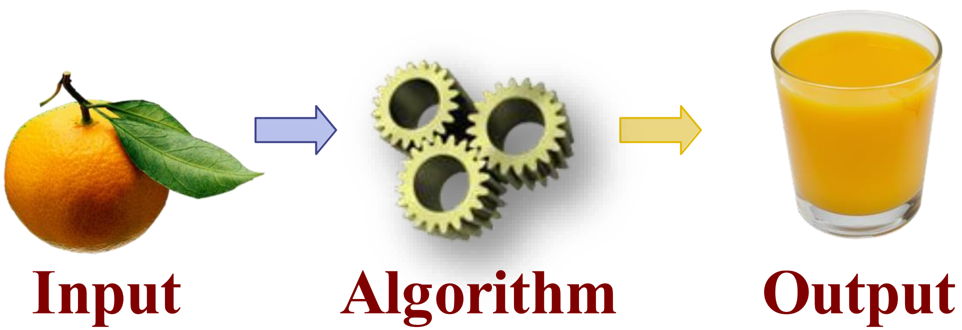
Beyond Worst-cast Algorithm Analysis: Introduction

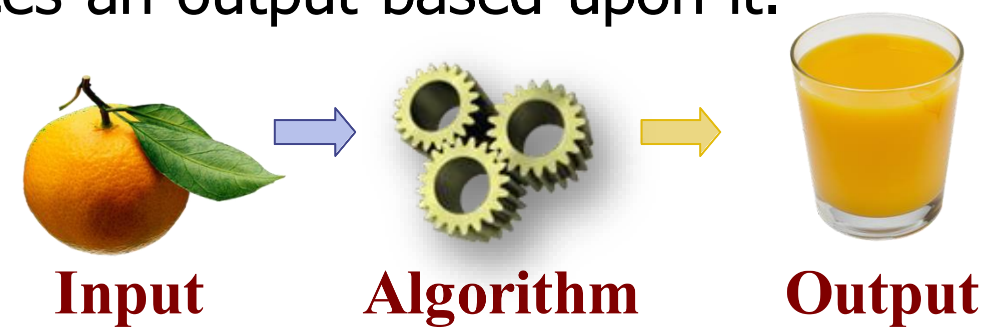
Michael T. Goodrich
CS 263
Univ. of California, Irvine



Some slides adapted from Uriel Feige, Michael Mitzenmacher

Algorithms and Data Structures

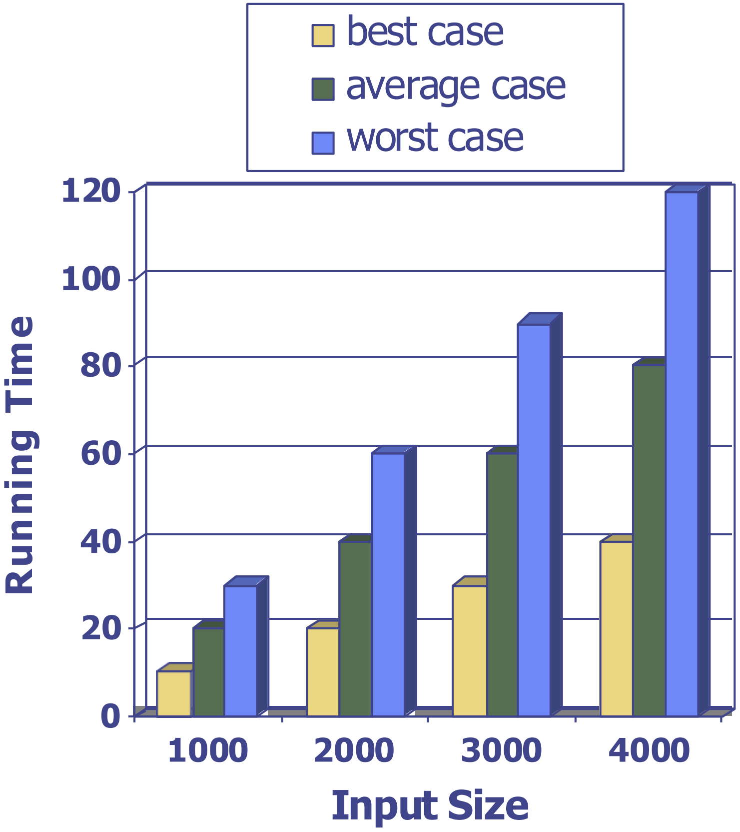
- An **algorithm** is a step-by-step procedure for performing some task in a finite amount of time.
  - Typically, an algorithm takes input data and produces an output based upon it.



- A **data structure** is a systematic way of organizing and accessing data.

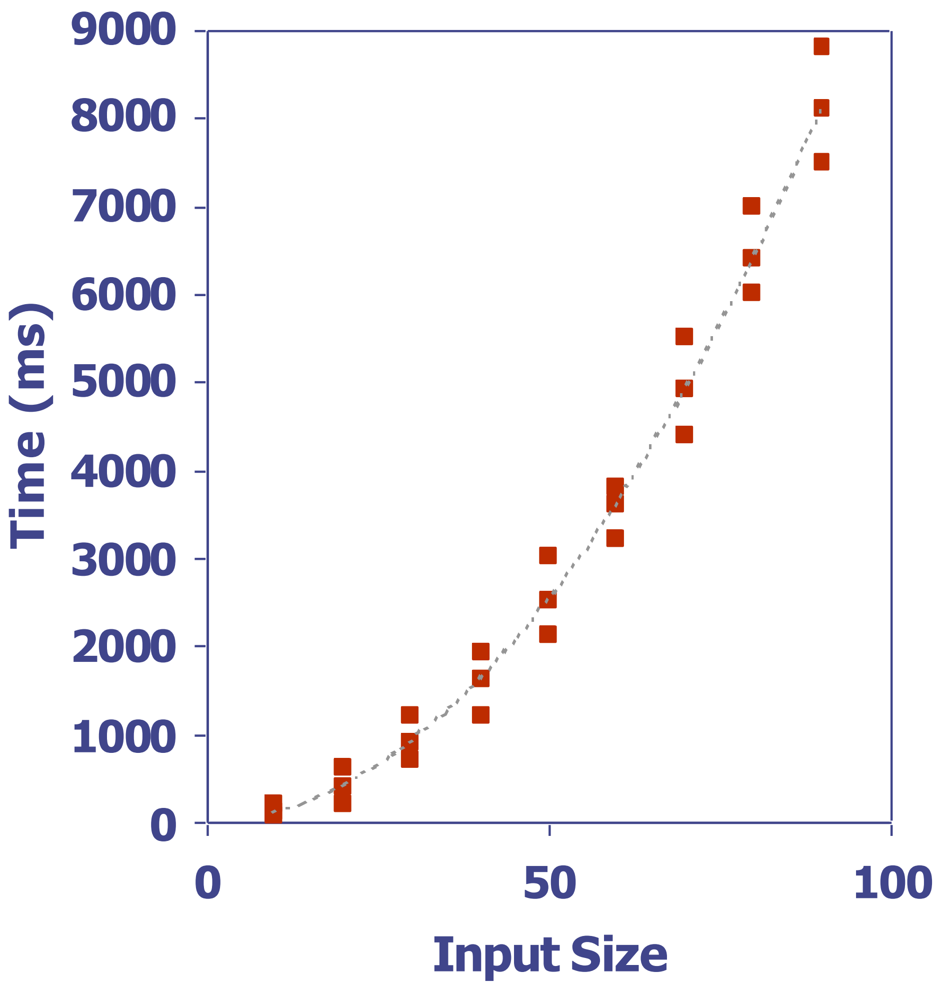
Running Times

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Traditionally, we focus on the **worst case running time**.
  - Theoretical analysis
  - Might not capture real-world performance



Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition, noting the time needed:
  - Plot the results
  - Try to match a curve to the times
- Is good for capturing average case analysis, but isn't theoretical and doesn't provide deep insights.



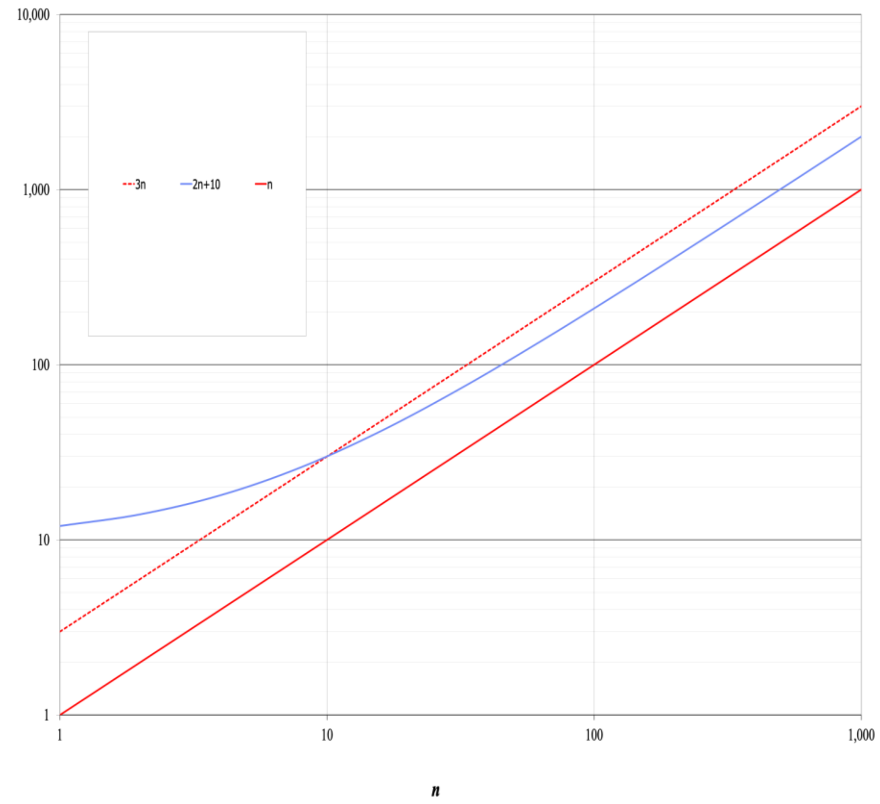
Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used

Big-Oh Notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

  $f(n) \leq cg(n)$ for $n \geq n_0$
- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$



Relatives of Big-Oh

big-Omega

- f(n) is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \geq c\, g(n) \text{ for } n \geq n_0$$

big-Theta

- f(n) is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that

$$c'g(n) \leq f(n) \leq c''g(n) \text{ for } n \geq n_0$$

little-o

For real or complex-valued functions of a real variable x with $g(x) > 0$ for sufficiently large x, one writes

$$f(x) = o(g(x)) \quad \text{as } x \to \infty$$

if

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = 0.$$

Traditional Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes the **worst-case** running time **only** as a function of the input size, n
- Takes into account all possible inputs, whereas some inputs might be rare or never occur in the real world

Worst case analysis

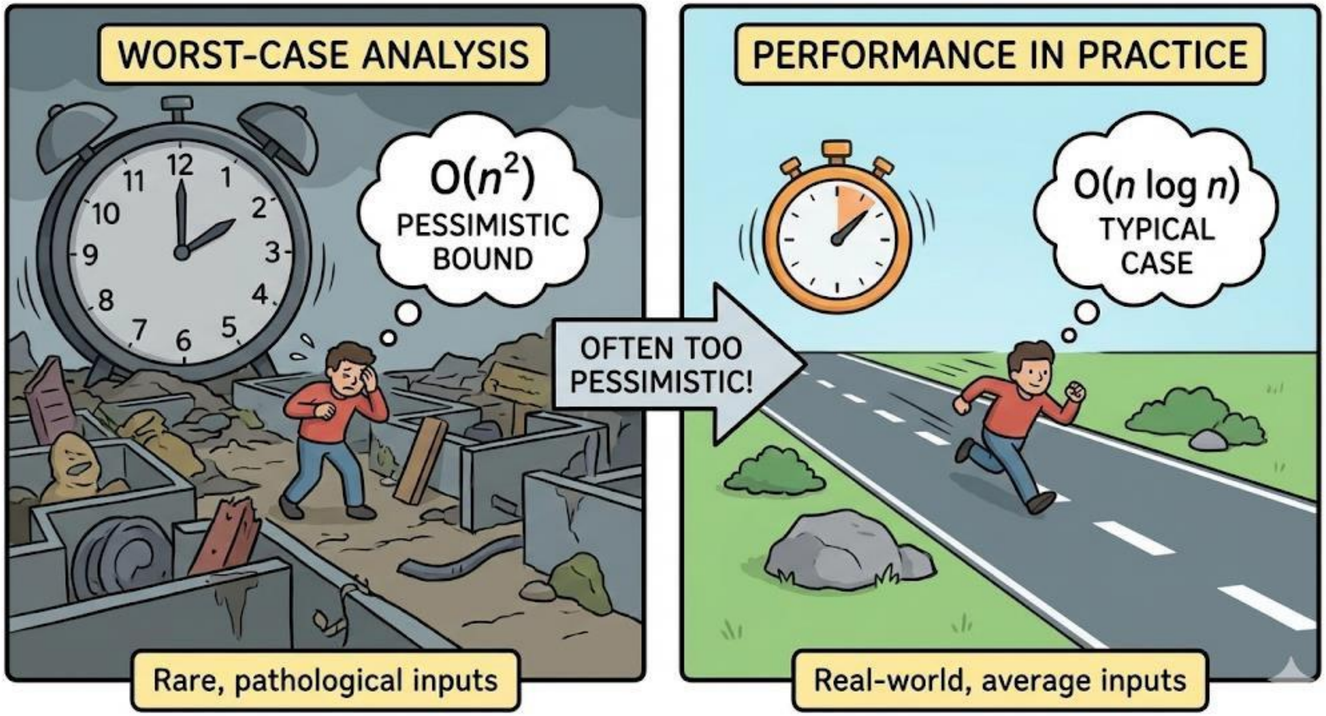
Performance guarantees that hold for every possible instance that might be encountered.

Benefits:

- Guarantees are applicable across all application domains.
- Many fundamental problems actually have algorithms with excellent worst-case guarantees (minimum spanning tree, fast Fourier transform).
- Leads to clean theory, well understood proof techniques, successful classification of many computational problems of interest (theory of NP-completeness, fined grained complexity within P).

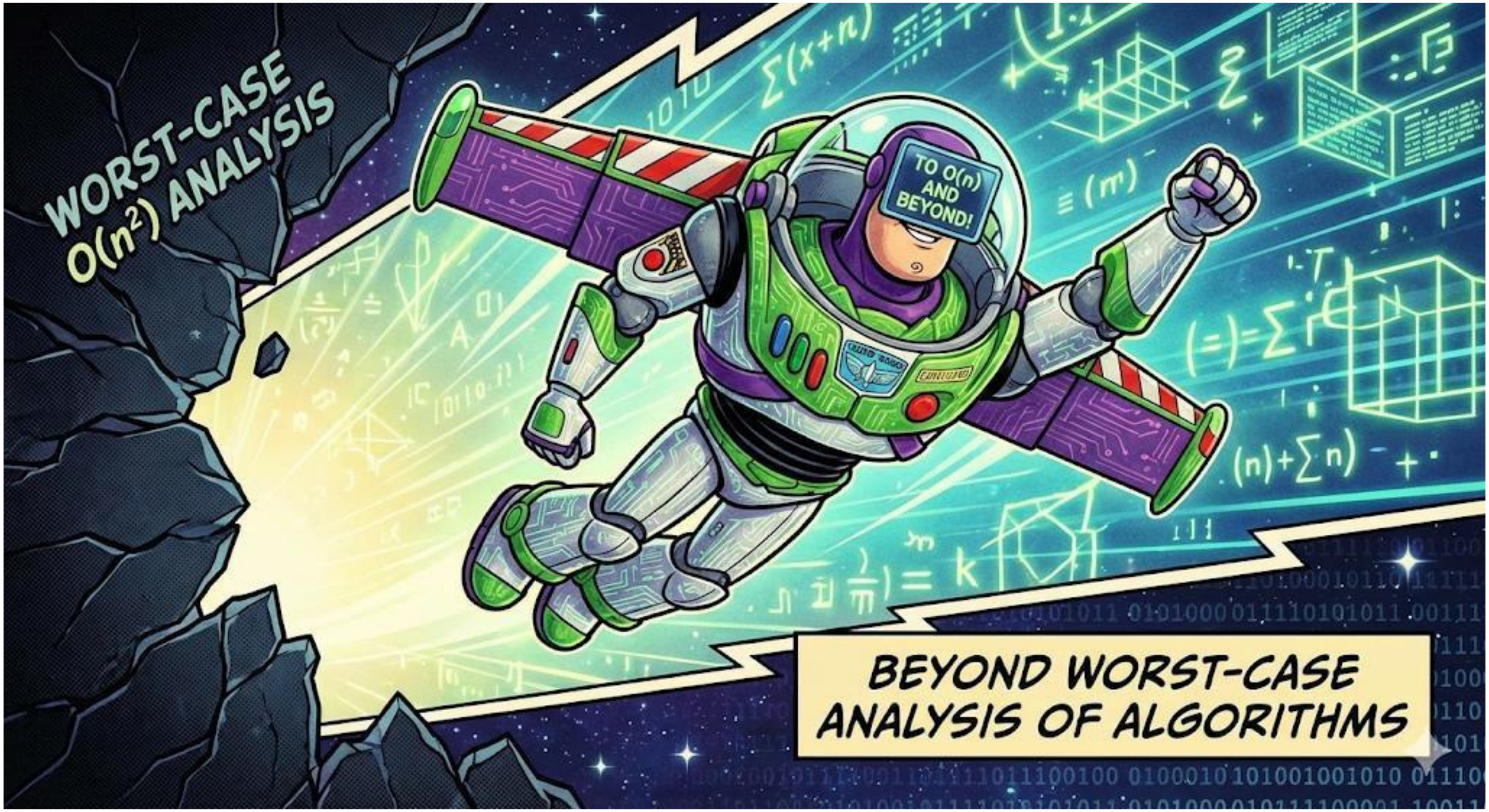
Deficiencies of worst-case analysis

- Might be much too pessimistic compared to performance in practice.
- Ranking algorithms by their worst case performance might turn out to be a misleading predictor for their typical performance in practice.



Decorative image generated using Nano Banana Pro

Beyond Worst-case Analysis

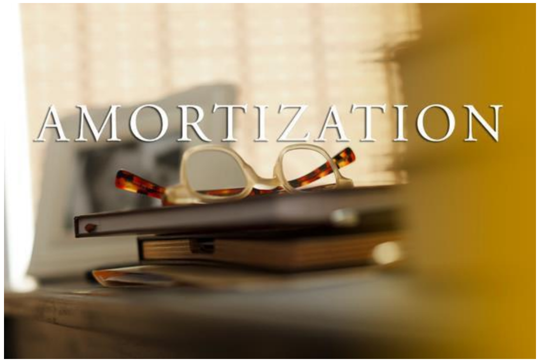


Decorative image generated using Nano Banana Pro

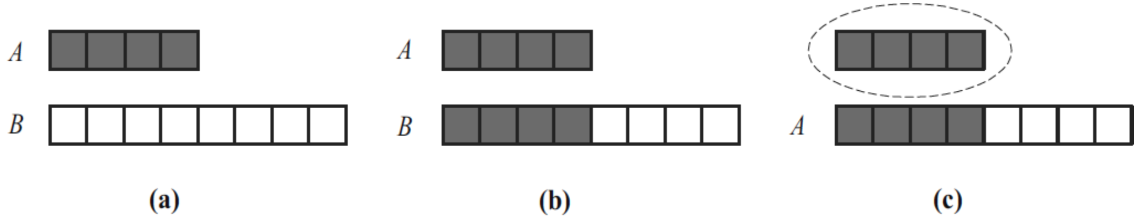
Beyond Worst-case Analysis

1. Amortized analysis
2. Randomized algorithms
3. Instance sensitivity
4. Fixed parameter tractability
5. Learning-augmented algorithm design

Amortized Analysis



- The **amortized running time** of an operation within a series of operations is the worst-case running time of the series of operations **divided** by the number of operations.
- Example: A growable array, S. When needing to grow:
  a. Allocate a new array B of larger capacity.
  b. Copy A[i] to B[i], for i = 0, . . . , n − 1, where n is size of A.
  c. Let A = B, that is, we use B as the array now supporting A.



Growable Array Description

- Let add(e) be the operation that adds element e at the end of the array, A
- When the array is full, we replace the array with a larger one
- But how large should the new array be?
  - Incremental strategy: increase the size by a constant c
  - Doubling strategy: double the size

Algorithm $A.add(e)$:
  if $n = A.length - 1$ **then**
    $B \leftarrow$ **new array of size …**
    for $i \leftarrow 0$ **to** $n-1$ **do**
      $B[i] \leftarrow A[i]$
    $A \leftarrow B$
  $n \leftarrow n + 1$
  $A[n-1] \leftarrow e$

Comparing the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n add operations
- We assume that we start with an empty list represented by a growable array of size 1
- We call amortized time of an add operation the average time taken by an add operation over the series of operations, i.e., $T(n)/n$

Incremental Strategy Analysis

- Over n add operations, we replace the array $k = n/c$ times, where c is a constant
- The total time $T(n)$ of a series of n add operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc =$$
$$n + c(1 + 2 + 3 + \ldots + k) =$$
$$n + ck(k + 1)/2$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- Thus, the amortized time of an add operation is $O(n)$

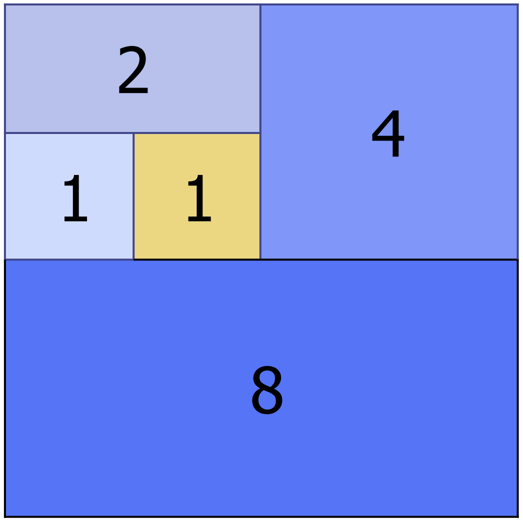
Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n push operations is proportional to

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$
$$n + 2^{k+1} - 1 =$$
$$3n - 1$$

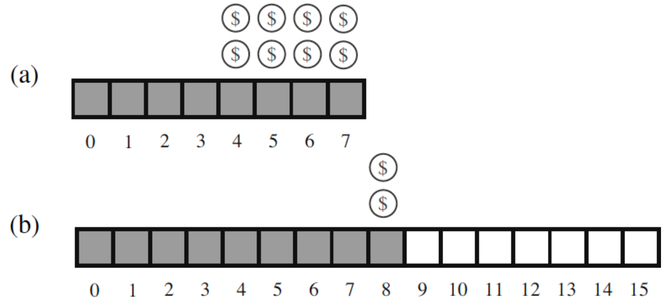
- $T(n)$ is $O(n)$
- The amortized time of an add operation is $O(1)$

geometric series



Accounting Method Proof for the Doubling Strategy

- We view the computer as a coin-operated appliance that requires the payment of 1 **cyber-dollar** for a constant amount of computing time.
- We shall charge each add operation 3 cyber-dollars, that is, it will have an amortized O(1) amortized running time.
  - We over-charge each add operation not causing an overflow 2 cyber-dollars.
  - Think of the 2 cyber-dollars profited in an insertion that does not grow the array as being “stored” at the element inserted.
  - An overflow occurs when the array A has 2^i elements.
  - Thus, doubling the size of the array will require 2^i cyber-dollars.
  - These cyber-dollars are at the elements stored in cells 2^{i-1} through 2^i-1.

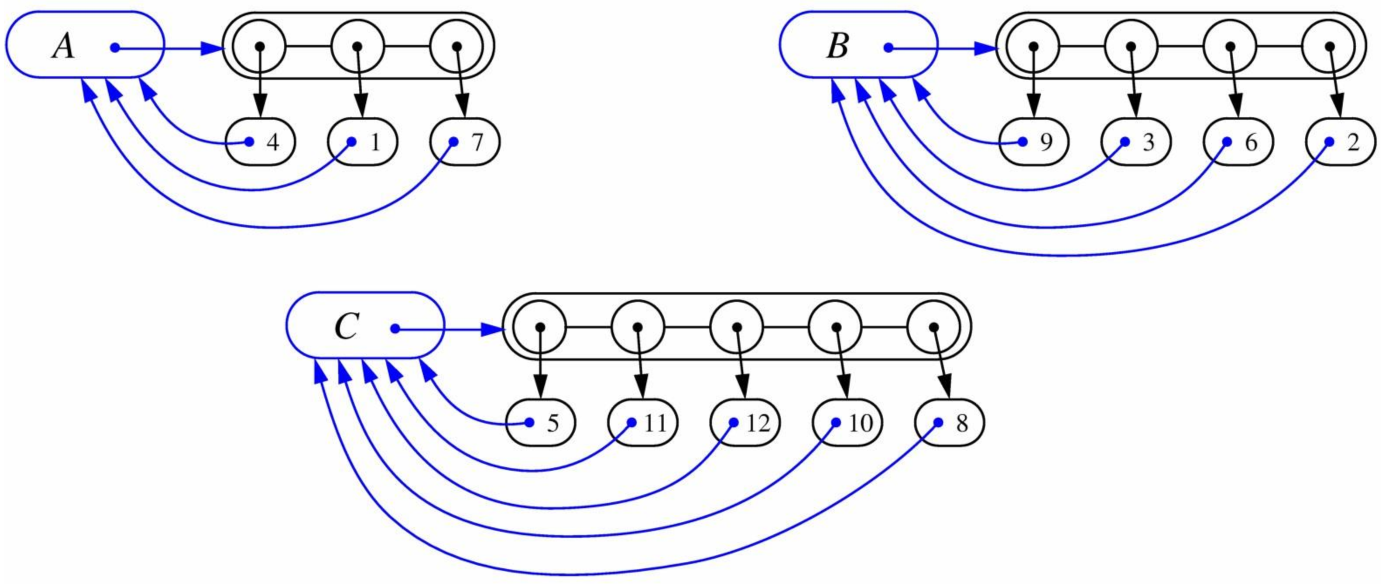


Union-Find Operations

- A **partition** or **union-find** structure is a data structure supporting a collection of disjoint sets subject to the following operations:
- makeSet(e): Create a singleton set containing the element e and return the position storing e in this set
- union(A,B): Return the set A U B, naming the result "A" or "B"
- find(e): Return the set containing the element e

List-based Implementation

- Each set is stored in a sequence represented with a linked-list
- Each node should store an object containing the element and a reference to the set name

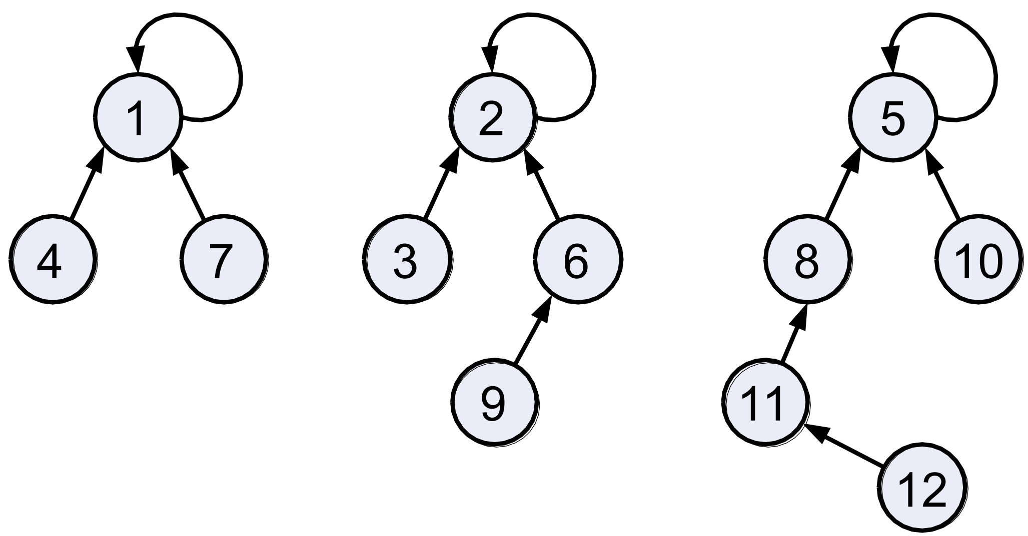


Amortized Analysis of List-based Representation

- When doing a union, always move elements from the smaller set to the larger set
  - Each time an element is moved it goes to a set of size at least double its old set
  - Thus, an element can be moved at most O(log n) times (charge a cyber dollar for each move)
- Total time needed to do n unions and m finds is O(n log n + m).

Tree-based Implementation

- Each element is stored in a node, which contains a pointer to a node
- A node v whose set pointer points back to v is also a set name, otherwise, the pointer is to a parent node.
- Can be implemented so that each union and find takes amortized time $O(\alpha(n))$, which is $O(\log^* n)$.
- See CS 261.

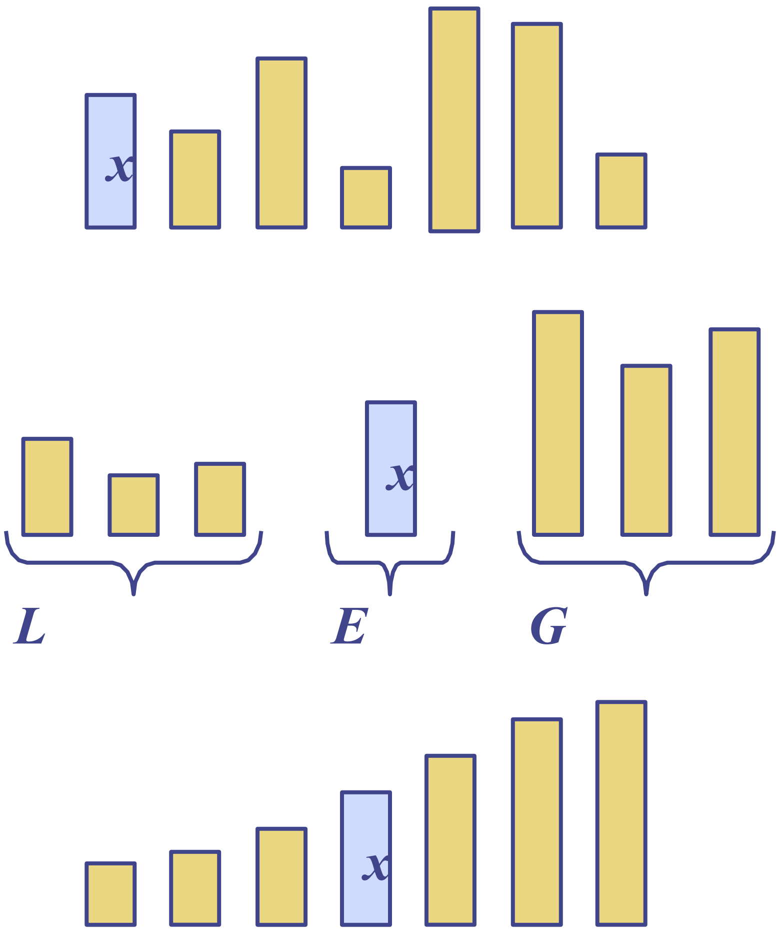


Randomized Algorithms

- A **randomized algorithm** is an algorithm whose behavior depends, in part, on the outcomes of random choices or the values of random bits.
- The main advantage of using randomization in algorithm design is that the results are often simple and efficient.
- In addition, there are some problems that need randomization for them to work effectively, because we use randomization to avoid the worst-case behavior with high probability.

Traditional Quick-Sort

- Quick-sort is a classic sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick the first element x (called pivot) and partition S into
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Recur: sort L and G
  - Conquer: concat L, E and G



Traditional Quick-Sort

- Worst-case running time is $O(n^2)$.
- $T(n) = T(n-1) + n$
- Example: The array is already sorted!

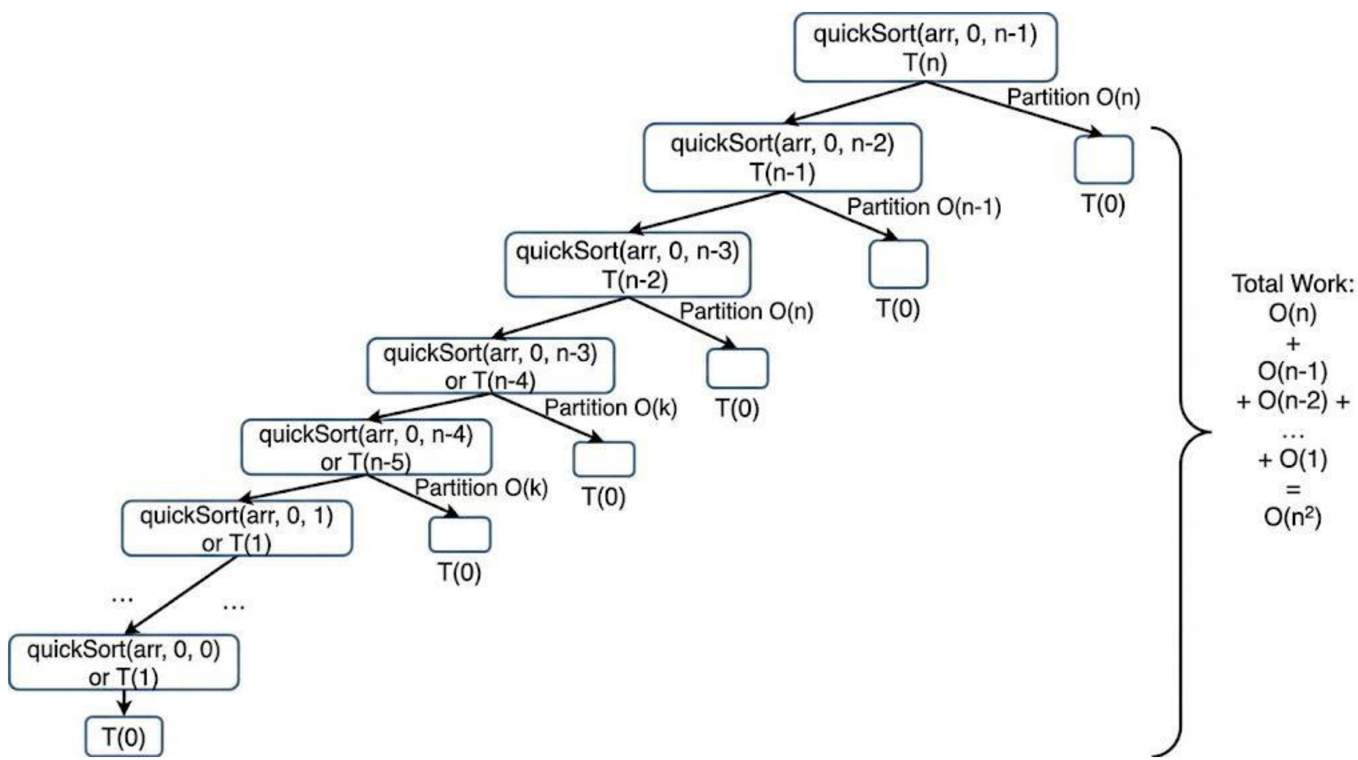
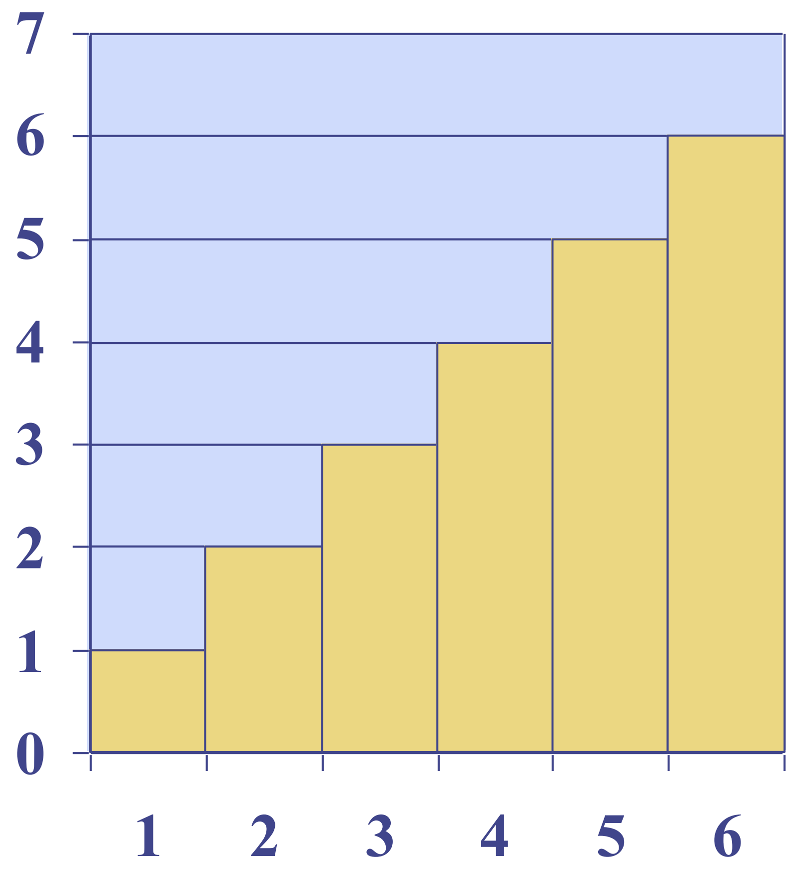


Illustration of a worst-case recursion tree for quick-sort generated using Nana Banana Pro

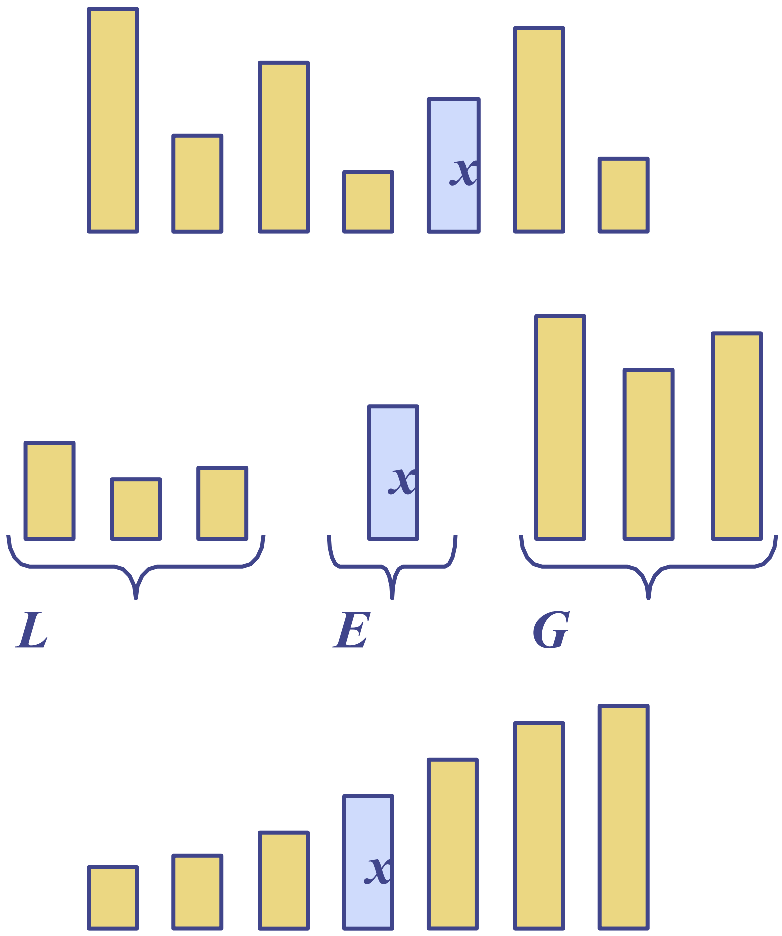
Visualizing the Worst-Case Time

- The worst-case running time of traditional Quick-Sort is
  $O(1 + 2 + \ldots + n)$
- The sum of the first n integers is $n(n + 1) / 2$
  - There is a simple visual proof of this fact
- Thus, algorithm Quick-Sort runs in $O(n^2)$ time



Randomized Quick-Sort

- Randomized Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
  - Divide: pick a **random** element x (called pivot) and partition S into
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Recur: sort L and G
  - Conquer: concat L, E and G



Independence and Conditional Probability

Two events A and B are ***independent*** if

$$\Pr(A \cap B) = \Pr(A) \cdot \Pr(B).$$

A collection of events $\{A_1, A_2, \ldots, A_n\}$ is ***mutually independent*** if

$$\Pr(A_{i_1} \cap A_{i_2} \cap \cdots \cap A_{i_k}) = \Pr(A_{i_1}) \Pr(A_{i_2}) \cdots \Pr(A_{i_k}),$$

for any subset $\{A_{i_1}, A_{i_2}, \ldots, A_{i_k}\}$.

The ***conditional probability*** that an event A occurs, given an event B, is denoted as $\Pr(A|B)$, and is defined as

$$\Pr(A|B) = \frac{\Pr(A \cap B)}{\Pr(B)},$$

assuming that $\Pr(B) > 0$.

Random Variables

- A **random variable** is a function X that maps outcomes from some sample space S to real numbers.
- An **indicator random variable** is a random variable that maps outcomes to the set {0, 1}.
- The **expected value** of a discrete random variable X is defined as

$$E(X) = \sum_{x} x \Pr(X = x),$$

where the sum is taken of the range of X.

- Two random variables X and Y are **independent** if

$$\Pr(X = x | Y = y) = \Pr(X = x),$$

for all real numbers x and y.

- If two random variables X and Y are independent, then we have E(XY) = E(X)E(Y).

Linearity of Expectation

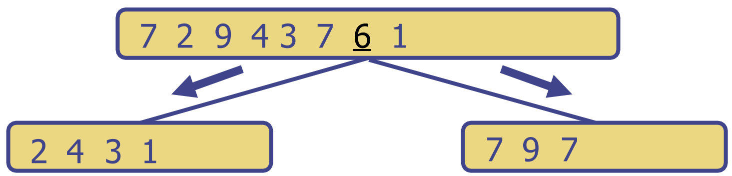
Theorem 1.25 (The Linearity of Expectation): *Let X and Y be two arbitrary random variables. Then $E(X+Y) = E(X) + E(Y)$.*

Proof:

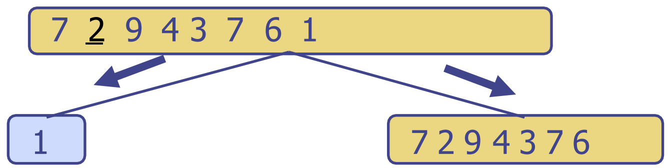
$$
\begin{aligned}
E(X+Y) &= \sum_x \sum_y (x+y)\Pr(X = x \cap Y = y) \\
&= \sum_x \sum_y x \Pr(X = x \cap Y = y) + \sum_x \sum_y y \Pr(X = x \cap Y = y) \\
&= \sum_x \sum_y x \Pr(X = x \cap Y = y) + \sum_y \sum_x y \Pr(Y = y \cap X = x) \\
&= \sum_x x \Pr(X = x) + \sum_y y \Pr(Y = y) \\
&= E(X) + E(Y).
\end{aligned}
$$

Expected Running Time

- Consider a recursive call of quick-sort on a sequence of size s
  - **Good call:** the sizes of L and G are each less than $3s/4$
  - **Bad call:** one of L and G has size greater than $3s/4$

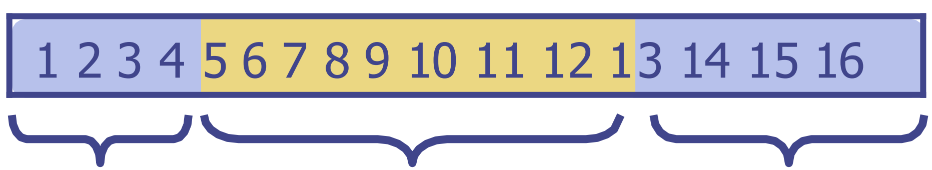


Good call



Bad call

- A call is good with probability $1/2$
  - 1/2 of the possible pivots cause good calls:

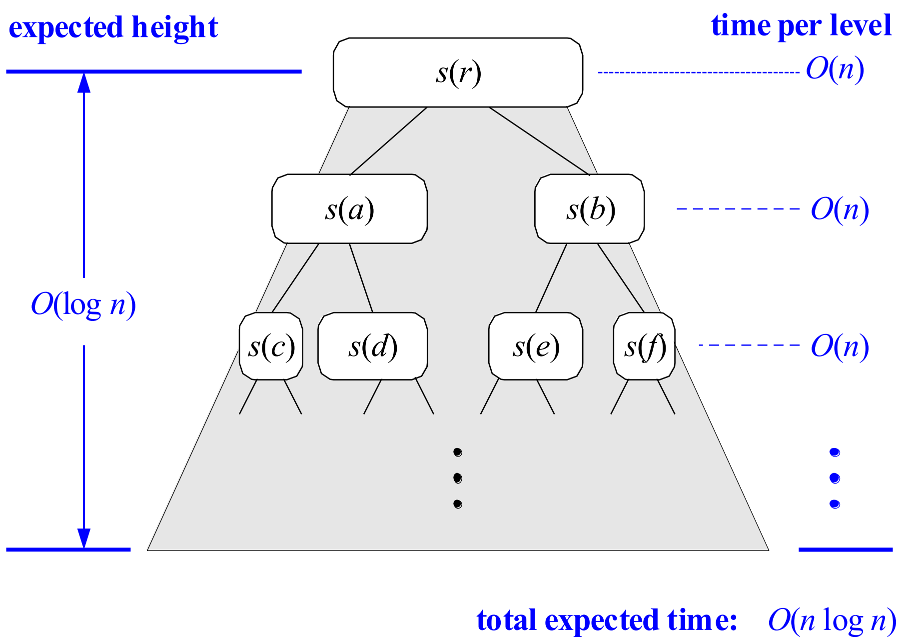


Bad pivots **Good pivots** **Bad pivots**

Expected Running Time, Part 2

- Probabilistic Fact: The expected number of coin tosses required in order to get $\boldsymbol{k}$ heads is $2\boldsymbol{k}$
- For a node of depth $\boldsymbol{i}$, we expect
  - $\boldsymbol{i}/2$ ancestors are good calls
  - The size of the input sequence for the current call is at most $(3/4)^{i/2}\boldsymbol{n}$

- Therefore, we have
  - For a node of depth $2\log_{4/3}\boldsymbol{n}$, the expected input size is one
  - The expected height of the quick-sort tree is $\boldsymbol{O}(\log \boldsymbol{n})$
- The amount or work done at the nodes of the same depth is $\boldsymbol{O}(\boldsymbol{n})$
- Thus, by the linearity of expectation, the expected running time of randomized quick-sort is $\boldsymbol{O}(\boldsymbol{n} \log \boldsymbol{n})$.



Instance Sensitivity

- Instance optimality algorithm analysis focuses on properties of input instances and differs from traditional worst-case analysis, which focuses on the maximum cost an algorithm might incur over all possible inputs of a given size.
- Instance optimality provides a finer-grained guarantee, ensuring that the algorithm performs well not just on average or in the worst case, but that it adapts to specific characteristics of individual inputs.
- Achieving instance optimality leverages favorable properties of the input data.

Insertion sort

- **insertion sort**: orders a list of values by repetitively inserting a particular value into a sorted subset of the list

- more specifically:
  - consider the first item to be a sorted sublist of length 1
  - insert the second item into the sorted sublist, shifting the first item if needed
  - insert the third item into the sorted sublist, shifting the other items as needed
  - repeat until all values have been inserted into their proper positions

Insertion sort

- Simple sorting algorithm.
  - n-1 passes over the array
  - At the end of pass i, the elements that occupied A[0]…A[i] originally are still in those spots and in sorted order.

2	15	8	1	17	10	12	5
0	1	2	3	4	5	6	7

after pass 2

2	8	15	1	17	10	12	5
0	1	2	3	4	5	6	7

after pass 3

1	2	8	15	17	10	12	5
0	1	2	3	4	5	6	7

Insertion sort code

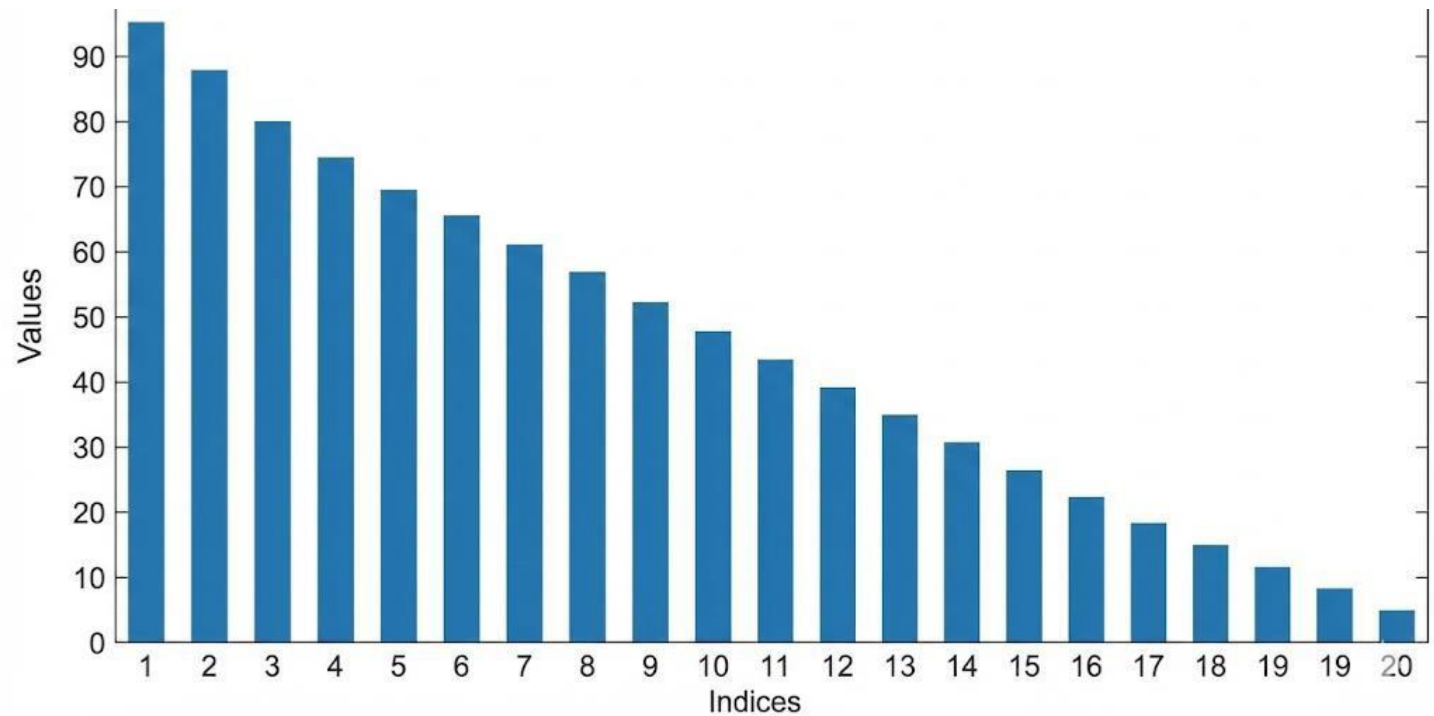
```
public static void insertionSort(int[] a) {
    for (int i = 1; i < a.length; i++) {
        int temp = a[i];

        // slide elements down to make room for a[i]
        int j = i;
        while (j > 0 && a[j - 1] > temp) {
            a[j] = a[j - 1];
            j--;
        }

        a[j] = temp;
    }
}
```

Worst-case Insertion-sort Analysis

- The worst case for for each loop of insertion sort is when the insertion of the i-th item takes O(i) time.
- This occurs for a list in reverse-sorted order.
- Worst-case time, therefore, is $O(n^2)$.

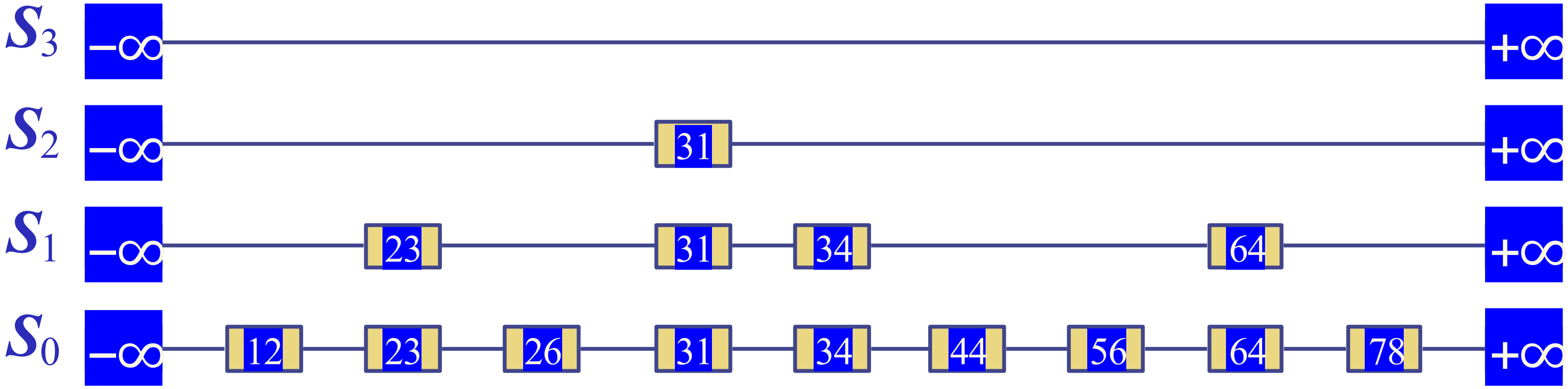


Instance Analysis of Insertion-sort

- An **inversion** in a permutation is the number of pairs that are out of order, that is, the number of pairs, (i,j), such that $i<j$ but $x_i>x_j$.
- Each step of insertion-sort fixes an inversion or stops the while-loop.
- Thus, the running time of insertion-sort is $O(n + Inv(X))$, where $Inv(X)$ is the number of inversions in the input, X.

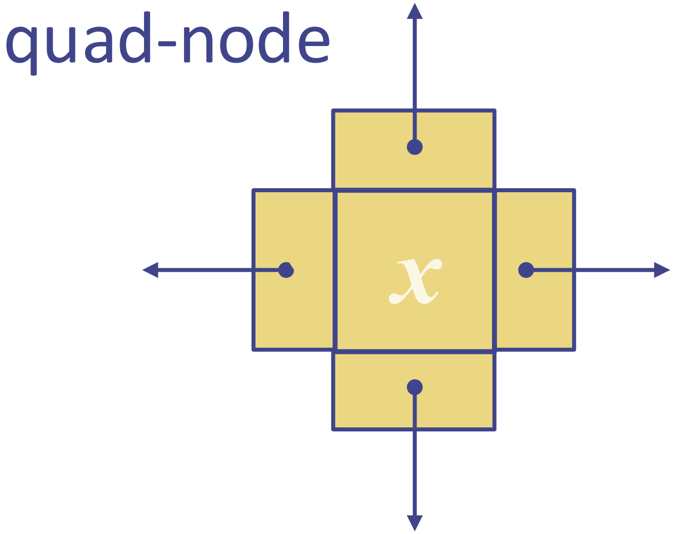
Skip List

- A skip list for a set S of distinct (key, element) items is a series of lists
  $S_0, S_1, \ldots, S_h$ such that
  - Each list S_i contains the special keys $+\infty$ and $-\infty$
  - List S_0 contains the keys of S in non-decreasing order
  - Each list is a subsequence of the previous one, i.e.,
    $$S_0 \supseteq S_1 \supseteq \ldots \supseteq S_h$$
  - List S_h contains only the two special keys, plus and minus infinity



Possible Implementation

- We can implement a skip list with quad-nodes
- A quad-node stores:
  - item
  - link to the node before
  - link to the node after
  - link to the node below
- Also, we define special keys PLUS_INF and MINUS_INF, and we modify the key comparator to handle them



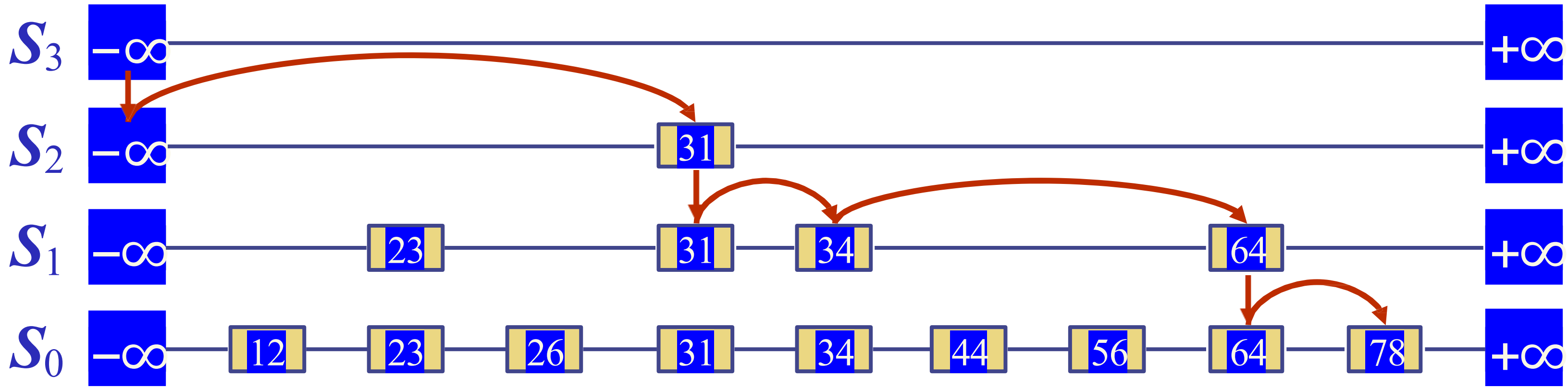
Top-Down Search

- We search for a key x in a a skip list as follows:
  - We start at the first position of the top list
  - At the current position p, we compare x with $y \leftarrow key(after(p))$

    $x = y$: we return $element(after(p))$

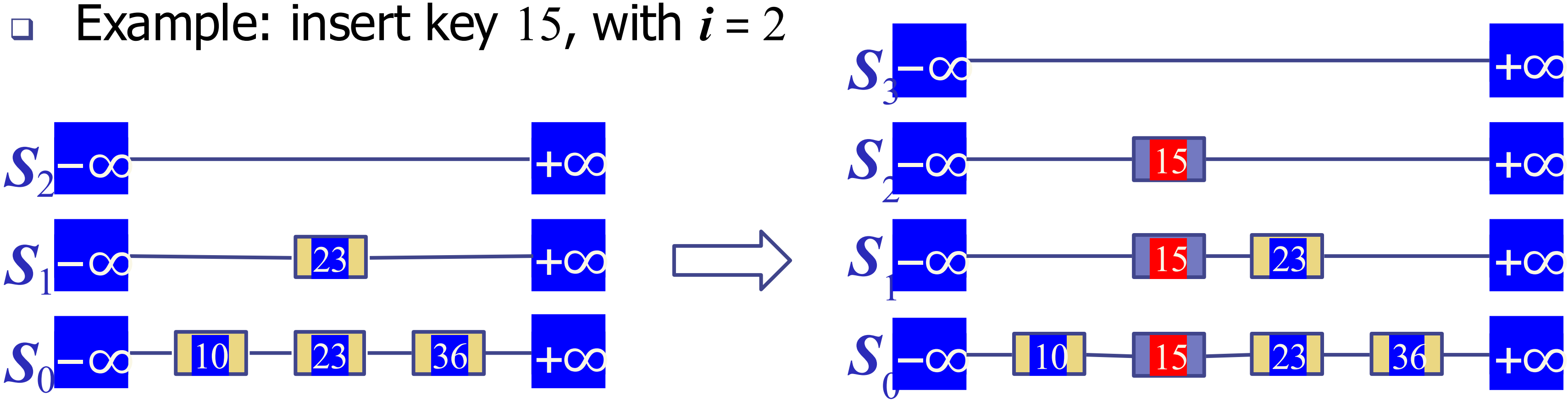
    $x > y$: we "scan forward"

    $x < y$: we "drop down"
  - If we try to drop down past the bottom list, we return *NO_SUCH_KEY*
- Example: search for 78



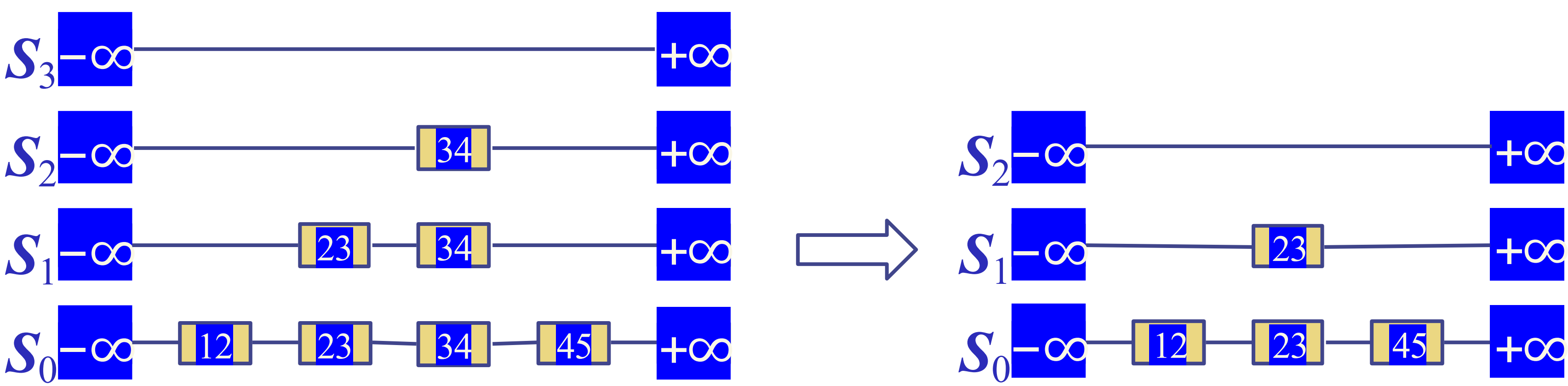
Insertion

- To insert an item (x, o) into a skip list, we use a randomized algorithm:
  - We repeatedly toss a coin until we get tails, and we denote with i the number of times the coin came up heads
  - If $i \geq h$, we add to the skip list new lists $S_{h+1}, \ldots, S_{i+1}$, each containing only the two special keys
  - We search for x in the skip list and find the positions $p_0, p_1, \ldots, p_i$ of the items with largest key less than x in each list $S_0, S_1, \ldots, S_i$
  - For $j \leftarrow 0, \ldots, i$, we insert item (x, o) into list S_j after position p_j
- Example: insert key 15, with $i = 2$



Deletion

- To remove an item with key x from a skip list, we proceed as follows:
  - We search for x in the skip list and find the positions $p_0, p_1, \ldots, p_i$ of the items with key x, where position p_j is in list S_j
  - We remove positions $p_0, p_1, \ldots, p_i$ from the lists $S_0, S_1, \ldots, S_i$
  - We remove all but one list containing only the two special keys
- Example: remove key 34



Space Usage

- The space used by a skip list depends on the random bits used by each invocation of the insertion algorithm
- We use the following two basic probabilistic facts:

  Fact 1: The probability of getting i consecutive heads when flipping a coin is $1/2^i$

  Fact 2: If each of n items is present in a set with probability p, the expected size of the set is np

- Consider a skip list with n items
  - By Fact 1, we insert an item in list S_i with probability $1/2^i$
  - By Fact 2, the expected size of list S_i is $n/2^i$
- The expected number of nodes used by the skip list is

$$\sum_{i=0}^{h} \frac{n}{2^i} = n \sum_{i=0}^{h} \frac{1}{2^i} < 2n$$

- Thus, the expected space usage of a skip list with n items is $O(n)$

Height

- The running time of the search an insertion algorithms is affected by the height h of the skip list
- We show that with high probability, a skip list with n items has height $O(\log n)$
- We use the following additional probabilistic fact:

  Fact 3: If each of n events has probability p, the probability that at least one event occurs is at most np

- Consider a skip list with n items
  - By Fact 1, we insert an item in list S_i with probability $1/2^i$
  - By Fact 3, the probability that list S_i has at least one item is at most $n/2^i$
- By picking $i = 3\log n$, we have that the probability that $S_{3\log n}$ has at least one item is at most

$$n/2^{3\log n} = n/n^3 = 1/n^2$$

- Thus a skip list with n items has height at most $3\log n$ with probability at least $1 - 1/n^2$

Search and Update Times

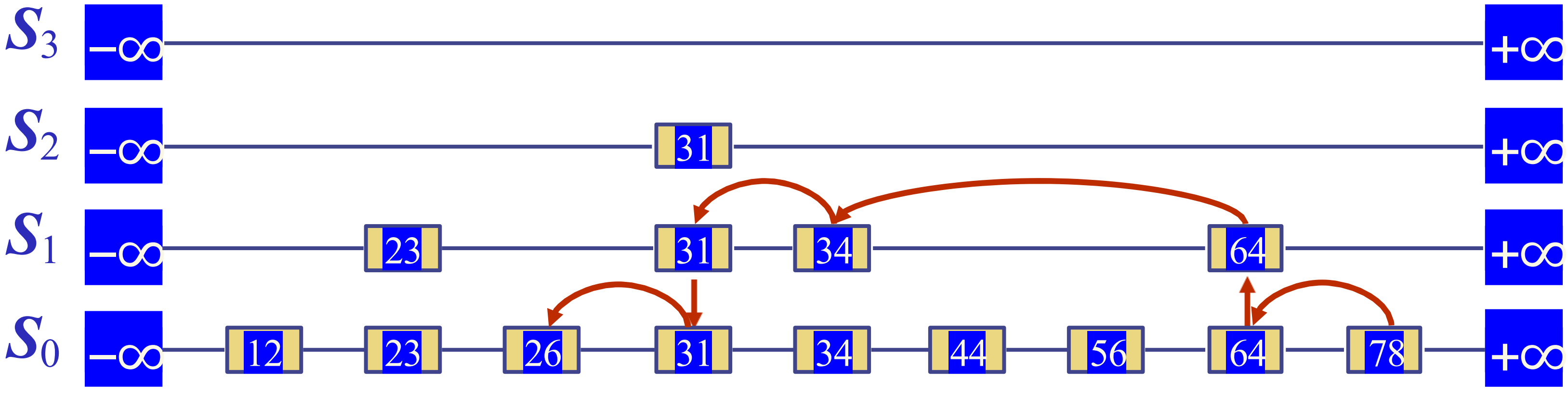
- The search time in a skip list is proportional to
  - the number of drop-down steps, plus
  - the number of scan-forward steps
- The drop-down steps are bounded by the height of the skip list and thus are $O(\log n)$ with high probability
- To analyze the scan-forward steps, we use yet another probabilistic fact:

  Fact 4: The expected number of coin tosses required in order to get tails is 2
- When we scan forward in a list, the destination key does not belong to a higher list
  - A scan-forward step is associated with a former coin toss that gave tails
- By Fact 4, in each list the expected number of scan-forward steps is 2
- Thus, the expected number of scan-forward steps is $O(\log n)$
- We conclude that a search in a skip list takes $O(\log n)$ expected time
- The analysis of insertion and deletion gives similar results

Up-Down Search

- We search for a key x in a a skip list as follows:
  - We start at the last bottom position of the top list
  - We move left and up until we reach a level with previous key less than the search key
  - Then we move down and left until we find the correct position

- Example: search for 26



Skip-List Sort

- Insert the elements x_1, x_2, ..., into a skip-list, doing up-down search from the bottom-left part of the skip-list for each element x_i.
- The expected time to insert x_i is O(log $d_i(X)$), where $d_i(X)$ is the distance in the bottom level to the place where x_i belongs.
- $d_i(X)$ is bounded by $I_i(X)$, where $I_i(X)$ is the number of inversions with x_i as the right element.

Analysis of Skip-List Sort

- The analysis of the expected running time uses the fact that the geometric mean is always at most the arithmetic mean:

$$
\begin{aligned}
c \sum_{i=1}^{|X|} (1 + \log[d_i(X) + 1]) \\
&= c|X| + c \log\left[\prod_{i=1}^{n} (d_i(X) + 1)\right] \\
&= c|X| + 2c|X| \log\left(\prod_{i=1}^{|X|} [I_i(X) + 1]\right)^{1/|X|} \\
&\leq c|X|\left(1 + 2 \log\left[\frac{Inv(X)}{|X|} + 1\right]\right).
\end{aligned}
$$

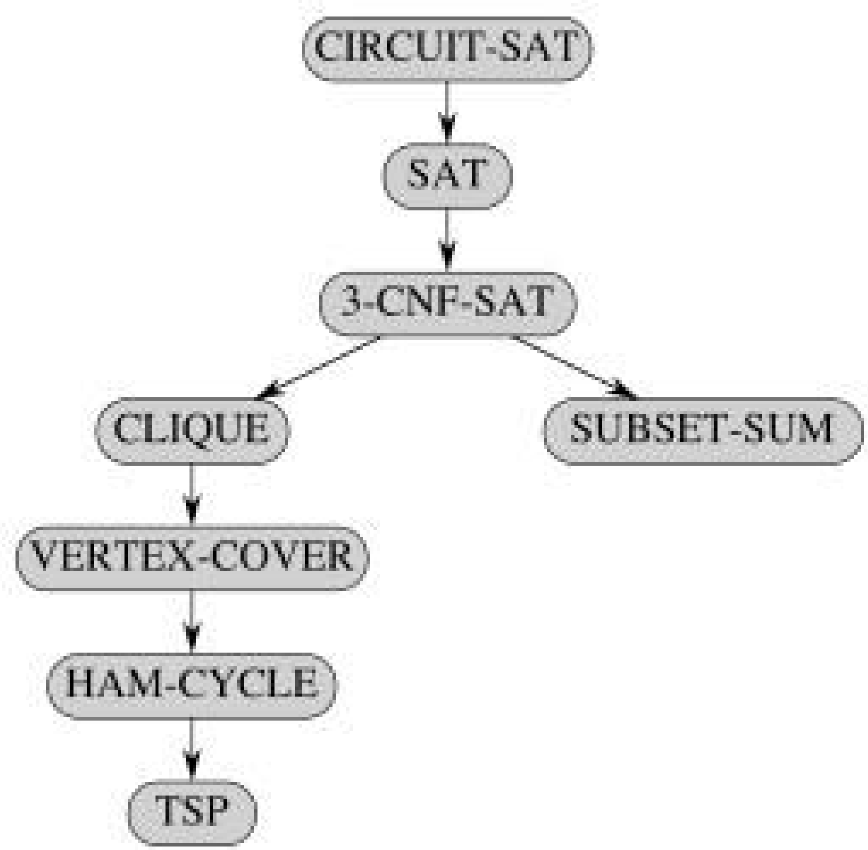
- So running time is O(n(1+ log (1+Inv(X)/n))).

Fixed Parameter Tractability

When standard worst-case analysis is not sufficiently informative, identify parameters of interest and perform worst case analysis with respect to these parameters.

Decorative image generated using Nano Banana

First, a review of NP-completeness…



Reductions for well-known NP-complete problems

Polynomial-Time Decision Problems

- To define a notion of "hardness," we will focus on the following:
  - Polynomial-time as the cut-off for efficiency
  - Decision problems: output is 1 or 0 ("yes" or "no")
    - Examples:
    - Does a text T contain a pattern P?
    - Is the sequence, S, in sorted order?
    - Is it possible to graduate with a Computer Science major from UCI in 3 years without any AP credits?

Problems and Languages

- A **language** L is a set of strings defined over some alphabet Σ
- Every decision algorithm A defines a language L
  - L is the set consisting of every string x such that A outputs “yes” on input x.
  - We say “A **accepts** x’ ’ in this case
    - Example:
    - If A determines whether or not a given graph G has an Euler tour, then the language L for A is all graphs with Euler tours.

The Complexity Class P

- A **complexity class** is a collection of languages
- P is the complexity class consisting of all languages that are accepted by **polynomial-time** algorithms
- For each language L in P there is a polynomial-time decision algorithm A for L.
  - If n=|x|, for x in L, then A runs in p(n) time on input x.
  - The function p(n) is some polynomial

The Complexity Class NP

- We say that an algorithm is non-deterministic if it uses the following operation:
  - Choose(b): chooses a bit b
  - Can be used to choose an entire string y (with |y| choices)
- We say that a non-deterministic algorithm A **accepts** a string x if there exists some sequence of choose operations that causes A to output "yes" on input x.
- NP is the complexity class consisting of all languages accepted by **polynomial-time non-deterministic** algorithms.

The Complexity Class NP Alternate Definition

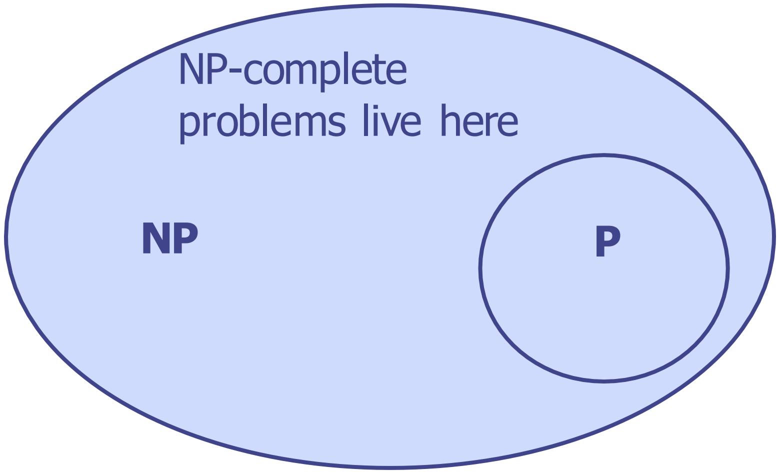
- We say that an algorithm B **verifies** the acceptance of a language L if and only if, for any x in L, there exists a certificate y such that B outputs “yes” on input (x,y).
- NP is the complexity class consisting of all languages verified by **polynomial-time** algorithms.

- We know: P is a subset of NP.
- Major open question: P=NP?
- Most researchers believe that P and NP are different.

NP-Completeness

- A language M is polynomial-time **reducible** to a language L if an instance x for M can be transformed in polynomial time to an instance x' for L such that x is in M if and only if x' is in L.
  - Denote this by M→L.
- A problem (language) L is **NP-hard** if every problem in NP is polynomial-time reducible to L.
  - Many optimization problems are NP-hard.
- A problem (language) is **NP-complete** if it is in NP and it is NP-hard.

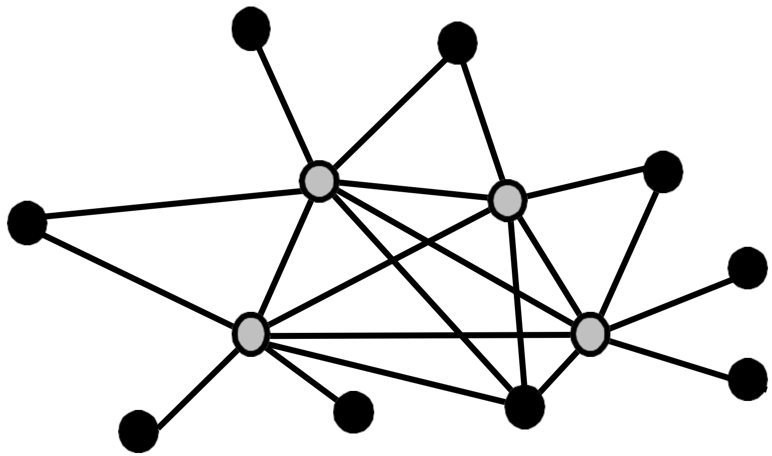
Some Thoughts about P and NP



- Belief: P is a proper subset of NP.
- Implication: the NP-complete problems are the hardest in NP.
  - Why: Because if we could solve an NP-complete problem in polynomial time, we could solve every problem in NP in polynomial time.
- That is, if an NP-complete problem is solvable in polynomial time, then P=NP.
- Since so many people have attempted without success to find polynomial-time solutions to NP-complete problems, showing your problem is NP-complete is equivalent to showing that a lot of smart people have worked on your problem and found no polynomial-time algorithm.
- If you prove or disprove the P=NP, you will win $1 million.
  - See https://en.wikipedia.org/wiki/Millennium_Prize_Problems

Vertex Cover

- A **vertex cover** of graph G=(V,E) is a subset W of V, such that, for every (a,b) in E, a is in W or b is in W.
- OPT-VERTEX-COVER: Given an graph G, find a vertex cover of G with smallest size.
- OPT-VERTEX-COVER is NP-hard.
- Very unlikely to be solvable for all inputs in polynomial time…



A parameter for vertex cover

- n vertices and m edges.
- k – the size of the smallest vertex cover.
- Can find the minimum vertex cover in time $\binom{n}{k}m \leq kn^{k+1}$ by exhaustive search.
- Polynomial in n for constant k.
- Can we have a polynomial time algorithm when k is a (slowly) growing function of n, like $\log n$?

A better algorithm for vertex cover

Let S denote an arbitrary vertex cover (unknown to us) of size k.

Pick an arbitrary uncovered edge.

Fork into two processes, each including a different endpoint of the edge in its vertex cover.

Necessarily, at least one of the forks includes only vertices of S.

At depth k, found S.

Running time $O(2^k kn)$. Polynomial for $k = O(\log n)$.

Fixed parameter tractability

A computational problem (e.g., vertex cover) is fixed parameter tractable (FPT) with respect to parameter k if it has an algorithm running in time O($f(k) \cdot n^c$).

- c is independent of n and k.
- f can be a fast-growing function (exponential, or even more).

Learning-Augmented Algorithms

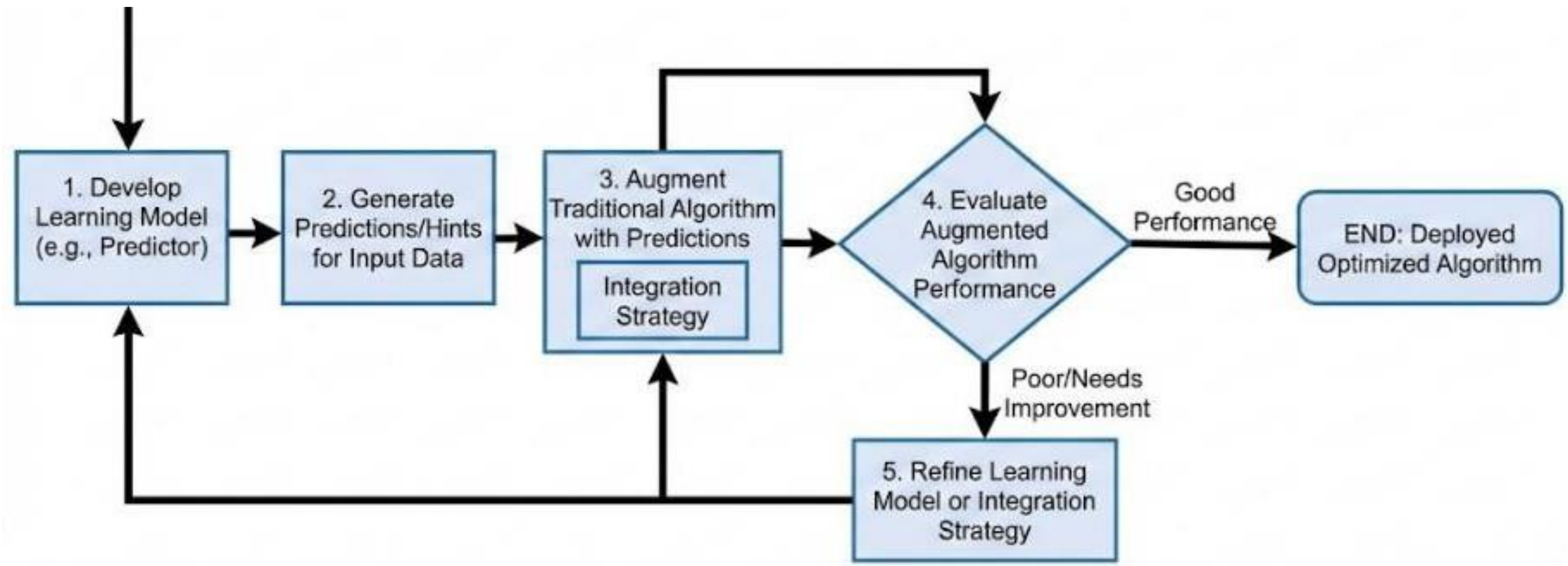
Decorative image generated using Nano Banana

Learning-Augmented Algorithms

- Learning-augmented algorithm design combines traditional algorithms with machine learning (ML) predictions to achieve better performance, by balancing the speed/accuracy of ML with the robustness of classical algorithms, providing strong guarantees even when predictions are imperfect.
- It uses ML to forecast future data or parameters, and the core algorithm adapts, aiming for near-optimal results with perfect predictions (consistency) but ensuring acceptable performance even with bad advice (robustness).

The Goal and Example Flow

- **The Goal**: To leverage a fast ML model for a given input and reliable, worst-case algorithms, creating algorithms that are both fast and robust.



Flowchart created using Nano Banana Pro

How it Works

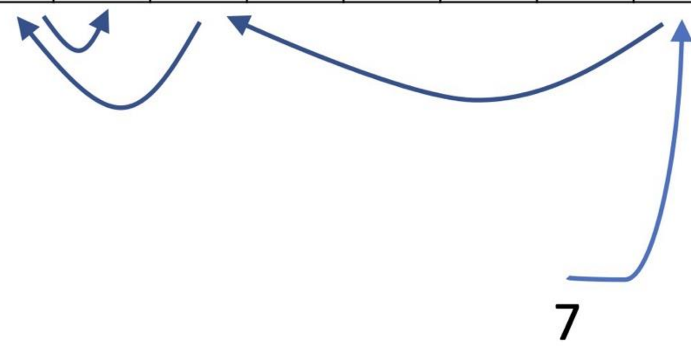
1. **Prediction**: An ML model (the "predictor") forecasts properties of the current or future problem instance (e.g., future data values, optimal parameters).
2. **Augmentation**: The online algorithm uses this prediction as extra information (advice).
3. **Adaptation**: The algorithm uses the advice to make better decisions, perhaps by computing the offline optimal solution for the predicted scenario or by using a hybrid strategy (like switching between following the prediction and a purely robust approach).

Learning-Augmented Algorithm

Motivating Example: Search

Given a sorted array of integers A[1...n], and a query **q** check if **q** is in the array.

2	4	7	11	16	22	37	38	44	88	89	93	95	96	97	98	99

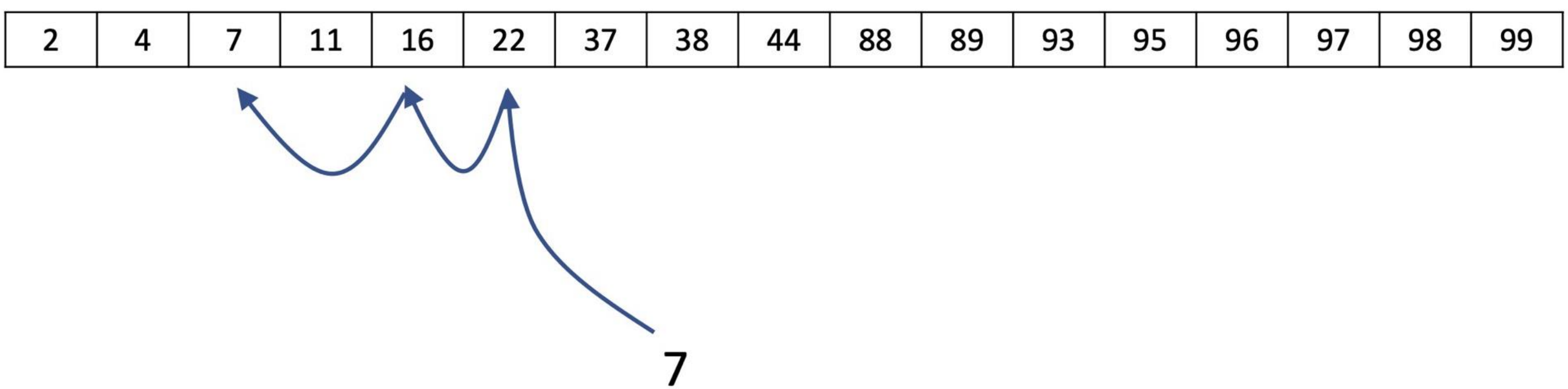


Binary Search

Learning-Augmented Algorithm

Motivating Example: Search

Given a sorted array of integers A[1...n], and a query **q** check if **q** is in the array.



Predict where **q** appears; use doubling binary search.

Learning-Augmented Algorithm

Search Costs

- Binary search: $O(\log n)$
- Prediction-based search: $O(\log |\text{prediction error}|)$
  - Plus time to do the prediction.
- Robust: In the worst case, prediction-based search is also $O(\log n)$
  - Not "worse" than binary search (at least symptotically)
- Consistent: In the best case (and even "near-best-case"), prediction-based search is constant-time.
  - Essentially optimal with perfect information.

Desired Properties

- **Consistency**: if predictions are correct, algorithm gives close to optimal solution.
- **Robustness**: Even under adversarial predictions, algorithm maintains a worst-case guarantee (ideally comparable to best known online algorithm).
- **Smoothness**: Performance degrades nicely in the error of the predictor.