Top-Down approach

- Patterned after the strategy employed by Napoleon
- Divide an instance of a problem recursively into two or more smaller instances until the solutions to the small instances are obtainable.
- Top-down approach used by recursive routines

Divide-and-Conquer

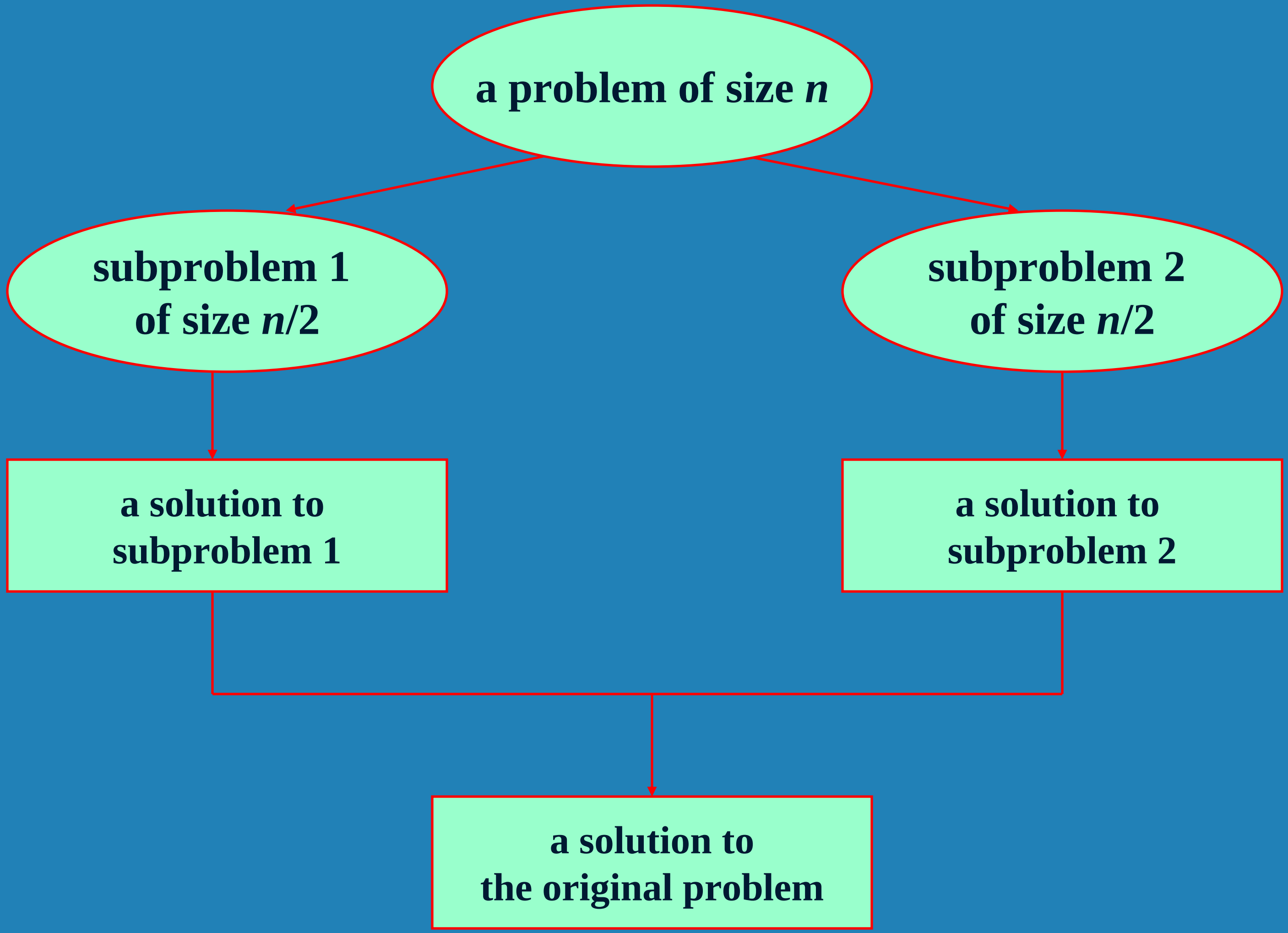
The most-well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances

2. Solve smaller instances recursively

3. Obtain solution to original (larger) instance by combining these solutions

Divide-and-Conquer Technique (cont.)



a problem of size *n*

subproblem 1
of size *n*/2

subproblem 2
of size *n*/2

a solution to
subproblem 1

a solution to
subproblem 2

a solution to
the original problem

Control Abstraction for Divide & Conquer

```
Algorithm DAndC(P)
{
    if Small(P) then return S(P);
    else
    {
    divide P into smaller instances $P_1, P_2, \ldots, P_k, K \geq 1$;
    Apply DAndC to each of these sub problems;
    return Combine(DAndC(p1), DAndC(p2),…
    DAndC(pk));
    }
}
```

Analysis

- Input size =n;
- g(n) = time taken to compute the answer directly for small inputs.
- f(n) = time for dividing and combing the solution to sub-problems.
- $T(n) = g(n)$ n is small

  $T(n_1)+T(n_2)+\ldots\ldots\ldots\ldots+T(n_k)+f(n)$ otherwise

General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

Master Theorem: If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Note: The same results hold with O instead of Θ.

Examples: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$

$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$

$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$

Recurrence

- $T(n) = T(1) \quad n=1;$
- $\qquad\qquad a\,T(n/b) + f(n) \quad n>1;$

Examples

- Consider a=2, b=2. T(1)=2 and f(n)=n.
- N is a power of 2, T(n) = T(1) n=1;

  T(n/2)+c; n>1;

3. a=2, b=2 and f(n) = cn
4. t(n) = 7 t(n/2) +18 n^2, n>=2 and a power of 2.
5. t(n) = 9t(n/3)+4 n^6, n>=3, and a power of 3.

Binary Search

If *x* equals the middle item, quit. Otherwise:

1. Divide **the array into two subarrays about half as large. If *x* is smaller than the middle item, choose the left subarray. If *x* is larger than the middle item, choose the right subarray.**
2. Conquer **(solve) the subarray by determining whether *x* is in that subarray. Unless the subarray is sufficiently small, use recursion to do this.**
3. Obtain **the solution to the array from the solution to the subarray.**

Example

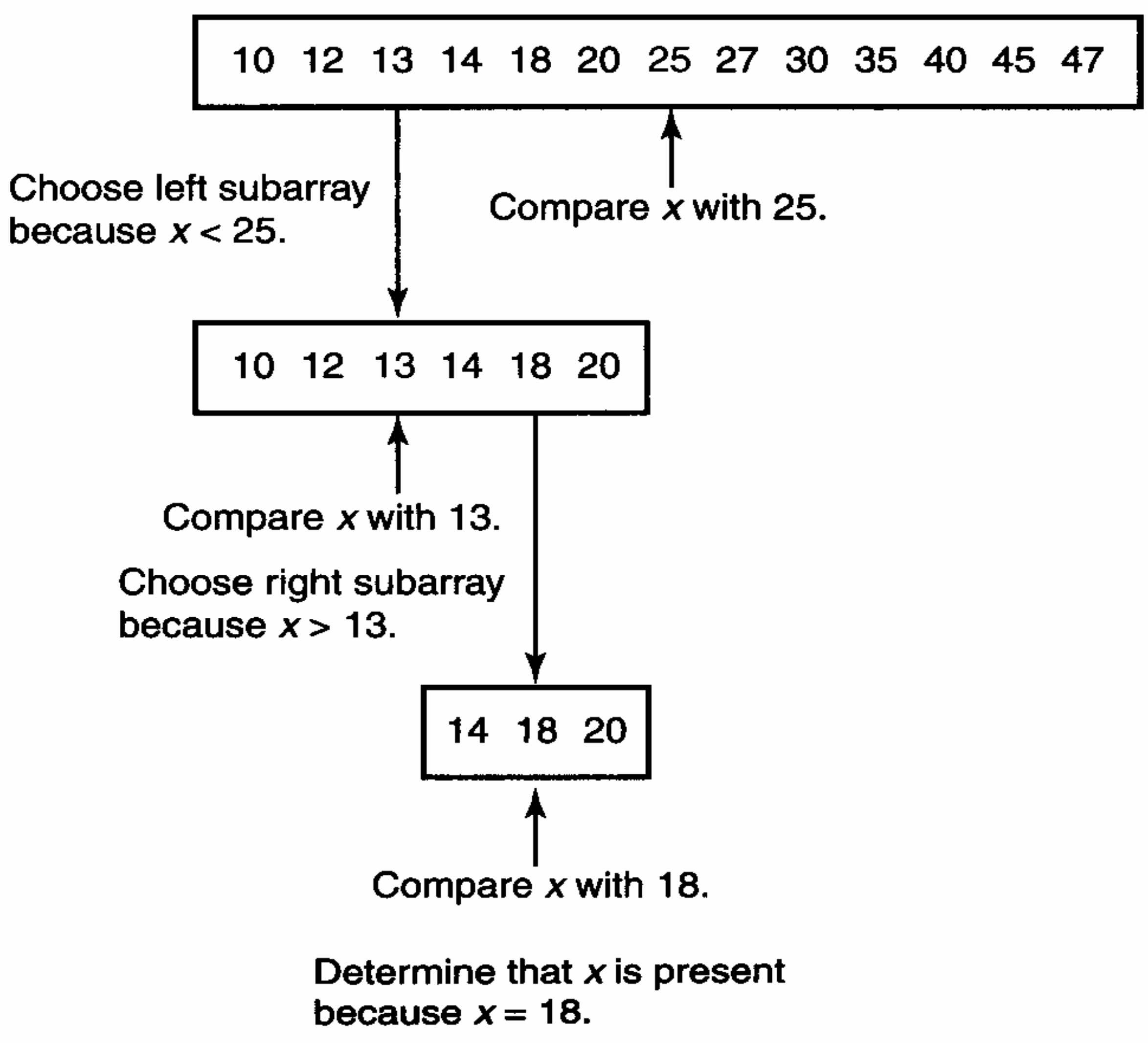
- Suppose *x* = 18 and we have the following array:

10 12 13 14 18 20 25 27 30 35 40 45 47

↑

middle item

Example



10 12 13 14 18 20 25 27 30 35 40 45 47

Choose left subarray because $x < 25$.

Compare x with 25.

10 12 13 14 18 20

Compare x with 13.

Choose right subarray because $x > 13$.

14 18 20

Compare x with 18.

Determine that x is present because $x = 18$.

Developing a recursive algorithm

- Develop a way to obtain the solution to an instance from the solution to one or more smaller instances
- Determine the terminal condition(s) that the smaller instance(s) is(are) approaching.
- Determine the solution in the case of the terminal condition(s).

Recursive algo

- // Input a[i:l] of elt in non-decreasing order 1<i<L, determine whether x is present, and if so, return j such that x = a[j]; else return 0;

```
Binsrch(a,i,l,x)
{
    if(l==i) then // Small(P)
    {
          if (x ==a[i]) then return i;
          else return 0;
    }
    else
    { // Reduce P into smaller sub problem
          mid = (i+l)/2;
          if (x == a[mid]) then return mid;
          else if( x < a[mid]) then
                    return Binsrch(a,I,mid-1,x);
               else
                    return Binsrch(a,mid+1,l,x);

    }
}
```

Recursive algorithm

- **Algorithm 2.1: Binary Search (Recursive)**
  - **Problem: Determine whether *x* is in the sorted array *S* of size *n*.**
  - **Inputs: positive integer *n*, sorted (nondecreasing order) array of keys *S* indexed from 1 to *n*, a key *x*.**
  - **Outputs: *location*, the location of *x* in *S* (0 if *x* is not in *S*).**

```
index location (index low, index high)
{
index mid;
if (low > high)
          return 0;
else {
          mid = [(low + high)/2];
          if (x == S[mid])
                         return mid
          else if (x < S[mid])
                         return location(low, mid - 1);
          else return location(mid + 1, high);
           }
}
```

Iterative binary search

- BinSearch(a,n,x)

```
 {
     low =1; high =n;
     while( low ≤ high) do
     {
       mid = floor((low+high)/2);
       if (x < a[mid]) then high = mid-1;
       else if (x > a[mid]) then low = mid +1;
                  else return mid;
     }
 return 0;
 }
```

Binary Search

Very efficient algorithm for searching in sorted array:

K

vs

$A[0]\ \ .\ \ .\ \ .\ \ A[m]\ \ .\ \ .\ \ .\ \ A[n-1]$

If $K = A[m]$, stop (successful search); otherwise, continue searching by the same method in $A[0..m-1]$ if $K < A[m]$ and in $A[m+1..n-1]$ if $K > A[m]$

```
$l \leftarrow 0$;   $r \leftarrow n-1$
while $l \le r$ do
    $m \leftarrow \lfloor (l+r)/2 \rfloor$
    if $K = A[m]$ return $m$
    else if $K < A[m]$ $r \leftarrow m-1$
    else $l \leftarrow m+1$
return -1
```

Example

- -15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151.

  Search x = 151;   Search x = -14;   Search x = 9;

solution

- Search x = 151;

low	high	mid
1	14	7
8	14	11
12	14	13
14	14	14 found

- Search x = -14;

low	high	mid
1	14	7
1	2	1
2	2	2
2	1	not found

Analysis of Binary Search

- Time efficiency
  - worst-case recurrence: $C_w(n) = 1 + C_w(\lfloor n/2 \rfloor)$, $C_w(1) = 1$
    solution: $C_w(n) = \lceil \log_2(n+1) \rceil$

    This is VERY fast: e.g., $C_w(10^6) = 20$

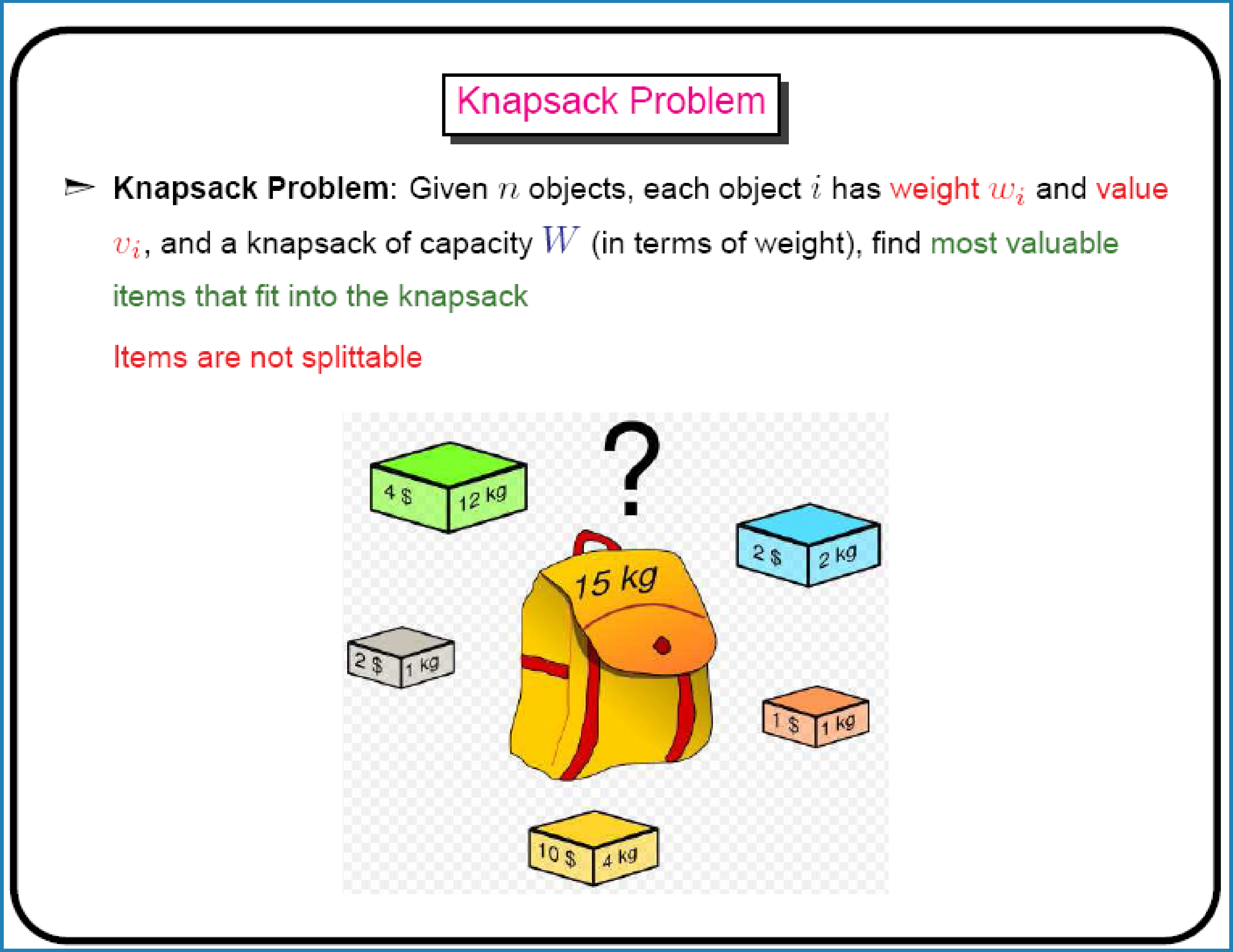
- Optimal for searching a sorted array

- Limitations: must be a sorted array (not linked list)

- Bad (degenerate) example of divide-and-conquer

- Has a continuous counterpart called *bisection method* for solving equations in one unknown $f(x) = 0$ (see Sec. 12.4)

Knapsack problem

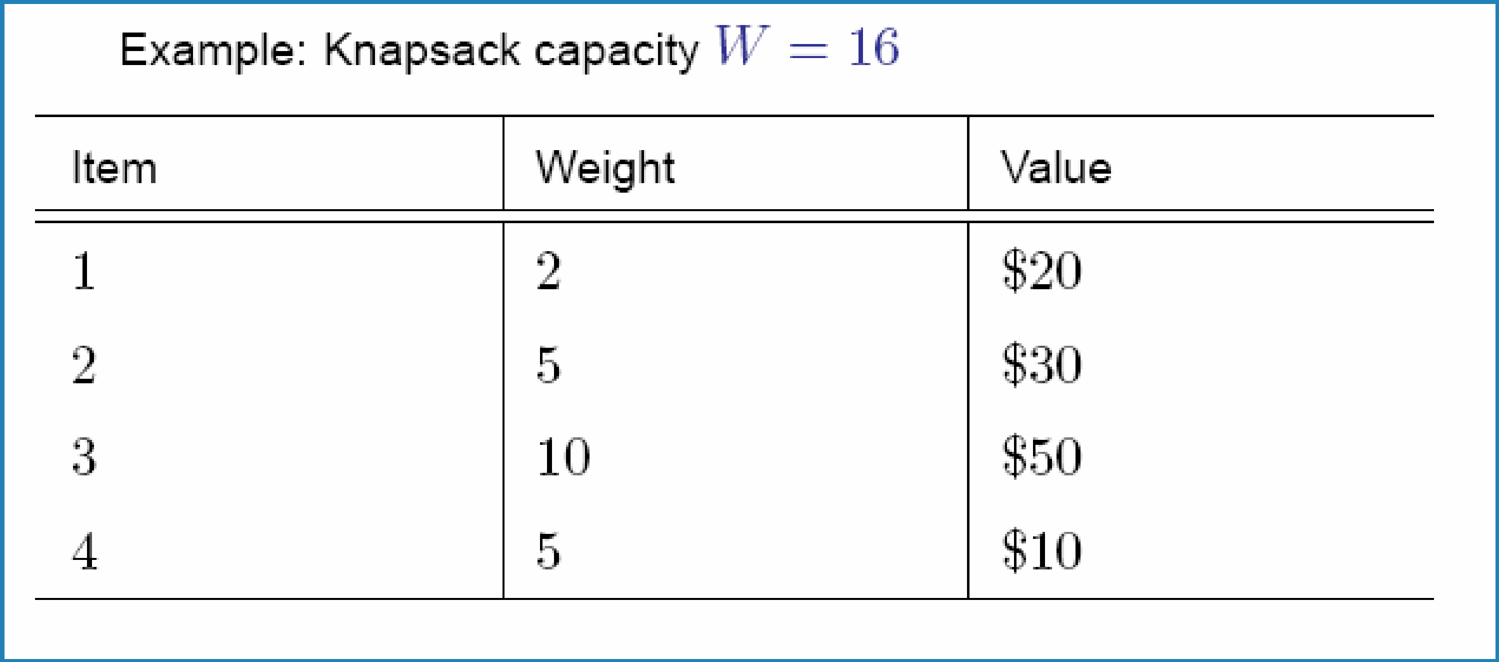


Knapsack Problem

- **Knapsack Problem**: Given n objects, each object i has weight w_i and value v_i, and a knapsack of capacity W (in terms of weight), find most valuable items that fit into the knapsack

  Items are not splittable

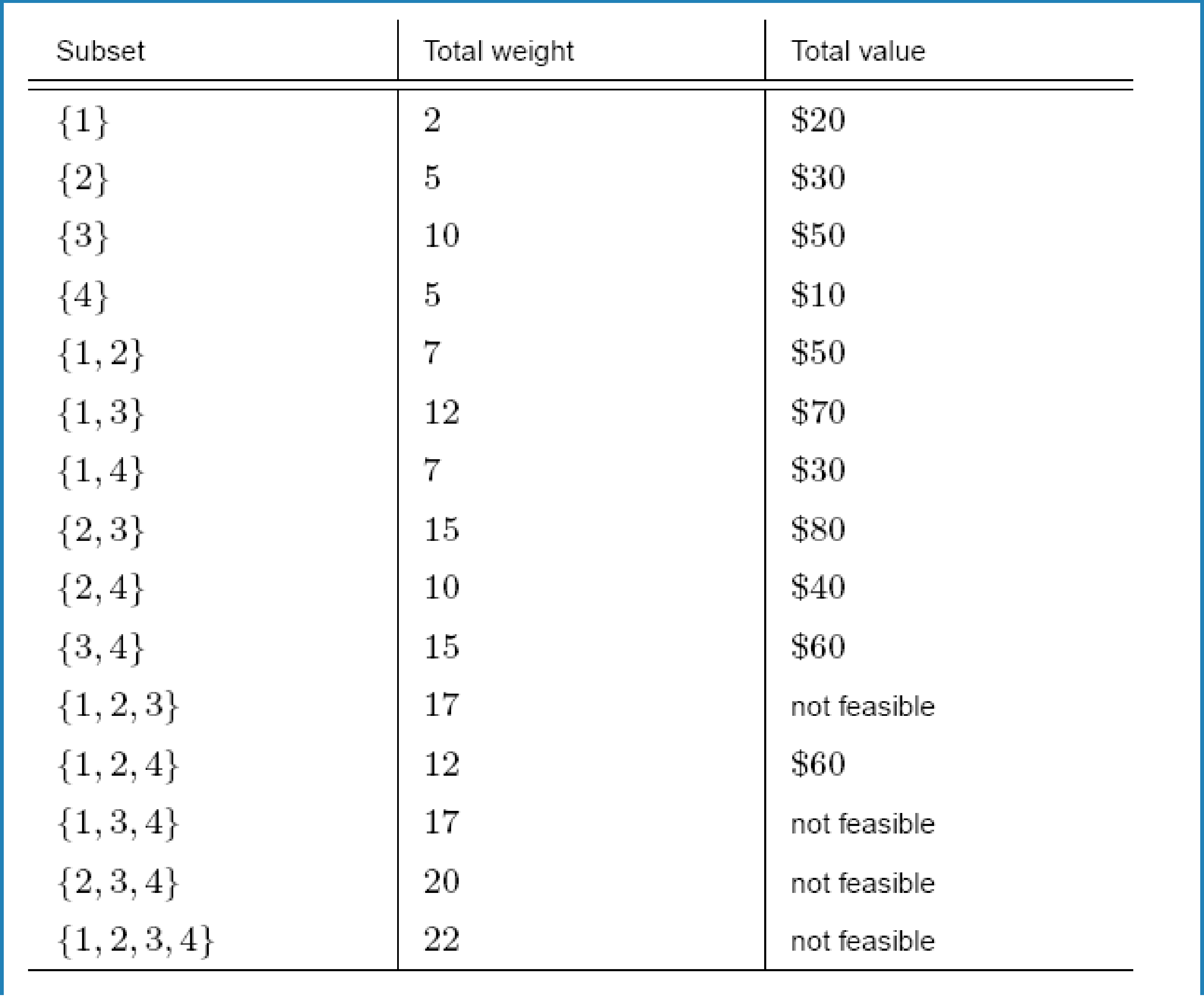
Example



Example: Knapsack capacity $W = 16$

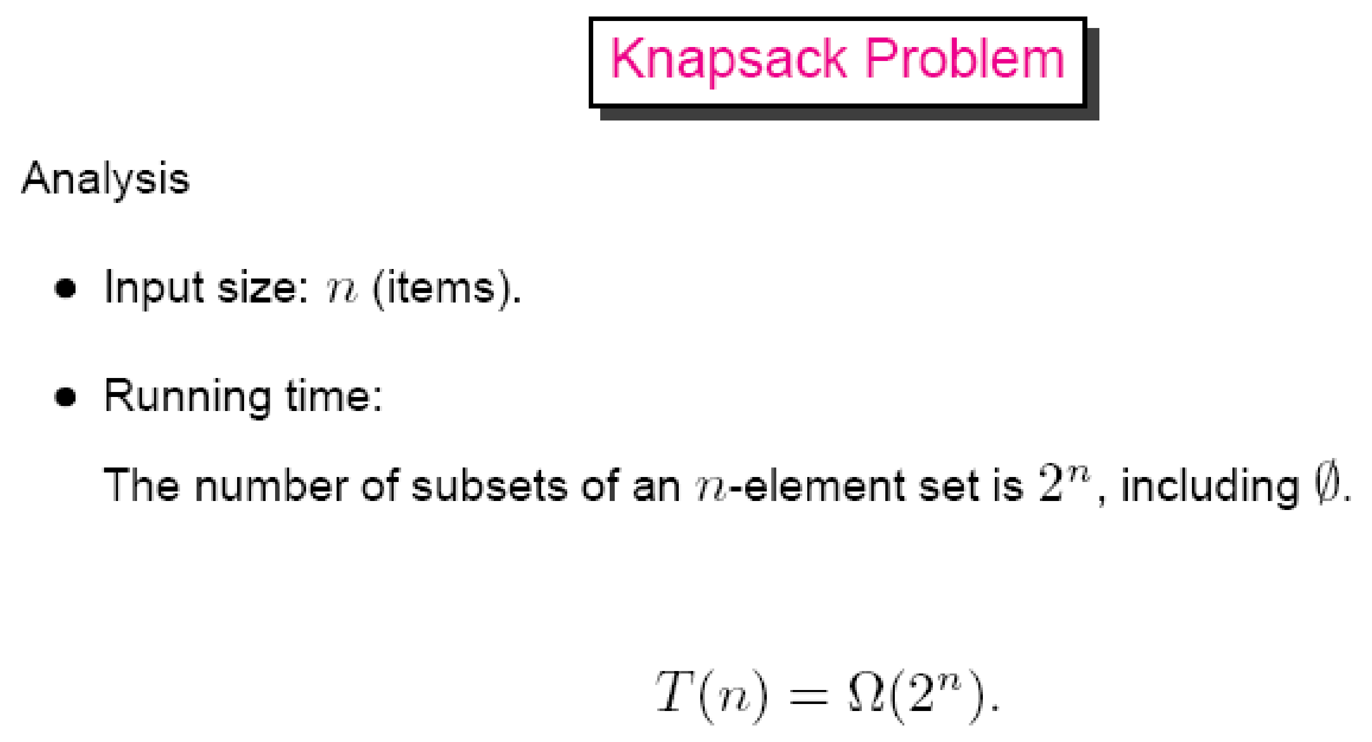
Item	Weight	Value
1	2	\$20
2	5	\$30
3	10	\$50
4	5	\$10

Example



Subset	Total weight	Total value
$\{1\}$	2	\$20
$\{2\}$	5	\$30
$\{3\}$	10	\$50
$\{4\}$	5	\$10
$\{1, 2\}$	7	\$50
$\{1, 3\}$	12	\$70
$\{1, 4\}$	7	\$30
$\{2, 3\}$	15	\$80
$\{2, 4\}$	10	\$40
$\{3, 4\}$	15	\$60
$\{1, 2, 3\}$	17	not feasible
$\{1, 2, 4\}$	12	\$60
$\{1, 3, 4\}$	17	not feasible
$\{2, 3, 4\}$	20	not feasible
$\{1, 2, 3, 4\}$	22	not feasible

Analysis



Knapsack Problem

Analysis

- Input size: n (items).
- Running time:

  The number of subsets of an n-element set is 2^n, including $\emptyset$.

$$T(n) = \Omega(2^n).$$

Divide-and-Conquer Examples

- Sorting: mergesort and quicksort
- Binary tree traversals
- Binary search (?)
- Multiplication of large integers
- Matrix multiplication: Strassen’s algorithm
- Closest-pair and convex-hull algorithms