Dynamic Programming

- Optimal Binary Search
- Max Independent Set on Trees

Optimal BST

Optimal BST

Optimal BST.

- Given a sorted array A[1..n] of search keys and an array f[1..n] of frequency counts.
- f[i] is the number of searches to A[i].
- Goal; Construct a BST that minimize the total cost of all searches.

The cost of a search on key k: the number of nodes on BST from the root to the node storing k.

Cost Function

Notation.

- T: a binary search tree
- v_r: the root storing the key A[r]
- v_i: the node storing key A[i]
- c(u, v)= the number of nodes on the path from u to v on T

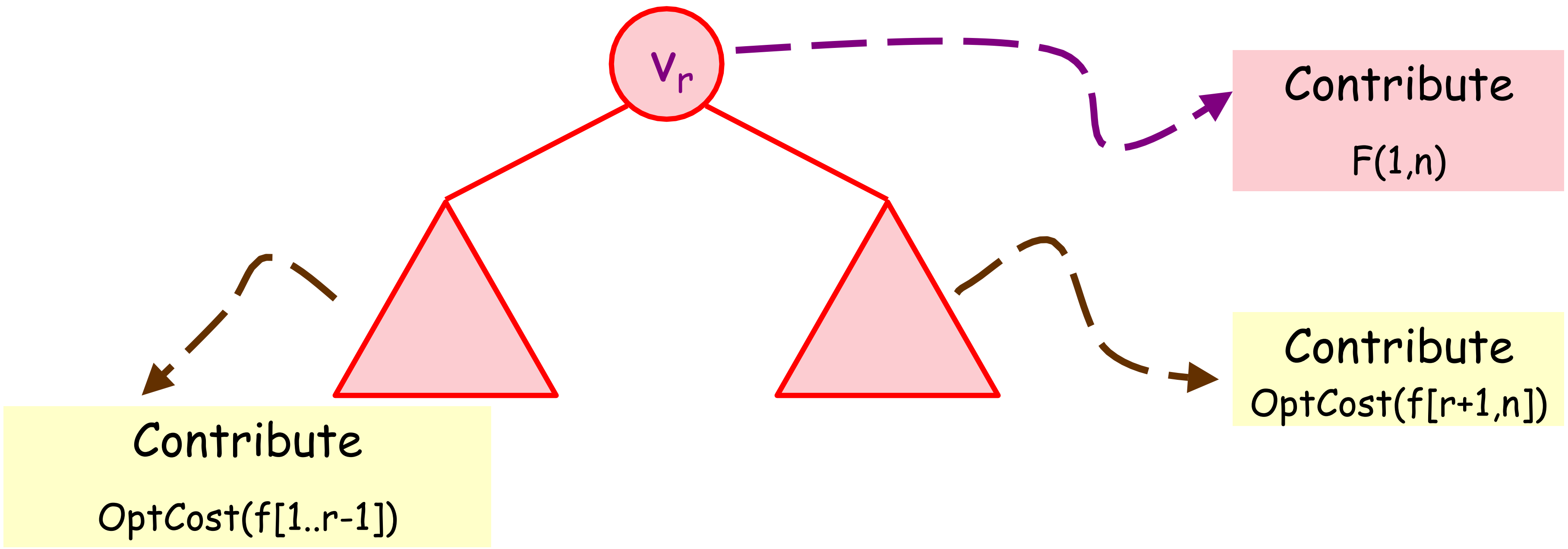
The total cost of all searches on T.

$$Cost(T, f[1..n]) = \sum_{i=1}^{n} f(i).c(v_r, v_i) =$$

$$\sum_{i=1}^{r-1} f(i).c(left(v_r), v_i) + \sum_{i=1}^{n} f[i] + \sum_{i=r+1}^{n} f(i).c(right(v_r), v_i) =$$

$$Cost(left(T), f[1..r-1]) + \sum_{i=1}^{n} f[i] + Cost(right(T), f[r+1..n])$$

Optimal Cost



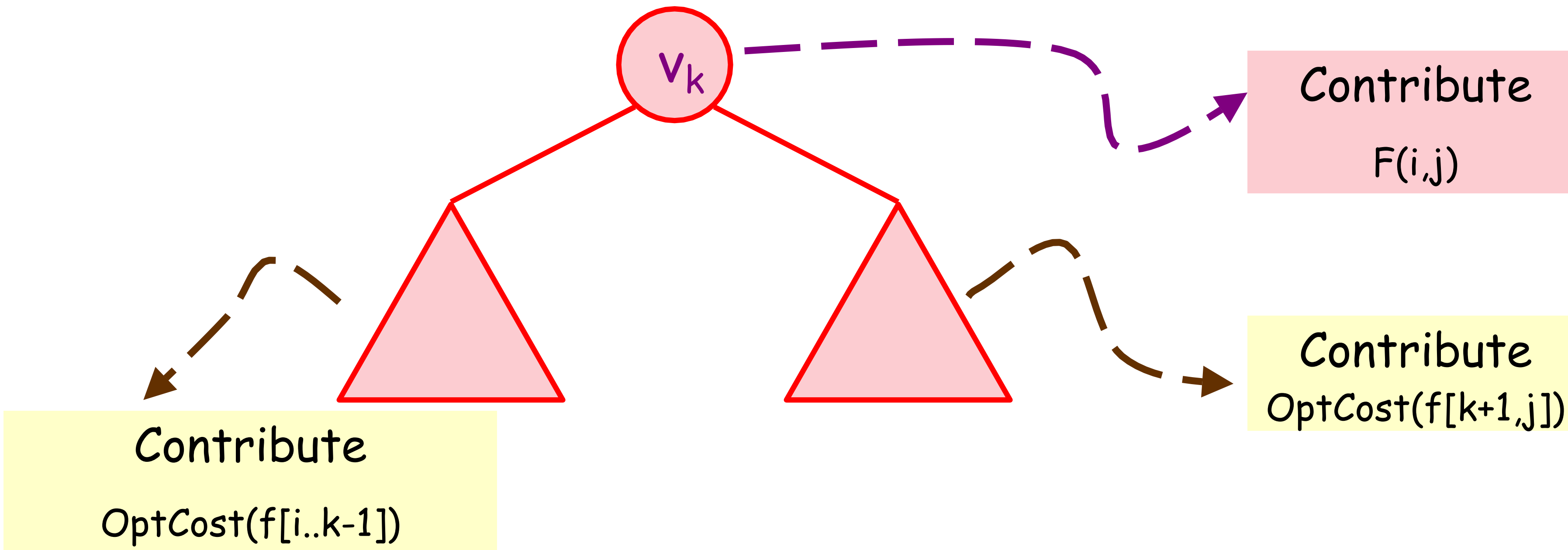
$$OptCost(f[1..n]) =$$

$$\min_{1 \le r \le n} \{OptCost(f[1..r-1]) + F(1,n) + OptCost(f(r+1..n))\} =$$

$$F(1,n) + \min_{1 \le r \le n} \{OptCost(f[1..r-1]) + OptCost(f(r+1..n))\}$$

Where F(1,n) = $\sum_{i=1}^{n} f(i)$

Optimal Cost



$$OptCost(f[i..j]) =$$

$$\min_{1 \le r \le n} \{OptCost(f[i..k-1]) + F(i,j) + OptCost(f(k+1..j))\} =$$

$$F(i,j) + \min_{1 \le r \le n} \{OptCost(f[i..k-1]) + OptCost(f(k+1..j))\}$$

Where F(i,j) = $\sum_{t=i}^{j} f(t)$

Computing F[i,j]

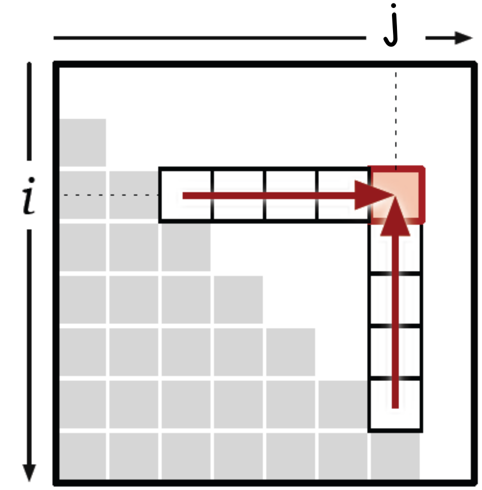
```
Input f[1..n]

sum[0] = 0
for k = 1 to n do
    sum[k] = sum[k-1]+f(k)
for i = 1 to n do
   for j = i to n do
       F[i,j] = sum[j]-sum[i-1]
return F
```

Running time: $O(n^2)$

Dynamic Programming

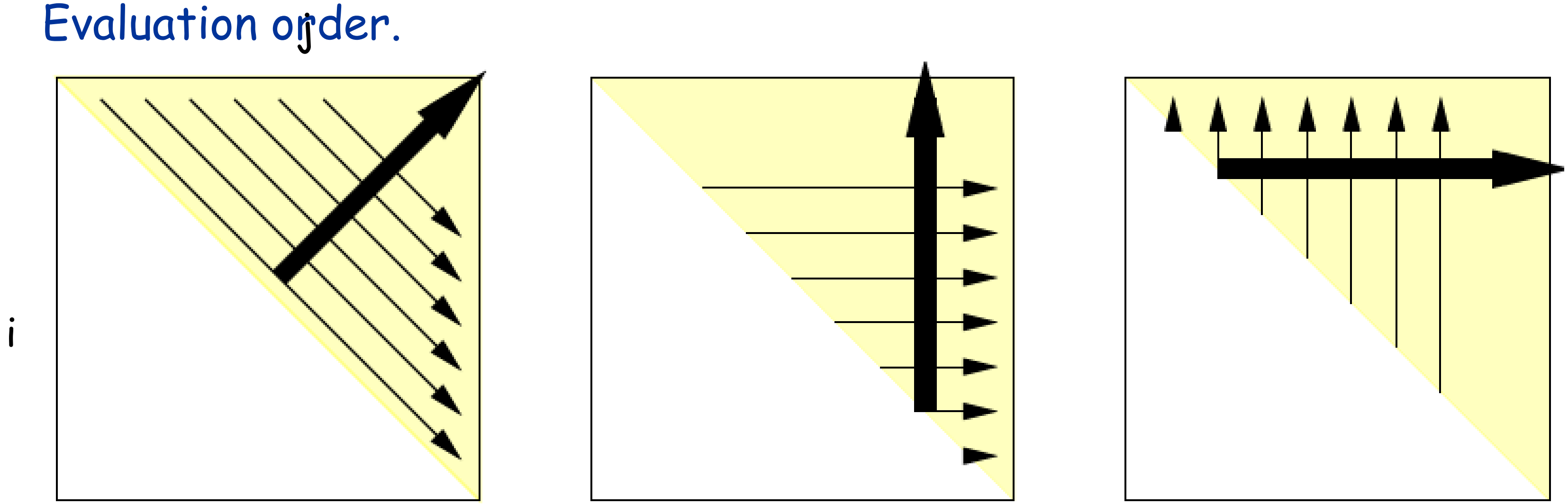
Subproblems: each subproblem is specified by two integer i and j

Memoization: Store all possible values of OptCost in a two dimensional array M.



Dependencies.

Evaluation order.



Dynamic Programming

```
Input f[1..n]

for i = 0 to n do
  for j = 0 to n do
     M[i,j] = 0
for i = 1 to n do
  for j = i to n do
     M[i,j] = +infinity
for i = 1 to n do
  for j = i to 1 do
    for k = i to j do
       M[i,j] = min(M[i,j], M[i,k-1]+M[k+1,j]+F[i,j])
return M[1,n]
```

Running time. $O(n^3)$

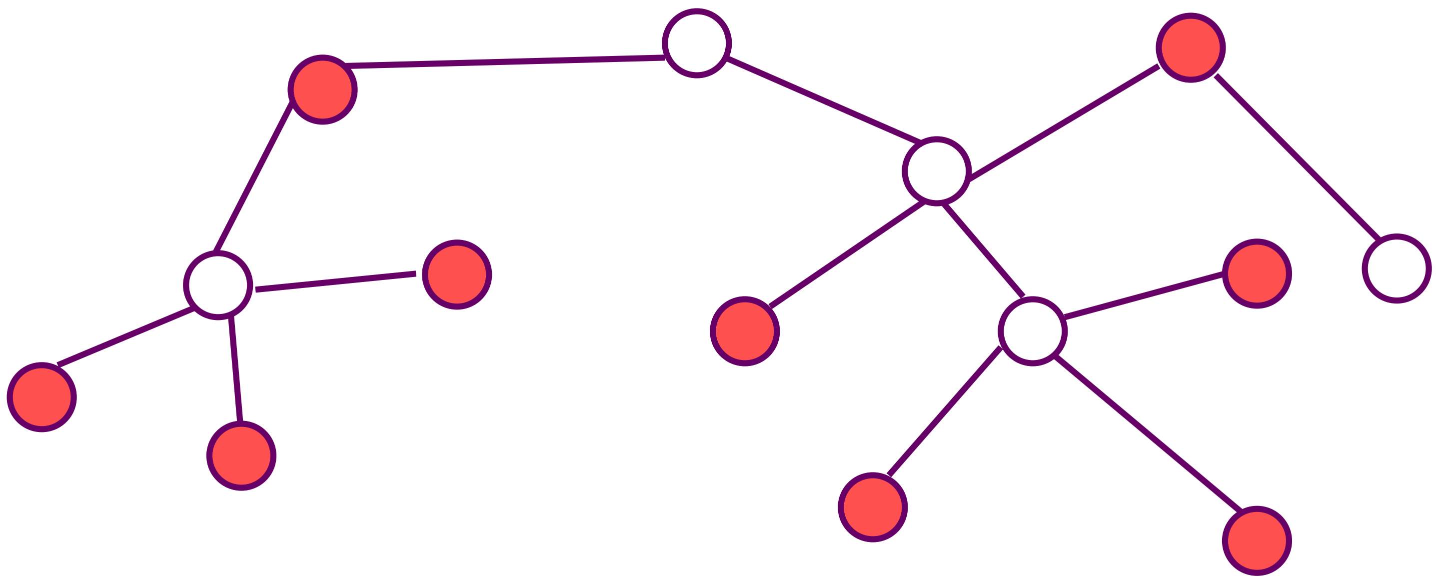
Maximum Independent Set

Max Independent Set on Trees

Def. An independent set in a graph is a subset of the vertices that have no edges between them.

MIS on trees.

- Given a tree T
- Goal; compute the Maximum Independent Set of T



A Recursive Solution

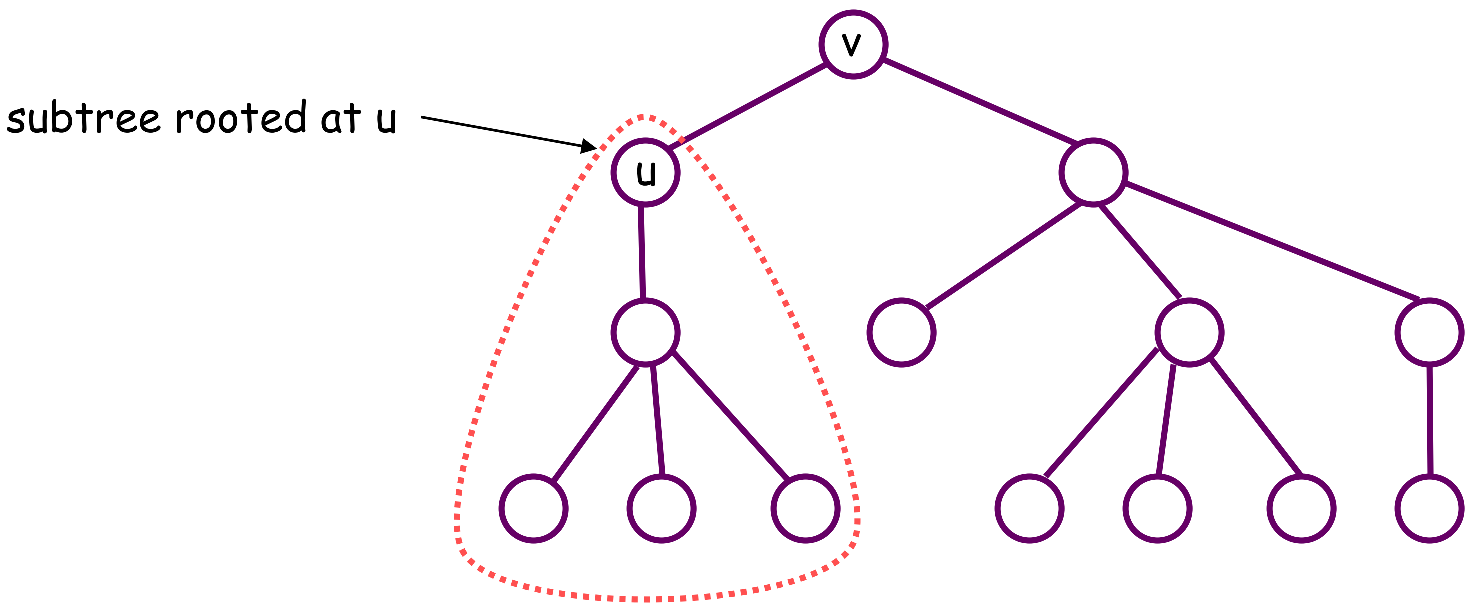
Def. N(v) is the neighborhood of v

```
MIS(T)
If T is empty then
  return 0
v = any node in T
withv = 1
for each tree T′in T \ N(v) do
  withv = withv+MIS(T′)
withoutv = 0
for each tree T′in T \ v do
  withoutv = withoutv+MIS(T′)
return max(withv, withoutv)
```

- Each recursive subproblem considers a subtree
- A tree can have exponentially many subtrees

A Recursive Solution (Improvement)

- There is a degree of freedom: we get to choose the vertex v.
- We need a recipe for choosing v in each subproblem that limits the number of subproblems the algorithm considers.
- Make tree T rooted and Let v be the root.
- This choice guarantees that each recursive sublproblem considers a rooted subtree of T.
- Each vertex in T is the root of exactly one subtree, so the number of distinct subproblems is exactly n.



A Recursive Solution (Improvement)

- There is a degree of freedom: we get to choose the vertex v.
- We need a recipe for choosing v in each subproblem that limits the number of subproblems the algorithm considers.
- Make tree T rooted and Let v be the root.
- This choice guarantees that each recursive sublproblem considers a rooted subtree of T.
- Each vertex in T is the root of exactly one subtree, so the number of distinct subproblems is exactly n.
- We can simplify the algorithm by only passing a node instead of the entire subtree.

```
MIS(v)
withv = 1
for each grandchild x of T do
  withv = withv+MIS(x)
withoutv = 0
for each child x of v do
  withoutv = withoutv+MIS(x)
return max(withv, withoutv)
```

Memoized Solution

- Storing Intermediate results. Store MIS(v) in a new field v.MIS
- Runnig time. The non-recursive time associated with each node v is proportional to the number of children and grandchildren of v. This number can be very different from one vertex to the next. But we can turn the analysis around. Each vertex contributes a constant amount of time to its parent and its grandparent
- A good order to consider subproblems. The subproblem associated with any node v depends on the subproblems associated with the children and grandchildren of v. So, we can visit the node in any order provided all children are visited before their parent.

```
MIS(v)
withoutv = 0
for each child x of v do
  withoutv = withoutv+MIS(x)
withv = 1
for each grandchild x of T do
  withv = withv+x.MIS
v.MIS = max(withv, withoutv)
return v.MIS
```

Summary

Summary

Dynamic programing is not about filling in tables. It is about smart recursion.

Dynamic programming algorithms are developed in two distinct stages:

Formulate the problem recursively.

- Write down a recursive formula or program for the whole problem in terms of answers to smaller subproblems. This is the hard part.

Build solutions to your recurrence from the bottom to up.

- Identify subproblems
- Choose a memoization data structure
- Identify dependencies
- Find a good evaluation order

References

References

- Section 3.9 and 3.10 of the text book "algorithms" by Jeff Erikson
- Section 15.5 of the text book "introduction to algorithms" by CLRS, 3rd edition.