BM40A1500 DATA STRUCTURES AND ALGORITHMS

ALGORITHM DESIGN PRINCIPLES 1

2024

ALGORITHM DESIGN PRINCIPLES

- No one 'silver bullet' for all the problems.
  - Different problems require different techniques.

- Useful design principles
  - Greedy approach
  - Backtracking
    - Branch and bound
  - Divide and conquer
  - Dynamic programming
  - Probabilistic algorithms
    - Las Vegas algorithms
    - Monte Carlo algorithms

week 7

week 8

GREEDY APPROACH

- Algorithms that make locally optimal choices at each step
  - based on the information available at that time.
- Often does not lead to the optimal solution.
  - Depends on the problem: for some problems, a greedy algorithm producing optimal solution exists.
- Typically helps to find a reasonably good solution fast.

- Example: Task is to schedule jobs from a set of N jobs, so that the profit is maximized.
  - Each job has a deadline and profit.
  - Jobs cannot be done after the deadline has passed.

Job	Deadline	Profit	Schedule
1	3	100	
2	2	50	
3	1	20	
4	2	120	

LUT University

GREEDY APPROACH

- Greedy algorithm 1:
  1. select the remaining job with the highest profit and do it next.

$P = 100 + 120 = 220$

Job	Deadline	Profit	Schedule		
1	3	100		2	
2	2	50			
3	1	20			
4	2	120	1		

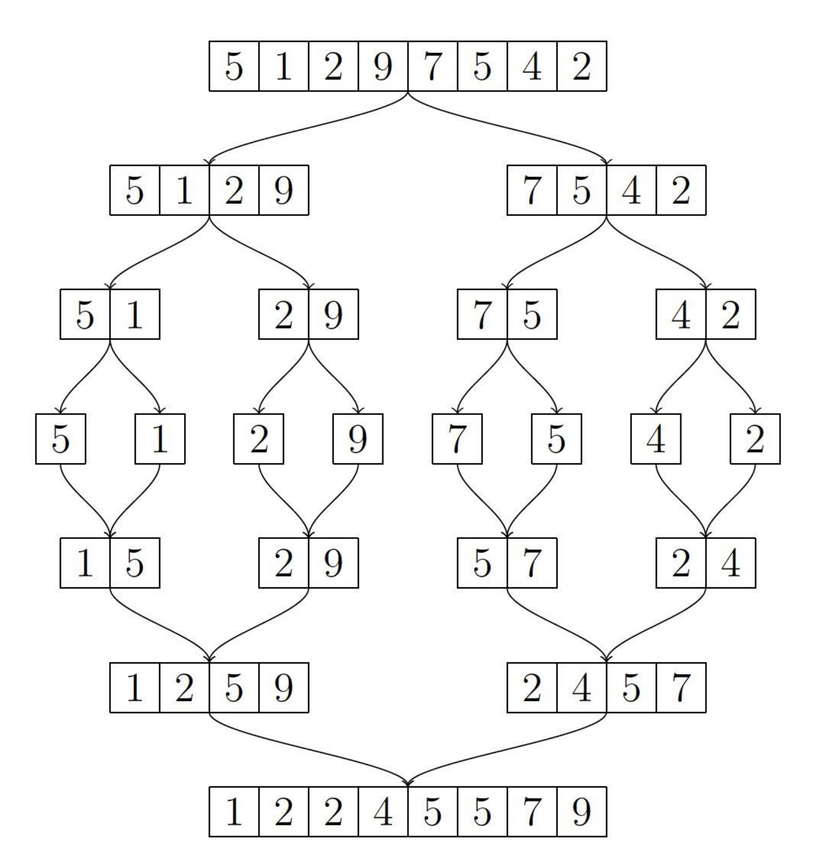
- Greedy algorithm 2:
  1. sort the jobs based on profit
  2. starting from the job with the highest profit, assign jobs to the latest free slot meeting the deadline.

$P = 270$

Job	Deadline	Profit	Schedule		
4	2	120		2	
1	3	100			3
2	2	50	1		
3	1	20			

DIVIDE AND CONQUER: MERGE SORT

- **Divide and conquer:** A solution is found by breaking the problem into smaller (similar) subproblems, solving the subproblems, then combining the subproblem solutions to form the solution to the original problem.
- Example 1: Merge sort
  - Split the list in half, sort the halves, and then merge the sorted halves together.
  - Can be implemented recursively.



Source: Laaksonen, Tietorakenteet ja algoritmit

DIVIDE AND CONQUER: QUICKSORT

- Example 2: Quicksort
  - Different approach to split the list:
    - One element is selected as pivot.
    - Splitting (partition) between the elements that smaller than the pivot and elements that are larger than the pivot.
  - Pivot can be, for example, the first, last, or middle element.
  - The fastest known general-purpose in-memory sorting algorithm in the average case.
  - Note: if the pivot is always a very small or large value the partition step is not efficient (all values on one side of the pivot).
    - This can happen, for example, if the list is already sorted and we select the first or last element as pivot.

LUT University

DIVIDE AND CONQUER: QUICKSORT

6	2	10	5	1	7	11	4	8

2 | 5 | 1 | 4 | 6 | 10 | 7 | 11 | 8

1 | 2 | 5 | 4 | 6 | 7 | 8 | 10 | 11

1 | 2 | 4 | 5 | 6 | 7 | 8 | 10 | 11

BACKTRACKING

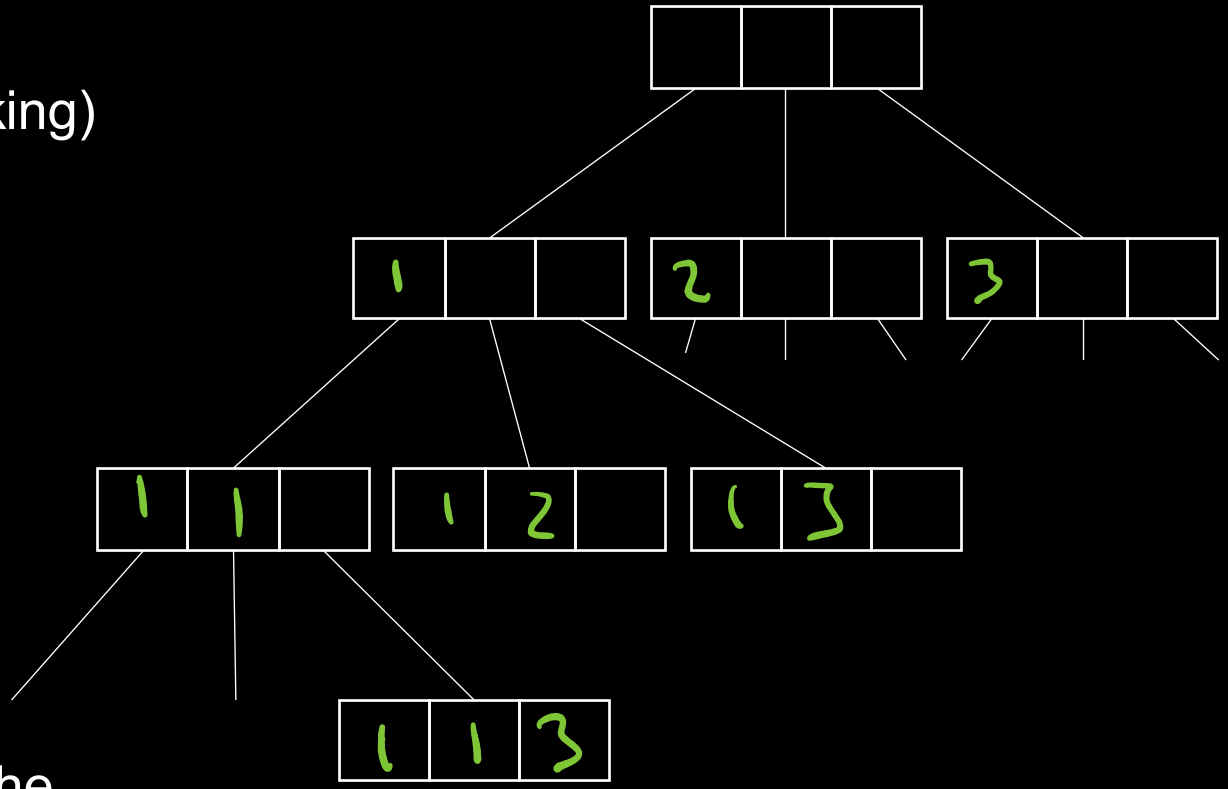
- Technique to systematically test all the possible solutions.
  - E.g., all the possible combinations: [1,1,1], [1,1,2], … , [1,1,m], [1,2,1], … , [m,m,m]
  - All permutations: [1,2,3,4], [1,2,4,3], [1,3,2,4], …
  - All subsets: [1], [2], [3], [1,2], …, [1,2,3] ~ [1, 0, 0], [0, 1, 0], [0, 0, 1], [1, 1, 0], …, [1, 1 ,1]
- For example: test all the combinations of *n* numbers, where each number is an integer between 1 and *m.*
  - Can be implemented using for loops.
  - What if *n* varies between inputs?

```
procedure search
for i = 1 to m
    for j = 1 to m
         for k = 1 to m
         …
            test_solution([i,j,k,…])
```

BACKTRACKING

- A better approach: use recursion (backtracking)

```
procedure search(k, numbers)
if k == n
    test_solution(numbers)
else
    for i = 1 to m
        numbers[k] = i
        search(k+1, numbers)
```



- Backtracking can be seen as traversing of the solution tree.

BACKTRACKING

{1, 2, 3, 4, 5}
{2, 4, 5}
0 1 0 1 1

- Backtracking all subsets:
  - We can present all subsets on *n* elements with a binary number with length *n*
    - *k*th digit is 1 if the *k*th element is selected to the subset and vice versa.
  - The same as backtracking all combinations of *n* numbers, where each number is 0 or 1.
  - 2^n different subsets (binary numbers)

```
procedure search(k, selected)
if k == n
    test_solution(selected)
else
    for i = 0 to 1
        selected[k] = i
        search(k+1, selected)
```

- Backtracking all permutations:
  - Numbers are not repeated.
  - We need to keep track on what numbers are already included.
  - This can be done with on additional list (`included`)

```
procedure search(k, numbers, included)
if k == n
    test_solution(numbers)
else
    for i = 1 to n
        if not included[i]
            included[i] = true
            numbers[k] = i
            search(k+1,numbers,included)
            included[i] = false
```

BRANCH AND BOUND

- A variation on backtracking that applies to optimization problems.
- Ideally, we would like to avoid traversing the whole tree.
- Proceeding deeper in the solution tree generally requires additional cost.
- If we remember the best-cost solution found so far, we can use it avoid exploring branches that cannot contain the optimal solution:
  - The best-cost solution found so far can be seen as the upper bound
    - the optimal solution cannot be worse than this.
  - The current cost of the solution being formed is the lower bound
    - the solutions found from the current branch of solution tree cannot be better than this.
  - If the lower bound is higher than the upper bound, the optimal solution cannot be in the current branch
    - we can immediately back up and take another branch.
  - We can further optimize this by calculating better estimate for the lower bound.

BRANCH AND BOUND

[ ]

[1] [2] [3] . . .

[1,1] [1,2]

.
.
.

[1,1,1,1,1,1]

C_{Best} (UB)

$LB = C_{cost} + C_{est}$

$LB = C_{cur}$

$LB > UB$ → this branch cannot contain the optimal solution

```
ub = inf

procedure search(k, numbers)
lb = cost(numbers)
if lb < ub
  if k == n
    ub = cost(numbers)
  else
    for i = 1 to m
      numbers[k] = i
      search(k+1, numbers)
```


LUT
University