The following lecture is divided into three parts, in the first one we will consider the Subset Sum problem and see how to solve it using variation over the well-known meet-in-the-middle technique. The second part will consist of interesting application of divide-and-conquer in designing moderately exponential algorithms. In third part we will get familiar with the memoization technique and see how to apply it to the Perfect Matching Count problem.

1 Subset Sum problem

1.1 Introduction

Subset Sum

**Input:** Sequence of \( n \) integers \( a_1, a_2, \ldots, a_n \) and an integer \( S \)

**Question:** Is there any subset of indices \( \{i_1, \ldots, i_k\} \) such that \( a_{i_1} + a_{i_2} + \ldots + a_{i_k} = S \)

It’s well known that the above problem is NP-complete, there is trivial solution with computational complexity \( O^*(2^n) \) and space complexity \( O^*(1) \). There’s also classic dynamic pseudopolynomial algorithm which solves that problem in time \( O(nS) \) and space \( O(S) \). Those two are going to be our base for further improvements.

Our goal at this point is to find solution more effective than \( 2^n \), independent on \( S \), having in mind space complexity (this is relevant when \( S \) is large).

1.2 \( k \)-Sum problem

Let us reformulate the above problem and generalize it a little in order to have a better sight. Consider the following problem:
K-SUM problem

**Input:** $k$ sequences of integers $(a_j^i)^n_{i=1}$ for $1 \leq j \leq k$ and an integer $S$.

**Question:** Are there indices $i_1, i_2, \cdots, i_k$, s. t.

$$\sum_{j=1}^{k} a_{i_j}^j = S$$

**Remark 1.** The SUBSET SUM problem can be reduced to the K-SUM problem for any $k$, where $n'$ is $2^{n/k}$.

Indeed, given an instance of the SUBSET SUM problem, say: $(a_1, a_2, \cdots, a_n; S)$ one might take an instance of the K-SUM problem as $(a_1^i)^n_{i=1}$ being sequence of sums of all subsets of $a_1, a_2, \cdots, a_n$, $(a_2^i)^n_{i=1}$ are all sums of subsets of second block of length $n'$ and so on. Now each choice of indices $i_j$ in K-SUM problem is the same as determining what is the intersection of an unknown subset for the SUBSET SUM problem with the blocks $k^\frac{n}{k}, \ldots, (k + 1)^\frac{n}{k} - 1$.

**Remark 2.** The K-SUM problem can be solved in time $O(n^k)$ (trivially).

Note that the remark above alone is useless for our purpose: after reducing SUBSET SUM to K-SUM as above and applying this trivial solution we acquire solution with complexity $O^*(2^n)$ to SUBSET SUM.

1.3 2-Sum problem

In order to get better algorithm for the subset sum problem, we focus on trying to solve the K-SUM problem for some particular $k$ better than in $O(n^k)$. Lets start with 2-Sum.

For fixed $i_1$ we do know we have chosen $a_{i_1}^1$, what we want to check is if there exists $S - a_{i_1}^1$ in the second sequence. It suffices then to sort the second sequence in advance, to make those queries easy.

**Corollary 3.** The 2-SUM problem can be solved in time $O(n \log n)$ and linear space.

**Corollary 4.** The SUBSET SUM problem can be solved in time $O^*(2^{\frac{n}{2}})$ and space $O^*(2^{\frac{n}{2}})$.

1.4 4-Sum

Our next goal is to improve space complexity, preserving time complexity.
Let's make precise algorithm for $k = 2$: Sort first sequence in the increasing, and second one in the decreasing order. Now one can solve the $k$-SUM problem linearly, moving two pointers simultaneously, as below:

```plaintext
while $i \leq n \& j \leq n$ do
    $z = a^1_i + a^2_j$
    if $z = S$ then
        END
    end if
    if $z < S$ then
        Increment $i$
    end if
    if $z > S$ then
        Increment $j$
    end if
end while
```

Let's now try to solve the 4-SUM problem in $O(n^2)$ time and linear space. It would give us SUBSET SUM solution in time $O^*(2^{n/2})$ and space $O^*(2^{n/2})$ via reduction from above.

We want to modify the algorithm from above: it would be sufficient to be able to efficiently generate all pairs of indices $i_1, i_2$ in the order of increasing value $a^1_{i_1} + a^2_{i_2}$ — we would do the same with $i_3, i_4$, generating them in decreasing order and apply the algorithm from above.

Indeed, one can generate all such pairs in $O(n)$ space and $O(n^2 \log n)$ time. Let's sort both sequences $a^1$ and $a^2$ in the increasing order, and maintain a heap with pairs of active indices: at any point in the heap for every $i_1 \in \{1 \cdots n\}$ there is exactly one pair of active indices $(i_1, i_2)$: $i_2$ being the lowest s.t. that pair have not yet been generated. Heap is ordered by $a^1_{i_1} + a^2_{i_2}$, now at each point there's exactly $n$ elements in heap. When we're asked for the next pair, one just takes the first active pair from the heap (i.e. the one with smallest value $a^1_{i_1} + a^2_{i_2}$ and put the pair $(i_1, i_2 + 1)$ into the heap as active.

That leads us to the following corollaries, proved by Schroeppel and Shamir [3].

**Corollary 5.** The 4-SUM problem can be solved in $O(n^2 \log n)$ time and linear space.

**Corollary 6.** The SUBSET SUM problem can be solved in $O^*(2^{n/2})$ time and $O^*(2^{n/2})$ space.

Those are currently best-known bounds for this problem. There are
two important open question around this topic: is there any algorithm to the Subset Sum problem which solves it in polynomial space, and time complexity $O^*(2^{\varepsilon n})$? Is it possible to achieve better time complexity than $O^*(2^{\frac{n}{2}})$?

2 Divide and conquer

2.1 Introduction; Traveling Salesman Problem

Lets take a look into the very well known divide-and-conquer technique and its application in designing moderate exponential algorithm for one of the most famous NP-hard problems.

<table>
<thead>
<tr>
<th>Traveling Salesman Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Directed, weighted graph $G$</td>
</tr>
<tr>
<td><strong>Question:</strong> Cheapest Hamilton cycle in $G$</td>
</tr>
</tbody>
</table>

The dynamic programming solution presented in one of the previous lectures for the Hamilton Cycle problem is still valid: for every state in dynamic programming, one keeps information about cheapest partial solution. That leads to a solution with time complexity $O^*(2^n)$ and space complexity $O^*(2^n)$.

The solution based on the inclusion-exclusion principle can’t be extended to this generalized case.

2.2 Solution

Our goal now is to get an algorithm of time complexity $O^*(c^n)$ and polynomial space complexity.

It’s obvious that our problem is polynomially equivalent to the problem of finding the cheapest Hamilton path between $s$ and $t$ for fixed $s, t$. We will consider this problem from now on.

It would suffice to guess properly a vertex $v$ which is in the middle of the cheapest path, then for every other vertex of the graph guess on which side of this vertex it is on this path. Now we could solve two smaller problems independently.

More precisely: for each vertex $v \in V$, and for each division $V \setminus \{s, t, v\} = V_1 \cup V_2$ into two sets s.t. $V_1 \cap V_2 = \emptyset$ and $|V_1| = \lfloor \frac{n-3}{2} \rfloor$, solve recursively independent instances:

- $(G[V_1 \cup \{s, v\}], s, v)$
- $(G[V_2 \cup \{v, t\}], v, t)$
This gives us recursive equation of time complexity:

\[ T(n) = n \cdot 2^n \cdot 2 \cdot T(n/2) \]

With rather easy solution:

\[ T(n) = O^*(n^{\log_2 n} \cdot 4^n) \]

**Corollary 7.** There is an algorithm for the Traveling Salesman Problem of time complexity \( O^*((4 + \varepsilon)^n) \) and space complexity \( O^*(1) \).

## 3 Memoization

### 3.1 Introduction; Perfect Matching Counting problem

At this part of lecture we will discuss memoization, which is effectively another way of looking at the dynamic programming method. With a careful examination, one might recognize that not all states of dynamic programming are relevant to the final solution, so with a top-down implementation and more precise analysis it is possible to acquire a better time complexity — sometimes by cost of a little worse space complexity.

We will see an example of how one can use memoization to solve the **Perfect Matching Counting** problem, defined as follows:

<table>
<thead>
<tr>
<th><strong>Perfect Matching Counting</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> An undirected graph ( G ) with ( n ) vertices</td>
</tr>
<tr>
<td><strong>Question:</strong> Calculate how many there are different 1-regular subgraphs of ( G ) on ( n ) vertices (perfect matchings of ( G ))</td>
</tr>
</tbody>
</table>

### 3.2 Solution

There is an obvious dynamic solution for the problem posted above: for every subset \( X \) of set \( V \) one wants to calculate \( T[X] \) as the number of perfect matchings in \( G[X] \). Now a solution for our problem is just \( T[V] \), and there is a recursive formula for calculating \( T[X] \): for a set \( X \) we fix a vertex \( v \in X \) with the smallest index. Now:

\[ T[X] = \sum_{uv \in E} t[X \setminus \{u, v\}] \]

There is an obvious upper bound of \( O^*(2^n) \) on the complexity of the above algorithm, but maybe we could do better than this, and examine which fields of \( T[X] \) are relevant. The following lemma will be helpful:
Lemma 8. If recursion touches a set $X$ of size $n-2i$, $X \cap \{v_1, v_2, \cdots, v_i\} = \emptyset$

Proof. Trivial, as at every point we remove two vertices, one of them having smallest index among all remaining. \hfill $\square$

Now it follows directly from the lemma above, that number of states we are trying to estimate does not exceed:

$$\sum_{i=0}^{n/2} \binom{n-i}{i}$$

One can easily (but somehow strikingly) bound this sum from above:

$$\sum_{i=0}^{n/2} \binom{n-i}{i} \leq \sum_{i=0}^{\infty} \binom{n-i}{i} = F_{n+1}$$

while the latter equality follows from simple induction:

$$\sum_{i=0}^{\infty} \binom{n-i}{i} = \sum_{i=0}^{\infty} \binom{n-i-1}{i} + \sum_{i=0}^{\infty} \binom{n-i-1}{i-1}$$

$$= F_n + \sum_{i=0}^{\infty} \binom{n-2-i}{i}$$

$$= F_n + F_{n-1} = F_{n+1}$$

This leads to the following algorithm by Koivisto [2].

Corollary 9. There exist an algorithm for the Perfect Matching Counting problem which runs in $O^*(\left(1+\sqrt{5}\right)^n)$ time and space complexity.

Informal, but useful hint: During designing this kind of algorithms often suffices to see that number of subsets of size roughly $\frac{n}{2}$ which are visited during recursion is strictly less (asymptotically) than $2^n$.

4 Perfect Matching Counting once again

In this section there will be presented a sketch of a solution in time $O^*(2^{n/2})$ for the Perfect Matching Counting problem. Space could be limited to polynomial. The algorithm below is a simplification of the algorithm of Björklund [1].

Lets take a graph $G$, mark its edges as black, then add $n/2$ red edges: $v_1v_2, v_3v_4, \cdots$; that leads us to a multigraph $G'$ with black and red edges.
Remark 10. The number of perfect matchings in $G$ equals the number of cycle covers of $G'$, where every cycle is alternating (even edges are black, odds are red).

We want to calculate this number faster (in time $O^*(2^{n/2})$). Let us use the inclusion-exclusion principle, as in one of the previous lectures. Define $\Omega$ as the set of all tuples of closed alternating walks of sum of length $n$. $A_i \subset \Omega$ are those which contain the $i$-th red edge. What we want to calculate is $|\bigcap A_i|$, and it is fairly simple by inclusion-exclusion principle. Yet, it is still not a solution, as same matching is taken into account more than once. If we take $\Omega_d$ being $d$-tuples as above, our number of perfect matching is roughly:

$$\sum_d \left| \bigcap_{1 \leq i \leq \frac{n}{2}} A_i^d \right| /d!$$

(1)

One minor issue one has to work out are automorphisms of any single cycle, but it is possible to get through it.

References

