CSE202
3 – Dynamic Programming

TA: Joseph L.

I. Introduction

- **Key Idea**: *Iteratively* solve smaller subproblems whose answers can be reused to solve larger problems.
  - In divide-and-conquer, by comparison, we started with the original, large problem, *recursively* split it up, and recombined at the end.
- **Key Components**
  - **Base Case**: The problem in its smallest possible case - e.g., if it's a problem dealing with a string, then this might be the case with length 0 or 1.
  - **Iteration**: A general procedure for solving, e.g., a problem of size $n+1$ from problem(s) of size $\leq n$.
  - **Solution**: Having computed the solutions to all smaller subproblems, how do we derive the final solution? Often, this is just the solution to the final subproblem, but it should still be explicitly stated even if it is.
- **Proofs of Correctness**
  - As with divide-and-conquer, the structure of a dynamic programming algorithm - wherein smaller problems solve successively larger ones - lends itself well to proof by induction.
    - **Base Case**: Justify the aforementioned "Base Case" of your algorithm.
    - **Induction Hypothesis**: E.g., assume that your algorithm has correctly computed the solutions for *all* subproblems of size up to n. You may have to adapt this statement to correctly capture what information your algorithm's "Iteration" component requires.
    - **Induction Step**: Prove that if your Hypothesis holds, your algorithm's "Iteration" component will correctly solve all subproblems of the next tier.
      - For example, in many problems, your iteration step will involve taking a max or min over a set of options. You will want to prove that this set exhaustively covers all possibilities.

II. Fibonacci Numbers: Iterative or Recursive DP?

It is entirely valid to formulate a DP algorithm recursively, rather than iteratively; I, however, do not like thinking of DP as a recursive process since recursion is spatially inefficient and harder to understand than iteration. K&T [1] does a comparison for weighted interval scheduling and also concludes that the iterative view is likely simpler. Following is another problem showing both options: computing the nth Fibonacci number $fib(n)$.

- The first and second Fibonacci numbers are 0 and 1, respectively. Further Fibonacci numbers are the sums of the previous two numbers: $fib(n) = fib(n-1) + fib(n-2)$.
- There's an obvious, exponential ($O(2^n)$) algorithm for *recursively* computing $fib(n)$. As the following figure suggests, this duplication of effort is what's responsible for the outlandish runtime.

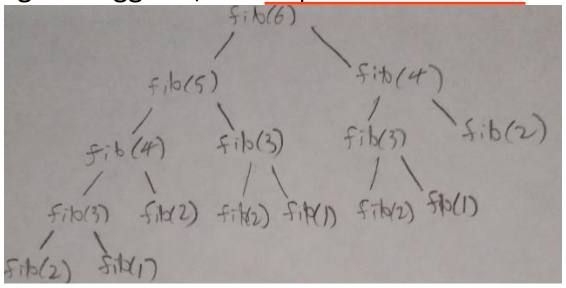
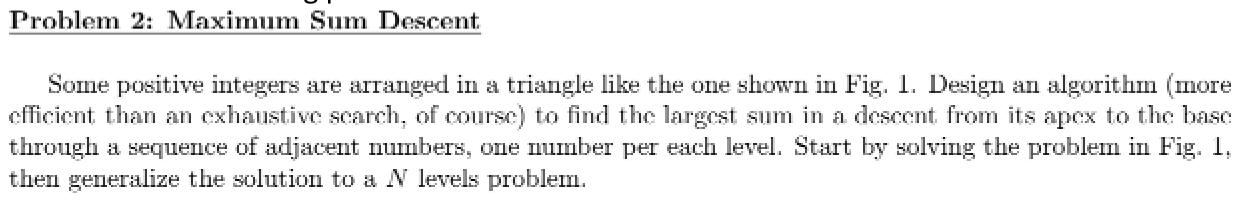


Figure 1. A recurrence tree for $fib(6)$. Even with this small value of n, we see significant duplication of effort - subtrees rooted at $fib(3)$ and $fib(4)$ appear multiple times.

- One dynamic programming solution keeps the above recursive structure and adapts it to DP:
    - Let $fib(1), fib(2)$ be base cases.
    - $fib(n)$, the root of the tree, is the solution that we want.
    - The first time $fib(i)$ is called for a given $i \in \{3,4,\dots,n-1\}$ we *memoize* (save) its solution and simply use the saved value for further calls of $fib(i)$.
    - $fib(i)$ is now explicitly computed exactly once for each value of i, and thus $fib(n)$ is now $\boldsymbol{O(n)}$.
- An iterative dynamic programming solution:
    - Let $fib(1), fib(2)$ be base cases.
    - *Iteratively* compute $fib(3), fib(4), \dots, fib(n-1)$, and then finally $fib(n)$, which we return as the solution. Again, we compute $fib(i)$ once for each $i \in \{3,4,\dots,n-1\}$, so our complexity is $O(n)$.
- Both of the above solutions are DP, but most would argue that the *iterative* view is cleaner: we don't even see the bigger subproblems until we've completed the smaller ones.
- Aside: Linear? Not quite.
    - For the call $fib(n)$, what is the size of the input? The answer is m, where $m = log_2 n$, since n is that many bits big.
    - Therefore, our $O(n)$ solution is equivalently $O(2^m)$, and thus still exponential.
        - As an aside within an aside, this is the same logic by which the knapsack problem, despite being $O(nW)$ with DP, is still considered to be exponential.
    - However, this is a significant improvement over the original, naive recursive algorithm, which is $O(2^n)$ and therefore $O\left(2^{2^m}\right)$.
    - Lastly, note that the *output* grows roughly exponentially with respect to n, and no amount of algorithmic efficiency can change that.

III. Sample Problem

We will do the following problem from the course website.



Problem 2: Maximum Sum Descent

Some positive integers are arranged in a triangle like the one shown in Fig. 1. Design an algorithm (more efficient than an exhaustive search, of course) to find the largest sum in a descent from its apex to the base through a sequence of adjacent numbers, one number per each level. Start by solving the problem in Fig. 1, then generalize the solution to a N levels problem.

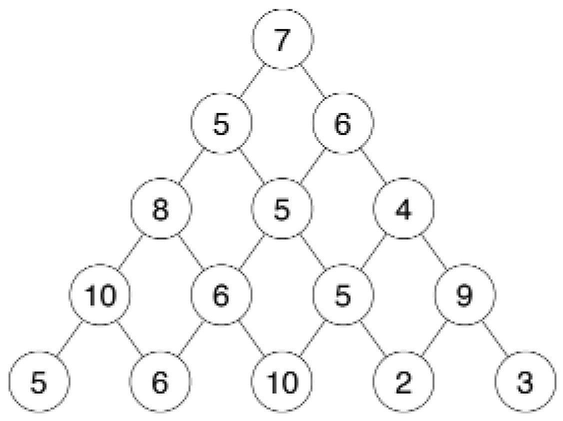


Figure 1: Example of numbers in the Maximum Sum Descent problem.

Input: A triangle N layers
Output: The maximum sum descent of the triangle

Observations: Let's list the details can be gleaned from the problem description and the figure.

- Our solution would select one node per level, and the nodes should be connected.
- Each non-leaf node has exactly two children.
  - Therefore, given an arbitrary solution that goes through a node n of layer k, there are exactly *two* possibilities for the node in layer $k-1$.
- The total number of nodes can be expressed as an arithmetic series: $1+2+...+N$. Therefore, the total number of nodes is $O(N^2)$. If we're to check all nodes, $O(N^2)$ is our best possible runtime.

The second point gives us an idea for a subproblem. We will use $k(n)$ to refer to the layer of a given node n (with the bottom being $k(n)=1$). Further, let $v(n)$ be the value of n.

Subproblem: For each node n, we will have a subproblem that computes the *largest sum* of the *sub-pyramid* for which n is the top node. We will refer to this as $P(n)$.

Base Case: For leaf nodes ($k(n)=1$), the solution is simply the value of that node.

Iteration: For a given node n, let the left and right children be l and r, respectively. Then, we can derive our update as follows:
$P(n)=v(n)+\max\{P(l),P(r)\}$
We will work upwards, one layer at a time.

Solution: Let $\hat{n}$ be the topmost node of the entire pyramid. Our solution is $P(\hat{n})$.

Complexity: There are $O(N^2)$ subproblems. Each one only does constant-time operations, and is thus $O(1)$. The total complexity is $\boldsymbol{O(N^2)}$.

Correctness
Base Case: A sub-pyramid at a leaf node is a subpyramid of height 1. It trivially has only one option for its maximum sum descent.

Induction Hypothesis: Suppose all subproblems are correctly computed up to an arbitrary layer $\hat{k}$.

Induction Step: We would like to prove that if the induction hypothesis holds, then our algorithm's "Iteration" component correctly computes the subproblems of layer $\hat{k}+1$. The following chain of logic should be sufficient:

1. Per the hypothesis, all the layers below are correct.
2. For each node in layer $\hat{k}+1$, we check both of its children. Therefore, we exhaustively check relevant subproblems.
3. We choose the child that yields a larger sum. Therefore, we choose the right option from among our enumerated options.
4. Thus, given that our hypothesis holds, our update correctly computes the next layer's solutions.

Because our base case, hypothesis, and step are correct, our algorithm is proven correct with a proof by induction.

Notes: A fairly similar DP algorithm can be built going top-down on the pyramid, rather than bottom-up. Most would likely agree that bottom-up is more intuitive for this problem.

References

1. Kleinberg, Jon, and Éva Tardos. *Algorithm Design*. Boston: Pearson/Addison-Wesley, 2006. Print.