Divide and Conquer

CS 2110: Software Development Methods

April 12, 2019

Recursion

- Recursion breaks a difficult problem into one or more simpler versions of itself
- A definition is **recursive** if it is defined in terms of itself
- Questions to ask yourself:
  - How can we reduce the problem into smaller version of the same problem?
  - How does each call make the problem smaller?
  - What is the **base case**?
  - Will we always reach the base case?

Definitions

Base case

The case for which the solution can be stated nonrecursively

Recursive case

The case for which the solution is expressed in terms of a smaller version of itself

Factorial

```
public int factorial(int n) {
    // base case (always first)
    if (n <= 0)
        return 1;

    // recursive case
    return n * factorial(n-1);
}
```

Recursion Summary

Recursion is tricky!

- Always put base case first
- Base case should eventually happen given any input
- Recursive solution may not always be the best

Divide and Conquer

Divide and Conquer: putting recursion to work for you!

- An **algorithm design strategy**, one of many you will learn
- Strategy: It's often easier to solve several *small* instances of a problem rather than one large problem
  - **Divide** the problem into k smaller instances
  - **Conquer**: solve each of those k problems (recursively)
  - **Combine** the solutions to subproblems into one final solution
- Note: must have a base case to solve *really small* problems directly

Divide and Conquer: Strategy

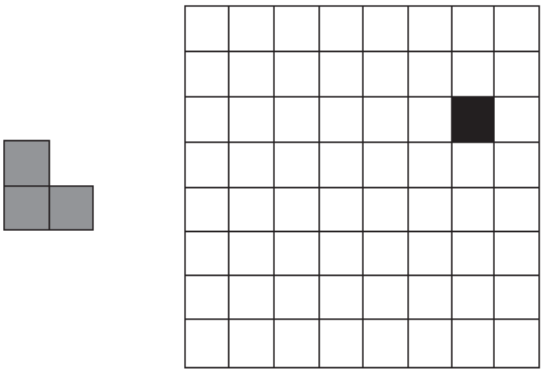
```
solve(A) {      // solve for input A
    n = size(A) // size of our problem is n
    // base case
    if (n <= smallSize) { // problem is small enough
        solution = directlySolve(A)
    } else { // recursive case
        divide A into A1, A2, ..., Ak
        for each i in {1, ..., k}:
            Si = solve(Ai)  // conquer each sub-problem
        solution = combine(S1, S2, ..., Sk)
    }
    return solution
}
```

Divide and Conquer: Why?

- Sometimes it is the simplest approach
- May be more efficient than an “obvious” approach
  - Ex: Binary Search instead of Sequential Search
  - Ex: Mergesort or Quicksort instead of Insertion Sort
- Not necessarily the most efficient solution
- Divide and Conquer algorithms illustrate a **top-down strategy**
  - Given a large problem, identify and break into smaller subproblems, then combine the results

Tromino Puzzle

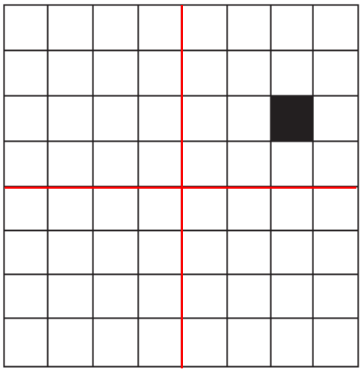
Given an 8×8 board with one piece missing, tile with L-shaped trominoes.



Interactive: http://goo.gl/npFQUD

Tromino Puzzle

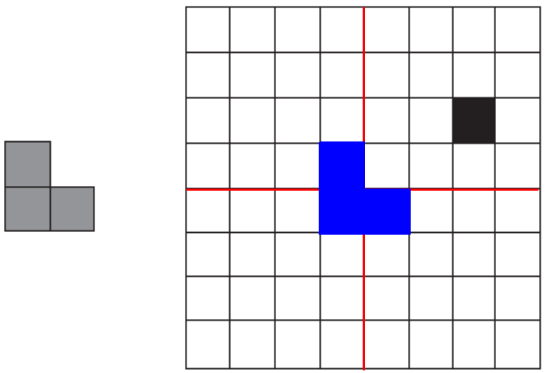
Given an 8×8 board with one piece missing, tile with L-shaped trominoes.



Interactive: http://goo.gl/npFQUD

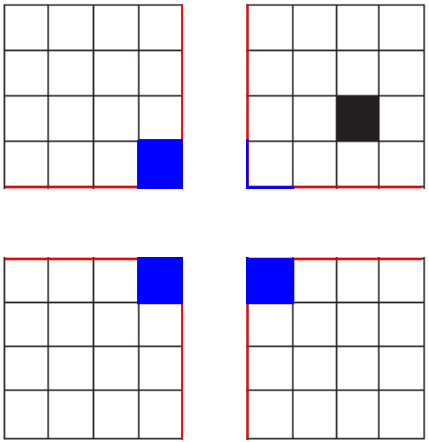
Tromino Puzzle

Given an 8×8 board with one piece missing, tile with L-shaped trominoes.



Interactive: http://goo.gl/npFQUD

Tromino Puzzle



Tromino Puzzle

Base case: simple enough to solve without recursion!

Binary Search

Quickly turn to page 394 (without magic)

Binary Search: Recursive

```
int binSearch(int[] array, int first, int last, int target) {
    if (first <= last) {
        int mid = (first + last) / 2;
        if (target == array[mid])
            return mid;
        if (target < array[mid])
            return binSearch(array, first, mid - 1, target);
        else if (target > array[mid]);
            return binSearch(array, mid + 1, last, target);
    }
    return -1;
}
```

Binary Search: Iterative

```
int binSearch(int[] array, int target) {
    int first = 0;
    int last = array.length - 1;
    while (first <= last) {
        int mid = (first + last) / 2;
        if (target == array[mid])
            return mid;
        if (target < array[mid])
            last = mid - 1;
        else if (target > array[mid]);
            first = mid + 1;
    }
    return -1;
}
```

Mergesort

Specification

- Input: Array `E` and indices `first` and `last`
- Output: Sorted rearrangement of the elements in `E` between `first` and `last`

Mergesort is a classic example of Divide and Conquer

- **Divide**: split the array into two halves
- **Conquer**: call mergesort() to recursively sort the two halves
- **Combine**: combine the two sorted halves into one final sorted array
  - This is the **merge** step of mergesort, and where it gets its name!

Base case:

Mergesort

Specification

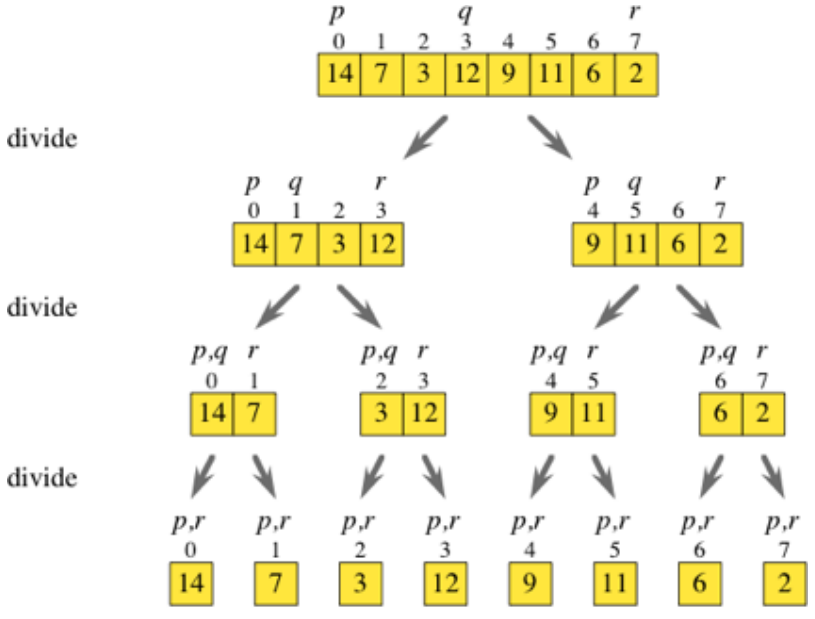
- Input: Array `E` and indices `first` and `last`
- Output: Sorted rearrangement of the elements in `E` between `first` and `last`

Mergesort is a classic example of Divide and Conquer

- **Divide**: split the array into two halves
- **Conquer**: call mergesort() to recursively sort the two halves
- **Combine**: combine the two sorted halves into one final sorted array
  - This is the **merge** step of mergesort, and where it gets its name!

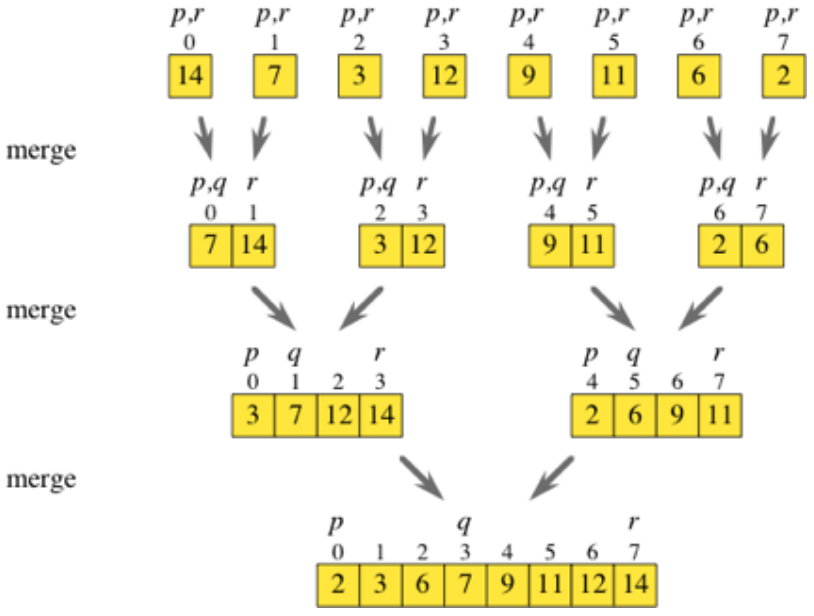
Base case: 1 element (sorted) or 2 elements (compare and swap)

Mergesort



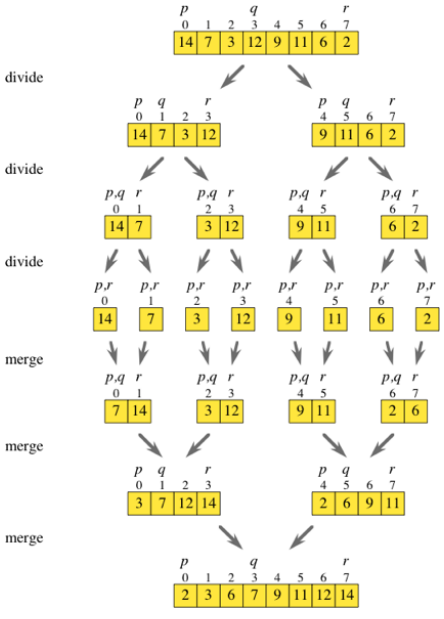
https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/overview-of-merge-sort

Mergesort



https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/overview-of-merge-sort

Mergesort



https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/overview-of-merge-sort

Mergesort

Mergesort in code:

```
public static void mergeSort(Element[] E, int first, int last) {
    if (first < last) { // base case == 1 element
        int mid = (first + last) / 2;
        mergeSort(E, first, mid); // sort first half
        mergeSort(E, mid + 1, last); // sort second half
        merge(E, first, mid, last);  // merge two halves
    }
}
```

Mergesort

```
public static void merge(Element[] E, int first, int mid, int last) {
    Element[] C = new Element[last-first+1];
    int a = first;
    int b = mid + 1;
    int i = 0;
    while (a <= mid && b <= last) {
        if (E[a] <= E[b])
            C[i++] = E[a++];
        else
            C[i++] = E[b++];
    }
    while (a <= mid)
        C[i++] = E[a++];
    while (b <= last)
        C[i++] = E[b++];
    for (int j = 0; j < C.length; j++)
        E[first + j] = C[j];
}
```

Mergesort

Mergesort is $O(n \log n)$

- Same order-class as the most efficient sorts (quicksort, heapsort)
- Faster than Selection Sort, Bubble Sort, and Insertion sort
- Plug: Take CS 2150 and CS 4102 to study the efficiency of this and other recursive algorithms!
- The Divide and Conquer approach matters, and in this case, it's a "win!"

Recursion

Recursion is a natural way to solve many problems

- Sometimes it's a clever way to solve a problem that is not clearly recursive
- Sometimes it produces an efficient solution (mergesort)
- Sometimes it produces a less efficient solution (fibonacci)
- Whether or not to use recursion is a design decision for your "toolbox"

Recursion

Don't forget the rules of recursion

- Identify one or more (simple) **base cases** that can be solved without recursion
  - Handle these FIRST in your code!
- Determine what **recursive call(s)** are needed to solve the subproblems
- Note how to use the results to solve the larger problem
- **Hint**: At this step, don't think about how recursive calls process smaller inputs, just assume they work!