Recursive Search with Backtracking continued
Introduction to Analysis of Algorithms

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, October 5, 2009

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

CHAPPELLG@member.ams.org

Unit Overview
Recursion & Searching

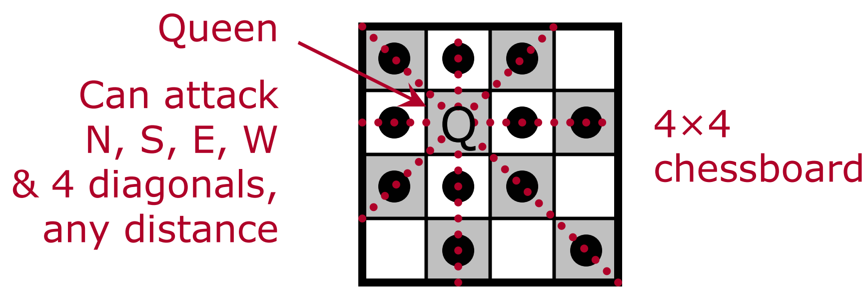
Major Topics

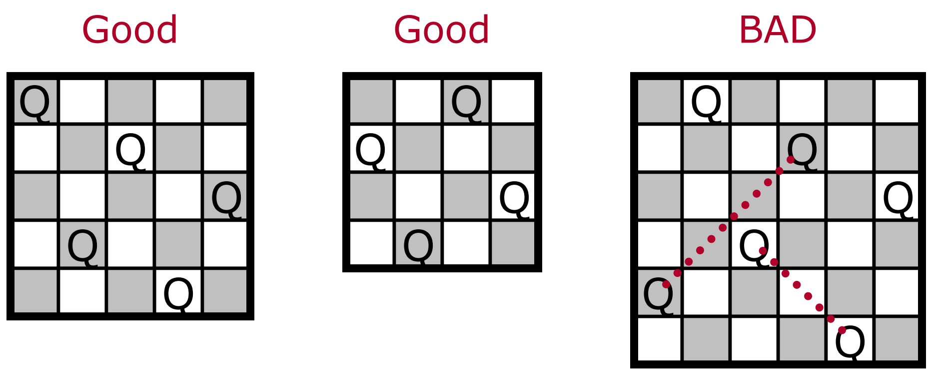
- ✓ Introduction to Recursion
- ✓ Search Algorithms
- ✓ Recursion vs. Iteration
- ✓ Eliminating Recursion
- (part) Recursive Search with Backtracking

Review
Recursive Search with Backtracking — Printing Solutions [1/3]

We looked at how to solve the ***n*-Queens Problem**.

- Place n queens on an $n \times n$ chessboard so that none of them can attack each other.

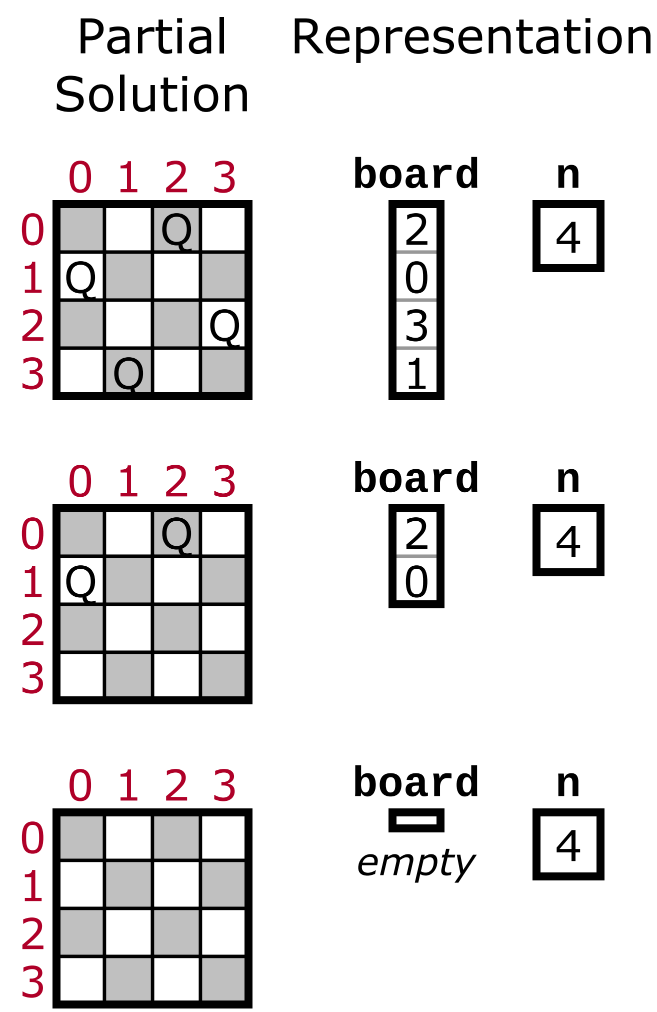




Review
Recursive Search with Backtracking — Printing Solutions [2/3]

Representing a Partial Solution

- Number rows and columns 0 .. *n*–1.
- Two variables:
  - Variable **board** (**vector** of **int**s).
  - Variable **n** (**int**).
- Variable **n** holds the number of rows/columns in a full solution.
- Variable **board** holds the columns of the queens already placed, one per row.
- The size of **board** is the number of rows in which queens have been placed.



Review
Recursive Search with Backtracking — Printing Solutions [3/3]

The Code

- **Nonrecursive wrapper function**
  - Creates an empty partial solution.
  - Calls the workhorse function with this partial solution.
- **Recursive workhorse function** is given a partial solution, prints all full solutions that can be made from it.
  - Do we have a full solution?
    - If so, output it.
  - Do we have a clear dead end?
    - If so, simply return.
  - Otherwise:
    - Make a recursive call for each way of extending the partial solution.

Note:
This part **might** not be necessary. Another way to handle dead ends is simply not to make any recursive calls when we get to this part.

TO DO

- Write a recursive function to print solutions to the *n*-Queens Problem.

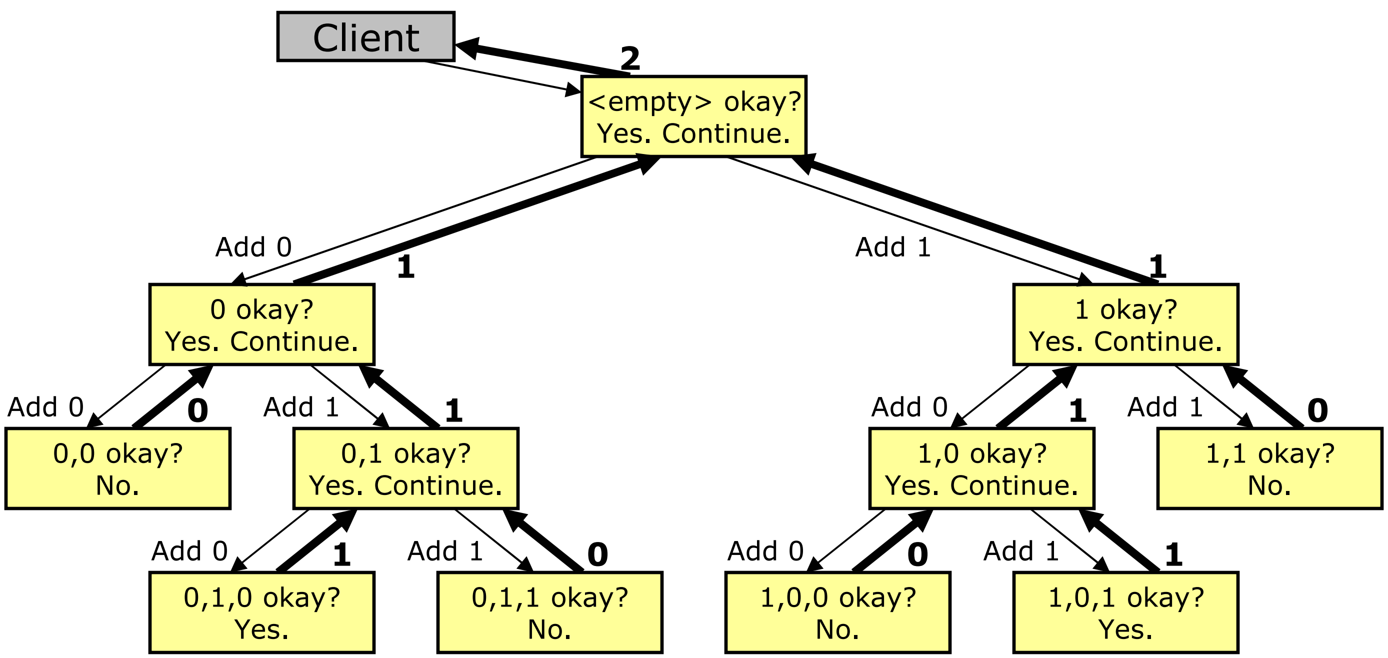
Done. See **`nqueen.cpp`**, *on the web page.*

Recursive Search with Backtracking
Counting Solutions — Diagram

continued

We can use a similar approach to **count solutions**. Each recursive call returns the number of full solutions based on the given partial solution.

- Base Cases
  - "Found a solution" returns 1.
  - "Didn't work" returns 0.
- Recursive Case
  - Make recursive calls, add their return values, and return the total.



Recursive Search with Backtracking
Counting Solutions — How to Do It

The Code

- **Nonrecursive wrapper function**
  - Creates an empty partial solution.
  - Calls the workhorse function with this partial solution.
  - Returns the return value of the workhorse function.
- **Recursive workhorse function** is given a partial solution, returns the number of full solutions that can be made from it.
  - Do we have a full solution?
    - If so, return 1.
  - Do we have a clear dead end?
    - If so, return 0.

  As before, this might be unnecessary.

  - Otherwise:
    - Set *total* to zero.
    - For each way of extending the current partial solution, make a recursive call, and add its return value to *total*.
    - Return *total*.

Recursive Search with Backtracking
Counting Solutions — Write It

TO DO

- Modify the *n*-Queens code to **count** all possible non-attacking arrangements of *n* queens, instead of printing them.

Done. See **`nqueencount.cpp`***, on the web page.*

Recursive Search with Backtracking
Holey Spider Walks — Problem Description [1/4]

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

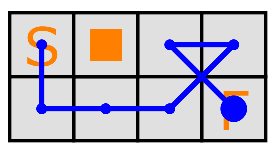
Consider a rectangle divided into squares. One square is marked as **forbidden**, another is labeled **start**, and a third is labeled **finish**. We call the result a **board**.

- An example 4 × 2 board is shown below.

What happens:

- Place a “spider” on the start square.
- The spider can step north, south, east, west, or in one of the four diagonal directions, to an adjacent square.
- We want the spider to walk thus around the board, stepping on each square except the forbidden square (the “hole”) exactly once, and ending on the finish square. A path that accomplishes this is a **holey spider walk**.

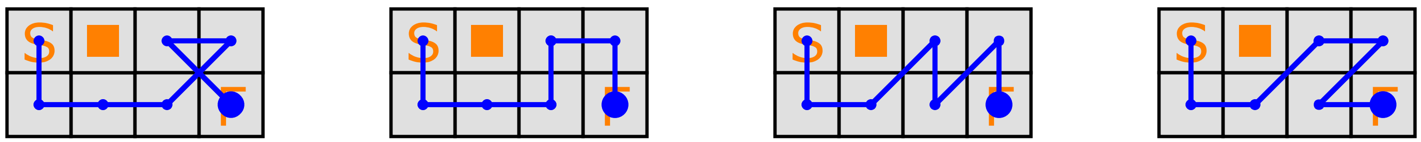
An example holey spider walk on the above board is shown below.



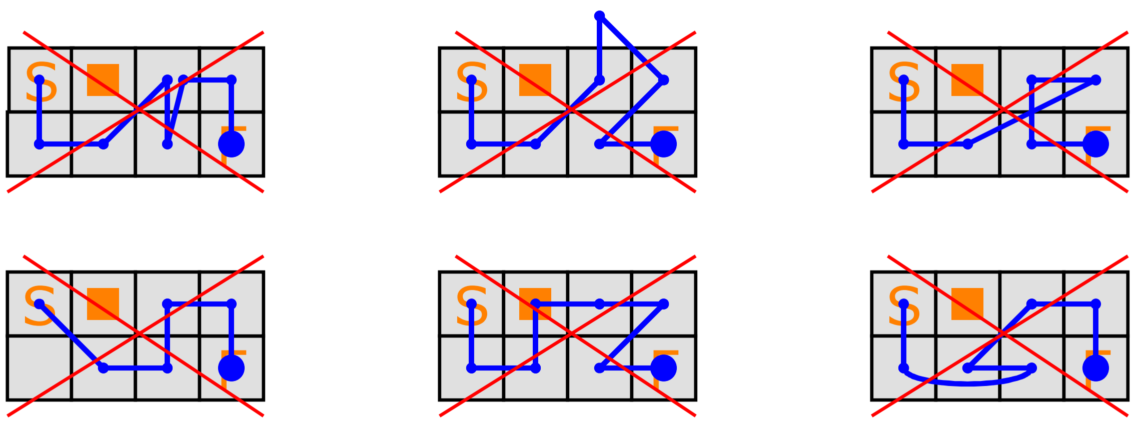
Holey Spider Walks — Problem Description [2/4]

How many different holey spider walks does this board have?

- The answer turns out to be four.



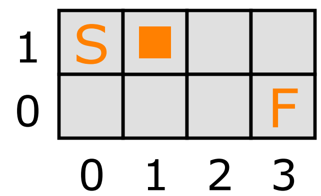
Here are some paths that are *not* valid holey spider walks.



Holey Spider Walks — Problem Description [3/4]

Place a coordinate system on a board, as shown below. Specify squares as *x*, *y*.

- Giving the horizontal coordinate first, start is (0, 1).



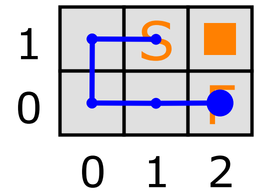
Your job in Assignment 4 is to write a function **`countHSW`** that takes a board size, forbidden square, and start and finish squares, each specified as *x*, *y*, and returns the number of holey spider walks on that board.

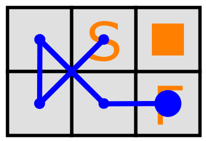
- So $\texttt{\textbf{countHSW(}}\underbrace{\texttt{\textbf{4,2}}}_{\text{size}}\texttt{\textbf{, }}\underbrace{\texttt{\textbf{1,1}}}_{\text{forbidden}}\texttt{\textbf{, }}\underbrace{\texttt{\textbf{0,1}}}_{\text{start}}\texttt{\textbf{, }}\underbrace{\texttt{\textbf{3,0}}}_{\text{finish}}\texttt{\textbf{)}}$ should return **`4`**.

Recursive Search with Backtracking
Holey Spider Walks — Problem Description [4/4]

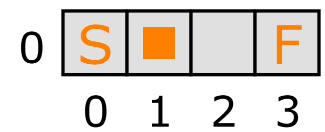
Other board sizes and forbidden/start/finish squares are possible.

- Here is a board with a different number of holey spider walks (all are shown).
  - Thus, **`countHSW(3,2, 2,1, 1,1, 2,0)`** returns **`2`**.

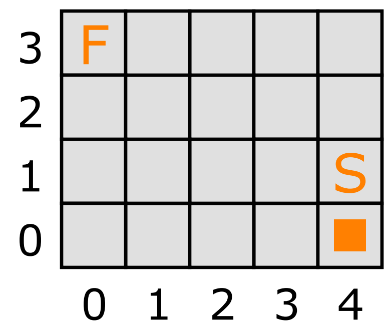




- Here is a 4 × 1 board.
  - There are no holey spider walks on this board; **`countHSW(4,1, 1,0, 0,0, 3,0)`** returns **`0`**.



- Here is a 5 × 4 board.
  - There are 40,887 holey spider walks on this board; **`countHSW(5,4, 4,0, 4,1, 0,3)`** returns **`40887`**.



Recursive Search with Backtracking
Holey Spider Walks — Writing It [1/7]

We consider how to write **`countHSW`** 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 **`countHSW`**.
- Recursive workhorse function **`countHSW_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
Holey Spider Walks — Writing It [2/7]

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

Ideas:

- A board is a vector of **`int`**s, each corresponding to a square. An item is **`1`** if the spider has visited it or it is forbidden, otherwise **`0`**. Thus: **`0`** means the square needs to be visited.

```
typedef std::vector<int> BoardType;
```

We do *not* need to remember which square is forbidden.

- We also need to know:
  - The board size: *x* & *y*.
  - The finish square: *x* & *y*.
  - The spider's current position: *x* & *y*.
- It would be helpful to know:
  - The number of squares left to visit.
    - I will call this **`squaresLeft`**.

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

Do not use global variables!

- Remember: recursion.

Recursive Search with Backtracking
Holey Spider Walks — 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. I suggest avoiding them.
- 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_x][size_y];`**"), 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_y+j`**.

```
b[i*size_y+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.

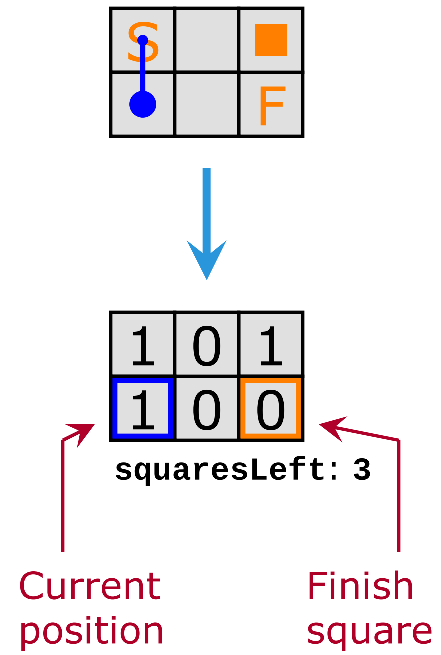
Recursive Search with Backtracking
Holey Spider Walks — Writing It [4/7]

What is a **partial solution**?

- It represents some point along the spider's journey.
- Rules. A partial solution:
  - Starts at the start square.
  - Makes legal moves (N/S/E/W/diagonal, on board, not forbidden).
  - Does not visit any square twice.

Therefore, we have a *full* solution if:

- The number of squares left to visit is zero.
- The spider is on the finish square.




```
int countHSW_recurse(BoardType board,
                     int size_x, int size_y,
                     int finish_x, int finish_y
                     int pos_x, int pos_y,
                     int squaresLeft)
{
    if (squaresLeft == 0 && pos_x == finish_x && pos_y == finish_y)
        return 1;  // We have a full solution
```

In an *empty* solution:

- The board is all **0**'s, except for the start square and the forbidden square, which are **1**.
- The spider is on the start square.
- The number of squares left to visit is the number of squares on the board minus **2**.

Recursive Search with Backtracking
Holey Spider Walks — Writing It [5/7]

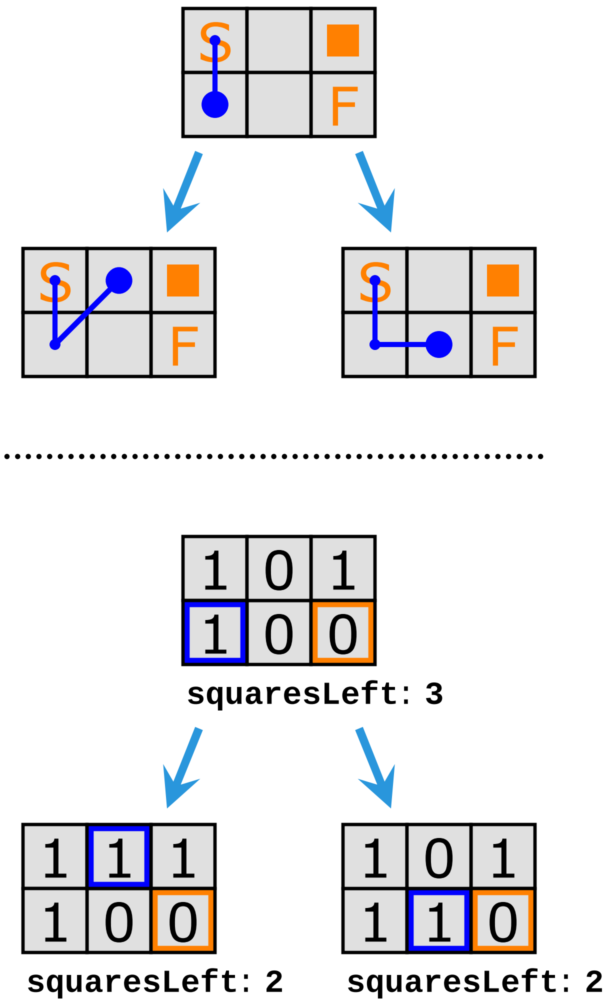
Procedure for workhorse function:

- Check for a full solution (previous slide).
  - If so, return 1.
- Set *total* to zero.
- For each of the eight squares adjacent to the spider's current position:

Do first

Second

  - Check if this (1) lies on the board and (2) is not-yet-visited.
  - If so:
    - Move current spider position.
    - Mark new square as visited.
    - Decrement number of squares left.
    - Make recursive call.
    - Add return value to *total*.
    - Restore all changes, except change to *total*.
- Return *total*.



Recursive Search with Backtracking
Holey Spider Walks — Writing It [6/7]

More Suggestions

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

```
int countHSW_recurse(BoardType & board, …
```

- If you use the 1-D array idea (recommended!) then you should still treat it like a 2-D array. Keep track of *x* and *y*, not the array index. Do not iterate through it using a single for-loop; use nested loops.
- You may find formulas for the number of walks in special situations (or in general). These are certainly of interest, but do not use them in your code; they will not help you meet the requirements of the assignment.
- Remember to subtract **`1`** from **`squaresLeft`** when making a recursive call.

Recursive Search with Backtracking
Holey Spider Walks — 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 **`countHSW`**: prototyped as required.
  - Function **`countHSW_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

We will (partly) follow the text.

- Efficiency and sorting are in Chapter 9.

After 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.

Introduction to Analysis of Algorithms
TO BE CONTINUED …

Introduction to Analysis of Algorithms will be continued next time.