04-630
Data Structures and Algorithms for Engineers

David Vernon
Carnegie Mellon University Africa

vernon@cmu.edu
www.vernon.eu

Lecture 23

Algorithmic Strategies

- Classes of algorithms
- Brute force
- Divide and conquer
- Greedy algorithms
- Dynamic programming
- Combinatorial search and backtracking
- Branch and bound

Brute Force

- Brute force is a straightforward approach to solve a problem based on a simple formulation of problem
- Often without any deep analysis of the problem
- Perhaps the easiest approach to apply and is useful for solving small-size instances of a problem
- May result in naïve solutions with poor performance

Brute Force

- Some examples of brute force algorithms are:
  - Computing a^n ($a > 0$, n a non-negative integer) by repetitive multiplication: a x a x … x a
    - For a more efficient approach, see https://en.wikipedia.org/wiki/Exponentiation_by_squaring
  - Computing n! by repetitive multiplication: n x n-1 x n-2, …
    - For more efficient approaches, see http://www.luschny.de/math/factorial/FastFactorialFunctions.htm
  - Sequential (linear) search
  - Selection sort, Bubble sort

Brute Force

- Maximum sub-array problem / Grenander's Problem
  - Given a sequence of integers $i_1, i_2, \ldots, i_n$, find the sub-sequence with the maximum sum
    - If all numbers are negative the result is 0
  - Examples:

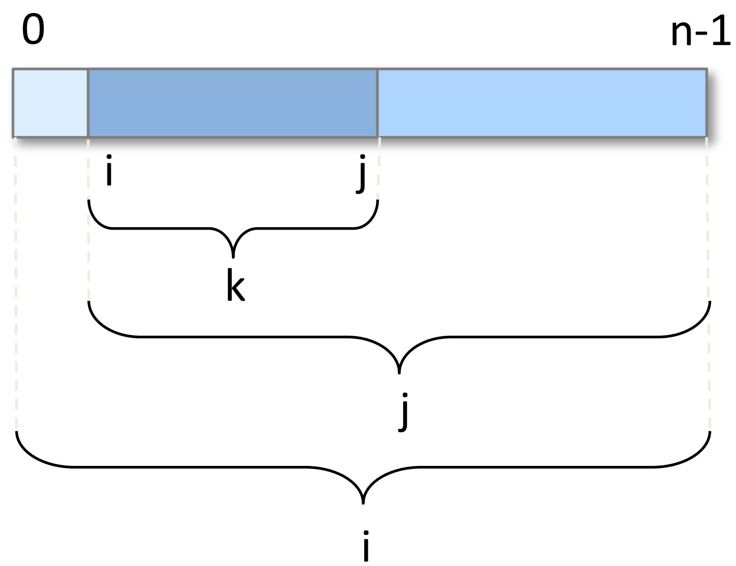
    -2, 11, -4, 13, -4, 2 has the solution 20

    1, -3, 4, -2, -1, 6 has the solution 7

Brute Force

- Maximum subarray problem: brute force solution $O(n^3)$

```
int grenanderBF(int a[], int n) {
    int maxSum = 0;
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            int thisSum = 0;
            for (int k = i; k <= j; k++) {
                thisSum += a[ k ];
            }
            if (thisSum > maxSum) {
                maxSum = thisSum;
            }
        }
    }
    return maxSum;
}
```



Brute Force

- Maximum sub-array problem
  - Divide and Conquer algorithm $O(n \log n)$
  - Kadane's algorithms $O(n)$ … dynamic programming

Divide and Conquer

- Divide-and conquer (D&Q)
  - Given an instance of the problem
  - Divide this into smaller sub-instances (often two)
  - Independently solve each of the sub-instances
  - Combine the sub-instance solutions to yield a solution for the original instance
- With the D&Q method, the size of the problem instance is reduced by a factor (e.g. half the input size)

Divide and Conquer

- Often yield a recursive formulation

- Examples of D&Q algorithms

  - Quicksort algorithm

  - Mergesort algorithm

  - Fast Fourier Transform

Divide and Conquer

Mergesort

UNSORTEDSEQUENCE

UNSORTED SEQUENCE

UNSO RTED SEQU ENCE

UN SO RT ED SE QU EN CE

NU OS RT DE ES QU EN CE

NOSU DERT EQSU CEEN

DENORSTU CEEENQSU

CDEEEEENNOQRSSTUU

Divide and Conquer

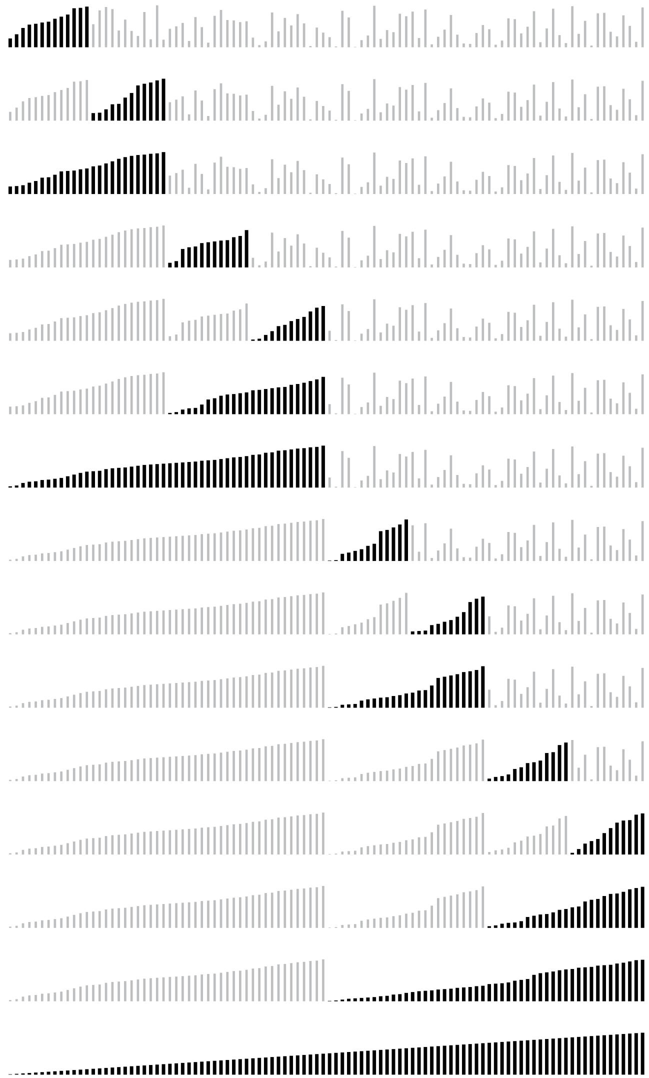
```
void mergesort(Item a[], int l, int r) {
    if (r-l <= 1) {
        return;
    } else {
        int m = (r + l) / 2;
        mergesort(a, l, m);
        mergesort(a, m+1, r);
        merge(a, l, m, r);
    }
}

void mergesort(Item a[], int size) {
    mergesort(a, 0, size-1);
}
```

Already sorted?

Divide the list into two equal parts

Sort the two halves recursively

Merge the sorted halves into a sorted whole



Divide and Conquer

```
int grenanderDQ(int a[], int l, int h) {
    if (l > h) return 0;
    if (l = h) return max(0, a[l]);
    int m = (l + h) / 2;
    int sum = 0;
    int maxLeft = 0;
    for (int i = m; i >= l; i--) {
        sum += a[i];
        maxLeft = max(maxLeft, sum);
    }
    sum = 0;
    int maxRight = 0;
    for (int i = m + 1; i <= h; i++) {
        sum += a[i];
        maxRight = max(maxRight, sum);
    }
    int maxL = grenanderDQ(a, l, m);
    int maxR = grenanderDQ(a, m+1, h);
    int maxC = maxLeft + maxRight;
    return max(maxC, max(maxL, maxR));
}
```

Divide the problem

Solve the sub-problem

Solve the sub-problem

Solve the sub-problems

Combine the solutions

Divide and Conquer

```
// Generic Divide and Conquer Algorithm

divideAndConquer(Problem p) {
   if (p is simple or small enough) {
      return simpleAlgorithm(p);
   } else {
      divide p in smaller instances p1, p2, ..., pn
      Solution solutions[n];
      for (int i = 0; i < n; i++) {
         solutions[i] = divideAndConquer(pi);
      }
      return combine(solutions);
   }
}
```

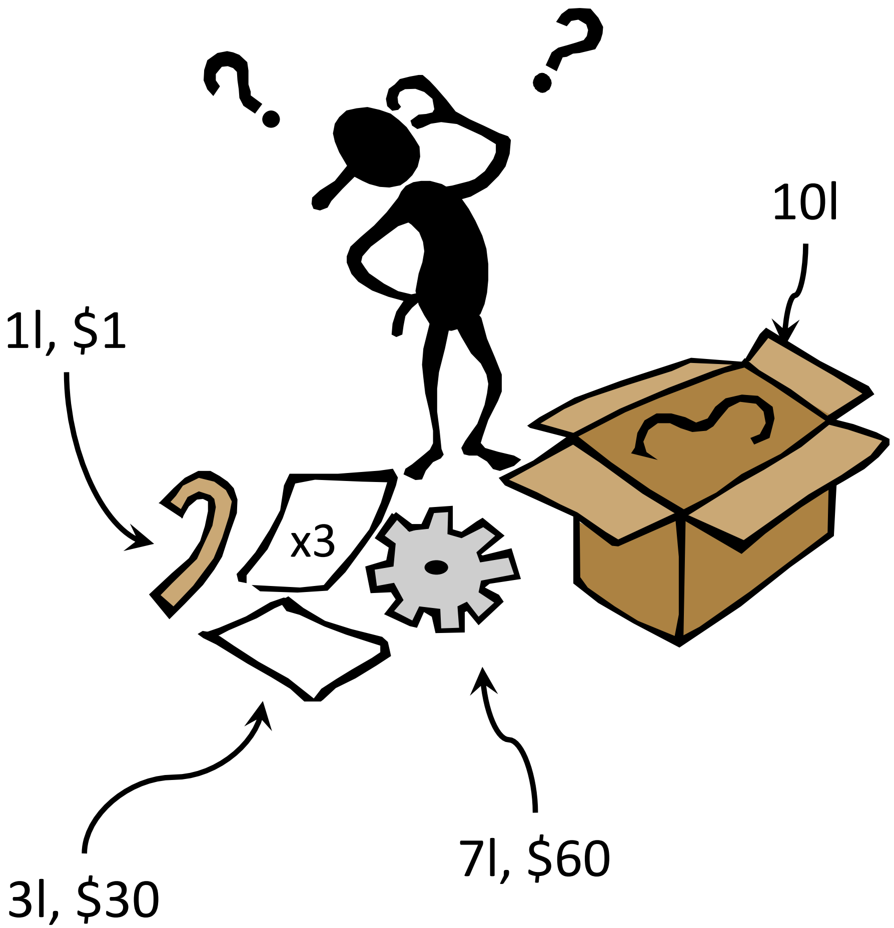
Greedy Algorithms

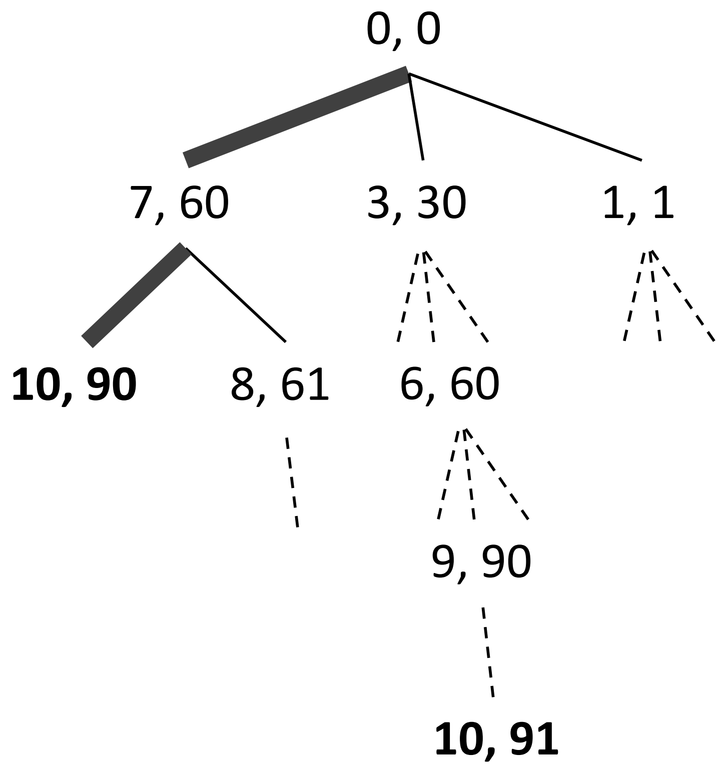
- Try to find solutions to problems step-by-step
  - A partial solution is incrementally expanded towards a complete solution
  - In each step, there are several ways to expand the partial solution:
  - The best alternative for the moment is chosen, the others are discarded
- At each step the choice must be **locally optimal** – this is the central point of this technique

Greedy Algorithms

- Examples of problems that can be solved using a greedy algorithm:
  - Finding the minimum spanning tree of a graph (Prim's algorithm)
  - Finding the shortest distance in a graph (Dijkstra's algorithm)
  - Using Huffman trees for optimal encoding of information
  - The Knapsack problem

Greedy Algorithms





Dynamic Programming

- Dynamic programming is similar to D&Q
  - Divides the original problem into smaller sub-problems
- Sometimes it is hard to know beforehand which sub-problems are needed to be solved in order to solve the original problem
- Dynamic programming solves a large number of sub-problems
- … and uses *some* of the sub-solutions to form a solution to the original problem

Dynamic Programming

- In an optimal sequence of choices, actions or decisions each sub-sequence must also be optimal:
  - An optimal solution to a problem is a combination of optimal solutions to some of its sub-problems
  - Not all optimization problems adhere to this principle

Dynamic Programming

- One disadvantage of using D&Q is that the process of recursively solving separate sub-instances can result in **the same computations being performed repeatedly**

- The idea behind dynamic programming is to avoid calculating the same quantity twice, usually by **maintaining a table of sub-instance results**

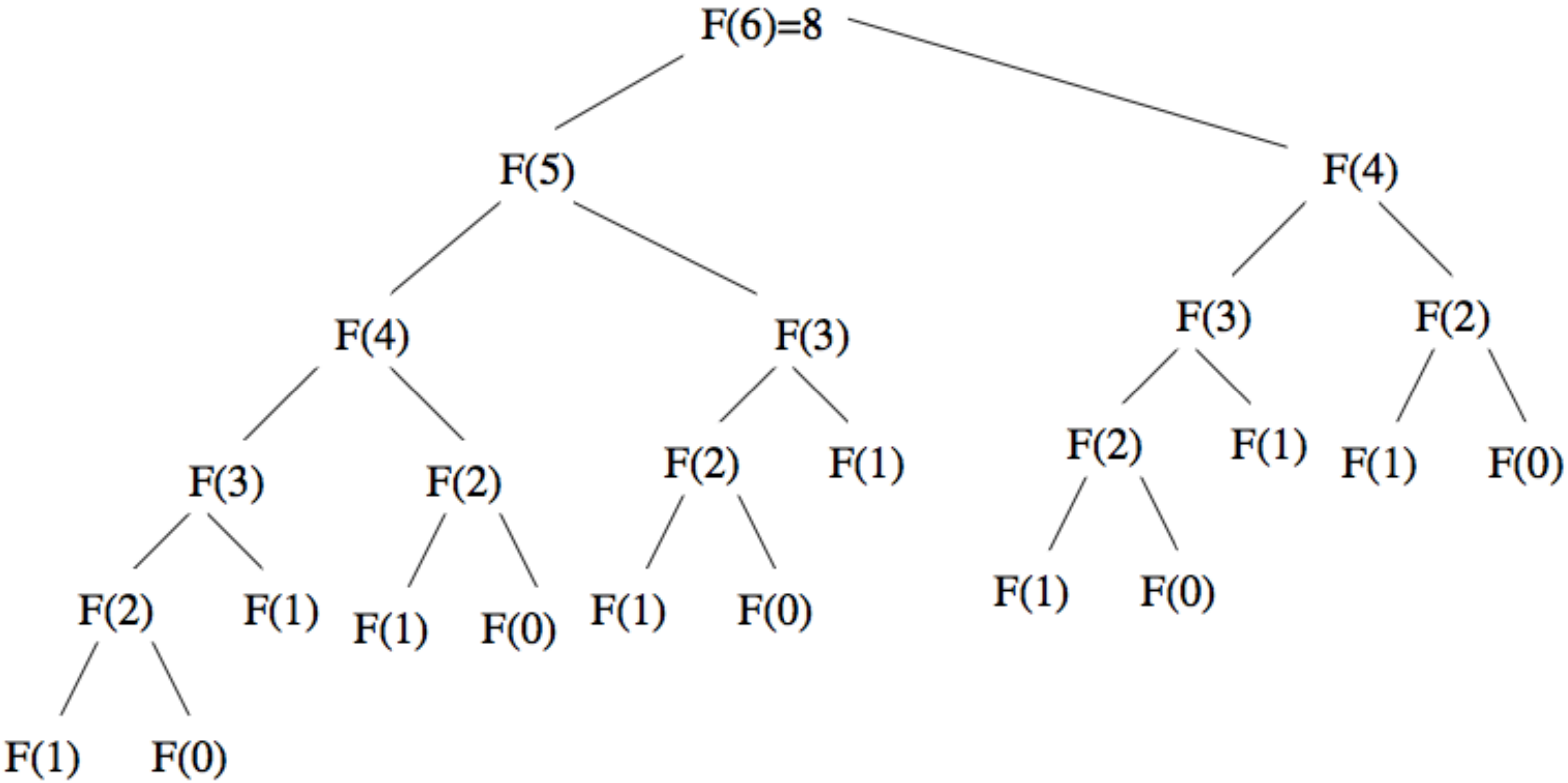
Dynamic Programming

- The same sub-problems may reappear
- To avoid solving the same sub-problem more than once, sub-results are saved in a data structure that is updated dynamically
- Sometimes the result structure (or parts of it) may be computed beforehand

Dynamic Programming

```
/* fibonacci by recursion O(1.618^n) time complexity */

long fib_r(int n) {
   if (n == 0)
      return(0);
   else
      if (n == 1)
         return(1);
      else
         return(fib_r(n-1) + fib_r(n-2));
}


fib_r(4)  → fib(3) + fib(2)
          → fib(2) + fib(1) + fib(2)
          → fib(1) + fib(0) + fib(1) + fib(2)
          → fib(1) + fib(0) + fib(1) + fib(1) + fib(0)
```

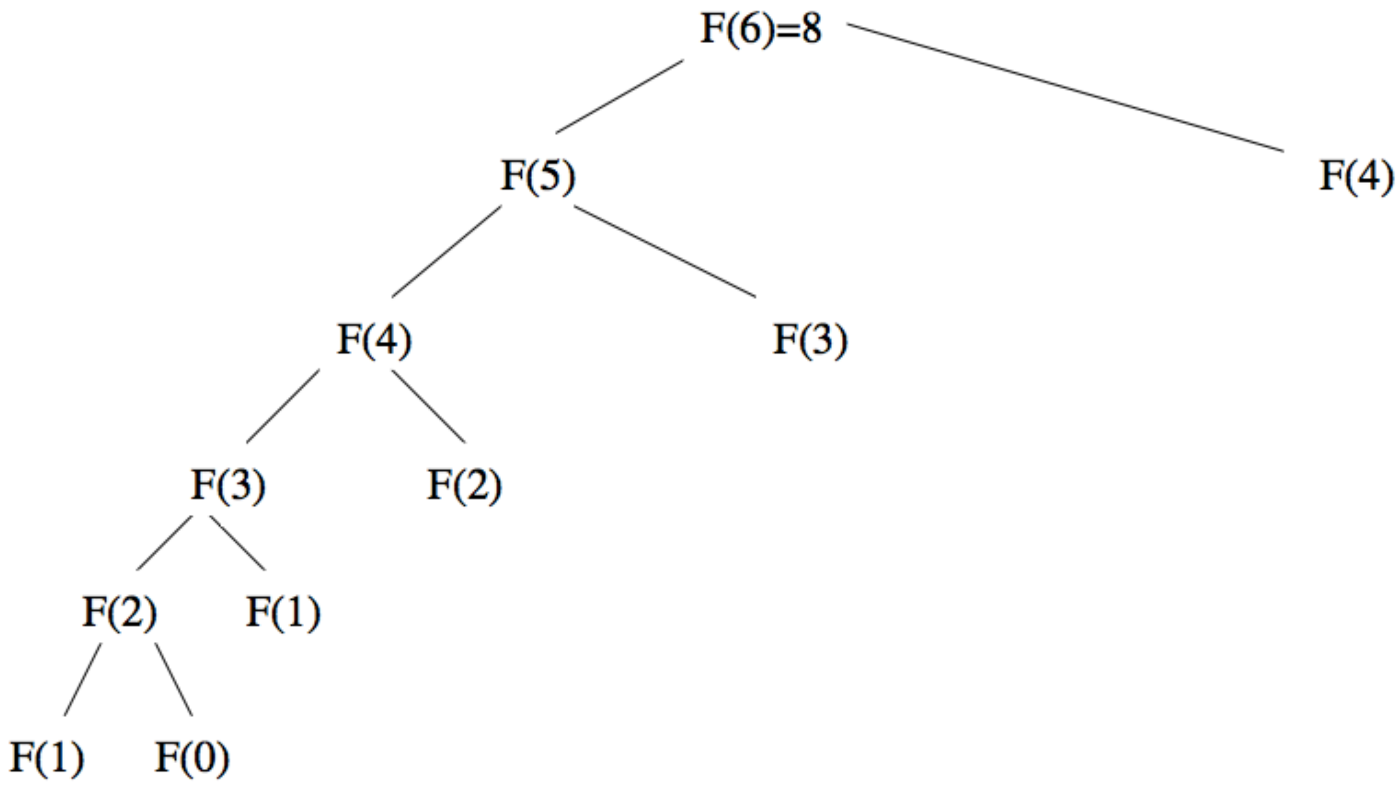
Dynamic Programming



Dynamic Programming

```
#define MAXN 45      /* largest interesting n                     */
#define UNKNOWN -1   /* contents denote an empty cell             */
long f[MAXN+1];      /* array for caching computed fib values     */

/* fibonacci by caching: O(n) storage & O(n) time                 */

long fib_c(int n) {
   if (f[n] == UNKNOWN)
      f[n] = fib_c(n-1) + fib_c(n-2);
   return(f[n]);
}

long fib_c_driver(int n) {
   int i; /* counter */

   f[0] = 0;
   f[1] = 1;
   for (i=2; i<=n; i++)
      f[i] = UNKNOWN;
   return(fib_c(n));
}
```

Dynamic Programming



Dynamic Programming

```
/* fibonacci by dynamic programming: cache & no recursion           */
/* NB: need correct order of evaluation in the recurrence relation  */
/* O(n) storage & O(n) time                                         */


long fib_dp(int n) {
   int i; /* counter */
   long f[MAXN+1]; /* array to cache computed fib values */

   f[0] = 0;
   f[1] = 1;

   for (i=2; i<=n; i++)
      f[i] = f[i-1]+f[i-2];

   return(f[n]);
}
```

Dynamic Programming

```
/* fibonacci by dynamic programming: minimal cache & no recursion        */
/* O(1) storage & O(n) time                                               */

long fib_ultimate(int n) {

   int i;                  /* counter */
   long back2=0, back1=1;  /* last two values of f[n] */
   long next;              /* placeholder for sum */

   if (n == 0) return (0);

   for (i=2; i<n; i++) {
      next = back1+back2;
      back2 = back1;
      back1 = next;
   }
   return(back1+back2);
}
```

Dynamic Programming

```
int grenanderDP(int a[], int n) {
   int table[n+1];
   table[0] = 0;
   for (int k = 1; k <= n; k++)
      table[k] = table[k-1] + a[k-1];
   int maxSoFar = 0;
   for (int i = 1; i <= n; i++)
      for (int j = i; j <= n; j++) {
         thisSum = table[ j ] - table[i-1];
         if (thisSum > maxSoFar)
            maxSoFar = thisSum;
      }
   return maxSoFar;
}
```

Dynamic Programming

- There are three steps involved in solving a problem by dynamic programming:

    1. Formulate the answer as a recurrence relation or recursive algorithm

    2. Show that the number of different parameter values taken on by your recurrence is bounded by a (hopefully small) polynomial

    3. Specify an order of evaluation for the recurrence so the partial results you need are always available when you need them

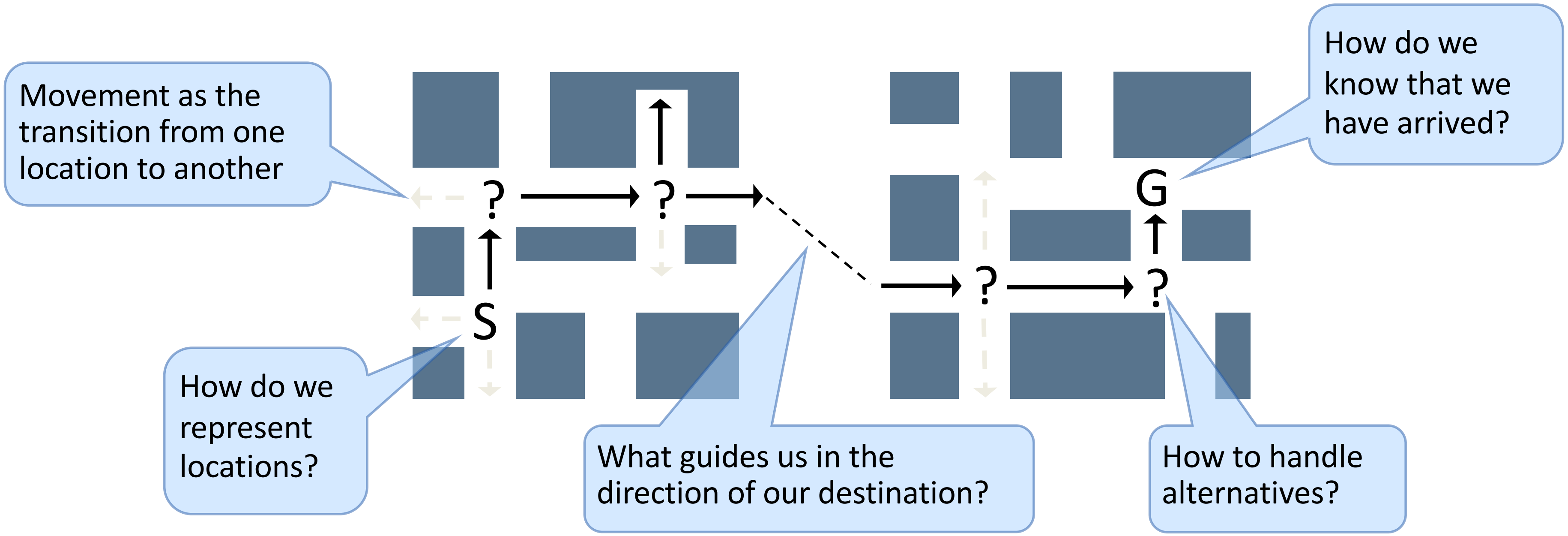
Combinatorial Search / State Space Search

- We can find optimal solutions to many problems using exhaustive search technique
  - However, the complexity can be huge so we need to be careful
  - If the complexity is $O(2^n)$ it will be feasible to consider problems where $n < 40$
  - If the complexity is $O(n!)$ it will be feasible to consider problems where $n < 20$

Combinatorial Search / State Space Search

- Solving problems through the **systematic search** for solutions in a (large) **state space**

- The general idea is to incrementally **extend partial solutions until a complete solution is obtained**



Combinatorial Search / State Space Search

- Search is the systematic process of
  - choosing one of many possible alternatives,
  - saving the rest in case the alternative selected first does not lead to the goal
- Search can be viewed as the construction and traversal of **search trees**

Combinatorial Search / State Space Search

- Characterization of the state space
  - The **initial state** (e.g., a location)
  - A **set of operators** which take us from one state to another state (e.g., drive straight, turn left, …)
  - A **goal-test** which decides when the goal is reached (e.g., comparing locations)
    - Explicit states (e.g., a specific address)
    - Abstractly described states (e.g., any post office)

Combinatorial Search / State Space Search

- Characterization of the state space
  - A **description of a solution** (e.g., the address, the path between locations or the moves used)
    - The search path (e.g., the shortest path between your home and your office)
    - Just the final state (e.g., the post office)

Combinatorial Search / State Space Search

- Characterization of the state space
  - A **cost function** (e.g., time, money, distance or number of moves): *true cost* for going from start to where we are now + *estimated cost* for going from we are now to the nearest goal
    - Search cost, the cost for concluding that a certain operator should be used (e.g., the time it takes to ask someone for directions or thinking about a move) +
    - Path cost, the cost for using a operator (e.g., the energy it takes to walk or time)

Combinatorial Search / State Space Search

- Reminder of the potential size of state spaces

- Propositional satisfiability problem (SAT):

  – Decide if there is an assignment to the variables of a propositional formula that satisfies it:

$$f = (\bar{x}_2 + \bar{x}_4)(x_3 + x_4)(\bar{x}_3 + x_4)(x_1 + x_2)(\bar{x}_1 + \bar{x}_3)$$

  – 100 variables → 2^{100} ~ 10^{30} combinations
  1000 evaluations/second →
  31,709,791,983,764,586,504 years required to evaluate all combinations

Combinatorial Search / State Space Search

- Reminder of the potential size of state spaces

- Traveling salesman problem (TSP)

  – Given a number of cities along with the cost of travel between each pair of them, find the cheapest way of visiting all the cities exactly once and returning to the starting point

  – There are 2 identical tours for each permutation of n cities → the number of tours are $n!/(2n) = (n-1)!/2$ ...

    - divide by n if we don't care where we start
    - divide by 2 if we don't care which direction we take the tour

  – A 50-city TSP therefore has about **$3*10^{62}$ potential solutions**

Backtracking

- A systematic method to iterate through all the possible configurations of a search space

  - All possible arrangements of object: permutations
  - All possible ways of building a collection of objects: subsets
  - Generation of all possible spanning trees of a graph
  - Generation of all possible paths between two vertices
  - …

- Exhaustive search … check each solution generated to see if is the required solution (satisfies some optimality criterion)

- General technique

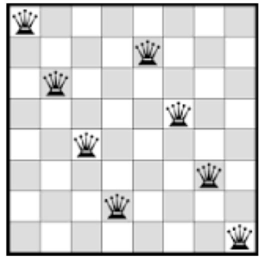
  - Must be customized for each individual application

Backtracking

- Based on the construction of a state space tree
  - nodes represent states,
  - root represents the initial state
  - one or more leaves are goal states
  - each edge represents the application of an operator
- The solution is found by expanding the tree until a goal state is found

Backtracking

- Examples of problems that can be solved using backtracking:
  - Puzzles (e.g., eight queens puzzle, crosswords, Sudoku)



  - Combinatorial optimization problems (e.g., parsing and layout problems)
  - Logic programming languages such as Icon, Planner and Prolog, which use backtracking internally to generate answers

Backtracking

- Generate each possible configuration exactly once
- Avoiding repetitions and not missing configurations means we must define a systematic generation order
- Let the solution be a vector

  $\boldsymbol{a} = (a_1, a_2, \ldots a_\mathrm{n})$

  where each element is selected from a finite ordered set S_i

  - For example, $\boldsymbol{a}$ might represent a permutation and a_i might be the $\boldsymbol{i}^{th}$ element of the permutation
  - For example, $\boldsymbol{a}$ might be a subset S, and a_i would be true if and only if the $\boldsymbol{i}^{th}$ element of the universal set is in S
  - For example, $\boldsymbol{a}$ might be a sequence of moves in a game or a path in a graph, where a_i contains the $\boldsymbol{i}^{th}$ event in the sequence

Backtracking

– At each step, start from a partial solution

  $\boldsymbol{a} = (a_1, a_2, \ldots a_n)$

– Try to extend it by adding another element at the end

– After extending, test whether what we have so far is a solution

– If it is, use it (e.g. check to see if it's the best solution so far)

– If it isn't, check to see whether it can be extended to form a complete solution

– If it can, continue with recursion

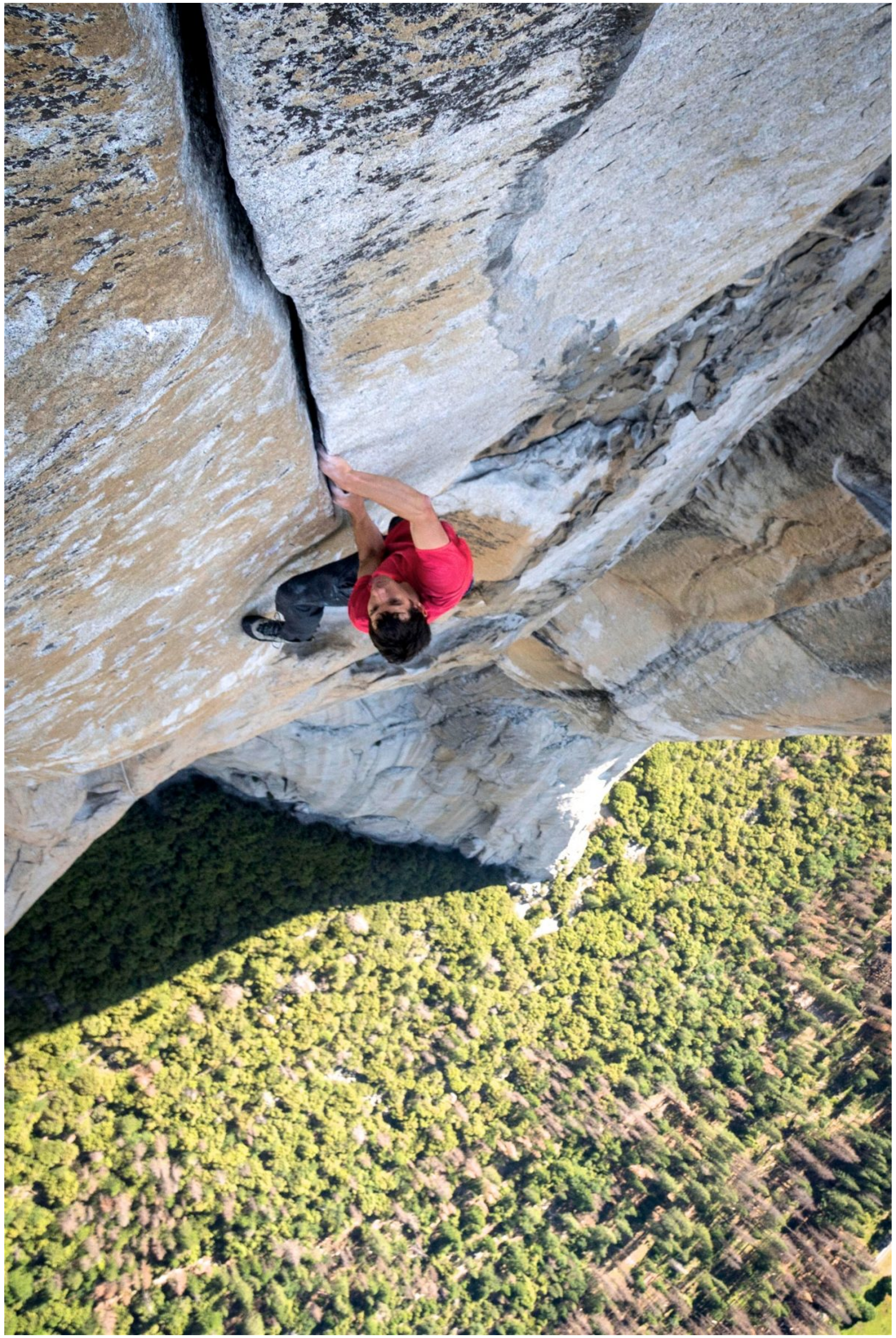
– If it can't, delete the last element from a and try another possibility from that position if it exists

Backtracking

- Backtracking constructs a tree of partial solutions
  - Each vertex represents one partial solution
  - There is an edge from one node x to node y if node y was created by advancing from x
  - Constructing the solutions can be viewed as doing a dept-first traversal of the backtrack tree
- Backtracking ensures correctness by enumerating all possibilities
- Backtracking ensures efficiency by never visiting a state more than once

https://www.youtube.com/watch?v=urRVZ4SW7WU

https://www.mensjournal.com/adventure/alex-honnold-on-his-free-solo-ascent-of-yosemites-el-capitan-w486186/



https://theknow.denverpost.com/2018/09/27/alex-honnold-climbing-tips/196513/

Backtracking

- Backtracking as a depth-first traversal

Backtrack-DFS(A, k)
    if $A = (a_1, a_2, ..., a_k)$ is a solution, report it.
    else
        $k = k + 1$
        compute S_k
        while $S_k \neq \emptyset$ do
            $a_k =$ an element in S_k
            $S_k = S_k - a_k$
            Backtrack-DFS(A, k)

Backtracking

```
bool finished = FALSE; /* found all solutions yet? */

backtrack(int a[], int k, data input) {

   int c[MAXCANDIDATES]; /* candidates for next position  */
   int ncandidates;      /* next position candidate count */
   int i;                /* counter                       */

   if (is_a_solution(a,k,input))
      process_solution(a,k,input);
   else {
      k = k+1; // k==1 => we need to choose a1, ...
      construct_candidates(a,k,input,c,&ncandidates);
      for (i=0; i<ncandidates; i++) {
         a[k] = c[i];
         make_move(a,k,input);
         backtrack(a,k,input);
         unmake_move(a,k,input);
        if (finished) return;  /* terminate early */
      }
   }
}
```

Backtracking

- Note how recursion yields an elegant and easy implementation of the backtracking algorithm
  - The new candidates array c is allocated with each recursive procedure call
  - Consequently, the not-yet-considered extension candidates at each position don't interfere with each other
- The application-specific parts are dealt with in functions

```
1. is_a_solution(a,k,input)
2. construct_candidates(a,k,input,c,&ncandidates)
3. process_solution(a,k,input)
4. make_move(a,k,input)
5. unmake_move(a,k,input)
```

Backtracking

`is_a_solution(a,k,input)`

- Boolean function
- Tests whether the first k elements of vector `a` form a complete solution for the given problem
- The argument `input` allows us to pass general information to the routine
- We could used it to specify n, the size of a target solution, e.g. when constructing permutations or subsets of n elements

Backtracking

`construct_candidates(a,k,input,c,&ncandidates)`

- Fills an array `c` with the complete set of possible candidates for the k^{th} position of `a`, given the contents of the first k-1 positions

- The number of candidates returned in this array is given by `ncandidates`

- Again, `input` may be used to pass auxiliary information

Backtracking

`process_solution(a,k,input)`

– Prints, counts, or otherwise processes a complete solution once it is constructed

Backtracking

```
make_move(a,k,input)
unmake_move(a,k,input)
```

- These functions enable us to modify a data structure in response to the latest move

- or clean up this data structure if we decide to take back the move

- You could build such a data structure from scratch from the solution `a` if required but it can be more efficient to do it this way if the changes involved in a move can be easily undone

Backtracking

- Many combinatorial optimization problems require the enumeration of all subsets / permutations of some set (and testing each enumeration for optimality / success)

- Being able to compute the number of subset / permutations is far easier than enumerating them

  – There are $n!$ permutations of n elements

  – There are 2^n subsets of n elements

- Recall earlier comments on the exponential size of a state space

Backtracking

- To construct all $n!$ permutations
  - Set up an integer array a of n cells
  - The set of candidates for the i^{th} element will be the set of elements that have not appeared in the $(i\text{-}1)$ elements of the partial solution, corresponding to the first elements of the i-1 permutation
  - In terms of our general backtrack algorithm

    $S_k = \{1, \ldots, n\} - a$

    a is a solution whenever $k = n$

Backtracking

```
/* Construct all permutations                                        */

bool is_a_solution(int a[], int k, int n) {
        return (k == n);
}

void construct_candidates(int a[], int k, int n, int c[], int *ncandidates) {
   int i;                          /* counter */
   bool in_perm[NMAX];             /* who is in the permutation? */

   for (i=1; i<NMAX; i++) in_perm[i] = FALSE;

   // we are finding candidates for a_k, a_k+1, ... a_n
   // when k == 1, all candidates are valid because we haven't selected any yet
   // when k == 2, all candidates except a_1 are valid
   // when k == n, all candidates except a_1 .. a_n-1 are valid
   for (i=1; i<k; i++) in_perm[ a[i] ] = TRUE;

   *ncandidates = 0;
   for (i=1; i<=n; i++)
      if (in_perm[i] == FALSE) {
         c[ *ncandidates] = i;
         *ncandidates = *ncandidates + 1;
      }
}
```

NMAX must be the number of elements in the permutation + 1 to allow for counting from 1, rather than 0

Backtracking

```
void process_solution(int a[], int k, data input) {

   int i;                              /* counter */

   for (i=1; i<=k; i++) printf(" %d",a[i]);

   printf("\n");
}


void generate_permutations(int n){
        int a[NMAX];

        backtrack(a,0,n);
}
```

Backtracking

```
#define TRUE  1
#define FALSE 0

backtrack(a,0,3)
   k: 1
   i: 0
```

in_perm		F	F	F	F	F	F	F
c	1	2	3					
a		1						

```
      backtrack(a,1,3)
         k: 2
         i: 0
```

in_perm		T	F	F	F	F	F	F
c	2	3						
a		1	2					

```
         backtrack(a,2,3)
            k: 3
            i: 0
            backtrack(a,3,3)
            -> process_solution(a,3,3): 1 2 3
```

in_perm		T	T	F	F	F	F	F
c	3							
a		1	2	3				

```
         k: 2
         i: 1
```

in_perm		T	F	F	F	F	F	F
c	2	3						
a		1	3					

```
         backtrack(a,2,3)
            k: 3
            i: 0
            backtrack(a,3,3)
            -> process_solution(a,3,3): 1 3 2
```

in_perm		T	F	T	F	F	F	F
c	2							
a		1	3	2				

When studying this walkthrough, remember that the variable `i` iterates through all the candidate digits (at each level of recursion) and the variable `k` identifies the position in the permutation that is currently being filled.

Backtracking

```
#define TRUE  1
#define FALSE 0
```

```
backtrack(a,0,3)
   k: 1
   i: 1
```

in_perm		F	F	F	F	F	F	F
c	1	2	3					
a		2						

```
   backtrack(a,1,3)
      k: 2
      i: 0
```

in_perm		F	T	F	F	F	F	F
c	1	3						
a		2	1					

```
      backtrack(a,2,3)
         k: 3
         i: 0
         backtrack(a,3,3)
         -> process_solution(a,3,3): 2 1 3
```

in_perm		T	T	F	F	F	F	F
c	3							
a		2	1	3				

```
      k: 2
      i: 1
```

in_perm		T	F	F	F	F	F	F
c	1	3						
a		2	3					

```
      backtrack(a,2,3)
         k: 3
         i: 0
         backtrack(a,3,3)
         -> process_solution(a,3,3): 2 3 1
```

in_perm		F	T	T	F	F	F	F
c	1							
a		2	3	1				

When studying this walkthrough, remember that the variable `i` iterates through all the candidate digits (at each level of recursion) and the variable `k` identifies the position in the permutation that is currently being filled.

Backtracking

```
#define TRUE  1
#define FALSE 0

backtrack(a,0,3)
   k: 1
   i: 2
```

in_perm		F	F	F	F	F	F	F
c	1	2	3					
a		3						

```
   backtrack(a,1,3)
      k: 2
      i: 0
```

in_perm		F	F	T	F	F	F	F
c	1	2						
a		3	1					

```
      backtrack(a,2,3)
         k: 3
         i: 0
         backtrack(a,3,3)
         -> process_solution(a,3,3): 3 1 2
```

in_perm		T	F	T	F	F	F	F
c	2							
a		3	1	2				

```
      k: 2
      i: 1
```

in_perm		F	F	T	F	F	F	F
c	1	2						
a		3	2					

```
      backtrack(a,2,3)
         k: 3
         i: 0
         backtrack(a,3,3)
         -> process_solution(a,3,3): 3 2 1
```

in_perm		F	T	T	F	F	F	F
c	1							
a		3	2	1				

When studying this walkthrough, remember that the variable `i` iterates through all the candidate digits (at each level of recursion) and the variable `k` identifies the position in the permutation that is currently being filled.

Backtracking

- To construct all 2^n subsets
  - Set up an Boolean array a of n cells
  - Element a_i signifies whether the i^{th} element of the set is in the subset
  - In terms of our general backtrack algorithm

    S_k = (true, false)
    a is a solution whenever $k = n$

Backtracking

```
bool finished = FALSE; /* found all solutions yet? */

backtrack(int a[], int k, data input) {

   int c[MAXCANDIDATES]; /* candidates for next position  */
   int ncandidates;      /* next position candidate count */
   int i;                /* counter                       */

   if (is_a_solution(a,k,input))
      process_solution(a,k,input);
   else {
      k = k+1; // k==1 => we need to choose a1, ...
      construct_candidates(a,k,input,c,&ncandidates);
      for (i=0; i<ncandidates; i++) {
         a[k] = c[i];
         backtrack(a,k,input);
         if (finished) return;  /* terminate early */
      }
   }
}
```

Backtracking

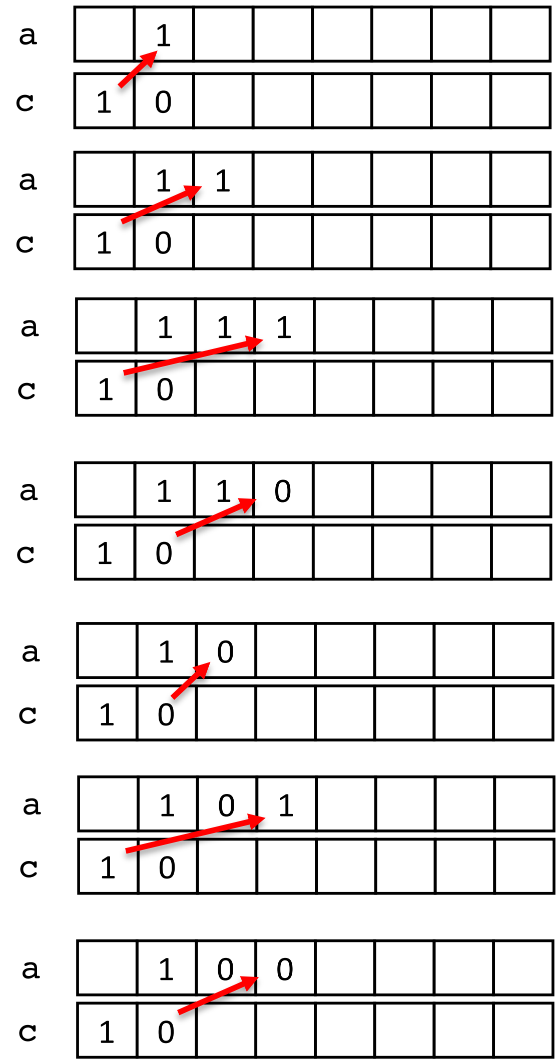
```
/* Construct all subsets                                                  */

bool is_a_solution(int a[], int k, int n) {
   return (k == n);                    /* is k == n? */
}

void construct_candidates(int a[], int k, int n, int c[], int *ncandidates) {
   c[0] = TRUE;
   c[1] = FALSE;
   *ncandidates = 2;
}
```

Backtracking

```
/* Construct all subsets                                              */

void process_solution(int a[], int k) {
   int i;  /* counter */

   printf("{");
   for (i=1; i<=k; i++)
      if (a[i] == TRUE) printf(" %d",i);

   printf(" }\n");
}

void generate_subsets(int n) {
        int a[NMAX];
        backtrack(a,0,n); /* solution vector */
}
```

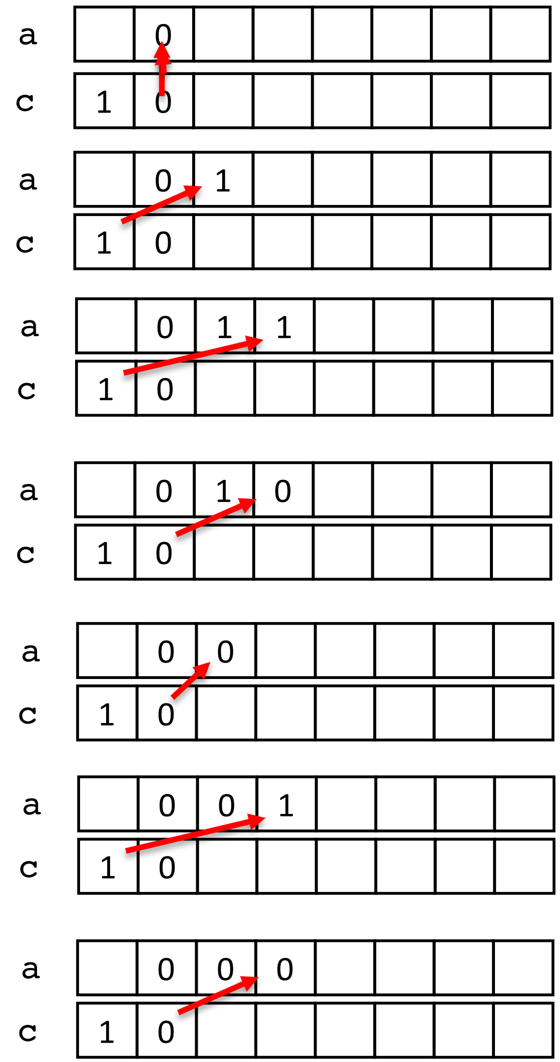
Backtracking

```
#define TRUE  1
#define FALSE 0

backtrack(a,0,3)
   k: 1
   i: 0

   backtrack(a,1,3)
      k: 2
      i: 0

      backtrack(a,2,3)
         k: 3
         i: 0
         backtrack(a,3,3) -> process_solution(a,3,3): {1 2 3}

         k: 3
         i: 1
         backtrack(a,3,3) -> process_solution(a,3,3): {1 2}

      k: 2
      i: 1

      backtrack(a,2,3)
         k: 3
         i: 0
         backtrack(a,3,3) -> process_solution(a,3,3): {1 3}

         k: 3
         i: 1
         backtrack(a,3,3) -> process_solution(a,3,3): {1}
```



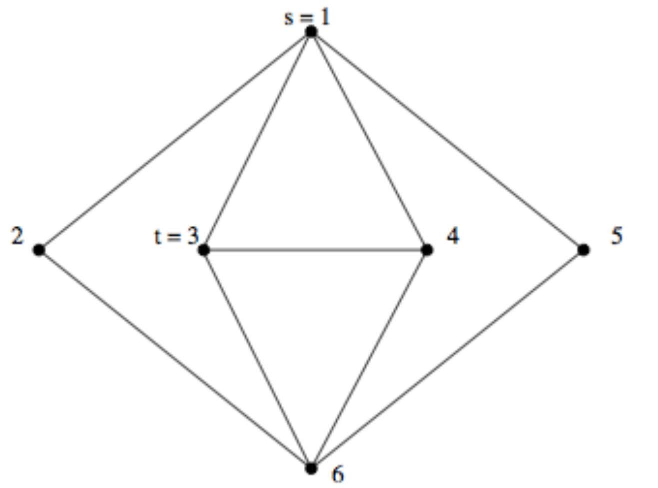
Backtracking

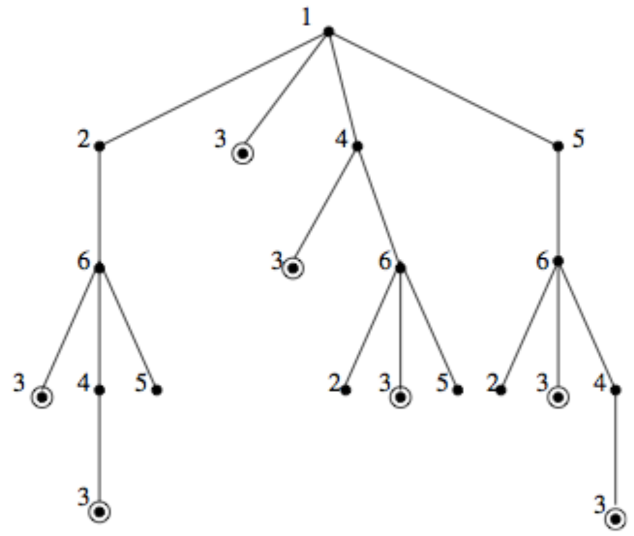
```
#define TRUE  1
#define FALSE 0

backtrack(a,0,3)
   k: 1
   i: 1

   backtrack(a,1,3)
      k: 2
      i: 0

      backtrack(a,2,3)
         k: 3
         i: 0
         backtrack(a,3,3) -> process_solution(a,3,3): {2 3}

         k: 3
         i: 1
         backtrack(a,3,3) -> process_solution(a,3,3): {2}

      k: 2
      i: 1

      backtrack(a,2,3)
         k: 3
         i: 0
         backtrack(a,3,3) -> process_solution(a,3,3): {3}

         k: 3
         i: 1
         backtrack(a,3,3) -> process_solution(a,3,3): {}
```



Backtracking

- To construct all paths in a graph
  - More complicated than listing permutations or subsets
  - No explicit formula that counts the number of solutions as a function of the number of edges or vertices (it depends on the structure of the graph)





Backtracking

```
/* Construct all paths in a graph                              */

void construct_candidates(int a[], int k, int n, int c[], int *ncandidates) {
   int i;              /* counters                           */
   bool in_sol[NMAX]; /* what's already in the solution? */
   edgenode *p;        /* temporary pointer                  */
   int last;           /* last vertex on current path        */

   for (i=1; i<NMAX; i++) in_sol[i] = false;
   for (i=1; i<k; i++) in_sol[ a[i] ] = true;

   if (k==1) {   /* always start from vertex 1 */
        c[0] = 1;
        *ncandidates = 1;
   }
```

Backtracking

```
    else {
        *ncandidates = 0;
        last = a[k-1];    // last vertex included in solution
        p = g.edges[last];
        while (p != NULL) { // for each edge, the connected vertex is a candidate
            if (!in_sol[ p->y ]) {
                c[*ncandidates] = p->y;
                *ncandidates = *ncandidates + 1;
            }
            p = p->next;
        }
    }
}

bool is_a_solution(int a[], int k, int t){

    /* We report a successful path whenever a[k] = t */

    return (a[k] == t);
}

void process_solution(int a[], int k) {
    solution_count ++; /* count all s to t paths */
}
```

Backtracking

- Pruning
  - Backtracking ensures correctness by enumerating all possibilities
  - Enumerating all $n!$ permutations of n vertices of a graph and selecting the best one certainly yields the correct algorithm to find the optimal travelling salesman tour
    - For each permutation, check to see if the tour exists in the graph (do the edges exist?)
    - If so, add all the weights and see if it is the best solution

Backtracking

- Pruning
  - But it is very wasteful to construct all the permutations first and then analyze them later
    - For example, if the search starts at vertex v_1 and if (v_1, v_2) is not in the graph
    - The next $(n\text{-}2)!$ permutations enumerated starting with would be a complete waste of effort
    - Much better to prune the search after (v_1, v_2) and continue next with (v_1, v_3)
    - By restricting the set of next elements to reflect only moves that are legal / valid from the current partial configuration, we significantly reduce the search complexity

Backtracking

- Pruning
  - Is the technique of cutting off the search the instant we have established that a partial solution cannot be extended into a full solution
  - Combinatorial searches, when augmented with tree pruning techniques, can be used to find the optimal solution of small optimization problems
    - The actual size depends on the problem
    - Typical size limit are somewhere from 15 to 50 items

Branch-and-Bound

- In backtracking, we used depth-first search with pruning to traverse the state space
- We can achieve better performance for many problems using breadth-first search with pruning
- This approach is known as branch-and-bound
  - The implicit stack in depth first search is replaced by an explicit queue in breadth first search
  - If we use a priority queue, we have a **best-first** traversal of the state space

Branch-and-Bound

- Advantage of using breadth-first (or best-first) search:
  - When a node (i.e. a partial solution) that is judged to be promising (i.e. a possible candidate for a full solution) when it is first encountered and placed in the queue, it may no longer be promising when it is removed
  - If it is no longer promising, it is discarded and the evaluation and testing of its children (i.e. remainder of the solution) is avoided
- Branch-and-bound is by far the most widely used tool for solving large scale NP-hard combinatorial optimization problems
- However, it is an algorithm paradigm that has be be customized for each specific problem type

Branch-and-Bound

- Use bounds for the function to be optimized & the value of the current best solution to limit the search space