EC504 ALGORITHMS AND DATA STRUCTURES FALL 2020 MONDAY & WEDNESDAY 2:30 PM - 4:15 PM

Prof: David Castañón, dac@bu.edu

GTF: Mert Toslali, toslali@bu.edu

Haoyang Wang: haoyangw@bu.edu

Christopher Liao: cliao25@bu.edu

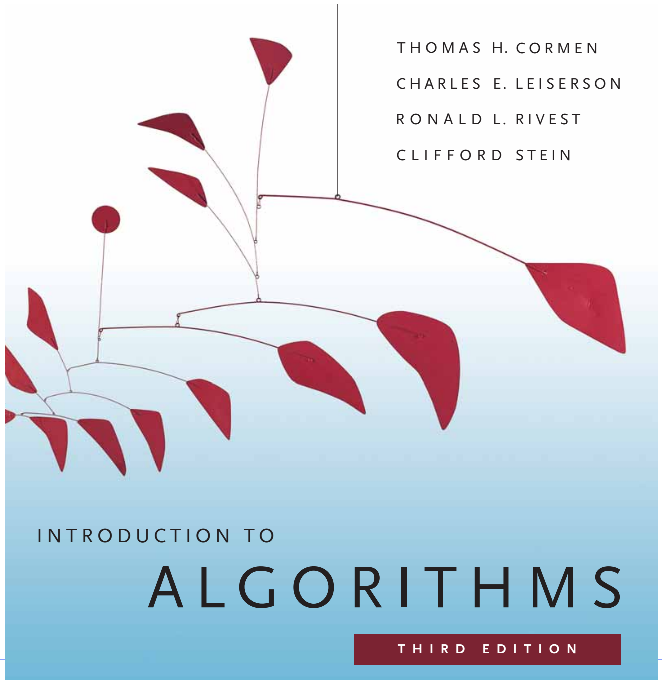
Course Organization

- Text:
  - Cormen, Leiserson, Rivest & Stein (CLRS), Fundamental text
    "Introduction to Algorithms" 3rd Edition MIT Pres
- Reference:

- Wikipedia
- Mark Allen Weiss "Data Structure and Algorithms in C++".
- Handouts on specific topics as needed

- Grading:

– HW (Written)	20%
– HW: Software	20%
– Project	20%
– Exams (2)	40%



Course Organization - 2

- Web Site in learn.bu.edu
- On-line classes
- Regular HW (not weekly; 8 or so during term)
- Office hours: TBD, or by appointment.
- Software: C++, using C++ 11 compatible
- Development: Accounts set up
- HW submission: On Blackboard, with code submitted separately
- Important: This is not a course to learn software. We use software to learn about data structures and algorithms, not the other way around…

Course Organization - 3

- Lectures are recorded, will be posted to web site within 5 hours
- In-class questions welcome
  - Unmute and ask
  - Or raise hand (less clear to instructor!)
- HW will have on-line component, hand-in component and software component
  - On-line short questions with instant feedback using Blackboard quiz
  - SW will provide main program, input files, and output files for debugging. You will develop needed functions.
- We will use Piazza for asynchronous question/answers at all hours
  - Enables all of you to benefit from answers

Course Organization - 4

- Course includes a project
  - Some topics will be suggested
  - Topics can be suggested by students
- Projects can be done in teams of up to 4 people
  - Or individually…Depends on how large a project one proposes
- Typical project: Develop a set of algorithms for solving a class of problems, implemented with different data structures, and evaluate the computation performance on realistic problem data.

Course Organization - 5

- Policy on Cooperation
  - HW: Discussions with classmates, colleagues and googling on web are allowed in general
  - But you have to write down the solution yourself and fully understand what you write
  - Your code must be your own: base it on what you read on the web, but don't copy verbatim
  - Plagiarism tools will be used extensively. Any violations will result in course failure and referral to College Academic Council

Course Organization - 5

- Policy on Cooperation
  - HW: Discussions with classmates, colleagues and googling on web are allowed in general
  - But you have to write down the solution yourself and fully understand what you write
  - Your code must be your own: base it on what you read on the web, but don't copy verbatim
  - Plagiarism tools will be used extensively. Any violations will result in course failure and referral to College Academic Council

What is EC504 About?

- Data structures store data in manners that support efficient algorithm computation.
- Algorithms with appropriate data structures can improve problem solution efficiency by orders of magnitude.
- In this course, we study both advanced algorithms and appropriate data structures for solution of practical problems.
- We will focus on rigorous analysis, realistic applications and implementation.
- Software implementations of algorithms/data structures will be part of the course.

Course Outline

1. Review: Algorithm Analysis (CLRS 2-4)
   - Asymptotic complexity
   - Recursions
2. Sorting: Classical and modern approaches (CLRS 4,9, Notes on TimSort)
3. Review: Trees and their Properties (CLRS App. B)
4. Efficient Search in one-dimension
   - Hash Tables (CLRS 11)
   - Balanced Search Trees (Red-Black, Splay, B-Trees, VEB trees) (CLRS 12,13,18,20)
   - Priority queues (Binary, Binomial, Fibonacci Heaps, VEB trees) (CLRS 6,19,20)

Course Outline - 2

6. Graphs and Network Optimization (CLRS 22)
   - Minimum spanning trees: Greedy algorithms (CLRS 23)
   - Shortest Paths: Greedy algorithms, Dynamic Programming (CLRS 24,25)
   - Max-flow: Ford-Fulkerson, Preflow-push (CLRS 26)
   - Min-cost flow: Assignment problems, auction algorithms, successive shortest path algorithms (Notes)
   - Applications (Notes)
7. Complexity Theory
   - NP-Complete Problems - Definition and examples (CLRS 34)
   - Approximation Algorithms - Knapsack, Traveling Salesperson, … (CLRS 35)
8. Advanced Topics

What is an algorithm? An unambiguous list of steps (program) to transform some input into some output.

- Pick a Problem (set)
- Find method to solve
  1. Correct for all cases (elements of set)
  2. Each step is finite ($\Delta\ t_{step}$ < max time)
  - Next step is unambiguous
  - Terminate in finite number of steps
- You know many examples:

  GCD, Multiply 2 N bit integers, …

Abu Ja'far Muhammid ibn Musa Al-Khwarizmi
Bagdad (Iraq) 780-850

Growth of Algorithm Run Time with Size n: Big "Oh"

- O(g(n)) : Set of functions of n that grow no faster than g(n) as n gets large

$$f(n) \in O(g(n)) \text{ if and only if there exists } c > 0, N_0 > 0$$
$$\text{such that} |f(n)| \leq c|g(n)| \text{ whenever } n > N_0$$

$$\text{Alternative: } f(n) \in O(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) \in O(g(n)), \text{ or } f(n) \text{ is } O(g(n)), \text{ or } f(n) = O(g(n))$$

$$f(n) \in o(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Growth of Algorithms: Other concepts

- o(g(n)) : grow strictly slower than g(n) $f(n) \in o(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
- Ω(g(n)) : grow no slower than g(n) as n gets large $f(n) \in \Omega(g(n))$ if and only if there exists $c > 0, N_0 > 0$ such that $|f(n)| \geq c|g(n)|$ whenever $n > N_0$
- ω(g(n)) : grow strictly faster than g(n) $f(n) \in \omega(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$
- Θ(g(n)) : grow similar to g(n): $f(n) \in O(g(n)), f(n) \in \Omega(g(n))$

Intuition

- $f = O(g)$
- $f = o(g)$
- $f = \Omega(g)$
- $f = \omega(g)$
- $f = \Theta(g)$

$\Leftrightarrow$

- $f \leq g$
- $f < g$
- $f \geq g$
- $f > g$
- $f = g$

Rules of thumb

- For polynomials, only the largest term matters.

$$a_0 + a_1 N + a_2 N^2 + \cdots + a_k N^k \in O(N^k)$$

- log N is in o(N)

Proof: As N ➔ 1 the ratio log(N)/N ➔ 0

- Some common functions in increasing order

1 log N √N N NlogN N^2 N^3 N^{100} 2^N 3^N N! N^N

- For r > 1, d > 0, $\Rightarrow n^d \in o(r^n)$
- For b > 1, r > 0, $\Rightarrow (\log(n))^b \in o(n^r)$

Some examples

- Which increases faster? What is the relation?
  - $(100n^2,\ 0.01 * 2^n)$
  - $(0.1 * \log n,\ 10n)$
  - $(10^{10}n,\ 10^{-10}n^2)$
  - $(0.01n^3 - 10,\ 100n^2)$
  - $(\ln(300n^4),\ \ln(n))$
  - $(n^4 + 3n^2 + 1,\ n^{0.01})$
  - $(2^n,\ 3^n)$
  - $(n^{10},\ n^n)$
  - $(n!,\ n^2)$ (Stirling's formula: $n! \approx (2\pi n)^{1/2}(\frac{n}{e})^n$)

Why is big-O important?

(processor doing ~1,000,000 steps per second)

input size \ time						
N	**10**	**20**	**30**	**40**	**50**	**60**
log n	3.3μsec	4.4μsec	5μsec	5.3μsec	5.6μsec	5.9μsec
n	10μsec	20μsec	30μsec	40μsec	50μsec	60μsec
n^2	100μsec	400μsec	900μsec	1.5msec	2.5msec	3.6msec
n^5	0.1sec	3.2sec	24.3sec	1.7min	5.2min	13min
3^n	59msec	48min	6.5yrs	385,500yrs	$2x10^8$ centuries...	
n!	3sec	$7.8x10^8$ millennia ……				

Non polynomial algorithms are terrible!
Logs are great!

Analysis of Algorithms

- Algorithms don't have constant run time for a given input size
  - Run time depends on nature of input
  - Best-Case, Average Case, Worst Case
  - Typically care about worst case (other cases useful in practice)

- Example: Insertion Sort(a[0:n-1]):

```
for (i=1;  i < n;  i ++){
     key = a[i]; j = i-1;
     while (j >= 0 && a[j] > key) {
          a[j + 1] = a[j];
          j = j - 1;
     }
     a[j + 1] = key;
}
```

OUTER LOOP TRACE FOR INSERTION SORT:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	(Swaps)
6 |	5	2	8	3	4	7	1	(1)
5 ←	→ 6							
5	6 |	2	8	3	4	7	1	(2)
	2 ←	→ 6						
2 ←	→ 5							
2	5	6 |	8	3	4	7	1	(0)
2	5	6	8 |	3	4	7	1	(3)
2	3	5	6	8 |	4	7	1	(3)
2	3	4	5	6	8 |	7	1	(1)
2	3	4	5	6	7	8 |	1	(7)
1	2	3	4	5	6	7	8	(17 total swaps)

Analysis of Algorithms

- Example: Insertion Sort(a[0:n-1]):

```
for (i=1;  i < n;  i ++){
     key = a[i]; j = i-1;
      while (j >= 0 && a[j] > key) {
           a[j + 1] = a[j];
           j = j - 1;
      }
      a[j + 1] = key;
}
```

- Analysis:
  - Best case: input is sorted, so while loop ends after 2 ops: $\Theta(n)$
  - Average case: while loop goes over 1/2 of list to find insert
  - Worst case: input is sorted in reverse, so while look ends at j = -1.

```
for ( i = 0; i< N; i++)
     for(j = i; j < N; j++){
       Sum = 0;
       for(k=i; k<j+1; k++)
           Sum += a[k];
       if(Sum > MaxSum)
           MaxSum = Sum;
    }
}
```

```
for ( i = 0; i< N; i++){
      Sum = 0;
      for(j=i; j<N; j++){
         Sum += a[j];
         if(Sum > MaxSum)
                MaxSum = Sum;
      }
}
```

Different Algorithm

```
MergeSort(int *array, int l, int r):
  if(l < r) {
      int m = l+(r-l)/2;
      mergeSort(array, l, m);
      mergeSort(array, m+1, r);
      merge(array, l, m, r);
   }
```

Recursive equation algorithm:

T(n): run time for a list of size n. m(n): time to merge arrays of length n

$$T(n) = 2T\left(\frac{n}{2}\right) + m(n)$$

Merge Algorithm

```
void merge(int *array, int l, int m, int r) {
   int i, j, k, nl, nr;
   nl = m-l+1; nr = r-m;
   int larr[nl], rarr[nr];
   for(i = 0; i<nl; i++)
      larr[i] = array[l+i];
   for(j = 0; j<nr; j++)
      rarr[j] = array[m+1+j];
   i = 0; j = 0; k = l;
   //marge temp arrays to real array
   while(i < nl && j<nr) {
      if(larr[i] <= rarr[j]) {
         array[k] = larr[i];
         i++;
      }else{
         array[k] = rarr[j];
         j++;
  }
  k++;
 }
```

```
  while(i<nl) {     //extra element in left array
    array[k] = larr[i];
    i++; k++;
  }
  while(j<nr) {    //extra element in right array
    array[k] = rarr[j];
    j++; k++;
  }
}
```

Mergesort Illustration

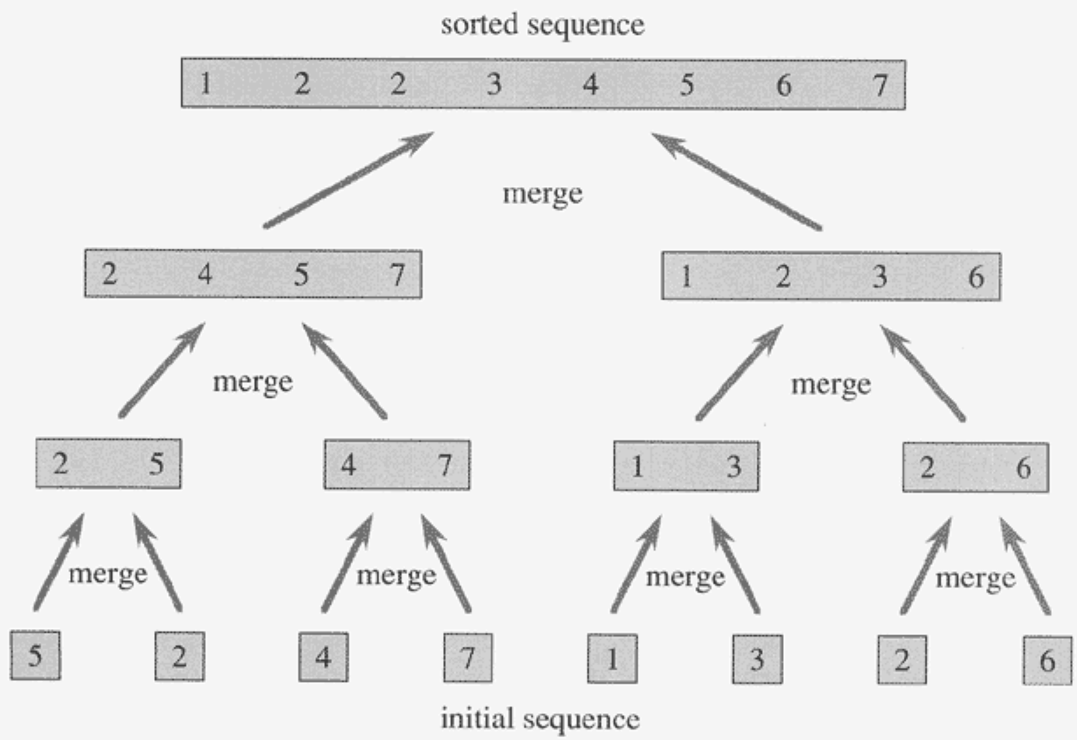


Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

MASTER EQUATION: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

Theorem: The asymptotic Solution is:

- Define $\gamma = \log_b(a) \rightarrow a = b^\gamma$
- Then, there are three cases

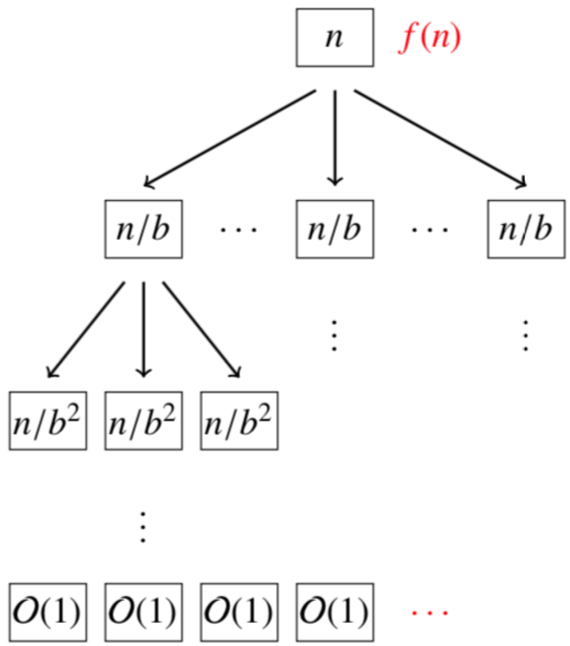
1. $T(n) \in \Theta(n^\gamma)$ if $f(n) \in O(n^{\gamma-\epsilon})$ for some $\epsilon > 0$
2. $T(n) \in \Theta(f(n))$ if $f(n) \in \Omega(n^{\gamma+\epsilon})$ for some $\epsilon > 0$
3. $T(n) \in \Theta(n^\gamma \log(n))$ if $f(n) \in \Theta(n^\gamma)$

BUILD TREE TO SOLVE

$$T(n) = aT(n/b) + f(n)$$

$$\boxed{n/b^h = 1 \implies h = \log_b(n)}$$

Height



$a \times f(n/b)$

$a^2 \times f(n/b^2)$

$\vdots$

$\underbrace{a^{\log_b(n)}}_{\text{number of leafs}} \times f(1)$

Breadth

$$\lim \frac{a^k f(n/b^k)}{n^\gamma} = ???$$

$$\boxed{T(n) = f(n) + af(n/b) + \cdots + a^{log_b(n)-1} f(b^2) + a^h T(1)}$$

LET'S ANALYZE FOR $n = b^h, f(n) \in O(n^k),$ so $f(n) = cn^k$

$$\boxed{T(n) = f(n) + af(n/b) + \cdots + a^{log_b(n)-1}f(b^2) + a^hT(1)}$$

$$T(n) = cn^k + ac(n/b)^k + a^2c(n/b^2)^k + \ldots + a^{h-1}c(b^2)^k + a^hT(1)$$

$$= cn^k + (\frac{a}{b^k})cn^k + (\frac{a}{b^k})^2cn^k + \ldots + (\frac{a}{b^k})^{h-1}cn^k + a^hT(1)$$

Note: $a^h = n^\gamma$

If $k < \gamma,$ then $b^k < a$. Case 1

If $k > \gamma,$ then $b^k > a$. Case 2

If $k = \gamma,$ then $b^k = a$. Case 3

Back to Mergesort

$T(n) = 2T(\frac{n}{2}) + m(n)$

$m(n) \in \Theta(n)$

Therefore, $a = 2,\ b = 2,\ \gamma = 1 \Rightarrow T(n) \in \Theta(n \log(n))$

Worst case for mergesort:

Average case for mergesort:

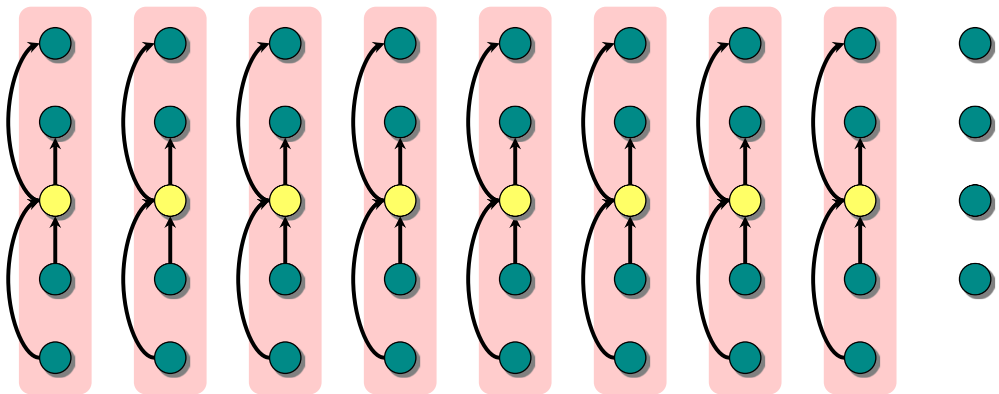
Best case for mergesort:

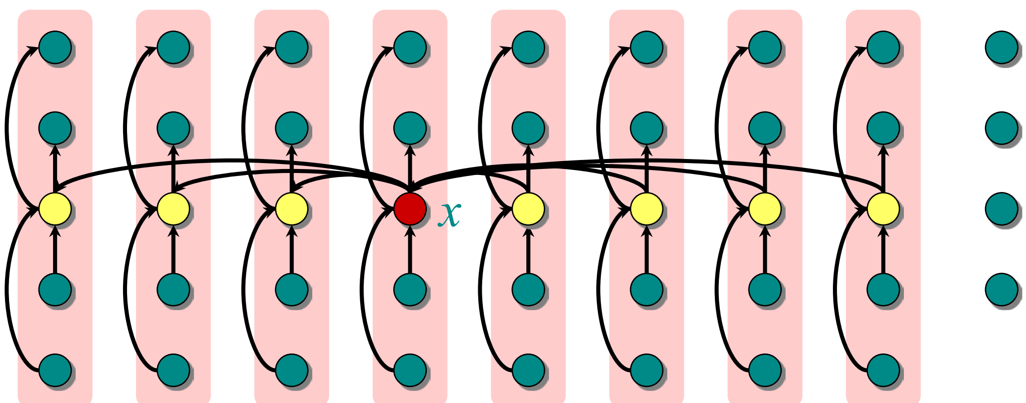
Order Statistics

- Select the i-th smallest of n elements (the element with rank i).
  - $i = 1$: minimum;
    $i = n$: maximum;
    $i = \lfloor (n+1)/2 \rfloor$ or $\lceil (n+1)/2 \rceil$: median.
- Naive algorithm: Sort and index ith element.
- Worst-case running time = $\Theta(n \lg n) + \Theta(1) = \Theta(n \lg n)$,
  - using merge sort or Timsort or …
- Can one do better?

Order Statistics

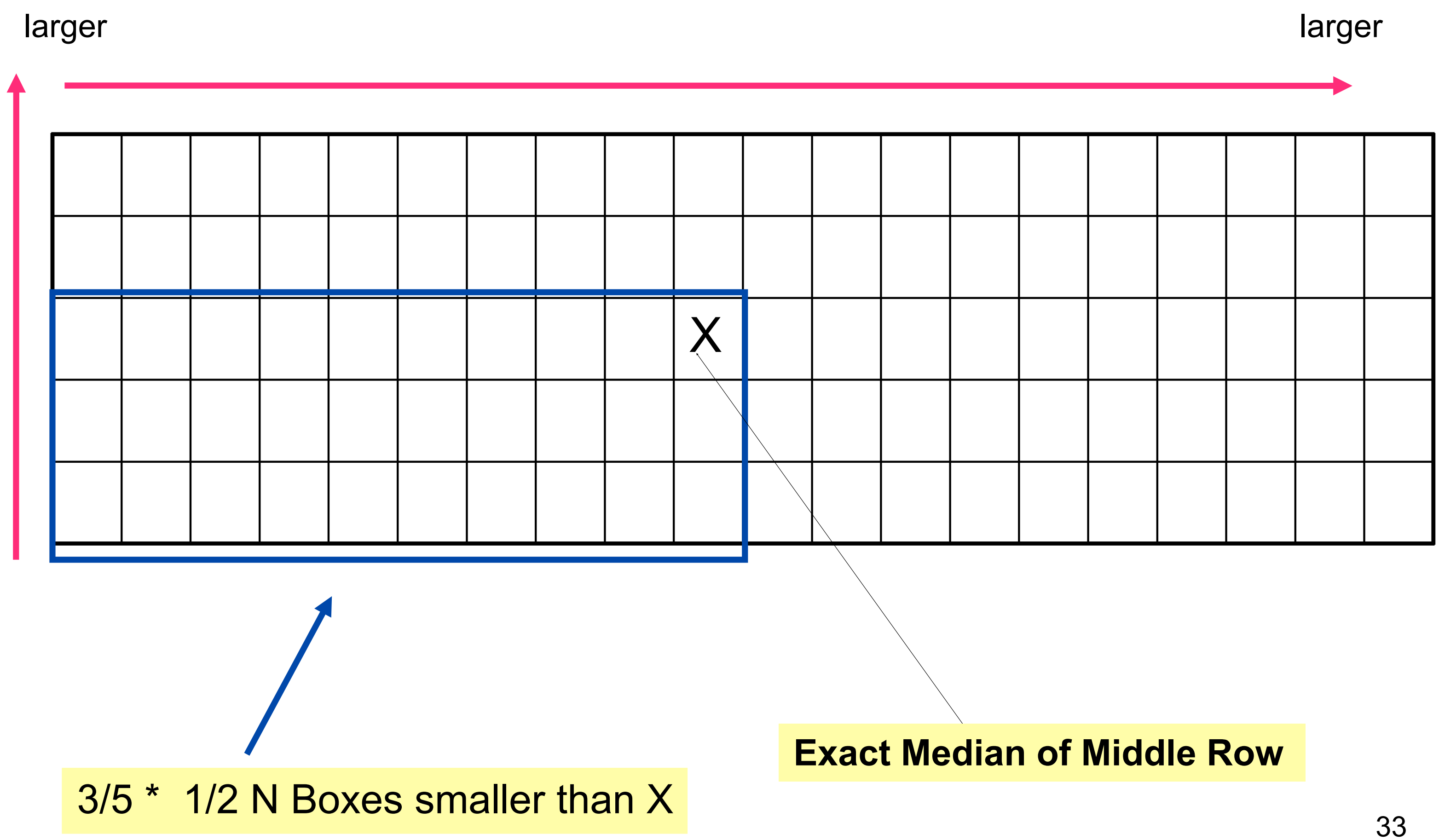
- SELECT(i, n)
  - Divide the n elements into groups of 5. Find the median of each 5-element group by rote.
  - Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.
  - Partition around the pivot x. Let k = rank(x).
  - if i=k then return x else if i < k
    - recursively SELECT the i-th smallest element in the lower part
  - else recursively SELECT the (i-k) th smallest element in the upper part





- At least half the group medians are $\leq x$, which is at least $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$ group medians. Therefore, at least $3 \lfloor n/10 \rfloor$ elements are $\leq x$.
- Similarly, at least $3 \lfloor n/10 \rfloor$ elements are $\geq x$.
- Recursion: $T(n) \leq T(n/5) + T(7n/10) + cn$
- Result: $T(n) \in \Theta(n)$

5 row of N/5 Columns



Generalization of Master's Equation

Let $\alpha_1, \alpha_2, \ldots, \alpha_k > 0$ be such that $\sum_{j=1}^{k} \alpha_j < 1$. Then,

If $T(n) = T(\alpha_1 n) + T(\alpha_2 n) + \cdots + T(\alpha_k n) + cn$, then $T(n) \in \Theta(n)$

Other useful recursive solutions

$T(n) = aT(n-1) + bT(n-2)$ (Discrete difference equation, constant coefficient)

Can solve via z-transforms (if you know EC 401..) or follow method below:

Solution form: k^n

Characteristic equation: Substitute into equation to get

$$k^n = ak^{n-1} + k^{n-2} \Rightarrow k^2 - ak - b = 0$$

Two solutions k_1, k_2 (or a repeated root).

If $k_1 \neq k_2$ then $T(n) = Ak_1^n + Bk_2^n$. Otherwise, $T(n) = Ak_1^n + Bnk_1^n$.

Example: Fibonacci numbers

$$F(n) = F(n-1) + F(n-2)$$

Characteristic equation:

Solutions:

Initial condition: F(0) = 0; F(1) = 1

.

Simple Data Structures: Abstract Data Type (ADT)

- In Object Oriented Programming data and the operations that manipulate that data are grouped together in classes

- Abstract Data Types (ADTs) or data structures or collections store data and allow various operations on the data to access and change it

- High level languages like C++, Java, Python often provide implementations of ADTs

The List ADT

- List of size N: $A_0, A_1, \ldots, A_{N-1}$
- Each element A_k has a unique position in the list
- Elements can be arbitrarily complex
- Operations
  - Depend on what the list is for
- Implementation: either as arrays or as linked lists

Sample List (C++) ADT

```
class List {
  public:
  List();                           // constructor
  List(const List& list);           // copy constructor
  ~List();                          // destructor
  List& operator=(const List& list);  // assignment operator

  bool empty() const;               // boolean function
  void addHead(double x);           // add to the head
  double deleteHead();              // delete, get head

  void addEnd(double x);            // add to the end
  double deleteEnd();               // delete, get end

  bool findNode(double x);        // search for a given x
  void deleteNode(double x);        // delete node with value x

  int length() const;               // count the number of
 elements

};
```

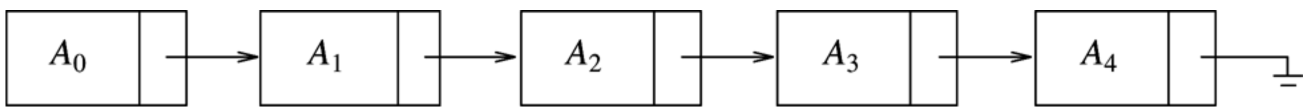
Lists Using Arrays

- Operations
  - addHead(X) – O(N)
  - deleteHead() – O(N)
  - addEnd(X) – O(1)
  - deleteEnd() – O(1)
  - findNode(X) – O(N)

Linked Lists

- Elements not stored in contiguous memory

- Nodes in list consist of data element and next pointer



Lists Using Linked Lists

- Operations
  - addHead(X) – O(1)
  - deleteHead() – O(N)
  - addEnd(X) – O(N)
  - deleteEnd() – O(N)
  - findNode(X) – O(N)

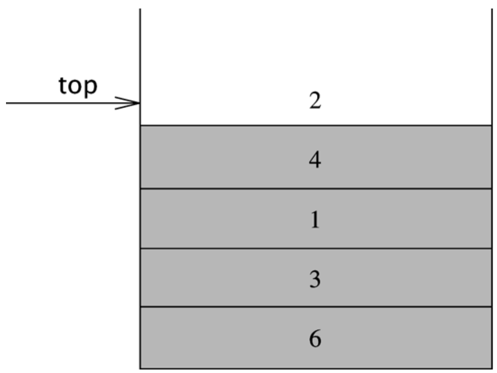
- Can modify using doubly-linked lists, pointers to end
  - Make addEnd(X), deleteEnd O(1).

List Applications: Stacks

- Stack is a list where insert and remove take place only at the “top”

- Operations

  - Push (insert) element on top of stack
  - Pop (remove) element from top of stack
  - Top: return element at top of stack



- LIFO (Last In First Out) queue
- Stack is a list, any list implementation will do, but will have different advantages

Stack ADT

```
class Stack {
  public:
      Stack();                          // constructor
      Stack(const Stack& stack); // copy constructor
      ~Stack();                         // destructor

      bool empty() const;
      void push(const double x);
      double pop();                     // change the stack

      double top() const;   // keep the stack unchanged

  private:
};
```

Stack using linked lists

```
struct Node{
   public:
        double data;
        Node* next;
};

class Stack {
   public:
        Stack();                  // constructor
        Stack(const Stack& stack); // copy constructor
        ~Stack();                 // destructor

        bool empty() const;
        void push(const double x);
        double pop();             // change the stack

        bool full(); // unnecessary for linked lists
        double top() const;       // keep the stack unchanged
        void print() const;

   private:
        Node* top;
};
```

Stack using arrays

```
class Stack {
public:
	Stack(int size = 10);			//
constructor
	~Stack() { delete [] values; }		// destructor

	bool empty() { return top == -1; }
	void push(const double x);
	double pop();

	bool full() { return top == maxTop; }
	double top();
	void print();
private:
	int maxTop;		// max stack size = size - 1
	int top;		// current top of stack
	double* values;	// element array
```

Array versus Linked List Implementation

- push, pop, top are all O(1) constant-time operations in both array and linked list implementation
  - For array implementation, the operations are performed in very fast constant time
- Linked List structures are larger - need pointers
- Array allocate space contiguously - useful for cache, etc.
- Arrays may have to be resized (lots of data movement…) but ***amortized analysis*** says worst case is still O(1)!

Stack Application: Balancing Symbols

- To check that every right brace, bracket, and parentheses must correspond to its left counterpart (e.g. MATLAB, code syntax checkers, …)
  - e.g. [( )] is legal, but [( ]) is illegal
- Algorithm

  (1) Make an empty stack.

  (2) Read characters until end of file

    i. If the character is opening symbol, push onto the stack

    ii. If closing symbol and stack is empty, error

    iii. Otherwise, pop stack. If symbol popped is not corresponding opening symbol, error

  (3) At end of file, if stack is not empty, error

- Other applications
  - Function calls, recursion, …
  - Back/Forward stacks on browsers
  - Undo/Redo stacks in Excel or Word

Queue

- Queue is a list where insert takes place at back, but remove takes place at the front
- Operations
  - Enqueue (insert) element at the back of the queue
  - Dequeue (remove and return) element from the front of the queue
- FIFO (First In First Out)
- Applications: simple job scheduling, graph traversals, priority queues, routing, printer queues, I/O streams

Implementation of Queue

- Just as stacks can be implemented as arrays or linked lists, so with queues.
- Need to have fast access to beginning and end of queues
  - Dequeue at front
  - Enqueue at back
- Similar advantages for arrays vs linked lists
  - But there are different ways to do it with arrays which can result in much data movement, O(n) computation!

Array implementation of Queue

- Naive: front is always position 0, back is variable.
  - Dequeue is O(n)! Must move positions
- Better way: Use circular array
  - When an element moves past the end of a circular array, it wraps around to the beginning, e.g. size 9, top 5, bottom 8
  - Enqueue(4); $000007963 \rightarrow 400007963$ (top 5, bottom 0)
  - Detect an empty or full queue: Use a counter of the number of elements in the queue.
  - O(1)
  - Only issue is fixed size… reallocate if full. Amortized cost O(1).