Dynamic Programming
Version 1.0

1 Dynamic programming: the key idea

The solution to many problems can be written as recurrences. Often these problems are optimization problems. This usually means that an instance of the problem can be solved by combining instances of sub- problems of the original problem. In many such situations the sub- problems overlap. That is, if we unroll the entire solution to the orginal problem instance as a tree of sub-problem instances then the same sub-problem instance occurs at multiple locations in the tree (see figure 2 for an example).

Dynamic programming is an algorithmic technique that allows us to design efficient algorithms for such problems. The technique was invented by Richard Bellman [1] to solve a class of optimization problems. But in computer science we now think of it as a general technique to design algorithms for problems where the original problem instance can be solved by combining solutions of overlapping sub-problem instances.

2 Examples

We start by looking at several simple example problems.

2.1 Calculation of the n^{th} Fibonacci number

The n^{th} Fibonacci number, where n is a non-negative integer, is defined by the Fibonacci function, say F, as follows:

$$
\begin{aligned}
&F(0) = 0\\
&F(1) = 1\\
&F(n) = F(n-1) + F(n-2), \quad n > 1
\end{aligned}
$$

There are two ways (or algorithms) to calculate F:

- Calculate top-down recursively by directly using the definition of F.
- Calculate bottom up iteratively by starting with the known values $F(0)$ and $F(1)$ and successively add values of $F(n-2)$ and $F(n-1)$ to get the value for $F(n)$. So, we can add $F(0)$, $F(1)$ to get $F(2)$ then add $F(1)$, $F(2)$ to get $F(3)$ and so on.

The two methods yield algorithms with dramatically different time complexity.

Let us begin by using the top-down algorithm. Figure 1 gives the C implementation for a top-down algorithm. It directly uses the recurrence and defines a recursive function.

Let us do a sample trace of the algorithm in figure 1 for $F(5)$. To avoid writing long names we are calling the function fib instead of `fibTopDown`.

$$
\begin{aligned}
fib(5) &= fib(4) + fib(3)\\
&= (fib^1(3) + fib(2)) + fib^2(3) \quad \text{(The superscript indicates repetition. We expand } fib^1(3) \text{ first.)}\\
&= ((fib(2) + fib(1)) + fib(2)) + fib^2(3)\\
&= (((fib(1) + fib(0)) + (fib(1) + fib(0))) + fib^2(3)\\
&= \ldots + \text{the } fib^2(3) \text{ calculation will repeat } fib^1(3)
\end{aligned}
$$

Each $fib(n)$ requires two *fib* calculations ($fib(n-1)$ and $fib(n-2)$), until $n = 1$ or $n = 0$. So, we clearly observe overlap here where the same calculation is being repeated - for example $fib^1(3)$, $fib^2(3)$. It is easy to see the repetitions

```c
#include<stdio.h>
int fibTopDown(int n) {//assumes n non-negative
    int fib=0;
    if (n==0) fib=0;
    else if (n==1) fib=1;
         else fib=fibTopDown(n-1)+fibTopDown(n-2);
    return fib;
}
int main() {
    int n;
    printf("Input="); scanf("%d",&n);
    printf("Fib=%d\n",fibTopDown(n));
}
```

Figure 1: A top-down C implementation of the Fibonacci function F

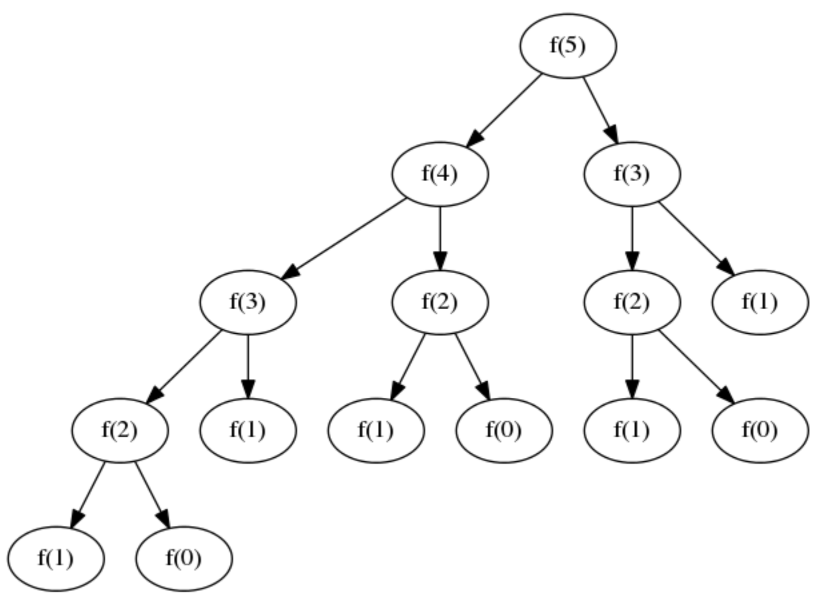


Figure 2: An unfolding of $F(5)$ as a tree.

```c
#include<stdio.h>
int fibBottomUpFn(int n,int f1,int f2) {
    //assumes n non-negative
    if (n>1) fibBottomUpFn(n-1,f2,f1+f2);
    else return f2;
}
int main() {
    int n;
    printf("Input="); scanf("%d",&n);
    printf("Fib=%d\n",fibBottomUpFn(n,0,1));
}
```

Figure 3: A C implementation of the bottom-up algorithm.

if the trace is represented as a tree (figure 2). In the tree see how $f(3)$ is computed twice, $f(2)$ thrice, and $f(1)$ 5 times.This tells us that the number of computations will be exponential in n. We can confirm this by writing the recurrence formula for the number of operations required to calculate $F(n)$. Let $T(n)$ be that number. Then assuming $T(0) = 1,\ T(1) = 1$ the recurrence for the top down program is:

$$\begin{aligned}
T(n) &= T(n-1) + T(n-2) + O(1) \\
&> 2T(n-2) \\
&> 4T(n-4) \\
&> \dots \\
&> 2^{\lfloor \frac{n}{2} \rfloor} T(1)
\end{aligned}$$

So, we get $T(n) \in \Omega(2^{\frac{n}{2}})$ that is time complexity is at least exponential in n.

Let us now do a bottom-up calculation. Figure 3 shows a C implementation of the bottom-up algorithm. Though the function looks like a recursive function it is actually iterating since the recursive call is a terminal call - that is there are no executable statements after the recursive call. This is called *tail recursion*.

Below we trace the bottom-up algorithm for $F(5)$. We use the name $Fibfn$ instead of the longer $fibBottomUpFn$ in the trace:

$$\begin{aligned}
F(5) &= Fibfn(5,0,1) \quad \text{Initial call to } Fibfn \text{ - arguments are } n\text{,0,1} \\
&= Fibfn(4,1,1) \\
&= Fibfn(3,1,2) \\
&= Fibfn(2,2,3) \\
&= Fibfn(1,3,5) \\
&= 5
\end{aligned}$$

It is clear that the number of calls to $Fibfn$ is n and the complexity is $O(n)$. If we write a recurrence we get: $T(n) = T(n-1) + O(1)$. This recurrence can be easily solved by the iteration method to get $T(n) \in \Theta(n)$. So, the bottom-up program complexity is $\Theta(n)$. We see a dramatic difference in the time complexity of the two algorithms. The difference in complexity arises from number of overlapping computations In the top-down algorithm each overlapping computation is calculated afresh. The dynamic programming technique reduces the complexity by either avoiding overlapping computations (bottom-up approach) or storing the values of already calculated sub-problems thereby avoiding recalculations.

2.2 Coin problems

We now look at some coin problems as further examples of the dynamic programming technique.

```
#include<stdio.h>
void coinChoose(int a[],int c[],int i){
//This is the backtrack for finding which coins are chosen.
    if (i>1)
        if (a[i]==(c[i-1]+a[i-2])) {coinChoose(a,c,i-2); printf("%d ",c[i-1]);}
        else coinChoose(a,c,i-1);
    else if (i==1) printf("%d ",c[0]);
}
void coinfn(int n, int c[]) {
//Finds the maximum value.
    int a[51],i;
    a[0]=0; a[1]=c[0];
    for(i=2;i<=n;i++)
        a[i]=max(a[i-1],c[i-1]+a[i-2]);//$c$ starts from c[0]
    printf("Coins chosen:"); coinChoose(a,c,n);
    printf("\nMax value=%d\n",a[n]);
}

int max(int a,int b) {return (a>b)?a:b;}
int main() {
    int n,i,c[50];
    printf("No. of coins (< 51)=");
    scanf("%d",&n); printf("\nGive value of each coin in the sequence\n");
    for(i=0;i<n;i++) scanf("%d",&c[i]);
    coinfn(n,c);
}
```

Figure 4: A C implementation of the algorithm for the coin game including the backtrack to find the coins.

Consider the following coin game. Assume we have a bag of coins of varying denominations (multiple coins of the same denomination can be present). n of these coins are arranged in a sequence. A player has to pick coins such that the total value of the picked coins is a maximum with the constraint that no two coins that are picked are adjacent to each other in the sequence. So, if c_i is picked then c_{i-1} and c_{i+1} cannot be picked except when $i = 1$ (c_{i-1} does not exist) or $i = n$ (c_{i+1} does not exist).

Let the coin sequence be $c_1, \ldots, c_n$. Denote by $A(n)$ the maximum value of the coins that can be picked by a player for the given sequence of n coins. We can write $A(n)$ as a recurrence by arguing as follows. The picked coins will either include the last coin c_n or not. If c_n is picked then $A(n) = c_n + A(n-2)$ and if it is not picked then $A(n) = A(n-1)$. Since $A(n)$ represents the maximum value we get the recurrence $A(n) = \max(A(n-1), c_n + A(n-2))$ for $n > 1$ with $A(1) = c_1$ and $A(0) = 0$. We can use the bottom-up approach to efficiently implement it starting from $A(0)$ and $A(1)$ and using the recurrence for the iteration from 2 to n. Figure 4 gives a C implementation of the coin game. It finds the maximum value and also gives the actual coins that are picked up by doing a backtrack over the recurrence. From the code both the time and space complexity of the algorithm in figure 4 is clearly $\Theta(n)$.

Let us now consider another related problem important in the context of the sudden demonetization and shortage of notes - how to pay a particular amount using the minimum number of notes/coins. For simplicity we assume that we have unlimited number of notes/coins of each type. Let the types we have be: $t_1, \ldots, t_n$, that is n types of notes/coins. Currently, in the Indian context $n = 9$ and the types are $1 < 2 < 5 < 10 < 20 < 50 < 100 < 500 < 2000$. Normally, t_1 is asssumed to be 1. We derive a recurrence for an amount x that requires the smallest number of notes/coins. As in the earlier coin game let $a(x)$ be the minimum number of notes/coins required that add up to x. If we choose a particular denomination say t_i such that $x \geq t_i$ then whenever $x > 0$ we get the following recurrence $a(x) = (\min_{i \in 1..n \ni x \geq t_i} a(x - t_i)) + 1$ with $a(0) = 0$ (read $\ni$ as such that).

To implement it assume that we have a memory object called *mem* that stores associations. Each association is the pair (x, m) where m is the minimum number of notes/coins to make up amount x. We assume that *mem* is initialized by putting in associations for $t[1], \ldots, t[n]$. Each of these is associated with the value 1 since we have notes of these denominations. Let $changeFn(x)$ give the minimum number of notes/coins for amount x. To get the value of $changeFn(x)$ we calculate $changeFn(x - t[1]) + 1$ to $changeFn(x - t[n]) + 1$ and find the value that is the minimum among all n values. We do this recursively and store the minimum value in *mem* every time to avoid recalculation. This is a strategy that is widely adopted when the dynamic programming technique is used. Calculating bottom-up will always give more efficient solutions than applying the recurrence top-down but in many cases only some sub-problems need to be solved to solve a particular instance. The bottom-up calculation ends up solving all sub-problems a majority of which may never be needed. So, it makes sense to solve the recurrence top-down but cache sub-problem answers so that they can be looked up and not re-solved. The complexity of the top-down method is high because each sub-problem is re-solved whenever it is encountered. By caching answers to sub-problems the repeated calculation is avoided. This overall approach is called *memoization* and allows us to write simple recursive code for the top-down method without paying a performance penalty. This is also an example of the space-time trade off strategy. By using extra space we can reduce time. We will see this again in later applications of the dynamic programming technique.

Figure 5 gives the C implementation for the above algorithm where *mem* has been simulated by a C array called *mem* with x as an index and the minimum number of notes/coins as value. The rest of the implementation is as described above. The function *changeCoins* in the code finds the actual coins/notes that will make up the amount.

The complexity of the 'change' problem is clearly $O(xn)$. And since we are using the array *mem* the space complexity is $\Theta(x)$.

We look at a third and the last coin problem. Assume there is board with $m \times n$ cells. A cell may be free or contain a coin. A coin picker begins in the topmost, leftmost cell and traverses to the bottommost, rightmost cell which is the end point. The picker has only two moves either one step right or one step down. If the picker visits a cell with a coin the coin must be picked up. We have to devise an algorithm so that the picker picks up the maximum number of coins on reaching the end point. The algorithm should also output the path followed by the picker.

Once again we can write a recurrence for the problem. Assume the cells are numbered like a matrix. Let $a[i, j]$ be the maximum coins the picker can pick while reaching cell $[i, j]$. Given the moves the picker can make s/he can reach $[i, j]$ either by moving right that is from $[i, j-1]$ or moving down that is from $[i-1, j]$. This gives us the following recurrence:

$$
\begin{aligned}
a[i,j] &= \max\{a[i-1,j], a[i,j-1]\} + c_{ij}, \quad 1 \le i \le m, 1 \le j \le n \\
a[0,j] &= 0, \quad 1 \le j \le n \\
a[i,0] &= 0, \quad 1 \le i \le m
\end{aligned}
$$

$c[i, j] = 1$ if cell $[i, j]$ has a coin else it is 0.

The algorithm to calculate $a[m, n]$ uses the dynamic programming technique to fill the $m \times n$ table of values. The filling can be done either row or column wise. Figure 6 gives a C implementation of the algorithm. Finding the path (all paths) that the picker takes (or can take) is left as an exercise. The time and space complexity of the picker problem is clearly $\Theta(mn)$. We are filling up a table of size $m \times n$ so both time and space complexity is bounded both above and below by some multiples of mn.

Exercise 2.1. Write a function that will backtrack over the recurrence and find the actual path taken by the robot once the $m \times n$ tabled is filled up.

Bellman proposed the *optimality principle* [1] which is a feature of problems where we can apply the dynamic programming technique. The optimality principle states that a solution to an optimization problem instance can be obtained by combining (or extending) the optimal solutions of sub-problem instances of the original problem instance. We see this principle in operation in all our coin/notes related problems.

```
#include<stdio.h>
int changeFn(int x,int mem[],int t[],int n) {
/*Finds minimum number of coins/notes that add to x*/
    int tmp=mem[x],cmin,j;
    if (tmp>=0) cmin=tmp;
    else {
        cmin=x+1;j=0;
        while (j<n && x>=t[j]) {
            tmp=changeFn(x-t[j],mem,t,n)+1;
            cmin=min(tmp,cmin);
            j=j+1;
        }
        mem[x]=cmin;
    }
    return mem[x];
}
void changeCoins(int x,int mem[],int t[],int n) {
  int j=mem[x],i,tmp;
  while (j>1) {
    /*Find coin i s.t. mem[x-t[i]]==j-1*/
    i=0;
    while (i<n && x>=t[i]) {
      tmp=mem[x-t[i]];
      if ((tmp>0) && (tmp==(j-1))){printf("%d␣",t[i]); x=x-t[i]; break;}
      i=i+1;
    }
    j=j-1;
  }
  printf("%d\n",x);/*j==1 means x is equal to one of the notes/coins*/
  return;
}

int min(int a,int b) {return(a<b?a:b);}

int main() {
    int x,i,mem[5000],n=9,t[9];

    /*Initialize t (Indian currency) and mem*/
    t[0]=1;t[1]=2;t[2]=5;t[3]=10;t[4]=20;t[5]=50;t[6]=100;t[7]=500;t[8]=2000;
    for(i=0;i<5000;i++) mem[i]=-1;
    mem[0]=0;mem[1]=1;mem[2]=1;mem[5]=1;mem[10]=1;mem[20]=1;
    mem[50]=1;mem[100]=1;mem[500]=1;mem[2000]=1;
    printf("Give␣the␣amount(<5000)="); scanf("%d",&x);
    if (x>4999) {printf("Sorry␣%d␣is␣not␣less␣than␣5000\n",x); return 0;}
    printf("No.␣of␣coins/notes=%d\n",changeFn(x,mem,t,n));
    printf("Coins/Notes␣making␣up␣amount␣%d=",x);
    changeCoins(x,mem,t,n);
    return 0;
}
```

Figure 5: A C implementation of the algorithm for finding minimum number of notes/coins to make up a given amount x.

```
#include<stdio.h>

int pickFn(int m,int n,int b[][100],int a[][100]) {
    int i,j,k;
//Fill table row-wise
    for(i=1;i<=m;i++)
        for(j=1;j<=n;j++)
            a[i][j]=max(a[i-1][j],a[i][j-1])+b[i][j];
    return a[m][n];
}

int max(int a,int b) {return(a>b?a:b);}

int main() {
    int m,n,b[100][100],a[100][100],i,j;//b is the board
    printf("Give␣no.␣of␣rows␣and␣columns␣(<␣(100x100))=");
    scanf("%d%d",&m,&n);
    printf("Indicate␣presence␣of␣coins␣row-wise,␣1-present,0-absent\n");
    //Read in board b
    for(i=1;i<=m;i++)
        for(j=1;j<=n;j++)
            scanf("%d",&b[i][j]);
    //Initialize a
    for(i=0;i<m;i++)
        for(j=0;j<n;j++)
            a[i][j]=0;
    printf("Maximum␣no.␣of␣coins=%d\n",pickFn(m,n,b,a));
}
```

Figure 6: A C implementation of the algorithm for finding maximum number of coins on an $m \times n$ board.

3 The Simple knapsack problem(SKP)

The simple knapsack problem is the problem of packing a knapsack of capacity C with a subset of n items or objects that have capacities $c_1, \ldots, c_n$, $c_i \in \mathbb{N}$, and values $v_1, \ldots, v_n$, $v_i \in \mathbb{R}^+$, respectively such that the total value of the subset of items in the knapsack is a maximum. Let us denote the solution(s) to such a problem by $k(n, C)$.

A trivial and very inefficient algorithm is to enumerate all possible subsets of the n items; find all those subsets whose capacity adds up to at most C and then choose those subsets that have the maximum value. This algorithm clearly has complexity $\Omega(2^n)$ since we are creating all possible subsets of n items.

To apply the dynamic programming technique we need to set up a recurrence. Let us consider the first i items of the set of n items and a knapsack of capacity $j \leq C$ then $k(i, j)$ is the optimal solution to the above sub problem of the original knapsack problem. To set up the recurrence divide the subsets of the first i items that fit into the knapsack of capacity j into those that include item i and those that do not. We can argue as follows:

1. Consider subsets that contain item i and are optimal. We can break such a solution into an optimal solution for a subset with $(i-1)$ items and the i^{th} item giving $k(i, j) = k(i-1, j-c_i) + v_i$.
2. For subsets not containing item i the optimal subsets are given by $k(i-1, j)$.

The solution $k(i, j)$ will be the maximum of the above two options whenever $j - c_i \geq 0$. If $j - c_i < 0$ then clearly $k(i, j)$ is the same as $k(i-1, j)$. This allows us to write the following recurrence:

$$k(i,j) = \begin{cases} \max(k(i-1, j-c_i) + v_i, k(i-1, j)), & j - c_i \geq 0 \\ k(i-1, j), & j - c_i < 0 \end{cases}$$

The base cases are: $k(0, j) = 0$, $j \geq 0$ and $k(i, 0) = 0$, $i \geq 0$.

Figure 7 gives a C-implementation for the simple knapsack problem using dynamic programming with memoization. To find the items that go into the knapsack we have to trace back from the final entry in the $k(i, j)$ array. This is another very standard and useful technique closely linked with the dynamic programming method. The basic recurrence gives the optimal solution value. To get the details of the optimal solution (for example the items in the knapsack in this case) one has to trace back from the optimal value using the recurrence in reverse to access the details.

Exercise 3.1. Do a C implementation of the trace back for the knapsack problem to give the details of the items that went into the knapsack.

3.1 Optimal binary search tree

A binary search tree (BST) is a popular search data structure where data size (that is number of keys) is medium to large.

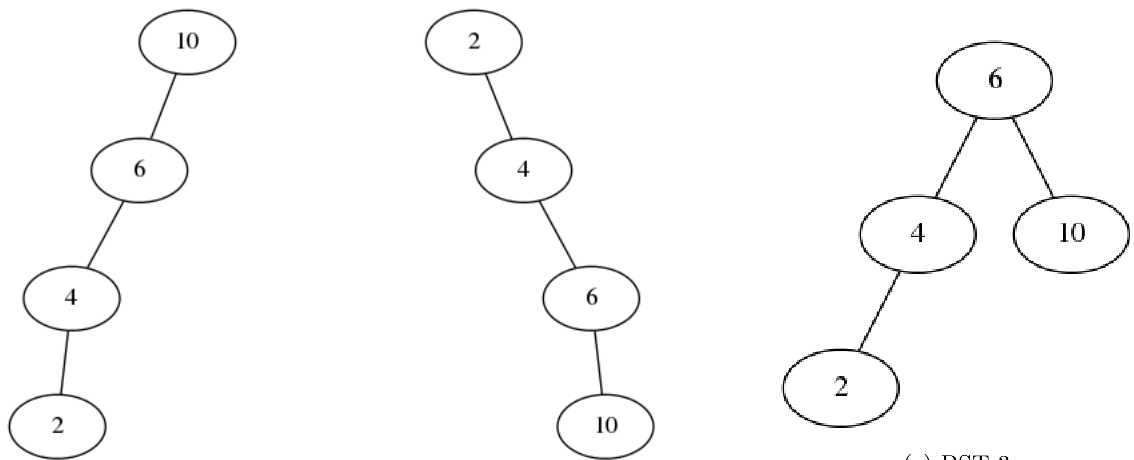
A BST T is either empty or is a binary tree with a parent or root node with a left sub-tree (L) and right-subtree (R) both of which are also binary search trees. The key value of the root of the left sub-tree is less than the parent/root key value ($L.key < T.key$) and the key value of the root of right sub-tree is greater than or equal to the parent or root key value ($R.key \geq T.key$). See figure 8 for some example BSTs with key values: 2, 4, 6, 10.

Now consider the problem of building a binary search tree for the key values in figure 8 where the frequencies with which different keys are searched are as follows: $(2, 0.1), (4, 0.3), (6, 0.4), (10, 0.2)$. Then the expected number of comparisons for the three trees shown in figure 8 will be:

$$\text{BST-1} = 4 \times 0.1 + 3 \times 0.3 + 2 \times 0.4 + 1 \times 0.2 = 2.3$$
$$\text{BST-2} = 1 \times 0.1 + 2 \times 0.3 + 3 \times 0.4 + 4 \times 0.2 = 2.7$$
$$\text{BST-3} = 3 \times 0.1 + 2 \times 0.3 + 1 \times 0.4 + 2 \times 0.2 = 1.7$$

```
#include<stdio.h>

int c[100];//capacities of items, >0
float v[100];//value of items, >0
float k[100][200];//optimal soln for knapsack[i,j] - also the memory

float maxf(float a,float b){return (a>b?a:b);}

float skpFn(int i, int j) {
    float val;
//implement the recurrence directly, array k is the memory
    if (k[i][j]<0) {//this sub-problem not solved yet
        if (c[i]>j) val=skpFn(i-1,j);
        else val=maxf(skpFn(i-1,j-c[i])+v[i], skpFn(i-1,j));
        k[i][j]=val;
    }
    return k[i][j];
}

int main(){
    //initialize k
    int n;//no of items
    int C;//capacity
    int i,j;
    for(i=0;i<100;i++) k[i][0]=0;//base case
    for(i=0;i<200;i++) k[0][i]=0;//base case
    for(i=1;i<100;i++)
        for(j=1;j<200;j++)
            k[i][j]=-1;//-1 indicates not solved yet
    printf("Give␣no.␣of␣items="); scanf("%d",&n);
    printf("Give␣capacities␣of␣each␣item=");
    for(i=1;i<=n;i++) scanf("%d",&c[i]);
    printf("Give␣corresp.␣value␣of␣each␣item=");
    for(i=1;i<=n;i++) scanf("%f",&v[i]);
    printf("Give␣capacity␣of␣knapsack="); scanf("%d",&C);
    printf("Knapsack␣soln=%f\n",skpFn(n,C));
}
```

Figure 7: A C implementation of the algorithm for the knapsack problem.



(a) BST-1 (b) BST-2 (c) BST-3

Figure 8: Three example BSTs for key values 2, 4, 6, 10.

The question we want to ask is: which BST will be optimal (i.e. has the least expected number of comparisons)? The naive algorithm that calculates the expected number of comparisons for all possible BSTs with n keys (symbolized by $BST(n)$) and chooses the minimum is hopelessly inefficient. The number of possible BSTs with n keys is given by the n^{th} *Catalan* number: $BST(n) = \frac{1}{n+1}\binom{2n}{n}$. For our example this is: $\frac{1}{5}\frac{8!}{4! \times 4!} = 14$. $BST(n)$ scales exponentially with n (see exercise 3.2) and for large values of n the naive algorithm is infeasible.

Exercise 3.2. Show that the rate of growth of $BST(n)$ dominates $\frac{4^n}{n^{1.5}}$ for large n.

The above problem is called the *static* optimal binary search tree problem because we assume that the search is done in a fixed or static tree and there are no *insert* or *delete* operations on the tree. If *insert*, *delete* operations are allowed then we get the *dynamic* version of the problem. Also, in our example problem we assumed that all searches were for keys that were present in the BST. In the more general case a search key may not exist in the tree and the number of comparisons then depends on the actual value of the key. One possibility for the more general case is to assume that for any absent key the number of comparisons is the maximum possible (gives an approximation to the optimal tree). Another is to estimate frequencies of absent keys that lie between two consecutive keys (assuming keys are sorted). This will give us the actual optimal tree in the general case.

Let the distinct keys be $k_1, \ldots, k_n$, ordered in ascending order, and let the corresponding search probabilities be $p_1, \ldots, p_n$. If T is the optimal BST and each key k_i occurs at level l_i in T (where the root is at level 1) then the expected number of searches in BST T is: $s = \sum_{i=1}^{n} p_i l_i$. If s_l, s_r are the expected number of searches for the left and right sub-trees (say T_l, T_r) respectively then we can write: $s = s_l + s_r + \sum_{i=1}^{n} 1 \times p_i = s_l + s_r + 1$. Note that the level for any key in T_l or T_r is one less than in T. If T is optimal then BSTs T_l and T_r are also optimal BSTs.

From the above observation we can now construct a recurrence as follows. Let us consider only a sub-sequence of keys k_i to k_j, $j > i$ from the original ordered sequence $k_1, \ldots, k_n$. Let s_{ij} be the expected number of searches in the optimal BST T_{ij} for this set of keys. To find T_{ij} we have to search for the best BST with root k_r where $i \leq r \leq j$, that is k_r is a key between k_i and k_j (both keys inclusive). The root k_r will have two optimal BST sub-trees $T_{i(r-1)}$ on the left and $T_{(r+1)j}$ on the right that are recursively constructed in the same way. Figure 9 gives a pictorial view to set up the recurrence.

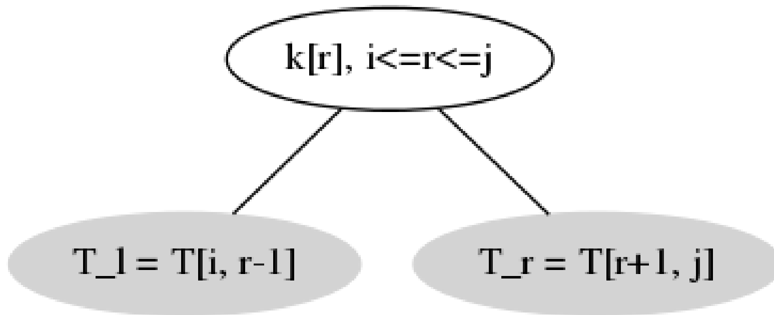


Figure 9: BST for setting up recurrence.

Now we can write the recurrence as follows:

$$
\begin{aligned}
s_{ij} &= \min_{r \in i \text{ to } j} (p_r \times 1 + \sum_{m=i}^{r-1} p_m (l_m^{T_{i(r-1)}} + 1) + \sum_{m=r+1}^{j} p_m (l_m^{T_{(r+1)j}} + 1)) \\
&= \min_{r \in i \text{ to } j} (\sum_{m=i}^{j} p_m + \sum_{m=i}^{r-1} p_m l_m^{T_{i(r-1)}} + \sum_{m=r+1}^{j} p_m l_m^{T_{(r+1)j}}) \\
&= \min_{r \in i \text{ to } j} (\sum_{m=i}^{j} p_m + s_{i(r-1)} + s_{(r+1)j}) \\
&= \sum_{m=i}^{j} p_m + \min_{r \in i \text{ to } j} (s_{i(r-1)} + s_{(r+1)j})
\end{aligned}
$$

Note that the minimization is over the choice of root k_r, where $r \leq j$.

The base cases are:

$s_{ii} = p_i$, $\forall i$ - we have just a single node k_i as the BST.
sij, $j < i$=0 - the only one we need is when $j = i - 1$ so $i = 1$ to $n + 1$.

The recurrence has been converted to an algorithm in Algorithm 1. The algorithm calculates s_{ij} bottom-up starting with the diagonal s_{ii} and going up to calculate s_{1n} which is the required value for the BST. If we want the actual optimal BST then we need a second array similar to $s[.,.]$, say $root[.,.]$, that stores the root key k_r for each sub-tree.

Exercise 3.3. Do a C implementation of the algorithm in Algorithm 1 and simultaneously find the optimal BST using the *root* array.

The algorithm to find the optimal BST has a time complexity of $O(n^3)$ and space complexity of $O(n^2)$. Time complexity can be improved to $O(n^2)$.

3.2 Transitive closure and all pairs shortest paths

We are given a connected directed graph (digraph) and we wish to know whether any two nodes in the graph are connected by a non-trivial (that is path of length greather than 0) directed path. An example of such a graph is given in figure 10a. We observe that nodes `1, 2, 3,` can reach all nodes by non-trivial paths but node `4` is unable to reach any other node. Figure 11b is a representation of the digraph using an adjacency matrix. This is a $n \times n$ matrix $A = [a_{ij}]$, where n is the number of vertices or nodes in the graph. If $a_{ij} = 1$ then there is a directed edge from node i to node j else $a_{ij} = 0$.

Input : Vector $p[1..n]$ of probabilities ordered by ascending key value k_i.
Output: The minimum expected value s_{1n} and the *root* array from which the BST can be constructed.

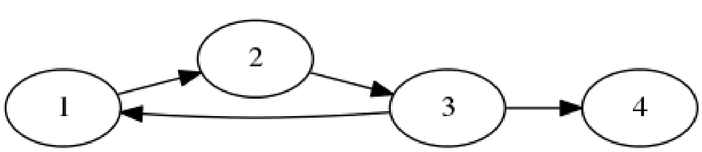
```
// Initialization
```

for $i \leftarrow 1$ **to** n **do**
    $s[i,i] \leftarrow p[i]$;
    $s[i,i-1] \leftarrow 0$;
    $root[i,i] \leftarrow i$;
end
$s[n+1,n] \leftarrow 0$;

`// Bottom-up calculation of arrays` s `and` $root$

for $d \leftarrow 1$ **to** $n-1$ **do**
    for $i \leftarrow 1$ **to** $n-d$ **do**
        $j = i + d$;
        $min \leftarrow \infty$;
        `//` min `should be larger than largest key`
        for $r \leftarrow i$ **to** j **do**
            if $(s[i,r-1] + s[r+1,j] < min)$ **then**
                $min \leftarrow s[i,r-1] + s[r+1,j]$;
                $rmin \leftarrow r$;
            end
        end
        $root[i,j] \leftarrow rmin$;
        $tmp \leftarrow 0$;
        for $k \leftarrow i$ **to** j **do**
            $tmp \leftarrow tmp + p[k]$;
        end
        $s[i,j] = min + tmp$;
    end
end
return $s[1,n]$ *and root*;

Algorithm 1: A dynamic programming algorithm to find the optimal BST.



(a) An example of a directed graph.

$$A = \begin{bmatrix} Nodes & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} \\ \mathbf{1} & 0 & 1 & 0 & 0 \\ \mathbf{2} & 0 & 0 & 1 & 0 \\ \mathbf{3} & 1 & 0 & 0 & 1 \\ \mathbf{4} & 0 & 0 & 0 & 0 \end{bmatrix}$$

(b) The adjacency matrix of the digraph.

Figure 10: The digraph and its adjacency matrix.

The transitive closure algorithm works by defining a sequence of matrices:

$A^{(0)}, \ldots, A(k), \ldots, A^{(n)}$ where $a_{ij}^{(k)} = 1$ an element of $A^{(k)}$, means that there is a non-trivial path from node i to node j via intermediate nodes with labels at most k otherwise it is 0. For example, $a_{14}^{(2)} = 0$ since there is no path from 1 to 4 via nodes labelled at most 2. The only path passes through 3. But $a_{14}^{(3)} = 1$ since we have the path: `1, 2, 3, 4`.

We argue below that the matrix $A^{(k)}$ can be completely constructed if we know the matrix $A^{(k-1)}$. This allows us to construct the matrix $A^{(n)}$ by starting with the matrix $A^{(0)} = A$ the adjacency matrix and successively computing $A^{(1)}, \ldots, A^{(n)}$.

Now consider the entry $a_{ij}^{(k)}$ in $A^{(k)}$. We can see that $a_{ij}^{(k)} = 1$ can be broken down into the following two alternatives: Either $a_{ij}^{(k-1)} = 1$ - there is a non-trivial path from i to j passing through intermediate nodes labelled at most $k-1$ or there is a path of the kind $i, \ldots, k, \ldots, j$ that is path form i to j that passes via the intermediate node labelled k exactly once.

Note that if k occurs more than once we can remove all nodes after the first k till the last k since we are only interested in any one path between i and j that passes through k. This means that all nodes between i and k must be labelled at most $k-1$ and similarly all nodes between k and j must also be labelled at most $k-1$. This allows us to write the following recurrence:

$$(a_{ij}^{(k)} = 1) \text{ iff } \begin{cases} (a_{ij}^{(k-1)} = 1) \quad \texttt{or} \\ (a_{ik}^{(k-1)} = 1) \ \texttt{and} \ (a_{kj}^{(k-1)} = 1) \end{cases}$$

The above equation shows that $a_{ij}^{(k)}$ can be computed from the entries of matrix $A^{(k-1)}$ by the following boolean equation (where we have used the C operators `or=||`, `and=&&`):

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} \,||\, (a_{ik}^{(k-1)} \,\&\&\, a_{kj}^{(k-1)})$$

The base case or starting matrix $A^{(0)}$ is the adjacency matrix. This directly gives us Algorithm 2.

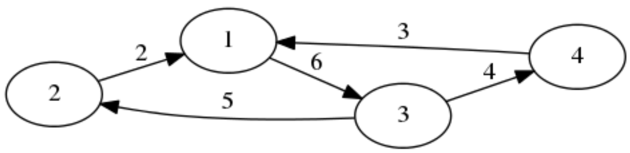
Input : The adjacency matrix A
Output: The transitive closure of A computed insitu
`// Iterative calculation of` $A^{(n)}$
for $k \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      A[i,j]=A[i,j] || (A[i,k] && A[k,j]);
    end
  end
end
return A;

Algorithm 2: Computes transitive closure of adjacency matrix A by dynamic programming (Warshall's algorithm).

Warshall's algorithm clearly has time complexity $O(n^3)$ and space complexity $O(n^2)$.

By an almost identical argument we can compute the shortest directed paths in a digraph between any two nodes. Now each edge in the directed graph has an edge weight and is represented by the distance or weight matrix $D = [d_{ij}]_{n \times n}$ where $d_{ij} \in \mathbb{R}$ is the weight of the edge between node i and j if i and j have an edge between them else it is ∞. Figure 11 shows a weighted digraph and its distance/weight matrix.

Similar to the transitive closure case we can construct the sequence of matrices $D^{(0)} = D, \ldots, D^{(k)}, \ldots, D^{(n)}$. The meaning of $d_{ij}^{(k)}$ is the minimum total path weight between nodes i and j where the intermediate nodes are labelled at most k. In analogy with Warshall's algorithm we get the following recurrence where we assume the digraph does



(a) An example of a weighted directed graph.

$$D = \begin{array}{c|cccc} Nodes & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} \\ \hline \mathbf{1} & 0 & \infty & 6 & \infty \\ \mathbf{2} & 2 & 0 & \infty & \infty \\ \mathbf{3} & \infty & 5 & 0 & 4 \\ \mathbf{4} & 3 & \infty & \infty & 0 \end{array}$$

(b) The distance/weight matrix of the digraph.

Figure 11: The digraph and its distance/weight matrix.

not have negative cycles:

$$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, (d_{ik}^{(k-1)} + d_{kj}^{(k-1)}))$$

The recurrence immediately gives us Algorithm 3.

Input : The weight matrix D
Output: The all pairs shortest path matrix computed insitu
`// Iterative calculation of` $D^{(n)}$
for $k \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      D[i,j]=min(D[i,j], (D[i,k] + D[k,j]);
    end
  end
end
return D;

Algorithm 3: Computes all pairs shortest paths matrix D by dynamic programming (Floyd's algorithm).

The time and space complexity of Floyd's algorithm by analogy to Warshall's algorithm are $O(n^3)$ and $O(n^2)$ respectively.

Exercise 3.4. Trace Warshall's and Floyd's algorithms on the example matrices in the figures 10 and 11 to find the transitive closure and the all pairs shortest distances.

References

[1] Richard Bellman, Dynamic programming, Dover (paperback edition), 2003.