CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: Davis 117

Business

- Apologies for missing class! (I *really* didn't want to…)
- Project 2 Grading update:
  - In progress, but will be slow (catching up from illness may take a while…)
  - Please *meet with me* if you'd like prompt feedback on any part of Project 2!
- PS3 due 11:59pm today
- Expect PS4 out soon
  - Due no sooner than 1 week after it's assigned
- Project 3 out
  - Will discuss today
  - Due Nov. 3; deadline may be slightly extended

For Proj3: Evaluating Expressions

- How do we, as people, evaluate an arithmetic expression like (((3+7)*5) – (4*2))?
  - Do we read left to right? Not really….

For Proj3: Evaluating Expressions

- How do we, as people, evaluate an arithmetic expression like (((3+7)*5) – (4*2))?
  - Do we read left to right? Not really….

  - There are lots of components to that, and we first evaluate each component, and then combine them to evaluate the entire expression

For Proj3: Evaluating Expressions

- How do we, as people, evaluate an arithmetic expression like (((3+7)*5) – (4*2))?
  - Do we read left to right? Not really….
  - There are lots of components to that, and we first evaluate each component, and then combine them to evaluate the entire expression
- It's as if we had an *Evaluate*() function
  - … And applied it *recursively*! We use it on each component, then combine to get our end result

For Proj3: Evaluating Expressions

- How do we, as people, evaluate an arithmetic expression like (((3+7)*5) – (4*2))?
  - Do we read left to right? Not really….
  - There are lots of components to that, and we first evaluate each component, and then combine them to evaluate the entire expression
- It's as if we had an *Evaluate*() function
  - … And applied it *recursively*! We use it on each component, then combine to get our end result
- Evaluating *propositional logic* expressions works the same way, in general
  - Let's get *formally* introduced to propositional logic!

Propositions

- Defn: *proposition* – a statement that has the property of truth or falsity
- Propositions are the key elements to represent, analyze, or explain declarative knowledge

Propositions:

•Washington, D.C. is the capital of the USA.

•Waterville is the capital of Maine.

•1 + 1 = 2

•2 + 2 = 3

The first and third of these are true; the second and fourth are false.

Non-Propositions:

•What time is it?

•Pass the salt.

•x + 1 = 2

•x*y + 5*z

presuming values for x, y, z are not given / known

Propositional operators

- Recall: *proposition* – a statement that has the property of truth or falsity
  - Often, we use *propositional letters* (or *variables*) to represent propositions: e.g., *p* stands for "Poughkeepsie is the capital of NY"
- There are several *operators* (sometimes called *boolean operators*) that can construct new propositions from old ones
  - *Negation* ("not"): if *P* is a proposition, *not P* is a proposition
  - *Conjunction* ("and"): *P and Q*
  - *Disjunction* ("or"): *P or Q*
  - *Implication* ("if – then"): *if P then Q*

Propositional operator: Negation

- Whatever the value of p, True or False, the value of proposition *not p* (written $\neg p$) is the opposite
  - If p is "Today is Monday," $\neg p$ is "It is not the case that today is Monday," or more simply "Today is not Monday."
- Negation can be expressed with a *truth table*

p	¬p
T	F
F	T

proposition

truth values

Propositional operator: Conjunction

- Conjunction—the "and" operator
  - Whatever the values of propositions p, q, conjunction *p and q* (written $p \wedge q$ or *p && q*) is also a proposition
  - If p is "Today is Monday" and q is "It is snowing today," then $p \wedge q$ is "Today is Monday and it is snowing today."
    - $p \wedge q$ is true on snowy Mondays and false on any day that is not Monday, and on any day that is Monday but not snowing
- Conjunction values as a *truth table*

p	q	p^q
T	T	T
T	F	F
F	T	F
F	F	F

Propositional operator: Disjunction

- Disjunction—the "or" operator
  - Whatever the values of propositions *p*, *q*, disjunction *p or q* (written $p \vee q$ or *p* || *q*) is also a proposition
  - If *p* is "Today is Monday" and *q* is "It is snowing today," then $p \vee q$ is "Today is Monday or it is snowing today."
    - $p \vee q$ is true on any day that is a Monday or on which it is snowing – including snowy Mondays (it is not *exclusive*) – and false only on days that are not Mondays on which it is not snowing
- Disjunction values as a *truth table*

p	q	p ∨ q
T	T	T
T	F	T
F	T	T
F	F	F

The *non-exclusive* sense of "or" can be a bit subtle

Exercise: What would the *exclusive-or* operator's truth table look like?

It turns out there *is* such an operator, and it's commonly used in logic! The English word "or" is a complicated thing to understand!

Propositional operator: Implication

- Implication—the "if…then" operator (also called *conditional*)
  - Whatever the values of propositions *p*, *q*, implication *if p then q* (written $p \rightarrow q$) is also a proposition
  - If *p* is "Today is Monday" and *q* is "It is snowing today," then $p \rightarrow q$ is "If today is Monday then it is snowing today."
  - Vocabulary: in $p \rightarrow q$, *p* is called the *hypothesis* (or *antecedent*) and *q* is called the *conclusion* (or *consequent*)
- Implication values as a *truth table*

p	q	p → q
T	T	T
T	F	F
F	T	T
F	F	T

Really? *These* are the truth values for implication?

They look like the values for (¬p v q)! (Exercise: Check for yourselves!!)

Sounds if-y: *Material Implication*

- Meaning for implication symbol → in propositional logic is referred to as *material implication*
  - It says that $p \rightarrow q$ is False exactly when p is True and q is False
  - Not the same as every meaning of "if…then" in English, but it's what's used in logic

p	q	p → q
T	T	T
T	F	F
F	T	T
F	F	T

Examples of material implication and natural language usage:

•Politician says: "If I am elected, then I will fix the environment"

•False if the speaker is elected and doesn't fix the environment

•True if, e.g., the speaker doesn't get elected

•"If today is Friday, then 2 + 2 = 4"

•True *no matter what day it is*

•"If today is Friday, then 2 + 2 = 5"

•True except on Fridays, even though 2 + 2 = 5 is false!

Exercise: Evaluating propositional logic expressions

- Defn: Propositions are *boolean*-valued expressions—i.e., their values are either True or False
- Propsotional expressions are evaluated like any other mathematical expressions

Examples: Let p = True, q = False, r = True. What do the following expressions evaluate to?

1. $(p \wedge \neg r)$
2. $(q \vee \text{False})$
3. $(p \rightarrow q)$
4. $(r \vee (p \wedge q))$
5. $((p \vee r) \rightarrow ((p \vee q) \wedge r))$
6. $(\text{True} \rightarrow r)$

For Proj3: *Satisfiability*, and *Assignments of Truth Values to Variables*

- An assignment of values to variables is… what you think it is! Something like *[p = True, q = False, r = True]* is as assignment of ***truth values*** to the variables *p, q, r*
- We say a ***propositional logic expression*** (***PLE***) *P* is ***satisfiable*** if there is an assignment of truth values to the variables in *P* so that *P* evaluates to True

Examples: Let p = True, q = False, r = True. What do the following expressions evaluate to?

1. (p ^ ¬r)
2. (q v False)
3. (p → q)
4. (r v (p ^ q))
5. ((p v r) → ((p v q) ^ r))
6. (True → r)

Business: Project 3

Have you read through Proj3 yet?

Please note the separate, supplementary document about Truth Tables!

- Project 3 out, due Nov. 19
  - Deadline may be *slightly* extended
- Parts of Project 3:
  1. **Recursive Algorithm to Evaluate Propositional Logic Expressions**: Your team will create an algorithm to do the kind of thing a programming language does—evaluate propositional logic (boolean) expressions.
  2. **Exhaustive Search Algorithm for *Satisfiability***: The *Satisfiability* problem (*SAT*, for short) asks if there is a way for a propositional logic expression to be made True. Your team will create an exhaustive search algorithms to solve this problem.
  3. **Improvements**: Your team will improve upon your exhaustive search algorithm.
  4. **Create and Give a Presentation**: Your team will present work from the previous three parts of the assignment, *using recurrences and inductive arguments loop invariants* where appropriate.

Solving Recurrences

- We'll cover three common techniques for solving recurrences—i.e., getting θ or O bounds on the solution:
  - *Unwinding* (or *backward substitution*): "Unroll" the recurrence until it reaches a base case, then count / analyze the cost represented

    We already did an example of unwinding, and we'll do another one soon!
  - *Recursion-tree method*: Represent costs as nodes in a tree and analyze total cost
  - *Master method*: Solve recurrences of the form

    $$T(n) = a*T(n/b) + f(n)$$

Unwinding

This name may make it sound more relaxing than it actually is, but as methods for solving recurrences go, it's pretty mellow.

- An example: Solve $T(n) = 2*T(n/2) + n$

What information is missing from this recurrence, which we will need to be able to solve it?

- *Unwind* the recurrence by plugging in the definition on successively smaller arguments:
  - From the definition, T(n) = 2T(n/2) + n
  - By that same definition, T(n/2) = 2T((n/2)/2) + (n/2) = 2T(n/4) + n/2
  - So, by plugging that in: T(n) = 2[2T(n/4) + n/2] + n
  - What would the next step(s) be in this unwinding process?
  - Where would it stop?

Unwinding

This name may make it sound more relaxing than it actually is, but as methods for solving recurrences go, it's pretty mellow.

- An example: Solve $T(n) = 2*T(n/2) + n$

What information is missing from this recurrence, which we will need to be able to solve it?

- *Unwind* the recurrence by plugging in the definition on successively smaller arguments:
  - From the definition, T(n) = 2T(n/2) + n

```
T(n)   = 2*T(n/2) + n
       = 2[2*T(n/4) + n/2] + n = 4T(n/4) + 2n
       = 4[2*T(n/8) + n/4] + 2n = 8T(n/8) + 3n
       ...
```

Do you see a pattern here? And when does this unwinding end?

Unwinding

This name may make it sound more relaxing than it actually is, but as methods for solving recurrences go, it's pretty mellow.

- An example: Solve $T(n) = 2*T(n/2) + n$
- For a base case, let's use T(1) = 1 (or θ(1), if we want)
- *Unwind* the recurrence by plugging in the definition on successively smaller arguments:
  - From the definition, T(n) = 2T(n/2) + n

```
T(n)   = 2*T(n/2) + n
       = 2[2*T(n/4) + n/2] + n = 4T(n/4) + 2n
       = 4[2*T(n/8) + n/4] + 2n = 8T(n/8) + 3n
       ...
  →    = 2^k[T(n/2^k)] + k*n
       ...
```

The *k'th step* shown here illustrates the pattern that holds for any relevant *k*. It can help with our analysis to show this in our work!

Unwinding

This name may make it sound more relaxing than it actually is, but as methods for solving recurrences go, it's pretty mellow.

- An example: Solve $T(n) = 2*T(n/2) + n$
- For a base case, let's use T(1) = 1 (or θ(1), if we want)
- *Unwind* the recurrence by plugging in the definition on successively smaller arguments:
  - From the definition, T(n) = 2T(n/2) + n

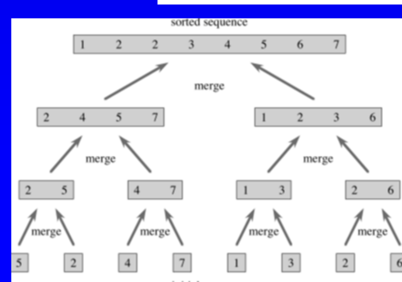
```
T(n)   = 2*T(n/2) + n
       = 2[2*T(n/4) + n/2] + n = 4T(n/4) + 2n
       ...
       = 2^k[T(n/2^k)] + k*n
       ...
       = n*T(1) + (lg n)*n
       = θ(n lg n)
```

The lg n term comes because the recurrence unwinds lg n times before hitting the base case… do you see why?

Recursion Trees: An Overview

- Recursion trees can represent how a recursive algorithm…
  - Breaks input down into recursive calls on sub-problems,
  - Or, *equivalently*, combines recursive calls into a solution on the original problem
- Here's an example from CLRS: Mergesort
  - Each node shows input size at that level of recursive calls
    - Here, original input size 8, breaks into sub-problems of size 4, etc.

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```



- **This example shows the recursion going *up* the tree—combining solutions**
- **Note that the input sizes at each node would be the same for the recursion going *down* the tree, breaking into sub-problems**

Recursion Trees For Solving Time-Complexity Recurrences

- When using recursion trees to solve for time complexity, though, we don't need quite that much information
  - We *do* need the structure, showing how the algo divides and re-combines its inputs
  - We *do* need the input size at each node
  - We *do not* need details about exactly what the input is at each node
- What we need, for each node of the tree:
  - Input size at each node
  - A way to represent the work done (i.e., the runtime) at *that node* of the tree—*not including any other work done above or below it*

Recall: Asymptotic complexity is in terms of input size *n*, not individual inputs of a given size!

Let's do an example!

Recursion-Tree Method

- An example: Mergesort

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

T(n)

n

What's the recursion tree structure?

What's the cost at each tree-level (i.e., not counting levels below it)?

Recursion-Tree Method

- An example:
  Mergesort

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

- Set up a tree to total up the work done by the algorithm

T(n)

What's the recursion tree structure?

n

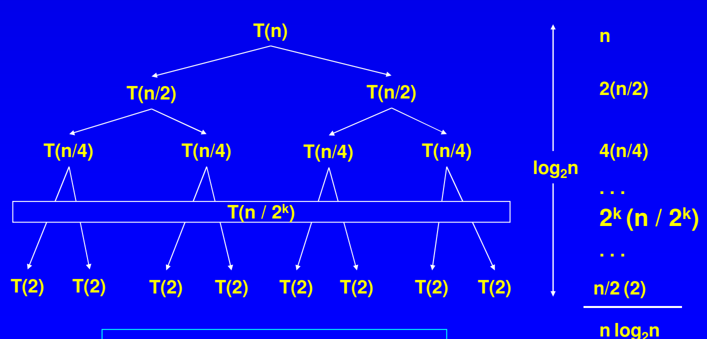
What's the cost at each tree-level (i.e., not counting levels below it)?

- Tree structure for complexity analysis corresponds to tree of recursive calls by the algorithm
- Total work by the algorithm: Sum of work at all levels of the tree

Recursion-Tree Method

- An example:
  Mergesort

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$



Recursion tree for algorithm

Total work done

Recursion Tree Exercises

- Use the recursion-tree method to solve the following recurrences for $n \geq 1$
  - $T(n) = 3T(n/3) + n$ if $n \geq 3$; 1 if $n < 3$ [assume n is a power of 3]
  - $T(n) = 4T(n/2) + n$ if $n \geq 2$; 1 if $n = 1$ [assume n is a power of 2]
  - $T(n) = T(n/2) + n$ if $n \geq 2$; 1 if $n = 1$ [assume n is a power of 2]
  - $T(n) = 2T(n/2) + n$ if $n \geq 2$; 1 if $n = 1$ [assume n is a power of 2]

Keep in mind the formula for the sum of a geometric series, from Appendix A:

$\sum_{i=0:n} c^i = (c^{n+1} - 1) / (c-1)$ **[for constant c \neq 1]**

Recursion Tree Exercises

- Use the recursion-tree method to solve the following recurrences for $n \geq 1$
  - $T(n) = 3T(n/3) + n$ if $n \geq 3$; 1 if $n < 3$
  - $T(n) = 4T(n/2) + n$ if $n \geq 2$; 1 if $n = 1$
  - $T(n) = T(n/2) + n$ if $n \geq 2$; 1 if $n = 1$
  - $T(n) = 2T(n/2) + n$ if $n \geq 2$; 1 if $n = 1$

- Those last three examples illustrate three different cases:
  1. The amount of work per level increases, with the most work done at the leaves of the tree

In fact, it increases geometrically—this means (see me for a proof, if you'd like), the total amount of work is θ(work done at leaves)

Recursion Tree Exercises

- Use the recursion-tree method to solve the following recurrences for $n \geq 1$
  - $T(n) = 3T(n/3) + n$ if $n \geq 3$; 1 if $n < 3$
  - $T(n) = 4T(n/2) + n$ if $n \geq 2$; 1 if $n = 1$
  - $T(n) = T(n/2) + n$ if $n \geq 2$; 1 if $n = 1$
  - $T(n) = 2T(n/2) + n$ if $n \geq 2$; 1 if $n = 1$
- Those last three examples illustrate three different cases:
  1. The amount of work per level increases, with the most work done at the leaves of the tree
  2. The amount of work per level decreases, with the most work done at the root

In fact, it decreases geometrically—this means (see me for a proof, if you'd like), the total amount of work is θ(work done at root)

Recursion Tree Exercises

- Use the recursion-tree method to solve the following recurrences for $n \geq 1$
  - $T(n) = 3T(n/3) + n$ if $n \geq 3$; 1 if $n < 3$
  - $T(n) = 4T(n/2) + n$ if $n \geq 2$; 1 if $n = 1$
  - $T(n) = T(n/2) + n$ if $n \geq 2$; 1 if $n = 1$
  - $T(n) = 2T(n/2) + n$ if $n \geq 2$; 1 if $n = 1$

Note: These three cases are important—we'll come back to them on a later slide!

- Those last three examples illustrate three different cases:
  1. The amount of work per level increases, with the most work done at the leaves of the tree
  2. The amount of work per level decreases, with the most work done at the root

  3. The amount of work per level is constant—and there are (lg n + 1) levels in the tree

There are (lg n + 1) levels in all three cases, really, but it's of particular importance here