15-750: Graduate Algorithms February 1, 2016

Lecture 7: Dynamic Programming I: Optimal BSTs

Lecturer: David Witmer *Scribes: Ellango Jothimurugesan, Ziqiang Feng*

1 Overview

The basic idea of dynamic programming (DP) is to solve a problem by breaking it up into smaller subproblems and *reusing* solutions to subproblems.

1.1 Requirements

In generally, there are several requirements to apply dynamic programming to a problem [Algorithm Design, Kleinberg and Tardos]:

1. Solution to the original problem can be computed from solutions to (independent) subproblems.
2. There are polynomial number of subproblems.
3. There is an ordering of the subproblems such that the solution to a subproblem depends only on solutions to subproblems that precede it in this order.

1.2 Dynamic Programming Steps

1. **Define subproblems.** This is the critical step. Usually the recurrence structure follows naturally after defining the subproblems.
2. **Recurrence.** Write solution to (sub)problem in forms of solutions to smaller subproblems, e., *recursion*. This will give the algorithm.
3. **Correctness.** Prove the recurrence is correct, usually by *induction*.
4. **Complexity.** Analyze the runtime complexity. Usually:

$$runtime = \#subproblems \times timeToSolveEachOne$$

But sometimes there are more clever ways.

2 Computing the Number of Binary Search Trees

Suppose we have N keys. Without loss of generality, let's assume the keys are integers $\{1, 2, 3, \ldots, N\}$. We ask the question: *How many different binary search trees (BST) can we construct for these keys?* Note that we need to maintain the property of a BST. That is, the key of a node is greater than all keys in its left sub-tree, and ditto for the right sub-tree.

2.1 Define subproblems

Definition 2.1.

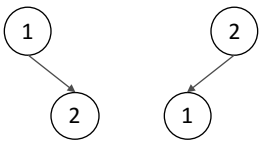
$$B(n) = \text{number of BSTs of } n\text{-nodes}$$

Examples 2.2. $B(n)$ for small n's.

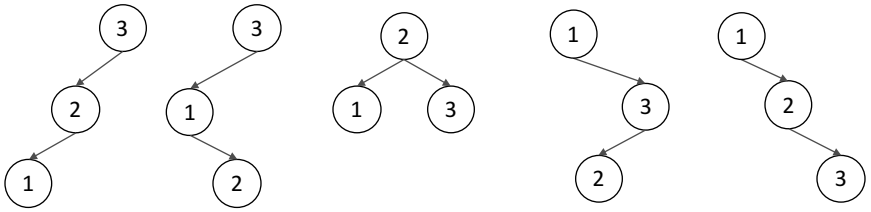
$B(0) = 1$ (the empty tree).

$B(1) = 1$ (a single node).

$B(2) = 2$



$B(3) = 5$



2.2 Recurrence

Definition 2.3.

$$f(n) = \sum_{i=0}^{n-1} f(i) \cdot f(n-i-1)$$

$$f(0) = 1, f(1) = 1, f(2) = 2$$

Note:

1. Since the function is defined in recursive form, it is necessary to give base case values (e.g., $f(0)$).

2. The base cases given here are also the values of $B(0), B(1), B(2)$.

2.3 Correctness

Claim 2.4.

$$B(n) = f(n)$$

Proof. (by induction)

Base case: It is obvious as $B(0) = f(0) = 1, B(1) = f(1) = 1, B(2) = f(2) = 2$.

Inductive case: Assume $B(n) = f(n)$ holds for all $n < m$. We will show $B(m) = f(m)$.

Let

$$S_m = \{m\text{-node BSTs}\}$$

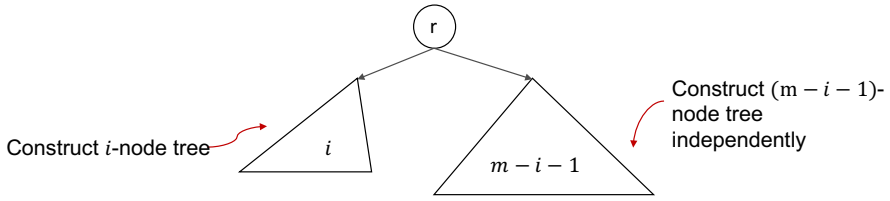
$$S_m^i = \{m\text{-node BSTs with } i \text{ nodes in its left subtree}\}$$

We can divides S_m into disjoint cases where it has a different number of nodes in its *left* subtree:

$$B(m) = |S_m| = \sum_{i=0}^{m-1} |S_m^i|$$

Note that because we are counting the number in the left sub-tree, excluding the root, the limit of sum goes only up to $m-1$. Also note that we must maintain the fundamental property of a BST. So fixing the i means fixing the root node, and the set of nodes that goes to its left or right.

Next let's consider $|S_m^i|$. An m-node BST having a i-node left sub-tree means it has a $(m-1-i)$-node right sub-tree.



Observation: If we keep the right sub-tree unchanged, and change the left-subtree (for the same i nodes), we will get a different valid m-node tree. Ditto for the right sub-tree. For example, if we have 2 possible different left sub-tree and 3 possible different right sub-tree, we can have $2 \times 3 = 6$ different trees.

$$\implies |S_m^i| = B(i) \cdot B(m-i-1)$$

$$\implies B(m) = \sum_{i=0}^{m-1} |S_m^i| = \sum_{i=0}^{m-1} B(i) \cdot B(m-i-1)$$

$$\implies B(m) = \sum_{i=0}^{m-1} f(i) \cdot f(m-i-1) = f(m) \quad \text{from inductive hypothesis.}$$

□

The Algorithm follows immediately from the recurrence structure.

Algorithm 1 Compute $B(n)$

Input: $n \in \mathbb{N}$
Output: $B(n)$
  if $n = 0$ or $n = 1$ **then**
    return 1
  else if $n = 2$ **then**
    return 2
  else
    return $\sum_{i=0}^{m-1} B(n) \cdot B(n-i-1)$
  end if

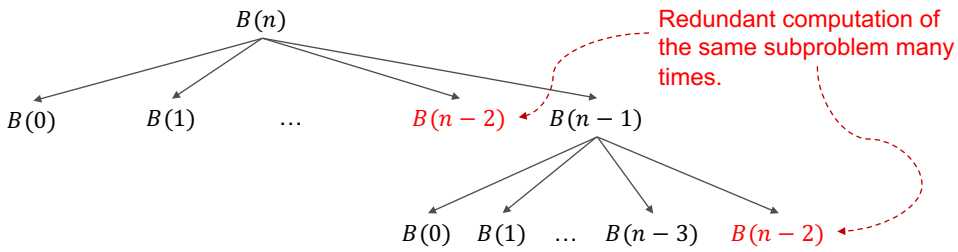
2.4 Runtime analysis

Define $T(n)$ to be the runtime of $B(n)$.

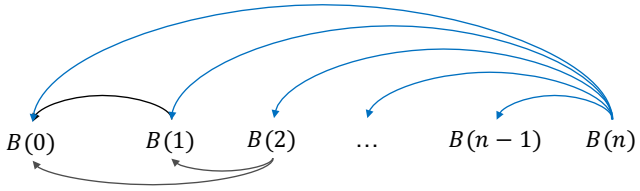
Naive analysis. In the naive implementation of the algorithm, we can analyze the runtime with the recurrence tree.

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1} T(i) + n \\ &\geq T(n-1) + T(n-2) \\ &\geq F(n) && F(n) \text{ is the Fibonacci sequence} \\ &= 2^{\Theta(n)} \end{aligned}$$

(Better) Reusing solutions to subproblems. Observe in the recurrence tree below, there are recurrent calls to the same subproblems, for example, $B(n-2)$. In the naive implementation, this solution will be computed repeatedly many times. We can improve the runtime by computing it only once, and reuse it for further calls.



Formally, we note there exist dependencies among the subproblems. $B(n)$ depends on $B(n-1), \ldots, B(1), B(0)$. And $B(2)$ depends on $B(1), B(0)$, etc. The dependency graph is typically a DAG (directed acyclic graph).



From the DAG, we can specify an order to compute the solutions to sub-problems. In this case, we compute in the order of $B(0), B(1), B(2), \cdots, B(n)$.

Runtime Let T_i be the cost the compute $B(i)$ *given* $B(0), B(1), \ldots, B(i-1)$, aka. all solutions to its dependent subproblems.

$$T_i = O(i) \quad \text{(multiplications and additions)}$$

$$\therefore T(n) = \sum_{i=1}^{n} T_i = \sum_{i=1}^{n} O(i) = O(n^2)$$

Remark 2.5. $B(n)$ is also called the Catalan number. It can be approximated by

$$B(n) \approx \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

See https://en.wikipedia.org/wiki/Catalan_number for more details.

3 Optimal Binary Search Trees

Suppose we are given a list of keys $k_1 < k_2 < \ldots < k_n$, and a list of probabilities p_i that each key will be looked up. An optimal binary search tree is a BST T that minimizes the expected search time

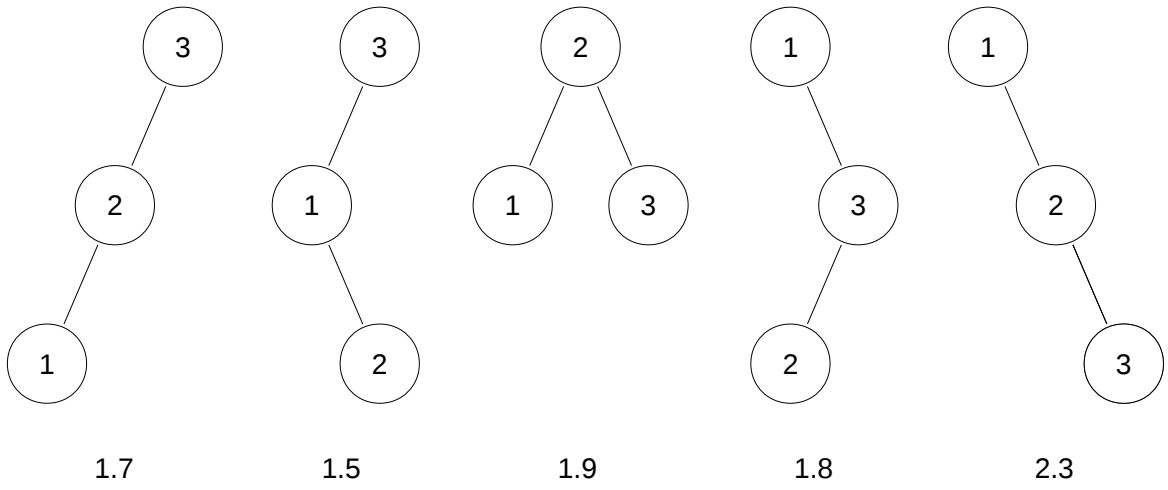
$$\sum_{i=1}^{n} p_i(\text{depth}_T(k_i) + 1).$$

where the depth of the root is 0. We will assume WLOG that the keys are the numbers $1, 2, \ldots, n$.

Example 3.1. Input:

k_i	1	2	3
p_i	0.3	0.1	0.6

For $n = 3$, there are 5 possible BSTs. The following figure enumerates them and their corresponding expected search times. The optimal BST for the given input is the second tree.



In general, there are $\Theta(4^n n^{-3/2})$ binary trees, so we cannot enumerate them all. By using dynamic programming, however, we can solve the problem efficiently.

1. **Define subproblems**

   As we will commonly do with dynamic programming, let us first focus on computing the numeric value of the expected search time for an optimal BST, and then we will consider how to modify our solution in order to find the corresponding BST.

   Let $1 \leq i \leq j \leq n$, and T be any BST on $i, \ldots, j$. We will define the cost of T

   $$C(T) = \sum_{l=i}^{j} p_l(\text{depth}_T(l) + 1)$$

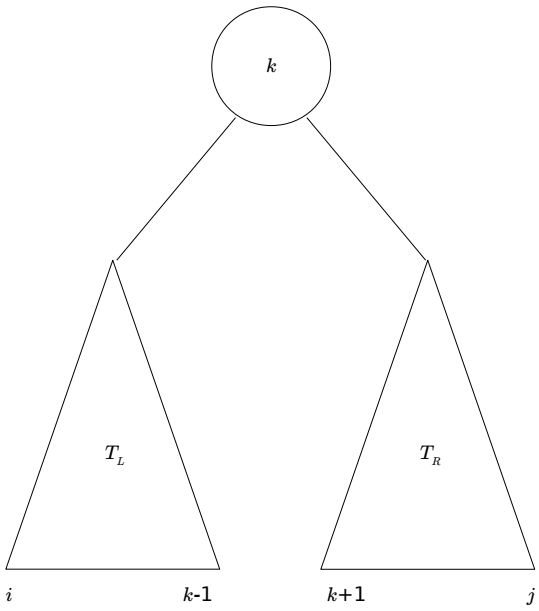
   and the subproblems

   $$C_{ij} = \min_{T \text{ on } i,\ldots,j} C(T).$$

   The expected search time for the optimal BST is C_{1n}.

2. **Recurrence relation**

   Suppose the root of T on $i, \ldots, j$ is k.



   The cost of T is

   $$\begin{aligned} C(T) &= \sum_{l=i}^{j} p_l(\text{depth}_T(l) + 1) \\ &= \sum_{l=i}^{k-1} p_l(\text{depth}_{T_L}(l) + 1 + 1) + p_k + \sum_{l=k+1}^{j} p_l(\text{depth}_{T_R}(l) + 1 + 1) \\ &= C(T_L) + C(T_R) + \sum_{l=i}^{j} p_l. \end{aligned}$$

And so we define the recurrence C'_{ij}

$$C'_{ij} = \begin{cases} \min_{i \le k \le j}\{C'_{i,k-1} + C'_{k+1,j}\} + \sum_{l=i}^{j} p_l & \text{if } i < j \\ p_i & \text{if } i = j \\ 0 & \text{if } i > j \end{cases}$$

3. **Correctness**

Claim 3.2. $C'_{ij} = C_{ij}$.

Proof. The proof is by induction on $j - i$. The base case is trivial.

$C_{ij} \le C'_{ij}$: Per the previous calculation, C'_{ij} is the cost of some BST on $i, \ldots, j$ and C_{ij} is the cost of an optimal BST.

$C_{ij} \ge C'_{ij}$: Suppose the root of the optimal BST is k. Then,

$$\begin{aligned} C_{ij} &= C_{i,k-1} + C_{k+1,j} + \sum_{l=i}^{j} p_l \\ &\ge C'_{i,k-1} + C'_{k+1,j} + \sum_{l=i}^{j} p_l \\ &\ge \min_{i \le k \le j}\{C'_{i,k-1} + C'_{k+1,j}\} + \sum_{l=i}^{j} p_l \\ &= C'_{ij}. \end{aligned}$$

□

From the recurrence, we have the following memoized algorithm.

Algorithm 2 Expected Search Time of Optimal BST

```
function C(i,j)
    if C(i, j) already computed then
        return C(i, j)
    end if
    if i > j then
        return 0
    else if i = j then
        return p_i
    else
        return min_{i≤k≤j}{C(i, k − 1) + C(k + 1, j)} + Σ_{l=i}^{j} p_l
    end if
end function
```

In order to compute the actual BST, for each subproblem we can also store the root of the corresponding subtree

$$r(i, j) = \operatorname*{argmin}_{i \le k \le j}\{C(i, k-1) + C(k+1, j)\}.$$

4. **Runtime** There are a total of n^2 subproblems, and each subproblem takes $O(n)$ time to compute, assuming all its subproblems are already solved. Thus, the total running time is $O(n^3)$.

   But, we can actually do better than this.

   Theorem 3.3 (Knuth, 1971)**.**

$$r(i, j-1) \leq r(i, j) \leq r(i+1, j).$$

   By the theorem, for each subproblem we can restrict the possible values of k to $r(i+1, j) - r(i, j-1) + 1$ choices instead of trying all $i \leq k \leq j$. For this case, the total runtime is

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=1}^{n} r(i+1,j) - r(i,j-1) + 1 &= n^2 + \sum_{i=1}^{n}\sum_{j=1}^{n} r(i+1,j) - \sum_{i=1}^{n}\sum_{j=1}^{n} r(i,j-1) \\
&= n^2 + \sum_{i=2}^{n+1}\sum_{j=1}^{n} r(i,j) - \sum_{i=1}^{n}\sum_{j=0}^{n-1} r(i,j) \\
&\leq n^2 + \sum_{i=1}^{n+1}\sum_{j=1}^{n} r(i,j) - \sum_{i=1}^{n}\sum_{j=1}^{n-1} r(i,j) \\
&= n^2 + \left(\sum_{i=1}^{n}\sum_{j=1}^{n} r(i,j) + \sum_{j=1}^{n} r(n+1,j)\right) \\
&\qquad - \left(\sum_{i=1}^{n}\sum_{j=1}^{n} r(i,j) - \sum_{i=1}^{n} r(i,n)\right) \\
&= n^2 + \sum_{j=1}^{n} r(n+1,j) + \sum_{i=1}^{n} r(i,n) \\
&= O(n^2)
\end{aligned}$$

   where we used the fact that $r(-,-) \leq n$.