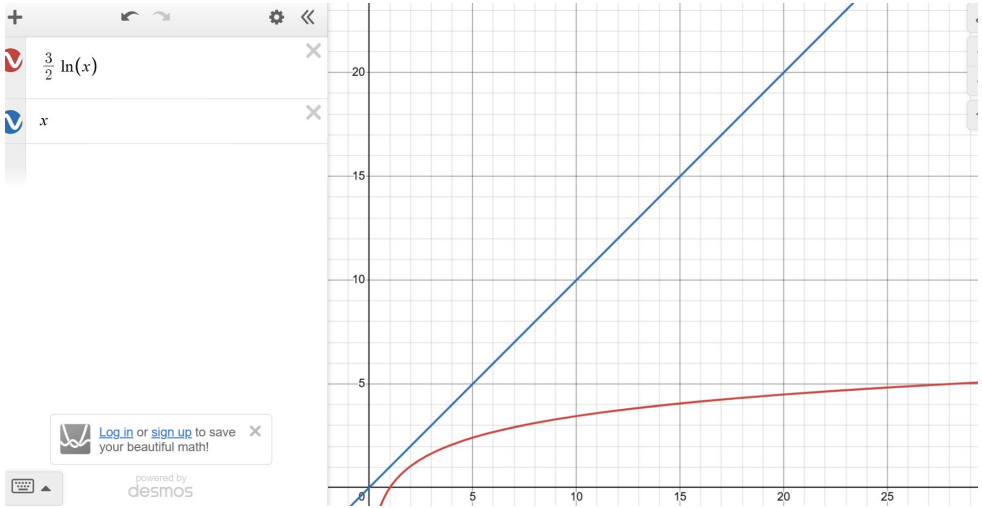
for all values of N greater than some N_0.

i.e. very large N

Big-Theta Challenge (Visualization)

Suppose $R(N) = (4N^2+3N*\ln(N))/2$.

- $(4N^2+3N*\ln(N))/2 = 2N^2+3/2N*\ln(N)$
- When $3/2*\ln(N) < N$, $3/2N*\ln(N) < N^2$. $3/2*\ln(N) < N$ is true for all N.
- $2N^2+3/2N*\ln(N) < 2N^2+N^2$
- $2N^2+3/2N*\ln(N) \leq 3N^2$
- $R(N) \leq 3f(N)$



Big-Theta Challenge (Visualization)

Suppose R(N) = $(4N^2+3N*\ln(N))/2$.

- $f(N) \leq R(N) \leq 3f(N)$ for all $N>0$
- So $R(N) \in \Theta(N^2)$

$$R(N) \in \Theta(f(N))$$

means there exist positive constants k_1 and k_2 such that:

$$k_1 \cdot f(N) \leq R(N) \leq k_2 \cdot f(N)$$

for all values of N greater than some N_0.

i.e. very large N

Big-Theta and Runtime Analysis

The formal definition doesn't change anything about runtime analysis (no need to find k_1 or k_2 or anything like that).

- We can still do the same simplifications we did earlier, as long as we prove they don't affect our Theta class

operation	worst case count
i = 0	1
j = i + 1	$\Theta(N)$
less than (<)	$\Theta(N^2)$
increment (+=1)	$\Theta(N^2)$
equals (==)	$\Theta(N^2)$
array accesses	$\Theta(N^2)$

Worst case runtime: $\Theta(N^2)$

Big O and Big Omega

Lecture 11, CS61B, Fall 2024

Goal: Measuring Code Efficiency

Intuitive Runtime Metrics

- Clock Time
- Exact Operation Counting
- Exact Count Exercise

Asymptotic Analysis

- Why Scaling Matters
- Computing Worst Case Order of Growth (Tedious Approach)
- Computing Worst Case Order of Growth (Simplified Approach)

Asymptotic Notation

- Big Theta (a.k.a. Order of Growth)
- **Big O and Big Omega**

Big Theta

We used Big Theta to describe the order of growth of a function.

function R(N)	order of growth
$N^3 + 3N^4$	$\Theta(N^4)$
$1/N + N^3$	$\Theta(N^3)$
$1/N + 5$	$\Theta(1)$
$Ne^N + N$	$\Theta(Ne^N)$
$40 \sin(N) + 4N^2$	$\Theta(N^2)$

We also used Big Theta to describe the rate of growth of the runtime of a piece of code.

Big O and Big Omega

Whereas Big Theta can informally be thought of as something like “equals”, Big O can be thought of as “less than or equal” and Big Omega can be thought of as "greater than or equal"

Example, the following are all true:

- $N^3 + 3N^4 \in \Theta(N^4)$
- $N^3 + 3N^4 \in O(N^4)$
- $N^3 + 3N^4 \in O(N^6)$
- $N^3 + 3N^4 \in O(N^{N!})$
- $N^3 + 3N^4 \in \Omega(N^4)$
- $N^3 + 3N^4 \in \Omega(N^2)$
- $N^3 + 3N^4 \in \Omega(1)$

Big-Theta: Formal Definition

$$R(N) \in \Theta(f(N))$$

means there exist positive constants k_1 and k_2 such that:

$$k_1 \cdot f(N) \leq R(N) \leq k_2 \cdot f(N)$$

for all values of N greater than some N_0.

i.e. very large N

Big O

$$R(N) \in O(f(N))$$

means there exists a positive constant k_2 such that:

$$R(N) \leq k_2 \cdot f(N)$$

for all values of N greater than some N_0.

i.e. very large N

Big Omega

$$R(N) \in \Omega(f(N))$$

means there exists a positive constant k_1 such that:

$$k_1 \cdot f(N) \leq R(N)$$

for all values of N greater than some N_0.

i.e. very large N

Big Theta vs. Big O

We will see why big O is practically useful in the upcoming Disjoint Sets lecture.

	Informal meaning:	Family	Family Members
Big Theta $\Theta(f(N))$	Order of growth is f(N).	$\Theta(N^2)$	$N^2/2$ $2N^2$ $N^2 + 38N + N$
Big O $O(f(N))$	Order of growth is less than or equal to f(N).	$O(N^2)$	$N^2/2$ $2N^2$ $\lg(N)$
Big Omega $\Omega(f(N))$	Order of growth is greater than or equal to f(N).	$\Omega(N^2)$	$N^2/2$ $2N^2$ $N^{N!}$

Summary

Given a code snippet, we can express its runtime as a function R(N), where N is some property of the input of the function (often the size of the input).

Rather than finding R(N) exactly, we instead usually only care about the order of growth of R(N). We can then classify R(N) in its appropriate Theta Class

One approach (not universal):

- Reduce constant time operations to "1 unit of time", and let R(N) be the count of how many times that operation occurs as a function of N.
  - Often (but not always) we consider the worst case count.
- Determine order of growth f(N) for R(N), i.e. $R(N) \in \Theta(f(N))$
- Can use O as an alternative for Θ. O is used for upper-bounding R(N) (especially if R(N) is hard to evaluate). Ω isn't used often in practical settings, but is often used in theoretical CS for lower bounds.

Citations

TSP problem solution, title slide:
http://support.sas.com/documentation/cdl/en/ornoaug/65289/HTML/default/viewer.htm#ornoaug_optnet_examples07.htm#ornoaug.optnet.map002g

Table of runtimes for various orders of growth: Kleinberg & Tardos, Algorithm Design.