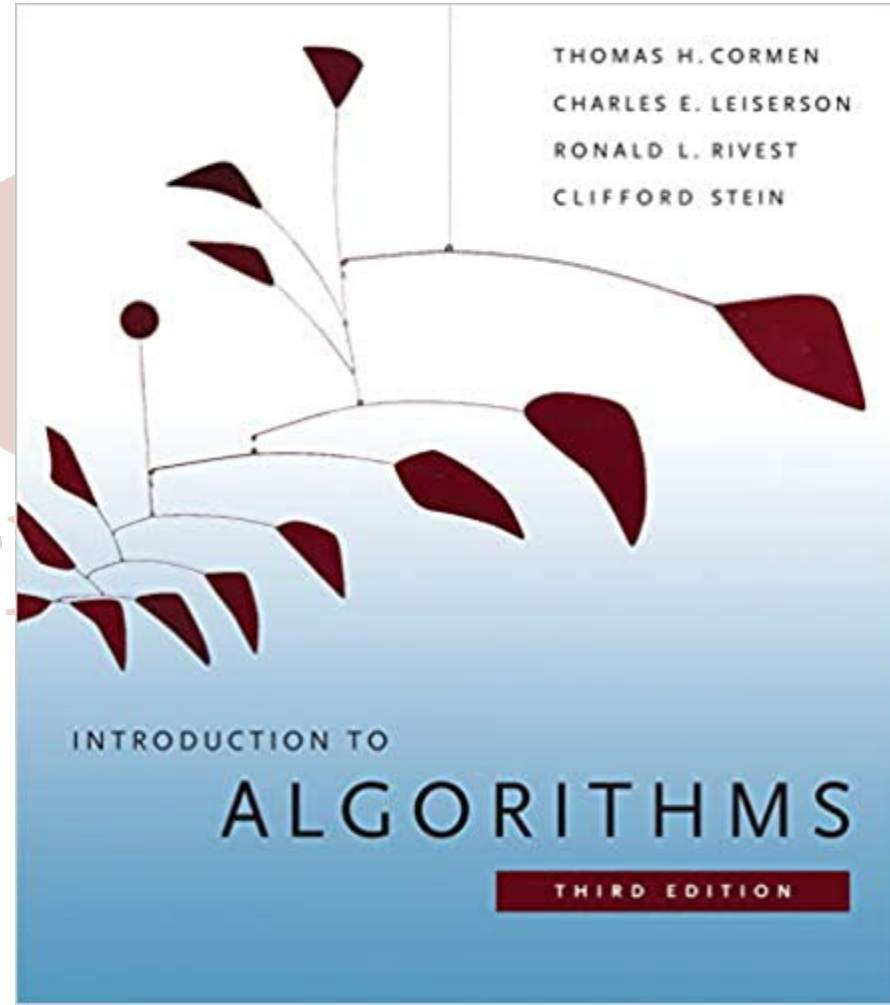


CS 07540 Advanced Design and Analysis of Algorithms

Week 5

- Amortized Analysis
 - Motivation
 - Aggregate Method
 - Accounting Method
 - Potential Method
- Information Theoretic Lower Bounds
 - Motivation
 - Comparison Sort



Amortized Analysis (Run-time Analysis)

So far, we have only analyzed the performance of algorithms in a **worst-case** scenario, for instance in:

- Search
 - Array – $O(1)$
 - BST – $O(h)$ (making a range of $O(1) - O(n)$)
 - Scapegoat Tree – $O(\log(n))$
- Delete
 - Array – $O(1)$
 - BST – $O(h)$
 - Scapegoat Tree – $O(n)$

Another way to provide a performance guarantee is to amortize cost by keeping track of the total cost of *all* operations, divided by the number of operations. In this setting, we can allow *some* expensive operations, while keeping the average cost of operations low. Hence, an **amortized analysis** is an attempt at a wholistic view at run time cost when an algorithm mixes some costly operations with other, mostly faster operations – it is about a sequence of operations. We are looking for a **worst-case average time**: amortized analysis is not average-case analysis – **probability is not involved**; an amortized analysis guarantees the **average performance** of each operation **in the worst case**.

It is the job of the algorithm **analyst** to figure out as much information about an algorithm as possible, and it is the job of the **programmer** who implements an algorithm to apply that knowledge to develop programs that solve problems effectively.

Toe in the Water

A **dynamic table** is a list data structure with random access (array) and variable size. It provides an interface for getting, adding, and removing items. Random access arrays have $O(1)$ run-time cost, just like arrays. However, a dynamic table needs to be resized when its size limit is reached. That is a costly operation. We want to account for the $O(1)$ run-time cost balanced with the $O(n)$ cost of resizing. This is an example of aggregate analysis which is one of the methods we will investigate. Do we know any [dynamic array/resizable array](#) structures?

- **ArrayList** in `java.util`
- **std::vector** in C++ STL
- `[]` in Python

Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual [capacity](#) greater than the storage strictly needed to contain its elements (i.e., its [size](#)). Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of [size](#) so that the insertion of individual elements at the end of the vector can be provided with **amortized constant time** complexity (see [push_back](#)).

Constructing objects with predefined properties often involves an unknown number of results at both compile time and run time. In order to collect these results, a dynamic table/list/array/set is the appropriate data structure. Here is an example with a C++ vector, used much like a stack. It needs to resize.

- What would be an appropriate strategy?
- What does that mean for insert performance?

```
typedef uint64_t PSK_index;
...
std::vector<PSK_index> packed_sequences;
for(PSK_index i : PSK_index_enumerator{radix, length}) {
    if(PSK_sequence.is_complementary_sequence(i) {
        packed_sequences.push_back(gcs);
    }
}
```

A possible strategy is to double the size of the array each time it needs to be resized. If we insert with **push_back**, then each insert is $O(1)$ until we need to resize the array (which means reallocate memory and copy the existing entries with $O(2 \cdot \text{size})$). So what is the “cost” of n insert operations? For simplicity in the calculations, let $n = 2^k$ be a power of 2.

$$\begin{aligned} (1 + 2 + 4 + 8 + \dots + n) + (n) &= \left(\sum_{i=0}^k 2^i \right) + 2^k \\ &= (2^{k+1} - 1) + 2^k \\ &= \mathbf{3n - 1} \end{aligned}$$

So even with copying, on average over n , the cost is still $O(1)$. 😊

A bad strategy is to increase the size one at a time:

$$\begin{aligned}(1 + 2 + 3 + 4 + 5 + \dots) + (n) &= \left(\sum_{i=0}^{n+1} i \right) + (n) \\ &= \frac{(n+1)(n+2)}{2} + n \\ &= \frac{1}{2}n^2 + \frac{5}{2}n + 1\end{aligned}$$

On average, this would be $O(n)$. 😞 The first strategy is obviously better (and gives us a better bound, which is part 2 of today's lecture).

These kinds of calculations are what we will look at today!

We will look at three methods:

- ***Aggregate analysis***

- Calculate upper bound on the total cost of a sequence of n operations and average over n .

- ***Accounting method***

- Assign costs to operations. “Overcharge” cheap early operations and use as “credit” for later, more expensive operations.

- ***Potential method***

- Similar to accounting method. Interpret charge as negative potential energy.

To illustrate each method, we will use ADT Stack and ADT Counter.

Aggregate Method

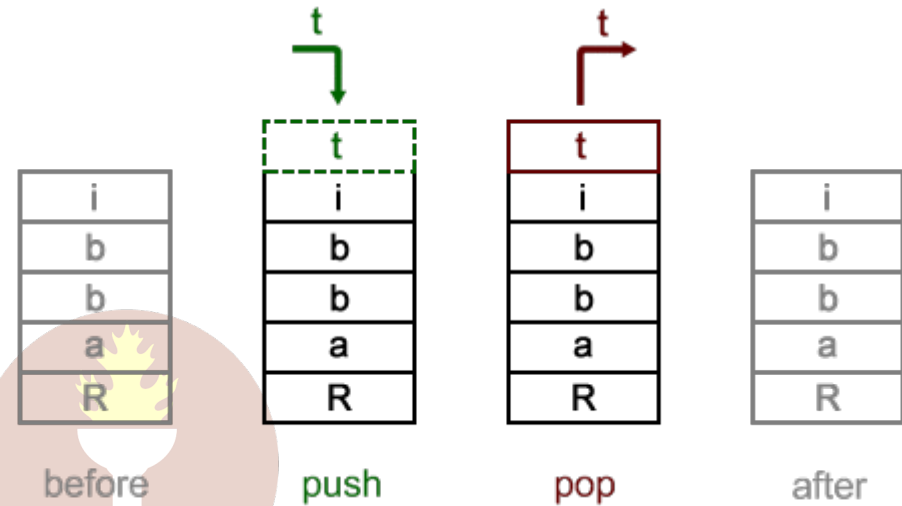
We want to determine an upper bound $T(n)$ on the total cost of a sequence of n operations (so this time n is not the size of data but the number of operations). The average, or ***amortized cost*** will then be $T(n) / n$.

This method is very straight forward. However, we will see that we need to have a concrete definition of how much our sequence will cost for it to be applicable.

ADT Stack

A stack is a linear data structure that stores items linearly (bottom to top). A new element is added at one end (top) and an element can be removed from that end only. The insert (at top) and remove (at top) operations are often called push and pop. These are the literal names in assembler.

For example, on x86 the stack is pointed to by **esp**, and **push eax** will move the content of register **eax** onto the stack while register **esp** is increased by 4 (size of **eax**). **pop eax** reverses that.



ADT Stack Implementation using Python List

The Python Documentation shows how to [implement stacks using Python lists](#).

size()	We can use the len function of list.
top()	The list accessor stack[-1] peeks the last entry, which is what top is supposed to accomplish.
push(a)	The list function append implements a stack push as we would want it.
pop()	The list function pop implements a stack pop as we would want it.

```
class listStack:
    def __init__(self, iList):
        if type(iList) != list:
            raise TypeError("stack needs to initialized with a list")
        self._stack = iList

    # Python empty objects are "false"
    def isEmpty(self):
        return not self._stack

    def size(self):
        return self._stack.__len__()

    def top(self):
        return self._stack[-1]

    def push(self, value):
        self._stack.append(value)

    def pop(self):
        return self._stack.pop()

    def __str__(self):
        return self._stack.__str__()

    def __repr__(self):
        return self._stack.__repr__()
```

ADT Stack Implementation with Java Array

Historically, stacks are implemented in CPUs using arrays as a call stack. There are problems with that approach. Since arrays are not resizable, too little stack space leads to a stack overflow. Too much stack space wastes resources.



```
public class stackIntArray implements stackInterface {
    private int[] stackArray;      private int topIndex;

    public stackIntArray(int size) {
        if (size <= 0)
            throw new IllegalArgumentException("stack size must be positive integer");
        stackArray = new int[size];      topIndex = -1;
    }

    public boolean isEmpty() {
        return topIndex == -1;
    }

    public void push(int value) throws ArrayIndexOutOfBoundsException
        stackArray[++topIndex] = value;
    }

    public int pop() throws EmptyStackException {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        else {
            return stackArray[topIndex--];
        }
    }
}
```

```
public int peek() throws EmptyStackException {  
    if (isEmpty()) {  
        throw new EmptyStackException();  
    }  
    else {  
        return stackArray[topIndex];  
    }  
}  
public void popAll() {  
    topIndex = -1;  
}  
}
```



We can use a stack to implement a crude delimiter check. In programming it is often important to match brackets (parentheses, brackets, curly brackets, angular brackets).

```
public static boolean hasBalancedParentheses(String msg) {
    final char[] openingBrackets = {'(', '[', '{', '<'};
    final char[] closingBrackets = {')', ']', '}', '>'};
    stackIntArray stack = new stackIntArray(msg.length());
    for (int i = 0; i < msg.length(); i++) {
        char c = msg.charAt(i);
        boolean isOpeningBracket = false;
        for (int b = 0; b < openingBrackets.length; b++) {
            if (c == openingBrackets[b]) {
                stack.push(b);
                isOpeningBracket = true;
            }
        }
        if (!isOpeningBracket) {
            for (int b = 0; b < closingBrackets.length; b++) {
                if (c == closingBrackets[b]) {
                    if (stack.isEmpty())
                        return false;
                    int s = stack.pop();
                    if (s != b)
                        return false;
                }
            }
        }
    }
    return stack.isEmpty();
}
```


Based on this sample implementation of a stack, consider the following stack operations:

Push (S, x) pushes object x on stack S

Pop (S) removes top object from stack S

Multipop (S, k) remove the top k objects from stack S , or all the object from the stack S if $k > \text{size}(S)$

What is the run-time of these operations?

Push (**S**, **x**) $O(1)$

Pop (**S**) $O(1)$

Multipop (**S**, **k**) $O(\min(k, \text{size}(n)))$

Traditional analysis of a run-time for a sequence of **Push**, **Pop**, and **Multipop** operations involves the worst-case **Multipop** (**S**, **k**) which is $O(n)$ if there are n elements on the stack. Worst-case sequence complexity is $O(n^2)$ if we have n **Multipop** operations.

Amortized Analysis of Stack

Any sequence of n **Push**, **Pop**, and **Multipop** operations will cost at most $O(n)$. The number of **Pop** operations, including **Multipop** must not exceed the number of **Push** operations.

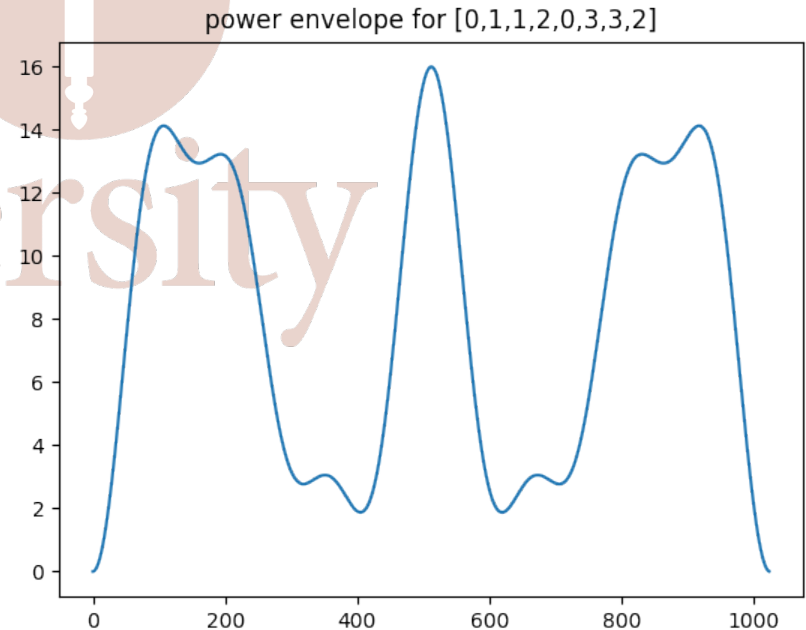
- Worst case scenario $(n - 1)$ Push followed by 1 **Multipop** $(n-1)$ means $2(n - 1)$ which is of order $O(n)$
- Amortized cost of each operation is $O(n) / n$ which is of order $O(1)$.

ADT Counter for radix^{length}

Phase-shift-keying sequences in digital communications can be represented as arrays of **length** values, each with $0 \leq \text{value} < \text{radix}$:

[0,1,1,2,0,3,3,2]

One way of encoding such sequences is as a base **radix** number with fixed **length** digits. Iterating through all such sequences is to **INCREMENT** with carry, but in base **radix**.



```

#include <cstdint>
#include <cstdint>
#include <cstdlib>
typedef uint64_t psk_index;

class psk_iterator {
public:
    class iterator {
        friend class psk_iterator;
    public:
        long int operator *() const { return __i; }
        const iterator &operator ++() { ++__i; return *this; }
        iterator operator ++(int) { iterator copy(*this); ++__i; return copy; }
        bool operator ==(const iterator &other) const { return __i == other.__i; }
        bool operator !=(const iterator &other) const { return __i != other.__i; }
    protected:
        iterator(psk_index start) : __i (start) { }
    private:
        psk_index __i;
    };
    iterator begin() const { return __begin; }
    iterator end() const { return __end; }
    psk_iterator(std::size_t radix, std::size_t length) : __begin((psk_index) 0), __end(max_index(radix, length)) {}
    psk_index max_index(std::size_t radix, std::size_t length) { ... } // HOMEWORK: POW(RADIX, LENGTH)
private:
    iterator __begin;
    iterator __end;
};

... // NOW WE CAN INVOKE WITH AUTO
    for(auto i : psk_iterator(4, 8)) {
        std::cout << i << std::endl;
    }

```

For binary counters of length k , it can be implemented as follows (from our book):

INCREMENT(A)

```
1   $i = 0$ 
2  while  $i < A.length$  and  $A[i] == 1$ 
3       $A[i] = 0$ 
4       $i = i + 1$ 
5  if  $i < A.length$ 
6       $A[i] = 1$ 
```

The cost of each **INCREMENT** operation is linear in the number of bits flipped. A single execution of **INCREMENT** takes $\Theta(k)$ in worst case (all-1 array). Hence, a sequence of n **INCREMENT** operations on an initially zero counter takes time $\Theta(nk)$ in worst case. However, not all bits are flipped for each **INCREMENT** call. How often is each bit flipped?

Our Calculus II skills tell us

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} \\ = 2n ,$$

which averages out to $O(1)$.



Accounting Method

In the ***accounting method***, we assign different charges to different operations, with some operations charged more or less than they actually cost. The amount charged is called amortized cost.

The accounting method overcharges some operations early in the sequence, storing the overcharge as “prepaid credit” on specific objects in the data structure. The credit is used later in the sequence to pay for operations that are charged less than they actually cost.

$$\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i, \forall \text{ sequences of } n \text{ operations}$$

\hat{c}_i – amortized run-time of i^{th} operation

c_i – actual run-time of i^{th} operation

The Accounting Method can be easily applied to multiple operation types to deduce the amortized cost of each. By contrast, the Aggregate Method will always treat every operation the same way and will not return different amortized costs for multiple types of operations in a sequence.



Amortized Analysis of Stack

Once more, consider a sequence of n **Push**, **Pop**, and **Multipop** operations. We charge each **Push** operation one extra run-time unit for future **Pop** operations. When we execute a **Pop** operation we charge it nothing, but will use credit stored by previous **Push** operations. Similarly, we charge **Multipop** operation nothing.

We assign the following amortized costs:

Push (S, x)	2
Pop (S)	0
Multipop (S, k)	0

Any sequence of n **Push**, **Pop**, and **Multipop** operations has total amortized cost of $O(n)$, as before. On average we get $O(1)$.

Amortized Analysis of Binary Counter

We are going to assign costs using the bit flips needed in an **Increment**. We charge an amortized(!) cost of 2 for setting a bit by setting the bit with a cost of 1 and putting aside a credit of 1.

INCREMENT(*A*)

```
1  i = 0
2  while i < A.length and A[i] == 1
3      A[i] = 0
4      i = i + 1
5  if i < A.length
6      A[i] = 1
```

Cost to set bit from 0 to 1 : 2

- An **Increment** sets at most one bit 0 to 1. Cost at most 2.
- A bit reset from 1 to 0 can use the credit from when it was set. Cost 0.
- The number of bits is never < zero, so we have non-negative credit.

Any sequence of n **Increment** operations has amortized cost $O(n)$.

Potential Method

In contrast to the accounting method, the ***potential method*** interprets prepaid work as “potential energy” of the whole data structure which can pay for future work. We need some notation.

- D_0 initial data structure
- c_i actual cost of operation $\#i$
- D_i data structure obtained after operation $\#i$ from D_{i-1}
- Φ potential function that maps data structure D_i to a real number $\Phi(D_i)$ which is the potential energy of that structure

Amortized cost of each operation is its actual cost plus the increase in potential due to operation.

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

The total amortized cost of n operations is

$$\sum_{i=1}^n \hat{c}_i = \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0)$$

where

$$\Phi(D_i) \geq \Phi(D_0), \forall i$$

Note that

$$\Phi(D_i) - \Phi(D_{i-1}) > 0$$

amortized cost represents overcharge of i th operation (an increase in the potential of data structure from D_i to D_{i-1}) which pays for the actual cost of the operation.



Amortized Analysis of Stack

Consider a sequence of n **Push**, **Pop**, and **Multipop** operations. Let Φ represent the number of items on the stack (that is a potential function).

Initially, when the stack is empty $\Phi(D_0) = 0$. The number of items on the stack is never negative, so $\Phi(D_i) \geq 0 = \Phi(D_0)$ for all i .

The potential difference after a **Push** is

- $s = \text{size}(S)$
- $\Phi(D_i) - \Phi(D_{i-1}) = (s + 1) - s = 1$

The amortized cost after a **Push** is

- $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (1) = 2$

The potential difference after a **Pop** operation is

- Cost **Pop** (**S**) containing s elements: $\min(1, s)$
- $\Phi(D_i) - \Phi(D_{i-1}) = (s - \min(1, s)) - s = -\min(1, s)$

Amortized cost after a **Pop** operation is

- $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = \min(1, s) + (-\min(1, s)) = 0$

The potential difference after a **Multipop** operation is

- Cost for **Multipop** (**S**, **k**) is $\min(1, s)$
- $\Phi(D_i) - \Phi(D_{i-1}) = (s - \min(s, k)) - s = -\min(s, k)$

Amortized cost after **Multipop** operation is:

- $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = \min(s, k) - \min(s, k) = 0$

Amortized cost of each operation is $O(1)$ and the total cost of n operations is $O(n)$.

Back to Dynamic Tables

A ***dynamic table*** is an ADT List implementation using an array that increases in size when it is full. After initialization with a certain size, we use a dynamic table to insert data until it is full. Then

- allocate memory for a larger table, typically twice the size of the old table.
- Copy the contents of old table to new table.
- Free the old table.

We want to analyze the cost of **Insert** and **Delete** operations.

Amortized Analysis of Dynamic Table

Load factor $\alpha(T)$ is the ratio between the number of items stored in the table to number of available slots. If the table is empty then $\alpha(T) = 1$.

Insert:

Double the table size if it is full upon insert operation. Consider a sequence of n **Insert** operations.

- Cost $c_i = i$ if $(i - 1)$ is an exact power of 2 (i.e., expansion)
- Cost $c_i = 1$, otherwise.

The total cost of n **Insert** operations is

$$\sum_{i=1}^n c_i \leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n + 2n = 3n$$

Thus, amortized cost of a single **Insert** is $O(1)$. Intuitively (using the accounting method) the cost of an **Insert** operation is 3 because

- Insert element e into a table
- Move e when table expands
- Move another item that has already been moved once after the table expands

Potential Method Analysis for Dynamic Table Expansion

We use

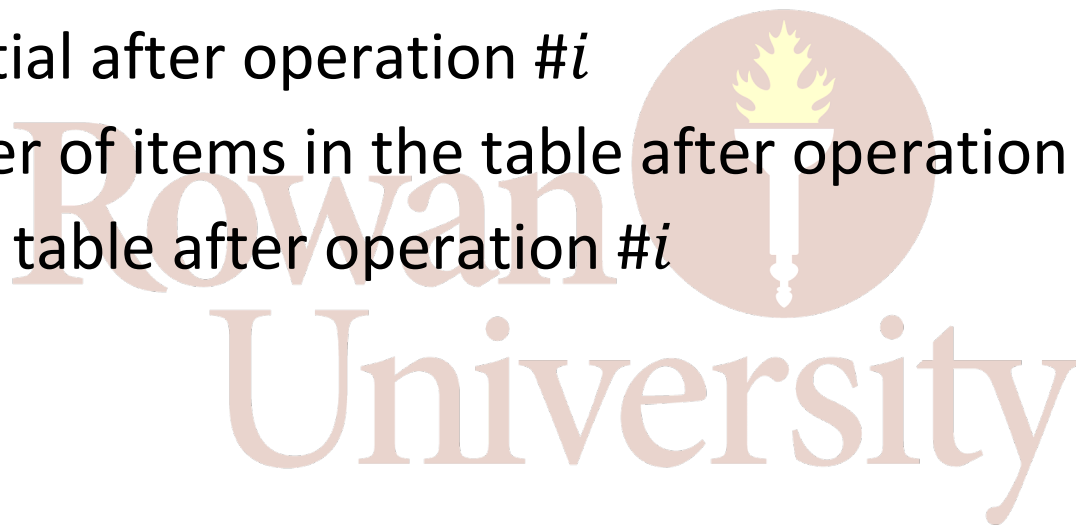
- $\Phi(T) = 0$ after expansion and
- $\Phi(T) = 2 T.\text{num} - T.\text{size}$ when expanding from size *num* to *size*.

Just before expansion, $T.\text{num} = T.\text{size}$ so $\Phi(T) = T.\text{num}$. At any point, $T.\text{num} \geq T.\text{size}/2$, so $\Phi(T) \geq 0$.

We show that amortized cost of an **Insert** operation is still 3 (meaning $O(1)$ on average)

Some notation.

- $c_{a,i}$ amortized cost of operation $\#i$
- c_i actual cost of operation $\#i$
- Φ_i potential after operation $\#i$
- $T.\text{num}$ number of items in the table after operation $\#i$
- $T.\text{size}$ size of table after operation $\#i$



No expansion

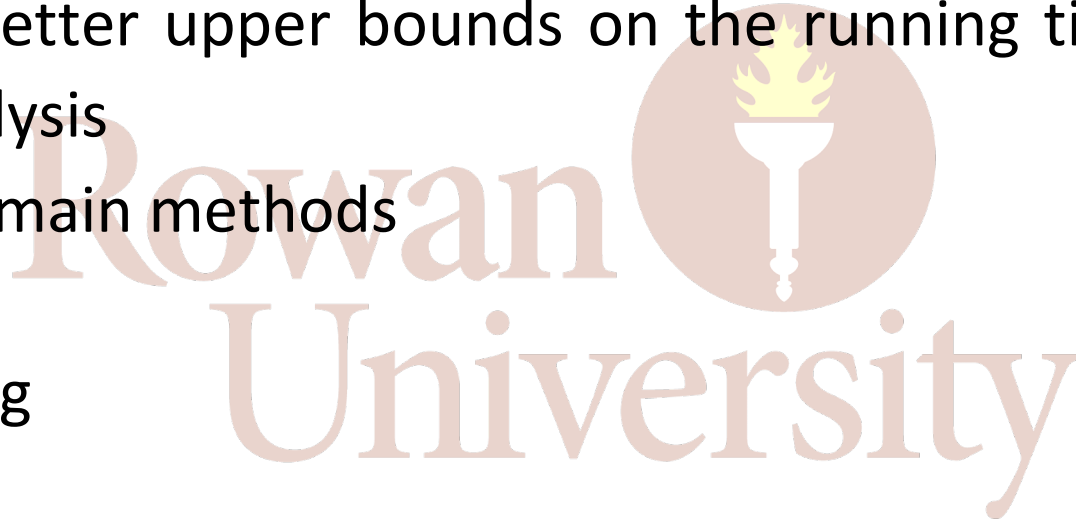
$$\begin{aligned}
 c_{a,i} &= c_i + \Phi_i - \Phi_{i-1} \\
 \square &= 1 + (2 T.\text{num}_i - T.\text{size}_i) - (2 T.\text{num}_{i-1} - T.\text{size}_{i-1}) \\
 \square &= 1 + (2 T.\text{num}_i - T.\text{size}_i) - (2 (T.\text{num}_{i-1} - 1) - T.\text{size}_{i-1}) \\
 \square &= 3
 \end{aligned}$$

With expansion

$$\begin{aligned}
 c_{a,i} &= c_i + \Phi_i - \Phi_{i-1} \\
 \square &= T.\text{num}_i + (2 T.\text{num}_i - T.\text{size}_i) - (2 T.\text{num}_{i-1} - T.\text{size}_{i-1}) \\
 \square &= T.\text{num}_i + (2 T.\text{num}_i - 2(T.\text{num}_i - 1)) \\
 \square &= -(2 (T.\text{num}_i - 1) - (T.\text{num}_i - 1)) \\
 \square &= T.\text{num}_i + (2) - (T.\text{num}_i - 2 + 1) \\
 \square &= T.\text{num}_i + 2 - T.\text{num}_i + 1 \\
 \square &= 3
 \end{aligned}$$


Amortized Analysis Summary

- analyzes a sequence of operations
- evaluates average (worst-case) cost
- usually gives better upper bounds on the running time compared to traditional analysis
- involves three main methods
 - aggregate
 - accounting
 - potential



Information Theoretic Lower Bounds

Selection Sort Example as Motivation

Selection sort is an in-place comparison sorting algorithm. It has an $O(n^2)$ time complexity, which makes it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity and has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited. 


```
import java.util.ArrayList;
import java.util.List;
import static java.util.Collections.swap;

public class SelectionSort {
    void sort(List<Integer> A) {
        for (int i = 0; i < A.size(); i++) {
            int smallestElementIndex = i;
            for (int j = i + 1; j < A.size(); j++)
                if (A.get(j) < A.get(smallestElementIndex))
                    smallestElementIndex = j;
            swap(A, smallestElementIndex, i);
        }
    }

    public static void main (String[] args){
        SelectionSort obj = new SelectionSort();
        List<Integer> A = new ArrayList<>(List.of(126,0,3, 215, 12, 22, 11));
        System.out.println("Given array");
        System.out.println(A);
        obj.sort(A);
        System.out.println("Sorted array");
        System.out.println(A);
    }
}
```

FileEditViewNavigateCodeRefactorBuildRunToolsVCSWindowHelpJava Sort - Main.java

Java SortMainmain

Project

Java Sortsources root, G:\My Drive\2023SP\CS075

.idea

out

Main

SelectionSort

External Libraries

Scratches and Consoles

SelectionSort.javaMain.java

```
1 import java.util.ArrayList;
2 import java.util.List;
3
4 public class Main {
5     public static void main (String[] args){
6         SelectionSort obj = new SelectionSort();
7         List<Integer> A = new ArrayList<>(List.of(126,0,3, 215, 12, 22, 11));
8         System.out.println("Given array");
9         System.out.println(A);
10        obj.sort(A);
11        System.out.println("Sorted array");
12        System.out.println(A);
13    }
14 }
15
```

Run: Main

```
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2022.2.3\lib\idea_rt.jar=65492:C:\Program Fi
Given array
[126, 0, 3, 215, 12, 22, 11]
Sorted array
[0, 3, 11, 12, 22, 126, 215]

Process finished with exit code 0
```

Version ControlRunTODOProblemsTerminalServicesBuild

Build completed successfully in 3 sec, 148 ms (2 minutes ago)

13:6CRLFUTF-84 spaces

If we start with a list of size n , we can analyze the loop structure and count the number of iterations. For $i = 0$ we get an inner loop of size n . For $i = 1$ we get $n - 1$, and so on. Since

$$\sum_{i=0}^{n-1} (n - i) = \frac{1}{2} (n - 1)n$$

we get an asymptotic runtime of $\left(\frac{1}{2}n^2 - \frac{1}{2}n\right)$ steps. Note that the n^2 term dominates for large n .

Formally we say that the worst-case runtime $T_A(X)$ of algorithm A with input X ($|X| = n$) is

$$T_A(n) = \max_{|X|=n} T_A(X)$$

Given a particular problem P (such as sorting), we first need an algorithm A that solves P before we can talk about runtime. Our algorithm is selection sort. With A being selection sort, the runtime T_A is $\frac{1}{2}(n^2 - n)$ for any input X of size n . The **worst-case complexity of a problem** P is the worst-case running time of the **fastest algorithm** for solving it.

$$T_P(n) = \min_{A \text{ solves } P} T_A(n)$$

$$T_P(n) = \min_{A \text{ solves } P} \left(\max_{|X|=n} T_A(X) \right)$$

When we describe an algorithm A that solves P in $O(f(n))$ time, we have an **upper bound** on the complexity of P .

$$T_P(n) \leq T_A(n) = O(f(n))$$

We have

- P is sorting
- A is selection sort (which is a particular implementation of sorting)
- $f(n) \frac{1}{2}n^2 - 1/2 n \sim n^2$

Hence an upper bound on the complexity of sorting n objects is n^2 , written $O(n^2)$.

But how do we know whether the algorithm we used to solve our problem is any good? Good compared to what? It would be nice to have both **upper** and **lower tight** bounds to evaluate an algorithm.


- $O(\cdot)$ describes the upper bound of the complexity.
- $\Omega(\cdot)$ describes the lower bound of the complexity.
- $\Theta(\cdot)$ describes the exact bound of the complexity (upper and lower are the same, up to scalars and constants).
- $o(\cdot)$ describes the upper bound excluding the exact bound.

If we have $f(n) = \frac{1}{2}n^2 - \frac{1}{2}n$, then

- $f(n)$ is $O(n^3)$ and also $O(n^2)$ but not $O(n)$
- $f(n)$ is $\Omega(n)$, but we can also argue for $\Omega(n^2)$ for larger n
- $f(n)$ is $\Theta(n^2)$
- $f(n)$ is $o(n^3)$

Usually, when we write $O(\cdot)$, we really want $\Theta(\cdot)$.

Sorting Complexity

Selection sort has complexity $O(n^2)$. Is this the best upper bound? **No.** ***Heap sort*** has complexity $O(n \log n)$ (as does ***merge sort***). All of these are general comparison sorting algorithms without additional assumptions on the data. 

We are going to prove **the worst-case runtime of comparison-based sorting is $\Omega(n \log n)$** soon.

RSA Encryption

A standard problem in cyber security is to send messages over unsecure channels. A solution based on number theory comes in the form of the famed RSA algorithm (by Rivest, Shamir, and Adleman). At its core, the encryption part turns a message M into an encrypted form C using

$$M^e \equiv C \pmod{N}$$

Here M is an integer (number), e is an integer, and C and N are integers. The message M^e is reduced to C modulo N . For example, with a pair $(e, N) = (5, 14)$ and a message $M = 3$ we would calculate $M^e = 3^5$ as 243, and $243 \equiv 5 \pmod{14}$ (243 has remainder 5 when divided by 14). So our encrypted message will be $C = 5$. For small values, we can calculate exponentiation in a loop.

Python has an efficient implementation of $M^e \bmod N$ with the **pow** function. It can be used as **pow(base, exp)** (standard C math library implementation) where **base** and **exp** are float or int variables and the result is the usual exponentiation, or as **pow(base, exp, mod)** where all variables are integers and the result will be an integer (remainder).

We will use an idea from Section 4.6.3 of Knuth, *Seminumerical algorithms*, 3rd edition, The Art of Computer Programming Vol 2, Addison-Wesley (1997) to investigate how to find a better algorithm for exponentiation than a plain loop.

Coding Exponentiation Using Binary Expansion

Let $n = 151$. Its binary representation is 10010111_2 .

$$151 = 2^7 + 2^4 + 2^2 + 2^1 + 2^0$$

Then

$$M^{151} = M^{2^7} M^{2^4} M^{2^2} M^{2^1} M^{2^0}$$

How can we use this to speed up exponentiation over the naïve loop?

```
def exponentiationMod(base, exponent, modulus):  
    result = 1  
    for i in range(exponent):  
        result *= base  
        result %= modulus  
    return result
```

Lower Bounds

Instead of looking at upper bounds, we will now argue that certain problems are hard, by proving lower bounds on their complexity. This is harder than proving an upper bound. It is no longer enough to examine a single algorithm.

To show that $T_P(n) = \Omega(f(n))$, we have to prove that **every** algorithm that solves P has a worst-case running time $\Omega(f(n))$ or worse.

When asked to implement a sorting algorithm with worst-case runtime $O(n \log(\log(n)))$, we want to be able to say: **No**. That is below the lower bound on sorting algorithms. Nobody can do that!

Why learn about lower bounds?

Since we are always interested in building faster algorithms, we want to know the limit. There is always a lower limit. Once we have an understanding of the limit of an algorithm's performance, we get insights about how to approach that limit and make the best algorithm.

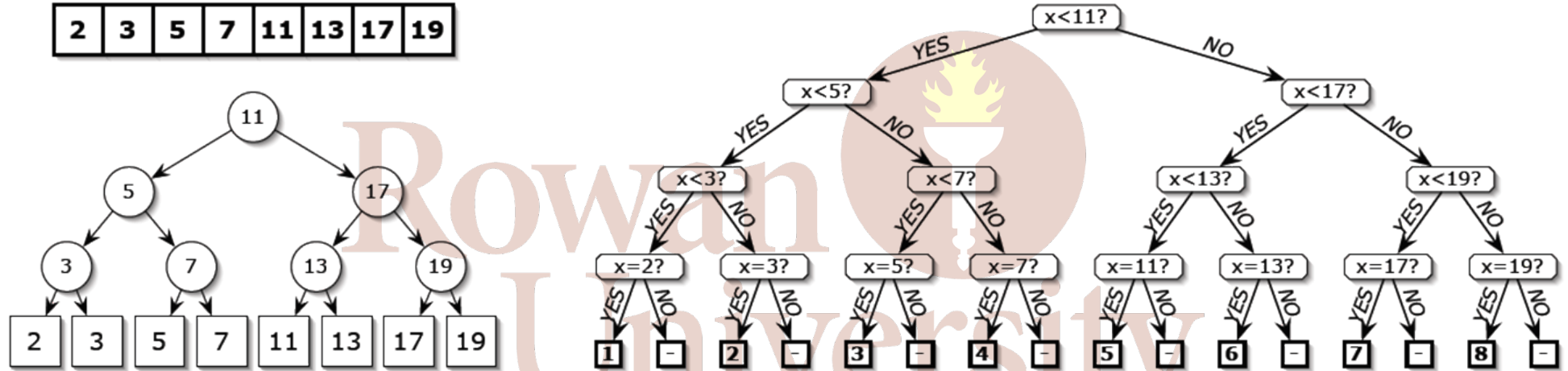
To derive lower bounds, we need to specify precisely what kind of algorithms we consider and how to measure their running time. This specification is called a model of computation. We will look at [decision trees](#), a powerful model of computation for the analysis of algorithms that only contain conditional control statements.

Decision Tree Model

A ***decision tree*** is a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size. Control, data movement, and all other aspects of the algorithm are ignored.

Each internal node is labeled by a query, which is just a question about the input. The edges out of a node correspond to the possible answers to that node's query. Each leaf of the tree is labeled with an output. The worst-case running time is the depth of the tree.

Two Examples



Left: A binary search tree for the first eight primes.

Right: The corresponding binary decision tree for the dictionary problem (– = 'none').

Create a decision tree for sorting a set $\{x, y, z\}$. Note that this decision tree needs to end with six leaves (one each for the six possible orders).



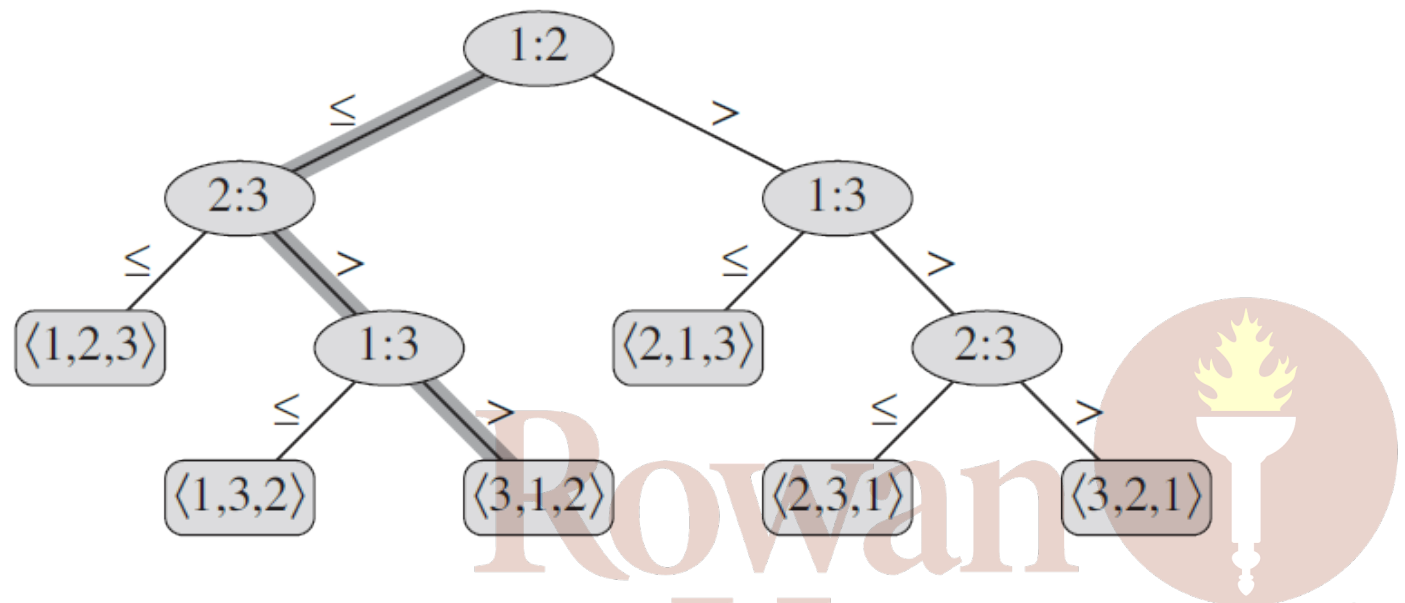


Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node annotated by $i:j$ indicates a comparison between a_i and a_j . A leaf annotated by the permutation $\langle \pi(1), \pi(2), \dots, \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \dots \leq a_{\pi(n)}$. The shaded path indicates the decisions made when sorting the input sequence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; the permutation $\langle 3, 1, 2 \rangle$ at the leaf indicates that the sorted ordering is $a_3 = 5 \leq a_1 = 6 \leq a_2 = 8$. There are $3! = 6$ possible permutations of the input elements, and so the decision tree must have at least 6 leaves.

The length of the longest simple path from the root of a decision tree to any of its reachable leaves represents the worst-case number of comparisons that the corresponding sorting algorithm performs. Consequently, the worst-case number of comparisons for a given comparison sort algorithm equals the height of its decision tree. A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf is therefore a lower bound on the running time of any comparison sort algorithm. The following theorem establishes such a lower bound.

Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case.

Proof

In a decision tree for sorting, each leaf corresponds to a reordering of the data, which is a permutation on the indices $0 - (n - 1)$. Suppose the height of the tree is h and there are ℓ reachable leaves. Since there are $n!$ possible permutations, we have

$$n! \leq \ell \leq 2^h$$

Taking logarithms on both sides (which works since logarithms are monotone functions), we see that

$$h \geq \log(n!) \sim \log(n^n) = n \log n$$

Hence the height of the tree is of order $n \log n$ and the corresponding algorithm is of complexity $\Omega(n \log n)$.

Now that we have an asymptotic lower bound, we can say that **heapsort and merge sort are asymptotically optimal comparison sorts**. That does not mean that in practical applications they are always faster than other sorting algorithms.



Sometimes decision trees with higher degrees are needed, say when a possible query is “Is x greater than, equal to, or less than y ?”. A ***k*-ary decision tree** is one where every query has (at most) k different answers.

Most lower bounds for decision trees are based on the following simple observation:

The answers to the queries must give you enough information to specify any possible output. If a problem has N different outputs, then any decision tree must have at least N leaves. It is possible for several leaves to specify the same output. If every query has at most k possible answers, then the depth of the decision tree must be at least:

$$\text{ceil}(\log_k N) = \Omega(\log N)$$

Comparison trees describe almost all sorting algorithms.

- Bubble sort
- Selection sort
- Insertion sort
- Shell sort
- Quicksort
- Heapsort (asymptotically optimal)
- Mergesort (asymptotically optimal)

But not radix sort or bucket sort.



Median Select

Given a set $\{x_1, x_2, \dots, x_n\}$ unordered numbers we want to find the k^{th} smallest number where $1 \leq k \leq n$. We assume no duplicates in the set but the problem can be easily extended to account for duplicates.

We could order the sequence using heapsort in $O(n \log n)$ and then pick index k . Optimal run-time of the sorting algorithm is $O(n \log n) + O(1)$. This does more than required, though, since we also order all other entries. So heapsort is probably not giving us a lower bound for the problem.

QuickSelect as a Median Selection Algorithm

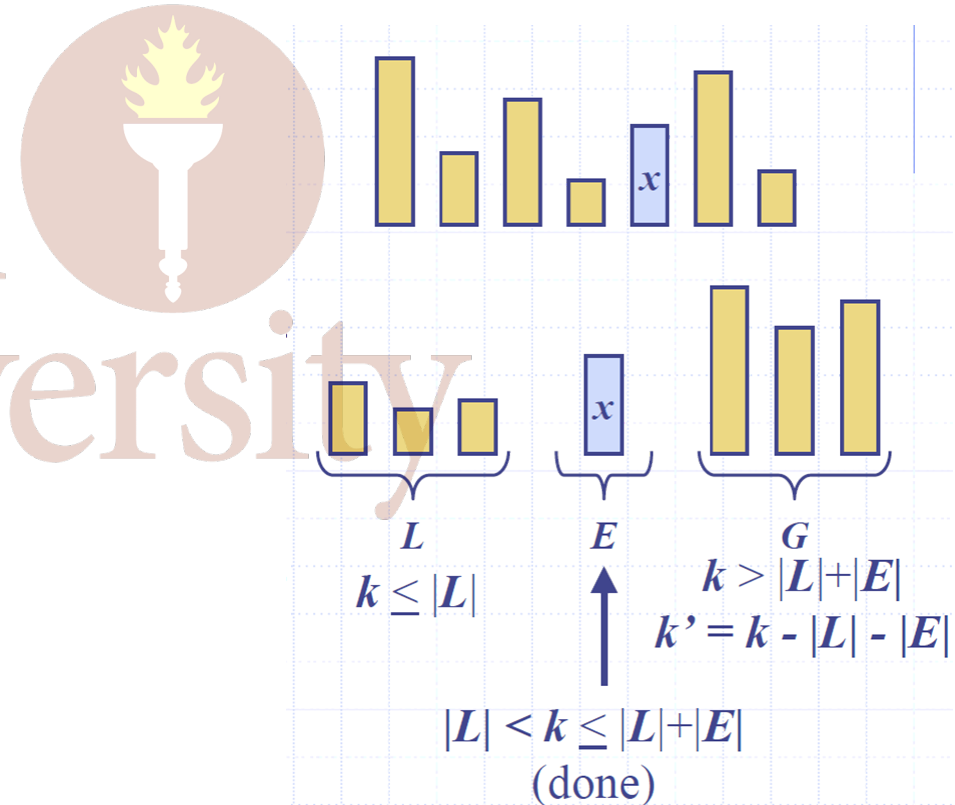
QuickSelect is an adaption of quicksort to find the k^{th} smallest element in an unordered set.

- Randomly pick a pivot element.
- Partition the set into two:
 - Elements greater than the pivot.
 - Elements less than the pivot.
- If the pivot is located in position k then the algorithm terminates. Otherwise recursively partition the set which must contain the k^{th} smallest element.

On average this algorithm takes $O(n)$, and in worst case $O(n^2)$.

Quick-select is a randomized selection algorithm. It uses the prune-and-search paradigm:

- Prune: pick a random element x (called pivot) and partition sequence S into
 - L elements less than x
 - E elements equal x
 - G elements greater than x
- Search: depending on k ,
 - the answer is in E , or
 - recursively repeat in L or G



In worst case, the pivot is always selected poorly so that all elements are smaller than the pivot, each time we will reduce the problem size by 1.

- Each partition requires $\Theta(n)$ steps
- $T(n) = T(n - 1) + \Theta(n) = O(n^2)$

On average, we can consider a pivot k and a fixed index i where we want to find the i^{th} smallest element.

$$T(n) = \frac{1}{n-1} \left(\underbrace{\sum_{k=1}^{i-1} T(n-k)}_{\text{pivot lies before } i} + \underbrace{\sum_{k=i+1}^n T(k-1)}_{\text{pivot lies after } i} \right) + \Theta(n) = O(n)$$

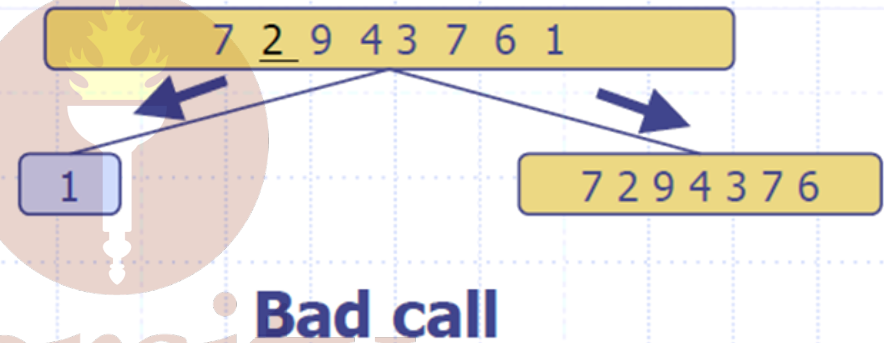
Linear Time Selection Algorithm as a Median Selection Algorithm

The Linear Time [Selection Algorithm](#) is an $O(n)$ algorithm by Blum, Floyd, Pratt, Rivest, and Tarjan which uses a divide and conquer strategy. There are some Factors that influence the run-time:

- Recursion – to get $O(n)$ we must reduce the run-time of recursion
- Finding a pivot – will increase to cost of finding a “good” pivot to reduce the cost of recursion
- Partition – will still take $O(n)$ time, no way to reduce this time

Consider a recursive call of quick-select on a sequence of size s .

- Good call: the sizes of L and G are each less than $(\frac{3}{4} \times s)$
- Bad call: one of L and G has size greater than $(\frac{3}{4} \times s)$



A call is good with probability $\frac{1}{2}$



The main idea is to pick the pivot that “lives” in the middle of the sorted list. We will need recursion on the part of the list that has at most $\frac{3}{4}$ of the size of the array, and we need to develop a selection algorithm that will always return a “good” pivot value. Then use quick select with a good pivot. The run time will be

$$T(n) = T(3/4 n) + \Theta(n) + \text{time to find good pivot}$$

Since

$$T(n) = T(c_1 n) + T(c_2 n) + \Theta(n) = O(n)$$

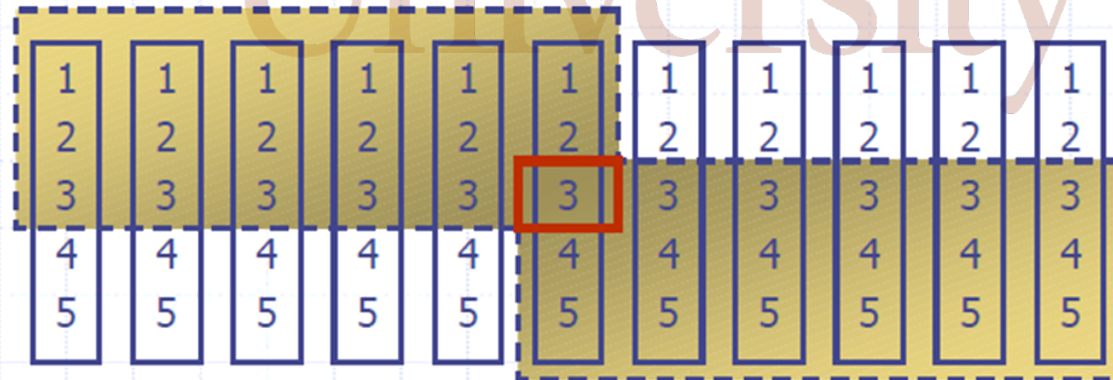
for $c_1 + c_2 < 1$, the time to find a “good” pivot should be $T(cn)$ where $c < 1/4$.

Finding a good pivot

Recursively use the selection algorithm itself to find a good pivot for quick-select:

- Divide S into $n/5$ sets of 5 each
- Find a median in each set
- Recursively find the median of the “baby” medians
- Use QuickSelect with median of the “baby” medians

Min size
for L



1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5	5	5	5

Min size
for G

- Step 1
 - If n is small, just sort and return the k^{th} smallest number. This will take constant time.
- Step 2
 - Group the given number in subsets of 5 in $O(n)$ time
- Step 3
 - Sort each of the group in $O(1)$ time. Total time is $O(n)$.
 - Find median of each group.
 - Recursively find the median of the medians.
- Step 4
 - Use QuickSelect with the median-of-medians

- Finding medians for $n/5$ subsets takes $O(n)$ time
- Finding median of medians takes $T(n/5)$ time
 - Recursive call on $n/5$ elements (medians)
- The recursive QuickSelect step will take $T(7n/10)$ time
 - Minimum number of elements greater than pivot is $3n/10$, or
 - Minimum number of elements smaller than pivot is $3n/10$
 - In worst-case we perform recursion on remaining $7n/10$ elements

Hence

- Finding medians for $n/5$ subsets takes $\Theta(n)$ time
- Finding median of medians takes $T(n/5)$ time
- The recursive QuickSelect step will take $T(7n/10)$ time
- $T(n) = T(7n/10) + T(n/5) + \Theta(n) = O(n)$
- If the size of the set is less than 80 then it is quicker to sort the set and pick i^{th} smallest element
- If the size of the set is greater than 80 then it is faster to do partition with the median find.

Information Theoretic Lower Bounds Summary

- An upper bound on a problem P can be established with any algorithm A that solves P .
- Comparison based search has a lower bound of complexity $\Theta(n \log n)$.
- Heapsort and merge sort are asymptotically optimal comparison sort algorithms.

