CS315 Class Notes

Raphael Finkel

May 4, 2021

1 Intro

Class 1, 1/26/2021

- Handout 1 — My names
- TA: Patrick Shepherd
- Plagiarism — read aloud
- Assignments on web. Use C, C++, or Java.
- E-mail list: CS315001@cs.uky.edu
- accounts in MultiLab
- text — we will skip around

2 Basic building blocks: Linked lists (Chapter 3) and trees (Chapter 4)

Linked lists and trees are examples of **data structures**:

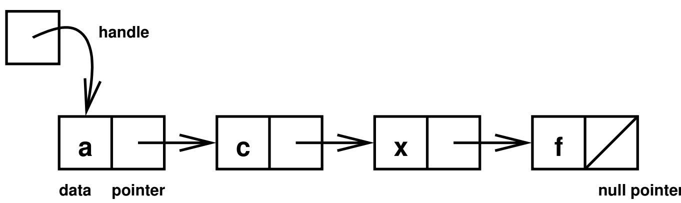
- way to represent information
- so it can be manipulated
- packaged with routines that do the manipulations

Leads to an **Abstract Data Type (ADT)**: has an API (specification) and hides its internals.

3 Tools

4 Singly-linked list

- used as a part of several ADTs.
- Can be considered an ADT itself.
- Collection of **nodes**, each with optional arbitrary **data** and a **pointer** to the next element on the list.



operation	**cost**
create empty list	$\mathcal{O}(1)$
insert new node at front of list	$\mathcal{O}(1)$
delete first node, returning data	$\mathcal{O}(1)$
count length	$\mathcal{O}(n)$
search by data	$\mathcal{O}(n)$
sort	$\mathcal{O}(n \log\ n)$ to $\mathcal{O}(n^2)$

5 Sample code (in C)

```c
#define NULL 0
#include <stdlib.h>

typedef struct node_s {
   int data;
   struct node_s *next;
} node;

node *makeNode(int data, node* next) {
   node *answer = (node *) malloc(sizeof(node));
   answer->data = data;
   answer->next = next;
   return(answer);
} // makeNode

node *insertAtFront(node* handle, int data) {
   node *answer = makeNode(data, handle->next);
   handle->next = answer;
   return(answer);
} // insertAtFront

node *searchDataIterative(node *handle, int data) {
   // iterative method
   node *current = handle->next;
   while (current != NULL) {
      if (current->data == data) break;
      current = current->next;
   }
   return current;
} // searchDataIterative

node *searchDataRecursive(node *handle, int data) {
   // recursive method
   node *current = handle->next;
   if (current == NULL) return NULL;
   else if (current->data == data) return current;
   else return searchDataRecursive(current, data);
} // searchDataRecursive
```

6 Improving the efficiency of some operations

- To make count() fast: maintain the count in a separate variable. If we need the count more often than we insert and delete, it is worthwhile.
- To make insert at rear fast: maintain two handles, one to the front, the other to the rear of the list.
- Combine these new items in a header node:

```
typedef struct {
   node *front;
   node *rear;
   int count;
} nodeHeader;
```

- Class 2, 1/28/2021
- To make search faster: remove the special case that we reach the end of the list by placing a **pseudo-data node** at the end. Keep track of the pseudo-data node in the header.

```
typedef struct {
   node *front;
   node *rear;
   node *pseudo;
   int count;
} nodeHeader;

node *searchDataIterative(nodeHeader *header, int data) {
   // iterative method
   header->pseudo->data = data;
   node *current = header->front;
   while (current->data != data) {
      current = current->next;
   }
   return (current == header->pseudo ? NULL : current);
} // searchDataIterative
```

- Exercise: If we want both pseudo-data and a `rear` pointer, how does an empty list look?
- Exercise: If we want pseudo-data, how does `searchDataRecursive()` change?

- Exercise: Is it easy to add a new node *after* a given node?
- Exercise: Is it easy to add a new node *before* a given node?

7 Aside: Unix pipes

- Unix programs automatically have three "files" open: standard input, which is by default the keyboard, standard output, which is by default the screen, and standard error, which is by default the screen.
- In C and C++, they are defined in `stdio.h` by the names `stdin`, `stdout`, and `stderr`.
- The command interpreter (in Unix, it's called the "shell") lets you invoke programs redirecting any or all of these three. For instance, `ls | wc` redirects `stdout` of the `ls` program to `stdin` of the `wc` program.
- If you run your `trains` program without redirection, you can type in arbitrary numbers.
- If you run `randGen.pl` without redirection, it generates an unbounded list of pseudo-random numbers to `stdout`.
- If you run `randGen.pl | trains`, the list of numbers from `randGen.pl` is redirected as input to `trains`.

8 Stacks, queues, dequeues: built out of either linked lists or arrays

- We'll see each of these.

9 Stack of integer

- Abstract definition: either **empty** or the result of pushing an integer onto the stack.
- operations
  - stack makeEmptyStack()
  - **`boolean`** isEmptyStack(stack S)
  - **`int`** popStack(stack *S) // modifies S

- **`void`** pushStack(stack *S, **`int`** I) // modifies S

10 Implementation 1 of Stack: Linked list

- `makeEmptyStack` implemented by `makeEmptyList()`
- `isEmptyStack` implemented by `isEmptyList()`
- `pushStack` inserts at the front of the list
- `popStack` deletes from the front of the list

11 Implementation 2 of Stack: Array

- Class 3, 2/2/2021

- **Warning:** it's easy to make off-by-one errors.

```c
#define MAXSTACKSIZE 10
#include <stdlib.h>

typedef struct {
   int contents[MAXSTACKSIZE];
   int count; // index of first free space in contents
} stack;

stack *makeEmptyStack() {
   stack *answer = (stack *) malloc(sizeof(stack));
   answer->count = 0;
   return answer;
} // makeEmptyStack

void pushOntoStack(stack *theStack, int data) {
   if (theStack->count == MAXSTACKSIZE) {
      (void) error("stack overflow");
   } else {
      theStack->contents[theStack->count] = data;
      theStack->count += 1;
   }
} // pushOntoStack

int popFromStack(stack *theStack) {
   if (theStack->count == 0) {
      return error("stack underflow");
   } else {
      theStack->count -= 1;
      return theStack->contents[theStack->count];
   }
} // popFromStack
```

- The array implementation limits the size. Does the linked-list implementation also limit the size?
- The array implementation needs one cell per (potential) element, and one for the count. How much space does the linked-list implementation need?
- We can position two opposite-sense stacks in one array so long as

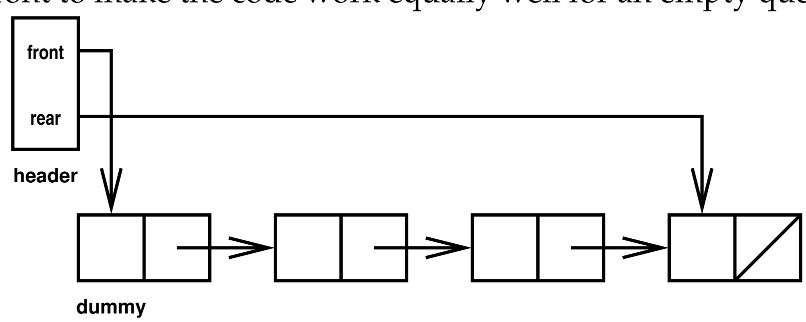
their combined size never exceeds MAXSTACKSIZE.

12 Queue of integer

- Abstract definition: either **empty** or the result of inserting an integer at the rear of a queue or deleting an integer from the front of a queue.
- operations
  - queue makeEmptyQueue()
  - **`boolean`** isEmptyQueue(queue Q)
  - **`void`** insertInQueue(queue Q, **`int`** I) // modifies Q
  - **`int`** deleteFromQueue(queue Q) // modifies Q

13 Implementation 1 of Queue: Linked list

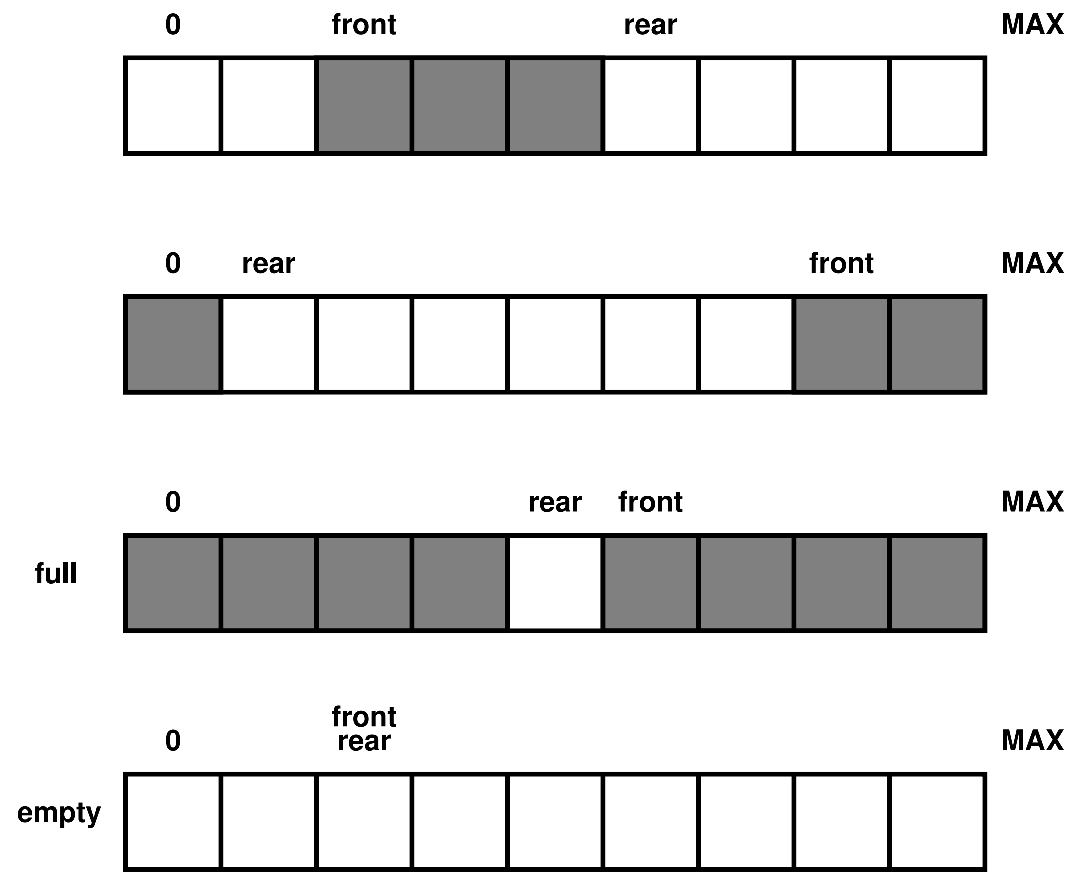
We use a header to represent the front and the rear, and we put a dummy node at the front to make the code work equally well for an empty queue.



```c
#include <stdlib.h>

typedef struct node_s {
   int data;
   struct node_s *next;
} node;

typedef struct {
   node *front;
   node *rear;
} queue;

queue *makeEmptyQueue() {
   queue *answer = (queue *) malloc(sizeof(queue));
   answer->front = answer->rear = makeNode(0, NULL);
   return answer;
} // makeEmptyQueue

bool isEmptyQueue(queue *theQueue) {
   return (theQueue->front == theQueue->rear);
} // isEmptyQueue

void insertInQueue(queue *theQueue, int data) {
   node *newNode = makeNode(data, NULL);
   theQueue->rear->next = newNode;
   theQueue->rear = newNode;
} // insertInQueue

int deleteFromQueue(queue *theQueue) {
   if (isEmptyQueue(theQueue)) return error("queue␣underflow");
   node *oldNode = theQueue->front->next;
   theQueue->front->next = oldNode->next;
   if (theQueue->front->next == NULL) {
      theQueue->rear = theQueue->front;
   }
   return oldNode->data;
} // deleteFromQueue
```

14 Implementation 2 of Queue: Array

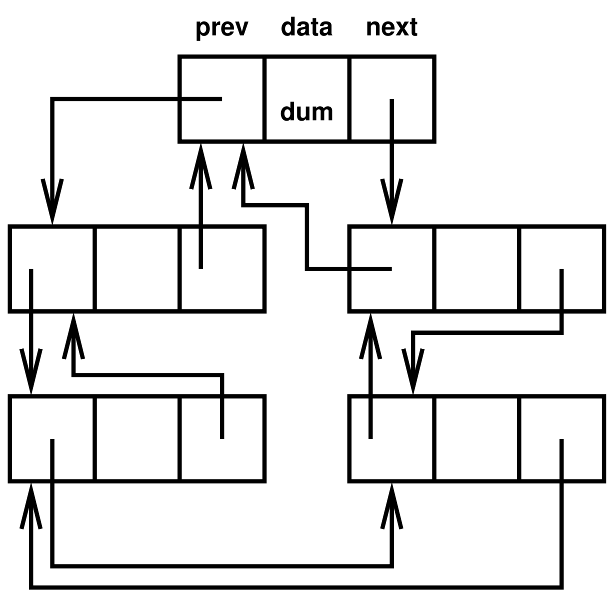
Warning: it's easy to make off-by-one errors.



```c
#define MAXQUEUESIZE 30

typedef struct {
   int contents[MAXQUEUESIZE];
   int front; // index of element at the front
   int rear; // index of first free space after the queue
} queue;

bool isEmptyQueue(queue *theQueue) {
   return (theQueue->front == theQueue->rear);
} // isEmptyQueue

int nextSlot(int index) { // circular
   return (index + 1) % MAXQUEUESIZE;
} // nextSlot

void insertInQueue(queue *theQueue, int data) {
   if (nextSlot(theQueue->rear) == theQueue->front)
      error("queue␣overflow");
   else {
      theQueue->contents[theQueue->rear] = data;
      theQueue->rear = nextSlot(theQueue->rear);
   }
} // insertInQueue

int deleteFromQueue(queue *theQueue) {
   if (isEmptyQueue(theQueue)) {
      return error("queue␣underflow");
   } else {
      int answer = theQueue->contents[theQueue->front];
      theQueue->front = nextSlot(theQueue->front);
      return answer;
   }
} // deleteFromQueue
```

15 Dequeue of integer

- Abstract definition: either **empty** or the result of inserting an integer at the front or rear of a dequeue or deleting an integer from the front or rear of a queue.
- operations
  - dequeue makeEmptyDequeue()
  - `boolean` isEmptyDequeue(dequeue D)
  - `void` insertFrontDequeue(dequeue D, `int` I) // modifies D
  - `void` insertRearDequeue(dequeue D, `int` I) // modifies D
  - `int` deleteFrontDequeue(dequeue D) // modifies D
  - `int` deleteRearDequeue(dequeue D) // modifies D
- Exercise: code the insertFrontDequeue() and deleteRearDequeue() routines using an array.
- All operations for a singly-linked list implementation are $\mathcal{O}(1)$ except for deleteRearDequeue(), which is $\mathcal{O}(n)$.
- The best list structure is a **doubly-linked list** with a single dummy node.



- Exercise: Code all the routines.
- Exercise: Is it easy to add a new node *after* a given node?
- Exercise: Is it easy to add a new node *before* a given node?

16 Searching

- $\boxed{\text{Class 4, 2/4/2021}}$
- Given n data elements (we will use integer data), arrange them in a data structure D so that these operations are fast:
  - **`void`** insert(**`int`** data, *D)
  - **`boolean`** search(**`int`** data, D) (can also return entire data record)
- We don't care about the speed of deletion (for now).
- Much of this material is in Chapter 4 of the book (trees)
- Representation 1: Linked list
  - insert(i) is $\mathcal{O}(1)$: Place new element at the front.
  - search(i) is $\mathcal{O}(n)$: We may need to look at whole list; we use pseudo-data i to make search as fast as possible
- Representation 2: Sorted linked list
  - insert(i) is $\mathcal{O}(n)$: On average, $n/2$ steps. Use pseudo-data (value ∞) at end to make insertion as fast as possible.
  - search(i) is $\mathcal{O}(n)$: We may need to look at whole list; on average, we look at $n/2$ elements if the search succeeds; all n elements if it fails. Use pseudo-data (value ∞) to make search as fast as possible.
- Representation 3: Array
  - insert(i) is $\mathcal{O}(1)$: We place new element at the rear.
  - search(i) is $\mathcal{O}(n)$: We may need to look at whole list; use pseudo-data i at rear.
- Representation 4: Sorted array
  - insert(i) is $\mathcal{O}(n)$: We need to search and then shove cells over.
  - search(i) is $\mathcal{O}(\log n)$: We use **binary search**.

```
// warning: it's easy to make off-by-one errors.
bool binarySearch(int target, int *array,
      int lowIndex, int highIndex) {
   // look for target in array[lowIndex..highIndex]
   while (lowIndex < highIndex) { // at least 2 elements
      int mid = (lowIndex + highIndex) / 2; // round down
      if (array[mid] < target) lowIndex = mid + 1;
      else highIndex = mid;
   } // while at least 2 elements
   return (array[lowIndex] == target);
} // binarySearch
```

17 Quadratic search: set mid based on discrepancy

Also called **interpolation search**, **extrapolation search**, **dictionary search**.

```
bool quadraticSearch(int target, int *array,
      int lowIndex, int highIndex) {
   // look for target in array[lowIndex..highIndex]
   while (lowIndex < highIndex) { // at least 2 elements
      if (array[highIndex] == array[lowIndex]) {
         highIndex = lowIndex;
         break;
      }
      float percent = (0.0 + target - array[lowIndex])
         / (array[highIndex] - array[lowIndex]);
      int mid = int(percent * (highIndex-lowIndex)) + lowIndex;
      if (mid == highIndex) {
         mid -= 1;
      }
      if (array[mid] < target) {
         lowIndex = mid + 1;
      } else {
         highIndex = mid;
      }
   } // while at least 2 elements
   return(array[lowIndex] == target);
} // quadraticSearch
```

Experimental results

- It is hard to program correctly.
- For $10^6 \approx 2^{20}$ elements, binary search always makes 20 probes.
- This result is consistent with $\mathcal{O}(\log n)$.
- Quadratic search: 20 tests with uniform data. The range of probes was 3 – 17; the average about 9 probes.
- Analysis shows that if the data are uniformly distributed, quadratic search should be $\mathcal{O}(\log \log n)$.

18 Analyzing binary search

- Binary search: $c_n = 1 + c_{n/2}$ where c_n is the number of steps to search for an element in an array of length n.
- We will use the **Recursion Theorem**: if $c_n = f(n) + ac_{n/b}$, where $f(n) = \Theta(n^k)$, then

when	c_n
$a < b^k$	$\Theta(n^k)$
$a = b^k$	$\Theta(n^k \ \log \ n)$
$a > b^k$	$\Theta(n^{log_b a})$

- In our case, $a = 1$, $b = 2$, $k = 0$, so $b^k = 1$, so $a = b^k$, so $c_n = \Theta(n^k \ \log \ n) = \Theta(\log \ n)$.
- Bad news: any comparison-based searching algorithm is $\Omega(\log \ n)$, that is, needs at least on the order of $\log \ n$ steps.
- Notation, slightly more formally defined. All these ignore multiplicative constants.
  - $\mathcal{O}(f(n))$: no worse than $f(n)$; at most $f(n)$.
  - $\Omega(f(n))$: no better than $f(n)$; at least $f(n)$.
  - $\Theta(f(n))$: no better or worse than $f(n)$; exactly $f(n)$.

19 Representation 5: Binary tree

- Class 5, 2/9/2021
- Example with elicited values
- Pseudo-data: in the universal "null" node.
- insert(i) and search(i) are both $\mathcal{O}(\log \ n)$ if we are lucky or data are random.

```c
#define NULL 0
#include <stdlib.h>

typedef struct treeNode_s {
   int data;
   treeNode_s *left, *right;
} treeNode;

treeNode *makeNode(int data) {
   treeNode *answer = (treeNode *) malloc(sizeof(treeNode));
   answer->data = data;
   answer->left = answer->right = NULL;
   return answer;
} // makeNode

treeNode *searchTree(treeNode *tree, int key) {
   if (tree == NULL) return(NULL);
   else if (tree->data == key) return(tree);
   else if (key <= tree->data)
      return(searchTree(tree->left, key));
   else
      return(searchTree(tree->right, key));
} // searchTree

void insertTree(treeNode *tree, int key) {
   // assumes empty tree is a pseudo-node with infinite data
   treeNode *parent = NULL;
   treeNode *newNode = makeNode(key);
   while (tree != NULL) { // dive down tree
      parent = tree;
      tree = (key <= tree->data) ? tree->left : tree->right;
   } // dive down tree
   if (key <= parent->data)
      parent->left = newNode;
   else
      parent->right = newNode;
} // insertTree
```

- We will deal with balancing trees later.

20 Traversals

- A **traversal** walks through the tree, visiting every node.
- **Symmetric traversal** (also called **inorder**)

```
void symmetric(treeNode *tree) {
   if (tree == NULL) { // do nothing
   } else {
      symmetric(tree->left);
      visit(tree);
      symmetric(tree->right);
   }
} // symmetric()
```

- **Pre-order traversal**

```
void preorder(treeNode *tree) {
   if (tree == NULL) { // do nothing
   } else {
      visit(tree);
      preorder(tree->left);
      preorder(tree->right);
   }
} // preorder()
```

- **Post-order traversal**

```
void postorder(treeNode *tree) {
   if (tree == NULL) { // do nothing
   } else {
      postorder(tree->left);
      postorder(tree->right);
      visit(tree);
   }
} // postorder()
```

- Does pseudo-data make sense?

21 Representation 6: Hashing (scatter storage)

- Hashing is often the best method for searching (but not for sorting).

- insert(data) and search(data) are $\mathcal{O}(\log\, n)$, but we can generally treat them as $\mathcal{O}(1)$.
- We will discuss hashing later.

22 Finding the jth largest element in a set

- If $j = 1$, a single pass works in $\mathcal{O}(n)$ time:

```
largest = -∞; // priming
foreach (value in set) {
   if (value > largest) largest = value;
}
return(largest);
```

- If $j = 2$, a single pass still works in $\mathcal{O}(n)$ time, but it is about twice as costly:

```
largest = nextLargest = -∞; // priming
foreach (value in set) {
   if (value > largest) {
      nextLargest = largest;
      largest = value;
   } else if (value > nextLargest) {
      nextLargest = value;
   }
} // foreach value
return(nextLargest);
```

- It appears that for arbitrary j we need $\mathcal{O}(jn)$ time, because each iteration needs t tests, where $1 \leq t \leq j$, followed by modifying $j + 1 - t$ values, for a total cost of $j + 1$.
- $\boxed{\text{Class 6, 2/11/2021}}$
- Clever algorithm using an array: **QuickSelect** (Tony Hoare)
  - Partition the array into "small" and "large" elements with a pivot between them (details soon).
  - Recurse in either the small or large subarray, depending where the jth element falls. Stop if the jth element is the pivot.
- Cost: $n + n/2 + n/4 + \ldots \;=\; 2n = \mathcal{O}(n)$

- We can also compute the cost using the recursion theorem (page 17):
  - $c_n = n + c_{n/2}$ (if we are lucky)
  - $c_n = n + c_{2n/3}$ (fairly average case)
  - $f(n) = n = \mathcal{O}(n^1)$
  - $k = 1$, $a = 1$, $b = 2$ (or $b = 3/2$)
  - $a < b^k$
  - so $c_n = \Theta(n^k) = \Theta(n)$

23 Partitioning an array

- Nico Lomuto's method
- Online demonstration.
- The method partitions `array[lowIndex .. highIndex]` into three pieces:
  - `array[lowIndex .. divideIndex -1]`
  - `array[divideIndex]`
  - `array[divideIndex + 1 .. highIndex]`

  The elements of each piece are in order with respect to adjacent pieces.

```
int partition(int array[], int lowIndex, int highIndex) {
   // modifies array, returns pivot index.
   int pivotValue = array[lowIndex];
   int divideIndex = lowIndex;
   for (int combIndex = lowIndex+1; combIndex <= highIndex;
         combIndex += 1) {
      // array[lowIndex] is the partitioning (pivot) value.
      // array[lowIndex+1 .. divideIndex] are < pivot
      // array[divideIndex+1 .. combIndex-1] are ≥ pivot
      // array[combIndex .. highIndex] are unseen
      if (array[combIndex] < pivotValue) { // see a small value
         divideIndex += 1;
         swap(array, divideIndex, combIndex);
      }
   } // each combIndex
   // swap pivotValue into its place
   swap(array, divideIndex, lowIndex);
   return(divideIndex);
} // partition
```

- Example

5 d	2 c	1	7	9	0	3	6	4	8	
5	d,c 2 d	1 c	7	9	0	3	6	4	8	
5	2	d,c 1 d	7 c	9	0	3	6	4	8	
		d d	c	c						
		d d		c	c					
5	2	1	d 0 d	9	c 7	3 c	6	4	8	
5	2	1	0	d 3 d	7	c 9	6 c	4	8	
				d d			c	c		
5	2	1	0	3	d 4 d	9	6	c 7	8 c	
					d d				c	c
4	2	1	0	3	5	9	6	7	8	

24 Using partitioning to select jth smallest

```
int selectJthSmallest (int array[], int size, int targetIndex) {
   // rearrange the values in array[0..size-1] so that
   // array[targetIndex] has the value it would have if the array
   // were sorted.
   int lowIndex = 0;
   int highIndex = size-1;
   while (lowIndex < highIndex) {
      int midIndex = partition(array, lowIndex, highIndex);
      if (midIndex == targetIndex) {
         return array[targetIndex];
      } else if (midIndex < targetIndex) { // look to right
         lowIndex = midIndex + 1;
      } else { // look to left
         highIndex = midIndex - 1;
      }
   } // while lowIndex < highIndex
   return array[targetIndex];
} // selectJthSmallest
```

25 Sorting

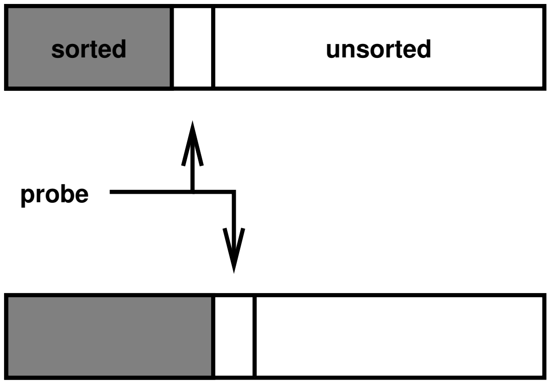
- Class 7, 2/16/2021
- We usually are interested in sorting an array **in place**.
- Sorting is $\Omega(n \log n)$.
- Good methods are $\mathcal{O}(n \log n)$.
- Bad methods are $\mathcal{O}(n^2)$.

26 Sorting out sorting

- `https://www.youtube.com/watch?v=HnQMDkUFzh4` Original film.
- `https://www.youtube.com/watch?v=kPRA0W1kECg` 15 methods in 6 minutes.

27 Insertion sort

- Comb method:



- n iterations.
- Iteration i may need to shift the probe value i places.
- $\Rightarrow\ \mathcal{O}(n^2)$.
- Experimental results for Insertion Sort:
  compares + moves $\approx n$.

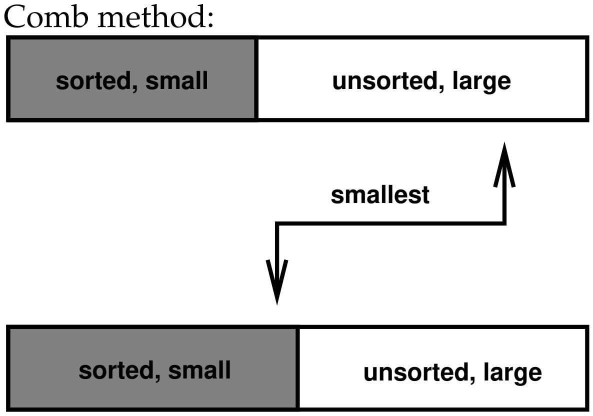
n	compares	moves	$n^2/2$
100	2644	2545	5000
200	9733	9534	20000
400	41157	40758	80000

```
void insertionSort(int array[], int length) {
   // array goes from 1..length.
   // location 0 is available for pseudo-data.
   int combIndex, combValue, sortedIndex;
   for (combIndex = 2; combIndex <= length; combIndex += 1) {
      // array[1 .. combIndex-1] is sorted.
      // Place array[combIndex] in order.
      combValue = array[combIndex];
      sortedIndex = combIndex - 1;
      array[0] = combValue - 1; // pseudo-data
      while (combValue < array[sortedIndex]) {
         array[sortedIndex+1] = array[sortedIndex];
         sortedIndex -= 1;
      }
      array[sortedIndex+1] = combValue;
   } // for combIndex
} // insertionSort
```

- *Stable*: multiple copies of the same key stay in order.

28 Selection sort

- Comb method:



- n iterations.
- Iteration i may need to search through $n - i$ places.
- $\Rightarrow \mathcal{O}(n^2)$.
- Experimental results for Selection Sort:
  compares + moves $\approx n$.

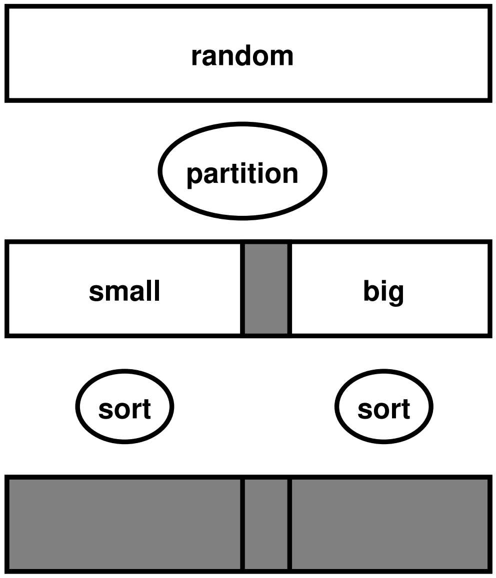
n	compares	moves	$n^2/2$
100	4950	198	5000
200	19900	398	20000
400	79800	798	80000

```
void selectionSort(int array[], int length) {
   // array goes from 0..length-1
   int combIndex, smallestValue, bestIndex, probeIndex;
   for (combIndex = 0; combIndex < length; combIndex += 1) {
      // array[0 .. combIndex-1] has lowest elements, sorted.
      // Find smallest other element to place at combIndex.
      smallestValue = array[combIndex];
      bestIndex = combIndex;
      for (probeIndex = combIndex+1; probeIndex < length;
            probeIndex += 1) {
         if (array[probeIndex] < smallestValue) {
            smallestValue = array[probeIndex];
            bestIndex = probeIndex;
         }
      }
      swap(array, combIndex, bestIndex);
   } // for combIndex
} // selectionSort
```

- Not *stable*, because the swap moves an arbitrary value into the unsorted area.

29 Quicksort (C. A. R. Hoare)

- Class 8, 2/18/2021
- Recursive based on partitioning:



- about $\log\ n$ depth.
- each depth takes about $\mathcal{O}(n)$ work.
- $\Rightarrow\ \mathcal{O}(n\ \log\ n)$.
- Can be unlucky: $\mathcal{O}(n^2)$.
- To prevent worst-case behavior, partition based on median of 3 or 5.
- Don't Quicksort small regions; use a final insertionSort pass instead. Experiments how that the optimal break point depends on the implementation, but somewhere between 10 and 100 is usually good.
- Experimental results for QuickSort:
  compares + moves $\approx 2.4\, n \log n$.

n	compares	moves	$n \log n$	$n^2/2$
100	643	824	664	5000
200	1444	1668	1528	20000
400	3885	4228	3457	80000
800	8066	8966	7715	320000
1600	17583	18958	17030	1280000

- Analysis if lucky: $C_n = n + 2C_{n/2}$, so $k = 1, a = 2, b = 2$, so $a = b^k$, so $C_n = \Theta(n^k \log n) = \Theta(n \log n)$.
- Analysis if unlucky: $C_n = n + C_{n/3} + C_{2n/3} < n + 2C_{2n/3}$, so $k = 1, a = 2, b = 3/2$, so $a > b^k$, so $C_n < \Theta(n^{log_b a}) = \Theta(n^{log_{3/2} 2}) \approx \Theta(n^{1.70951})$, which is still better than quadratic.

```
void quickSort(int array[], int lowIndex, int highIndex){
   if (highIndex - lowIndex <= 0) return;
      // could stop if <= 6 and finish by using insertion sort.
   int midIndex = partition(array, lowIndex, highIndex);
   quickSort(array, lowIndex, midIndex-1);
   quickSort(array, midIndex+1, highIndex);
} // quickSort
```

30 Shell Sort (Donald Shell, 1959)

- Each pass has a **span** s.

```
for (int span in reverse(spanSequence)) {
   for (int offset = 0; offset < span; offset += 1) {
      insertionSort(a[offset], a[offset+span], ... )
   } // each offset
} // each span
```

- The last element in spanSequence must be 1.
- Tokuda's sequence: $s_0 = 1$; $s_k = 2.25s_{k-1} + 1$; $span_k = \lceil s_k \rceil$ = 1, 4, 9, 20, 46, 103, 233, 525, 1182, 2660, ...
- Experimental results for Shell Sort: compares + moves $\approx 2.2n \log n$.

n	compares	moves	$n \log n$	$n^2/2$
100	355	855	664	5000
200	932	1932	1528	20000
400	2266	4666	3457	80000
800	5216	10816	7715	320000
1600	11942	24742	17030	1280000

31 Heaps: a kind of tree

- $\boxed{\text{Class 9, 2/25/2021}}$
- Heap property: the value at a node is $\leq$ the value of each child (for a **top-light heap**) or $\geq$ the value of each child (for a **top-heavy heap**).
- The smallest (largest) value is therefore at the root.
- All leaves are at the same level ± 1.

- To insert
  - Place new value at "end" of tree.
  - Let the new value sift up to its proper level.
- To delete: always delete the least (root) element
  - Save value at root to return it later.
  - Move the last value to the root.
  - Let the new value sift down to its proper level.
- Storage
  - Store the tree in an array $[1 \ldots]$
  - `leftChild[index] = 2*index`
  - `rightChild[index] = 2*index+1`
  - the last occupied place in the array is at index `heapSize`.
- Applications
  - Sorting
  - Priority queue

```
// basic algorithms (top-light heap)

void siftUp (int heap[], int subjectIndex) {
   // the element in subjectIndex needs to be sifted up.
   heap[0] = heap[subjectIndex]; // pseudo-data
   while (1) { // compare with parentValue.
      int parentIndex = subjectIndex / 2;
      if (heap[parentIndex] <= heap[subjectIndex]) return;
      swap(heap, subjectIndex, parentIndex);
      subjectIndex = parentIndex;
   }
} // siftUp

int betterChild (int heap[], int subjectIndex, int heapSize) {
   int answerIndex = subjectIndex * 2; // assume better child
   if (answerIndex+1 <= heapSize &&
         heap[answerIndex+1] < heap[answerIndex]) {
      answerIndex += 1;
   }
   return(answerIndex);
} // betterChild

void siftDown (int heap[], int subjectIndex, int heapSize) {
   // the element in subjectIndex needs to be sifted down.
   while (2*subjectIndex <= heapSize) {
      int childIndex = betterChild(heap, subjectIndex, heapSize);
      if (heap[childIndex] >= heap[subjectIndex]) return;
      swap(heap, subjectIndex, childIndex);
      subjectIndex = childIndex;
   }
} // siftUp
```

```c
// intermediate algorithms

void insertInHeap (int heap[], int *heapSize, int value) {
   *heapSize += 1; // should check for overflow
   heap[*heapSize] = value;
   siftUp(heap, *heapSize);
} // insertInHeap

int deleteFromHeap (int heap[], int *heapSize) {
   int answer = heap[1];
   heap[1] = heap[*heapSize];
   *heapSize -= 1;
   siftDown(heap, 1, *heapSize);
   return(answer);
} // deleteFromHeap
```

```c
// advanced algorithm

void heapSort(int array[], int arraySize){
   // sorts array[1..arraySize] by first making it a
   // top-heavy heap, then by successive deletion.
   // Deleted elements go to the end.
   int index, size;
   array[0] = −∞; // pseudo-data
   // The second half of array[] satisfies the heap property.
   for (index = (arraySize+1)/2; index > 0; index -= 1) {
      siftDown(array, index, arraySize);
   }
   for (index = arraySize; index > 0; index -= 1) {
      array[index] = deleteFromHeap(array, &arraySize);
   }
} // heapSort
```

- This method of heapifying is $\mathcal{O}(n)$:
  - 1/2 the elements require no motion.
  - 1/4 the elements may sift down 1 level.
  - 1/8 the elements may sift down 2 levels.
  - Total motion = $(n/2) \cdot \sum_{1 \le j} j/2^j$

   - That formula approaches n as $j \to \infty$

- Total complexity is therefore $\mathcal{O}(n + n \log n) = \mathcal{O}(n \log n)$.
- This sorting method is not *stable*, because sifting does not preserve order.

Experimental results for Heap Sort: compares + moves $\approx 3.1 n \log n$.

n	compares	moves	$n \log n$	$n^2/2$
100	755	1190	664	5000
200	1799	2756	1528	20000
400	4180	6196	3457	80000
800	9621	14050	7715	320000
1600	21569	31214	17030	1280000

32 Bin sort

- Assumptions: values lie in a small range; there are no duplicates.
- Storage: build an array of bins, one for each possible value. Each is 1 bit long.
- Space: $\mathcal{O}(r)$, where r is the size of the range.
- Place each value to sort as a 1 in its bin. Time: $\mathcal{O}(n)$.
- Read off bins in order, reporting index if it is 1. Time: $\mathcal{O}(r)$.
- Total time: $\mathcal{O}(n + r)$.
- Total memory: $\mathcal{O}(r)$, which can be expensive.
- Can handle duplicates by storing a count in each bin, at a further expense of memory.
- This sorting method does not work for arbitrary data having numeric keys; it only sorts the keys, not the data.

33 Radix sort

- Example: use base 10, with values integers 0 – 9999, with 10 bins, each holding a list of values, initially empty.
- Pass 1: insert each value in a bin (at rear of its list) based on the last digit of the value.

- Pass 2: examine values in bin order, and in list order within bins, placing them in a new copy of bins based on the second-to-last digit.
- Pass 3, 4: similar.
- The number of digits is $\mathcal{O}(\log\ n)$, so there are $\mathcal{O}(\log n)$ passes, each of which takes $\mathcal{O}(n)$ time, so the algorithm is $\mathcal{O}(n\ \log\ n)$.
- This sorting method is *stable*.

34 Merge sort

```
void mergeSort(int array[], int lowIndex, int highIndex){
   // sort array[lowIndex] .. array[highIndex]
   if (highIndex - lowIndex < 1) return; // width 0 or 1
   int mid = (lowIndex+highIndex)/2;
   mergeSort(array, lowIndex, mid);
   mergeSort(array, mid+1, highIndex);
   merge(array, lowIndex, highIndex);
} // mergeSort

void merge(int array[], int lowIndex, int highIndex) {
   int mid = (lowIndex+highIndex)/2;
   // copy the relevant parts of array to two temporaries
   // walk through the temporaries in tandem,
   // placing smaller in array, ties honor left version.
} // merge
```

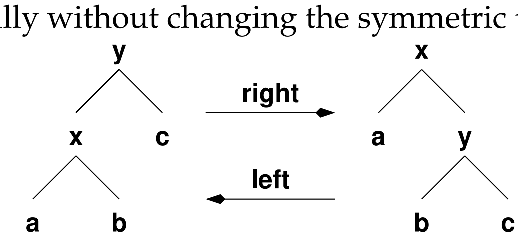
- $c_n = n + 2c_{n/2}$
- $a = 2, b = 2, k = 1 \Rightarrow \mathcal{O}(n \log n)$.
- This time complexity is **guaranteed**.
- Space needed: $2n$, because merge in place is awkward (and expensive).
- The sort is also *stable*: it preserves the order of identical keys.
- Insertion, radix, and merge sort are stable, but not selection, Quicksort or Heapsort.

Experimental results for Merge Sort: compares + moves $\approx 2.9n \log n$.

n	compares	moves	$n \log n$	$n^2/2$
100	546	1344	664	5000
200	1286	3088	1528	20000
400	2959	6976	3457	80000
800	6741	15552	7715	320000
1600	15017	34304	17030	1280000

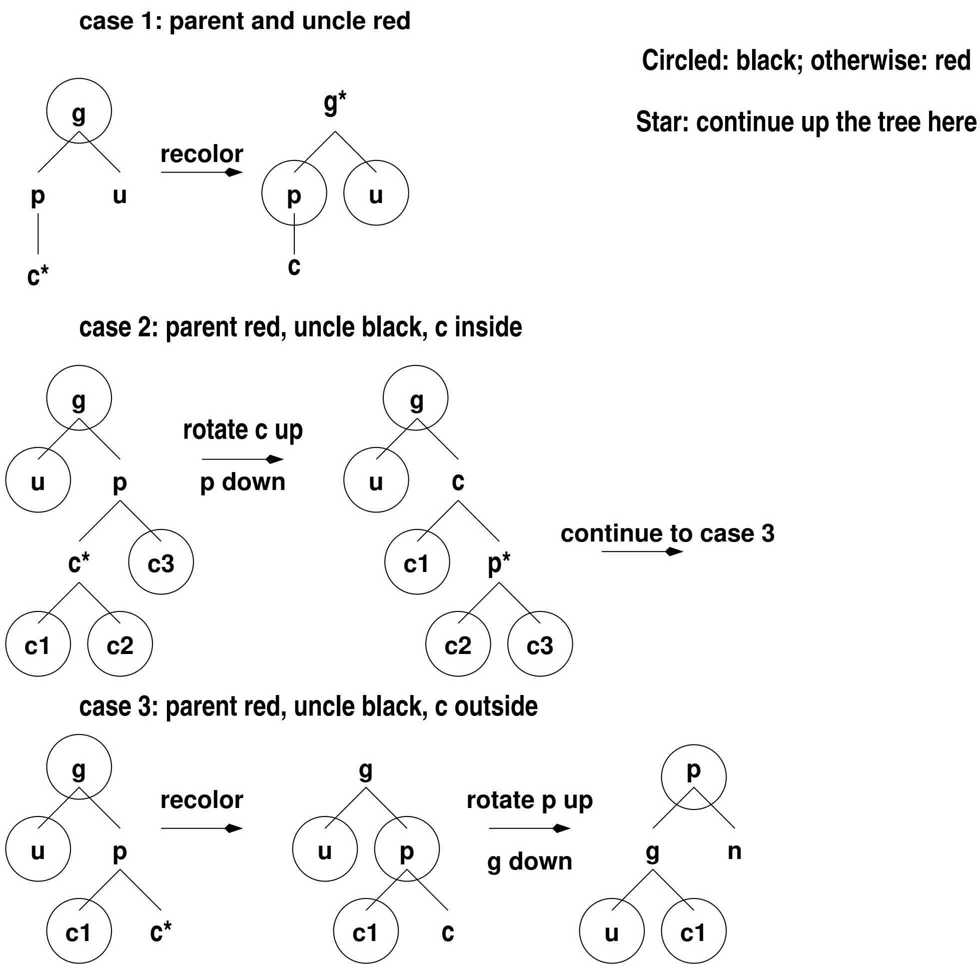
35 Red-black trees (Guibas and Sedgewick 1978)

- Class 10, 3/2/2021
- Red-black trees balance themselves during online insertion.
- Their representation requires pointers both to children and to the parent.
- Each node is **red** or **black**.
- The pseudo-nodes (or null nodes) at bottom are black.
- The root node is black.
- Red nodes have only black children. So no path has two red nodes in a row.
- All paths from the root to a leaf have the same number of black nodes.
- For a node x, define black-height(x) = number of black nodes on a path down from x, not counting x.
- The algorithm manages to keep height of the tree $\leq 2\log(n+1)$.
- To keep the tree acceptable, we sometimes **rotate**, which reorganizes the tree locally without changing the symmetric traversal.

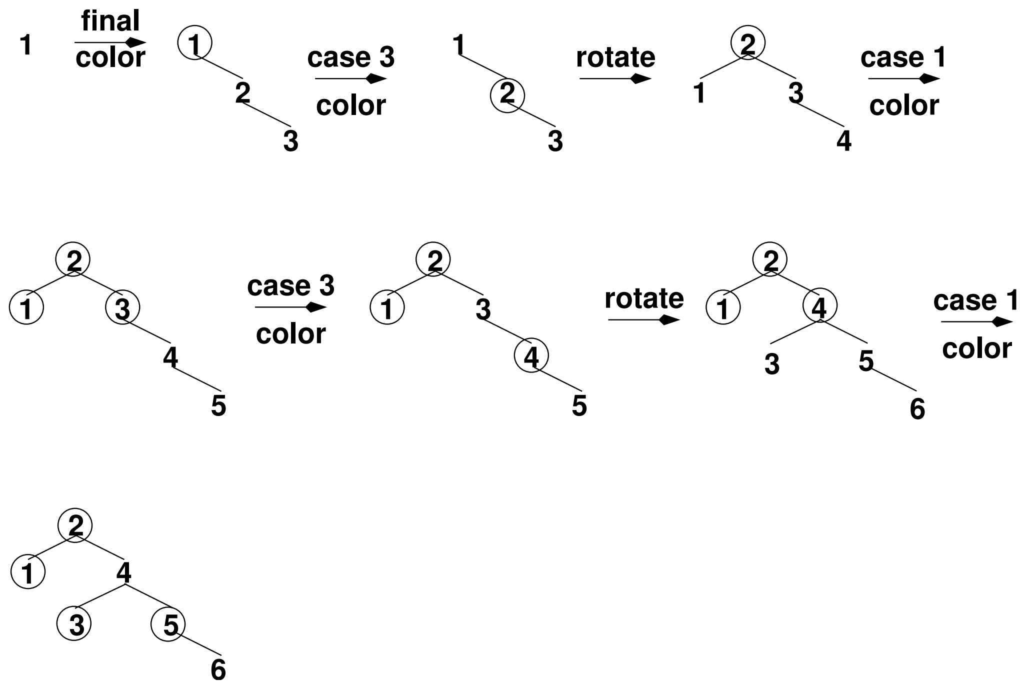


- To insert
  - place new node $\boxed{\text{n}}$ in the tree and color it red. $\mathcal{O}(\log\ n)$.

- walk up the tree from $\boxed{\text{n}}$, rotating as needed to restore color rules. $\mathcal{O}(\log\, n)$.
- color the root black.



- try with values 1..6:



- try with these values: 5, 2, 7, 4 (case 1), 3 (case 2), 1 (case 1)

36 Review of binary trees

- Binary trees have expected $\mathcal{O}(\log\ n)$ depth, but they can have $\mathcal{O}(n)$ depth.
- insertion
- traversal: preorder, postorder, inorder=symmetric order.
- deletion of node D
  - If D is a leaf, remove it.
  - If D has one child C, move C in place of D.
  - If D has two children, find its successor: S = RL^*. Move S in place of D. S has no left child, but if it has a right child C, move C in place of S.

37 Ternary trees

- $\boxed{\text{Class 11, 3/4/2021}}$

- By example.
- The depth of a balanced ternary tree is $\log_3 n$, which is only 63% the depth of a balanced binary tree.
- The number of comparisons needed to traverse an internal node during a search is either 1 or 2; average 5/3.
- So the number of comparisons to reach a leaf is $\frac{5}{3}\log_3 n$ instead of (for a binary tree) $\log_2 n$, a ratio of 1.05, indicating a 5% degradation.
- The situation gets only worse for larger arity. For quaternary trees, the degradation (in comparison to binary trees) is about 12.5%.
- And, of course, an online construction is not balanced.
- Moral: binary is best; higher arity is not helpful.

38 Quad trees (Finkel 1973)

- Extension of sorted binary trees to two dimensions.
- Internal nodes contain a **discriminant**, which is a two-dimensional (x,y) value.
- Internal nodes have four children, corresponding to the four quadrants from the discriminant.
- Leaf nodes contain a **bucket** of b values.
- Insertion
  - Dive down the tree, put new value in its bucket.
  - If the bucket overflows, pick a good discriminant and subdivide.
  - Good discriminant: one that separates the values as evenly as possible. Suggestion: median (x, y) values.
- Offline algorithm to build a balanced tree
  - Put all elements in a single bucket, then recursively subdivide as above.
- Generalization: for d-dimensional data, let each discriminant have d values. A node can have up to 2^d children. This number becomes cumbersome when d grows above about 3.

- Heavily used in 3-d modeling for graphics, often with discriminant chosen as midpoint, not median.

39 k-d trees (Bentley and Finkel 1973)

- Extension of sorted binary trees to d dimensions.
- Especially good when d is high.
- Internal nodes contain a **dimension** number ($0 .. d-1$) and a **discriminant** value (real).
- Internal nodes have two children, corresponding to values $\leq$ and $>$ the discriminant in the given dimension.
- Leaf nodes contain a **bucket** of b values.
- Offline construction and online insertion are similar to quad trees.
  - To split a bucket of values, pick the dimension number with the largest range across those values.
  - Given the dimension, pick the median of the values in that dimension as the discriminant.
  - That choice of dimension number tends to make the domain of each bucket roughly cubical; that choice of discriminant balances the tree.
- Nearest-neighbor search: Given a d-dimensional probe value p, to find the nearest neighbor to p that is in the tree.
  - Dive into the tree until you find p's bucket.
  - Find the closest value in the bucket to p. Cost: b distance measures. Result: a **ball** around p.
  - Walking back up to the root, starting at the bucket:
    - If the domain of the other child of the node overlaps the ball, dive into that child.
    - If the ball is entirely contained within the node's domain, done.
    - Otherwise walk one step up toward the root and continue.
  - complexity: Initial dive is $\mathcal{O}(n)$, but the expected number of buckets examined is $\mathcal{O}(1)$.

- Used for cluster analysis, categorizing (as in optical character recognition).

40 2-3 trees (John Hopcroft, 1970)

- Class 12, 3/9/2021
- By example.
- Like a ternary tree, but different rule of insertion
- Always completely balanced
- A node may hold 1, 2, or 3 (temporarily) values.
- A node may have 0 (only leaves), 2, 3, or 4 (temporarily) children.
- A node that has 3 values splits and promotes its middle value to its parent (recursively up the tree).
- If the root splits, it promotes a new root.
- Complexity: $\mathcal{O}(n \log n)$ for insertion and search, guaranteed.
- Deletion: unpleasant.

41 Stooge Sort

- A terrible method, but fun to analyze.

```c
#include <math.h>

void stoogeSort(int array[], int lowIndex, int highIndex){
   // highIndex is one past the end
   int size = highIndex - lowIndex;
   if (size <= 1) { // nothing to do
   } else if (size == 2) { // direct sort
      if (array[lowIndex] > array[lowIndex+1]) {
         swap(array, lowIndex, lowIndex+1);
      }
   } else { // general case
      float third = ((float) size) / 3.0;
      stoogeSort(array, lowIndex, ceil(highIndex - third));
      stoogeSort(array, floor(lowIndex + third), highIndex);
      stoogeSort(array, lowIndex, ceil(highIndex - third));
   }
} // stoogeSort
```

- $c_n = 1 + 3c_{2n/3}$
- $a = 3$, $b = 3/2$, $k = 0$, so $b^k = 1$. By the recursion theorem (page 18), since $a > b^k$, we have complexity $\Theta(n^{log_b a})$ = $\Theta(n^{log_{3/2} 3}) \approx \Theta(n^{2.71})$, so Stooge Sort is worse than quadratic.
- However, the recursion often encounters already-sorted sub-arrays. If we add a check for that situation, Stooge Sort becomes roughly quadratic.

42 B trees (Ed McCreight 1972)

- A generalization of 2-3 trees when McCreight was at Boeing, hence the name.
- Choose a number m (the **bucket size**) such that m values plus m disk indices fit in a single disk block. For instance, if a block is 4KB, a value takes 4B, and an index takes 4B, then $m = 4\text{KB}/8\text{B} = 512$.
- $m = 3 \Rightarrow$ 2-3 tree.
- Class 13, 3/11/2021
- Each node has $1 .. m-1$ values and $0 .. m$ children. (We have room for m values; the extra can be used for pseudo-data.)

- Shorthand: $g = \lceil m/2 \rceil$ (the **half size**)
- Internal nodes (other than the root) have g .. m children.
- Insertion
    - Insert in appropriate leaf.
    - If current node overflows (has m values) split it into two nodes of g values each; hoist the middle value up one level.
    - When a node splits, its parent's pointer to it becomes two pointers to the new nodes.
    - When a value is hoisted, iterate up the tree checking for overflow.
- B+ tree variant: link leaf nodes together for quicker inorder traversal. This link also allows us to avoid splitting a leaf if its neighbor is not at capacity.
- A densely filled tree with n keys (values), height h:
    - Number of nodes $a = 1 + m + m^2 + \cdots + m^h = \frac{m^{h+1}-1}{m-1}$.
    - Number of keys $n = (m-1)a = m^{h+1} - 1 \Rightarrow \log_m(n+1) = h + 1 \Rightarrow h$ is $\mathcal{O}(\log\ n)$.
- A sparsely filled tree with n keys (values), height h:
    - The root has two subtrees; the others have $g = \lceil m/2 \rceil$ subtrees, so:
    - Number of nodes $a = 1 + 2(1 + g + g^2 + \cdots + g^{h-1}) = 1 + \frac{2(g^h-1)}{g-1}$.
    - The root has 1 key, the others have $g - 1$ keys, so:
    - Number of keys $n = 1+2(g^h-1) = 2g^h-1 \Rightarrow h = \log_g(n+1)/2 = \mathcal{O}(\log\ n)$.

43 Deletion from a B tree

- Deletion from an internal node: replace value with successor (taken from a leaf), and then proceed to deletion from a leaf.
- Deletion from a leaf: the bad case is that it can cause **underflow**: the leaf now has fewer than g keys.
- In case of underflow, borrow a value from a neighbor if possible, adjusting the appropriate key in the parent.

- If all neighbors (there are 1 or 2) are already minimal, grab a key from the parent and also **merge** with a neighbor.
- In general, deletion is quite difficult.

44 Hashing

- Very popular data structure for searching.
- Cost of insertion and of search is $\mathcal{O}(\log\ n)$, but only because n distinct values must be $\log\ n$ bits long, and we need to look at the entire key. If we consider looking at a key to be $\mathcal{O}(1)$, then hashing is expected (but not guaranteed) to be $\mathcal{O}(1)$.
- Idea: find the value associated with key k at `A[`$h(k)$`]`, where
  - $h()$ maps keys to integers in $0..s-1$, where s is the size of A[ ].
  - $h()$ is "fast". (It generally needs to look at all of k, though.)
- Example
  - k = student in class.
  - $h(k)$ = k's birthday (a value from 0 .. 365).
- Difficulty: collisions
  - Birthday paradox: Prob(no collisions with j people) = $\frac{365!}{(365-j)!365^j}$
  - This probability goes below 1/2 at $j = 23$.
  - At $j = 50$, the probability is 0.029.
- Moral: One cannot in general avoid collisions. One has to deal with them.

45 Hashing: Dealing with collisions: open addressing

- Overview
  - The following methods store all items in A[ ] and use a probe sequence. If the desired position is occupied, use some other position to consider instead.

- These methods suffer from clustering.
- Deletion is hard, because removing an element can damage unrelated searches. Deletion by **marking** is the only reasonable approach.

- Perfect hashing: if you know all n values in advance, you can look for a non-colliding hash function `h`. Finding such a function is in general quite difficult, but compiler writers do sometimes use perfect hashing to detect keywords in the language (like **`if`** and **`for`**).
- Linear probing. Probe p is at index $h(k) + p$ (mod s), for $p = 0,\ 1,\ \ldots$.

  - Terrible behavior when A[ ] is almost full, because chains coalesce. This problem is called "primary clustering".

- Additional hash functions. Use a family of hash functions, $h_1(),\ h_2(), \ldots$.

  - insertion: key probing with different functions until an empty slot is found.
  - searching: probe with different functions until you find the key (success) or an empty slot (failure).
  - You need a **family** of independent hash functions.
  - The method is very expensive when A[ ] is almost full.

46 Review for midterm

Class 14, 3/16/2021

Insert the following items: 3_1 1_1 4 1_2 5_1 9 2 6 5_2 3_2 into:

- binary tree. Preorder result: 3_1 1_1 1 2 2 3_2 5_1 5_2 9 6
- top-light heap. Breadth-order result: 1_1 1_2 2 3_1 3_2 9 4 6 5_2 5_1
- array, then heapify. Breadth-order result: 1_1 1_2 2 3_1 3_2 9 4 6 5_2 5_1
- ternary tree. Preorder result: $(1_1, 3)$ 1_2 $(2, 3_2)$ $(4, 5_1)$ 5_2 $(6, 9)$
- array, then 5 steps of selection sort. Result: 1_1 1_2 2 3_1 3_2 | 9 4 6 5_2 5_1
  Note: not stable.
- array, then 5 steps of insertion sort. Result: 1_1 1_2 3_1 4 5_1 | 9 2 6 5_2 3_2
  Note: stable. Can force anti-stable.
- array, then first step of Quicksort, using Lomuto's partitioning. final insertionSort.

- 2-3 tree. Preorder result: $3\ 1\ 1\ (2,3)\ 5\ (4,5)\ (6,9)$
- red-black tree. Preorder result: $3_b\ 1\ 1_b\ 2_b\ 3\ 5\ 4_b\ 5\ 9_b\ 6$

47 Midterm exam

Class 15, 3/18/2021

48 Midterm exam follow-up

Class 16, 3/23/2021

49 Hashing: more open-addressing methods

- Class 17, 3/25/2021
- Quadratic probing. Probe p is at index $h(k) + p^2$ (mod s), for $p = 0,\ 1,\ \ldots$.
  - When does this sequence hit all of A[ ]? Certainly it does if s is prime.
  - We still suffer "secondary clustering": if two keys have the same hash value, then the sequence of probes is the same for both.
- Add-the-hash rehash. Probe p is at index $(p+1) \cdot h(k)$ (mod s).
  - This method avoids clustering.
  - Warning: $h(k)$ must never be 0.
- Double hashing. Use two has functions, $h_1()$ and $h_2()$. Probe p is at index $h_1(k) + p \cdot h_2(k)$.
  - This method avoids clustering.
  - Warning: $h_2(k)$ must never be 0.

50 Hashing: Dealing with collisions: external chaining

- Each element in A is a pointer, initially null, to a **bucket**, which is a linked list of nodes that hash to that element; each node contains k and any other associated data.
- insert: place k at the front of A[$h(k)$].
- search: look through the list at A[$h(k)$].
  - optimization: When you find, promote the node to the start of its list.
- average list length is s/n. So if we set $s \cong n$ we expect about 1 element per list, although some may be longer, some empty.
- Instead of lists, we can use something fancier (such as 2-3 trees), but it is generally better to use a larger s.

51 Hashing: What is a good hash function?

- Want it to be
  - Uniform: Equally likely to give any value in $0..s-1$.
  - Fast.
  - Spreading: similar inputs $\rightarrow$ dissimilar outputs, to prevent clustering. (Only important for open addressing, as described below.)
- Several suggestions, assuming that k is a multi-word data structure, such as a string.
  - Add (or multiply) all (or some of) the words of k, discarding overflow, then mod by s. It helps if $s = 2^j$, because mod is then masking with $2^j - 1$.
  - `XOR` the words of k, shifting left by 1 after each, followed by mod s.
- Wisdom: The hash function doesn't make much difference. It is not necessary to look at all of k. Just make sure that $h(k)$ is not constant (except for testing collision resolution).

52 Hashing: How big should the array be?

- Some open-addressing methods prefer that $s = ||Array||$ be prime.
- Computing $h()$ is faster if $s = 2^j$ for some j.
- Open addressing gets very bad if $s < 2n$, depending on method. Linear probing is the worst; I would make sure $s \geq 3n$.
- External chaining works fine even when $s \cong n$, but it gets steadily worse.

53 Hashing: What should we do if we discover that s is too small?

- We can rebuild with a bigger s, rehashing every element. But that operation causes a temporary "outage", so it is not acceptable for online work.
- Extendible hashing
  - Start with one bucket. If it gets too full (list longer than 10, say), split it on the *last bit* of $h(k)$ into two buckets.
  - Whenever a bucket based on the last j bits is too full, split it based on bit $j + 1$ from the end.
  - To find the bucket
    - compute $v = h(k)$.
    - follow a tree that discriminates on the last bits of v. This tree is called a **trie**.
    - it takes at most $\log\ v$ steps to find the right bucket.
    - Searching within the bucket now is guaranteed to take constant time (ignoring the $\log\ n$ cost of comparing keys)

54 Hash tables (associative arrays) in scripting languages

- $\boxed{\text{Class 18, 3/30/2021}}$
- Like an array, but the indices are strings.

- Resizing the array is automatic, although one might specify the expected size in advance to avoid resizing during early growth.
- Perl has a built-in datatype called a **hash**.

```
my %foo;
foo{"this"} = "that".
```

- Python has **dictionaries**.

```
Foo = dict()
Foo['this'] = 'that';
```

- JavaScript arrays are all associative.

```
const foo = [];
foo['this'] = 'that';
foo.this = 'that';
```

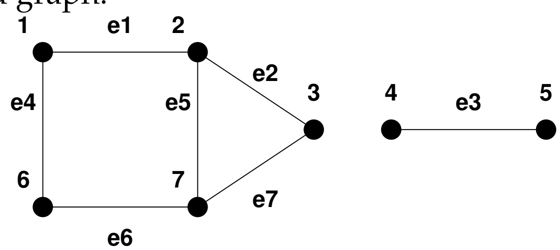
55 Cryptographic hashes: digests

- purpose: uniquely identify text of any length.
- these hashes are *not* used for searching.
- goals
  - fast computation
  - uninvertable: given $h(k)$, it should be infeasible to compute k.
  - it should be infeasible to find collisions k_1 and k_2 such that $h(k_1) = h(k_2)$.
- examples
  - MD5: 128 bits. Practical attack in 2008.
  - SHA-1: 160 bits, but (2005) one can find collisions in 2^{69} hash operations (brute force would use 2^{80})
  - SHA-2: usual variant is SHA256; also SHA-512.
- uses

- storing passwords (used as a trap-door function)
- catching plagiarism
- for authentication ($h(m + s)$ authenticates m to someone who shares the secret s, for example)
- tripwire: intrusion detection

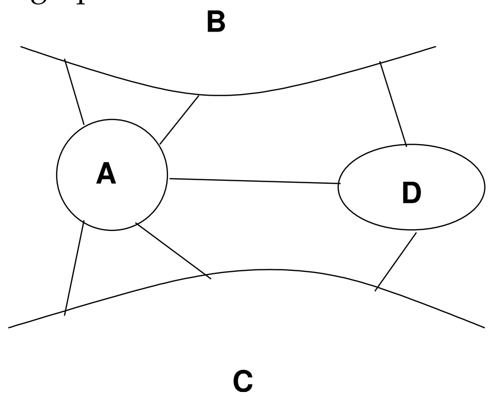
56 Graphs

- Our standard graph:



- Nomenclature
    - **vertices**: V is the name of the set, v is the size of the set. In our example, V = {1, 2, 3, 4, 5, 6, 7}.
    - **edges**: E is the name of the set, e is the size of the set. In our example, E = {e1, e2, e3, e4, e5, e6, e7}.
    - **directed graph**: edges have direction (represented by arrows).
    - **undirected graph**: edges have no direction.
    - **multigraph**: more than one edge between two vertices. We generally do not deal with multigraphs, and the word **graph** generally disallows them.
    - **weighted graph**: each edge has numeric label called its **weight**.
- Graphs represent situations
    - streets in a city. We might be interested in computing **paths**.
    - airline routes, where the weight is the price of a flight. We might be interested in minimal-cost cycles.
        - Hamiltonian cycle: no duplicated vertices (cities).
        - Eulerian cycle: no duplicated edges (flights).
    - Islands and bridges, as in the bridges of Königsburg, later called Kaliningrad (Euler 1707-1783). This is a multigraph, not strictly

a graph.



Can you find an Eulerian cycle?

- Family trees. These graphs are **bipartite**: Family nodes and person nodes. We might want to find the shortest path between two people.
- Cities and roadways, with weights indicating distance. We might want a minimal-cost spanning tree.

57 Data structures representing a graph

- Adjacency matrix
  - an array $n \times n$ of Boolean.
  - A[i, j] = true $\Rightarrow$ there is an edge from vertex i to vertex j.

	1	2	3	4	5	6	7
1		x				x	
2	x		x				x
3		x					x
4					x		
5				x			
6	x						x
7		x	x			x	

  - The array is symmetric if the graph is undirected
    - in this case, we can store only one half of it, typically in a 1-dimensional array
    - A[$i(i-1)/2 + j$] holds information about edge i, j.
  - Instead of Boolean, we can use integer values to store edge weights.

- Adjacency list
  - an array n of singly-linked lists.
  - j is in linked list A[i] if there is an edge from vertex i to vertex j.

1	$2 \to 6$
2	$1 \to 3 \to 7$
3	$2 \to 7$
4	5
5	4
6	$1 \to 7$
7	$2 \to 3 \to 6$

58 Computing the degree of all vertices

- Adjacency matrix: $\mathcal{O}(v^2)$.

```
foreach vertex (0 .. v-1) {
    degree[vertex] = 0;
    foreach neighbor in 0 .. v-1 {
        if (A[vertex, neighbor]) degree[vertex] += 1;
    }
}
```

- Adjacency list: $\mathcal{O}(v + e)$.

```
foreach vertex (0 .. v-1) {
    degree[vertex] = 0;
    for (neighbor = A[vertex]; neighbor != null;
            neighbor = neighbor->next) {
        degree[vertex] += 1;
    }
}
```

59 Computing the connected component containing vertex i in an undirected graph

- why: to segment an image.
- Class 19, 4/1/2021

- method: **depth-first search** (**DFS**).

```
void DFS(vertex here) {
   // assume visited[*] == false at start
   visited[here] = true;
   foreach next (successors(here)) {
      if (! visited[next]) DFS(next);
   }
} // DFS
```

- DFS is faster with adjacency list: $\mathcal{O}(e' + v')$, where e', v' only count to the number of edges and vertices in the connected component.
- DFS is slower with adjacency matrix: $\mathcal{O}(v + v')$.
- For our standard graph (page 49), assuming that the adjacency lists are all sorted by vertex number (or that we use the adjacency matrix), starting at vertex 1, we invoke DFS on these vertices: 1, 2, 3, 7, 6.
- DFS can be coded iteratively with an explicit stack

```
void DFS(vertex start) {
   // assume visited[*] == false at start
   workStack = makeEmptyStack();
   pushStack(workStack, start)
   while (! isEmptyStack(workStack)) {
      place = popStack(workStack);
      if (visited[place]) continue;
      visited[place] = true;
      foreach neighbor (successors(place)) {
         if (! visited[neighbor]) {
            pushStack(workStack, neighbor);
            // could record "place" as parent
         } // "neighbor" is not yet visited
      } // foreach neighbor
   } // while workStack not empty
```

60 To see if a graph is connected

- See if DFS hits every vertex.

```
bool isConnected() {
   foreach vertex (vertices)
      visited[vertex] = false;
   DFS(0); // or any vertex
   foreach vertex (vertices)
      if (! visited[vertex]) return false;
   return true;
} // isConnected
```

61 Breadth-first search

- applications
  - find shortest path in a family tree connecting two people
  - find shortest route through city streets
  - find fastest itinerary by plane between two cities
- method: place unfinished vertices in a queue. These are the ones we still need to visit, in order closest to furthest.

```
void BFS(vertex start) {
   // assume visited[*] == false at start
   workQueue = makeQueue();
   visited[start] = true;
   insertInQueue(workQueue, start)
   while (! emptyQueue(workQueue)) {
      place = deleteFromQueue(workQueue); // from front
      foreach neighbor (successors(place)) {
         if (! visited[neighbor]) {
            visited[neighbor] = true;
            insertInQueue(workQueue, neighbor); // to rear
            // or: insert (place, neighbor)
            // to remember path to start
         } // not visited
      } // foreach neighbor
   } // while queue not empty
} // BFS
```

- For our standard graph (page 49), assuming that the adjacency lists are all sorted by vertex number (or that we use the adjacency matrix), starting at vertex 1, BFS visits these vertices: 1, 2, 6, 3, 7.
- using adjacency lists, BFS is $\mathcal{O}(v' + e')$.

62 Shortest path between vertices i and j

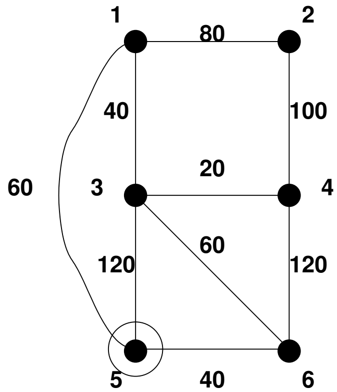
- Compute BFS(i), but stop when you visit j.
  - Actually, you can stop when you place j in the queue.
  - Construct the path by building a back chain when you insert a vertex in the queue. That is, you insert a pair: `(place, neighbor)`.
- If edges are weighted:
  - Use a heap (top-light) instead of a queue. That's why heaps are sometimes called **priority queues**.
  - stop when you visit j, not when you place j in the queue.
- Class 20, 4/6/2021

```
void weightedBFS(vertex start, vertex goal) {
   // assume visited[*] == () at start
   workHeap = makeHeap(); // top-light
   insertInHeap(workHeap, (0, start, start));
      // distance, vertex, from where
   while (! emptyHeap(workHeap)) {
      (distance, place, from) = deleteFromHeap(workHeap);
      if (visited[place] != ()) continue; // already seen
      visited(place) = (from, distance);
      if (place == goal) return; // could print path
      foreach (neighbor, weight) in (successors(place)) {
         insertInHeap(workHeap, (distance+weight, neighbor, place));
      } // foreach neighbor
   } // while queue not empty
} // BFS
```

63 Dijkstra's algorithm: Finding all shortest paths from a given vertex in a weighted graph

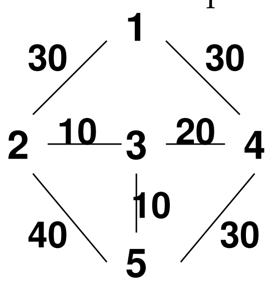
The weights must be positive. Weiss §9.3.2

- Rule: among all vertices that can extend a shortest path already found, choose the one that results in a shortest path. If there is a tie ending at the same vertex, choose either. If there is a tie going to different vertices, choose both.
- This is an example of a **greedy algorithm**: at each step, improve the solution in the way that looks best at the moment.
- Starting position: one path, length 0, from start vertex j to j.



path	length	
5	$\rightarrow$ 0	
5 6	$\rightarrow$ 40	
5 3	120	
5 1	60	
5 1	$\rightarrow$ 60	
5 6 3	100	better way to add vertex 3
5 6 4	160	
5 1 2	140	
5 6 3	$\rightarrow$ 100	
5 6 4	160	
5 1 2	140	
5 6 3 4	$\rightarrow$ 120	better way to add vertex 4

- Another example:

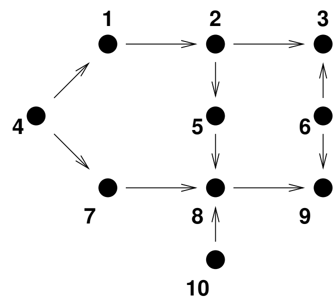


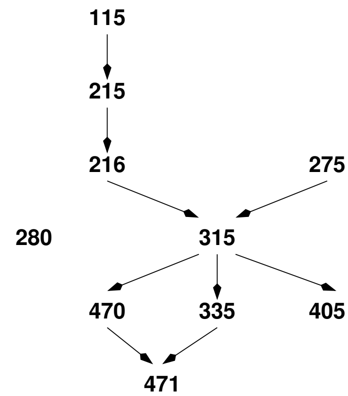
Start at 1.

path	length
1	$\rightarrow$ 0
1 2	$\rightarrow$ 3
1 4	$\rightarrow$ 3
1 2 3	$\rightarrow$ 4
1 2 5	7
1 4 3	5
1 4 5	6
1 2 5	7
1 4 5	6
1 2 3 5	$\rightarrow$ 5

64 Topological sort

- Sample application: course prerequisites place some pairs of courses in order, leading to a **directed, acyclic graph (DAG)**. We want to find a **total order**; there may be many acceptable answers.
- Weiss §9.2





Possible results:

```
    4 10 1 2 6 5 7 8 3 9
    10 4 7 1 2 5 8 6 3 9
```

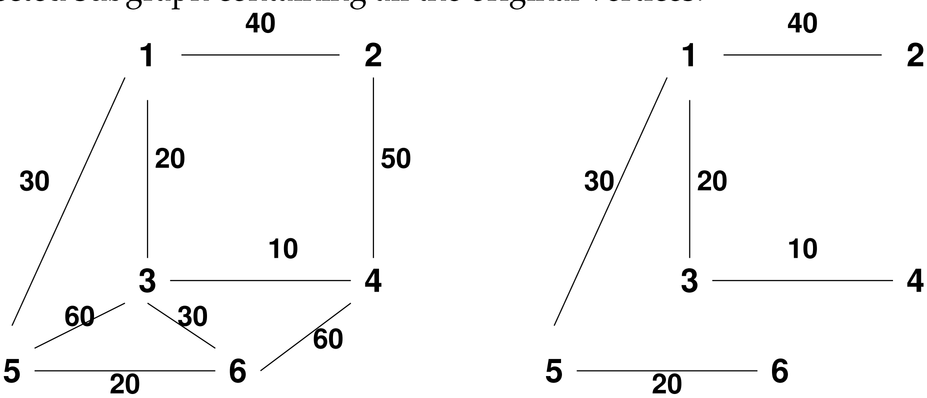
- method: DFS looking for sinks (degrees with fanout 0), which are then placed at the front of the growing result.

```
list answerList; // global

void topologicalSort () { // computes answerList
   foreach j (vertices) visited[j] = false;
   answerList = makeEmptyList();
   foreach j (vertices)
      if (! visited[j]) tsRecurse(j);
} // topologicalSort

void tsRecurse(vertex here) { // adds to answerList
   visited[here] = true;
   foreach next (successors(here))
      if (! visited[next]) tsRecurse(next);
   insertAtFront(answerList, here);
} // tsRecurse
```

65 Spanning trees

- Class 21, 4/8/2021
- Weiss §9.5
- **Spanning tree**: Given a connected undirected graph, a cycle-free

connected subgraph containing all the original vertices.



- **Minimum-weight panning tree**: Given a connected undirected weighted graph, a spanning tree with least total weight.
- Example: minimum-cost set of roads (edges) connecting a set of cities (vertices).

66 Prim's algorithm

```
Start with any vertex as the current tree.
do v − 1 times
   connect the current tree to the closest external vertex
```

- This is a **greedy algorithm**: at each step, improve the solution in the way that looks best at the moment.
- Example: start with 5. We add: (5,6), (5,1), (1,3), (3, 4), (1, 2)
- Implementation
  - Keep a top-light heap of all external vertices based on their distance to the current tree (and store to which tree vertex they connect at that distance).
  - Initially, all distances are ∞ except for the neighbors of the starting vertex.
  - Repeatedly take the closest vertex f and add its edge to the current tree.
  - For all external neighbors b of f, perhaps f is a better way to connect b to the tree; if so, update b's information in the heap. (Remove b and reinsert it with the better distance.)

- Complexity: $\mathcal{O}(v \cdot \log v + e)$, because for we add each vertex once, removing it from a heap that can have v elements; we need to consider each edge twice (once from each end).

67 Kruskal's algorithm

```
Start with all vertices, no edges.
do v − 1 times
   add the lowest-cost missing edge that does not form a cycle
```

- This is a **greedy algorithm**: at each step, improve the solution in the way that looks best at the moment.
- We can stop when we have added $v - 1$ edges; all the rest will certainly introduce cycles.
- Data representation: List of edges, sorted by weight
- Complexity: assuming that keeping track of the component of each vertex is $\mathcal{O}(\log^* v)$, the complexity is $\mathcal{O}(e \log e + v \log^* v)$, because we must sort the edges and then add $v - 1$ edges.

68 Cycle detection: Union-find

- general idea
  - As edges are added, keep track of which connected componenent every vertex belongs to.
  - Any new edge connecting vertices already in the same component would form a cycle; avoid adding such edges.
- operations
  - Each vertex starts as a separate component.
  - union(b,c): assign b and c to the same component (for instance, when an edge is introduced between them).
  - find(b): tell which component b is in (if b and c are in the same component, don't add an edge connecting them).
- method for union(b,c)

  - Every vertex has at most one **parent**, initially nil.
  - Find the **representative** b′ of b by following parent links until the end.
  - Find the representative c′ of c.
  - If b′ = c′, they are already in the same component. Done.
  - Point either b′ to c′ or c′ to b′ by introducing a parent link between them.
  - We want trees to be as shallow as possible. So record the height of each tree in its root. Point the shallower one at the deeper one.
  - We can compress paths while searching for the representative. In this case, the height recorded in the root is just an estimate.

- We use this data structure in Kruskal's algorithm to avoid cycles:

```
typedef struct vertex_s {
   int name; // need not be int
   struct vertex_s *representative; // NULL => me
   int depth; // only if I represent my group; 0 initially
} vertex_t;
```

- Class 22, 4/13/2021
- More examples of Union-Find

69 Numerical algorithms

- We will not look at algorithms for approximation to problems using real numbers; that is the subject of CS321.
- We will study integer algorithms.

70 Euclidean algorithm: greatest common divisor (GCD)

- Examples: gcd(12,60)=12, gcd(15,66)=3, gcd(15,67)=1.

```
int gcd(a, b) {
   while (b != 0) {
      (a,b) = (b, a % b);
   }
   return(a);
} // gcd
```

- Example:

a	12	60	12
b	60	12	0

- Example:

a	15	66	15	6	3
b	66	15	6	3	0

- Example:

a	15	67	15	7	1
b	67	15	7	1	0

71 Fast exponentiation

- Many cryptographic algorithms require raising large integers (thousands of digits) to very large powers (hundreds of digits), modulo a large number (about 2K bits).
- to get a^{64} we only need six multiplications: $(((((a^2)^2)^2)^2)^2)^2$
- to get a^5 we need three multiplications: $a^4 \cdot a \; = \; (a^2)^2 \cdot a$.
- General rule to compute a^e: look at the binary representation of e, read it from left to right. The initial accumulator has value 1.
  - 0: square the accumulator
  - 1: square the accumulator and multiply by a.
- Example: a^{11}. In binary, 11_{10} is expressed as 1011_2. So we get $((((1^2)a)^2)^2 \cdot a)^2 \cdot a$, a total of 4 squares and 3 multiplications, or 7 operations. The first square is always 1^2 and the first multiplication is $1 \cdot a$; we can avoid those trivial operations.
- In cryptography, we often need to compute $a^e \pmod{p}$. Calculate this quantity by performing $\bmod p$ after each multiplication.

- As we read the binary representation of e from left to right, we could start with the leading 0's without any harm.
- Example (run with the *bc* calculator program): $243^{745} \bmod 452$. $745_{10} = 1011101001_2$.

```
a = 243
m = 452
r = 1
r = r^2*a % m
r = r^2 % m
r = r^2*a % m
r = r^2*a % m
r = r^2*a % m
r = r^2 % m
r = r^2*a % m
r = r^2 % m
r = r^2 % m
r = r^2*a % m
r
```

72 Integer multiplication

- Class 23, 4/15/2021
- The BigNum representation: linked list of pieces, each with, say, 2 bytes of unsigned integer, with least-significant piece first. (It makes no difference whether we store those 2 bytes in little-endian or big-endian.)
- Ordinary multiplication of two n-digit numbers x and y costs n^2.
- Anatoly Karatsuba (1962) showed a **divide-and-conquer** method that is better.
  - Split each number into two chunks, each with $n/2$ digits:
    - $x = a \cdot 10^{n/2} + b$
    - $y = c \cdot 10^{n/2} + d$

    The base 10 is arbitrary; the same idea works in any base, such as 2.
  - Now we can calculate $xy = ac10^n + (bc + ad)10^{n/2} + bd$. This calculation uses four multiplications, each costing $(n/2)^2$, so it still

costs n^2. All the additions and shifts (multiplying by powers of 10) cost just $\mathcal{O}(n)$, which we ignore.

- We can use the Recursion Theorem (page 17): $c_n = n + 4c_{n/2}$. Then $a = 4$, $b = 2$, $k = 1$, so $a > b^k$, so $c_n = \Theta(n^{log_b(a)}) = \Theta(n^{log_2(4)}) = \Theta(n^2)$.
- But we can introduce $u = ac$, $v = bd$, and $w = (a+b)(c+d)$ at a cost of $(3/4)n^2$.
- Now $xy = u10^n + (w - u - v)10^{n/2} + v$, which costs no further multiplications.
- Example
  - $x = 3962$, $y = 4481$
  - $a = 39$, $b = 62$, $c = 44$, $d = 81$
  - $u = ac = 1716$, $v = bd = 5022$, $w = (a+b)(c+d) = 12625$
  - $w - u - v = 5887$
  - $xy = 17753722$.
- In *bc*:

```
x = 3962
y = 4481
a = 39
b = 62
c = 44
d = 81
u = a*c
v = b*d
w = (a+b)*(c+d)
x * y
u*10^4 + (w-u-v)*10^2 + v
```

- We can apply this construction recursively. $c_n = n + 3c_{n/2}$. We can again apply the Recursion Theorem (page 17): $a = 3$, $b = 2$, $k = 1$, so $a > b^k$, so $c_n = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$.
- For small n, this improvement is small. But for $n = 100$, we reduce the cost from $10,000$ to about 1480. Running `bc -l`:

```
    power=l(3)/l(2)
    a=100
    e(power*l(a))
```

```
bigInt bigMult(bigInt x, y; int n) {
   // n-chunk multiply of x and y
   bigInt a, b, c, d, u, v, w;
   if (n == 1) return(toBigInt(toInt(x)*toInt(y)));
   a = extractPart(x, 0, n/2 - 1); // high part of x
   b = extractPart(x, n/2, n-1); // low part of x
   c = extractPart(y, 0, n/2 - 1); // high part of y
   d = extractPart(y, n/2, n-1); // low part of y
   u = bigMult(a, c, n/2); // recursive
   v = bigMult(b, d, n/2); // recursive
   w = bigMult(bigAdd(a,b), bigAdd(c,d), n/2); // recursive
   return(
      bigAdd(
         bigShift(u, n),
         bigAdd(
            bigShift(bigSubtract(w, bigAdd(u,v)), n/2),
            v
         ) // add
      ) // add
   );
}
```

73 Strings and pattern matching — Text search problem

- Class 24, 4/20/2021
- The problem: Find a match for pattern p within a text t, where $|p| = m$ and $|t| = n$.
- Application: t is a long string of bytes (a "message"), and p is a short string of bytes (a "word").
- We will look at several algorithms; there are others.
  - Brute force: $\mathcal{O}(mn)$. Typical: $1.1n$ (operations).
  - Rabin-Karp: $\mathcal{O}(n)$. Typical: $7n$.
  - Knuth-Morris-Pratt: $\mathcal{O}(n)$ Typical: $1.1n$.
  - Boyer-Moore: worst $\mathcal{O}(mn)$. Typical: n/m.

- Non-classical version: approximate match, regular expressions, more complicated patterns.

74 Text search — brute force algorithm

- Return the smallest index j such that $t[j \mathrel{..} j+m-1] = p$, or -1 if there is no match.

```
int bruteSearch(char *t, char *p) {
   // returns index in t where p is found, or -1
   const int n = strlen(t);
   const int m = strlen(p);
   int tIndex = 0;
   p[m] = 0xFF; // impossible character; pseudo-data
   while (tIndex+m <= n) { // there is still room to find p
      int pIndex = 0;
      while (t[tIndex+pIndex] == p[pIndex]) // enlarge match
         pIndex += 1;
      if (pIndex == m) return(tIndex); // hit pseudo-data
      tIndex += 1;
   } // there is still room to find p
   return(-1); // failure
} // bruteSearch
```

- Example: p = ”001”, t = ”010001”.
- Worst case: $\mathcal{O}((n-m)m) = \mathcal{O}(nm)$
- If the patterns are fairly random, we observe complexity $\mathcal{O}(n-m) = \mathcal{O}(n)$; in practice, complexity is about $1.1n$.

75 Text search — Rabin-Karp

- Michael Rabin, Richard Karp (1987)
- The idea is to do a preliminary hash-based check each time we increment `tIndex` and skip this value of *tIndex* if there is no chance that this position works.
- Problem: how can we avoid m accesses to compute the hash of the next piece of t?

- We will start with **fingerprinting**, a weak version of the final method, just looking at parity, and assuming the strings are composed of 0 and 1 characters.
- The parity of a string of 0 and 1 characters is 0 if the number of 1 characters is even; otherwise the parity is 1.
- Formula: parity $= \sum_j p[j] \pmod 2$.
- We can compute the parities of windows of $m(= 6)$ bits in t. For example,

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
t	0	0	1	0	1	1	0	1	0	1	0	0	1	0	1	0	0	1	1
`tParity`	1	1	0	1	0	1	0	1	0	1	0	0	1	1					

- Say that $p = 010111$, which has `pParity` $= 0$. We only need to consider matches starting at positions 2, 4, 6, 8, 10, and 11.
- We have saved half the work.
- We can calculate `tParity` quickly as we move p by looking at only 2, not p, characters of t:
  - Initially, $\texttt{tParity}_0 = \sum_{0 \le j < m} t[j] \pmod 2$.
  - Then, $\texttt{tParity}_{j+1} = \texttt{tParity}_j + t[j] + t[j+m] \pmod 2$

```
bit computeParity(bit *string, int length) {
   bit answer = 0;
   for (int index = 0; index < length; index += 1) {
      answer += string[index];
   }
   return (answer & 01);
} // computeParity

int fingerprintSearch(bit *t, bit *p) {
   const int n = strlen(t);
   const int m = strlen(p);
   const int pParity = computeParity(p, m);
   int tParity = computeParity(t, m); // initial substring
   int tIndex = 0;
   while (tIndex+m <= n) { // there is still room to find p
      if (tParity == pParity) { // parity check ok
         int pIndex = 0;
         while (t[tIndex+pIndex] == p[pIndex]) { // enlarge match
            pIndex += 1;
            if (pIndex >= m) return(tIndex);
         } // enlarge match
      } // parity check ok
      tParity = (tParity + t[tIndex] + t[tIndex+m]) & 01;
      tIndex += 1;
   } // there is still room to find p
   return(-1); // failure
} // fingerprintSearch
```

- Instead of bits, we can deal with character arrays.
  - We generalize parity to the exclusive OR of characters, which are just 8-bit quantities.
  - The C operator for exclusive OR is ^.
  - The update rule for tParity is

    ```
    tParity = tParity ^ t[tIndex] ^ t[tIndex+m];
    ```

  - We now have reduced the work to 1/128 (for 7-bit ASCII), not 1/2, for the random case, because only that small fraction of starting positions are worth pursuing.

- The full algorithm extends fingerprinting.
    - Instead of reducing the work to 1/2 or 1/128, we want to reduce it to $1/q$ for some large q.
    - Use this hash function for m bytes $t[j] \ldots t[j+m-1]$: $\sum_{0 \le i < m} 2^{m-1-i} t[j+i] \pmod{q}$. Experience suggests that q should be a prime $> m$.
    - We can still update `tParity` quickly as we move p by looking at only 2, not p, characters of t: $\texttt{tParity}_{j+1} = (t[j+m] + 2(\texttt{tParity}_j - 2^{m-1} t[j])) \pmod{q}$.
    - We can use shifting to compute `tParity` without multiplication: $\texttt{tParity}_{j+1} = (t[j+m] + (\texttt{tParity}_j - (t[j] << (m-1)) << 1) \pmod{q}$. We still need to compute mod q, however.
- $\boxed{\text{Class 25, 4/22/2021}}$
- Monte-Carlo substring search
    - Choose q, a prime q close to but not exceeding mn^2. For instance, if $m = 10$ and $n = 1000$, choose a prime q near 10^7, such as 9,999,991.
    - The probability $1/q$ that we will make a mistake is very low, so just omit the inner loop. We will sometimes have a false positive, with probability, it turns out, less than $2.53/n$.
    - I don't think we save enough computation to warrant using Monte Carlo search. If false positives are very rare, it doesn't hurt to employ even a very expensive algorithm to remove them. Checking anyway is called the "Las-Vegas version".
- The idea is good, but in practice Rabin-Karp takes about $7n$ work.

76 Text search — Knuth–Morris–Pratt

- Donald Knuth, James Morris, Vaughan Pratt, 1970-1977.
- Consider t = `Tweedledee and Tweedledum`, p = `Tweedledum`.
- After running the inner loop of brute-force search to the `u` in p, we have learned much about t, enough to realize that none of the letters up to that point in t (except the first) are `T`. So the next place to start a match in t is not position 1, but position 8.

- Consider t = `pappappappar`, p = `pappar`.
- After running the inner loop of brute-force search to the `r` in p, we have learned much about t, enough to realize that the first place in t that can match p starts not at position 1, but rather in position 3 (the third `p`). Moving p to that position lets us continue in the middle of p, never retreating in t at all.
- How much to shift p depends on how much of it matches when we encounter a mismatch in the inner loop. This **shift table** describes the first example.

p		T	w	e	e	d	l	e	d	u	m
k	-1	0	1	2	3	4	5	6	7	8	9
shift	1	1	2	3	4	5	6	7	8	9	10

- If our match fails at p[8], use shift[7]=8 to reposition the pattern.
- Here is the shift table for the second example.

p		p	a	p	p	a	r
k	-1	0	1	2	3	4	5
shift	1	1	2	2	3	3	6

- Try matching that p against t = `pappappapparrassanuaragh`.

```
int KMPSearch(char *t, char *p) {
   const int n = strlen(t);
   const int m = strlen(p);
   int tIndex = 0;
   int pIndex = 0;
   char shiftTable[m];
   computeShiftTable(p, shiftTable);
   while (tIndex+m <= n) { // there is still room to find p
      while (t[tIndex+pIndex] == p[pIndex]) { // enlarge match
         pIndex += 1;
         if (pIndex >= m) return(tIndex);
      } // enlarge match
      const int shiftAmount = shiftTable[pIndex - 1];
      tIndex += shiftAmount;
      pIndex = max(0, pIndex-shiftAmount);
   } // there is still room to find p
   return(-1); // failure
} // KMPSearch
```

- Unfortunately, computing the shift table, although $\mathcal{O}(m)$, is not straightforward, so we omit it.

- The overall cost is guaranteed $\mathcal{O}(n+m)$, but $m < n$, so $\mathcal{O}(n)$. In practice, it makes about $1.1n$ comparisons.

77 Text search — Boyer – Moore simple

- Robert S. Boyer, J. Strother Moore (1977)
- We start by modifying `bruteSearch` to search from the end of p backwards.

```
int backwardSearch(char *t, char *p) {
   const int n = strlen(t);
   const int m = strlen(p);
   int tIndex = 0;
   while (tIndex+m <= n) { // there is still room to find p
      int pIndex = m-1;
      while (t[tIndex+pIndex] == p[pIndex]) { // enlarge match
         pIndex -= 1;
         if (pIndex < 0) return(tIndex);
      } // enlarge match
      tIndex += 1;
   } // there is still room to find p
   return(-1); // failure
} // backwardSearch
```

- **Occurrence heuristic**: At a mismatch, say at letter α in t, shift p to align the rightmost occurrence of α in p with that α in the text. But don't move p to the left. If α does not occur at all in p, move p to one position after α.
- Method: Initialize `location` array for p:

```
int location[256];
// location[c] is the last position in p holding char c

void initLocation(char *p) {
   const int m = strlen(p);
   for (int charVal = 0; charVal < 256; charVal += 1) {
      location[charVal] = -1;
   }
   for (int pIndex = 0; pIndex < m; pIndex += 1) {
      location[p[pIndex]] = pIndex;
   }
} // initLocation
```

- Let α be the failure character, which is found at a particular `pIndex` and `tIndex`.
- Slide p: `tIndex += max(1, pIndex - location[`α`])`
- This formula works in all cases.
    - α not in p and `pIndex = m-1` $\Rightarrow$ a full shift: `tIndex += m`
    - α not in p and `pIndex = j` $\Rightarrow$ a partial shift, larger if we haven't travelled far along p: `tIndex += pIndex + 1`
    - α is in p. We shift enough to align the rightmost α of p with the one we failed on, or at least shift right by 1.
- Examples
    - p = `rum`, t = `conundrum`. We shift p by 3, another 3, and find the match.
    - p = `drum`, t = `conundrum`. We shift p by 1, by 4, and find the match.
    - p = `natu`, t = `conundrum`. We shift p by 2, then fail.
    - p = `date`, t = `detective`. We would shift p left, so we just shift right 1, then 4, then fail.
- $\boxed{\text{Class 26, 4/27/2021}}$
- **Match heuristic**: Use a shift table (organized for right-to-left search) as with the Knuth–Morris–Pratt algorithm.
- Use both the occurrence and the match heuristics, and shift by the larger of the two suggestions.

- **Horspool's version** (Nigel Horspool, 1980): on a mismatch, look at β, which is the element in t where we started matching, that is, $\beta = t_{i+m-1}$. Shift so that β in t aligns with the rightmost occurrence of β in p (not counting p_{m-1}).
  - This method always shifts p to the right.
  - We need to precompute for each letter of the alphabet where its rightmost occurrence in p is, not counting p_{m-1}. In particular:
  - shift[β] = if β in $p_{0..m-2}$ then $m - 1 - max\{j|j < m - 1, p_j = \beta\}$ else m.

78 Advanced pattern matching, as in Perl

- Based on regular expressions; can be compiled into finite-state automata.
  - exact: `conundrum`
  - don't-care symbols: `con.ndr..`
  - character classes: `c[ou1-5]nundrum`
  - alternation: `c(o|u)nund(rum|ite)`
  - repetition:
    - `c(on)*und`
    - `c(on)+und`
    - `c(on){4,5}und`
  - predefined character classes: `c\wnundrum\d\W`
  - Unicode character classes: `c\p{ASCII}nundrum\p{digit}\p{Final_Punctuation}`
  - pseudo-characters: `^conundrum$`
- Beyond regular expressions in Perl
  - Reference to "capture groups": `con(un|an)dr\1m`
  - Zero-width assertions: `(?=conundrum)`

79 Edit distance

- How much do we need to change s (source) to make it look like d (destination) ?

- Charge 1 for each replacement (R), deletion (D), insertion (I).
- Example: ghost $\stackrel{D}{\to}$ host $\stackrel{I}{\to}$ houst $\stackrel{R}{\to}$ house
- The **edit distance** (s,d) is the smallest number of operations to transform s to d.
- We can build an edit-distance table d by this rule:
  $d_{i,j} = \min(d_{i-1,j} + 1, d_{i,j-1} + 1, d_{i-1,j-1} +$ if s[i] = d[j] then 0 else 1$)$.
- Example: peseta $\to$ presto (should get distance 3).

		-1	0	1	2	3	4	5
			p	e	s	e	t	a
-1		0	1	2	3	4	5	6
0	p	1	0	1	2	3	4	5
1	r	2	1	1	2	3	4	5
2	e	3	2	1	2	2	3	4
3	s	4	3	2	1	2	3	4
4	t	5	4	3	2	2	2	3
5	o	6	5	4	3	3	3	3

- We can trace back from the last cell to see exactly how to navigate to the start cell: pick any smallest neighbor to left/above.
  - $\downarrow$: delete a character from source (left string)
  - $\to$: insert a character from destination (top string)
  - $\searrow$: keep the same character (if number the same) or replace a character in the source (left string) with one from the destination (top string).
- complexity: $\mathcal{O}(nm)$ to calculate the array; the preprocessing is just to start up the array, of cost $\mathcal{O}(n+m)$.
- Another example: convert `banana` to `antenna`. It should take only 4 edits.
- $\boxed{\text{Class 27, 4/29/2021}}$
- This algorithm is in the **dynamic programming** category. Pascal's triangle is another, as is finding the rectangle in an array with the largest sum of values (some negative).

80 Categories of algorithms

- Divide and conquer

- Greedy
- Dynamic programming
- Search

81 Divide and conquer algorithms

- steps
  - if the problem size n is trivial, do it.
  - divide the problem into a easier problems of size n/b.
  - do the a easier problems
  - combine the answers.
- We can usually compute the complexity by the Recursion Theorem (page 17).
- cost: n^k for splitting, recomputing, so $C_n = n^k + aC_{n/b}$.
- Select jth smallest element of an array. $a = 1$, $b = 2$, $k = 1 \Rightarrow \mathcal{O}(n)$.
- Quicksort. $a = 2$, $b = 2$, $k = 1 \Rightarrow \mathcal{O}(n \log n)$.
- Binary search, search in a binary tree. $a = 1$, $b = 2$, $k = 0 \Rightarrow \mathcal{O}(\log n)$.
- Multiplication (Karatsuba) $a = 3$, $b = 2$, $k = 1 \Rightarrow \mathcal{O}(n^{\log_2 3})$.
- Tile an $n \times n$ board that is missing a single cell by a trimino: $a = 4$, $b = 4$, $k = 0 \Rightarrow \mathcal{O}(n)$.

- Mergesort:

```
void mergeSort(int array[], int lowIndex, int highIndex){
   // sort array[lowIndex] .. array[highIndex]
   if (highIndex - lowIndex < 1) return; // width 0 or 1
   int mid = (lowIndex+highIndex)/2;
   mergeSort(array, lowIndex, mid);
   mergeSort(array, mid+1, highIndex);
   merge(array, lowIndex, highIndex);
} // mergeSort
```

$a = 2$, $b = 2$, $k = 1 \Rightarrow \mathcal{O}(n \log n)$.

82 Greedy algorithms

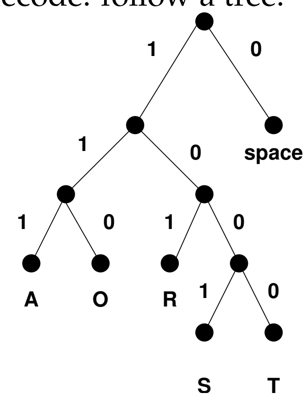
General rule: Enlarge the current solution by selecting (usually in a simple way) the best single-step improvement.

- Computing the coins for change: greedily apply the biggest available coin first.
  - not always optimal: consider denominations 1, 6, 10, and we wish to construct 12.
  - Denominations 1, 5, 10 guarantee optimality.
  - Power-of-two coins would be very nice: no more than 1 of each needed for change. British measures follow this rule: fluid ounce : tablespoon : quarter-gill : half-gill : gill : cup : pint : quart : half gallon : gallon
  - Similar problem: putting weights on barbells.
- Kruskal's algorithm for computing a minimum-cost spanning tree: greedily add edges of increasing weight, avoiding cycles.
- Prim's algorithm for computing a minimum-cost spanning tree: greedily enlarge the current spanning tree with the shortest edge leading out.
- Dijkstra's algorithm for all shortest paths from a source: greedily pick the cheapest extension of all paths so far.
- Hoffman codes for data compression
  - Start with a table of frequencies, like this one:
    ```
    space  60
    A      22
    O      16
    R      13
    S       6
    T       4
    ```
    A text containing all these characters in the given frequencies would take $60 + 22 + 16 + 13 + 6 + 4 = 121$ 7-bit units or 847 bits.
  - Build a table of codes, like this one:

```
space     0
A       111
O       110
R       101
S      1001
T      1000
```

The same text now uses $60 \cdot 1 + 22 \cdot 3 + \ldots + 4 \cdot 4 = 253$ bits.

- To decode: follow a tree:



- To build the tree
  - Each character is a node.
  - Greedily take the two least common nodes, combine them as children of a new parent, and label that parent with the combined frequency of the two children.
- Adding a million real numbers, all in the range $0 \ldots 1$, losing minimal precision
  - Remove the two smallest numbers from the set. (This step is greedy: take the numbers whose sum can be computed with the least precision loss.)
  - Insert their sum in the set.
  - Use a heap to represent the set.
- Continuous knapsack problem
  - Given a set of n objects x_i, each with a weight w_i and profit p_i, and a total weight capacity C, select objects (to put in a knapsack) that together weigh $\leq C$ and maximize profit. We are allowed to take fractions of an object.

- Greedy method
  - Start with an empty knapsack.
  - Sort the objects in decreasing order of p_i/w_i.
  - Greedy step: Take all of each object in the list, if it fits. If it fits partially, take a fraction of the object, then done.
  - Stop when the knapsack is full.
- Example.

chapter	pages (weight)	importance (profit)	ratio
1	120	5	.0417
2	150	5	.0333
3	200	4	.0200
4	150	8	.0533
5	140	3	.0214

  sorted: 4, 1, 2, 5, 3. If capacity $C = 600$, take all of 4, 1, 2, 5, and 40/200 of 3.
- This greedy algorithm happens to be optimal.

- 0/1 knapsack problem: Same as before, but no fractions are allowed. The greedy algorithm is still fast, but it is not guaranteed optimal.

83 Dynamic programming

General rule: Solve all smaller problems and use their solutions to compute the solution to the next problem.

- Compute Fibonacci numbers: $f_i = f_{i-1} + f_{i-2}$.
- Compute binomial coefficients: $C(n, i) = C(n-1, i-1) + C(n-1, i)$.
- Compute minimal edit distance.

84 Summary of algorithms covered

- Class 28, 5/4/2021
- Graphs
  - Computing the degree of all vertices: Loop over representation
  - Computing the connected component containing node i in an undirected graph: Depth-first search, recursive, avoiding vertices already visited

- Breadth-first search: use a queue, iterative, avoiding vertices already visited, perhaps with back-pointers
- Shortest path between nodes i and j: use a priority queue (heap) sorted by distance from i.
- Topological sort: recursive; build list as the last step.
- Dijkstra's algorithm: Finding all shortest paths from given node Greedy: extend currently shortest path
- Prim's algorithm for spanning trees: Greedy, repeatedly add shortest outgoing edge
- Kruskal's algorithm for spanning trees: Greedy, repeatedly add shortest edge that does not build a cycle.
- Cycle detection: Union-find: All vertices in a component point (possibly indirectly) to a representative; union joins representatives.

- Numerical algorithms
  - Euclidean algorithm for greatest common divisor (GCD): Repeatedly take modulus.
  - Fast exponentiation: represent the exponent in binary to guide the steps
  - Integer multiplication (Karatsuba): Subdivide a problem of size $n \times n$ into three problems of size $n/2 \times n/2$.
- Strings and pattern matching — Text search problem
  - Text search — Brute-force algorithm: Try each position for the pattern p.
  - Text search — Rabin-Karp: Hash-based pre-check each time p moves over.
  - Text search — Knuth–Morris–Pratt: Precomputed shift table tells how far to move p on a mismatch.
  - Text search — Boyer – Moore simple: Match starting at the end of p; can jump great distances.
  - Edit distance — Dynamic programming, finding edit distance of several subproblems to guide the next subproblem.
- Miscellaneous
  - Tiling (divide and conquer)

- Mergesort (divide and conquer)
- Computing Fibonacci numbers (dynamic programming)
- Computing binomial coefficients (dynamic programming)
- Continuous knapsack problem (greedy)
- Coin changing (greedy)
- Hoffman codes (greedy)

85 Tractability

- Formal definition of $\mathcal{O}$: $f(n) = \mathcal{O}(g(n))$ iff for adequately large n, and some constant c, we have $f(n) \leq c \cdot g(n)$. That is, f is bounded above by some multiple of g. We can say that f grows no faster than g.
- Formal definition of Θ: $f(n) = \Theta(g(n))$ iff for adequately large n, and some constants c_1, c_2, we have $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$. We say that f grows as fast as g.
- Formal definition of Ω is similar; $f(n) = \Omega(g(n))$ means that f grows at least as fast as g.
- We usually say that a problem is **tractable** if we can solve it in polynomial time (with respect to the problem size n). We also say the program is **efficient**.
  - constant time: $\mathcal{O}(1)$
  - logarithmic time: $\mathcal{O}(\log n)$
  - linear time: $\mathcal{O}(n)$
  - sub-quadratic: $\mathcal{O}(n \log n)$ (for instance)
  - quadratic time: $\mathcal{O}(n^2)$
  - cubic time: $\mathcal{O}(n^3)$
  - These are all bounded by $\mathcal{O}(n^k)$ for some fixed k.
  - However, if k is large, even tractable problems can be infeasible to solve. In practice, algorithms seldom have $k > 3$.
- There are many algorithms that take more than polynomial time.
  - exponential: $\mathcal{O}(2^n)$.
  - super-exponential: $\mathcal{O}(n!)$ (for example)
  - $\mathcal{O}(n^n)$
  - $\mathcal{O}(2^{2^n})$

86 Decision problems, function problems, P, NP

- Decision problems: The answer is just “yes” or “no”.
  - Primality: is n prime? (There are very fast probabilistic algorithms, and recently a polynomial algorithm).
  - Is there a path from a to b shorter than 10?
  - Are two graphs G and F isomorphic? (Apparently very hard)
  - Can graph G be colored with 3 colors? (Apparently very hard)
- Function problems: the answer is a number.
  - What is the smallest prime divisor of n?
  - What is the weight of a minimum-weight spanning tree of G?
- We use P to refer to the set of decision problems that can be decided in polynomial time. That is, for all problems $p \in P$, there must be an algorithm and a positive number k such that the time of the algorithm for p is $\mathcal{O}(|x|^k)$ where $|x|$ means the size of x.
- We use *NP* to refer to the set of decision problems that can be decided in polynomial time if we are allowed to guess a witness to a “yes” answer and only need to check it.
  - Is there a path from a to b shorter than 10? Guess the path, find its length. $\mathcal{O}(1)$.
  - Are two graphs G and F isomorphic? Guess the isomorphism, then check, requiring $\mathcal{O}(v+e)$.
  - Can graph G be colored with 3 colors? Guess the coloring, then demonstrate that it is right; $\mathcal{O}(v+e)$.
  - Is there a set of Boolean values for variables $x_1, ..., x_n$ that satisfies a given Boolean formula (using ”and”, ”or”, and ”not”)? Guess the values, check in linear time (in the length of the formula).
- Properties of P and *NP* (and *EXP*, decision problems that can be solved in $\mathcal{O}(k^n)$).
  - $P \subseteq NP \subseteq EXP$
  - $P \subset EXP$
  - if a problem in *NP* has g possible witnesses, then it has an algorithm in $\mathcal{O}(g^n)$.

- Some problems can be proved to be "hardest" in NP. The are called NP-complete problems. All other problems in NP can be reduced to such NP-complete problems.
- Nobody knows, but people suspect that $P \subset NP \subset EXP$.