CS 361
Data Structures & Algs
Lecture 5

Prof. Tom Hayes
University of New Mexico
9-04-2012

Reminders

- Written Assignment 1 - due Thursday. Exercises 1,2,3,4,6 from Chapter One
- Reading: through sec 2.4 assigned for today. Ask questions, and re-read!

Last Week

Analyzing the Gale-Shapley algorithm

Bounding the running time

At most n^2 proposals.

To do: handle proposals fast.

Proving correctness

3 aspects: crashes? finishes? output ok?

invariants

4/5 Examples: Algorithms Problem

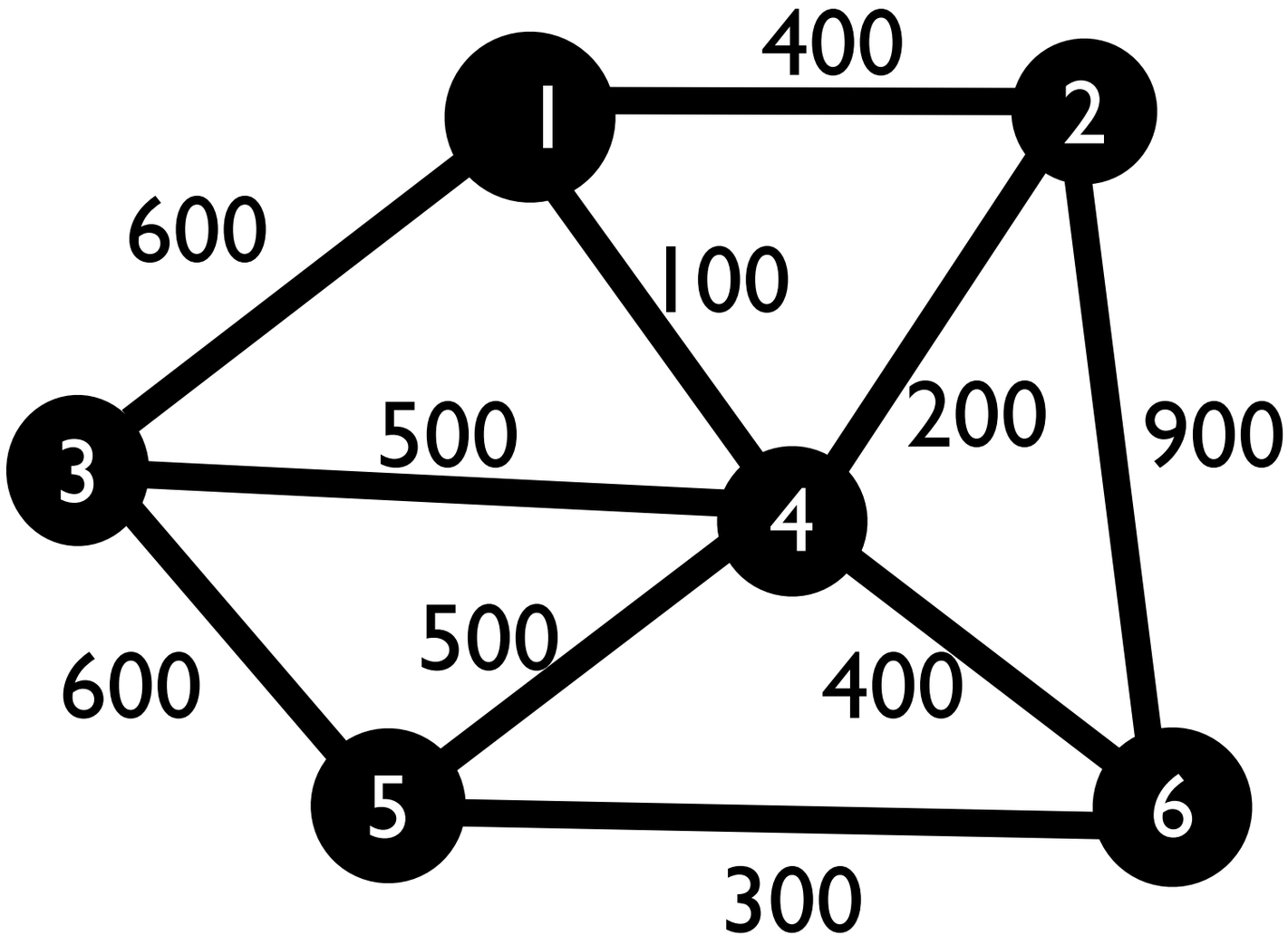
Today

Traveling Salesman

Running times.

“Big O” notation.

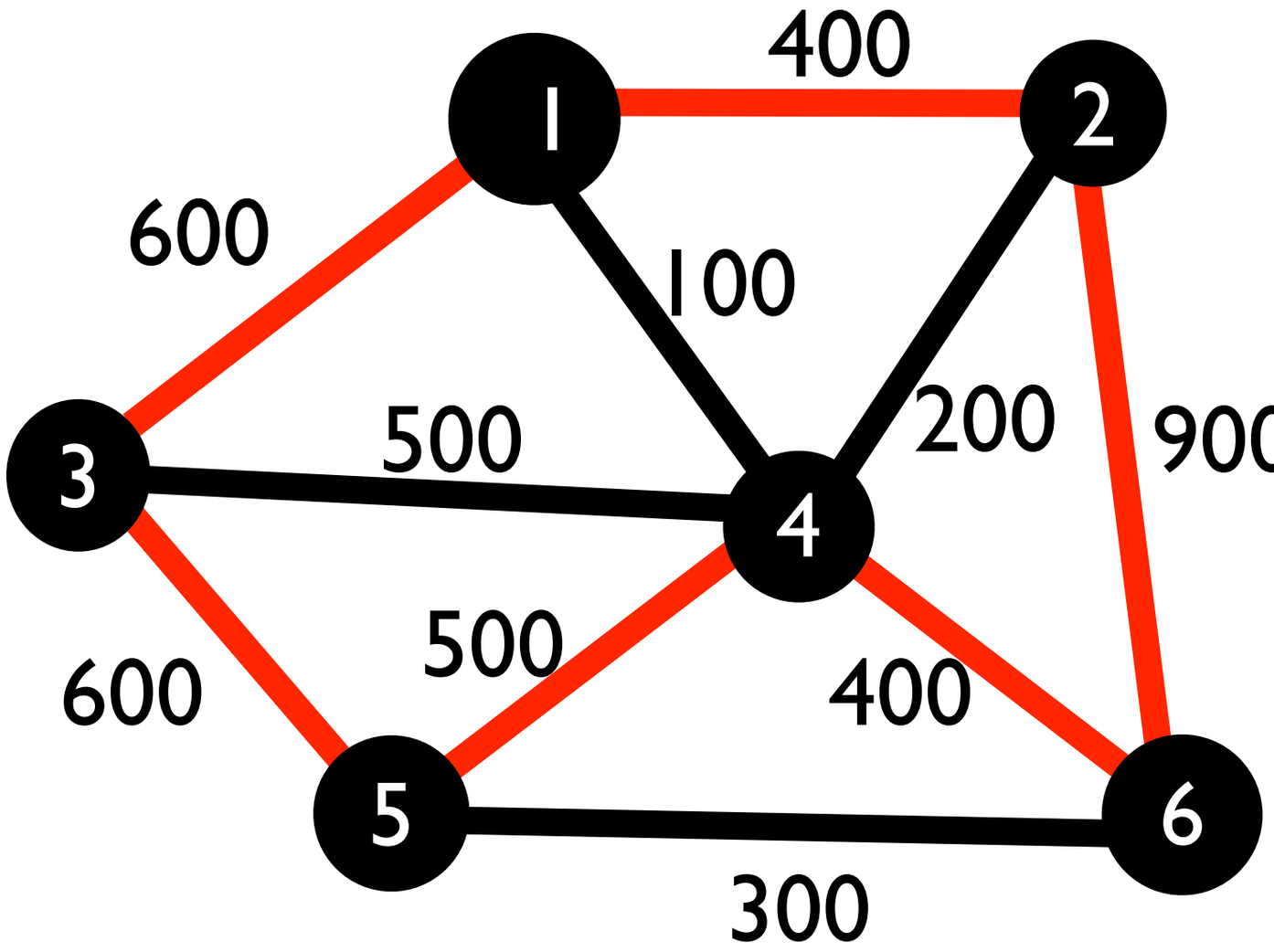
Traveling Salesman



Find: Loop visiting each town exactly once.

Minimize total cost.

Traveling Salesman

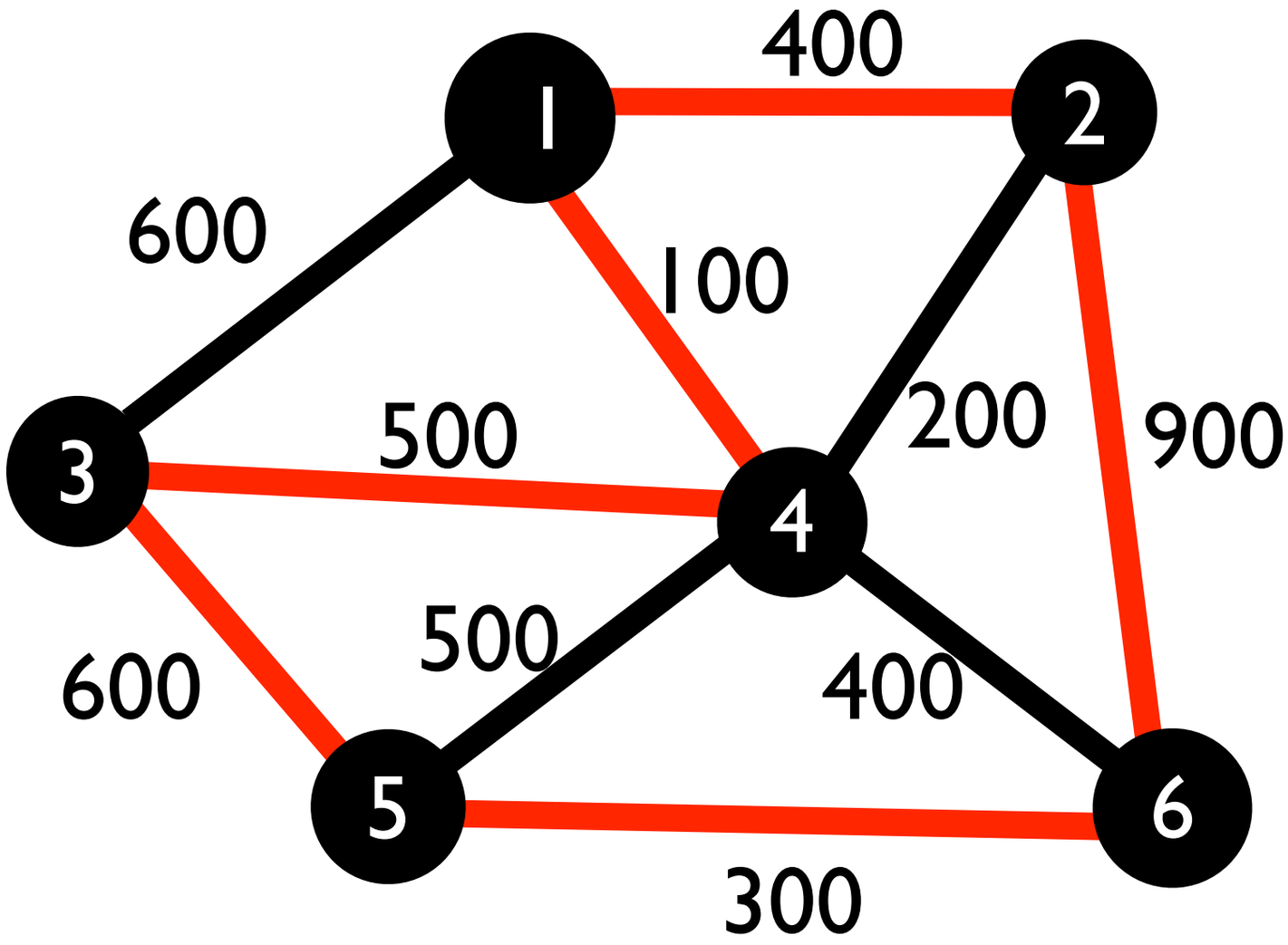


Cost: 3400

Find: Loop visiting each town exactly once.

Minimize total cost.

Traveling Salesman

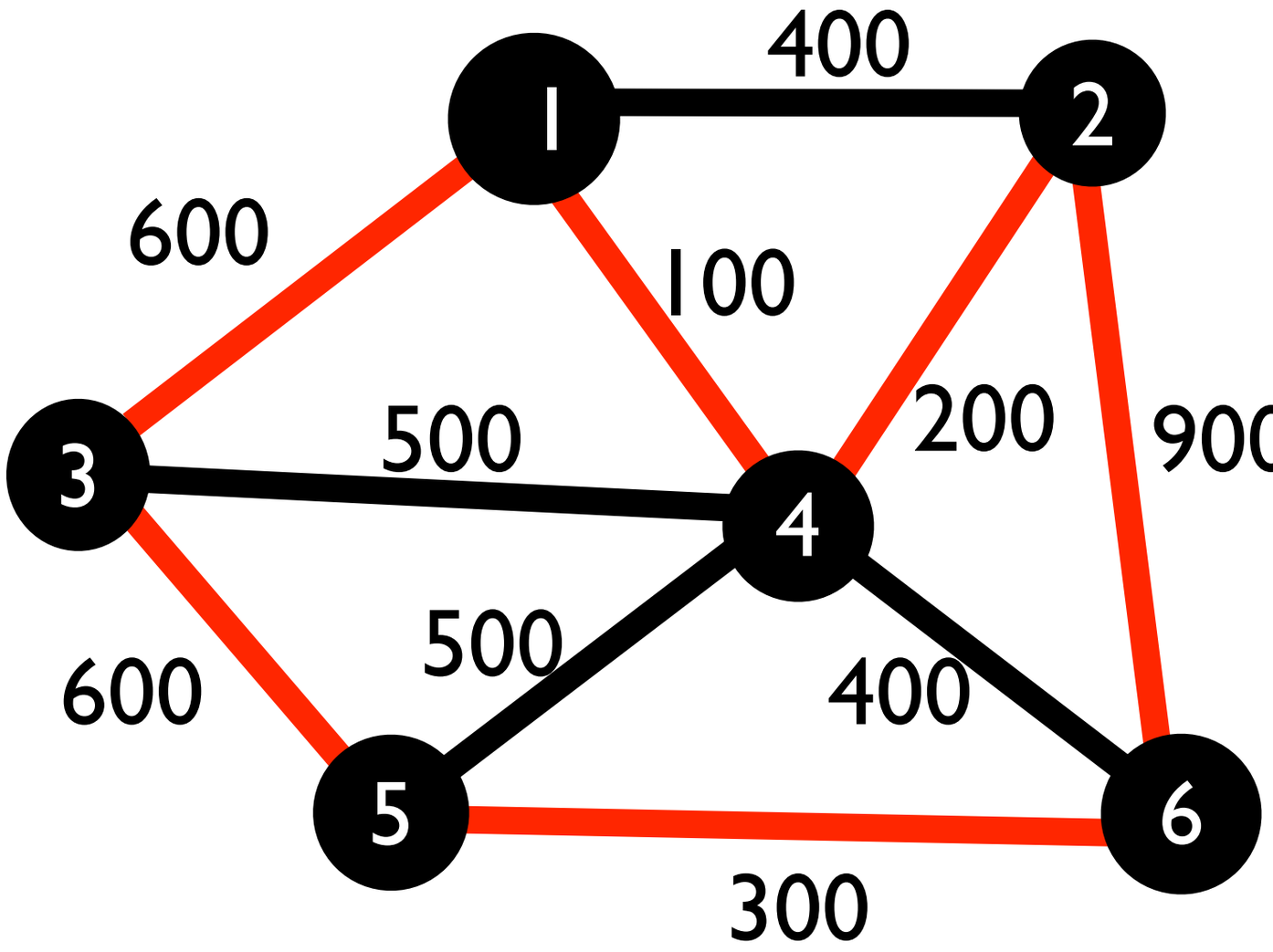


Cost: 2800

Find: Loop visiting each town exactly once.

Minimize total cost.

Traveling Salesman



Cost: 2700

Find: Loop visiting each town exactly once.

Minimize total cost.

Traveling Salesman

Input: n by n matrix of intercity costs

Output: a “complete tour” or n-city loop

Correctness: minimum cost

Traveling Salesman

Input: n by n matrix of intercity costs

Output: a “complete tour” or n-city loop

Correctness: minimum cost

How many possible outputs?

Traveling Salesman

Input: n by n matrix of intercity costs

Output: a “complete tour” or n-city loop

Correctness: minimum cost

How many possible outputs?

n! = n*(n-1)*(n-2)*...*2*1

Traveling Salesman

Input: n by n matrix of intercity costs

Output: a “complete tour” or n-city loop

Correctness: minimum cost

How many possible outputs?

n! = n*(n-1)*(n-2)*...*2*1

How can we avoid checking them all?

Traveling Salesman

A dynamical programming solution.

Start at city 1. Consider all ways to split the n cities into a “first half” and “second half”. Example: middle city is 5, first half visits 2 and 6. second half visits 3 and 4.

Find the best solution to each half, then add them.

What will this cost? Less than (n!) ?

How many splits?

Start at city 1. Consider all ways to split the n cities into a “first half” and “second half”. Example: middle city is 5, first half visits 2 and 6. second half visits 3 and 4.

choices: middle city?

How many splits?

Start at city 1. Consider all ways to split the n cities into a “first half” and “second half”. Example: middle city is 5, first half visits 2 and 6. second half visits 3 and 4.

choices: middle city? n-1

How many splits?

Start at city 1. Consider all ways to split the n cities into a “first half” and “second half”. Example: middle city is 5, first half visits 2 and 6. second half visits 3 and 4.

choices: middle city? n-1

cities in first half?

How many splits?

Start at city 1. Consider all ways to split the n cities into a “first half” and “second half”. Example: middle city is 5, first half visits 2 and 6. second half visits 3 and 4.

choices: middle city? n-1

cities in first half? $< 2^n$

How many splits?

Start at city 1. Consider all ways to split the n cities into a “first half” and “second half”. Example: middle city is 5, first half visits 2 and 6. second half visits 3 and 4.

choices: middle city? n-1

cities in first half? $< 2^n$

overall: $< n\ 2^n$

Work per split?

Need the best route from 1 through the "first half" of cities, to "middle city". How many options to check?

Work per split?

Need the best route from 1 through the "first half" of cities, to "middle city". How many options to check?

< (n/2)!

Work per split?

Need the best route from 1 through the "first half" of cities, to "middle city". How many options to check?

< (n/2)!

Overall work?

Work per split?

Need the best route from 1 through the "first half" of cities, to "middle city". How many options to check?

< (n/2)!

Overall work?

< n (2^n) (n/2)!

Work per split?

Need the best route from 1 through the "first half" of cities, to "middle city". How many options to check?

$< (n/2)!$

Overall work?

$< n\ (2^n)\ (n/2)!$

How does this compare to (n!) ?

Comparison

$n\ (2^n)\ (n/2)!$

How does this compare to (n!) ?

(see whiteboard -- no skipping class!)

Comparison

n (2^n) (n/2)!

How does this compare to (n!) ?

(see whiteboard -- no skipping class!)

So, we can see that n! is much bigger, nearly the square of $(2^n)((n/2)!)$

Comparison

n (2^n) (n/2)!

How does this compare to (n!) ?

(see whiteboard -- no skipping class!)

So, we can see that n! is much bigger, nearly the square of $(2^n)((n/2)!)$

Can we do better still?

Better Still, Recurse!

Instead of using brute force to solve the problem on the first half, instead, recursively apply the same idea.

Divide into two “halves” of size roughly n/4. Find the shortest route through each half, recursively using the same idea.

Better Still, Recurse!

Instead of using brute force to solve the problem on the first half, instead, recursively apply the same idea.

Divide into two “halves” of size roughly n/4. Find the shortest route through each half, recursively using the same idea.

Dynamic programming: keep track of solutions to all sub-problems, and re-use when possible. (i.e., memoize)

Intermission

We stopped here on Tuesday 9/4. The remaining slides are included, but will also be covered in Thursday's lecture.

Analysis, DP solution

How many sub-problems will we look at, total?

Each looks like this: start city, intermediate cities, end city.

$\leq n^2 2^n$ possibilities

Time to solve one sub-problem? Loop through all the ways to split it into 2 subproblems. Sum up the values for those, keeping the min. constant * $n^2 2^n$

TSP, final thoughts

Improved from (n!) brute force, to roughly $n^2\,2^n$ via our dynamic programming alg.

How big an improvement?

TSP, final thoughts

Improved from (n!) brute force, to roughly $n^2 2^n$ via our dynamic programming alg. How big an improvement?

(a) take logs. log(n!) is roughly (n log(n)), while log($n^2 2^n$) is roughly n.

TSP, final thoughts

Improved from (n!) brute force, to roughly $n^2\ 2^n$ via our dynamic programming alg. How big an improvement?

(a) take logs. log(n!) is roughly (n log(n)), while $\log(n^2\ 2^n)$ is roughly n.

(b) if we have time for 1 trillion operations, or 1 quadrillion, how big is n?

TSP, final thoughts

Improved from (n!) brute force, to roughly $n^2\ 2^n$ via our dynamic programming alg. How big an improvement?

(a) take logs. log(n!) is roughly (n log(n)), while $\log(n^2\ 2^n)$ is roughly n.

(b) if we have time for 1 trillion operations, or 1 quadrillion, how big is n?

For $n^2\ 2^n$, can do 30, or 40.

For n!, can do 15, or 18.

NP

NP is the class of YES/NO decision problems, where, for every input of size n, if the correct output is YES, then there exists an efficiently checkable proof of that fact.

TSP asks, “Is there a tour of these cities that costs at most M?”

This is in NP, because, when the answer is YES, then, if I give you the tour, you can verify the YES answer. This does not mean finding such a tour can be done quickly.

NP

NP is the class of YES/NO decision problems, where, for every input of size n, if the correct output is YES, then there exists an efficiently checkable proof of that fact.

We say a problem is *NP-hard* if it could be used to solve every problem in NP.

Say a problem in *NP-complete* if it is NP-hard *and* a member of NP.

P vs NP

This multimillion-dollar problem asks, are there any problems in NP, for which it is not possible to efficiently determine the answer when a proof is NOT GIVEN?

In other words, is it easier to verify a proof than to come up with one on your own?

TSP is NP-complete. This means, if you can find an efficient algorithm to solve it, you have proven every problem in NP is easy.

Efficient Algorithms

Our Dynamic Programming algorithm for TSP was a big improvement, but is still not efficient.

To be efficient, when the input size is n, our algorithm should run in, say, time 10n, or perhaps $50n^2$, or $100n^3 + 50n \log(n)$. To be general, let's say any function that is less than some power of n, such as n^{10}. For short, poly(n).

Big O notation

We want to be able to reason carefully about running times, but without "sweating irrelevant details."

Who cares if the running time is n^3 versus $n^3 + 10.5\ n^2 - 0.5n$? It can matter only for a few small values of n. In the "big picture" what really matters is, approximately how big an input can I handle in the trillion or so steps I have time to do.

Big O helps us achieve these goals.

Big O, formally

Suppose g is a function describing a running time. g(n) tells us the amount of time our program runs on an input of length n. The notation O(g) refers to the class of all functions that, for large inputs, do not grow faster than a constant times g. In other words, a function f is in O(g) if there exists a constant C such that, for every n, f (n) ≤ C g(n). *see caveat in a few slides.

Big O, informally

One generally writes “f = O(g)” to indicate that f is in the function class O(g). This is just a shorthand, and can lead you into trouble if you try to use any laws of “=”.

For instance, it would be correct to write $20n^2 + 6\ n = O(n^3)$ and also to write $20n^2 + 6\ n = O(n^2)$. However, $O(n^2)$ and $O(n^3)$ are not equal. Exercises 2.5(ab) illustrate some further pitfalls.

Caveat - zeros

$10(n-1)^3 = O(n^4)$. Why?

But, is $10n^3 = O((n-1)^4)$?

We want it to be.

But, for n=1, there is no constant C that could work. Why? $(1-1)^4 = 0$.

Fancier definition: f = O(g) means there exists C, n_0, such that, for every $n \geq n_0$, $f(n) \leq Cg(n)$.

Why define it like that?

f = O(g) means:

There exists C>0 and n_0 such that, whenever $n \geq n_0$, we have $f(n) \leq C\ g(n)$.

(1) Simplifies analysis: A sequence of O(1) “atomic” steps (no recursive function calls or loops) can be replaced by a single “step” conceptually.

(2) Gets at big question: limiting growth rates for f and g.

More on Big-O

f = O(g) is a 1-sided guarantee!

We know “f is not much bigger than g (for large inputs)” but we don’t know whether “g is much bigger than f (for large inputs)”

This is a good thing! (Less to prove)

Q: What if we want a 2-sided guarantee?

A: Ω, Θ notation

Ω, Θ notation

$f = \Omega(g)$ means $g = O(f)$. That is, g is (up to a constant factor, and for large inputs) a lower bound on f.

$f = \Theta(g)$ means both $f = O(g)$ and $f = \Omega(g)$. That is, f and g "are of the same order"

Practice with big-O

How to prove that 5 n + 2 = O(n)?

Reasoning: 5 n + 2 <= C n (goal)

Try C = 6. Plug in: 5 n + 2 ≤ 6 n solve

2 ≤ (6 - 5) n = n. Choose $n_0 = 2$.

We're now ready to fill in proof.

Practice with big-O

How to prove that 5 n + 2 = O(n)?

Proof: Let C = 6. Let $n_0 = 2$.

Assume n >= n_0. Then

5 n + 2

= 5n + n_0 (def of n_0)

<= 6 n (since n ≥ n_0)

= C n. (def of C)

Therefore, 5 n + 2 = O(n) by definition.

Practice with big-O

How to prove that $\log(3n^2) = O(\log(n))$?

Reasoning: $\log(3n^2) \leq C \log(n)$ (goal)

LHS = $\log(3) + \log(n^2) = \log(3) + 2 \log(n)$

Goal: $\log(3) + 2 \log(n) \leq C \log(n)$.

Take $C = 3 > 2$. Solve

$\log(3) + 2 \log(n) \leq 3 \log(n)$ for n, to find n_0

$\log(3) \leq \log(n)$. So $n \geq 3$. Take $n_0 = 3$.

Practice with big-O

How to prove that $\log(3n^2) = O(\log(n))$?

Proof: Choose $C = 3$ and $n_0 = 3$.

Suppose $n \geq n_0$.

$\log(3n^2)$

$= \log(3) + 2 \log(n)$ (arithmetic)

$= \log(n_0) + 2 \log(n)$ (def of n_0)

$\leq \log(n) + 2 \log(n) = 3 \log(n)$ (since $n \geq n_0$)

$= C \log(n)$ (def of C). Thus $f = O(g)$

Practice with big-O

Suppose f = O(g) and g = O(H).

Prove: f = O(H).

Reasoning: Goal: $f(n) \leq C\ H(n)$.

f = O(g) means: There is C_1, n_1 such that as long as $n \geq n_1$ we have $f(n) \leq C_1\ g(n)$.

g = O(H) means: There is C_2, n_2 such that as long as $n \geq n_2$ we have $g(n) \leq C_2\ H(n)$.

$f(n) \leq C_1\ g(n) \leq C_1\ (C_2\ H(n)) = (C_1\ C_2)\ H(n)$

Practice with big-O

f = O(g) means: There is C_1, n_1 such that as long as $n \geq n_1$ we have $f(n) \leq C_1\, g(n)$.

g = O(H) means: There is C_2, n_2 such that as long as $n \geq n_2$ we have $g(n) \leq C_2\, H(n)$.

$f(n) \leq C_1\, g(n) \leq C_1\, (C_2\, H(n)) = (C_1\, C_2)\, H(n)$

Guess $C = C_1\, C_2$

n_0 = ?. Need: $f(n) \leq C_1\, g(n)$. From top, need $n \geq n_1$. Need: $g(n) \leq C_2\, H(n)$. Thus need $n \geq n_2$. Choose $n_0 = \max\{n_1, n_2\}$.

Practice with big-O

Suppose f = O(g) and g = O(h).

Prove: f = O(h).

Proof: f = O(g) means: There is C_1, n_1 such that as long as $n \geq n_1$ we have $f(n) \leq C_1\, g(n)$.

g = O(H) means: There is C_2, n_2 such that as long as $n \geq n_2$ we have $g(n) \leq C_2\, H(n)$.

Choose $C = C_1\, C_2$, and $n_0 = \max\{n_1, n_2\}$.

Then

Proof: f = O(g) means: There is C_1, n_1 such that as long as $n \geq n_1$ we have $f(n) \leq C_1\, g(n)$.

g = O(H) means: There is C_2, n_2 such that as long as $n \geq n_2$ we have $g(n) \leq C_2\, H(n)$.

Choose $C = C_1\, C_2$, and $n_0 = \max\{n_1, n_2\}$.

Suppose $n \geq n_0$

Then

$$f(n) \leq C_1\, g(n) \leq C_1\, (C_2\, H(n)) = (C_1\, C_2)\, H(n)$$

$$= C\, H(n).$$

Thus f = O(H).

Choose $C = C_1\ C_2$, and $n_0 = \max\{n_1, n_2\}$.

Suppose $n \geq n_0$

Then

$f(n) \leq C_1\ g(n)$ (since $n \geq n_0 \geq n_1$ and above)

$\leq C_1\ (C_2\ H(n))$ (since $n \geq n_0 \geq n_2$ and above)

$= (C_1\ C_2)\ H(n)$ (arithmetic)

$= C\ H(n)$. (def of C)

Thus $f = O(H)$.