Motivation

- Algorithms are the core of many papers.
- The value is in *the problem solved*, not in the code.
- Why should the reader bother learning your Algorithm? What does "better" mean?
  - Faster: Theoretically, in practice, or both.
  - Fewer resources, such as memory, disk, code size.
  - Less error:
    - Improves the average case.
    - Improves the worst case.
    - Improves a class of cases.
  - Broader applicability.
  - Solves previously insoluble problems.
  - Enables a tradeoff between resources (e.g. time for memory)
  - Converts static to dynamic, or vice versa.
- What is the cost?

What's In It?

What should we expect to find in an algorithmic paper?

- **Steps:** Detailed steps that make up the algorithm.
- **Data:** Inputs, Outputs, and internal data structures.
- **Scope:** How is the algorithm used? Where does it apply? Is it globally applicable or limited to a class?
- **Limitations:** Where does it fail?
- **Correctness:**
  - How do we know it works? Theoretical proof? Empirical proof?
  - Can you demonstrate correctness?
- **Complexity Analysis:** Time, Space, Error.
  - At least the worst case.
  - Sometimes the average (expected) case, or distribution data.

Formalisms

Some examples follow, but the main categorizations are:

- **List code:** Numbered steps for each action, with "GOTO" statements.
  Control structure is obscure and ideas are buried.
- **Pseudo-code:** Numbered lines of a block-structured language. Detailed structure is clear but statements are terse and overall idea less obvious.
- **prose code:** Outline form: Numbered major steps with sub-numbered component steps; combined with explanatory text.
- **literate code:** Details are introduced gradually, intermingled with discussions of underlying ideas and even asymptotic analysis, proof of correctness, or details of key insights.

Do not use flowcharts or other large diagrams. No room for complexity, comments, and lack modularity.

WeightedEdit(S1, S2):

```
 1. L1 = len(S1)
 2. L2 = len(S2)
 3. M = 2 · (L1+L2)
 4. F[0,0] = 0
 5. for i from 1 to L1
 6.   F[i,0] = F[i-1,0]+M-i
 7.   for j from 1 to L2
 8.     F[0,j] = F[0,j-1]+M-j
 9.     for i from 1 to L1
10.       C = M-i
11.       for j from 1 to L1
12.       C = C-1
13.       F[i,j] = min(F[i-1, j] + C,
                       F[i,j-1] + C,
                       F[i-1,j-1]+C · isdiff(S1[i], S2[j]))
14. return(F[L1,L2])
```

1. (Set Penalty) Set $p \leftarrow 2 \cdot (k_s + k_t)$
2. (Initialize data structure) The boundaries of array F are initialized with the penalty for deletions at start of string. for example, $F_{i,0}$ is the penalty for deleting i characters from the start of s.
   (a) Set $F_{0,0} \leftarrow 0$
   (b) For each position i in s, set $F_{i,0} \leftarrow F_{i-1,0} + p - i$
   (c) For each position j in t, set $F_{0,j} \leftarrow F_{0,j-1} + p - j$
3. (Compute edit distance) For each position i in s and j in t:
   (a) The penalty is $C = p - i - j$
   (b) The cost of inserting a character into t (or equivalently, deleting from s) is $I = F_{i-1,j} + C$.
   (c) The cost of deleting a character from t is $D = F_{i,j-1} + C$.
   (d) If s_1 is identical to t_j, the replacement cost is set as $R \leftarrow F_{i-1,j-1}$, otherwise, we set $R \leftarrow F_{i-1,j-1} + C$.
   (e) Set $F_{i,j} \leftarrow min(I, D, R)$.
4. (Return) Return F_{k_s,k_l}.

The major steps of the algorithm are as follows:
1. Set the penalty.
2. Initialize the data structure, a dynamic programming array.
3. Compute the edit distance.

We now examine these steps in detail.
1. Set the penalty. The main property we require of the penalty is that costs reduce smoothly from the start to the end of the string. As we will see, the algorithm proceeds by comparing each position i in s to each position j in t. Thus a diminishing penalty can be computed with the expression $p - i - j$, where p is the maximum penalty. By setting the penalty with
(a) Set $p \leftarrow 2\text{x}(k_s + k_t)$
the minimum penalty is $p - k_s - k_t = p/2$ and the next smallest penalty is $(p/2) + 1$. This means that two errors (regardless of the positions in the strings) will outweigh one.
2. Initialize the data structure, a dynamic programming array.
...

Details. Or Not.

Algorithms are about *ideas.* **Details are distractions.**

- Your Audience: CS Journal readers or conference attendees.
- Write for those "skilled in the art". They don't need you to
  - Code a loop to add up a list of numbers,
  - Implement a binary search,
  - Write a quicksort,
  - Implement *any* textbook algorithm.
- Provide the detail needed to implement what *you* invented, *reference* work by Newton, Gauss, Djikstra, Rivest and Knuth.

Remember, you are presenting a *new insight*, not your wondrous coding style. Insights are best understood by one or two key relationships you have discovered. When the audience understands these the framework of your algorithm will fall into place. The details should address points in the implementation that require guidance *only you can provide.*

Notation

Use mathematical notation, not programming notation.

- Use positional notation: x_i, not $x[i]$. use x^n, not $x \wedge n$.
- Do not use '*' or 'x' to denote multiplication; use 'x' or '$\cdot$'.
- Avoid specific language syntax; '==', 'a=b=0', 'a++', 'a+=c', etc. Do not assume your audience programs in *C*. Do not use `for(i=0;i<n;i++)` or anything else that requires a knowledge of the syntax of a specific language or programming tool.
- Show nesting by indentation or by numbering style (as in an outline), don't use `BeginBlock`, `EndBlock`, {curly braces}, etc.
- Use mathematic shortcuts: Σ, Π, $\lceil n \rceil$, $\lfloor n \rfloor$, superscripting and subscripting in place of loops, function calls or braces.
- Good programming practice may be bad expositional practice. Variable names of one character leave no room for ambiguity; pq cannot be mistaken for $p \cdot q$.

Environment

Data Structures, I/O types, OS, Hardware

- Describe possible inputs.
- Describe possible outputs.
- If there are hardware constraints, specify them, and use realistic assumptions (current technology or likely improvements on it):
  - Memory requirements
  - Offline storage requirements (disk, non-volatile memory)
  - Communication speeds
- Provide data types when ambiguous
- Be consistent with notation and descriptions. If "`int`" and "`integer`" mean the same thing, pick one. If they do not, define the difference.
- Avoid pseudo-code for structures if possible, and use mathematical set description. e.g., "Each element is a triple, (string, length, positions), in which *positions* is a list of byte offsets."

How do you evaluate Performance?

- Formal Proof
- Mathematical Modelling
- Simulation
- Experimentation

Performance
Basis of Evaluation

- Functionality (Is your algorithm useful in more situations?)
- Speed (Theoretical or Actual? Measured how, exactly?)
- Asymptotic performance
- Typical performance
- Real data
- Synthetic data
- Compared to ...
  - Benchmark?
  - Standard Library?
  - Specific existing package or canonical algorithm?
- Be *realistic*. Those versed in the art, like your reviewers, will spot comparisons chosen to make you look good, and you may not get a second chance (e.g., submission to a conference).

What are you measuring? Is it what people care about?

- CPU time
- Memory requirements
- Disk requirements
- Memory traffic/throughput
- Disk traffic/throughput (fetch time, transfer rate)
- Network traffic/throughput
- Other measures (fewer collisions, fewer errors, more hits, more resilient, tougher to crack, etc)
- Be honest; don't leave out unflattering numbers. If you are twice as fast but use ten times as much memory, this is a tradeoff some people might want to make.

Asymptotic Complexity

How Well Does It Scale?

- $\mathcal{O}()$ notation; worst case.
- $\Omega()$ notation; best case.
- $\Theta()$ notation; bounded case.
- **A Common Abuse:** Big-O is often used to indicate complexity, as in, "comparison-based sorting takes $\mathcal{O}(n \log n)$ time"; which is clear and acceptable, even though formally we would say "comparison-based sorting has an operation complexity of $\Omega(n \log n)$".
- Be careful with ambiguous words: *"quadratic"*, *"constant"*, *"linear"*, *"logarithmic"*, *"exponential"*.

Asymptotic Complexity

Problems with Problems

- Database: Number of records. Record length too?
- An integer operation is unit cost. Even in cryptography?
- Does the dominant cost change with scale? $\mathcal{O}(n)$ disk accesses, $\mathcal{O}(n \log n)$ comparisons $= \mathcal{O}(n \log n)$ complexity. But disk takes 5,000,000 nanoseconds and comparison takes 1...
- What happens when the problem size exceeds the various cache sizes? What happens when it exceeds real memory?
- Does the limit matter? Consider floating point error; where the bound for a certain widely used class of data is hundreds of thousands of times what we see in practice. Limits are just caps and not always proportional expectation predictors.
- Are your simplifying assumptions justifiable and reliable? e.g., can you really assume the input data for a sort is randomly arranged and not already in order?

Conclusion

First: What are you selling, and why should they care?

- Describe both the algorithm and the environment it works in.
- Detail the edges of the environment.
- Choose the right formalism. When in doubt:
  - Prose code for relatively small or straightforward algorithms.
  - Literate code for complex algorithms.
- Waste as little time as possible on the obvious.
- Math notation is concise and well-defined. Use it.
- Proof of correctness, Asympotic proofs, Expected case proofs and experimental results show precisely how your algorithm is

better.

or, if they don't, then how you've wasted six months.