Dynamic Programming

Job Scheduling

Knapsack

Fibonacci

Stair Climbing

CSE 3318 – Algorithms and Data Structures
University of Texas at Arlington

Alexandra Stefan
(Includes images, formulas and examples from CLRS, Dr. Bob Weems, wikipedia)

Approaches for solving DP Problems

Greedy

- typically not optimal solution (for DP-type problems)
- Build solution
- Use a criterion for picking
- Commit to a choice and do not look back

DP

- Optimal solution
- *Write math function, **sol***, that captures the dependency of solution to current pb on solutions to smaller problems
- Can be implemented in any of the following: iterative, memoized, recursive

Brute Force

- Optimal solution
- Produce all possible combinations, [check if valid], and keep the best.
- Time: exponential
- Space: depends on implementation
- It may be hard to generate all possible combinations

Iterative (bottom-up) - BEST

- Optimal solution
- *sol* is an array (1D or 2D). Size: N+1
- Fill in *sol* from 0 to N
- Time: polynomial (or pseudo-polynomial for some problems)
- Space: polynomial (or pseudo-polynomial
- To recover the choices that gave the optimal answer, must backtrace => must keep picked array (1D or 2D).

Memoized

- Optimal solution
- Combines recursion and usage of *sol* array.
- *sol* is an array (1D or 2D)
- Fill in *sol* from 0 to n
- Time: same as iterative version (typically)
- Space: same as iterative version (typically) + space for frame stack. (Frame stack depth is typically smaller than the size of the *sol* array)

Recursive

- Optimal solution
- Time: exponential (typically) =>
- DO NOT USE
- Space: depends on implementation (code). E.g. store all combinations, or generate, evaluate on the fly and keep best seen so far.
- Easy to code given math function

Improve space usage

- Improves the iterative solution
- *Saves space*
- If used, cannot recover the choices (gives the optimal value, but not the choices)

DP can solve:

- some type of **counting problems** (e.g. stair climbing)
- some type of **optimization problems** (e.g. Knapsack)
- some type of **recursively defined** pbs (e.g. Fibonacci)

Some DP solutions have ***pseudo* polynomial** time

Dynamic Programming (DP) - CLRS

- Dynamic programming (DP) applies when a problem has both of these properties:
  1. **Optimal substructure**: "optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently".
  2. **Overlapping subproblems**: "a recursive algorithm revisits the same problem repeatedly".
- Dynamic programming is typically used to:
  – Solve optimization problems that have the above properties.
  – Solve counting problems –e.g. Stair Climbing or Matrix Traversal.
  – Speed up existing recursive implementations of problems that have overlapping subproblems (property 2) – e.g. Fibonacci.
- Compare **dynamic programming** with **divide and conquer**.

Iterative or Bottom-Up Dynamic Programming

- Main type of solution for DP problems
- We can define the problems size and solve problems from size 0 going up to the size we need.
- Iterative – because it uses a loop
- Bottom-up because you solve problems from the bottom (the smallest problem size) up to the original problem size.

Bottom-Up vs. Top Down

- There are two versions of dynamic programming.
  - Bottom-up.
  - Top-down (or memoization).

- Bottom-up:
  - Iterative, solves problems in sequence, from smaller to bigger.
- Top-down:
  - Recursive, start from the larger problem, solve smaller problems as needed.
  - For any problem that we solve, **store the solution**, so we never have to compute the same solution twice.
  - This approach is also called **memoization**.

Top-Down Dynamic Programming (Memoization)

- Maintain an array/table where solutions to problems can be saved.
- To solve a problem P:
  - See if the solution has already been stored in the array.
  - If yes, return the solution.
  - Else:
    - Issue recursive calls to solve whatever smaller problems we need to solve.
    - Using those solutions obtain the solution to problem P.
    - Store the solution in the solutions array.
    - Return the solution.

Steps for iterative (bottom up) (Dr. Weems)

1. Identify problem input
2. Identify the cost/gain function (name it, describe it)
3. Give the math formula for the cost function for all cases: *base cases* and *general case*
4. Order the problems & solve them
5. Recover the choices that gave the optimal value

Other types of solutions

1. Brute force solution
2. Recursive solution (most likely exponential and inneficient)
3. Memoized solution

Weighted Interval Scheduling

(Job Scheduling)

Weighted Interval Scheduling
(a.k.a. Job Scheduling)

Problem:

Given n jobs where each job has a start time, finish time and value, (s_i, f_i, v_i) select a subset of them that do not overlap and give the largest total value.

E.g.:

start	end	value
6	8	\$2
2	5	\$6
3	11	\$5
5	6	\$3
1	4	\$5
4	7	\$2

Weighted Interval Scheduling (a.k.a. Job Scheduling)

Problem:

Given n jobs where each job has a start time, finish time and value, (s_i,f_i,v_i) select a subset of them that do not overlap and give the largest total value.

E.g.:

(start,	end,	value)
(6,	8,	\$2)
(2,	5,	\$6)
(3,	11,	\$5)
(5,	6,	\$3)
(1,	4,	\$5)
(4,	7,	\$2)

Preprocessing:

- Sort jobs in increasing order of their finish time.
- For each job ,*i*, compute the last job prior to *i*, *p(i)*, that does not overlap with *i*.
  - p(4) is 1 (last job that does not overlap with job 4)
  - p(5) is 3
    - Max (sol(4), 2+sol(3))

After preprocessing:

JobId	(start,	end,	value,	p(i))
1	(1,	4,	\$5,	)
2	(2,	5,	\$6,	)
3	(5,	6,	\$3,	)
4	(4,	7,	\$2,	)
5	(6,	8,	\$2,	)
6	(3,	11,	\$5,	)

Weighted Interval Scheduling (a.k.a. Job Scheduling)

Problem:

– Given n jobs where each job has a start time, finish time and value, (s_i,f_i,v_i) select a subset of them that do not overlap and give the largest total value.

Preprocessing:

- Sort jobs in increasing order of their finish time. –already done here
- For each job ,*i*, compute the last job prior to *i*, *p(i)*, that does not overlap with *i*.

Solve the problem:

Steps: one step for each job.

Option: pick it or not

Smaller problems: 2:

pb1 = jobs 1 to i-1, => sol(i-1)

pb2 = jobs 1 to p(i) (where p(i) is the last job before i that does not overlap with i. => sol(p(i))

Solution function (gives the money value: sol(i) = the most money I can make using jobs 1,2,..,i):

$$sol(0) = 0$$

$$sol(i) = \max\{sol(i-1), v(i) + sol(p(i))\}$$

Time complexity: O(n)) *(if data is already preprocessed)* Fill out *sol(i) in* constant time for each i)

O(nlgn) (if jobs need to be sorted first and an nlgn sorting algorithm was used, and binary search for finding p(i))

Solve the problem:

Steps: one step for each job.
Option: pick it or not (pick job i or not pick it)
Smaller problems: 2:
pb1 = jobs 1 to i-1, => sol(i-1)
pb2 = jobs 1 to p(i) (where p(i) is the last job before i that does not overlap with i. => sol(p(i))
Solution function:

$$sol(0) = 0$$
$$sol(i) = \max\{sol(i-1), v(i) + sol(p(i))\}$$

Time complexity: _______

Original problem:
(start, end, value)
(6, 8, $2)
(2, 5, $6)
(3, 11, $5)
(5, 6, $3)
(1, 4, $5)
(4, 7, $2)

After preprocessing
(sorted by END time):
JobId (start, end, value, p(i))
1 (1, **4**, $5, __)
2 (2, **5**, $6, __)
3 (5, **6**, $3, __)
4 (4, **7**, $2, __)
5 (6, **8**, $2, __)
6 (3, **11**, $5, __)

i	v_i	p_i	sol(i) ($, money)	sol(i) used i	In optimal solution
0	*0*	**-1**	0		
1	*5*	**0**	5 = Max{0, 5+0}	Yes	
2	*6*	**0**	6 = Max{5,6+0}	Yes	yes
3	*3*	**2**	9 =Max{6, 3+6}	Yes	yes
4	*2*	**1**	9 = Max{9, 2+5}	No	
5	*2*	**3**	11 = max{9, 2+9}	Yes	yes
6	*5*	**0**	11 = max{11, 5+0}	No	

Optimal value: _**11**___, jobs picked to get this value: **2,3,5**

Solve the problem:

Steps: one step for each job.

Option: pick it or not (pick job i or not pick it)

Smaller problems: 2:

pb1 = jobs 1 to i-1, => sol(i-1)

pb2 = jobs 1 to p(i) (where p(i) is the last job before i that does not overlap with i. => sol(p(i))

Solution function:

$sol(0) = 0$

$sol(i) = \max\{sol(i-1), v(i) + sol(p(i))\}$

Time complexity: ______

Original problem:
(start, end, value)
(6, 8, $2)
(2, 5, $6)
(3, 11, $5)
(5, 6, $3)
(1, 4, $5)
(4, 7, $2)

After preprocessing
(sorted by END time):
JobId (start, end, value, p(i))
1 (1, **4**, $5, __)
2 (2, **5**, $6, __)
3 (5, **6**, $3, __)
4 (4, **7**, $2, __)
5 (6, **8**, $2, __)
6 (3, **11**, $5, __)

i	v_i	p_i	sol(i) ($, money)	sol(i) used i	In optimal solution
0	*0*	**-1**	0		
1	*5*	**0**	5 = max{0, 5+0}	Yes	
2	*6*	**0**	6 = max{5, 6+0}	Yes	yes
3	<u>*3*</u>	**2**	9 = max{6,<u>3</u>+6}	Yes	yes
4	*2*	**1**	9 = max{9, 2+ 5}	No	
5	*2*	**3**	11 = max{9, 2+ 9}	Yes	yes
6	*5*	**0**	11 = max{11, 5+0}	No	

Optimal value: __**11**__, jobs picked to get this value: **5,3,2**

Solve the problem:

Steps: one step for each job.

Option: pick it or not (pick job i or not pick it)

Smaller problems: 2:

pb1 = jobs 1 to i-1, => sol(i-1)

pb2 = jobs 1 to p(i) (where p(i) is the last job before i that does not overlap with i. => sol(p(i))

Solution function:

$$sol(0) = 0$$
$$sol(i) = \max\{sol(i-1), v(i) + sol(p(i))\}$$

Original problem:
(start, end, value)
(6, 8, $2)
(2, 5, $6)
(3, 11, $5)
(5, 6, $3)
(1, 4, $5)
(4, 7, $2)

After preprocessing
(sorted by END time):
JobId (start, end, value, p(i))
1 (1, 4, $5, _0_)
2 (2, 5, $6, _0_)
3 (5, 6, $3, _2_)
4 (4, 7, $2, _1_)
5 (6, 8, $2, _3_)
6 (3, 11, $5, _0_)

Time complexity: O(n) (if data is preprocessed)

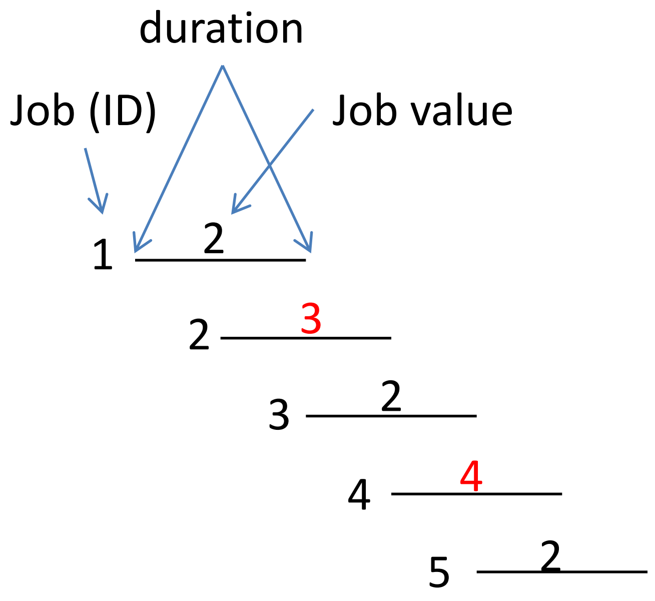
O(nlgn) (if jobs need to be sorted first and an nlgn sorting algorithm was used, and binary search for finding p(i))

i	v_i	p_i	sol(i)	sol(i) used i	In optimal solution
0	*0*	**-1**	0	-	
1	*5*	**0**	5 = max{0, 5+0}	Y	
2	*6*	**0**	6 = max{5, 6+0}	Y	Y
3	*3*	**2**	9 = max{6, 3+6}	Y	Y
4	*2*	**1**	9 = max{9, 2+5}	N	
5	*2*	**3**	11 = max{9, 2+9}	Y	Y
6	*5*	**0**	11 = max{11, 5+0}	N	

Optimal value: 11, jobs picked to get this value: 2,3,5

Another example

- Notations conventions:
  - Jobs are already sorted by end time
  - Horizontal alignment is based on time. *In this example, only consecutive jobs overlap*, (e.g. jobs 1 and 3 do not overlap).



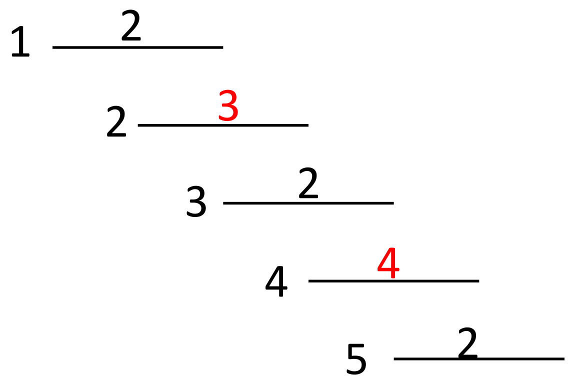
E.g.:

(Job,	start,	end,	value)
(1,	3pm,	5pm,	2$)
(2,	4pm,	6pm,	3$)
(3,	5pm,	7pm,	2$)
(4,	6pm,	8pm,	4$)
(5,	7pm,	9pm,	2$)

Time complexity: O(n)

Recovering the Solution

- Example showing that when computing the optimal gain, we *cannot decide which jobs will be part of the solution and which will not*. We can only recover the jobs picked AFTER we computed the optimum gain and by going from end to start.

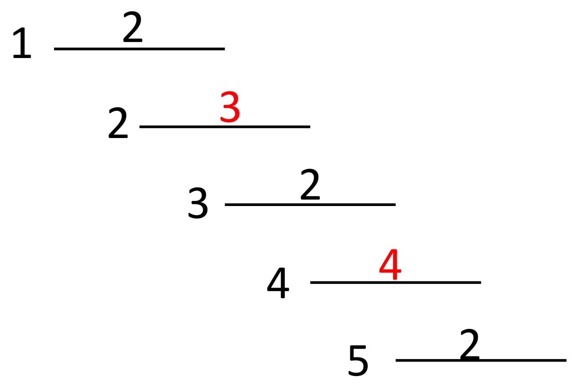


i	v_i	p_i	sol(i)	sol(i) used i	In optimal solution
0	0	0	0	-	-
1	2	0	2	Yes	-
2	3	0	3	Yes	Yes
3	2	1	4	Yes	-
4	4	2	7	Yes	Yes
5	2	3	7	No	-

Time complexity: O(n)

Job Scheduling – Brute Force Solution

- For each job we have the option to include it (1) or not(0). Gives:
  - The power set for a set of 5 elements, or
  - All possible permutations with repetitions over n positions with values 0 or 1=> $O(2^n)$
  - Note: exclude sets with overlapping jobs.
- Time complexity: $O(2^n)$



1	2	3	4	5	Valid	Total value
0	0	0	0	0	yes	0
0	0	0	0	1	yes	2
0	0	0	1	0	yes	4
0	0	0	1	1	no	
0	0	1	0	0	yes	2
0	0	1	0	1	yes	4 (=2+2)
0	0	1	1	1	no	
…	…	…	…	…	…	…
1	1	1	1	1	no	

Bottom-up (BEST)

Math function:
sol(0) = 0
$sol(i) = \max\{sol(i-1), v(i) + sol(p(i))\}$

The program will create an populate an array, sol, corresponding to the *sol* function from the math definition.

```
// Bottom-up (the most efficient solution)
int js_iter(int* v, int*p, int n){
   int j, with_j, without_j;
   int sol[n+1];
   // optionally, may initialize it to -1 for safety
   sol[0] = 0;
   for(j = 1; j <= n; j++){
      with_j = v[j] + sol[p[j]];
      without_j = sol[j-1];
      if ( with_j >= without_j)
         sol[j] = with_j;
      else
         sol[j] = without_j;
   }
   return sol[n];
}
```

The sol array must have size n+1 b.c. we must access indexes from 0 to n.

j	v_j	p_j	sol[j]
0	*0*	**-1**	0
1	*5*	**0**	5 = max{0, 5+0}
2	*6*	**0**	6 = max{5, 6+0}
3	*3*	**2**	9 = max{6, 3+6}
4	*2*	**1**	9 = max{9, 2+5}
5	*2*	**3**	11 = max{9, 2+9}
6	*5*	**0**	11 = max{11, 5+0}

Time complexity: Θ(N), Space complexity: Θ(N)

Recursive (inefficient) – SKIP for now, Fall 2020

Math function:
$sol(0) = 0$
$sol(i) = \max\{sol(i-1), v(i) + sol(p(i))\}$

```
// Inefficient recursive solution:
int jsr(int* v, int*p, int n){
    if (n == 0) return 0;
    int res;
    int with_n = v[n] + jsr(v,p,p[n]);
    int without_n = jsr(v,p,n-1);
    if ( with_n >= without_n)
         res = with_n;
    else
         res = without_n;
    return res;
}
```

In the recursive version:
- We write the solution for problem size n
- Instead of a look-up in the array, we make a recursive call for the smaller problem size.
- It will recompute the answer for the same problem multiple times (instead of saving it and looking it up) and that will make it inefficient.

j	v_j	p_j	sol[j]
0	*0*	**-1**	0
1	*5*	**0**	5 = max{0, 5+0}
2	*6*	**0**	6 = max{5, 6+0}
3	*3*	**2**	9 = max{6, 3+6}
4	*2*	**1**	9 = max{9, 2+5}
5	*2*	**3**	11 = max{jsr(v,p,4), 2+jsr(v,p,3)}
6	*5*	**0**	11 = max{jsr(v,p5), 5+jsr(v,p,0)}

We will revisit this at the end of the semester if there is time. – Fall 2020

Memoization (Recursion combined with saving)

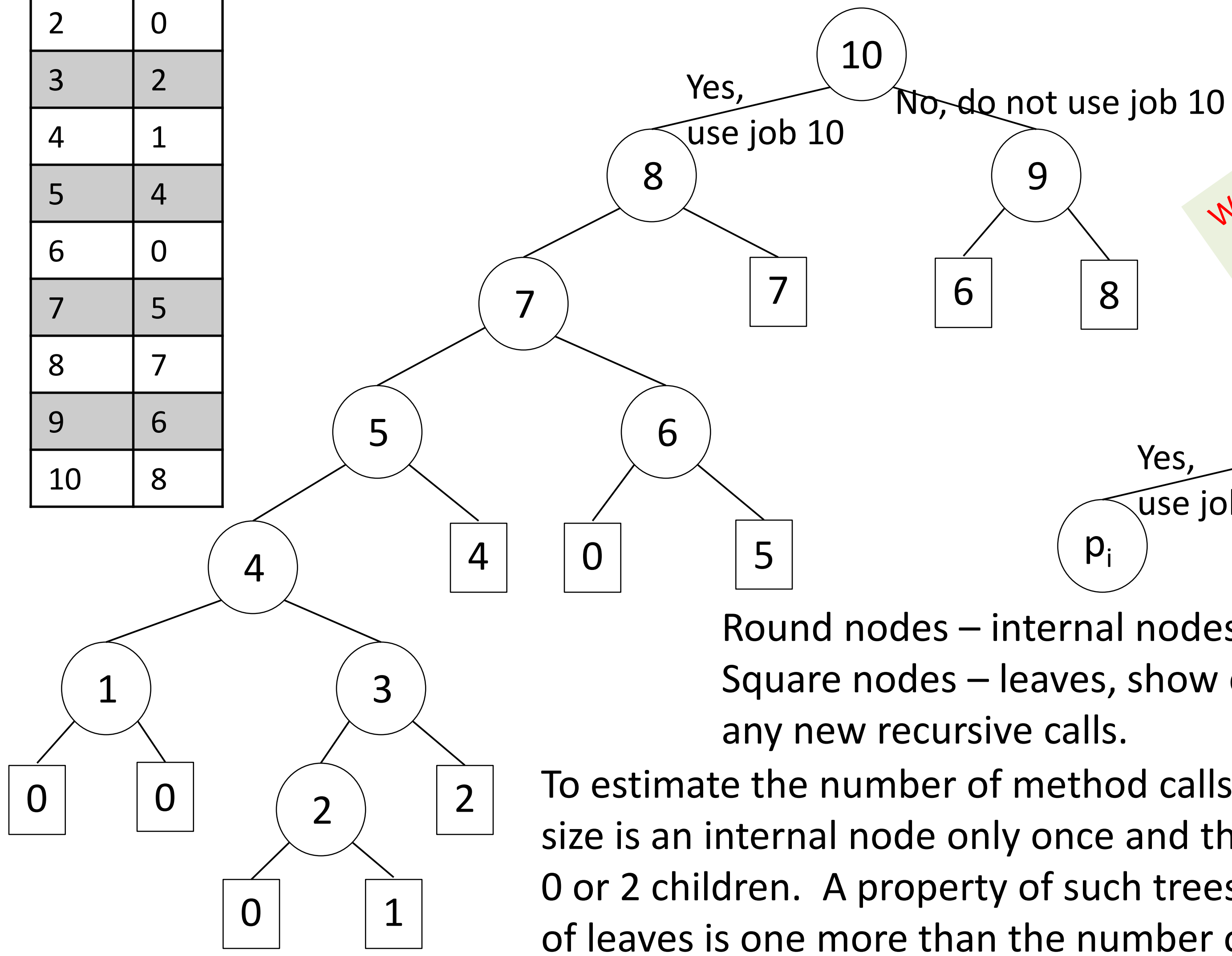
Math function:
$sol(0) = 0$
$sol(i) = \max\{sol(i-1), v(i) + sol(p(i))\}$

```
// Memoization efficient recursive solution:
int jsm(int* v, int*p, int n, int* sol){
    if (sol[n] != -1) // already computed.
        return sol[n]; // Used when rec call for a smaller problem.
    int res;
    int with_n = v[n] + jsm(v,p,p[n],sol);
    int without_n = jsm(v,p,n-1,sol);
    if ( with_n >= without_n)     res = with_n;
    else                          res = without_n;
    sol[n] = res;
    return res;
}
int jsr_out(int* v, int*p, int n){
    int sol[n+1];
    int j;
    sol [0] = 0;
    for (j = 1; j<= n; j++)  sol [j] = -1; //not computed
    jsm(v,p,n,sol);
    return sol[n];
}
```

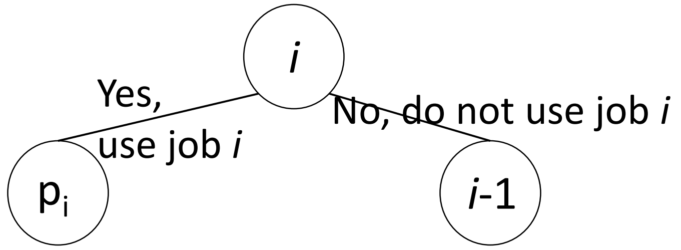
Skip for now. We will revisit this at the end of the semester if there is time. – Fall 2020

Function call tree for the memoized version

Job i	p(i)
1	0
2	0
3	2
4	1
5	4
6	0
7	5
8	7
9	6
10	8



We will revisit this at the end of the semester if there is time. – Fall 2020



Round nodes – internal nodes. Require recursive calls.
Square nodes – leaves, show calls that return without any new recursive calls.

To estimate the number of method calls note that every problem size is an internal node only once and that every node has exactly 0 or 2 children. A property of such trees states that the number of leaves is one more than the number of internal nodes => there are at most (1+2N) calls. Here: N = 10 jobs to schedule.

Fibonacci Numbers

Fibonacci Numbers

- Generate Fibonacci numbers
  - 3 solutions: inefficient recursive, memoization (top-down dynamic programming (DP)), bottom-up DP.
  - Not an optimization problem but it has overlapping subproblems => DP eliminates recomputing the same problem over and over again.

Fibonacci Numbers

- Fibonacci(0) = 0
- Fibonacci(1) = 1
- If N >= 2:

  Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)
- How can we write a function that computes Fibonacci numbers?

Fibonacci Numbers

- Fibonacci(0) = 0
- Fibonacci(1) = 1
- If N >= 2:   Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)
- Consider this function: what is its running time?

Notice the mapping/correspondence of the mathematical expression and code.

```
int Fib(int i)
{
  if (i < 1) return 0;
  if (i == 1) return 1;
  return Fib(i-1) + Fib(i-2);
}
```

Fibonacci Numbers

- Fibonacci(0) = 0
- Fibonacci(1) = 1
- If N >= 2: Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)
- Consider this function: what is its running time?
  - g(N) = g(N-1) + g(N-2) + constant

  $\Rightarrow$ g(N) $\geq$ Fibonacci(N) => g(N) = Ω(Fibonacci(N)) => g(N) = $\Omega(1.618^N)$

  Also g(N) $\leq$ 2g(N-1)+constant => g(N) $\leq c2^N$ => g(N) = $O(2^N)$

  => g(N) is exponential
  - We cannot compute Fibonacci(40) in a reasonable amount of time (with this implementation).
  - See how many times this function is executed.
  - Draw the tree

```
int Fib(int i)
{
  if (i < 1) return 0;
  if (i == 1) return 1;
  return Fib(i-1) + Fib(i-2);
}
```

Fibonacci Numbers

- Fibonacci(0) = 0
- Fibonacci(1) = 1
- If N >= 2:   Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)
- Alternative to inefficient recursion: compute from small to large and store data in an array.

Notice the mapping/correspondence of the mathematical expression and code.

linear version (Iterative, bottom-up):

```
int Fib_iter (int i)  {
  int F[i+1];
  F[0] = 0;   F[1] = 1;
  int k;
  for (k = 2; k <= i; k++) F[k] = F[k-1] + F[k-2];
  return F[i];
}
```

exponential version:

```
int Fib(int i)  {
  if (i < 1) return 0;
  if (i == 1) return 1;
  return Fib(i-1) + Fib(i-2);
}
```

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
F															

Applied scenario

- F(N) = F(N-1)+F(N-2), F(0) = 0, F(1) = 1,
- Consider a webserver where clients can ask what the value of a certain Fibonacci number, F(N) is, and the server answers it.

  How would you do that? (the back end, not the front end)

(Assume a uniform distribution of F(N) requests over time most F(N) will be asked.)

- Constraints:
  - Each loop iteration or function call costs you 1cent.
  - Each loop iteration or function call costs the client 0.001seconds wait time
  - Memory is cheap
- How would you charge for the service? (flat fee/function calls/loop iterations?)
- Think of some scenarios of requests that you could get. Think of it with focus on:
  - "good sequence of requests"
  - "bad sequence of requests"
  - Is it clear what good and bad refer to here?

Fibonacci Numbers

- Fibonacci(0) = 0 , Fibonacci(1) = 1
- If N >= 2:   Fibonacci(N) = Fibonacci(N-1) + Fibonacci(N-2)
- Alternative: remember values we have already computed.
- Draw the new recursion tree and discuss time complexity.

memoized version:

```
int Fib_mem_wrap(int i) {
  int sol[i+1];
  if (i<=1) return i;
  sol[0] = 0;  sol[1] = 1;
  for(int k=2; k<=i; k++)  sol[k]=-1;
  Fib_mem(i,sol);
  return sol[i];
}
int Fib_mem (int i, int[] sol)  {
  if (sol[i]!=-1) return sol[i];
  int res = Fib_mem(i-1, sol) + Fib_mem(i-2, sol);
  sol[i] = res;
  return res;
}
```

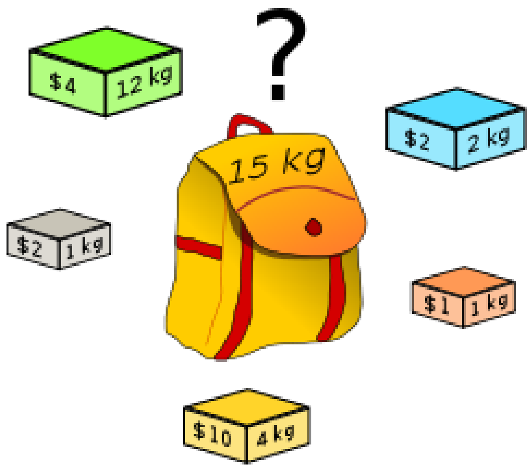
exponential version:

```
int Fib(int i)  {
  if (i < 1) return 0;
  if (i == 1) return 1;
  return Fib(i-1) + Fib(i-2);
}
```

Fibonacci and DP

- Computing the Fibonacci number is a DP problem.
- It is a counting problem (not an optimization one).
- We can make up an ‘applied’ problem for which the DP solution function is the Fibonacci function. Consider: A child can climb stairs one step at a time or two steps at a time (but he cannot do 3 or more steps at a time). How many different ways can they climb? E.g. to climb 4 stairs you have 5 ways: {1,1,1,1}, {2,1,1}, {1,2,1}, {1,1,2}, {2,2}

The Knapsack Problem



Problem:

- A thief breaks into a store.
- The maximum total weight that he can carry is *W*.
- There are *N* types of items at the store.
- Each type t_i has a value v_i and a weight w_i.
- What is the <u>maximum total **value**</u> that he can carry out?
- <u>What items should he pick</u> to obtain this maximum value?

Variations based on item availability:

- Unlimited amounts – *Unbounded* Knapsack
- Limited amounts – *Bounded* Knapsack
- Only one item – *0/1* Knapsack
- Items can be 'cut' – *Continuous* Knapsack (or *Fractional* Knapsack)

Image from Wikipedia:
https://en.wikipedia.org/wiki/Knapsack_problem

Variations of the Knapsack Problem

Unbounded:
Have unlimited number of each object.
Can pick any object, any number of times.
(Same as the stair climbing with gain.)

Fractional:
For each item can take the whole quantity, or a fraction of the quantity.

Bounded:
Have a limited number of each object.
Can pick object i, at most x_i times.

0-1 (special case of Bounded):
Have only one of each object.
Can pick either pick object i, or not pick it.
This is on the web.

All versions have:	
N	number of different types of objects
W	the maximum capacity (kg)
$v_1, v_2, \ldots, v_N$	Value for each object. ($$)
$w_1, w_1, \ldots, w_N,$	Weight of each object. (kg)

The bounded version will have the amounts: $c_1, c_2, \ldots, c_N$ of each item.

Worksheet: Unbounded Knapsack

Max capacity: W=17

Item type:	A	B	C	D	E
Weight (kg)	3	4	7	8	9
Value ($$)	<u>4</u>	<u>6</u>	<u>11</u>	<u>13</u>	<u>15</u>

Math cost function:

$$Sol(k) = 0, \quad \forall k < \min_{1 \le i \le n} w_i$$

$$(base\ cases, no\ item\ fits)$$

$$Sol(k) = \max_{\substack{\forall i, s.t. w_i \le k \\ (1 \le i \le n)}} \{val_i + Sol(k - w_i)\}$$

Where $k = current\ weight = current\ problem\ size$

	index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
solution	**Sol**																		
	Picked																		
Work (to compute solution)	**A, 3, <u>4</u>**																		
	B, 4, <u>6</u>																		
	C, 7,<u>11</u>																		
	D, 8,<u>13</u>																		
	E, 9,<u>15</u>																		

Rows A,B,C,D,E are used to compute the final solution, in Sol and Picked. They show your work.

Answers: Unbounded Knapsack

Math cost function:

$$Sol(k) = 0, \quad \forall k < \min_{1 \le i \le n} w_i$$

$$(base\ cases, no\ item\ fits)$$

$$Sol(k) = \max_{\substack{\forall i, s.t. w_i \le k \\ (1 \le i \le n)}} \{val_i + Sol(k - w_i)\}$$

Where $k = current\ weight = current\ problem\ size$

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Sol	0	0	0	4	6	6	8	11	13	15	15	17	19	21	22	24	26	28
Picked	-	-	-	A	B	B	A	C	D	E	A	A	A	B	C	C	C	D
A, 3, 4	-	-	-	0,**4**	1,4	2,4	3,**8**	4, 10	5, 10	6, 12	7, **15**	8, **17**	9, **19**	10, 19	11, 21	12, 23	13, 25	14, 26
B, 4, 6	-	-	-	-	0, **6**	1,**6**	2, 6	3, 10	4, 12	5, 12	6, 14	7, 17	8, 19	9, **21**	10, 21	11, 23	12, 25	13, 27
C, 7,11	-	-	-	-	-	-	-	0, **11**	1, 11	2, 11	3, 15	4, 17	5, 17	6, 19	7, **22**	8, **24**	9, **26**	10, 26
D,8,13	-	-	-	-	-	-	-	-	0, **13**	1, 13	2, 13	3, 17	4, 19	5, 19	6, 21	7, 24	8, 26	9, **28**
E,9,15	-	-	-	-	-	-	-	-	-	0, **15**	1, 15	2, 15	3, 19	4, 21	5, 21	6, 23	7, 26	8, 28

Red – optimal, underscore – value(money)

Unbounded Knapsack – recover the items

Find the items that give the optimal value. For example in the data below, what items will give me value 31 for a max weight of 22?

Note that the item values are different from those on the previous page. (They are from a different problem instance.)

Item type:	**A**	**B**	**C**	**D**	**E**
Weight (kg)	3	4	7	8	9

ID of picked item

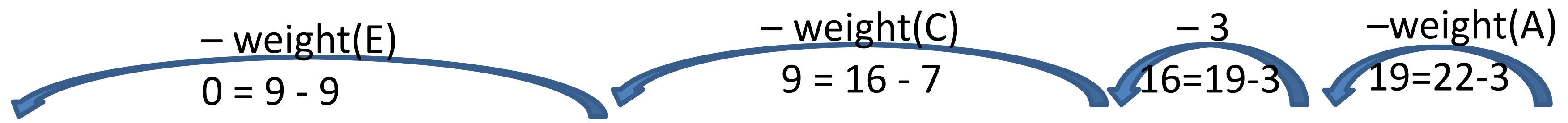
Kg	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
ID	-1	-1	-1	A	B	B	A	C	D	E	A	A	A	A	C	A	C	A	E	A	A	A	A
$$	0	0	0	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24	26	27	28	30	31

Unbounded Knapsack – recover the items

Find the items that give the optimal value. For example in the data below, what items will give me value 31 for a max weight of 22?

Note that the item values are different from those on the previous page. (They are from a different problem instance.)

Item type:	**A**	**B**	**C**	**D**	**E**
Weight (kg)	3	4	7	8	9



Kg	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
ID	-1	-1	-1	A	B	B	A	C	D	**E**	A	A	A	A	C	A	**C**	A	E	**A**	A	A	**A**
$$	0	0	0	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24	26	27	28	30	31

Answer: E, C, A, A

Iterative Solution for Unbounded Knapsack

```
/* Assume arrays v and w store the item info starting at
index 1: first item has value v[1] and weight w[1]   */
int knapsack(int W, int n, int * v, int * w){
   int sol[W+1]; int picked[W+1];
   sol[0] = 0;
   for(k=1; k<=W; k++) {
      mx = 0; choice = -1; // no item
      for(i=0;i<n;i++) {
         if (k>=w[i]) {
            with_i = v[i]+sol[k-w[i]];
            if (mx < with_i) {
               mx = with_i;
               choice = i;
            }
      }// for i
      sol[k]=mx; picked[k] = choice;
   }// for k
   return sol[W];
} //Time: Θ(nW) [pseudo polynomial: store W in lgW bits] Space: Θ(W)
```

Math cost function:

$$Sol(k) = 0,\ \ \forall k < \min_{1 \le i \le n} w_i$$

$$(base\ cases, no\ item\ fits)$$

$$Sol(k) = \max_{\substack{\forall i, s.t. w_i \le k \\ (1 \le i \le n)}} \{val_i + Sol(k - w_i)\}$$

Where $k = current\ weight = current\ problem\ size$

Worksheet: **0/1 Knapsack** (not fractional)

optimal solution (for a **smaller** problem size), excluding item i

optimal solution (for **this** problem size), excluding item i

Value_using_**first_i_items:**
Sol**[i]** [k] = **max**{Sol**[i-1]** **[** k – w[i]**]** + v[i], Sol**[i-1]** [k]

Math cost function:
$Sol(0, k) = 0, \forall k$
$Sol(i, k) = 0, , \forall k \ s.t. \ k < \min\limits_{\forall 0 \le t \le i} w_t$
$Sol(i, k) =$
$\max\{sol(i-1, k), vi + sol(i-1, k-w_i)\}$
Where $k = current\ weight = current\ problem\ size$

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
No item																		
A, 3, 4																		
B, 4, 6																		
C, 7,11																		
D, 8,13																		
E, 9,15																		

Answer: 0/1 Knapsack (not fractional)

optimal solution (for a **smaller** problem size), excluding item i

optimal solution (for **this** problem size), excluding item i

Value_using_**first_i_items:**
Sol**[i]** [k] = **max**{Sol**[i-1]** **[** k – w[i]**]** + v[i], Sol**[i-1]** [k]

Math cost function:

$Sol(0, k) = 0, \forall k$

$Sol(i, k) = 0, , \forall k \text{ s.t. } k < \min_{\forall 0 \le t \le i} w_t$

$Sol(i, k) =$
$\max\{sol(i-1, k), vi + sol(i-1, k - w_i)\}$

Where $k = current\ weight = current\ problem\ size$

E.g.: Value_using_**first_3_items(A,B,C):** Sol**[3]** [14] = **max**{Sol**[2]** **[14 - 7]** +11, Sol**[2]** [7] = max{10+11, 10} = 21

	index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
0	**No item**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	**A, 3, 4**	0	0	0	4*	4*	4*	4*	4*	4*	4*	4*	4*	4*	4*	4*	4*	4*
2	**B, 4, 6**	0	0	0	4	6*	6*	6*	10*	10*	10*	10*	10*	10*	10*	10*	10*	10*
3	**C, 7,11**	0	0	0	4	6	6	6	11*	11*	11*	15*	17*	17*	17*	+11 21*	21*	21*
4	**D, 8,13**	0	0	0	4	6	6	6	11	13*	13*	15	17*	19*	19*	21	**24***	**24***
5	**E, 9,15**	0	0	0	4	6	6	6	11	13	15*	15*	17	19*	21*	21*	**24**	**26***

Iterative Solution for 0/1 Knapsack

```
/* Assume arrays v and w store the item info starting at
index 1: first item has value v[1] and weight w[1]   */

int knapsack01(int W, int n, int * v, int * w){
   int sol[n+1][W+1];
   for(k=0; k<=W; k++) { sol[0][k] = 0;}
   for(i=1; i<=n; i++) {
       for(k=0;k<=W;k++) {
          sol[i][k] = sol[i-1][k]; // solution without item i
          if (k>w[i]) {
             with_i = v[i]+sol[i-1][k-w[i]];
             if (sol[i][k] < with_i) {   // better choice
                sol[i][k] = with_i;      // keep it
             }
          }
       }// for k
   }// for i
   return sol[n][W];
}  // Time: Θ(nW)  Space: Θ(nW) [pseudo polynomial]
```

Unbounded vs 0/1 Knapsack Solutions

- Unbounded (unlimited number of items)
  - Need only one (or two) 1D arrays: sol (and picked) of size (max_weight+1).
  - The other rows (one per item) are added to <u>show the work</u> that we do in order to figure out the answers that go in the table. There is NO NEED to store it.
  - Similar problem: Minimum Number of Coins for Change (solves a minimization, not a maximization problem): https://www.youtube.com/watch?v=Y0ZqKpToTic
- 0/1 (most web resources show this problem)
  - MUST HAVE one or two 2D tables, of size: (items+1) x (max_weight+1).
  - Each row (corresponding to an item) gives the solution to the problem using items from rows 0 to that row.
  - Whenever you look back to see the answer for a precomputed problem you look precisely <u>on the row above</u> because that gives a solution with the items in the rows above (excluding this item).
    - Unbounded knapsack can repeat the item => no need for sol excluding the current item => 1D

Improving memory usage

- Optimize the memory usage: store only smaller problems that are needed.
- NOTE: if a sliding window is used the choices cannot be recovered (i.e. cannot recover what items to pick to achieve the computed optimal value).
- Unbounded : the sliding window size is the max of the items weights => $\Theta(\max_i(w_i))$
- 0/1: the sliding window is 2 rows from the table => $\Theta(W)$
- Draw the sliding window arrays for the above problems.
- How do you update the arrays?
- Note: the sliding window term is used in another context (for a specific type of DP problems) and it means something else, so do NOT read the web resources on sliding window as they will NOT refer to the same thing.

Hint for DP problems

- For a DP problem you can typically write a MATH function that gives the solution for problem of size N in terms of smaller problems.
- It is straightforward to go from this math function to code:
  - Iterative: The math function ‘maps’ to the sol array
  - Recursive: The math function ‘maps’ to recursive calls
- Typically the math function will be a
  - Min/max (over itself applied to smaller N)
  - Sum (over itself applied to smaller N)

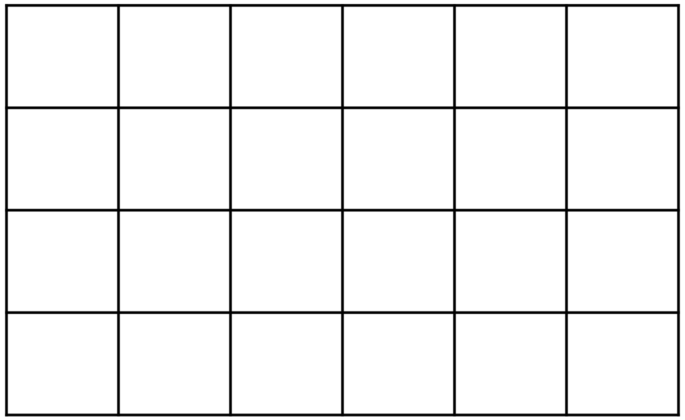
2D Matrix Traversal

P1. All possible ways to traverse a 2D matrix.

- Start from top left corner and reach bottom right corner.
- You can only move: 1 step to the right or one step down at a time. (No diagonal moves).
- Variation: Allow to move in the diagonal direction as well.
- Variation: Add obstacles (cannot travel through certain cells).

P2. Add fish of various gains. Take path that gives the most gain.

- Variation: Add obstacles.



Other DP Problems

- Rod cutting
- Stair climbing
- Make amount with smallest number of coins
- Matrix with gain
- House robber
- Many more on leetcode.

Application of the Knapsack problem

- https://en.wikipedia.org/wiki/Knapsack_problem

One early application of knapsack algorithms was in the construction and scoring of tests in which the test-takers have a choice as to which questions they answer. For small examples, it is a fairly simple process to provide the test-takers with such a choice. For example, if an exam contains 12 questions each worth 10 points, the test-taker need only answer 10 questions to achieve a maximum possible score of 100 points. However, on tests with a heterogeneous distribution of point values, it is more difficult to provide choices. Feuerman and Weiss proposed a system in which students are given a heterogeneous test with a total of 125 possible points. The students are asked to answer all of the questions to the best of their abilities. Of the possible subsets of problems whose total point values add up to 100, a knapsack algorithm would determine which subset gives each student the highest possible score