1 Intro

Lecture 1, 1/16/2020

- Handout 1 — My names
- TA: Patrick Shepherd
- Plagiarism — read aloud
- Assignments on web. Use C, C++, or Java.
- E-mail list: CS350001@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

Use

Specification

Implementation

4 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.

```
<table>
<thead>
<tr>
<th>operation</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>create empty list</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insert new node at front of list</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>delete first node, returning data</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>count length</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>search by data</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>sort</td>
<td>$O(n \log n)$ to $O(n^2)$</td>
</tr>
</tbody>
</table>
```
5 Sample code

```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 *insertHead(node* handle, int data) {
    return makeNode(data, handle->next);
} // insertHead

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

node *searchDataR(node *handle, int data) {
    // recursive method
    node *current = handle->next;
    if (current == NULL) return NULL;
    else if (current->data == data) return current;
    else return searchDataR(current, data);
} // searchDataR
```
6 Improving the efficiency of some operations

- To make insert at end fast: maintain two handles, one to the front, the other to the rear of the list.
- 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.
- Combine these new items in a header node:

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

- 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.

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

```c
node *searchDataI(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);
} // searchDataI
```

- 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 searchDataR() 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 Queues, stacks, deques: built out of either linked lists or arrays

- We’ll see each of these.

8 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

9 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

10 Implementation 2 of Stack: Array

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

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

- The array implementation limits the size. Does the linked-list implementation also limit the size?
- The array implementation needs one cell for (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.
11 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 insertQueue(queue Q, int I) // modifies Q
  - int deleteQueue(queue Q) // modifies Q

12 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 cards 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 | cards, the list of numbers from randGen.pl is redirected as input to cards.

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

int deleteQueue(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;
} // deleteQueue
```
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 advance(int index) { // circular advance
    return (index + 1) % MAXQUEUESIZE;
} // advance

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

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

- A singly-linked list is fine except for deleteRearDequeue(), which becomes $O(n)$.

- The proper list structure is a **doubly-linked list** with a single dummy.

- 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

- Given $n$ data elements (we will use integer data), arrange them in a data structure $D$ so that these operations are fast:
  - 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 $O(1)$ (place at front)
  - search(i) is $O(n)$ (may need to look at whole list; use pseudo-data $i$ to make search as fast as possible)
- Representation 2: Sorted linked list
  - insert(i) is $O(n)$ (average: $n/2$ steps)
  - search(i) is $O(n)$ (may need to look at whole list; on average, look at $n/2$ elements. Use pseudo-data (value $\infty$) to make search as fast as possible.)
- Representation 3: Array
  - insert(i) is $O(1)$ (place at end)
  - search(i) is $O(n)$ (may need to look at whole list; use pseudo-data $i$ at end)
- Representation 4: Sorted array
  - insert(i) is $O(n)$ (need to search, then shove members over)
- search(i) is $O(\log n)$ (binary search)

```c
bool search(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);
} // search
```
17 Quadratic search: set mid based on discrepancy

Also called interpolation search, extrapolation search, dictionary search.

```cpp
bool search(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);
} // search
```

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 $O(\log n)$.
- 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 $O(\log \log n)$.
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

<table>
<thead>
<tr>
<th>( a/b )</th>
<th>( c_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a &lt; b^k )</td>
<td>( \Theta(n^k) )</td>
</tr>
<tr>
<td>( a = b^k )</td>
<td>( \Theta(n^k \log n) )</td>
</tr>
<tr>
<td>( a &gt; b^k )</td>
<td>( \Theta(n^{\log a}) )</td>
</tr>
</tbody>
</table>

- In our case, \( a = 1 \), \( b = 2 \), \( k = 0 \), so \( b^k = 1 \), so \( a = b^k \), so \( c_n = \Theta(n^0 \log n) = \Theta(\log n) \).
- 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

- Example with elicited values
- Pseudo-data: in the universal “null” node.
- \( \text{insert}(i) \) and \( \text{search}(i) \) 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) {
        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)

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

- **Pre-order traversal**

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

- **Post-order traversal**

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

21 Representation 6: Hashing (scatter storage)

- Hashing is often the best method.
- `insert(data)` and `search(data)` are $O(\log n)$, but we can generally treat them as $O(1)$. 
• We will discuss hashing later.

22 Finding the $j$th largest element in a set

• Lecture 6, 2/4/2020

• If $j = 1$, a single pass works in $O(n)$ time:

```plaintext
largest = -\infty; // priming
foreach (value in set) {
  if (value > largest) largest = value;
}
```

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

```plaintext
largest = nextLargest = -\infty; // priming
foreach (value in set) {
  if (value > largest) {
    nextLargest = largest;
    largest = value;
  } else if (value > nextLargest) {
    nextLargest = value;
  }
}
```

• It appears that for arbitrary $j$ we need $O(jn)$ time, because each iteration needs $t$ tests, where $1 \leq t \leq j$, followed by sliding $j + 1 - t$ values over, for a total cost of $j + 1$.

• 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 $j$th element falls. Stop if the $j$th element is the pivot.

• Cost: $n + n/2 + n/4 + \ldots = 2n = O(n)$

• We can also compute the cost using the recursion theorem (page 16):
  
  • $c_n = n + c_{n/2}$ (if we are lucky)
  
  • $c_n = n + c_{2n/3}$ (fairly average case)
\[ f(n) = n = O(n^1) \]
\[ k = 1, a = 1, b = 2 \text{ (or } b = 3/2) \]
\[ a < b^k \]
\[ \text{so } c_n = \Theta(n^k) = \Theta(n) \]

## 23 Partitioning an array

- Nico Lomuto’s 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.

```c
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 partition (pivotValue) 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);
}
```

- Example
<table>
<thead>
<tr>
<th></th>
<th>5</th>
<th>2</th>
<th>d</th>
<th>c</th>
<th>1</th>
<th>7</th>
<th>9</th>
<th>0</th>
<th>3</th>
<th>6</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>d</td>
<td>d,c</td>
<td>d</td>
<td>c</td>
<td>1</td>
<td>7</td>
<td>9</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>d</td>
<td>c</td>
<td>c</td>
<td>1</td>
<td>d</td>
<td>c</td>
<td>9</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>d</td>
<td>d</td>
<td>0</td>
<td>d</td>
<td>9</td>
<td>c</td>
<td>7</td>
<td>3</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>d</td>
<td>d</td>
<td>c</td>
<td>c</td>
<td>3</td>
<td>d</td>
<td>7</td>
<td>c</td>
<td>9</td>
<td>6</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>d</td>
<td>d</td>
<td>c</td>
<td>c</td>
<td>4</td>
<td>d</td>
<td>9</td>
<td>6</td>
<td>c</td>
<td>7</td>
<td>8</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
24 Using partitioning to select $j$th smallest

```java
void 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;
        } else if (midIndex < targetIndex) { // look to right
            lowIndex = midIndex + 1;
        } else { // look to left
            highIndex = midIndex - 1;
        }
    } // while lowIndex < highIndex
} // selectJthSmallest
```

25 Sorting

- We usually are interested in sorting an array in place.
- Sorting is $\Omega(n \log n)$.
- Good methods are $O(n \log n)$.
- Bad methods are $O(n^2)$.

26 Sorting out sorting

- https://www.youtube.com/watch?v=YvTW7341kpA
- https://www.youtube.com/watch?v=plAi7kcqMNU
- https://www.youtube.com/watch?v=gtdfW3TbeYY
- https://www.youtube.com/watch?v=wdcoRfS8edM
# Insertion sort

- Comb method:

<table>
<thead>
<tr>
<th>sorted</th>
<th>unsorted</th>
</tr>
</thead>
</table>

  - probe

  - $n$ iterations.
  - Iteration $i$ may need to shift the probe value $i$ places.
  - $\Rightarrow O(n^2)$.
  - Experimental results

<table>
<thead>
<tr>
<th>$n$</th>
<th>compares</th>
<th>moves</th>
<th>$n^2/2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>2644</td>
<td>2545</td>
<td>5000</td>
</tr>
<tr>
<td>200</td>
<td>9733</td>
<td>9534</td>
<td>20000</td>
</tr>
<tr>
<td>400</td>
<td>41157</td>
<td>40758</td>
<td>80000</td>
</tr>
</tbody>
</table>
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

28 Selection sort

- Comb method:

```
sorted, small  unsorted, large

  smallest

sorted, small  unsorted, large
```

- $n$ iterations.
- Iteration $i$ may need to search through $n - i$ places.
- $\Rightarrow O(n^2)$.
- Experimental results
<table>
<thead>
<tr>
<th>n</th>
<th>compares</th>
<th>moves</th>
<th>(n^2/2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>4950</td>
<td>198</td>
<td>5000</td>
</tr>
<tr>
<td>200</td>
<td>19900</td>
<td>398</td>
<td>20000</td>
</tr>
<tr>
<td>400</td>
<td>79800</td>
<td>798</td>
<td>80000</td>
</tr>
</tbody>
</table>

```c
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
```
29 Quicksort (C. A. R. Hoare)

- Recursive based on partitioning:

```
random
  partition
  small  big
  sort   sort
```

- about $\log n$ depth.
- each depth takes about $O(n)$ work.
- $\Rightarrow O(n \log n)$.
- Can be unlucky: $O(n^2)$.
- To prevent worst-case behavior, partition based on median of 3 or 5.
- Don’t quicksort regions smaller than about 6 cells; use a final insertionSort pass instead.

- Experimental results

<table>
<thead>
<tr>
<th>$n$</th>
<th>compares</th>
<th>moves</th>
<th>$n \log n$</th>
<th>$n^2/2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>643</td>
<td>824</td>
<td>664</td>
<td>5000</td>
</tr>
<tr>
<td>200</td>
<td>1444</td>
<td>1668</td>
<td>1528</td>
<td>20000</td>
</tr>
<tr>
<td>400</td>
<td>3885</td>
<td>4228</td>
<td>3457</td>
<td>80000</td>
</tr>
<tr>
<td>800</td>
<td>8066</td>
<td>8966</td>
<td>7715</td>
<td>320000</td>
</tr>
<tr>
<td>1600</td>
<td>17583</td>
<td>18958</td>
<td>17030</td>
<td>1280000</td>
</tr>
</tbody>
</table>
```java
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)

- Donald Shell, 1959
- Each pass has a span $s$.
  ```java
  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: similar to Quicksort!
  \[
  \begin{array}{cccccc}
  n & \text{compares} & \text{moves} & n \log n & n^2/2 \\
  \hline
  100 & 355 & 855 & 664 & 5000 \\
  200 & 932 & 1932 & 1528 & 20000 \\
  400 & 2266 & 4666 & 3457 & 80000 \\
  800 & 5216 & 10816 & 7715 & 320000 \\
  1600 & 11942 & 24742 & 17030 & 1280000 \\
  \end{array}
  \]