CS2040S
Data Structures and Algorithms
(e-learning edition)

Welcome!

Announcements

Midterm : Monday March 9

- Location TBA (MPSH)
- < 50 students / room
- 9 rooms

Bring to quiz:

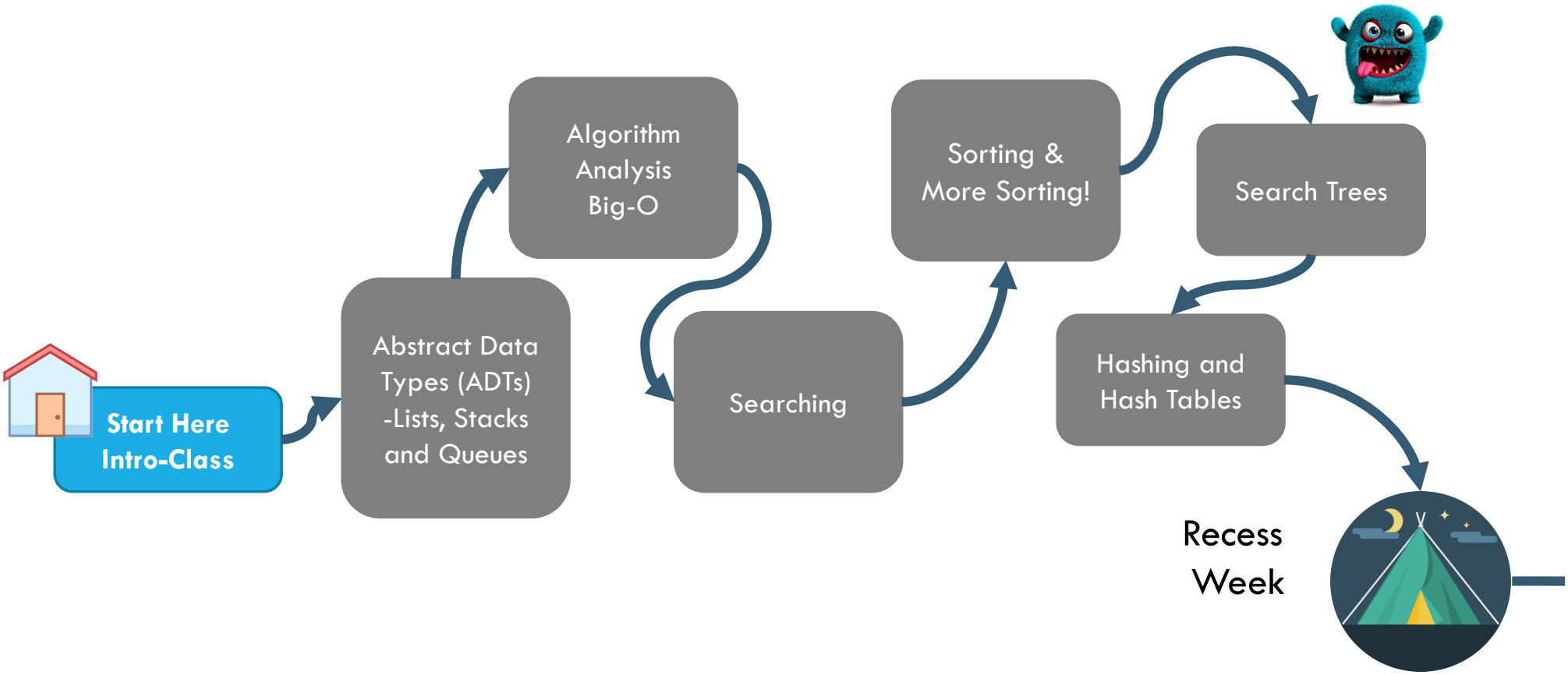
- One sheet of paper with any notes you like.
- Pens/pencils.
- You may not use anything else.

Announcements

Practice Quizzes (from 2015-17):

- Posted on Coursemology
- From CS2020
  - Beware: CS2020 != CS2040S
  - Different subset of material
- Discussed in tutorials

What have we done so far?



Part I: Organizing your Data

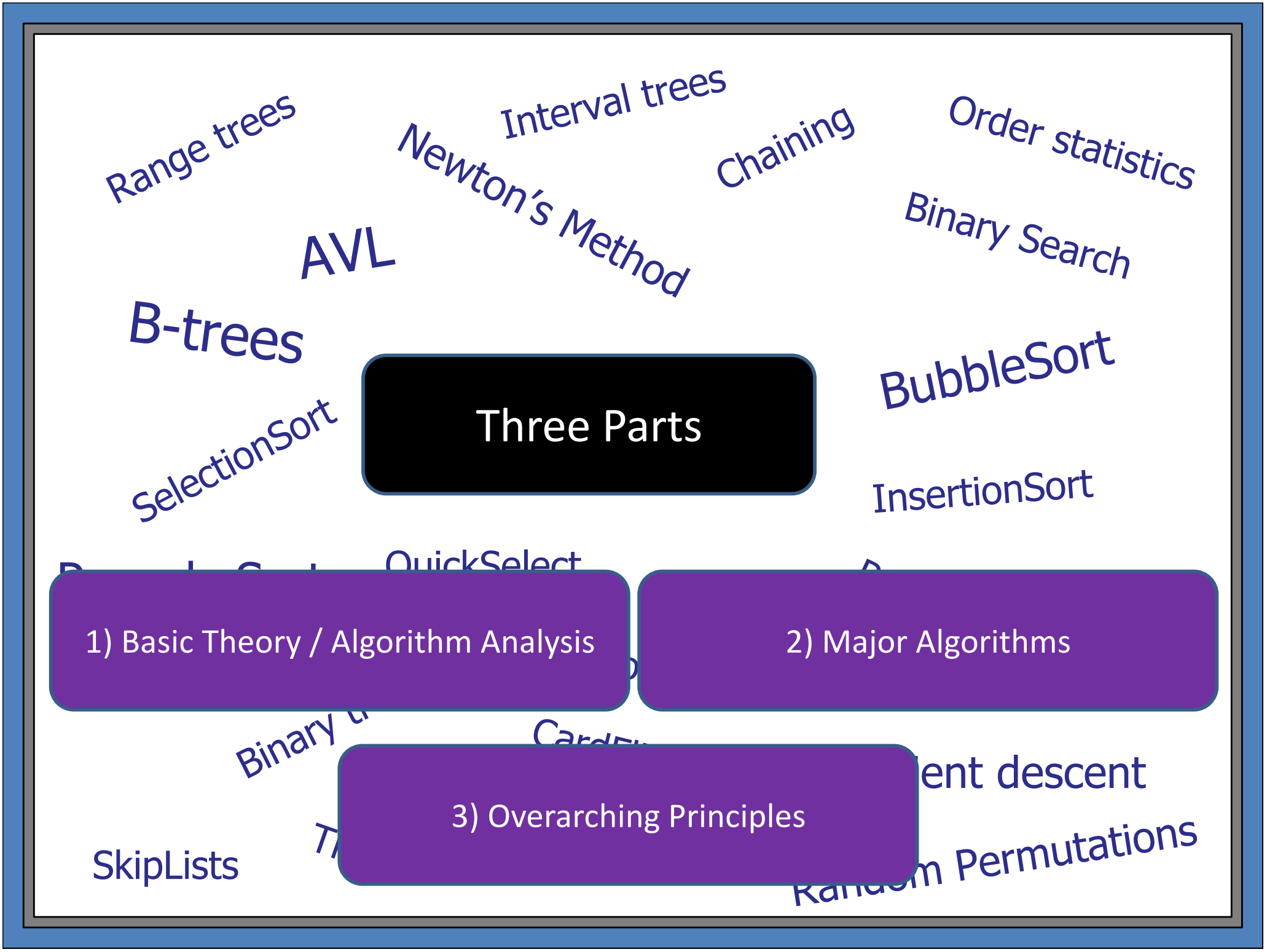

Range trees
Interval trees
Chaining
Order statistics
Newton's Method
Binary Search
AVL
B-trees
So much material?
BubbleSort
SelectionSort
InsertionSort
PancakeSort
QuickSelect
QuickSort
ReversalSort
BogoSort
MergeSort
Binary tree
CardFlipTrees
Gradient descent
Hash table
Tries
SkipLists
Random Permutations


Range trees
Interval trees
Chaining
Order statistics
Newton's Method
Binary Search
AVL
B-trees
So much material?
BubbleSort
SelectionSort
InsertionSort
PancakeSort
QuickSelect
ReversalSort
QuickSort
BogoSort
MergeSort
Binary tree
CardFlipTrees
Gradient descent
Hash table
Tries
SkipLists
Random Permutations


Range trees
Interval trees
Chaining
Order statistics
Newton's Method
Binary Search
AVL
B-trees
So much material?
BubbleSort
SelectionSort
InsertionSort
PancakeSort
QuickSelect
QuickSort
ReversalSort
BogoSort
MergeSort
Binary tree
CardFlipTrees
Gradient descent
Hash table
Tries
SkipLists
Random Permutations


Range trees
Interval trees
Chaining
Order statistics
Newton's Method
Binary Search
AVL
B-trees
So much material?
BubbleSort
SelectionSort
InsertionSort
PancakeSort
QuickSelect
ReversalSort
QuickSort
BogoSort
MergeSort
Binary tree
CardFlipTrees
Gradient descent
Hash table
Tries
SkipLists
Random Permutations


Range trees
Interval trees
Chaining
Order statistics
Newton's Method
Binary Search
AVL
B-trees
So much material?
BubbleSort
SelectionSort
InsertionSort
PancakeSort
QuickSelect
ReversalSort
QuickSort
BogoSort
MergeSort
Binary tree
CardFlipTrees
Gradient descent
Hash table
Tries
SkipLists
Random Permutations


Range trees
Interval trees
Chaining
Order statistics
Newton's Method
AVL
Binary Search
B-trees
BubbleSort
SelectionSort
Three Parts
InsertionSort
QuickSelect
1) Basic Theory / Algorithm Analysis
2) Major Algorithms
3) Overarching Principles
SkipLists
Random Permutations

Warning:

Today I am going to review some of the most important topics.

This will not be comprehensive.

1) Basic Theory / Algorithm Analysis

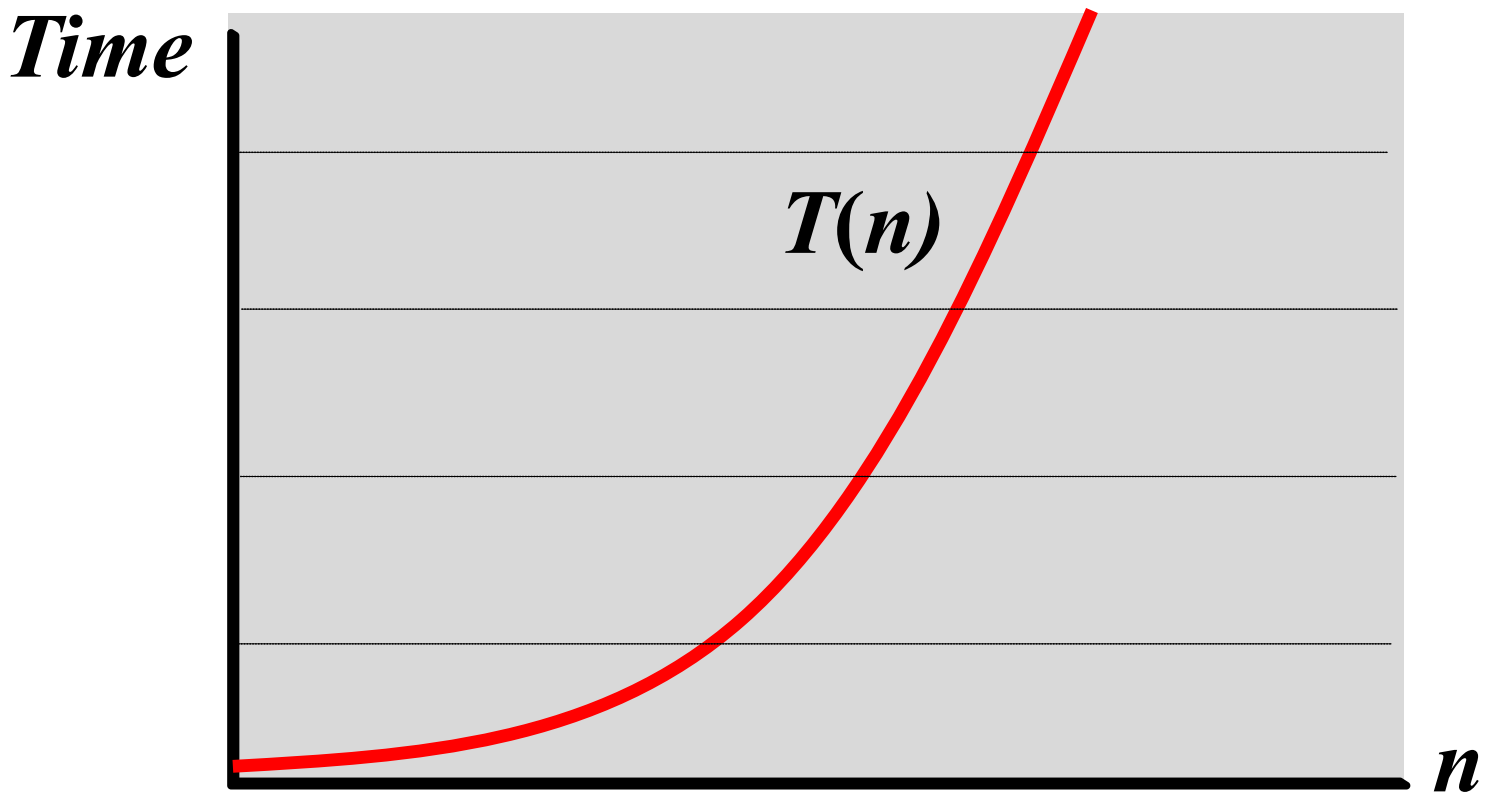
Key Topics

- Asymptotic notation (big-O, etc.)
- Simple recurrences
- Asymptotic analysis
- Basic probability (e.g., CS1231 review)

Big-O Notation

How does an algorithm scale?

- For large inputs, what is the running time?
- $T(n)$ = running time on inputs of size n



Big-O Notation

Definition: $T(n) = O(f(n))$ if T grows no faster than f

$T(n) = O(f(n))$ if:

- there exists a constant $c > 0$
- there exists a constant $n_0 > 0$

such that for all $n > n_0$:

$T(n) \leq c\, f(n)$

Example

$T(n)$	$f(n)$	big-O
$T(n) = 1000n$	$f(n) = n$	$T(n) = \mathrm{O}(n)$
$T(n) = 1000n$	$f(n) = n^2$	$T(n) = \mathrm{O}(n^2)$
$T(n) = n^2$	$f(n) = n$	$T(n) \neq \mathrm{O}(n)$
$T(\mathrm{n}) = 13n^2 + n$	$f(n) = n^2$	$T(n) = \mathrm{O}(n^2)$

Simple recurrences

Recurrences we have seen a lot:

$$T(n) = 2T(n/2) + n$$

$$T(n) = 2T(n/2) + 1$$

$$T(n) = T(n/2) + 1$$

$$T(n) = T(n/2) + n$$

Et cetera

Simple recurrences

Recurrences we have seen a lot:

$$T(n) = 2T(n/2) + n$$

$$T(n) = 2T(n/2) + 1$$

$$T(n) = T(n/2) + 1$$

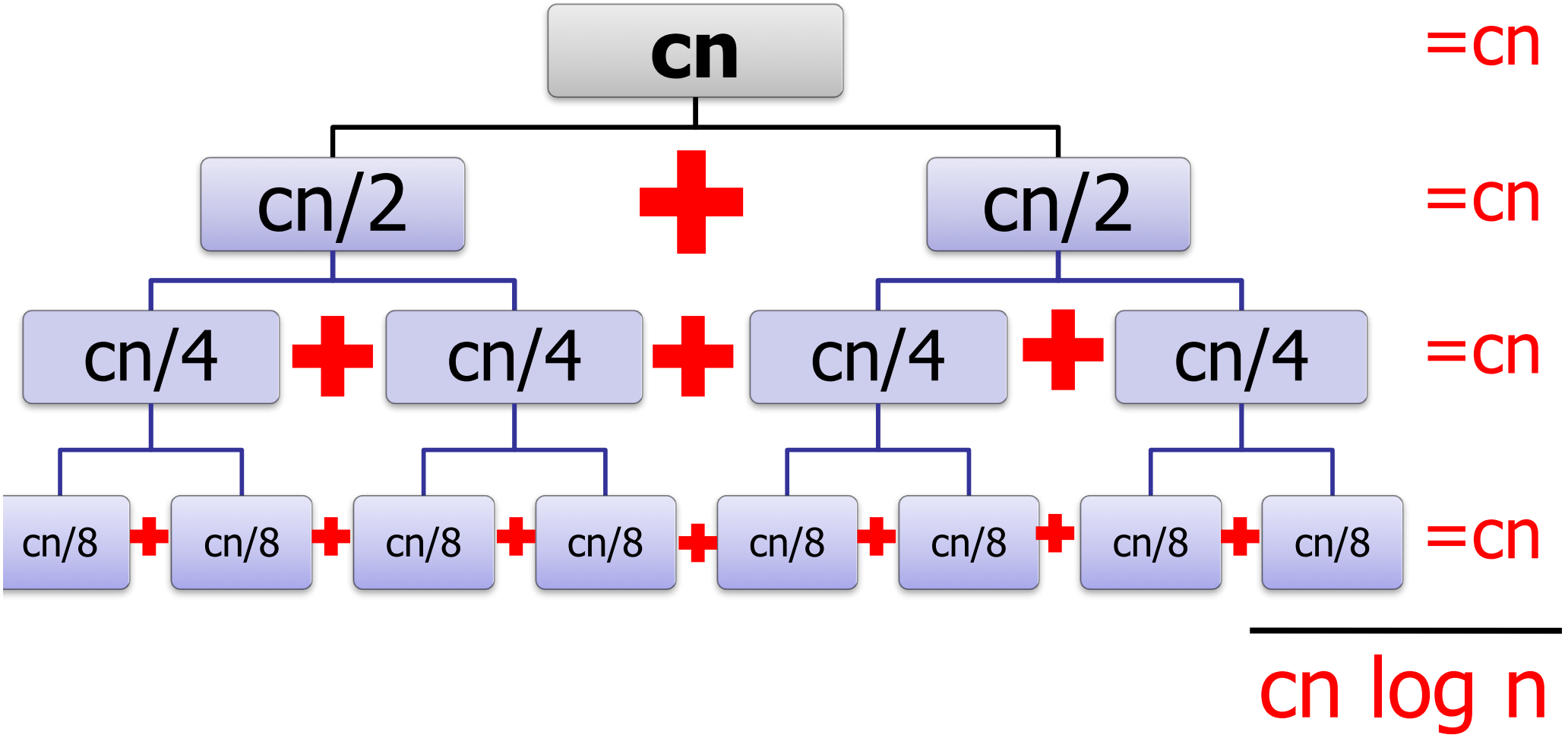
$$T(n) = T(n/2) -$$

Et cetera

1. Know the solution.
2. Be able to draw the recurrence tree.
3. Be able to substitute.

Example: MergeSort Analysis

$T(n) = 2T(n/2) + cn$



Guess: T(n) = c·n log n

T(1) = c

T(x) = c·x log x for all x < n.

$$
\begin{aligned}
T(n) &= 2T(n/2) + cn \\
&= 2(c(n/2)\log(n/2)) + cn \\
&= cn\log(n/2) + cn \\
&= cn\log(n) - cn\log(2) + cn \\
&= cn\log(n)
\end{aligned}
$$

Induction:
It works!

Recurrence being analyzed:

T(n) = 2T(n/2) + c·n

T(1) = c

Algorithm Analysis

Asymptotics?

```
void pushAll(int k) {
   for (int i=0;
         i<= 100*k;
         i++)
   {
       stack.push(i);
   }
}
```

O(k)

```
void pushAdd(int k) {
   for (int i=0; i<= k; i++)
   {
      for (int j=0; j<= k; j++){
            stack.push(i+j);
      }
   }
}
```

$O(k^2)$

Asymptotics on problem sets

Many questions asked:

What is the asymptotic running time of your program?

Many got that wrong (even when their program was right).

Algorithm Analysis

Example:

```
void sum(int k, int[] intArray) {
   int total=0;
   for (int i=0; i<= k; i++){
        total = total + intArray[i];
   }
   return total;
}
```

Total: 1 + 1 + (k+1) + 3k + 1 = 4k+4 = O(k)

Rules

Loops

cost = (# iterations)x(max cost of one iteration)

```
int sum(int k, int[] intArray) {
   int total=0;
   for (int i=0; i<= k; i++){
       total = total + intArray[i];
   }
   return total;
}
```

Rules

Nested Loops

cost = (# iterations)(max cost of one iteration)

```
int sum(int k, int[] intArray) {
    int total=0;
    for (int i=0; i<= k; i++){
      for (int j=0; j<= k; j++){
          total = total + intArray[i];
      }
   }
   return total;
}
```

Rules

Sequential statements

cost = (cost of first) + (cost of second)

```
int sum(int k, int[] intArray) {
   for (int i=0; i<= k; i++)
       intArray[i] = k;
   for (int j =0; j<= k; j++)
       total = total + intArray[i];
   return total;
}
```

Rules

if / else statements

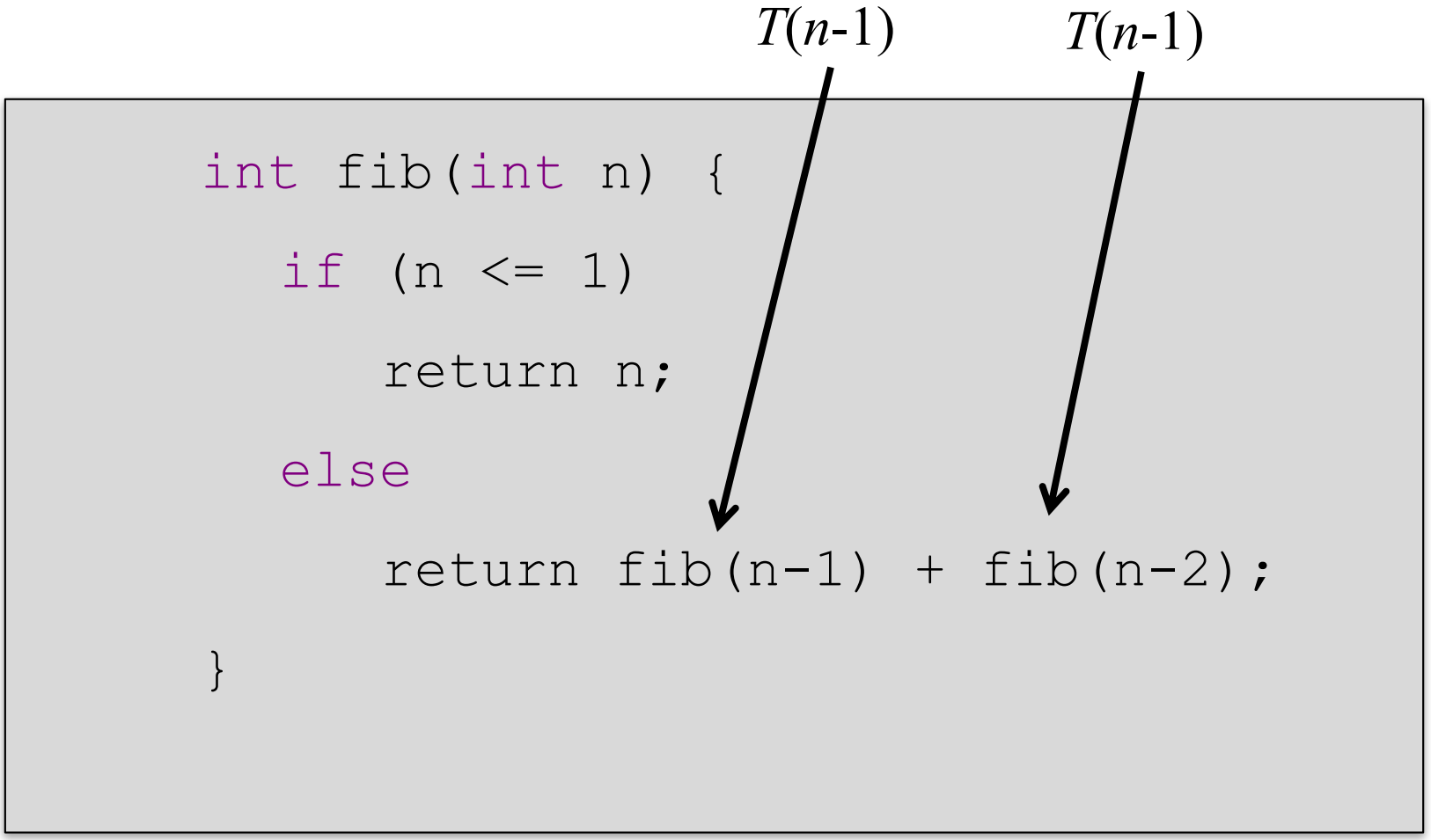
cost = 2max(cost of first, cost of second)

>= (cost of first) + (cost of second)

```
void sum(int k, int[] intArray) {
   if (k > 100)
       doExpensiveOperation();
   else
       doCheapOperation();
   return;
}
```

Recurrences

$$T(n) = 1 + T(n - 1) + T(n - 2)$$
$$= O(2^n)$$



What is the running time?

```
for (int i = 0; i<n; i++)
    for (int j = 0; j<i; j++)
        store[i] = i + j;
```

Probability Theory (CS1231)

Probability Theory

Flipping a coin:

- Pr(heads) = ½
- Pr(tails) = ½

Coin flips are independent:

- Pr(heads → heads) = ½ * ½ = ¼
- Pr(heads → tails → heads) = ½ * ½ * ½ = 1/8

You flip a coin 8 times. Which is more likely?

a. 4 heads, followed by 4 tails
b. 8 heads in a row
c. Same
d. Incomparable

Probability Theory

Events **A**, **B**:

- Pr(**A**), Pr(**B**)
- **A** and **B** are independent

  (e.g., unrelated random coin flips)

Then:

- Pr(**A** and **B**) = Pr(**A**)Pr(**B**)

Probability Theory

Set of outcomes for $X = (e_1, e_2, e_3, \ldots, e_k)$:

- $Pr(e_1) = p_1$
- $Pr(e_2) = p_2$
- …
- $Pr(e_k) = p_k$

Expected outcome:

$E[X] = e_1p_1 + e_2p_2 + \ldots + e_kp_k$

Probability Theory

Expected value:

- Weighted average

Example: event **A** has two outcomes:

- Pr(**A** = 12) = ¼
- Pr(**A** = 60) = ¾

Expected value of A:

E[A] = (¼)12 + (¾)60 = 48

Probability Theory

Flipping a coin:

- Pr(heads) = ½
- Pr(tails) = ½

In two coin flips: I expect one heads.

Probability Theory

Define event **A**:

- **A** = number of heads in two coin flips

In two coin flips: I expect one heads.

- Pr(heads, heads) = ¼ — 2 * ¼ = ½
- Pr(heads, tails) = ¼ — 1 * ¼ = ¼
- Pr(tails, heads) = ¼ — 1 * ¼ = ¼
- Pr(tails, tails) = ¼ — 0 * ¼ = 0

1

Probability Theory

Linearity of Expectation:

- $E[A + B] = E[A] + E[B]$

Example:

- A = # heads in 2 coin flips: $E[A] = 1$
- B = # heads in 2 coin flips: $A[B] = 1$
- A + B = # heads in 4 coin flips

$E[A+B] = E[A] + E[B] = 1 + 1 = 2$

Probability Theory

Flipping an (unfair) coin:

- Pr(heads) = p
- Pr(tails) = (1 – p)

How many flips to get at least one head?

$$\mathbf{E}[X] = (p)(1) + (1 - p)(1 + \mathbf{E}[X])$$

How many more flips to get a head?

Idea: If I flip "tails," the expected number of additional flips to get a "heads" is still **E**[X]!!

Probability Theory

Flipping an (unfair) coin:

- Pr(heads) = p
- Pr(tails) = (1 – p)

How many flips to get at least one head?

$$\mathbf{E}[X] = (p)(1) + (1 - p)(1 + \mathbf{E}[X])$$

$$= p + 1 - p + 1\mathbf{E}[X] - p\mathbf{E}[X]$$

$$p\mathbf{E}[X] = 1$$

$$\mathbf{E}[X] = 1/p$$

Probability Theory

Flipping an (unfair) coin:

- Pr(heads) = p
- Pr(tails) = (1 – p)

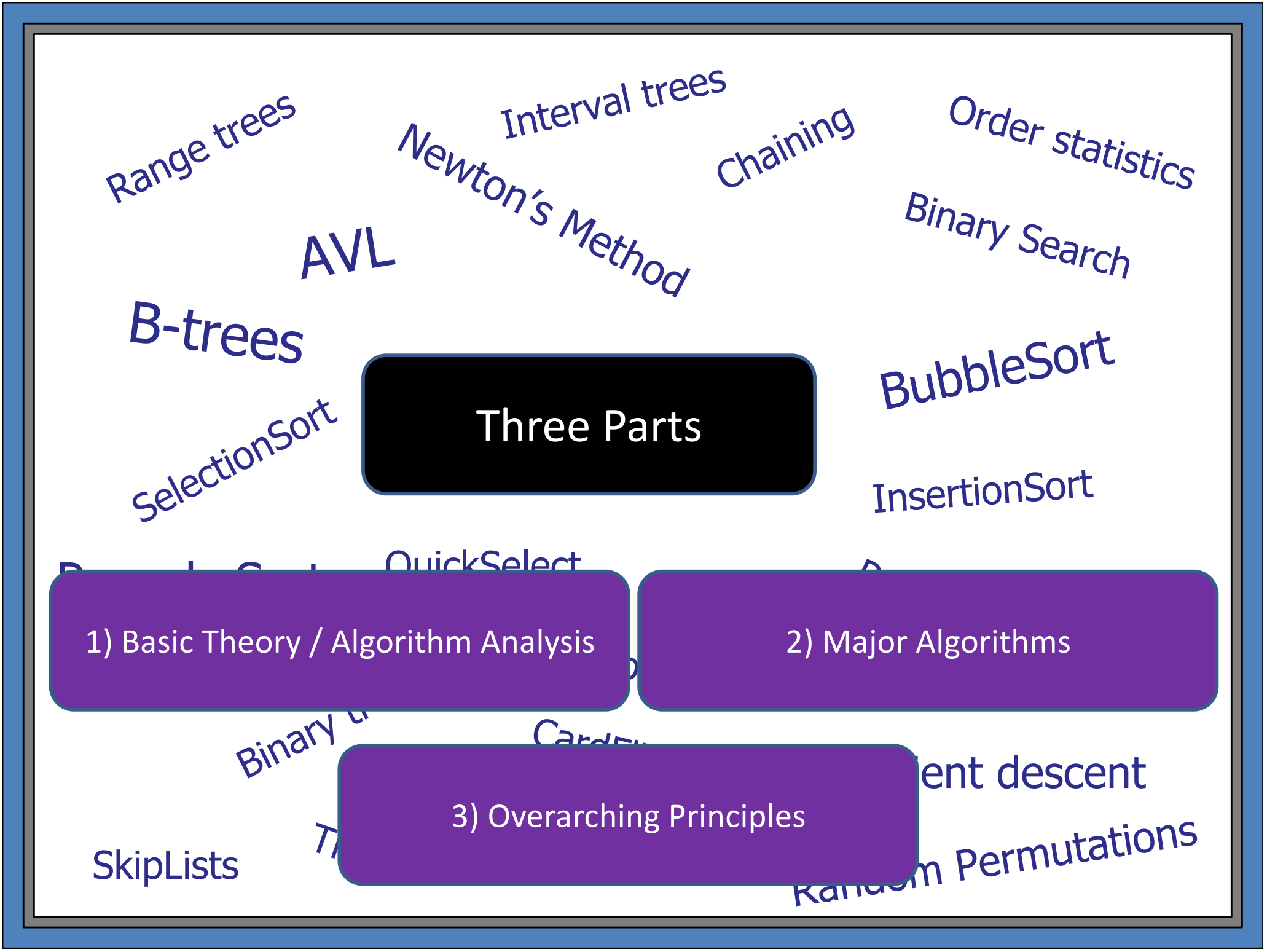
How many flips to get at least one head?

If p = ½, the expected number of flips to get one head equals:

$$\mathbf{E}[X] = 1/p = 1/½ = 2$$

Key Topics

- Asymptotic notation (big-O, etc.)
- Simple recurrences
- Asymptotic analysis
- Basic probability (e.g., CS1231 review)


Range trees
Interval trees
Chaining
Order statistics
Newton's Method
AVL
Binary Search
B-trees
BubbleSort
SelectionSort
Three Parts
InsertionSort
QuickSelect
1) Basic Theory / Algorithm Analysis
2) Major Algorithms
3) Overarching Principles
SkipLists
Random Permutations

Key Algorithms

- Binary Search
- Sorting
- Balanced binary tree (AVL tree)
- Hash table (with chaining, linear probing)

Searching

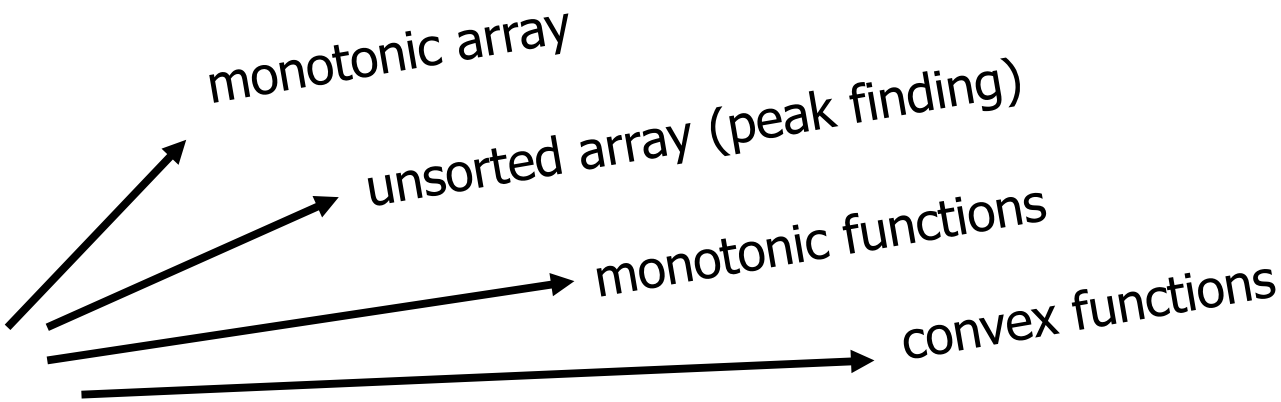
Binary Search

Reduce-and-conquer: make the problem smaller at every step.

At every step, move toward your target.

Key question: how far to move?

Searching



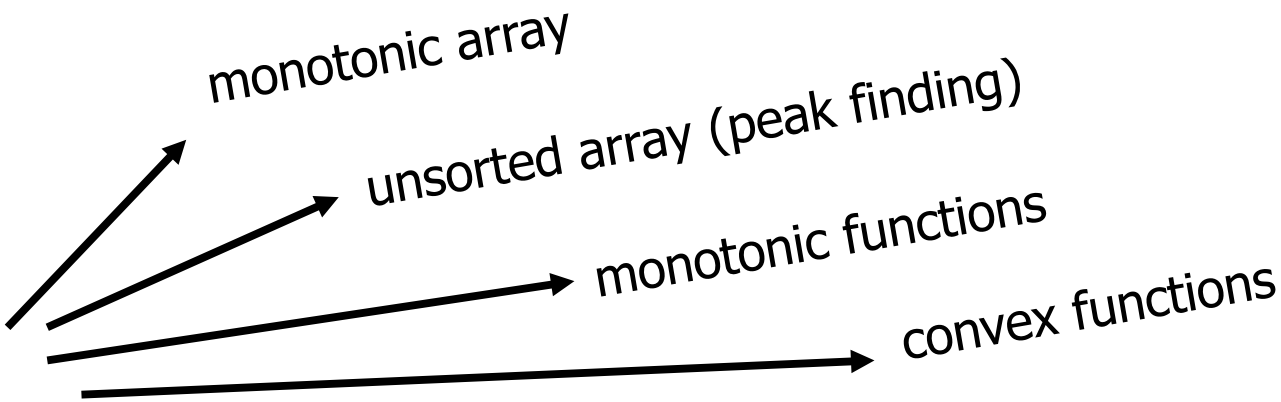
Binary Search

Reduce-and-conquer: make the problem smaller at every step.

At every step, move toward your target.

Key question: how far to move?

Searching



Binary Search

Reduce-and-conquer: make the problem smaller at every step.

At every step, move toward your target.

Key question: how far to move?

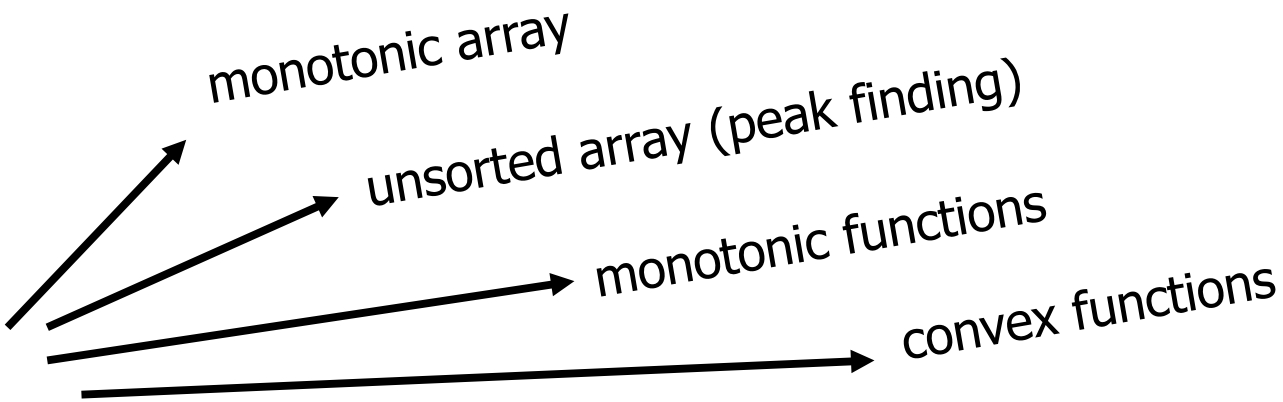
Peak Finding

QuickSelect

Newton's Method

Gradient Descent

Searching



Binary Search

Reduce-and-conquer: make the problem smaller at every step.

At every step, move toward your target.

Key question: how far to move?

Peak Finding	QuickSelect	Newton's Method	Gradient Descent
Eliminate ½ or ¾ of the matrix in every step.	Eliminate approximately ½ the array in each iteration	Always move downhill.	Always move downhill.
Use matrix bounds to choose how much to eliminate.	Use matrix bounds to choose how much to eliminate.	Use first and second derivative to figure out how far to move.	Use first derivative to figure out how far to move.

Sorting

BubbleSort

Look for inversions to fix.

SelectionSort

Find the next item to put in place.

InsertionSort

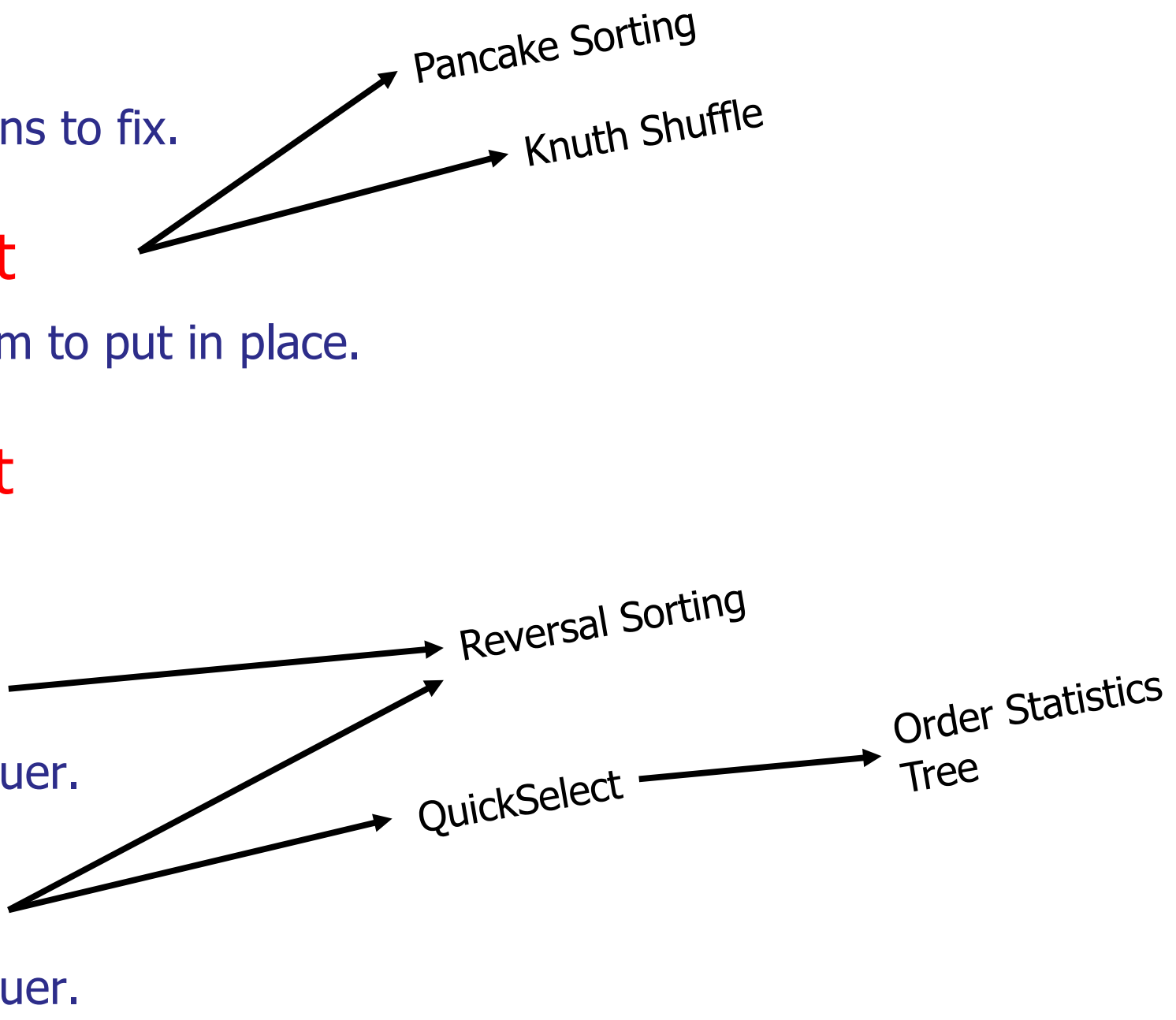
Sort the prefix.

MergeSort

Divide-and-Conquer.

QuickSort

Divide-and-Conquer.



Trees

Binary trees

Unbalanced, in-order, dynamic, traversable

AVL trees

Like binary trees, but height-balanced.

(a,b)-trees

Degree [a,b], balanced, uniform height

SkipLists

Balanced, randomized, list-based

Tries

Unbalanced, string-based

Trees

Binary trees

Unbalanced, in-order, dynamic, traversable

AVL trees

Like binary trees, but height-balanced.

Key idea:
Maintain height balance using rotatinos

(a,b)-trees

Degree [a,b], balanced, uniform height

Key idea:
All leaves have same height.

SkipLists

Balanced, randomized, list-based

Key idea:
Random height makes it balanced.

Tries

Unbalanced, string-based

Trees

Binary trees

Unbalanced, in-order, dynamic, traversable

AVL trees

Like binary trees, but height-balanced.

(a,b)-trees

Degree [a,b], balanced, uniform height

SkipLists

Balanced, randomized, list-based

Tries

Unbalanced, string-based

Design decisions:

Data stored at internal nodes or only at the leaves?

Which balance method?

- Height-balance
- Weight-balance
- Uniform-height
- Random-height

Trees

Summarizing data:

- Range trees
- Interval trees
- Order statistics trees
- Merkle trees
- Etc…

Hash Tables

Chaining

Resolve collisions using linked lists

Open addressing

Resolve collisions by finding a new slot

Finger print hash tables

Use a hash to identify an item

Bloom filters

Better finger print hash tables

Design decisions:

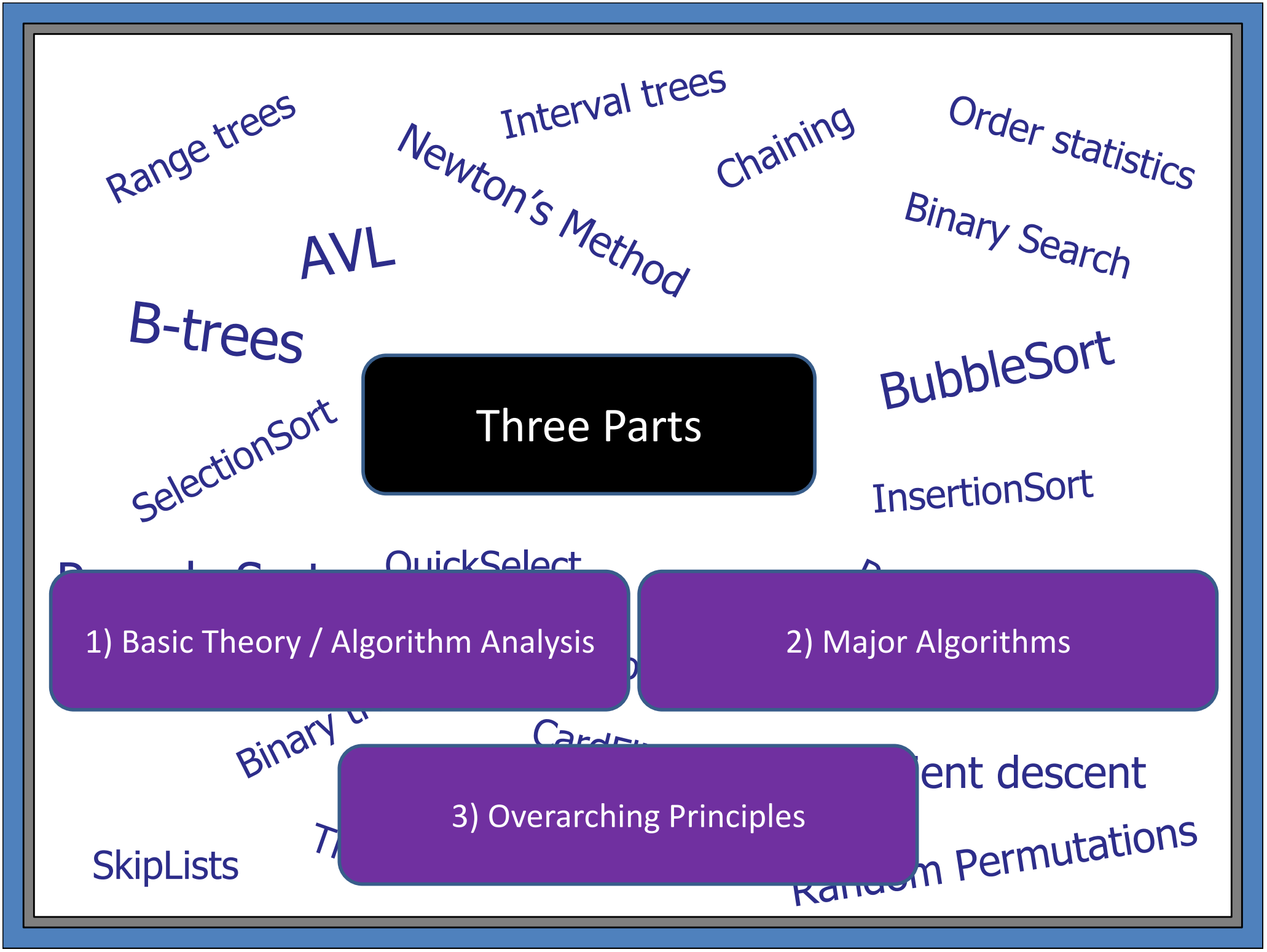
How do you resolve collision?

How big is your table?

Do you have false positives?

Key Algorithms

- Binary Search
- Sorting
- Balanced binary tree (AVL tree)
- Hash table (with chaining, linear probing)


Range trees
Interval trees
Chaining
Order statistics
Newton's Method
AVL
Binary Search
B-trees
BubbleSort
SelectionSort
Three Parts
InsertionSort
QuickSelect
1) Basic Theory / Algorithm Analysis
2) Major Algorithms
3) Overarching Principles
SkipLists
Random Permutations

Key Ideas

- Basic stratgies
- Invariants
- Trade-offs
- Augmenting data structures

Basic strategies for solving problems

Reduce-and-conquer

E.g., binary search

Move in the right direction

E.g., gradient descent

Divide-and-conquer

E.g., MergeSort

Maintaining an invariant

E.g., AVL trees

E.g., keep your data sorted

Augment an existing data structure

E.g., AVL trees

Key Ideas

- Basic stratgies
- Invariants
- Trade-offs
- Augmenting data structures

Two important questions:

How do you understand what an algorithm is doing?

How do you show that an algorithm is correct?

Invariants

What is an invariant?

A property or state-relation that is always true.

What is a loop invariant?

A property or state-relation that is true at the beginning or end of every iteration of a loop.

Invariants

Examples:

BinarySearch:

- At the end of iteration j, the active search region between begin and end is size at most $n/2^j$, and the item (if it exists) is in that region.

BubbleSort:

- At the end of iteration j, the last j slots in the array contain the largest j elements in the array in sorted order.

InsertionSort:

- At the end of iteration j, the first j+1 slots in the array are sorted.

Invariants

Explains how the algorithm works

Examples:

BinarySearch:

- At the end of iteration j, the active search region between begin and end is size at most $n/2^j$, and the item (if it exists) is in that region.

BubbleSort:

- At the end of iteration j, the last j slots in the array contain the largest j elements in the array in sorted order.

InsertionSort:

- At the end of iteration j, the first j+1 slots in the array are sorted.

Invariants

Examples:

BinarySearch: ➔ after log(n) iterations, search is done and finds items (if it exists).

- At the end of iteration j, the active search region between begin and end is size at most $n/2^j$, and the item (if it exists) is in that region.

BubbleSort: ➔ after n iterations, whole array is sorted.

- At the end of iteration j, the last j slots in the array contain the largest j elements in the array in sorted order.

InsertionSort: ➔ after n iterations, whole array is sorted.

- At the end of iteration j, the first j+1 slots in the array are sorted.

Proves that the algorithms works.

Invariants

More examples:

AVL trees:

- After every operation, every node is height balanced.
- The nodes satisfy the BST-order property.

(a,b)-trees:

- After every operation, every leaf has same depth.
- Every node has degree in the range [a,b].
- Nodes satisfy tree-order property.

Skip list:

- Every key is in list 0.
- Lists are in sorted order.

Invariants

More examples:

AVL trees: ➔ tree is height O(log n) and search works.

- After every operation, every node is height balanced.
- The nodes satisfy the BST-order property.

(a,b)-trees: ➔ tree is height $O(\log_a n)$ and search works.

- After every operation, every leaf has same depth.
- Every node has degree in the range [a,b].
- Nodes satisfy tree-order property.

Skip list: ➔ search works.

- Every key is in list 0.
- Lists are in sorted order.

To understand algorithms:

For every algorithm in the class, identify all of its important invariants.

Key Ideas

- Invariants
- Trade-offs
- Augmenting data structures

Trade-offs

Which data structure should you use?

Unsorted data:

- Storing and searching?
- Inserting and deleting?
- Handling order statistics queries?

Sorted data:

- Searching?
- Inserting and deleting?
- Enumerating items in order?
- Finding the median element?

Trade-offs

What are the strengths and weaknesses of each?

Linked lists:

- Fast insertion

Arrays:

- Efficient scanning
- Binary search
- Space efficient

Balanced Trees:

- Dynamic data structures
- Data summarization
- Predecessor / successor / enumeration

(a,b)-trees:

- All the advantages of balanced trees
- And cache-efficiency.

Hash tables:

- Fast insertion
- Fast lookup
- Space efficient

Fingerprint / Bloom filters:

- Very space efficient
- Fast lookup
- False positives

Simple Problem: Searching

Given:

a collection of n items

Find:

the largest item in the collection

Algorithm 1.

1. Sort the collection.
2. Return the largest item.

Simple Problem: Searching

Given:

a collection of n items

Find:

the largest item in the collection

Algorithm 1.

1. Sort the collection.
2. Return the largest item.

Algorithm 2.

1. Iterate through the collection.
2. Return the largest item.

Simple Problem: Searching

Given:

a collection of n items

Find:

the largest item in the collection

Algorithm 1.

1. Sort the collection.
2. Return the largest item.

$$O(n \log n)$$

Algorithm 2.

1. Iterate through the collection.
2. Return the largest item.

Simple Problem: Searching

Given:

a collection of n items

Find:

the largest item in the collection

Algorithm 1.

1. Sort the collection.
2. Return the largest item.

$O(n \log n)$

Algorithm 2.

1. Iterate through the collection.
2. Return the largest item.

$O(n)$

Simple Problem: Searching

Given:

a collection of n items

Find:

the largest k items

in the collection

Algorithm 1.

1. Sort the collection.
2. Return the largest k items.

Algorithm 2.

1. Iterate through the collection. Store the largest k items seen.
2. Return the largest k items.

Simple Problem: Searching

Extra memory: 1

Time: $O(n \log n + k)$

Algorithm 1.
1. Sort the collection.
2. Return the largest k items.

Extra memory: k

Time: $O(nk)$

Algorithm 2.
1. Iterate through the collection. Store the largest k items seen.
2. Return the largest k items.

Simple Problem: Searching

Extra memory: 1

Time: $O(n \log n + k)$

Algorithm 1.
1. Sort the collection.
2. Return the largest k items.

Extra memory: k

Time: $O(nk)$

Algorithm 2.
1. Iterate through the collection. Store the largest k items seen.
2. Return the largest k items.

Better solution: trees.

When to use a Symbol Table

Example 1: Pilot Scheduling

1. Check to see if feasible to schedule at time *t*.

   No two airplanes can land with 3 minutes of each other.

2. Find schedule of pilot *p*.

   Get a list of all the planes that are being flown by a specified pilot.

Which can be efficiently solved with a symbol table?

1. Both: scheduling and pilots info.
2. Only scheduling.
3. Only pilot info.
4. Neither.

Key Ideas

- Invariants
- Trade-offs
- Augmenting data structures

How to solve new problems?

1) Start from scratch and design a new data structure.

2) Augment an existing data structure.

How to solve new problems?

1) Start from scratch and design a new data structure.

2) Augment an existing data structure.

much more common

Augmenting data structures

What do you get when you augment a … ?

Array:

- Key-value pairs.
- Multidimensional data.
- Cross-linked data structures

Tree:

- Techniques for summarizing data
- Dynamic data structures
- Databases

Order statistics

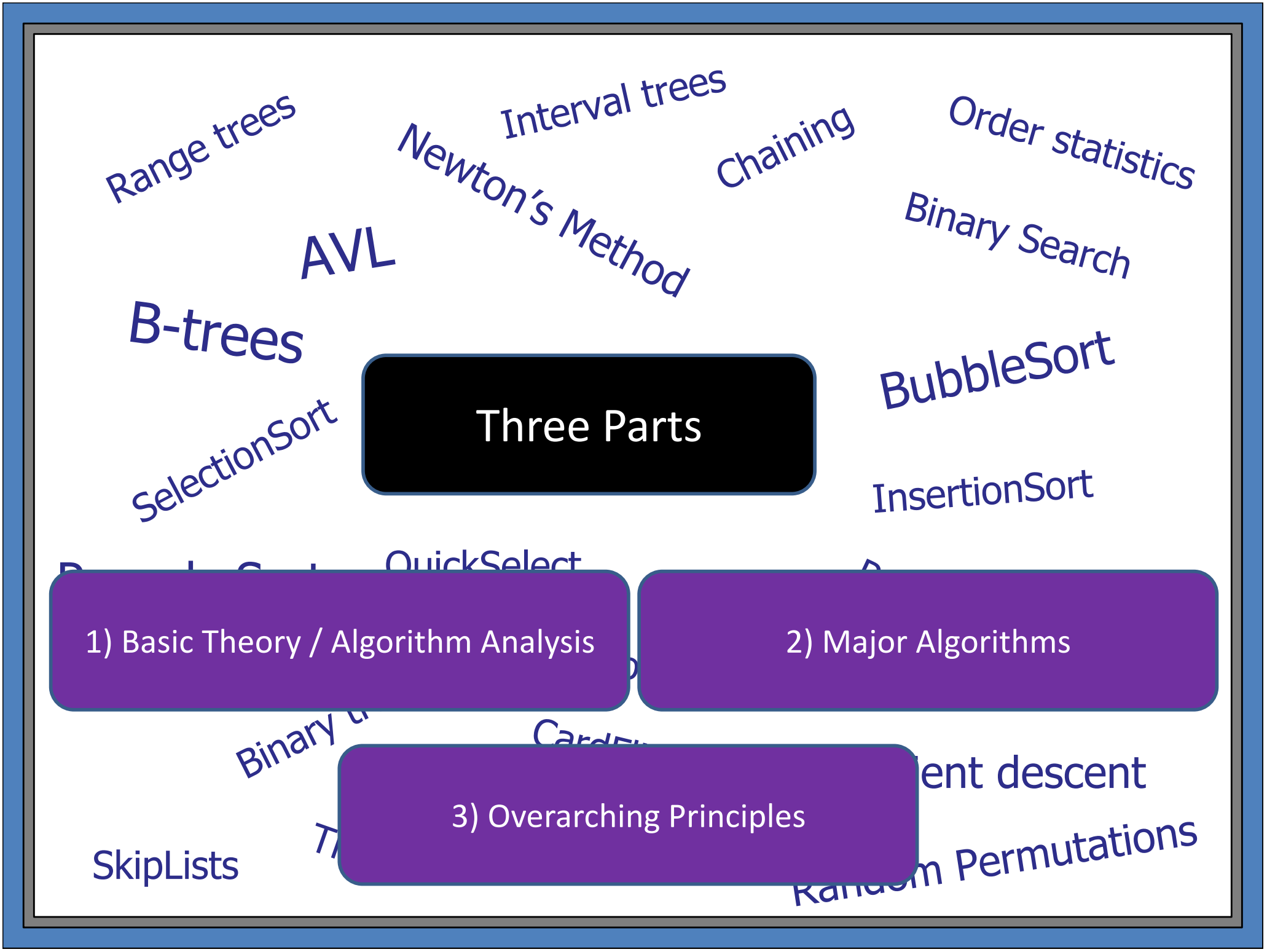
Interval trees

Range trees

Merkle trees

Trie and Hash Tables:

- All sorts of interesting things… ☺


Range trees
Interval trees
Chaining
Order statistics
Newton's Method
AVL
Binary Search
B-trees
BubbleSort
SelectionSort
Three Parts
InsertionSort
QuickSelect
1) Basic Theory / Algorithm Analysis
2) Major Algorithms
3) Overarching Principles
SkipLists
Random Permutations

A bit of advice…

Quiz Advice:

A bit of advice…

Quiz Advice:

Get the maximum number of points you can.

A bit of advice…

Quiz Advice:

Get the maximum number of points you can.

- Do not leave easy questions blank.
- Give a simple solution for partial credit.
- Bypass questions instead of getting stuck.
- Show me what you know.

A bit of advice…

Quiz Advice:

Be as clear as possible.

- Do not be ambiguous.
- Circle your final answer (if it is unclear).
- Cross out incorrect answers.
- Write neatly.

A bit of advice…

Quiz Advice:

State your assumptions.

- If the question is ambiguous, state precisely what you are assuming.
- If your assumptions are reasonable, and your answer is correct subject to those assumptions, you will (most likely) get full credit!

A bit of advice…

Quiz Advice:

Review the basics:

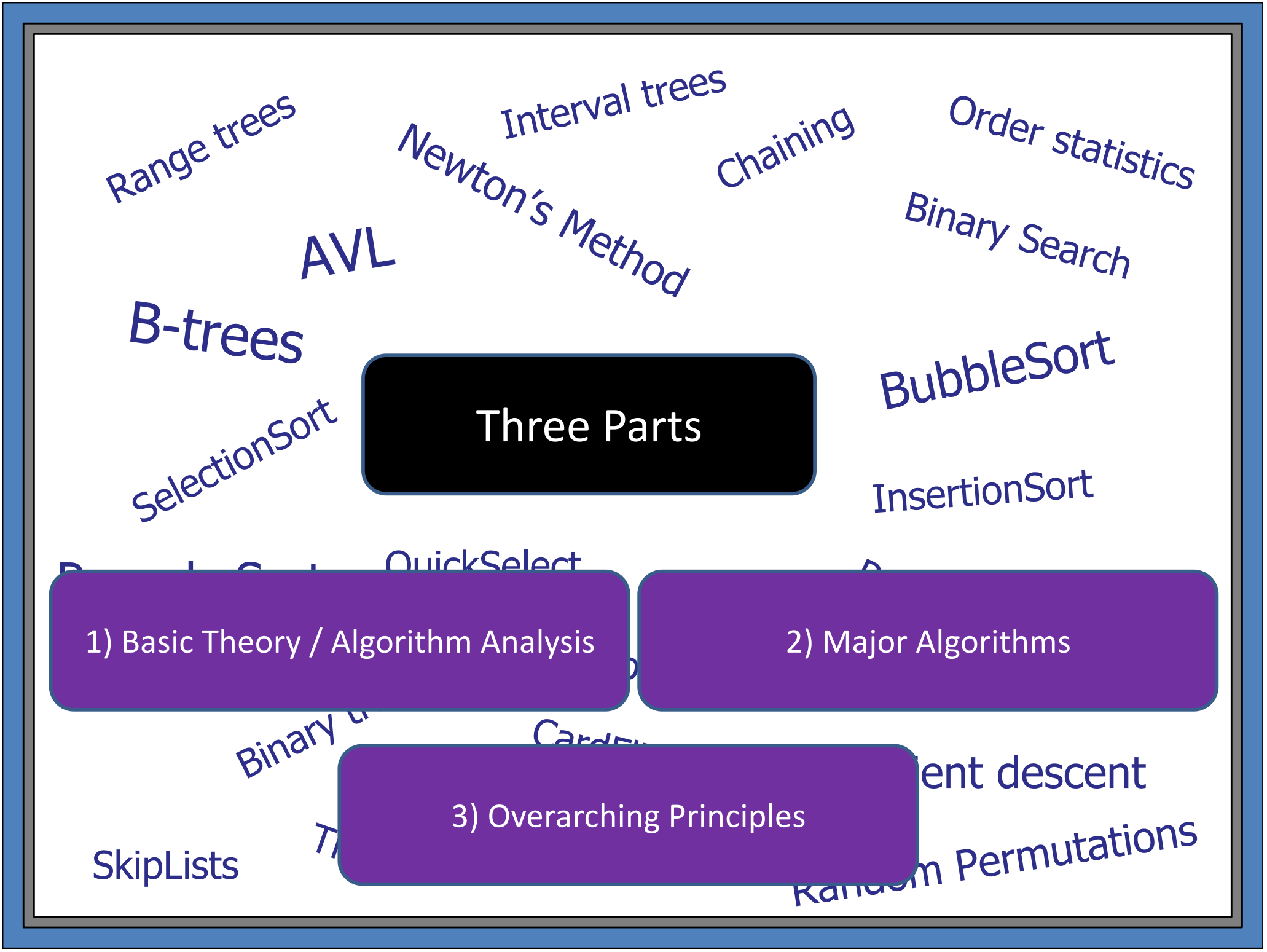
- Know the basic recurrences.
- Review how the algorithms we have studied work.
- Know the running time of the algorithms we have studied.

A bit of advice…

Quiz Advice:

Review problem solving strategies:

- Review problems we have solved on problem sets, in tutorials, in recitation, in class.
- What is the basic strategy used in the question?
  - Binary search? Divide-and-conquer? Sorting?
- What strategy is good for which types of problems?


Range trees
Interval trees
Chaining
Order statistics
Newton's Method
AVL
Binary Search
B-trees
BubbleSort
SelectionSort
Three Parts
InsertionSort
QuickSelect
1) Basic Theory / Algorithm Analysis
2) Major Algorithms
3) Overarching Principles
SkipLists
Random Permutations