CS 173 Examlet 6 Study Guide: Recursion Trees, Code Analysis & NP

Spring, 2025

▼ skills list screenshot

CS 173: Skills list for Examlet 6

- (Recap) Earlier algorithms and big-O content
- Recursion Trees
  - Given a recursively defined function, find its closed form by drawing a recursion tree and adding up the work at all levels. (Examples would look like those presented in the textbook, e.g. the level sums are constant or increase in a pattern based on the key summations given above.)
  - Find a big-O solution for a simple recursive definition, of the types that we've seen as examples of unrolling and recursion trees.
- Algorithms
  - Be familiar with the overall structure and big-O running times of the following algorithms.
    - Towers of Hanoi solver
    - Karatsuba's algorithm (you won't have to reproduce all the details)
  - Find a big-O solution for slightly harder recursive definitions than the ones for examlet 10, e.g. requiring use of the change of base formula.
  - Given an unfamiliar but fairly simple function in pseudo-code, analyze how long it takes using big-O notation. You should be able to analyze nested for loops, recursive functions, and simple examples of while loops.
  - For an algorithm involving loops (perhaps nested), express its running time using summations.
  - Given a recursive algorithm (familiar or unfamiliar) express its running time as a recursive definition.
- NP
  - Know that certain classes of sentences have exponentially many parse trees. (And thus producing all parses of a sentence requires exponential time.)
  - Know that the Towers of Hanoi puzzle has been proved to require exponential time.
  - Know that NP is the set of problems for which we can quickly (polynomial time) justify "yes" answers.
  - Know that co-NP is the set of problems for which we can quickly (polynomial time) justify "no" answers.
  - Know that problems in NP can be solved in exponential time, but it's not known whether they can be solved in polynomial time.
  - Know what an NP-complete problem is: a problem in NP for which a polynomial-time algorithm would imply that **any** problem in NP can be solved in polynomial time.
  - Know some examples of NP-complete problems: graph colorability, circuit satisfiability (Circuit SAT), propositional logic satisfiability, marker making, the travelling salesman problem.
  - Know that you can decide in polynomial time whether a graph is 2-colorable (aka bipartite).

Examlet 6 Skills:

(Recap) Earlier Algorithms & Big O Content

- Review the earlier Algorithms and Big-O analysis content From Examlet 5:

Recursion Trees

- Finding the closed form using a recursion tree.
- Find a big-O solution for simple recursive definitions

Algorithms

- Towers of Hanoi & Karatsuba’s algorithm
- Find a big-O solution requiring use of the change of base formula
- Analyze Big-O of unfamiliar but fairly simple functions in pseudo-code
- Express algorithms involving loops (possibly nested) using summations
- Express a recursive algorithm’s running time as a recursive definition

NP

- Certain sentences have exponentially many parse trees
- Tower of Hanoi puzzle is in EXP
- NP is the set of problems for which we can easily prove the answer is “yes”
- co-NP is the set of problems for which we can easily prove the answer is “no”
- NP problems can be solved in EXP, but may also be polynomial (P) time solvable
- An “NP-complete” problem solved in polynomial time would imply that any problem in NP can be solved in polynomial time.
- Know common examples of NP-complete problems
- Bipartite graphs are 2-colorable (Polynomial time solution)

Proof by contradiction

📌 **Quizlet Flashcards:** https://quizlet.com/1033386586/cs-173-examlet-6-study-guide-recursion-and-np-concepts-flash-cards/

📌 **Notion Website:** https://therapeutic-profit-af7.notion.site/CS-173-Examlet-6-Study-Guide-Recursion-Trees-Code-Analysis-NP-1da42e50f1858048ade6f51bdae09889?pvs=4

(Recap) Earlier Algorithms & Big O Content

📌 **Review the earlier Algorithms and Big-O analysis content From Examlet 5:**

Action: Recap the Algorithms I, II, and III lecture notes from the lecture schedule

- The Primitive function relationships

$\ll$ - strictly asymptotically less than

O - asymptotically less than or equal to

$\approx$ or Θ - asymptotically similar

- Skill: Be able to prove a primitive function relationship (using inequalities & induction)

- Three main patterns requiring analysis
  - nested loops
  - while loops
  - recursive

Recursion Trees

> 📌 **Finding the closed form using a recursion tree.**
>
> Skill: Given a recursively defined function, find its closed form by drawing a recursion tree and adding up the work at all levels. (Examples would look like those presented in the textbook, e.g. the level sums are constant or increase in a pattern based on the key summations given above.)

To find the closed form of a recursively defined function using a hand-drawn recursion tree you want to set up three columns alongside the drawing of your tree (column 4).

1. The column that you will write the height of your tree at each level
2. The column that you will write the node count at each level in your tree
3. The column that you will write the problem size at each level in your tree (problem size is the input into your recursively defined function at that level i.e

n is the p-size at the first level)

4. **The Drawing**: A visual representation of the nodes at each level in the tree.
    a. The key is to remember that each node represents a call of your recursive function.
        i. So it should be labeled with the amount of work done during that call of the function. (all of the non-recursive terms of that function definition). if the non-recursive term includes n, then you plug in the problem size at height k for n and use that to label each node at height k.

1. height (k)	2. node count - $f(bf, k)$	3. problem size	4) tree visual
0	1	n	draw one node and label it with work done on first call
1	$(branchingfactor)^1$	i.e (n - 1)	for each parent draw bf number of nodes beneath it
2	$(branchingfactor)^2$	i.e $(n-2)$	
...	...	...	...
k	$(branchingfactor)^k$ [this will be the node count at level k]	$(n-k)$ [this will be the p-size as a function of k]	draw leaf level nodes and work at height k

📌

Find a big-O solution for simple recursive definitions

Skill: Find a big-O solution for a simple recursive definition, of the types that we've seen as examples of unrolling and recursion trees.

You find the big-O solution *from memory*, by *unrolling* or by *drawing a recursion tree*.

Professor says it's worth memorizing the big-O running times of common routines so you can use them quickly in higher-level analysis.

Examples of simple recursive definitions that we've seen:

Tower of Hanoi:

- $T(1) = c$
- $T(n) = 2T(n-1) + d$

Runtime: $O(2^n)$

Merging two Sorted Lists:

- $T(1) = c$
- $T(n) = T(n-1) + d$

Runtime: $O(n)$

Divide and Conquer/ Binary Search:

- $T(1) = c$
- $T(n) = T(n/2) + d$

Runtime: $O(\log n)$

Mergesort:

- $T(1) = c$
- $T(2) = 2T(n/2) + d$

Runtime: $O(n \log n)$

Algorithms

📌 Towers of Hanoi & Karatsuba's algorithm

- Be familiar with the overall structure and big-O running times of the following algorithms.
  - Towers of Hanoi solver
  - Karatsuba's algorithm (you won't have to reproduce all the details)

Towers of hanoi solver:

```
01  hanoi(A,B,C: pegs, d_1, d_2 ... d_n: disks)
02          if (n = 1) move d_1 = d_n from A to B.
03          else
04                  hanoi(A,C,B,d_1, d_2, ... d_{n-1})
05                  move d_n from A to B.
06                  hanoi(C,B,A,d_1, d_2, ... d_{n-1})
```

- $T(1) = c$
- $T(n) = 2T(n-1) + d$

Runtime: $O(2^n)$

Karatsuba's algorithm:

- $T(1) = c$
- $T(n) = 3T(n/2) + dn$

Runtime: $O(n^{\log_3 2})$

[Solution uses change of base formula]

📌 Find a big-O solution requiring use of the change of base formula

- Find a big-O solution for slightly harder recursive definitions than the ones for Examlet 5, e.g. requiring use of the change of base formula.

An example of a recursive definition whose analysis requires the change of base formula is Karatsuba's algorithm. It has the form $T(n) = 3T(n/2) + O(n)$.

Solving for the work at leaf nodes:

The **tree height** is $\log_2 n$

*The **number of leaves*** is the branching factor (3) to the power of the height ($\log_2 n$)

$= 3^{log_2 n}$

multiply by c

Solving for the work at non-leaf nodes:

...

See mfleck's "Algorithms 4" notes for visuals and what karatsuba's algo. is for.

Change of base formula:

- $\log_a b = \log_a c * \log_c b$

📌 Analyze Big-O of unfamiliar but fairly simple functions in pseudo-code

Skill: Given an unfamiliar but fairly simple function in pseudo-code, analyze how long it takes using big-O notation. You should be able to analyze nested for loops, recursive functions, and simple examples of while loops.

For analyzing big-O runtime of simple functions it's helpful to know the runtimes of common routines such as (but not limited to) *nested loops, linear search (list traversal), binary search, sorted list merge, mergesort.*

You want to identify the ***dominant term*** of a routine by identifying the big-O running times of each of the code's sub-routines. The dominant term is its big-O running time.

📌

Express algorithms involving loops (possibly nested) using summations

Skill: For an algorithm involving loops (perhaps nested), express its running time using summations.

Nested loop Example:

```
01  closestpair(p_1, ..., p_n) : array of 2D points)
02          best1 = p_1
03          best2 = p_2
04          bestdist = dist(p_1,p_2)
05          for i = 1 to n
06                  for j = 1 to n
07                          newdist = dist(p_i,p_j)
08                          if (i ≠ j and newdist < bestdist)
09                                  best1 = p_i
10                                  best2 = p_j
11                                  bestdist = newdist
12          return (best1, best2)
```

Running time Expressed using summations:

$\sum_{i=1}^{n} \sum_{j=1}^{n} d$, where d is some constant work from lines 07-11

= $n(\sum_{j=1}^{n} d) = n * (n * d) = n^2 d$

[Recall that the sum from 1 to n of any constant c is nc]

So a 2-D nested loop is $O(n^2)$

📌

Express a recursive algorithm's running time as a recursive definition

Skill: Given a recursive algorithm (familiar or unfamiliar) express its running time as a recursive definition.

You'd want to write a recursive definition representative of the pseudo code like:

$T(z) = c$

$T(p_0) = bT(p_1) + w$

$p_0 = n$. z is the base case, b is the branching factor, p_k is the problem size at tree level k. w is the work done at each calling of $T(n)$

To find the closed form using a recursion tree,

You want to :

- identify the base case, z
- identify the height of the algorithm's recursion tree k, by solving $p_k = z$
- identify w (the work done at each call of $T(n)$
  - by analyzing the pseudo-code's structure and sub-components

    [w could be a function of n and/or include other recursive terms]
- Finally you'd find The sum of the work w, done at
  - the leaf level
  - + internal levels of the tree

NP

📌 Certain sentences have exponentially many parse trees

- Know that certain classes of sentences have exponentially many parse trees. (And thus producing all parses of a sentence requires exponential time.)

For example sentences with many prepositional statements. (subject-ambiguous descriptive statements)

I saw a unicorn in the shed, by the apple tree with a red hat.

📌 Tower of Hanoi puzzle is in EXP

- Know that the Towers of Hanoi puzzle has been proved to require exponential time.

📌 NP is the set of problems for which we can easily prove the answer is “yes”

- Know that **NP** is the set of problems for which we can quickly (polynomial time) justify "yes" answers.

(P, NP, EXP) technically contain only "decision problems." A decision problem is a problem where you are given some input and asked to produce a yes/no answer. Informally, people extend this idea to other sorts of problems, because an algorithm that delivers more interesting answer (e.g. the chromatic number of a graph) can typically be built on top of an algorithm for the corresponding decision problem.

📌 co-NP is the set of problems for which we can easily prove the answer is “no”

- Know that **co-NP** is the set of problems for which we can quickly (polynomial time) justify "no" answers.

📌 NP problems can be solved in EXP, but may also be polynomial (P) time solvable

- Skill: Know that problems in NP can be solved in exponential time, but it's not known whether they can be solved in polynomial time.

the existence of Polynomial time solutions for them have not yet been disproven.

📌 An “NP-complete” problem solved in polynomial time would imply that any problem in NP can be solved in polynomial time.

- Skill: Know what an NP-complete problem is: a problem in NP for which a polynomial-time algorithm would imply that **any** problem in NP can be solved in polynomial time.

that is NP and P would be equal.

📌 Know common examples of NP-complete problems

- Skill: Know some examples of NP-complete problems: graph colorability, circuit satisfiability (Circuit SAT), propositional logic satisfiability, marker making, the travelling salesman problem.

📌 Bipartite graphs are 2-colorable (Polynomial time solution)

- Skill: Know that you can decide in polynomial time whether a graph is 2-colorable (aka bipartite).

Proof by contradiction

- Skill: Understand proof by contradiction and when it’s useful (i.e to prove existential claims)