Algorithms, Data Structures and Invariants

First, let’s try something on the blackboard.

How are these things related?

- Let’s start with a problem. We don’t know how to solve the problem but we do know how to solve some similar, easier problems.
- So, we transform this problem into a few easier ones.
- We solve the easier problems and then we need to combine the results so that we have a solution to the original problem.
- This is called *Reduction:*
  - Problem *A* is *reduced* to the set of problems ***B***;
  - ***B*** are solved to form solutions ***B****;
  - ***B**** are combined into the solution *A*.*

There are different ways to subdivide problem *A*

- One method is called “divide and conquer”
  - Let’s say we want to sort an array of length *N*.
  - We could divide the array into two (or more) equal-length sub-arrays and sort each of them.
  - Then we would have to merge the results back together.
- The essence of this method is that each sort is *exactly the same* except that it is smaller than the original.
  - What happens when the sub-array has length 1?

There are different ways to subdivide problem *A* (2)

- Another common method is the head/tail method:
  - Let's say we have a list of *N* elements and we want to sum them:
    - We take the head* of the list and add that to the sum of the tail of the list.
    - The sum of an empty list is 0.
  - This is also the method we use to evaluate *N!*
    - It is the product of a list of numbers 1 thru *N*.

*** A list is defined as: a head element, followed by a tail, which itself is a list**

Data structures?

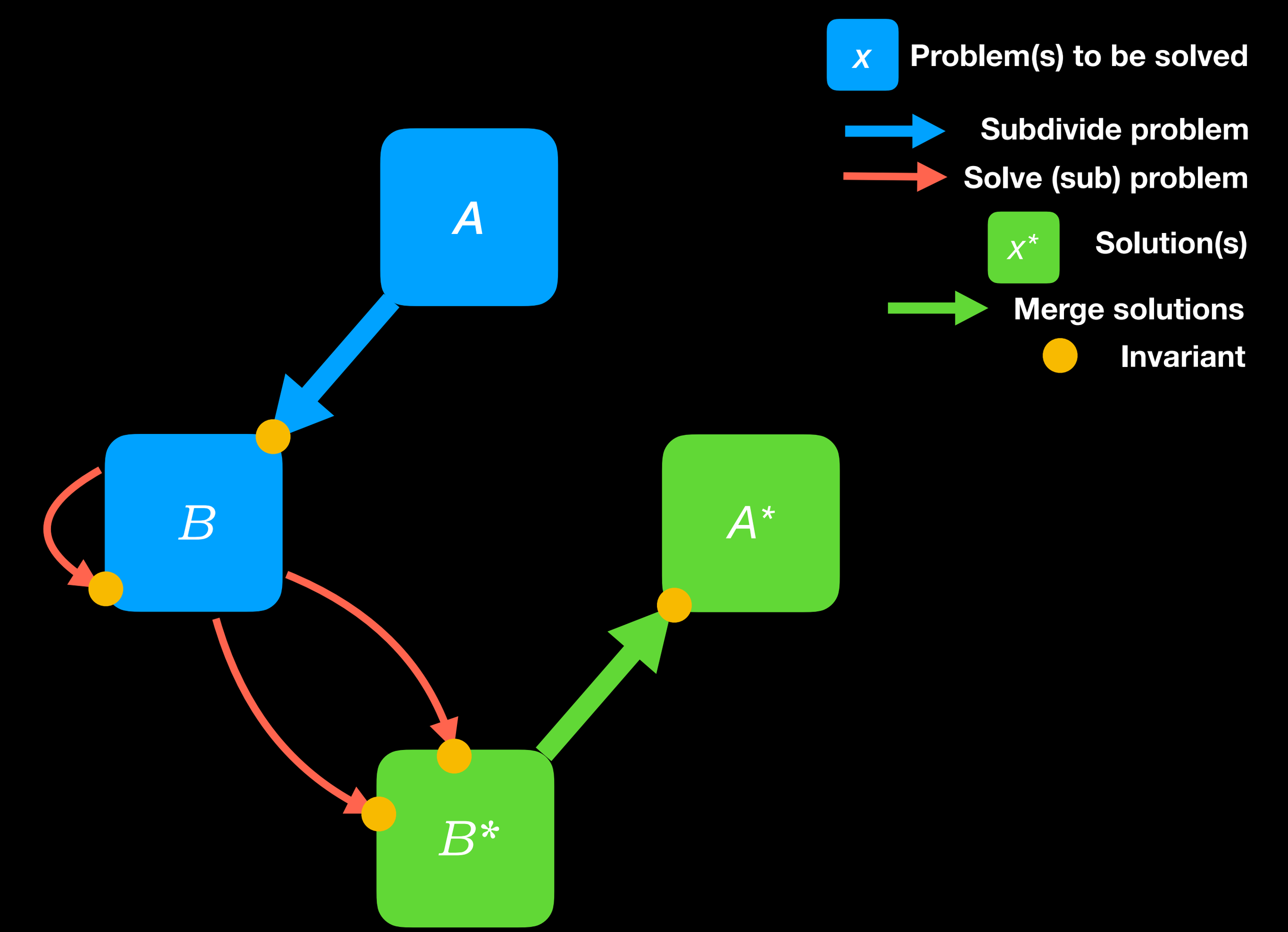
- Each of these problem solutions typically involves some data structure:
  - In the sort case, we created two (or more) sub-arrays to be sorted; *or*
  - We could use the arrays in-place provided that we stored the high and low indices of each sub-array somewhere.
  - If we use the second method, where do we typically put these indices?

The Stack

- A stack is a last-in-first-out (LIFO) data structure of infinite size (at least, in theory).
  - Not to be confused with “stack” (of layers) in the context of software application development.
- “The stack” is a free parking place for this sort of data—at least for those languages that support a stack (Java does).
  - But we could equally well create our own explicit stack for a data structure.

Invariants

- What is an invariant?
  - This is a mathematical concept regarding some property that doesn't vary—a constant, in other words.
- How does it relate to algorithms and data structures?
  - We've already seen how data structures enable a problem to be divided into smaller problems (stages);
  - Essentially, each stage takes one data structure (its problem) and transforms it into a new data structure (its solution).
  - The data structure between stages typically has some invariant property.


x
Problem(s) to be solved
Subdivide problem
Solve (sub) problem
x*
Solution(s)
Merge solutions
Invariant
A
B
A*
B*

That’s the secret to data structures and algorithms!

- You can all go home now. Good luck!

Details

- Well, maybe there are a few little details we should look into ;)

Example of reduction: The Dictionary Principle

- If there's only one thing you remember from this class, *this* should be it.

- The most important algorithmic improvement of all comes from turning an **O**(*N*) problem into an **O**(*log N*) problem.

- I call this "The Dictionary Principle," because it's the basis of how dictionaries work.

The Dictionary Principle (2)

- *How* does it work?
  - Suppose we have an array that is already sorted, i.e. in order. What advantage can that be for searching for something?
  - Well, you already know how to look up a word in a dictionary:
  - Let's say you're looking for the word *facetious*.
  - You open the dictionary at approximately the half-way point and look to see whether facetious comes before or after the words on that page.
  - If before, you look at the quarter point; if after, you look at the three-quarter point… and so on.

The Dictionary Principle (3)

- *Why* does it work?
  - Because at each iteration, we reduce the size of our problem by a factor of two. We can immediately eliminate any possibility of finding our word in the *other* half.
  - In other words, we have *reduced* our original problem to one problem—of half the size.
  - And the solution that we end up with—the definition of the word—doesn't need any further transformation back to the domain of the original problem—it is the solution to the original problem!

The Dictionary Principle (4)

- And how *well* does it work?
  - At each iteration, we reduce the size of our problem by a factor of two.
  - That means that it takes *lg N* (i.e. $log_2 N$) steps.
  - We turned a problem that, on average would have taken *N/2* steps into a problem that takes *lg N* steps.
  - For *N* = 1,000, that means it is 50 times faster!
  - For *N* = 1,000,000 that means it is 25,000 times faster!

The Dictionary Principle (5)

- But there must be a snag!
  - Well, yes, it's unlikely that our original list happens to be in order. So, we have to do work to put the list in order.
  - The amount of work that we have to do (if we use merge sort) is *N log N*. That's a lot of work!
  - But the key to the dictionary principle is:
    - Sort once—search many.

Example: 3-sum

- Find all triples of numbers (x_i, x_j, x_k) from the indexed set X where $x_i + x_j + x_k = 0$
- How do we do this? Brute force implementation is going to involve three nested *for* loops, i.e. it will be $O(N^3)$
- Is there an easy way to improve it?
  - Reduction
    - Memoization (Dynamic Programming)
- Example set: -3, -1, 1, 2, 4, 5

Improving 3-sum (problem A)

- Problem B1 is the following:
  - Given a set of elements in random order, create a table of pairs of numbers, indexed by their sum:

Sum	Pairs	
-4	-3,-1	
-2	-3,1	
-1	-3,2	
0	-1,1	
1	-3,4	-1,2
2	-3,5	
3	-1,4	1,2
4	-1,5	
5	1,4	
6	1,5	2,4
7	2,5	
9	4,5	

Improving 3-sum (problem A)

- Problem B2 is the following:
  - Given a table of pairs of numbers, indexed by their sum, find, for a value *v*, every pair p_j such that $v = -p_j$

Sum	Pairs	
-4	-3,-1	
-2	-3,1	
-1	-3,2	
0	-1,1	
1	-3,4	-1,2
2	-3,5	
3	-1,4	1,2
4	-1,5	
5	1,4	
6	1,5	2,4
7	2,5	
9	4,5	

Our reduction of A->B

- Invariant 0: we have a randomly-ordered list of *N* elements.
- Stage 1: Build the *sums* table (i.e. memoization) from the list.
  - There will of course be N^2 pairs.
  - There will be *s* keys (where *s* is the number of sums) and each will point to a list of, on average, N^2/s pairs.
  - This will take time proportional to N^2 plus *s lg s* because we will need two nested for loops to construct the pairs and because we want to sort the keys.
  - And it will require memory proportional to $s + N^2$

Our reduction of A->B

- Invariant 1: our table of sum->pairs is ordered by sum.
- Stage 2: for every element, identify pairs
  - For every element x_i in the set, get the set of pairs P_i from the table, thus forming a set of tuples: x_i -> P_i where $P_i = (x_j,x_k)$
  - This will take time proportional to $N \lg s$ where s is the number of sums
  - … because finding the key in the list requires a binary search

Our reduction of A->B

- Invariant 3: we have a set of key-value pairs (x_i,P_i) ordered as the original list.
- Transform the set of key-value pairs (x_i,P_i) into the set R where

  $R_i = (x_i, x_j, x_k)$

  - This will take time proportional to N (at worst) and is trivially easy to do.
- Invariant 4: we have a list of triples which sum to zero.

Total time for reduction?

- $N^2 \log N$ instead of N^3
  - Is that improvement really worth the trouble?
  - What if N is 1,000?
  - $N^3 = 10^9$
  - $N^2 \log N = 10^7$
- What did it cost us?
  - A bit more complexity in programming plus…
  - an algorithm to find the relevant index (and pairs) from the table without searching one-by-one.

What about space?

- Any extra space used?
  - Memory space: proportional to $N^2 + s$
  - Does this seem kind of unimportant?
  - Don’t you believe it! This is huge. Why?
  - Because in computing, we like to be able to operate on a list that uses *all* of our memory. But, clearly, we couldn’t do that with this algorithm.
  - If we had a terabyte of memory in our heap, the largest list we could process with this algorithm would be something like 1 Mbyte (depending on the constant of proportionality). That’s not even 1 million elements.

Example 2: Revisiting Merge sort

- This is the example of reduction that I discussed on the blackboard (hopefully):
  - Step 1: transform problem A (sorting an array of length *N*) into problems **B** (sorting two arrays each of length *N/2*);
  - Step 2: solve (recursively) each of the parts of problems **B** (when *N* gets below a threshold k, we use insertion sort: takes a total of $N^2/2$ time);
  - Step 3: transform the solutions to the problems **B** (i.e. two sorted sub-arrays) into the domain of problem A (by merging the two sorted sub-arrays into a sorted version of the original array [this operation takes linear time].
- We can show (later) that the entire operation takes time proportional to $N \log_2 N$.

Generalized Reduction

- We just looked at Merge Sort as a reduction. We quietly made two decisions:
  - We chose to use “head”-recursion wherein the recursive call is performed at the head of the recursion and the useful (invariant-establishing) work is applied at the tail; In the case of merge sort, this useful work is merge.
  - We chose to divide the work to be done into two more or less equal partitions.
  - Each of these two decision/properties has two possibilities.

Generalized Reduction (2)

- The type of recursion—head or tail:
  - Head:

```
recursiveFunction(x) {
 if (terminating condition) return default value;
 recursiveFunction(split(x,0));
 recursiveFunction(split(x,1));
 merge(x)
}
```

- Tail:

```
recursiveFunction(x) {
 partition(x)
 if (terminating condition) return default value;
 recursiveFunction(split(x,0));
 recursiveFunction(split(x,1));
}
```

Generalized Reduction (3)

- The type of sub-division—equal or head/tail:
  - Equal:
    - Work to be done is more or less equally divided into *r* sub-divisions (partitions).
    - Because this tends to result in $log_r N$ levels, and if each level requires linear work, the complexity is **O**($N \log_r N$)
  - Head/Tail:
    - Work is divided into a head (single element) and a tail (the remaining elements).
    - Because this tends to result in *N* levels, and if each level requires linear work, the complexity is **O**(N^2)

Sort by Reduction

- Let’s apply these generalizations to sorting.
  - Sub-division gives us the following two classes of algorithm:
    - Head/tail: “Simple sorts” (quadratic)
    - Equal: “entropy-optimal sorts” (linearithmic)
  - Recursive style gives us the following two classes of algorithm:
    - Head-recursion: post-apply invariant constraint (merge or insert)
    - Tail-recursion: pre-apply invariant constraint (partition or select).

Sort by Reduction (2)

	Tail-recursive	Head-recursive
Head-tail division	Selection sort: Move arrow by one, Select largest before recursion.	Insertion sort: Move arrow by one, Insert after recursion.
Equal division	Quick sort: Quasi-equal partition, Partition before recursion.	Merge sort: Equal partition, Merge after recursion.

Sort by Reduction (3)

- Note that all of these algorithms can be implemented iteratively, as well as recursively.

In general…

- We will use this type of technique (reduction) all the time throughout this course.
- Think of an algorithm that takes time *t* for *N* elements where:
  - $t = c\ (\log N)^q\ N^k$
- We may be able to use reduction to a problem whose solution takes time $t' = c'\ (\log N)^{q'}\ N^{k'}$ where $c' < c$ or where $q' < q$ or where $k' < k$ or…
- Perhaps we can reduce it to two (or more) problems where each problem involves a subset of *N*: this is the famous “divide and conquer” technique.