

# AMA2222 Principles of Programming

Leung Man Kin, Adam  
Instructor

[adam.leung@polyu.edu.hk](mailto:adam.leung@polyu.edu.hk)

TU720

Chapter 8: Algorithm II (con't)  
exhaustive search, power set,  
permutation, heap algorithm

## Exhaustive search

Recall example 8.6, we are looking for the possible solution satisfying the time constraint and with maximum marks. One naïve idea is to generate all cases exhaustively and evaluate the total marks and total time for decision. This is called **exhaustive search**.

Question number	0	1	2	3
Marks	7	9	5	12
Time needed (unit: 10 min)	3	4	2	6

In this example, notice that the ordering of questions to answer does not matter. We can represent each case as a subset of the universal set of four questions.

Since there are 4 distinct elements, there are  $2^4 = 16$  cases:

$\emptyset, \{Q_0\}, \{Q_1\}, \{Q_2\}, \{Q_3\},$   
 $\{Q_0, Q_1\}, \{Q_0, Q_2\}, \{Q_0, Q_3\}, \{Q_1, Q_2\}, \{Q_1, Q_3\}, \{Q_2, Q_3\},$   
 $\{Q_0, Q_1, Q_2\}, \{Q_0, Q_1, Q_3\}, \{Q_0, Q_2, Q_3\}, \{Q_1, Q_2, Q_3\},$   
 $\{Q_0, Q_1, Q_2, Q_3\}$

In general, consider a set  $S$  of  $N$  distinct elements, then a **power set** of  $S$  noted by  $P(S)$  or  $2^S$  is a collection of all subsets of  $S$ . This collection contains  $2^N$  elements. The interpretation is that each distinct element either exist or not exist in a subset, giving  $2 \times \cdots \times 2 = 2^N$  combinations. This can also be interpreted as a binary number with  $N$  digits, thus representing numbers from 0 to  $2^N - 1$ .

Before we can generate a power set of the given array in example 8.6, we can first look at an easier example with string.

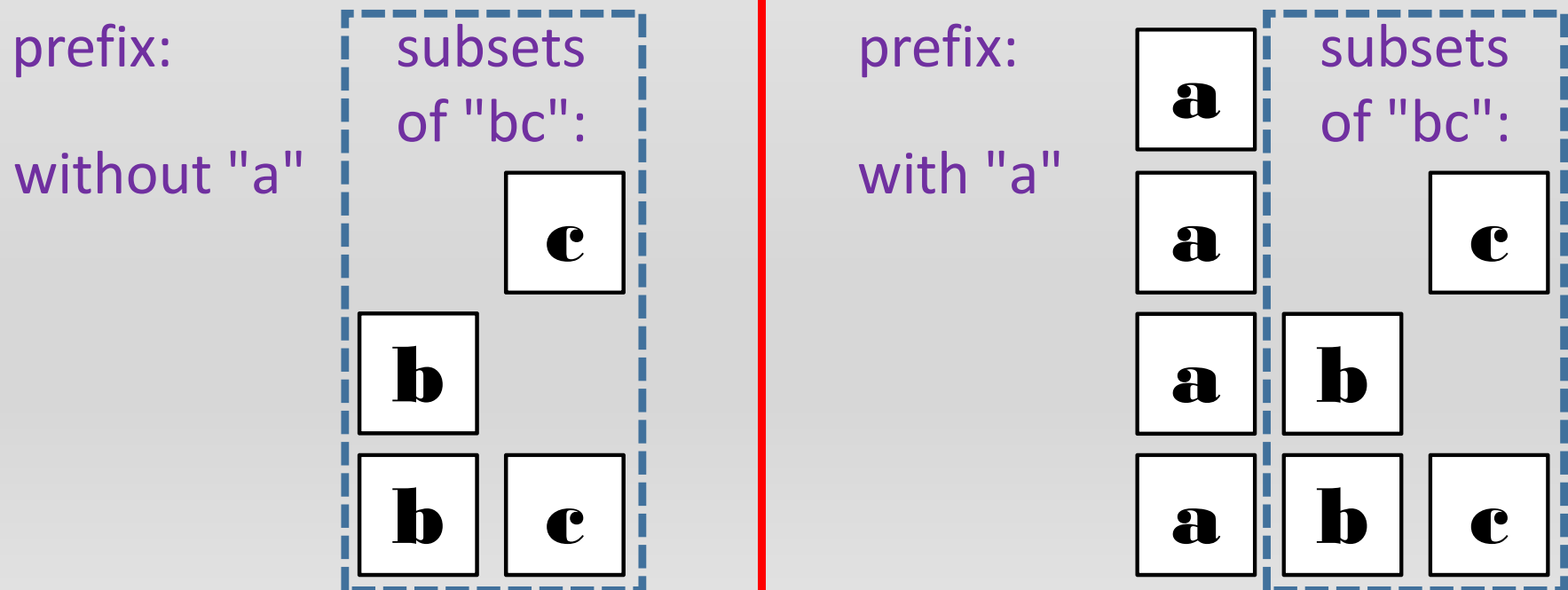
Example 8.7 A string can be regarded as a set of characters.

Write a function that generate all subsets of a string.

For example, if the string is "abc", we can generate:

"", "a", "b", "c", "ab", "bc", "ac", "abc"

This problem can be solved in a recursive approach.



binary form:

prefix:

without "a"

without "b"

subsets  
of "c":

000

001

with "b"

**b**

**b**

subsets  
of "c":

010

011

prefix:

with "a"

without "b"

subsets  
of "c":

100

101

with "b"

**a**

**a**

**b**

**b**

subsets  
of "c":

110

111

## Example program 8.7 Power set of string.

```
1 //eg 8.7 power set of string
2 #include <iostream>
3 #include <string>
4 using namespace std;
5 void powerset(string pre, string str){
6     int n = str.length();
7     if (n==0){
8         cout << pre << endl;
9         return;
10    }
11    powerset(pre, str.substr(1, n-1));
12    powerset(pre+str.substr(0,1), str.substr(1, n-1));
13 }
14 int main()
15 {
16     string s = "abcd";
17     powerset("", s);
18     return 0;
19 }
```

(base case) once there is nothing left in the string, print its prefix, terminate the function by return

The current element is str[0]. If we don't want this element, just keep the same prefix and input a new substring by skipping the beginning element str[0]

If we want the current element str[0], attach it to the prefix by concatenation. Then input a new substring by skipping the beginning element str[0]

When we call the function for the first time, the prefix should be a null string.

The same idea can also be applied to other data structures. You may try to implement the algorithm on Python list or numpy array. However, C++ array is not so convenient for concatenation and subarray. We will have to modify the powerset function so as to include the prefix and the remaining array and their sizes.

### Example 8.7 Power set of array

```
1 //eg 8.8 power set of array
2 #include <iostream>
3 using namespace std;
4 void print(int a[], int n)
5 {
6     for (int i=0; i<n; i++)
7         cout << a[i] << " ";
8     cout << endl;
9 }
```

Unlike a string, we cannot print an array directly. That's why we will first define a print function which prints an array with given size.

Input both the prefix and the remaining array as integer arrays to the function

```
10 void powerset(int p[], int pl, int a[], int al){
11     if (al==0){
12         print(p, pl);
13         return;
14     }
15     int b[al-1];
16     for (int i=0; i<al-1; i++)
17         b[i] = a[i+1];
18     int q[pl+1];
19     for (int i=0; i<pl; i++)
20         q[i] = p[i];
21     q[pl] = a[0];
22     powerset(p,pl,b,al-1);
23     powerset(q,pl+1,b,al-1);
24 }
25 int main()
26 {
27     int list[] = {2,3,5,7};
28     powerset({},0,list,4);
29     return 0;
30 }
```

If there is nothing left in the array, print the prefix

array b[] is a copy of array a[] after dropping the first element a[0]

array q[] is a copy of array p[] and then appended by the element a[0]

the case of without a[0]

the case of including a[0]

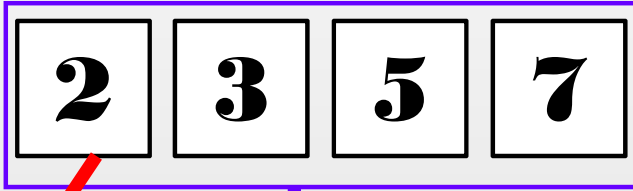
when we call the function for first time, use a null array as the prefix



`powerset({}, 0, list, 4)`

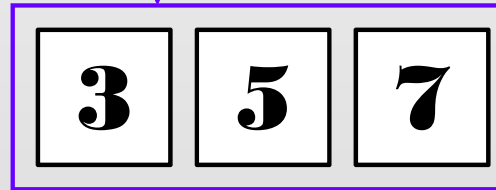
`p[]`

`a[]`



`q[]`

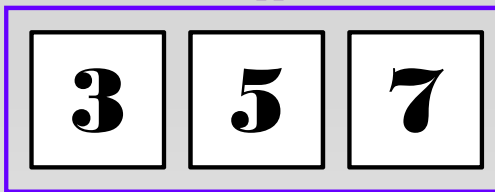
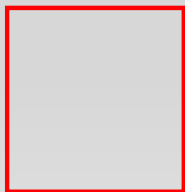
`b[]`



`powerset(p, p1, b, a1-1)`

`p[]`

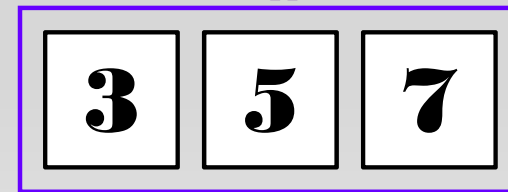
`b[]`



`powerset(q, p1+1, b, a1-1)`

`q[]`

`b[]`



Now we can go back to example 8.6. In order to find the solution with maximum total marks and within the time constraint, we can first generate the power set of the four question numbers {0,1,2,3}. Instead of printing it all cases, we replace the `print` function by a check function.

In the check function, the total marks and total time of the power set is evaluated. Only if the total time is within the constraint and the sum is greater than the current maximum value, the current maximum value will be updated. After generating and checking all the subsets, we can have the maximum marks.

For simplicity, we will store the two given arrays and the time allowed to be global variables which are accessible anywhere in the program. The current maximum has initial value of zero which is the total mark if no questions are chosen.

```

1 //eg 8.9 optimization by exhaustion
2 #include <iostream>
3 using namespace std;
4 int maxm = 0;
5 int marks[] = {7,9,5,12};
6 int time[] = {3,4,2,6};
7 int t = 6;
8 void check(int a[], int n)
9 {
10     int sm=0, st=0;
11     for (int i=0; i<n; i++){
12         sm += marks[a[i]];
13         st += time[a[i]];
14     }
15     if (sm>maxm && st<=t)
16         maxm = sm;
17 }

```

add up total mark and time of a given subarray from {0,1,2,3}

update the current maximum if it satisfies the time constraint and result in a higher total marks

```

18 void powerset(int p[], int pl, int a[], int al){
19     if (al==0){
20         print(p, pl);
21         return;
22     }
23     int b[al-1];
24     for (int i=0; i<al-1; i++)
25         b[i] = a[i+1];
26     int q[pl+1];
27     for (int i=0; i<pl; i++)
28         q[i] = p[i];
29     q[pl] = a[0];
30     powerset(p,pl,b,al-1);
31     powerset(q,pl+1,b,al-1);
32 }
33 int main()
34 {
35     int list[] = {0,1,2,3};
36     powerset({},0,list,4);
37     cout << maxm;
38     return 0;
39 }

```

use the question number (index)  
as the array

the maximum mark is stored as a  
global variable

## Permutation

The exhaustive search method in the previous question requires the generation of the power set of a given set, with a total of  $2^n$  subsets of the original set. Each subset refers to a case where each element exists or not. The ordering of those elements does not matter.

In some problems however, each case refers to an rearrangement of all the given elements denoted by a **permutation**. Ordering does matter in such problems. If there are  $N$  distinct elements, then there are  $N!$  permutations.

Now our question is: how to generate all permutations of a given set of distinct elements?

One efficient solution is the **Heap's algorithm**, published by Heap in a journal <https://doi.org/10.1093%2Fcomjnl%2F6.3.293> in 1963.

Heap's algorithm is based on a recursive approach. To generate a permutation of  $N$  elements, we might first put one element at the end, then generate a permutation of  $N-1$  elements in front. Until there is only one element left, there is only one possible permutation.

However, how to rotate the elements and avoid repetition of the permutations is a problem. Heap suggests that this can be done by swapping two elements at each step depending on the **parity** (odd or even) of the subarray size.

Example 8.10 Generating permutations

```

1 //eg 8.10 generating permutations
2 #include <iostream>
3 using namespace std;
4 void print(int a[], int n)
5 {
6     for (int i=0; i<n; i++)
7         cout << a[i] << " ";
8     cout << endl;
9 }
10 void permutation(int a[], int s, int n)
11 {
12     if (s==1) {
13         print(a,n);
14         return;
15     }
16     for (int i=0; i<s; i++){
17         permutation(a, s-1, n);
18         if (s%2==1)
19             swap(a[0],a[s-1]);
20         else
21             swap(a[i],a[s-1]);
22     }
23 }

```

if the subarray has size of 1, there is only one permutation, print the whole array and terminate the function

at each iteration of the for loop, we fix one element at the end of the array and generate all permutations of the s-1 elements in front.

after finding all permutations with a common element  $a[s-1]$  at the end, we will swap it with another element depending on the parity of size  $s$ .

Now we will try arrays with different length in our main program to examine the result.

### Sample 1

```
24 int main()  
25 {  
26     int list[] = {4,5,6};  
27     permutation(list,3,3);  
28     return 0;  
29 }
```

### Result 1

4	5	6
5	4	6
6	4	5
4	6	5
5	6	4
6	5	4

### Result 2

2	3	5	7
3	2	5	7
5	2	3	7
2	5	3	7
3	5	2	7
5	3	2	7
7	3	5	2
3	7	5	2
5	7	3	2
7	5	3	2
3	5	7	2
5	3	7	2
7	2	5	3
2	7	5	3
5	7	2	3
7	5	2	3
2	5	7	3
5	2	7	3
7	2	3	5
2	7	3	5
3	7	2	5
7	3	2	5
2	3	7	5
3	2	7	5

### Sample 2

```
24 int main()  
25 {  
26     int list[] = {2,3,5,7};  
27     permutation(list,4,4);  
28     return 0;  
29 }
```

To explain, we will use the first result with original array {4,5,6}.



$i = 0, s = 3:$

permutation of {4,5}

4	5	6
5	4	6

since  $s$  is odd,  
swap  $a[i]$  and  $a[s-1]$   
(i.e. swap  $a[0]$  and  $a[2]$ )

6	4	5
---	---	---

$i = 1, s = 3:$

permutation of {6,4}

6	4	5
4	6	5

since  $s$  is odd,  
swap  $a[i]$  and  $a[s-1]$   
(i.e. swap  $a[0]$  and  $a[2]$ )

5	6	4
---	---	---

$i = 2, s = 3:$

permutation of {5,6}

5	6	4
6	5	4

since  $s$  is odd,  
swap  $a[i]$  and  $a[s-1]$   
(i.e. swap  $a[0]$  and  $a[2]$ )

4	5	6
---	---	---