

CPSC 510: Algorithms

Unit 1: Analysis of Algorithms

Reading: Ch. 1-2

Outline

- **Introduction to Algorithms**
- Introduction to Algorithm Analysis
- Asymptotic Order of Growth
- Data Structures
- Probabilistic Analysis
- Lower-Bound Limits

Algorithms

- What is an algorithm?

A set of steps to achieve a particular goal.

Needs to be communicable (written down, implemented, etc.)

independent of an implementation.

takes a range of inputs. and produces a result (output)

An algorithm is any well-defined computational procedure that takes a set of values (input) and produces a set of values (output).

Algorithm Properties

- What are properties of “good” algorithms?
 - Completeness
 - Efficient
 - Deterministic
 - Unambiguous
 - Correctness
 - Easily implemented
 - Readable
 - Elegant

Correctness

- The most important property for an algorithm is to be correct.
 - A correct algorithm returns the expected output for every input.
- Need to be able to show, possibly prove, that your algorithm is correct.
- When implementing an algorithm, also need to worry about:
 - finiteness (memory, set of integers, FP precision)
 - programming mistakes
 - hardware issues (caching, pipelined execution)

Performance

- When designing an algorithm, it is desirable to make the algorithm as efficient as possible.
- Why is a lot of focus given to performance of algorithms?
 - Performance is the currency of computing.
 - Often draws the line between what is feasible and what is impossible.
 - Algorithms help us understand scalability. What happens if we add more hardware?
 - Algorithmic improvements have contributed significantly to overall computer performance over the past 30+ years.

Performance

- It is always necessary to write the fastest algorithm? *No*
- In what situations it is most desirable to have a fast algorithm?
 - *Bottlenecks*
 - *Resource-poor devices*
 - *Real-time / quick response time*
 - *Large inputs*
 - *Heavily used data structures*
 - *Necessary to meet performance requirements*

Correctness / Performance Tradeoff

- In some cases, an algorithm may sacrifice correctness for performance.
 - Typically when the problem has no computationally feasible solution.
- Examples:
 - Restrict the input in some manner.
 - Instead of obtaining an optimal solution, find a solution that is good enough.
- Important to document any decisions made in this regard.

Outline

- Introduction to Algorithms
- **Introduction to Algorithm Analysis**
- Asymptotic Order of Growth
- Data Structures
- Probabilistic Analysis
- Lower-Bound Limits

Computational Model

- For a pseudo-code algorithm, can only estimate run-time performance.
- Need a computational model that is simple but is realistic.
- For this course, use a computational model with the following characteristics:
 - Single processor (for now)
 - Instructions are executed sequentially.
 - All common statements take constant time.
 - There is a limit on the size of integers (and other basic data types).

Example: Insertion Sort

- Here is a pseudocode algorithm for insertion sort. It modifies an array A with length $A.length$.

INSERTION-SORT(A)

```
1  for  $j = 2$  to  $A.length$ 
2     $key = A[j]$ 
   // Insert  $A[j]$  into the sorted sequence  $A[1 .. j - 1]$ 
3     $i = j - 1$ 
4    while  $i > 0$  and  $A[i] > key$ 
5       $A[i + 1] = A[i]$ 
6       $i = i - 1$ 
7     $A[i + 1] = key$ 
```

Example: Analyzing Insertion Sort

- To determine how long insertion sort takes, we must consider:
 - the running-time of each statement
 - the number of times each statement executes
- Assume that each statement takes constant time c .
- The number of times a statement executes is dependent on whether the statement is enclosed in a conditional (if / switch) or a loop.
 - Statements not in a loop or conditional are executed once (unless the algorithm terminates earlier).
 - When a for / while loop exits in the usual way, the test is executed one time more than the loop body.

Example: Analyzing Insertion Sort

- Let:
 - $t(s)$ be the time it takes statement s to run
 - $f(s)$ be the number of times s executes
- The running time is then calculated as follows:

$$\begin{aligned} T &= \sum_{\text{all } s} t(s) \cdot f(s) \\ &= \sum_{\text{all } s} c \cdot f(s) \\ &= c \sum_{\text{all } s} f(s) \end{aligned}$$

Example: Analyzing Insertion Sort

s	Statement	$f(s)$ best-case	$f(s)$ worst-case
1	for $j = 2$ to $A.length$	n	n
2	$key = A[j]$	$n - 1$	$n - 1$
3	$i = j - 1$	$n - 1$	$n - 1$
4	while $i > 0$ and $A[i] > key$	$n - 1$	$\frac{n(n+1)}{2} - 1$
5	$A[i + 1] = A[i]$	0	$\frac{n(n+1)}{2} - 1 - (n - 1)$
6	$i = i - 1$	0	$\frac{n(n+1)}{2} - 1 - (n - 1)$
7	$A[i + 1] = key$	$n - 1$	$n - 1$

Let $n = A.length$

Example: Analyzing Insertion Sort

- What is the best case running time?

$$\begin{aligned} & \sum_{\text{all } s} t(n) \cdot f(n) \\ &= c \sum_{\text{all } s} f(n) \\ &= c (4(n-1) + n) \\ &= 5cn - 4c \\ &= an + b \quad \text{linear} \quad = \theta(n) \end{aligned}$$

Example: Analyzing Insertion Sort

- What is the worst case running time?

$$an^2 + bn + d \quad \text{quadratic}$$

Algorithm Running Time

- The time taken by an algorithm depends on the input.
 - Longer inputs typically take longer than shorter inputs.
 - An algorithm may even take differing amounts of time on two inputs of the same size.
- Focus on the relationship between the running time with respect to the size of the input.
- The size of the input depends on the problem being studied.
 - Usually the number of items in the input. Example: the size of the array being sorted.
 - But could be something else. If multiplying two integers, could be the total number of bits in the two integers.
 - Could be described by more than one number. For example: graph algorithm running times are usually expressed in terms of the number of vertices and the number of edges in the input graph.

Types of Algorithm Analysis

- **Best-case analysis:** The running time of the input (of size n) that causes the algorithm to run the fastest (best).
- **Worst-case analysis:** The running time of the input (of size n) that cause the algorithm to run the slowest (worst).
- **Average-case analysis:** The expected running time of the algorithm over all inputs of size n .

Average-case Algorithm Analysis

- What is the average-case running time for insertion sort?

Still quadratic

Assuming that all inputs are equally likely. In many cases, this is not true.

Efficient algorithms can exploit frequently executed inputs.

Running Time of Statements

- We assumed that all statements take constant time. Is this a valid assumption?

No.

- Division takes longer than addition
- High level statements translate to different amounts of assembly instructions
- Memory operations vary depending on caching

If just analyzing the relationship between run-time size and performance, this is OK.

Running Time of Statements

- For the duration of this course: assume that all pseudocode operations that refer to standard imperative operations take constant time.
- Function calls take as long as the called function takes.
 - Example: `printList(L)` takes variable time that is dependent on the size of `L`
- Beware of operating^{or} ~~ing~~ overloading!
 - Example: `if (stringA == stringB)` takes variable time that is dependent on the size of the strings.

Outline

- Introduction to Algorithms
- Introduction to Algorithm Analysis
- **Asymptotic Order of Growth**
- Data Structures
- Probabilistic Analysis
- Lower-Bound Limits

Big-O Notation

- Big-O notation is used to express run-times of algorithm as an asymptotic bound.

$$\Theta(g(n)) = \{f(n): \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$$

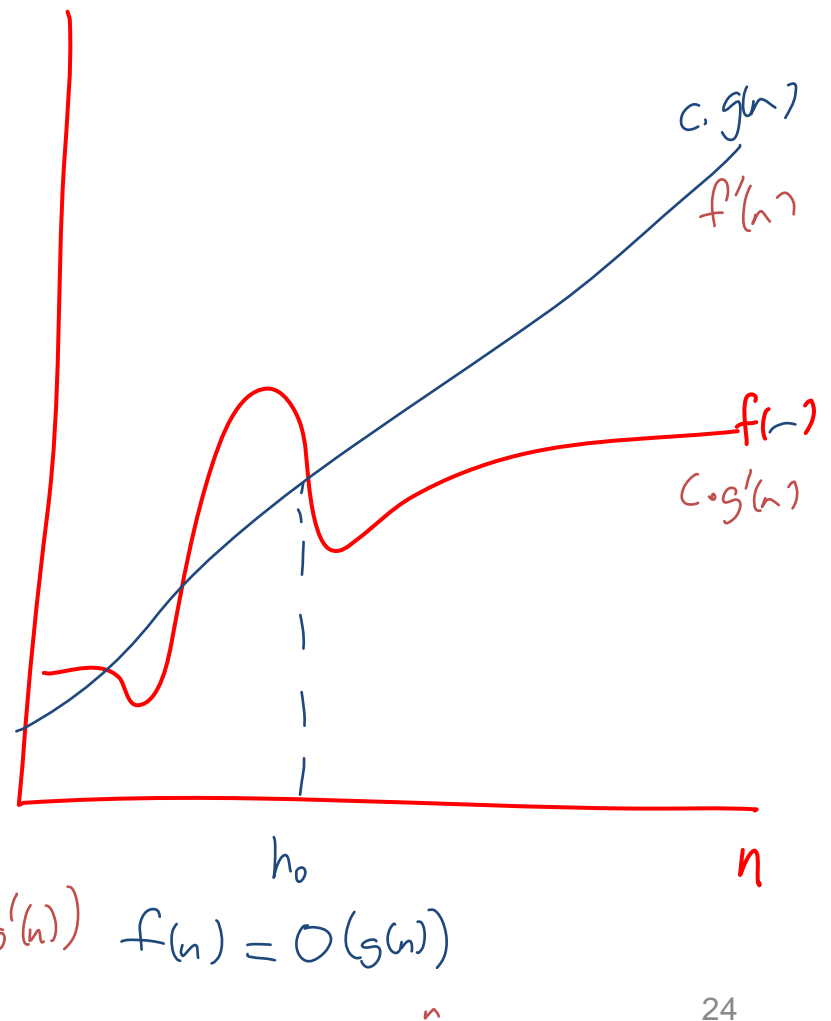
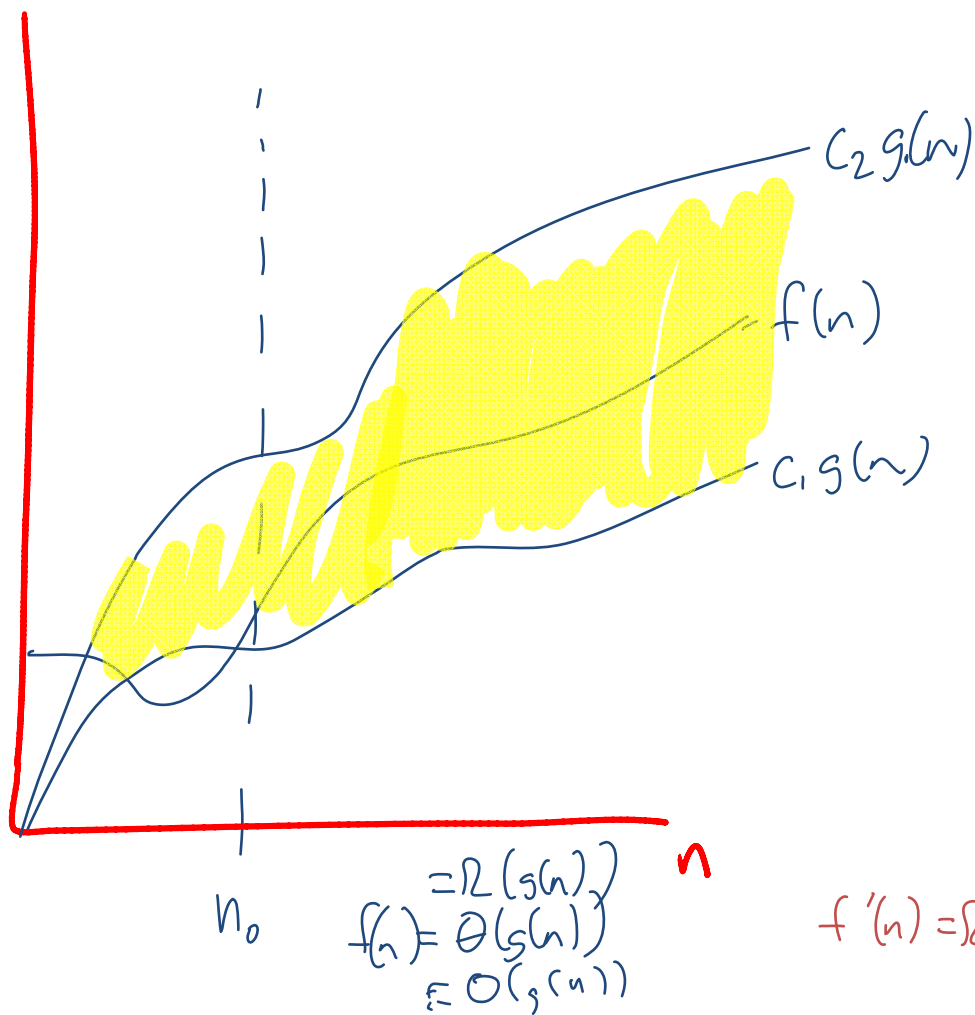
$$O(g(n)) = \{f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c g(n) \text{ for all } n \geq n_0\}$$

$$\Omega(g(n)) = \{f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq c g(n) \leq f(n) \text{ for all } n \geq n_0\}$$

$$o(g(n)) = \{f(n): \text{for any positive constant } c, \text{ there exists a positive constant } n_0 \text{ such that } 0 \leq f(n) < c g(n) \text{ for all } n \geq n_0\}$$

$$\omega(g(n)) = \{f(n): \text{for any positive constant } c, \text{ there exists a positive constant } n_0 \text{ such that } 0 \leq c g(n) < f(n) \text{ for all } n \geq n_0\}$$

Big-O Notation



Big-O Notation Notes

- The equations represent sets.
 - Example: $3n^2 - 5n + 2 \in \Theta(n^2)$. $\in O(n^{73}) \in o(n^{73}) \neq o(n^2)$
 - However, it is common to write $3n^2 - 5n + 2 = \Theta(n^2)$.
- Helpful way to summarize these notations:
 - Θ : asymptotically tight bound (=)
 - O : asymptotic upper bound (\leq)
 - Ω : asymptotic lower bound (\geq)
 - o : non-asymptotic upper bound ($<$)
 - ω : non-asymptotic lower bound ($>$)
- Often looking for an **asymptotically tight upper bound**.
 - $g(n)$ is an asymptotically tight upper bound for $f(n)$ if and only if $f(n) = O(g(n))$ and $f(n) \neq o(g(n))$, $\Rightarrow f(n) = \Theta(g(n))$

Exercise: Rates of Growth

For the following pairs of functions, which one grows at a faster rate as n increases? Or do they grow at the same rate?

a. $f_1(n) = a^n$ for some constant $a > 1$ \leftarrow faster

$f_2(n) = n^b$ for some constant $b > 1$

Exponential function a^n with $a > 1$ always grows faster than polynomials.

$$\lim_{n \rightarrow \infty} \frac{a^n}{n^b}$$

Exercise: Rates of Growth

b. $f_1(n) = 2^n$ ← faster

$$f_2(n) = 2^{n/2} = (2^{1/2})^n = (\sqrt{2})^n$$

$$2^n < c(\sqrt{2})^n \quad \text{for } n \geq n_0$$

Exponential functions with different bases grow at different rates

c. $f_1(n) = \log_2 n = \lg n = \frac{\log_3 n}{\log_3 2} = \left(\frac{1}{\log_3 2}\right) \log_3 n$

$$f_2(n) = \log_3 n$$

Grow at the same rate

$$f_1(n) = \Theta(f_2(n))$$

Running Time Comparison

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.

	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

Precise Running Time

- Though most of the running time equation has been abstracted away, obtaining a more precise running time may be important.
 - Precise measurement requires understanding expected machine and details about the underlying machine.
- For a particular problem, there are often several (infinite) algorithms that have the same optimal rate-of-growth.
 - Which one do you choose?
- Occasionally, you need a “memory efficient” algorithm.
 - Especially in memory-limited embedded systems.
 - Though the analysis is different, many of the concepts of running time analysis can be applied to space analysis.

Outline

- Introduction to Algorithms
- Introduction to Algorithm Analysis
- Asymptotic Order of Growth
- **Data Structures**
- Probabilistic Analysis
- Lower-Bound Limits

Data Structures

- The choice of data structure used to organize data has a large impact on the performance of an algorithm. Choices:
 - Use a “standard” data structure.
 - Modify a “standard” data structure.
 - Invent your own data structure.
- Many algorithms call for a dictionary data structure:
 - Each record in the data structure has a unique key used for retrieval.
 - Common operations: insert, delete, and search.
 - Internally, what data structure do you use?

Data Structure Summary

Data Structure		Insert	Search	Delete
Linked List (unsorted)	Worst-case:	$O(1)$	$O(n)$	$O(n)$
	Average-case:	$O(1)$	$O(n)$	$O(n)$
Hash Table <i>* size of table ↳ proportional to n</i>	Worst-case:	$O(1)$	$O(n)$	$O(n)$
	Average-case:	$O(1)$	$O(1)^*$	$O(1)^*$
Binary Search Tree	Worst-case:	$O(n)$	$O(n)$	$O(n)$
	Average-case:	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
AVL Tree (self-balancing BST)	Worst-case:	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
	Average-case:	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$

What about Arrays?

- Can arrays be used to implement this dictionary? Yes!
- Two possible implementations:
 - Like a hash table
 - Difference: array contains a unique slot for each possible key.
 - Like a linked list
 - Difference: elements are added to the end of the array, will periodically need to resize the array.

amortized analysis

Outline

- Introduction to Algorithms
- Introduction to Algorithm Analysis
- Asymptotic Order of Growth
- Data Structures
- **Probabilistic Analysis**
- Lower-Bound Limits

Average Case Running Time

- When computing average case running time, one approach is to compute the expected number of some event that is directly proportional to the running time.
- Examples of events to count:
 - number of comparisons
 - number of elements analyzed
 - number of multiplications
 - number of calls to function f
- Computing an exact expected value can be helpful.
 - Can compare to the exact expected value of a worst-case run.
 - Shows the expected performance difference even if they have the same asymptotic behavior.

Expectation

- Consider a single application of a random act.
 - Assume there are n possible outcomes numbered from 1 to n .
 - Exactly one of the outcomes will be randomly selected.
 - For each outcome j in S , let p_j be the probability that j is selected during a single application.
 - $0 \leq p_j \leq 1$
 - There is a probability $1 - p_j$ that j is not selected
- The expected value of a random variable X is computed by:

$$E[X] = \sum_{j=1}^n w_j \cdot p_j$$

where w_j refers to a numeric value associated with outcome j .

- In algorithm analysis, it will typically be the amount of work associated with that outcome.

Expectation Example

- Consider a game where you flip two fair coins. You earn \$3 for each head but lose \$2 for each tail. On average, how much do you earn?

– Outcome 1: 2 heads	$w_1 = 6$	$p_1 = 0.25$
– Outcome 2: 1 head, 1 tail	$w_2 = 1$	$p_2 = 0.5$
– Outcome 3: 2 tails	$w_3 = -4$	$p_3 = 0.25$

$$\begin{aligned} E[X] &= w_1 p_1 + w_2 p_2 + w_3 p_3 \\ &= 6(0.25) + 1(0.5) + (-4)(0.25) = 1 \end{aligned}$$

Example: Linked List Search

- What is the expected number of comparisons when performing a linear search of a linked list with n unique items under each of the following conditions?

Example: Linked List Search

a. Conditions:

- The order of the items in the list is arbitrary.
- The search term is guaranteed to be in the list.
- Each search term is equally likely.

$$w_j = j \quad p_j = \frac{1}{n}$$

$$E[X] = \sum_{j=1}^n w_j \cdot p_j = \sum_{j=1}^n j \cdot \frac{1}{n} = \frac{1}{n} \sum_{j=1}^n j = \frac{1}{n} \left(\frac{n(n+1)}{2} \right)$$
$$= \frac{n+1}{2}$$

Example: Linked List Search

b. Conditions:

- The order of the items in the list is arbitrary.
- The search term is not necessarily in the list. Let p be the probability that the search term appears in the list.
- Each search term is equally likely.

$$w_j = j, \quad p_j = \frac{p}{n}, \quad w_{\text{miss}} = n, \quad p_{\text{miss}} = 1 - p$$

$$E[X] = \left(\sum_{j=1}^n w_j p_j \right) + w_{\text{miss}} p_{\text{miss}} = \sum_{j=1}^n j \frac{p}{n} + n(1-p)$$

$$= \frac{p}{n} \left(\frac{n(n+1)}{2} \right) + n(1-p) = \left(1 - \frac{p}{2} \right) n + \frac{p}{2}$$

Example: Linked List Search

c. Conditions:

- The order of the items in the list is arbitrary.
- The search term is guaranteed to be in the list.
- Search terms are not equally likely: each specific search term j has probability p_j of being the selected search term.

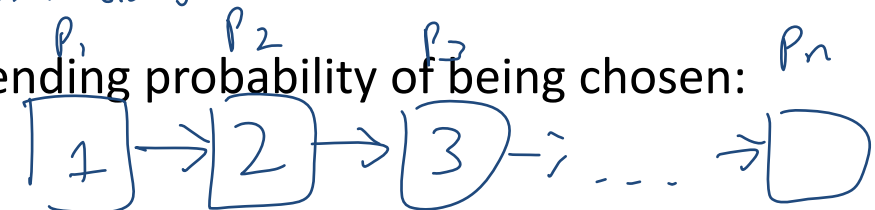
$$E[X] = \sum_{j=1}^n j p_j$$

s_j ← Search term approach

Example: Linked List Search

d. Conditions: Search term j is in slot j

- The list is reordered in descending probability of being chosen:
 $S_1 \geq S_2 \geq \dots \geq S_{n-1} \geq S_n$
 ~~$p_1 \geq p_2 \geq \dots \geq p_{n-1} \geq p_n$~~
- The search term is guaranteed to be in the list.
- Search terms are not equally likely: each specific search term j has probability S_j of being the selected search term and that



$$w_j = j, \quad p_j = S_j$$

$$S_1 \geq S_2 \geq \dots \geq S_{n-1} \geq S_n$$

$$E[X] = \sum_{j=1}^n w_j \cdot p_j = \sum_{j=1}^n j \cdot S_j$$

Zipf's Law

- Zipf's law states that the frequency of an item is inversely proportional to its rank on the frequency table.
- Assume the items are ranked where item 1 occurs the most frequently.
 - Item 2 occurs $1/2$ as often as item 1
 - Item 3 occurs $1/3$ as often as item 1
 - Item n occurs $1/n$ as often as item 1
- The empirical law refers to the fact that many types of data studied in the physical and social sciences can be approximated with this distribution.
 - Frequency of word use in English (and many other languages)
 - Population ranks of cities
 - Income of companies

Example: Linked List Search

e. Conditions:

- The list is reordered in descending probability of being chosen:

$$\begin{array}{l} s_1 \geq s_2 \geq \dots \geq s_{n-1} \geq s_n \\ \cancel{p_1 \geq p_2 \geq \dots \geq p_{n-1} \geq p_n} \end{array}$$

- The search term is guaranteed to be in the list.
- Search terms probabilities follow Zipf's law:

$$s_j = \frac{1}{j \cdot H(n)} \text{ where } H(n) \text{ is the } n\text{th harmonic number: } H(n) = \sum_{k=1}^n \frac{1}{k}$$

$$E[X] = \sum_{j=1}^n w_j \cdot p_j = \sum_{j=1}^n j \cdot \frac{1}{j \cdot H(n)} = \frac{1}{H(n)} \sum_{j=1}^n 1 = \boxed{\frac{n}{H(n)}}$$

$$H(5) = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5}$$

Example: Linked List Search

f. Conditions:

- The order of the items in the list is arbitrary.
- Instead of a search term, we look for the first term in the list that has a particular property. The probability of an item exhibiting the property is p .

Define outcome j such that element (slot) j is the first item in the list that does exhibit the property.

$$w_j = j, \quad p_j = \left[\prod_{k=1}^{j-1} (1-p) \right] \cdot p = (1-p)^{j-1} \cdot p$$

$$w_{\text{miss}} = n, \quad p_{\text{miss}} = (1-p)^n$$

$$E[X] = \left(\sum_{j=1}^n w_j p_j \right) + w_{\text{miss}} p_{\text{miss}} = \left(\sum_{j=1}^n j (1-p)^{j-1} \cdot p \right) + n (1-p)^n \approx \frac{1}{p}$$

if $p \gg \frac{1}{n}$

Outline

- Introduction to Algorithms
- Introduction to Algorithm Analysis
- Asymptotic Order of Growth
- Data Structures
- Probabilistic Analysis
- **Lower-Bound Limits**

Lower Bound Limits

- After creating an algorithm, how do we know whether this is the fastest possible algorithm or not?
- Getting a precise lower limit can be difficult or impossible in many cases.
 - The lower limit has to apply to all algorithms (known and unknown).
- However, there are many cases where it is possible to prove a particular lower bound.
- **CAUTION!** Just because an algorithm has a known lower bound (not $O(1)$) does not necessarily mean that there is a known algorithm with that lower bound.

Trivial Lower Bounds

- An algorithm has to read all of the items it needs to process and write all its output. This leads to trivial lower bounds.
- Example:
 - Matrix multiplication of $n \times n$ matrices:
Strassen's
 $\Omega(n^2)$ to read in the input matrices and to output the answer.
 - Generating all permutations of n items:
 $\Omega(n!)$
 - Finding the largest number of n numbers:
 $\Omega(n)$ need to read all numbers.

Adversary Arguments

- Consider an adversary who forces the algorithm to take as long as possible – let's say $f(n)$. Then the algorithm must be $\Omega(f(n))$.
- Example: Mystery number guessing game

The best worst-case running time is $\Omega(\lg n)$.

Adversary Arguments

- Example: Merging two sorted lists of size n (the merge step of merge sort).

Arrange the data such that the combined lists alternate:

$$b_1 < a_1 < b_2 < a_2 < \dots < b_n < a_n$$

a_i will need to be compared to b_i and b_{i+1} . a_i will need to be compared to b_i and b_{i+1} . Each term except a_n will need 2

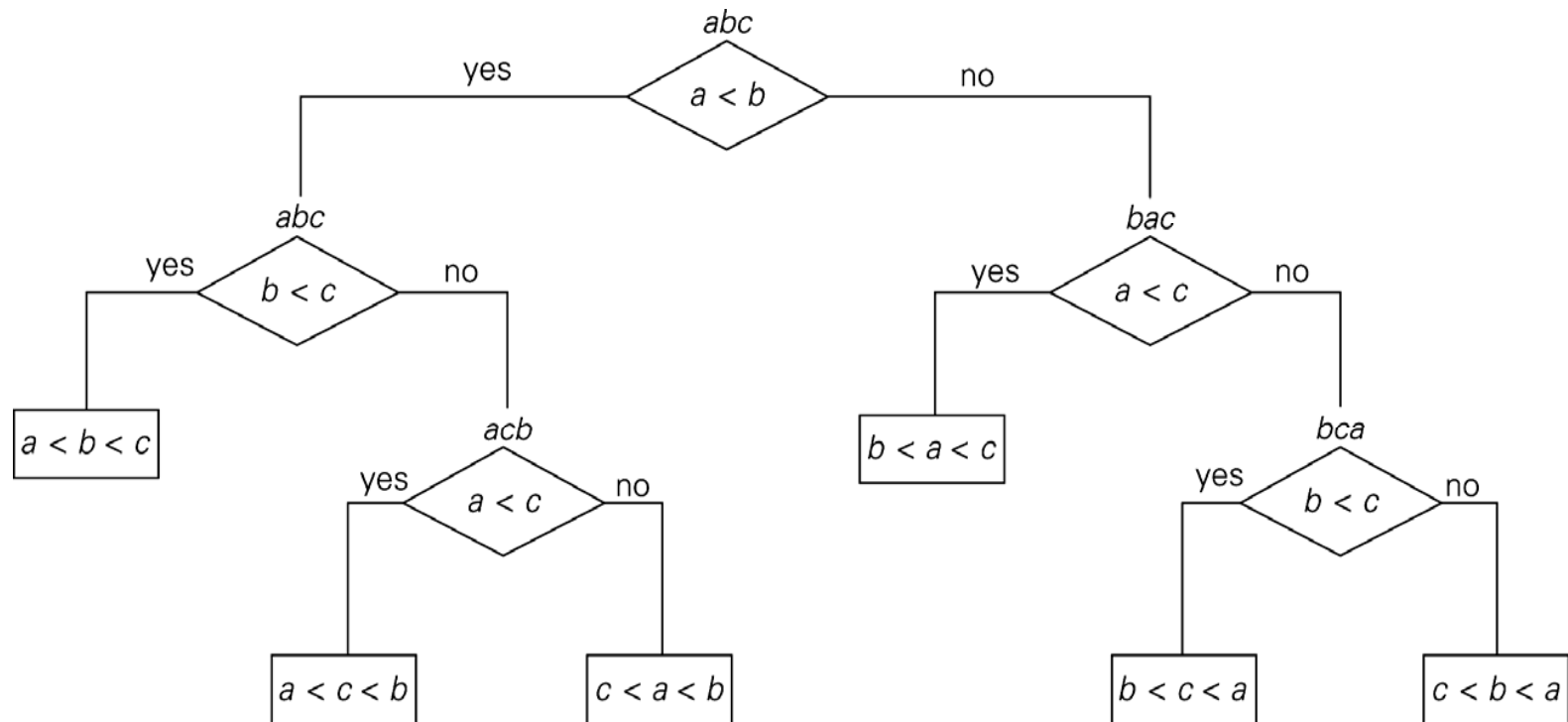
comparisons $\Rightarrow \Omega(n)$ comparisons for merging.

$$\Rightarrow \Omega(n)$$

Decision Trees

- A **decision tree** is useful for determining lower bounds for sorting.
 - Models all the comparisons needed in a tree.
- The leaves of the tree form the different permutations of the sort.
- The number of comparisons in the worst-case is the height of the tree.

Decision Trees



Lower Bound for Sorting

- Theorem: Any comparison sort algorithm takes $\Omega(n \lg n)$ comparisons.

Proof: Let l be the number of leaves and h be the height of the decision tree.

$$l = n!$$

In best tree :

$$2^h = l$$

$$\cancel{h} = \lg l$$

$$h = \lg n!$$

Stirling's approximation for $n!$

$$\Rightarrow \lg n! \Rightarrow \Theta(n \lg n)$$