Introduction to Analysis of Algorithms
Thoughts on Assignment 4

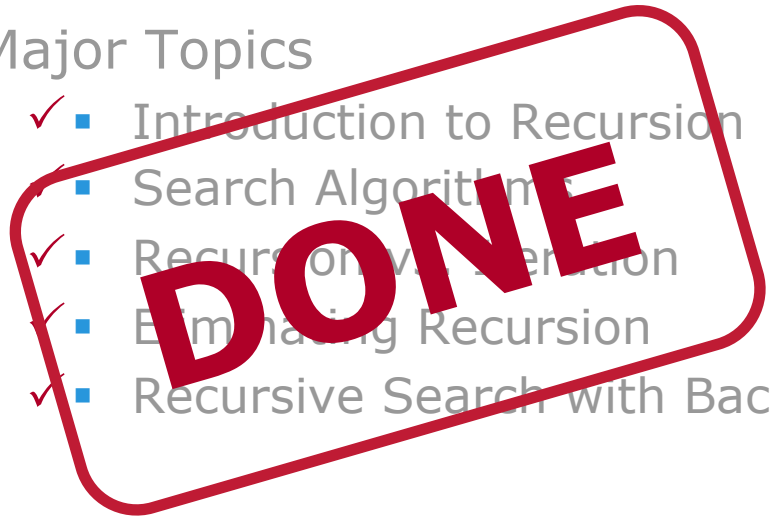
CS 311 Data Structures and Algorithms
Lecture Slides
Monday, February 25, 2013

Chris Hartman
Department of Computer Science
University of Alaska Fairbanks
cmhartman@alaska.edu
Based on material by Glenn G. Chappell

Unit Overview
Recursion & Searching

Major Topics

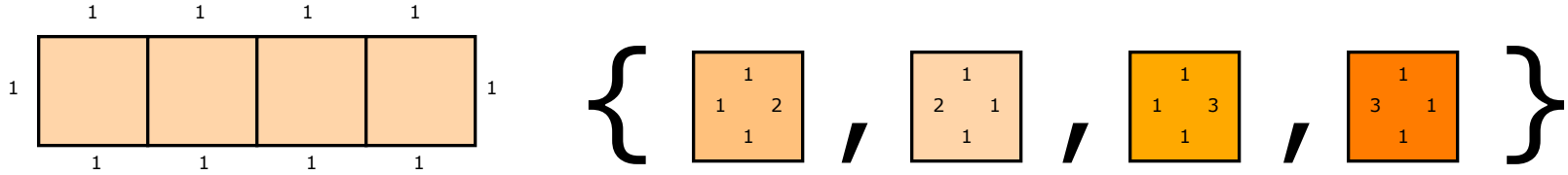
- ✓ Introduction to Recursion
- ✓ Search Algorithms
- ✓ Recursion vs. Iteration
- ✓ Eliminating Recursion
- ✓ Recursive Search with Backtracking



Recursive Search with Backtracking
Rectangle Tiling — Problem Description [1/4]

Now we look at the problem you are to solve in Assignment 4.

Consider a rectangle divided into squares, and a set of square tiles with a positive number on each edge. An example 4 by 1 board and some tiles are shown below.



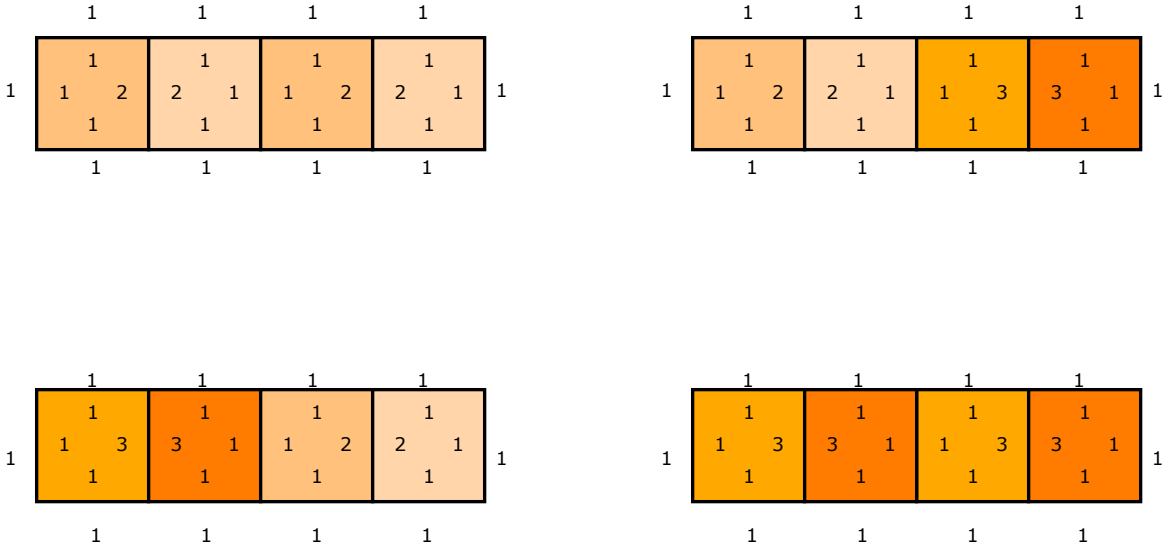
How many ways you can fill the rectangle with tiles from the set so that the numbers on the edges of the tiles match?

- The edges of the board must match “1”.
- You can use as many of each type of tile as you like.
- Tiles can not be rotated.
- The numbers will always be positive.

Recursive Search with Backtracking
Rectangle Tiling — Problem Description [2/4]

How many different tilings does this board have?

- The answer turns out to be four:



Recursive Search with Backtracking
Rectangle Tiling — Problem Description [3/4]

Describing a Tile (from Tile.h)

```
class Tile
{
public:
   Tile()
      :top_(0), right_(0), bot_(0), left_(0)
      {}

   Tile(int top, int right, int bot, int left)
      :top_(top), right_(right), bot_(bot), left_(left)
      {}

      int getTop() const {return top_;}
      int getRight() const {return right_;}
      int getBot() const {return bot_;}
      int getLeft() const {return left_;}

private:
   int top_;
   int right_;
   int bot_;
   int left_;
};

typedef vector<Tile> TileSet;
```

Recursive Search with Backtracking
Rectangle Tilings — Problem Description [4/4]

Your job in Assignment 4 is to write a function **`countRT`** that takes a a vector of tiles and a board size, and returns the number of rectangle tilings possible with that set of tiles on that board.

For example, the following code (representing the previous example) should return 4:

```
TileSet mytiles;

mytiles.push_back(Tile(1,2,1,1));
mytiles.push_back(Tile(1,1,1,2));
mytiles.push_back(Tile(1,3,1,1));
mytiles.push_back(Tile(1,1,1,3));

cout << countRT(mytiles,4,1);
```

Recursive Search with Backtracking
Rectangle Tilings — Writing It [1/7]

We consider how to write **`countRT`** using the recursive methods discussed earlier.

In the following slides I will explain how I did it. You may take these as *suggestions* for how you can do it. However, the requirements of the assignment allow for quite a bit of variation. You do not need to follow my suggestions.

Requirements in a nutshell:

- Wrapper function **`countRT`**.
- Recursive workhorse function **`countRT_recurse`**.
  - Given a partial solution.
  - Returns number of full solutions based on this partial solution.

Note: If you have trouble with this sort of thing, then *follow my suggestions!*

Recursive Search with Backtracking
Rectangle Tilings — Writing It [2/7]

What **information** do we need to maintain?

Ideas:

- A board is a vector of `Tile`s, each corresponding to a square. We can put default tiles (0,0,0,0) to represent unfilled squares.

```
typedef vector<Tile> BoardType;
```

- We also need to know:
  - The board size: *x* & *y*.
  - How much of the board we've filled: *x* & *y*.
  - How many squares are left to fill.

Wrap this information in an object, *if you want*. (I did not.)

Do not use global variables!

- Remember: recursion.

Recursive Search with Backtracking
Rectangle Tilings — Writing It [3/7]

Conceptually, a board is a 2-D array.

However, I suggested storing it in a 1-D (smart) array.

- Why: 2-D smart (or dynamic) arrays in C++ are a pain to declare and initialize. (Dr. Chappell suggests avoiding them. Dr. Hartman is ambivalent.)
- How: You can simulate a 2-D array using a 1-D array in the same way that C++ does it internally.

In order to simulate a 2-D **`int`** array with dimensions **`size_x`** and **`size_y`** (think "**`int array[size_y][size_x];`**"), we can use an ordinary vector with size **`size_x*size_y`**.

```
BoardType b(size_x*size_y);  // size = size_x*size_y
```

- To look up item **`i,j`** (think "**`array[i][j]`**") use the subscript **`i*size_x+j`**.

```
b[i*size_x+j] = 1; // item i,j in conceptual 2-D array
```

Don't change the way you *think* about the array; change the *syntax* you use to access it.

Recursive Search with Backtracking
Rectangle Tilings — Writing It [4/7]

What is a **partial solution**?

- It represents a board partially filled with tiles.
- Rules. A partial solution:
  - Starts at the top left.
  - Fills from top to bottom, left to right.

Therefore, we have a *full* solution if:

- There are no more squares left to fill.

```
int countRT_recurse(const Tileset& tiles,
                    BoardType board,
                    int size_x, int size_y,
                    int pos_x, int pos_y,
                    int squaresLeft)
{
    if (squaresLeft == 0)
        return 1;  // We have a full solution
```

In an *empty* solution:

- The board is all default tiles.
- pos_x and pos_y are 0 (upper left).
- The number of squares left to visit is the number of squares on the board.

Recursive Search with Backtracking
Rectangle Tilings — Writing It [5/7]

Procedure for workhorse function:

- Check for a full solution (previous slide).
  - If so, return 1.
- Set *total* to zero.
- For each possible tile
  - Check if this tile can be placed in this spot.
    - Is there a tile to the left? Does it match it?
      - Does it's left number match the tile to the left's right number?
    - Is there a tile above it? Does it match it?
      - Does it's top number match the tile above's bottom number?
  - If so:
    - Put this tile on the board.
    - Decrement number of squares left.
    - Calculate next tile position.
    - Make recursive call.
    - Add return value to *total*.
    - Restore all changes, except change to *total*.
- Return *total*.

Recursive Search with Backtracking
Rectangle Tilings — Writing It [6/7]

More Suggestions

- If you are careful to leave the board in the same state when **`countRT_recurse`** ends as when it began, then you can pass the board by reference, avoiding the copy.

```
int countRT_recurse(const Tileset& tiles,
                    BoardType& board, …
```

- If you use the 1-D array idea then you should still treat it like a 2-D array. Keep track of *x* and *y*, not the array index.
- Remember to subtract **`1`** from **`squaresLeft`** when making a recursive call.

Rectangle Tilings — Writing It [7/7]

Final Notes

- Again, you do **not** have to write your code the way I have outlined. Any code that meets the requirements of the assignment is acceptable. In particular:
  - Function **`countRT`**: prototyped as required (see blackboard Assignment).
  - Function **`countRT_recurse`**: recursive, takes partial solution and counts final solutions, does the bulk of the work.
- **Think first!** This assignment generally requires less writing than other assignments, but more thought.

Unit Overview
Algorithmic Efficiency & Sorting

We now begin a unit on algorithmic efficiency & sorting algorithms.

Major Topics

- Introduction to Analysis of Algorithms
- Introduction to Sorting
- Comparison Sorts I
- More on Big-*O*
- The Limits of Sorting
- Divide-and-Conquer
- Comparison Sorts II
- Comparison Sorts III
- Radix Sort
- Sorting in the C++ STL

Part way through this unit will be the in-class Midterm Exam.

Introduction to Analysis of Algorithms
Efficiency [1/3]

What do we mean by an "efficient" algorithm?

- We mean an algorithm that **uses few resources**.
- By far the most important resource is **time**.
- Thus, when we say an algorithm is **efficient**, *assuming we do not qualify this further*, we mean that it can be executed **quickly**.

How do we determine whether an algorithm is efficient?

- Implement it, and run the result on some computer?
- But the speed of computers is not fixed.
- And there are differences in compilers, etc.

Is there some way to measure efficiency that does not depend on the system chosen or the current state of technology?

Introduction to Analysis of Algorithms
Efficiency [2/3]

Is there some way to measure efficiency that does not depend on the system chosen or the current state of technology?

- Yes!

Rough Idea

- Divide the tasks an algorithm performs into "steps".
- Determine the maximum number of steps required for input of a given size. Write this as a formula, based on the size of the input.
- Look at the most important part of the formula.
  - For example, the most important part of "$6n \log n + 1720n + 3n^2 + 14325$" is "$n^2$".

Next we look at this in more detail.

Introduction to Analysis of Algorithms
Efficiency [3/3]

When we talk about **efficiency** of an algorithm, without further qualification of what "efficiency" means, we are interested in:

- **Time** Used by the Algorithm
  - Expressed in terms of number of steps.
- How the **Size of the Input** Affects Running Time
  - Larger input typically means slower running time. How much slower?
- **Worst-Case** Behavior
  - What is the maximum number of steps the algorithm ever requires for a given input size?

To make the above ideas precise, we need to say:

- What is meant by a **step**.
- How we measure the **size** of the input.

These two are part of our **model of computation**.

The **model of computation** used *in this class* will include the following definitions.

- The following operations will be considered a single **step**:
  - Built-in operations on fundamental types (arithmetic, assignment, comparison, logical, bitwise, pointer, array look-up, etc.).
  - Calls to client-provided functions (including operators). In particular, in a template, operations (i.e., function calls) on template-parameter types.
- From now on, when we discuss efficiency, we will always consider a function that is given a list of items. The **size** of the input will be the number of items in the list.
  - The "list" could be an array, a range specified using iterators, etc.
  - We will generally denote the size of the input by "*n*".

Notes

- As we will see later, we can afford to be *somewhat* imprecise about what constitutes a single "step".
- In a formal mathematical analysis of the properties and limits of computation, both of the above definitions would need to change.

Order & Big-*O* Notation — Definition

Algorithm *A* is *order* $f(n)$ [written $O(f(n))$] if

- There exist constants k and n_0 such that
- *A* requires **no more than** $k \times f(n)$ time units to solve a problem of size $n \geq n_0$.

We are usually not interested in the exact values of k and n_0.

Thus:

- We don't worry much about whether some algorithm is (say) five times faster than another.
- We ignore small problem sizes.

Big-*O* is important!

- We will probably use it *every day* for the rest of the semester (the concept, not the above definition).

When we use big-*O*, unless we say otherwise, we are always referring to the **worst-case** behavior of an algorithm.

- For input of a given size, what is the **maximum** number of steps the algorithm requires?

We can also do average-case analysis. However, we need to say so. We also need to indicate what kind of average we mean. For example:

- We can determine the average number of steps required over all inputs of a given size.
- We can determine the average number of steps required over repeated applications of the same algorithm.

Order & Big-*O* Notation — Example 1, Problem

Determine the order of the following, and express it using "big-*O*":

```
int func1(int p[], int n) // n is length of array p
{
    int sum = 0;
    for (int i = 0; i < n; ++i)
        sum += p[i];
    return sum;
}
```

See the next slide.

Order & Big-*O* Notation — Example 1, Solution

I count 9 single-step operations in **`func1`**.

Strictly speaking, it is correct to say that **`func1`** is $O(4n+6)$. In practice, however, we always place a function into one of a few well-known categories.

ANSWER: Function **`func1`** is $O(n)$.

- This works with (for example) $k = 5$ and $n_0 = 100$.
- That is, $4n + 6 \le 5 \times n$, whenever $n \ge 100$.

What if we count "**`sum += p[i]`**" as one step? What if we count the loop as one?

- Moral: collapsing a **constant** number of steps into one step does not affect the order.
- This is why I said we can be *somewhat* imprecise about what a "step" is.

Operation	Times Executed
`int p[]`	1
`int n`	1
`int sum = 0`	1
`int i = 0`	1
`i < n`	$n + 1$
`++i`	n
`p[i]`	n
`sum += …`	n
`return sum`	1
TOTAL	$4n + 6$

Why are we so interested in the running time of an algorithm for **very large** problem sizes?

- Small problems are easy and fast.
- We expect more of faster computers. Thus, problem sizes keep getting bigger.
- As we saw with search algorithms, the advantages of a fast algorithm become more important at very large problem sizes.

Recall:

- "The fundamental law of computer science: As machines become more powerful, the efficiency of algorithms grows more important, not less." — Nick Trefethen

An algorithm (or function or technique …) that works well when used with increasingly large problems & large systems is said to be **scalable**.

- Or, it **scales well**.
- This class is all about things that scale well.

This definition applies in general, not only in computing.

Introduction to Analysis of Algorithms
Order & Big-*O* Notation — Efficiency Categories

Know these!

An $O(1)$ algorithm is **constant time**.
- The running time of such an algorithm is essentially independent of the input.
- Such algorithms are rare, since they cannot even read all of their input.

An $O(\log_b n)$ [for some b] algorithm is **logarithmic time**.
- Again, such algorithms cannot read all of their input.
- As we will see, we do not care what b is.

An $O(n)$ algorithm is **linear time**.
- Such algorithms are not rare.
- This is as fast as an algorithm can be and still read all of its input.

An $O(n \log_b n)$ [for some b] algorithm is **log-linear time**.
- This is about as slow as an algorithm can be and still be truly useful (scalable).

An $O(n^2)$ algorithm is **quadratic time**.
- These are usually too slow for anything but very small data sets.

An $O(b^n)$ [for some b] algorithm is **exponential time**.
- These algorithms are *much* too slow to be useful.

Faster

Slower

Notes
- Gaps between these categories are *not* bridged by compiler optimization.
- We are interested in the **fastest category** above that an algorithm fits in.
  - Every $O(1)$ algorithm is also $O(n^2)$ and $O(237^n + 184)$; but "$O(1)$" interests us most.
- **I will also allow $O(n^3)$, $O(n^4)$, etc.** However, we will not see these much.